Doctoral Dissertation

Doctoral Program in Computer and Control Engineering (30th cycle)

Visual Analysis Algorithms for Embedded Systems

By

Syed Tahir Hussain Rizvi

******

Supervisor(s):

Prof. Gianpiero Cabodi

Politecnico di Torino

2018

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Syed Tahir Hussain Rizvi
2018

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving family for their constant support and encouragement.

Acknowledgements

I would like to thank my thesis supervisor, Prof. Gianpiero Cabodi, for allowing me to work under his supervision and constantly supporting me during my Ph.D. studies. I would also like to thank my research group members at the Joint Open Lab (JOL) Visible, especially Gianluca Francini and Alfonso Di Salvo, for all the help they provided. I am grateful to TIM, former Telecom Italia for financing my Ph.D. studies. I must also thank Denis Patti for his help regarding the realization of proposed framework on Android devices.

Abstract

The main contribution of this thesis is the design and development of an optimized framework to realize the deep neural classifiers on the embedded platforms. Deep convolutional networks exhibit unmatched performance in image classification. However, these deep classifiers demand huge computational power and memory storage. That is an issue on embedded devices due to limited onboard resources. The computational demand of neural networks mainly stems from the convolutional layers. A significant improvement in performance can be obtained by reducing the computational complexity of these convolutional layers, making them realizable on embedded platforms.

In this thesis, we proposed a CUDA (Compute Unified Device Architecture)-based accelerated scheme to realize the deep architectures on the embedded platforms by exploiting the already trained networks. All required functions and layers to replicate the trained neural networks were implemented and accelerated using concurrent resources of embedded GPU. Performance of our CUDA-based proposed scheme was significantly improved by performing convolutions in the transform domain. This matrix multiplication based convolution was also compared with the traditional approach to analyze the improvement in inference performance.

The second part of this thesis focused on the optimization of the proposed framework. The flow of our CUDA-based framework was optimized using unified memory scheme and hardware-dependent utilization of computational resources. The proposed flow was evaluated over three different image classification networks on Jetson TX1 embedded board and Nvidia Shield K1 tablet. The performance of proposed GPU-only flow was compared with its sequential and heterogeneous versions. The results showed that the proposed scheme brought the higher performance and enabled the real-time image classification on the embedded platforms with lesser storage requirements. These results motivated us towards the realization of useful real-time classification and recognition problems on the embedded platforms.

Finally, we utilized the proposed framework to realize the neural network-based automatic license plate recognition (ALPR) system on a mobile platform. This highly-precise and computationally demanding system was deployed by simplifying the flow of trained deep architecture developed for powerful desktop and server environments. A comparative analysis of computational complexity, recognition accuracy and inference performance was performed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Visual analysis is an ever-growing field with real-time applications reaching out into daily life. Convolutional neural networks (CNN) have recently enjoyed a great success in a range of visual applications. Deep convolutional neural networks exhibit unmatched recognition and classification accuracy in visual analysis applications [1–5]. This accuracy mainly comes from the deeper architecture [6, 7]. As the network grows deeper, the computational complexity of both training and testing stages of neural networks increase exponentially.

The requirements of both training and testing stages are slightly different. Training of a neural network is more intensive due to a huge number of input samples, parallel classifications and numerous iterations required for learning large datasets, it can only be performed offline on a powerful desktop workstation or using a cluster of computational accelerators. However, a trained network usually needs to classify only single images in its testing phase. This actual image classification may has to be performed online on an embedded device for the real-time visual analysis [8]. With the ongoing traction of embedded computing, deployment of deep classifiers on mobile devices is getting more and more attention. There are numerous practical real-time applications where the hand-held devices can be more useful due to their compact size and integrated resources (like Graphics Processing Unit (GPU), Camera, etc) [9, 10].

Due to the increasing demands of parallel computing in various fields, GPU has evolved from a graphics accelerator into a programmable computer. GPU is an efficient accelerator with the good price per performance ratio. There are different programming environments available (e.g. OpenCL and CUDA) to perform and assign the general purpose tasks to a GPU. The computational power of the embedded GPUs can be exploited to deploy complex general-purpose applications on the hand-held devices [11, 12].

1.1 Problem Statement

Many frameworks like Torch and Caffe are currently available for implementing the neural classifiers [13–16]. These frameworks exploit the computational power of modern GPUs for faster training and deployment of deep neural networks. However, most of such libraries are designed for desktop and server environments; therefore, do not consider the unique peculiarities of embedded platforms and cannot be directly used on a mobile device due to software/hardware constraints and dependences.

Some of these constraints are limited computational resources and battery life of the mobile devices. Furthermore, the embedded platforms have limited storage capacity; so the size of trained models, computing framework and required computational packages must also be considered as limiting factor for the realization of neural classifiers on the embedded devices. Due to these massive computational and storage requirements, the realization of deep architectures on the embedded platforms is still a challenging problem.

1.2 Contribution

The goal of the thesis is to design and develop an optimized framework that can be used to deploy the deep classifiers on the embedded platforms. The CUDA computing framework is used for the realization of identical neural architectures on embedded devices to exploit the already trained networks. The CUDA computing language is selected to design the proposed scheme because it is widely used by the deep learning community and more mature than the OpenCL language in terms of performance. The main contributions of our research activities are:

- Development of a CUDA-based framework that supports nearly all layer types of deep neural architectures and suitable for deploying complex state-of-the-art deep classifiers. Intermediate frameworks and memory consuming computational packages are avoided using the proposed CUDA-only scheme to resolve the problem of software dependencies and reduce the storage requirements. The proposed framework can be easily integrated into an Android application for actual classification and provides compatibility for models trained with other desktop or server frameworks.

- Optimization of proposed framework to improve the inference performance, storage requirements and energy efficiency. Only CUDA-based functions and libraries are used to optimize the framework. An efficient memory transfer scheme is employed to optimize the flow of implemented functions. Additionally, the proposed framework is further accelerated and optimized using the hardware-dependent selection of GPU resources.

- Utilization of proposed optimized framework to realize the neural network-based real-time classification problems on embedded GPUs. A comparative analysis of arithmetical complexity and inference performance is performed by simplifying the flow of trained deep architecture developed for computationally powerful desktop and server environments.

1.3 Organization of the thesis

A brief description of each chapter is presented here.

Chapter 2 presents a short overview of the recent evolution of deep convolutional neural networks and their contributions to image classification.

Chapter 3 gives a description of OpenCL and CUDA Programming Languages for GPU based computing. Different concepts like the arrangement of computational resources and memory model of CUDA language are explained.

Chapter 4 discusses the architecture of deep neural classifiers and our proposed scheme to realize them on embedded platforms. This is done by replicating all required functions to construct neural architecture in CUDA computing framework and exploiting the computational power of the embedded GPUs. This chapter also discusses the relevant literature concerning deep neural networks on mobile devices.

Chapter 5 describes the optimization techniques that we used to improve the performance of our proposed framework, making deep classifiers applicable to embedded applications.

Chapter 6 presents a practical application of the proposed framework where a neural network-based automatic license plate recognition (ALPR) system is realized on a mobile platform.

This research work is finally concluded in **Chapter 7** along with a brief description of the future directions to further improve the performance of proposed framework.

1.4 Publications

This section provides a list of publications which are the result of our research activities carried out in the Ph.D. program.

Journals:

1. Rizvi, S.T.H.; Cabodi, G.; Patti, D.; Francini, G. GPGPU Accelerated Deep Object Classification on a Heterogeneous Mobile Platform, *Electronics*, 2016, 5, 88.

2. Rizvi, S.T.H.; Cabodi, G.; Francini, G. Optimized Deep Neural Networks for Real-Time Object Classification on Embedded GPUs, *Applied Sciences*, 2017, 7, 826.

3. Rizvi, S.T.H.; Patti, D.; Björklund, T.; Cabodi, G.; Francini, G. Deep Classifiers-Based License Plate Detection, Localization and Recognition on GPU-Powered Mobile Platform, *Future Internet*, 2017, 9, 66.

4. Rizvi, S.T.H.; Cabodi, G.; Patti, D.; Gulzar, M.M. A General-Purpose Graphics Processing Unit (GPGPU)-Accelerated Robotic Controller Using a Low Power Mobile Platform, *Journal of Low Power Electronics and Applications*, 2017, 7, 10.

Conference proceedings:

1. Rizvi, S.T.H.; Cabodi, G.; Gusmao, P.; Francini, G. Gabor Filter based Image Representation for Object Classification, *Proc. of IEEE International Conference on Control, Decision and Information Technologies (CoDIT)*, 2016, pp. 628-632.

2. Rizvi, S.T.H.; Cabodi, G.; Patti D.; Gulzar, M.M. Comparison of GPGPU based robotic manipulator with other embedded controllers, *Proc. of IEEE International Conference on Development and Application Systems (DAS)*, 2016, pp. 10-15.

3. Rizvi, S.T.H.; Cabodi, G.; Arif, A.; Javed M.Y.; Gulzar, M.M. GPGPU based concurrent classification using trained model of handwritten digits, *Proc. of IEEE International Conference on Open Source Systems & Technologies (ICOSST)*, 2016, pp. 142-146.

4. Rizvi, S.T.H.; Cabodi, G.; Francini, G. GPU-only unified ConvMM layer for neural classifiers, *Proc. of IEEE International Conference on Control, Decision and Information Technologies (CoDIT)*, 2017, pp. 0539-0543.

Chapter 2

Deep Learning for Image Classification

Neural networks have the ability to perform analysis of different types of data such as text, image, audio and video [1, 17–20]. However, the field of image processing is heavily impacted by the neural networks.

An Artificial Neural Network (ANN) is a computational model composed of interconnected units called Neurons. A neuron receives inputs from other neurons or sources and computes an output. Each Neuron has associated weights for its inputs and a threshold value (bias). The weighted sum of inputs, bias and activation function of neuron define the final output as shown in Figure 2.1.

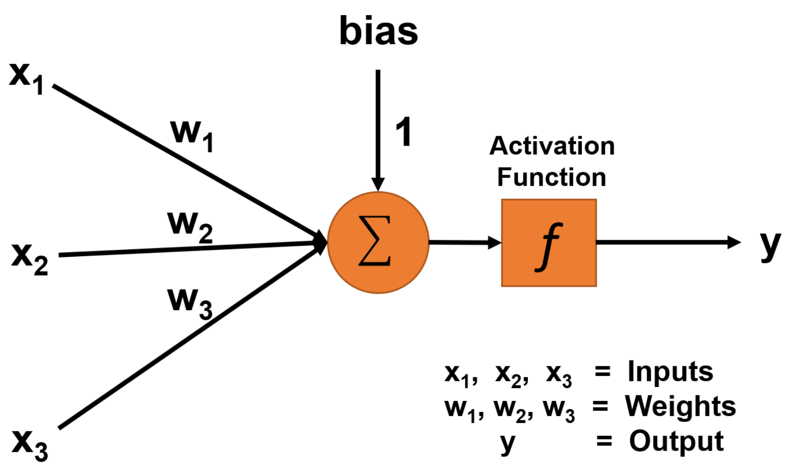


Fig. 2.1 A single neuron of ANN.

These neurons are arranged in layers and form a directed graph known as feedforward neural network. Neurons of these adjacent layers are normally fully interconnected and have weights associated with them as shown in Figure 2.2. A feedforward neural network is composed of input, hidden and output layers. Input and output layers of these feedforward neural networks are used to get the input data and provide the final output. There can be multiple hidden layers in a neural network to perform computations and transfer information from the input layer to the output layer.

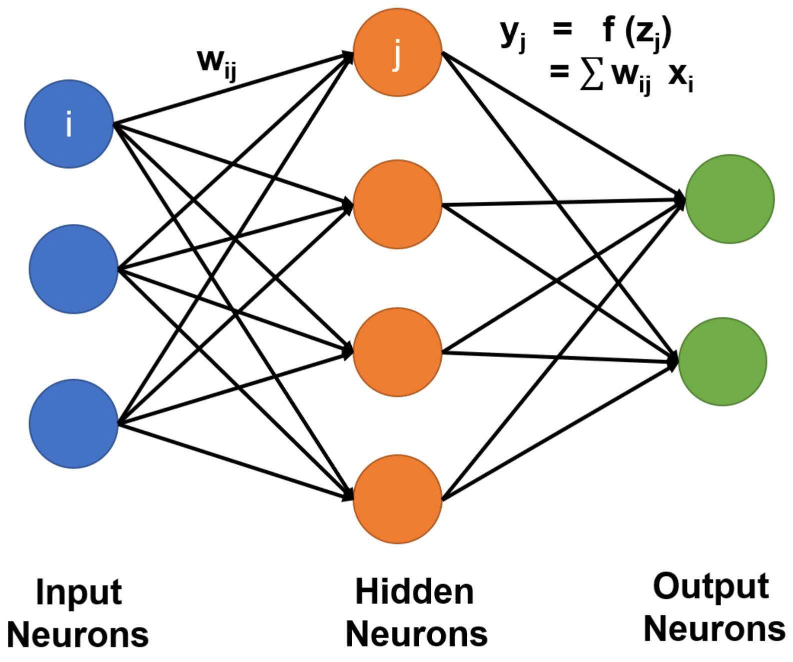


Fig. 2.2 Feedforward neural network.

Deep models are composed of multiple hidden processing layers stacked together to extract the meaningful information from the input data [21]. These networks contain millions of trainable parameters (weights and biases). Deep classifiers learn from the input data using an iterative learning procedure by reducing the difference between the actual and the target values with help of an objective function. During training, the trainable parameters (weights and biases) of classifiers are adjusted repetitively to reduce the error or difference.

Convolutional neural networks (CNN) exhibit unmatched performance in classification tasks. Convolutional neural networks are also feedforward ANNs, but with a special structure having shared weights that help to capture the local properties of the input data or signal. A typical convolutional neural network (CNN) is composed of two stages as shown in Figure 2.3. The first stage of CNN is for feature extraction while other is trained for feature classification. The first stage is mainly composed of convolutional and pooling layers. Convolution operation extracts the features by convolving a trainable filter (weights) with the input image. Convolutional layers preserve the spatial resolution of input data (image) and provide feature maps while pooling layers are responsible for reducing the resolution of feature maps for the manageable representation. Fully connected layers in the second stage classify the features and provide the final output. There are also some others layers for regularization and normalization in both of these stages that improve the performance of a neural network. The deep architectures and their basic components are further discussed in detail in **Chapter 4**.

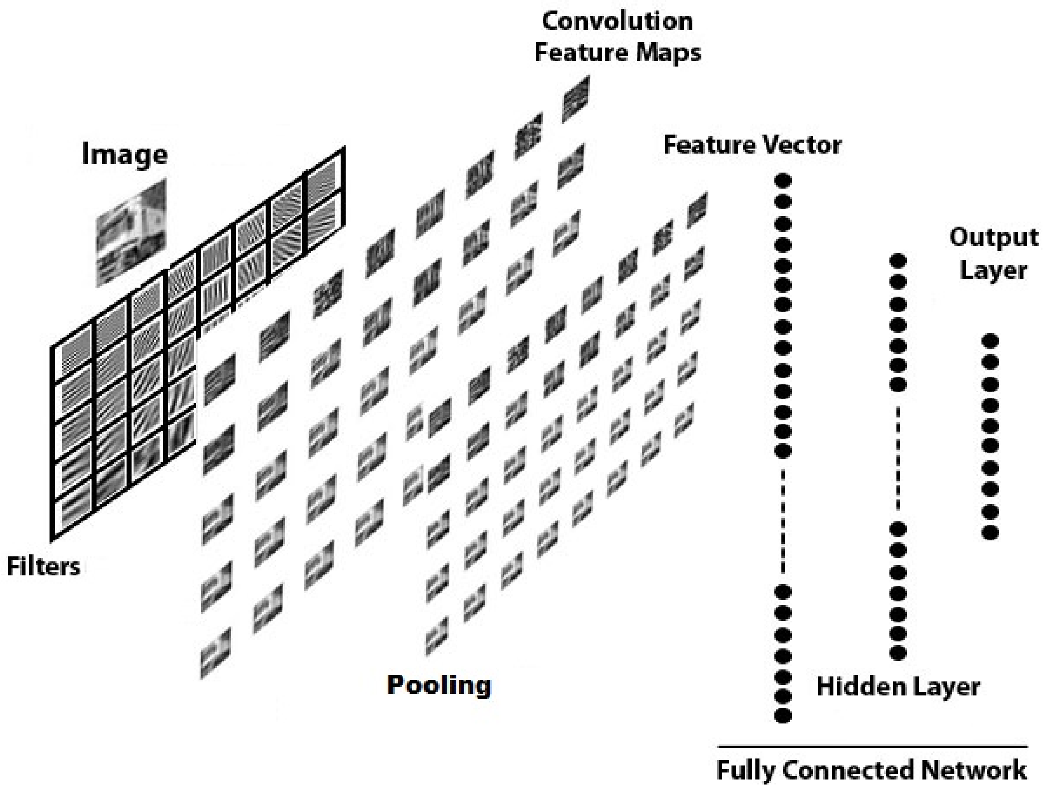


Fig. 2.3 The architecture of a typical CNN.

2.1 Deep Architectures

The neural networks have largely remained neglected till last few years. From 2012, a yearly competition ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) diverted the attention of researchers to neural classifiers through some ground-breaking results and competitive solutions. However, these recent deep architectures demand massive storage capacity and computational power for training and realization. With the evolution of processing units, the computationally and memory intensive neural classifiers can be easily trained and deployed using the powerful desktop workstations and GPU servers often in multi-GPU configuration. This section presents a short overview of the recent evolution of deep classifiers and their contributions to the image classification.

2.1.1 AlexNet

It all started with a very deep convolutional network that was proposed and won the ILSVRC competition in 2012 [1]. This deep classifier was comprised of five convolutional layers and three fully connected layers as shown in Figure 2.4. The network was trained using a dataset of about a million images to classify 1000 different categories.

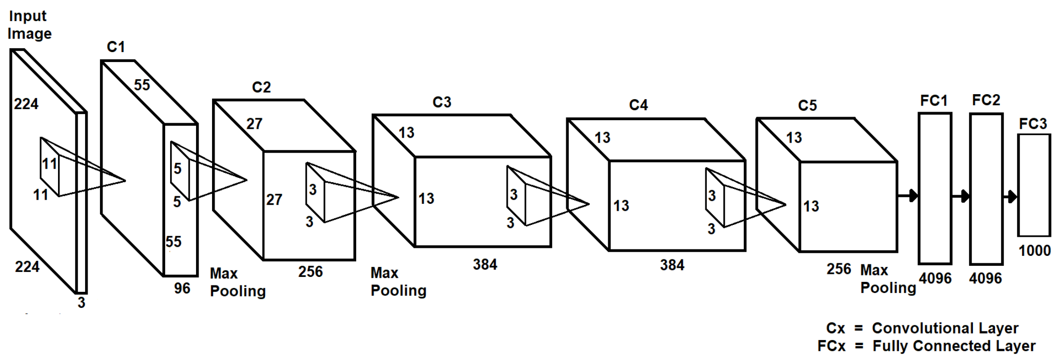


Fig. 2.4 AlexNet.

Results proved the effectiveness of deep classifiers over other state-of-the-art techniques. This success was made possible by the efficient use of GPU, Rectifier Linear unit and a regularization technique ’dropout’ [22]. This was the first time when such a deep network is trained efficiently using the GPU and performed very well on a large and difficult dataset like ImageNet.

2.1.2 OverFeat

In 2013, a multi-scale, sliding window based deep learning approach was proposed for object classification, localization and detection with a single network [23]. This network obtained some astonishing results that proved the effectiveness of CNN for localization and detection tasks. These points were not addressed in previous works.

2.1.3 VGG Net

This research work proposed and evaluated the importance of depth in visual representation [2]. In this work, very small (3x3) convolutional filters were used to steadily increase the depth of a network. Instead of using large receptive fields like previous networks, small filters were used to achieve very deep convolutional networks (having up to 19 layers). Figure 2.5 shows the architecture of a VGG-16 network having 16 convolutional layers. These networks were trained over ImageNet dataset and showed state-of-the-art performance in ILSVRC 2014 challenges. Achieved results reinforced the notion that the classification accuracy of a neural network can be improved significantly by increasing the depth of network.

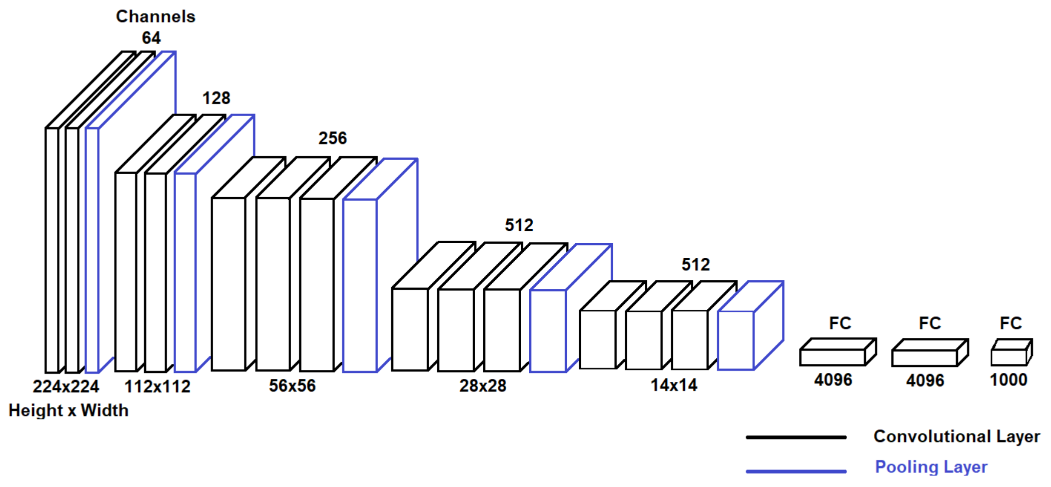


Fig. 2.5 Architecture of VGG-16.

2.1.4 GoogLeNet

Google proposed a computationally efficient deep architecture named GoogLeNet having 22 convolutional layers [6]. A new module known as inception was introduced and used to increase the width and depth of this architecture by keeping the computational complexity constant. The structure of inception module can be visualized using Figure 2.6. Instead of stacking convolutional and pooling layers on each other in sequential manners, inception module concatenated layers in serial and parallel structure to improve the performance and computational efficiency. Using inception module, the GoogLeNet achieved state-of-the-art results for classification and detection in ILSVRC 2014.

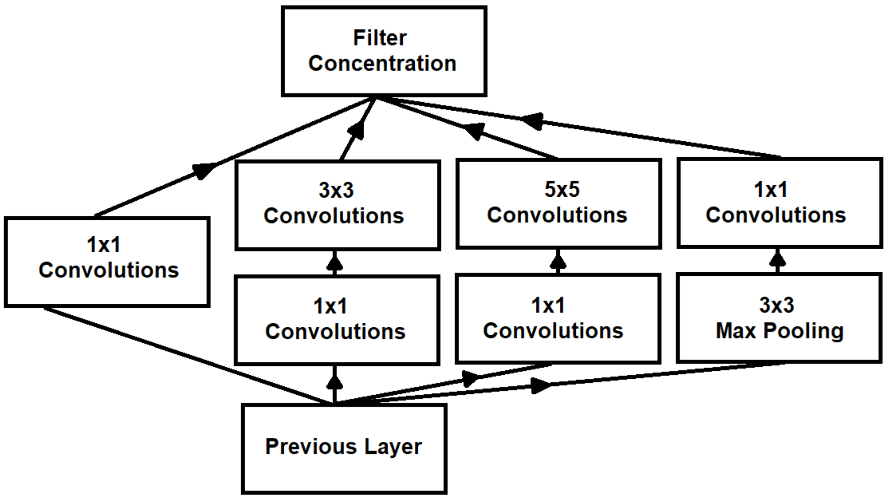


Fig. 2.6 The inception module.

2.1.5 ResNet

A new deep architecture named Residual network (ResNet) was proposed by Microsoft Research Asia in late 2015 [7]. This network used the concept of shortcut connections to learn the difference of representation using residual block shown in Figure 2.7. The purpose of these shortcut connections in the residual network is to force the layers to refine the features [24].

Residual network set the new record in terms of number of layers in a deep architecture and also significantly reduced the number of trainable parameters compared to other networks like VGG Net. The 152-layer ResNet model won ILSVRC classification competition in 2015. These residual networks are the current state-of-the-art on ImageNet dataset.

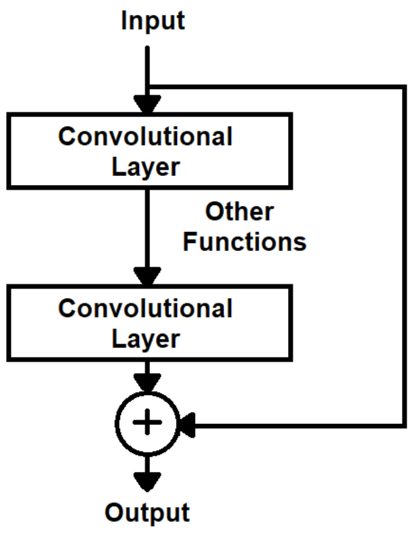


Fig. 2.7 Residual block.

Table 2.1 and Figure 2.8 present a comparative analysis of depth, number of parameters and error rate of previously discussed neural architectures. Since 2012, the error rates of image classifiers have reduced significantly and at the same time, the number of convolutional layers has also increased with the reduction in trainable parameters. State-of-the-art deep neural architectures (Like GoogLeNet and ResNet) have human-level classification accuracy and, are less computationally and memory demanding due to the reduced number of parameters. These deep architectures can be realized on an embedded platform for the real-time image classification with the help of an optimized framework.

Table 2.1 Comparison of convolutional neural networks (ILSVRC challenges).

Model	No. of Layers	Top-5 Error Rate	Number of Parameters
AlexNet	8	16.4 %	60 million
OverFeat	8	11.7 %	145 million
VGG	19	7.3 %	144 million
GoogLeNet	22	6.7 %	6.8 million
ResNet	152	3.57 %	1.7 million

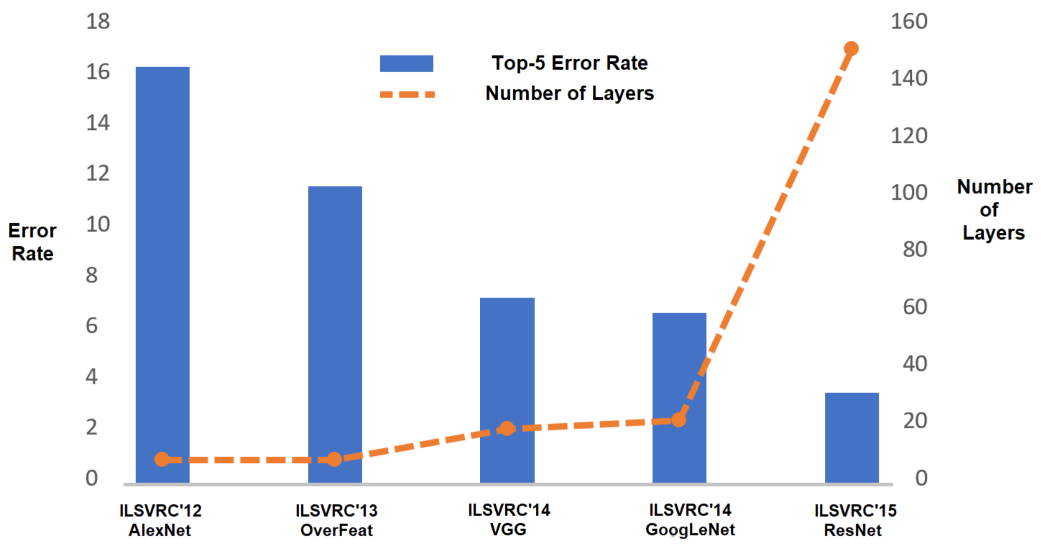


Fig. 2.8 Evolution of deep convolutional neural networks.

Chapter 3

Overview of GPGPU Programming

With the rapidly evolving computing technology, the General-Purpose Graphics Processing Unit (GPGPU) has emerged as a competitive accelerator for computationally demanding applications like training of neural networks. The architecture of a GPGPU is highly parallel and can provide massive computational power than the CPUs. A GPGPU can be programmed using a high-level programming language to exploit its multi-core multi-thread structure for accelerating the general-purpose applications.

3.1 Programming Languages for GPU Computing

A GPGPU can be programmed using two different programming interfaces, Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) [25, 26]. OpenCL is an open standard that can be used to program various devices (GPUs, CPUs etc.) from different vendors, while CUDA is proprietary language that can only be used to program Nvidia GPUs.

A GPU program is composed of the device (kernel) and the host codes. A kernel code is only executable on the device (GPU) and can only be called by host side, while the host code is executed by the CPU and manages the resources like initialization of device, allocation of memory, transfer of data and launch of kernel etc.

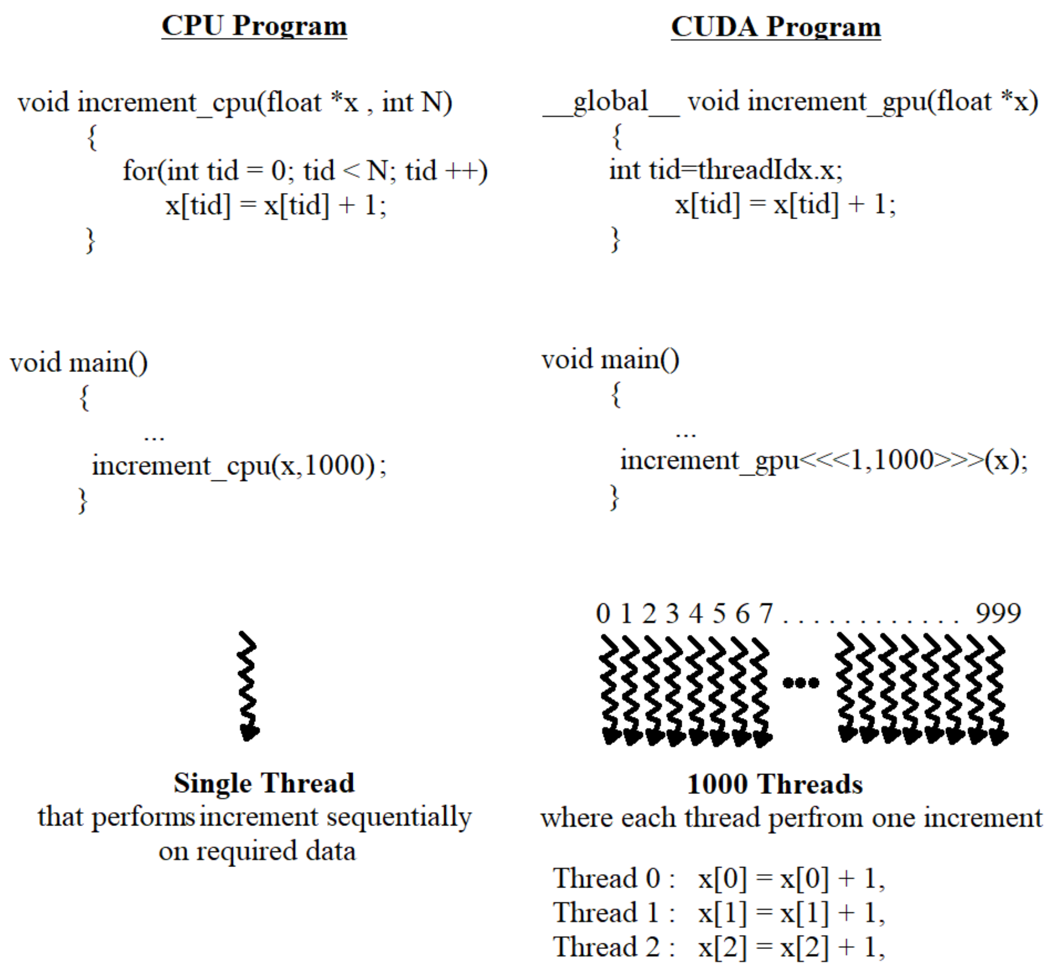
Both OpenCL and CUDA launch the kernel from host side (CPU) and execute it on the device (GPU). Porting the kernel code from one language to the other requires minimal modifications. However, the host code for both languages is substantially different and needs to be completely rewritten according to the syntax and libraries of the required programming environment. Both languages provide some similar host functions to set up the GPGPU for execution of the kernel code. Context initialization, data allocation and data transfer are few examples of these host functions.

Although OpenCL is a portable language and supports a variety of devices, but it is less efficient than the CUDA-based libraries and functions in terms of execution time due to its generality [27, 28]. OpenCL functions need to be optimized carefully for a specific hardware. Furthermore, OpenCL is not directly supported on Android Operating system from version 4.3 onwards. An intermediate library is required to access the mobile GPU and its resources. Such a library not only adds extra computational overhead, but also increases the storage requirements and complexity of source application. Therefore, it is not feasible to use the standalone OpenCL for the realization of deep classifiers on mobile devices. In addition, CUDA libraries are widely used by the deep learning community. In this work, CUDA computing language is used to realize the deep classifiers on embedded platforms. The next sections of this chapter briefly discuss the thread and memory hierarchy.

3.2 Arrangement of Computing Resources

As a device code is invoked, the GPU launches a number of threads to concurrently execute the instructions of the kernel function; while the same program written for CPU executes the instructions sequentially using a single thread. As shown in Figure 3.1, the required operation of increment is performed concurrently on a GPU using a block of threads. The concurrent execution of threads can significantly improve the performance of the data independent applications.

GPGPU is a multi-core, multi-thread system where these threads are organized into a multi-dimensional grid of independent blocks. Each block is comprised of concurrent threads that operate cooperatively within each block as shown in Figure 3.2. The Single instruction multiple threads (SIMT) programming model of GPU is an extension of single instruction multiple data (SIMD) approach. In SIMD programming model, the same operation is applied to all data items, while SIMT execution model provides flexibility by combining the SIMD model with multi-threading where a group of threads (called warp) perform the same operation. A group of 32 threads forms a warp.



CPU Program

```
void increment_cpu(float *x , int N)
      {
         for(int tid = 0; tid < N; tid ++)
              x[tid] = x[tid] + 1;
      }


void main()
      {
            ...
       increment_cpu(x,1000);
      }
```

CUDA Program

```
__global__ void increment_gpu(float *x)
      {
      int tid=threadIdx.x;
            x[tid] = x[tid] + 1;
      }


void main()
      {
            ...
       increment_gpu<<<1,1000>>>(x);
      }
```

Single Thread
that performs increment sequentially on required data

1000 Threads
where each thread perfrom one increment

```
Thread 0 :   x[0] = x[0] + 1,
Thread 1 :   x[1] = x[1] + 1,
Thread 2 :   x[2] = x[2] + 1,
```

Fig. 3.1 Structure of a CUDA program.

Depending on the architecture of GPGPU, there is a limit on the number of threads per block. There can be maximum 1024 threads in a block, so multiple blocks may be required to execute a single kernel. Total number of threads invoked during execution can be calculated using following formula:

$$Total\ number\ of\ Threads = Threads/block \times Number\ of\ blocks \tag{3.1}$$

The number of blocks in a grid is defined with accordance to the size of data to be processed. These blocks execute independently across the different cores of a GPU.

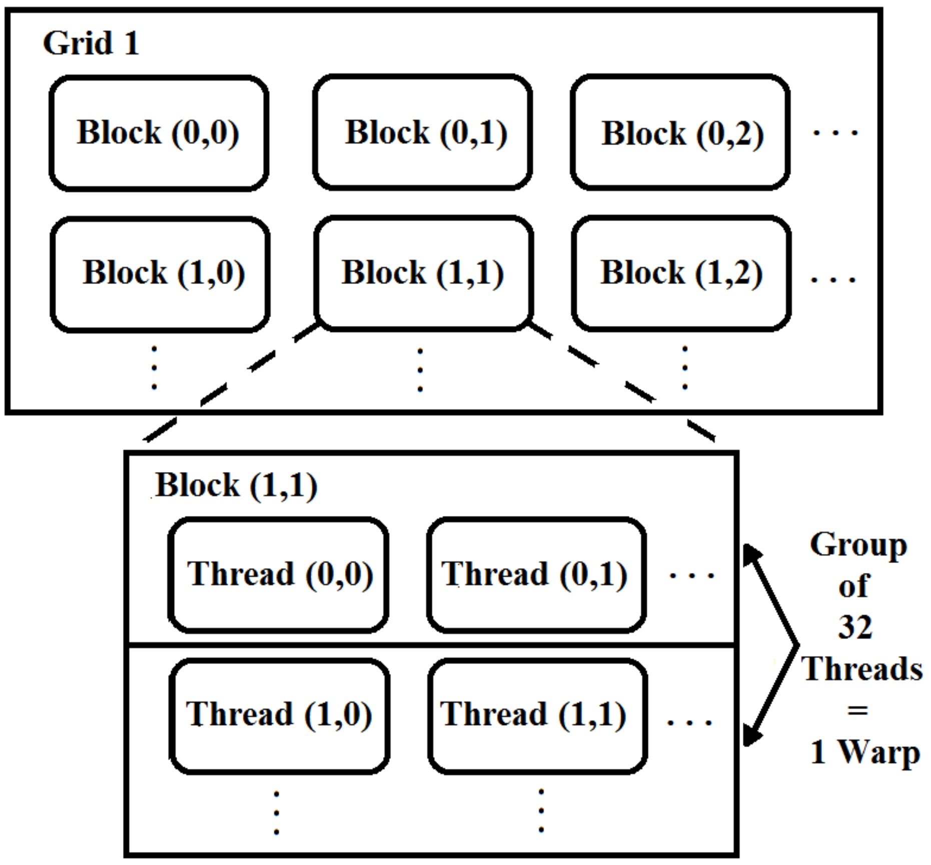


Fig. 3.2 Arrangement of GPU resources.

3.3 CUDA Memory Model and Hierarchy

For GPU-based computations, the input data to be processed must be copied from the host (CPU) memory to the device (GPU) memory and the output data (final results) must be retrieved from the GPU memory to the CPU memory. CUDA devices have different types of memory spaces which offer various performance characteristics. Data from these different memory spaces may be accessed by threads during execution. Hierarchy of CUDA memory model can be visualized using Figure 3.3.

Every thread has registers or private local memory that is only available to one thread. Each block has its own shared memory that is available and visible to all threads within the same block. Data present in global memory can be accessed by all threads of an application. Figure 3.4 presents a visual representation of threads, blocks and grids with corresponding per-thread private, per-block shared and per-application global memory spaces.

Constant and texture memory spaces are two additional read-only memory portions in CUDA processors that are generally used to store the constants, kernel arguments and to support optimized 2D access pattern.

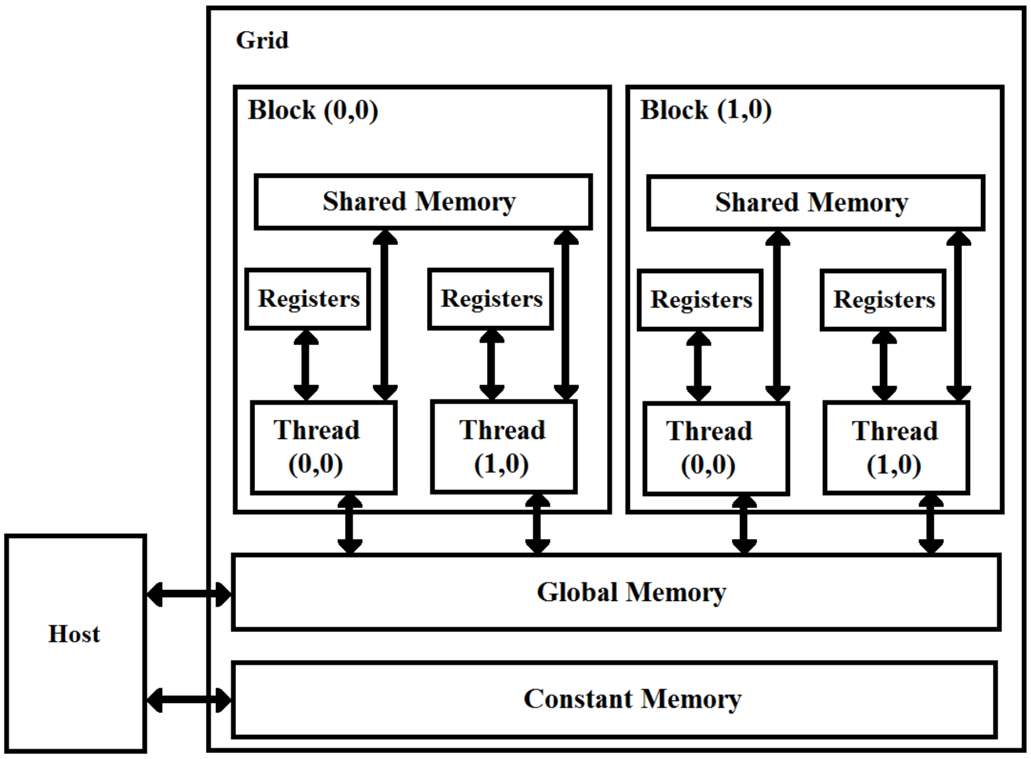


Fig. 3.3 CUDA memory types.

The global memory is the main and largest memory space in the graphics processing unit. The computational data is usually stored and transferred between the host (CPU) and the device (GPU) through global memory. The latency to access global memory is generally higher. However, its performance can be improved by coalescing the memory accesses. Shared memory is on-chip and faster than global memory, but it is limited and its performance is subject to bank conflicts among threads.

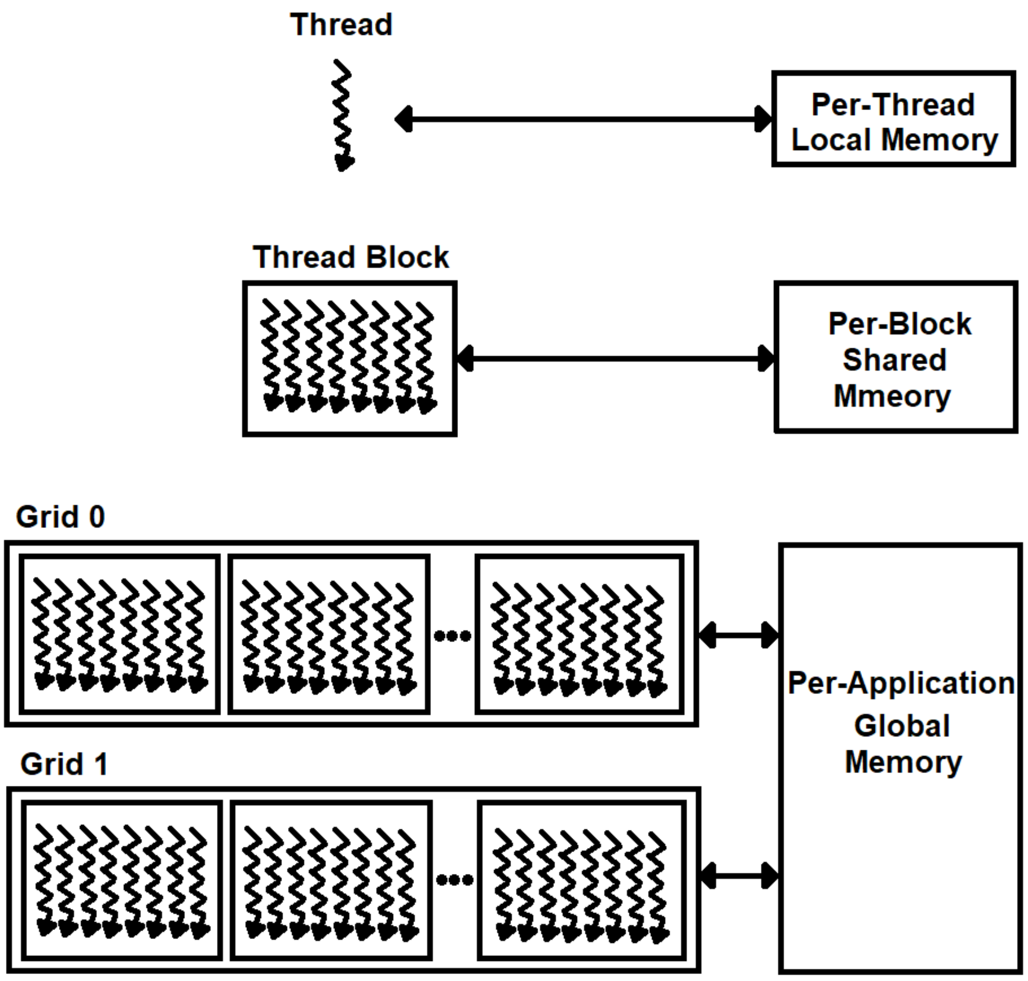


Fig. 3.4 CUDA hierarchy of threads, blocks and grids with corresponding memory spaces.

Proper utilization of computing and memory resources can significantly improve the performance of CUDA-based general purpose applications. The next chapter presents the adopted methodology to realize the deep classifiers on embedded platforms using CUDA computing framework.

Chapter 4

Neural Architecture and CUDA-based Proposed Realization

This chapter describes the most common layers required by the neural networks to perform image classification and our proposed approach to realize the deep classifiers on the embedded platforms[1]. In this work, the CUDA computing language is used to replicate the neural architectures and exploit the computational power of the embedded GPUs. Namely, our proposed framework relies on a GPU-only scheme for performing convolutions in the transform domain. The performance of this GPU-only scheme is compared with the heterogeneous and sequential versions. Different neural architectures are considered to evaluate the performance of the proposed CUDA-based scheme.

4.1 Introduction

Image classification via convolutional neural networks (CNN) involves two stages. The first stage is an offline learning state to train the required neural architecture over a set of labeled input data (images) and the second stage is testing or inference phase where a proper image classification can be performed using the trained network. During training stage, the network parameters are iteratively updated to predict the

[1]Part of the work described in this chapter has been previously published in **Electronics** Journal as "Rizvi, S.T.H., Cabodi, G., Patti, D. and Francini, G. "GPGPU Accelerated Deep Object Classification on a Heterogeneous Mobile Platform," Electronics 2016, 5, 88.

output values corresponding to the input labels. A trained network can classify the new images using the learned set of parameters (weights and biases).

Deep convolutional neural networks demand high computational power and memory storage. These are the constraints that limit the possibility to realize the deep classifiers on the embedded devices. Due to the huge demand of resources, the training stage can only be performed offline on a dedicated GPU or server infrastructure. However, the actual image classification may has to be performed online on an embedded device for the real-time visual analysis [8]. There are numerous practical real-time applications where the hand-held devices can be more useful due to portability and on-board resources [29, 30].

Recent embedded devices employ the heterogeneous architectures and are well suited for challenging real-time applications [31–34]. As mentioned earlier, training and testing phases are computationally intensive problems, so heterogeneous (GPU-CPU) resources of an embedded platform can be exploited to accelerate the sequential and concurrent processes in such problems. GPUs can be exploited to accelerate the concurrent processes, whereas the CPUs can execute the sequential tasks efficiently due to their higher operational frequencies. Training and classification time of neural classifiers can be largely improved by proper scheduling of computational resources of the heterogeneous architectures [32, 33, 35–37].

4.2 Related Work

As mentioned earlier, many computing frameworks like Torch and Caffe are currently available for implementing the neural networks. These frameworks rely on a number of third-party libraries to leverage the computational resources of modern GPUs for faster training and deployment of deep neural classifiers. However, such libraries are designed for computationally powerful environments like desktop workstations and server platforms. Therefore, these libraries do not consider the unique peculiarities of embedded platforms and cannot be directly used on a mobile device. Realization of neural architectures on a low power embedded platform is still a challenging problem, and there have been very few studies conducted on this.

An OpenCL-based deep CNN framework is proposed for mobile devices in [38]. This scalable framework leverages a variety of optimization techniques like memory vectorization and half floating-point processing to execute several CNN models in real-time, with no or marginal accuracy tradeoffs. However, this scheme relies on OpenCL language, which is less efficient than the CUDA-based same libraries and built-in functions in terms of execution time; therefore, demand much optimization [28]. Furthermore, OpenCL is not directly supported on Android Operating system from version 4.3 onwards.

An efficient mobile GPU-accelerated Deep Neural Network flow is presented in [39]. Different optimization techniques to increase the computing efficiency are discussed. This DNN flow shows significant speedup and higher energy efficiency over the CPU-based sequential version. However, the information that is presented is not enough to reproduce the results.

An open source GPU-based library "CNNdroid" is introduced in [40]. This library is designed to realize the trained neural classifiers on Android-based mobile devices. However, this library does not provide support for all CNN layers required to realize the state-of-the-art neural classifiers like ENet and Residual Network.

An efficient embedded framework named "Quantized CNN" is proposed in [41]. This framework simultaneously improves the computation time and reduces the storage/memory overhead and power consumption. This gain in performance is achieved by compressing the convolutional neural networks for realization on mobile devices. However, results also show that the quantization and compression of parameters jeopardize the precision of results [42–44]. Furthermore, state-of-the-art deep classifiers can be directly used on a mobile platform without any need of quantization, because these classifiers already have fewer parameters and provide state-of-the-art classification accuracy [45].

Above mentioned schemes focus on the efficient realization of testing/inference phase on the embedded platforms, however, a lot of research work is also going on the training of memory efficient neural architectures [46–48]. These works propose that more computationally and memory efficient deep neural architectures can be constructed by approximating, quantizing or compressing some of the parameters during the training phase.

Concluding, the contributions of our proposed framework and improvements over the above-mentioned schemes can be summarized as follows. First of all, the proposed scheme has the merit of supporting nearly all layer types of deep neural architectures and suitable for deploying complex state-of-the-art deep classifiers. All required layers and functions are realized using CUDA computing framework. Intermediate frameworks and memory consuming computational packages are avoided using the proposed CUDA-only scheme to resolve the problem of software dependencies and reduce the storage requirements. In addition, the accuracy of our proposed scheme is similar to the existing desktop and server frameworks. Furthermore, the proposed framework can be easily integrated into an Android application for actual classification and provides compatibility for models trained with other desktop or server frameworks.

4.3 Adopted Methodology

Figure 4.1 presents the proposed flow to realize a deep classifier on an embedded platform. First of all, Torch computing framework is used to train the state-of-the-art deep neural classifiers. These neural classifiers are trained using the powerful desktop workstation or GPU server depending on the required computational power. Some preprocessing techniques are performed to enhance the computational efficiency of training process and accuracy of results [49]. Neural networks are normally trained on a fixed size of training data, while the training data can be of various sizes and dimensions because of its collection from different sources. Therefore, the training images are required to be scaled or cropped to fit the defined architecture. Scaling and cropping can be performed using different approaches.

Normalization of input and target data is also an important preprocessing step for proper training of a neural network. Data normalization is generally useful to map the widely-spaced scale data to a uniform scale. It is performed here by normalizing the data to have zero mean and unity variance, also called mean-standard deviation normalization.

Since the deep networks need a large amount of training data to improve the performance and classification accuracy, data augmentation is used to artificially increase the amount of training data. This image augmentation is performed through different augmenting techniques like color jittering, random crops, horizontally flipping, etc. Finally, the neural architectures are trained and can be used for classification purposes in the field.

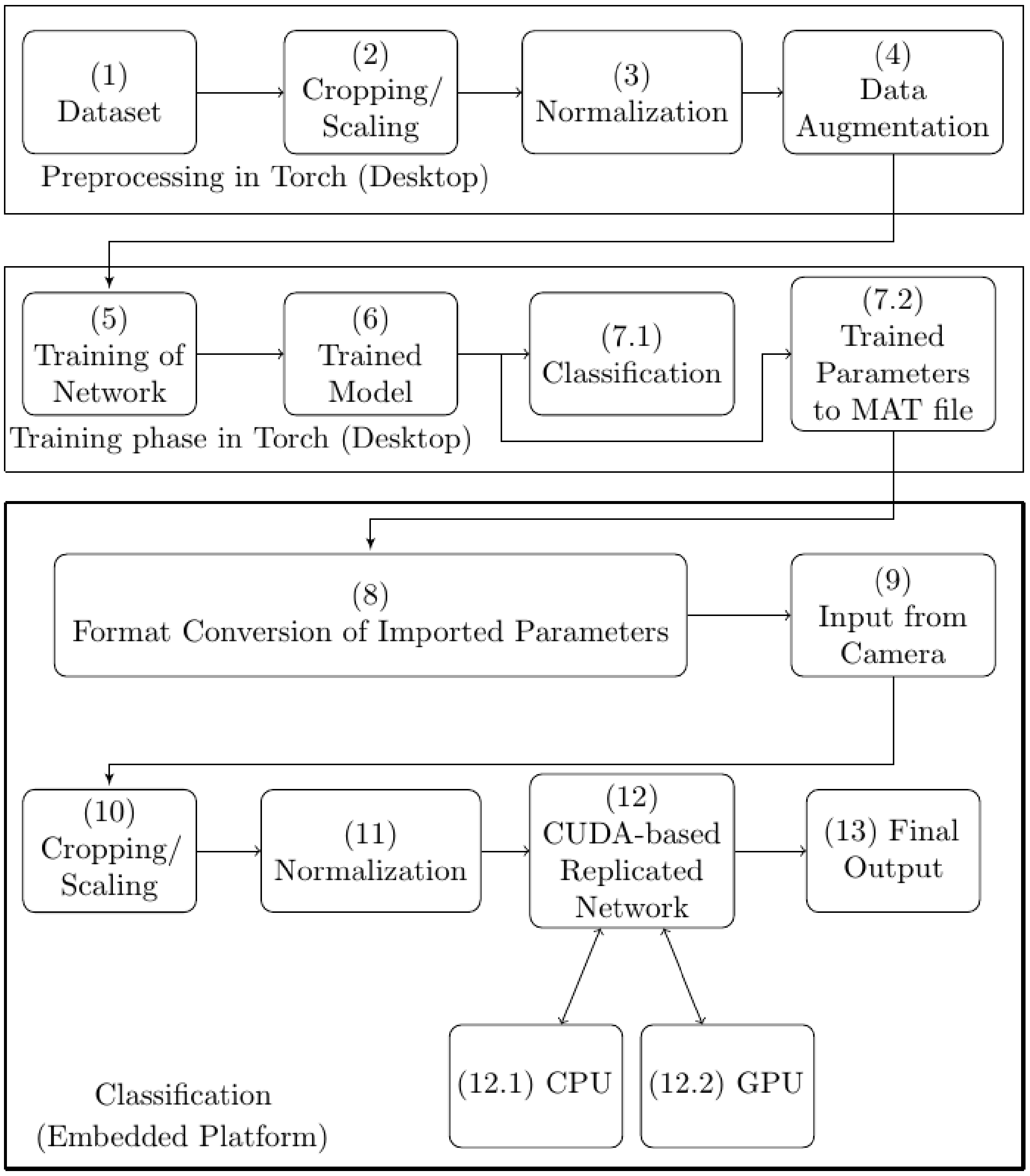


Fig. 4.1 Block diagram of the CUDA-based neural classifier for embedded platform.

Once a network is trained, its trained parameters can be imported for the actual classification. In this work, the CUDA programming framework is used to replicate the trained neural architectures on the embedded platforms. All required layers and activation functions are implemented in CUDA to realize the identical architectures. The computational power of GPGPU is exploited to accelerate these CUDA-based identical functions (Convolution, pooling, batch normalization, rectifier linear unit, etc.). Convolution is the most computationally intensive layer of neural classifiers.

Two versions of convolution operation are implemented to perform a comparative analysis and support the different neural architectures trained in torch framework. The trained parameters from Torch framework are formatted in different ways depending on the used convolution method. So it is important to perform the format conversion of imported parameters to achieve the same results on the embedded platforms. By importing the trained parameters and realizing the identical architecture, these CUDA-based deep classifiers are executed on the embedded platform. Furthermore, these classifiers are realized on a portable embedded device as a part of the Android application.

The onboard/integrated camera of the embedded platforms is used to acquire the input image as shown in Figure 4.1. Real-time image classification is performed on this captured image using the trained networks. Same preprocessing steps are required to be performed on the captured image as done at the time of training. As the trained classifiers are designed for a specific input dimension and the captured image can be of different aspect ratio depending on the camera of the embedded platform, so it is essential to crop or scale the image to the exact size required by a neural network. Similarly, it is necessary to normalize the input image to minimize the bias towards different features and map the input/output values to a specific range. After these steps, the captured image is fed and classified by the CUDA-based replicated networks on the embedded platform using imported trained parameters. Final classification results can be displayed on the screen of a portable embedded device or can be used by another application for the further developments.

4.4 Architecture of the Convolutional Neural Networks

This section presents the architecture of deep classifiers and the entailed layers.

4.4.1 Convolutional Layer

The convolutional layer is core building block of deep classifiers. It extracts the robust pattern or features from the input images. Mathematically, convolution can be expressed as the summation of point-wise multiplication of filter/mask coefficients with the input image/function. A filter is an integral component of the layered architecture. Convolution operation extracts the features by convolving a filter with the input image. These features or output of convolution operation are called activation maps. Filters are trainable parameters that are adjusted during the training to predict the meaningful aspects of the input data.

State-of-the-art deep classifiers are composed of multiple convolutional layers [6, 7, 23]. By stacking a number of convolutional layers, accuracy of a neural classifier can be significantly improved [1, 6]. However, convolution is a computationally intensive operation and demands high computational power and memory storage. Therefore, these factors limit the realization of convolutional neural networks on the embedded platforms. By an effective and efficient realization of the convolution operation, the computational burden of these layers can be reduced and an embedded GPU can be exploited to accelerate the neural classifiers.

There are different methods to compute the convolution operation. In this work, two approaches to compute the multi-channel convolution operation are discussed, realized and compared for the adaptation of torch-based trained networks on the embedded platforms.

The first and traditional approach is full convolution. In this approach, convolution is computed by the sum of products of the filter and the input image. This operation is extended to perform the multi-dimensional convolutions as required by the neural classifiers.

Both input and output images are three dimensional. If the input and output images have C and D channels respectively and sizes of the trained filter bank and the input image are X x Y and I x J respectively, then the output of a multi-channel fully-convolutional layer can be expressed by the following equation:

$$Output\_map_{i,j,d} = \sum_{c=1}^{C} \sum_{i1=1}^{X} \sum_{j1=1}^{Y} Input\_map_{c,i+i1,j+j1} Filters_{d,c,i1,j1} \tag{4.1}$$

Figure 4.2 illustrates the example of a fully convolutional layer where the pixels of input maps, output maps and filters are defined by their index positions. As both input and output maps have three dimensions, so their pixels are represented by (row, column and channel number) while the pixels of filter banks are represented as (row, column, input channel number and output channel number) because the filter bank is 4 dimensional. The index value (3,2,1) of input map represents the pixel present on the first channel of the input image at the third row and second column. In this particular example, the input image is of size 3 x 3 and has 3 channels. There are 2 filter banks each having 3 channels. Size of each filter is 2 x 2. The number of channels in a filter bank depends on the channels of input maps while the number of filter banks defines the number of channels of output map. So for this particular example, output map would have 2 channels and size of output map would be defined by the size of the filter and few other parameters defined in Section 4.4.6. In this approach, filters and input maps have to convolve in a sliding manner that can be a slow process which cannot fully exploit the concurrent resources of a GPU [50].

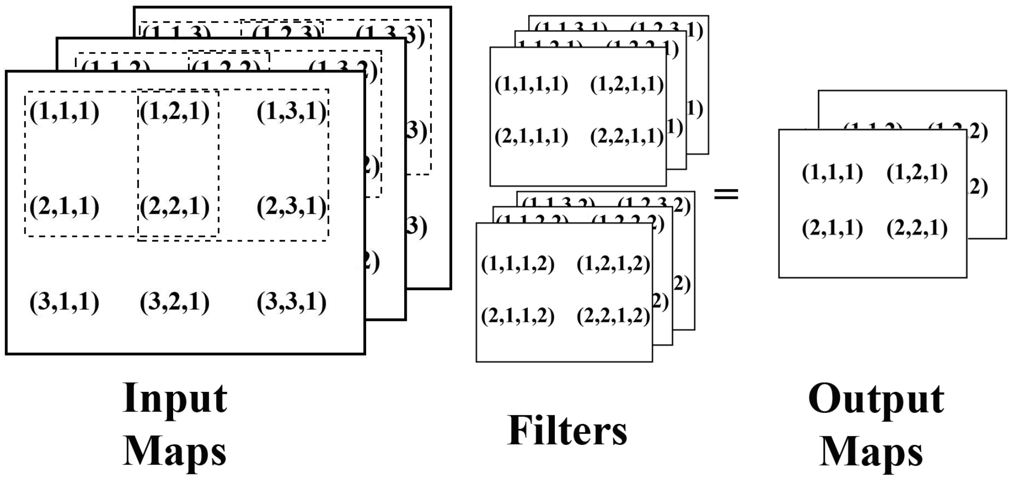


Fig. 4.2 Multi-dimensional full convolution.

The second approach realized for the computation of convolution is based on Matrix Multiplication. In this Matrix Multiplication based Convolution (ConvMM) approach, first, the input maps and filter banks are transformed into two-dimensional matrices and then directly multiplied [51, 52]. Figure 4.3 illustrates the matrix multiplication based version of the convolutional layer where the input image and filters are arranged as the matrices in the transform domain.

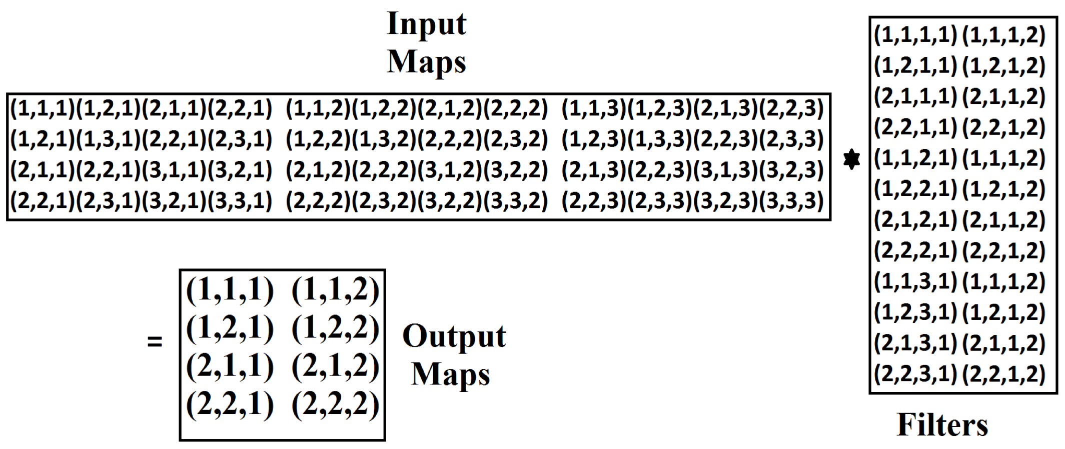


Fig. 4.3 Matrix multiplication based convolution (ConvMM).

Figure 4.3 also explains the mapping of pixel values from the multi-dimensional input maps and filter bank to the two-dimensional matrices. Figure 4.3 represents the pixels shown in Figure 4.2 in two-dimensional transformed form. Resultant matrix (output map) can be calculated by multiplying these transformed input and filter matrices. Values of the resultant matrix can be arranged to achieve the output representation equivalent of the traditional convolutional layer.

Convolution can also be computed using other approaches like Winograd minimal filtering algorithm, fast Fourier transformation and lookup table-based approaches [42, 51–55]. Using fast Fourier transformation (FFT)-based approach, convolution can be computed as point-wise products in the Fourier domain [53]. This FFT-based convolution requires large memory space to store the transformed data that can be a problem for existing GPUs having a limited amount of on-chip memory. Furthermore, FFT-based approach is fast for large filters, while state-of-the-art deep classifiers use small filters [7]. For networks having small filters, minimal filtering algorithm pioneered by Winograd can be employed to compute the convolution over small tiles [42]. This algorithm is memory friendly and fast for small filter and batch sizes. The Winograd technique minimizes the complexity by performing element-wise multiplication instead of matrix multiplication and reduces the number of arithmetic operations. Some optimization techniques can also be used to improve the performance of these convolution approaches [56–58].

4.4.2 Pooling Layer

In a traditional convolutional network, every convolutional layer is followed by a pooling layer. Convolutional layers preserve the spatial resolution of input data (image) while pooling layers are responsible for reducing the resolution of feature maps for the manageable representation. This down-sampling doesn't affect the depth (Channels) of feature maps. The most common pooling approaches are max- and average-pooling. These layers do not have any trainable parameter like convolution. There are also other pooling techniques and regularization methods that can be used to improve the training speed and classification accuracy of the convolutional neural networks [50, 59].

4.4.3 Batch Normalization

Batch normalization is also an important part of state-of-the-art deep classifiers. It dramatically speedup the training of neural classifiers [60]. By adding some extra trainable parameters, batch normalization preserves the representation ability of the network. Furthermore, the resulting networks do not require any further regularization. This function can be expressed as follows:

$$BN(x) = \frac{x - mean}{\sqrt{Var}} * gamma + beta \tag{4.2}$$

In batch normalization layer, values of mean and variance are estimated from the training data; while beta and gamma are the trainable parameters. This layer first normalizes each scalar feature by making it have the zero mean and unit variance, and then gamma and beta are used to scale and shift the normalized output.

4.4.4 Activation Functions (ReLu, Tanh and Threshold)

There are several activation functions for neural networks, like the rectifier linear unit, threshold unit and tangent hyperbolic unit. Activation function is one of the important components of a neural network. It is a non-linear decision-making function that determines the presence of a particular feature. It improves the training of a neural network and further eliminates the problems like vanishing gradient [61].

4.4.5 Linear or Fully Connected Layers

As the name implies, this layer is fully connected to the output of the previous layer. These layers are typically used in the last stages of a classifier. Where convolutional layers extract the robust features, these fully connected layers (multilayer perceptron) perform the feature classification and provide the final output. These layers also have trainable parameters (weights and biases) like convolutional layers. Furthermore, most of the parameters of a network are in the fully connected layers.

4.4.6 Sizing Convolutional Neural Networks

The size of the output map of a convolutional or pooling layer is dependent on the size of the filter,the striding size and the number of padding bits used in the input image.

The stride size controls the movement of the filter around the input map in the convolution/pooling operation. It defines the amount by which the filter shifts and directly affects the size of the output map as shown in Figure 4.4.

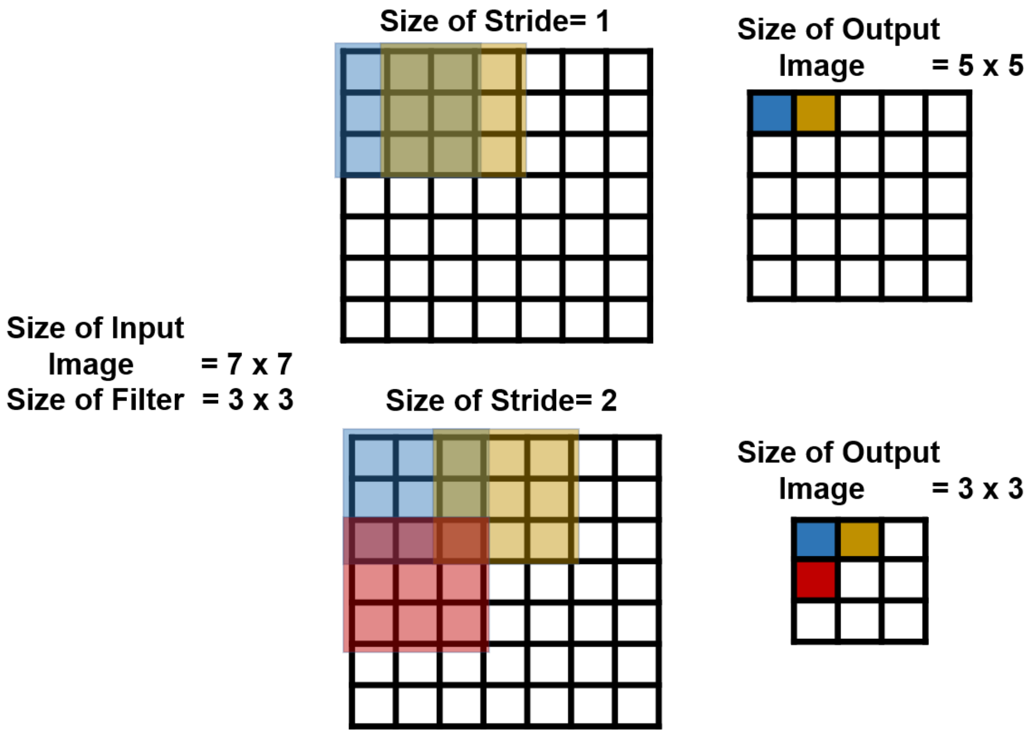

Size of Stride= 1
Size of Output Image = 5 x 5
Size of Input Image = 7 x 7
Size of Filter = 3 x 3
Size of Stride= 2
Size of Output Image = 3 x 3

Fig. 4.4 Effect of stride length on the size of output maps.

It can be visualized from the previous figure that the size of output map would always decrease after the convolution and pooling operations. In some cases, it is required to preserve and attain the size of maps. Zero padding is used to resolve this problem and achieve the desired size of output maps. As shown in Figure 4.5, the size of output map is same as the input map after required operation due to additionally padded zeros around the border of input maps.

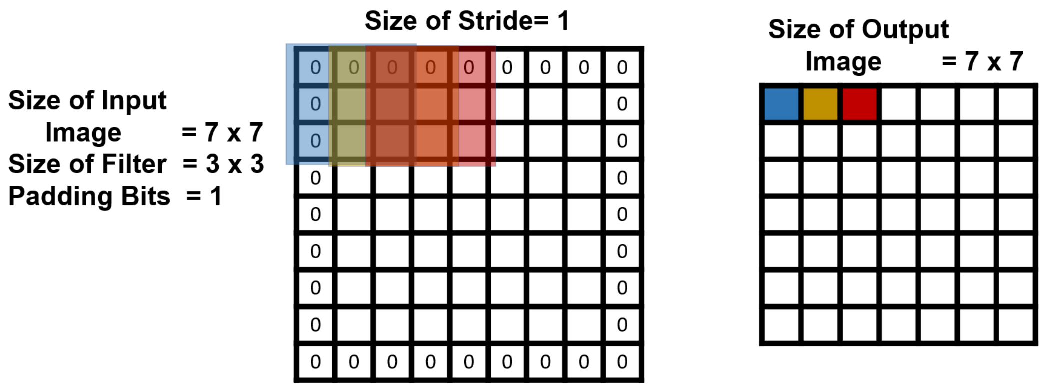


Fig. 4.5 Effect of padding and stride length on the size of output maps.

4.5 CUDA-based Proposed Framework

This section discusses the CUDA-based realization of the deep architecture for image classification on embedded devices.

4.5.1 GPGPU-accelerated Fully Convolutional Layer

The CUDA computing language is used to implement the traditional fully convolutional layer. The computational power of GPGPU is exploited to accelerate this layer. A 3-dimensional grid of blocks and threads is used to offload the complete workload into the GPU and concurrently compute the convolution operation. To realize the Torch-based identical classifier, formatting of the acquired input image and trained parameters is one of the most important tasks. As mentioned earlier, trained parameters of the fully-convolutional layer and matrix multiplication based convolution have structurally different format. So format conversion of imported parameters is essential to acquire the same results as provided by the Torch-based model. The row-major layout is used to store and access the multi-dimensional trained parameters, input image and results. This contiguous memory allocation speedup the access of array elements. Padding function is also implemented and accelerated using GPGPU. This CUDA-based padding function is realized to provide the identical functionality and architecture as provided by the convolutional layers of Torch framework.

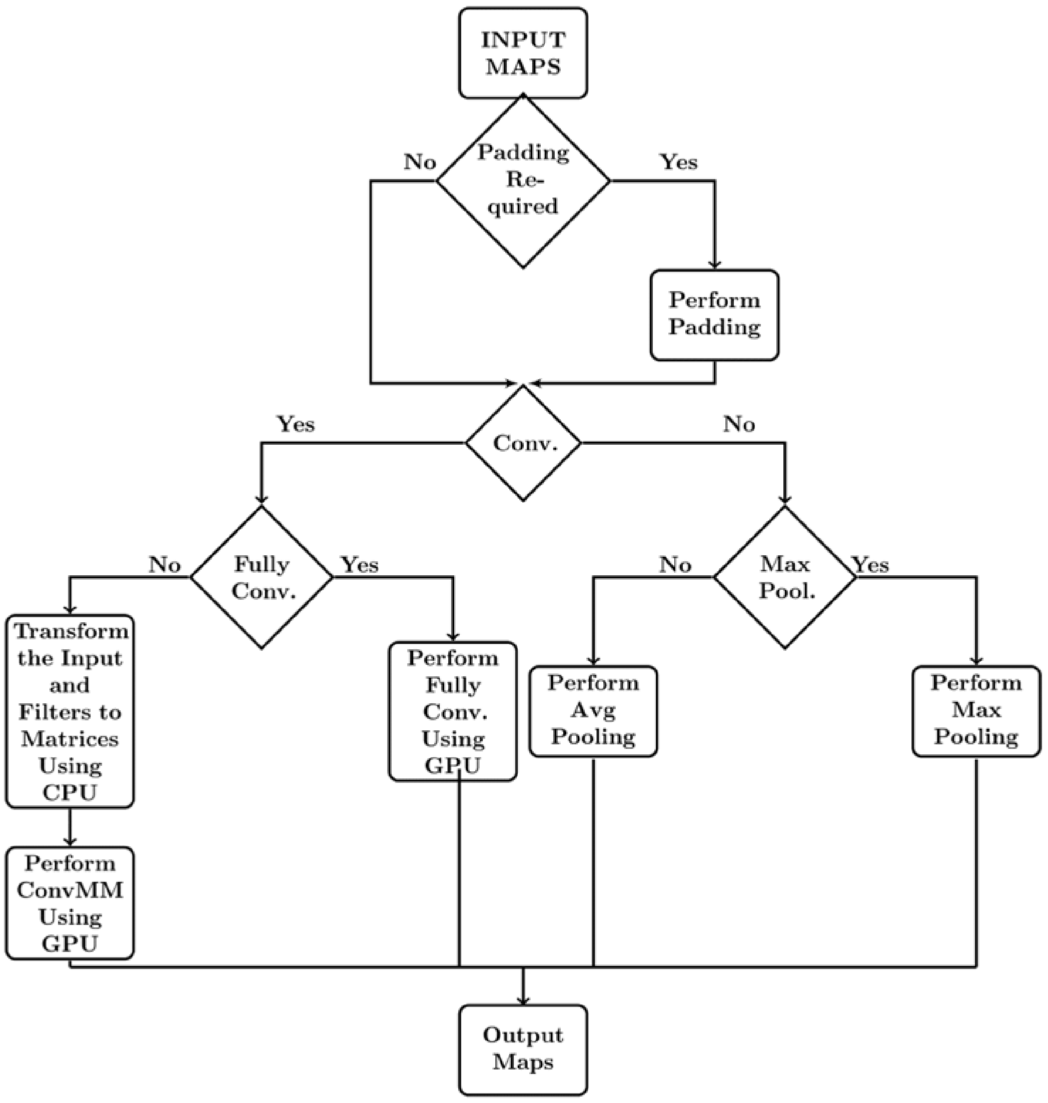


Fig. 4.6 Flow of GPGPU-based convolutional and pooling layers.

The realized convolutional and pooling layers have same flow because both are composed of two kernels. The first kernel performs the concurrent padding of input data and the second kernel executes the selected operation of pooling or convolution on this padded input. All others layers like the batch normalization and threshold unit, where the padding function is not required are computed using a single kernel. The flow of both convolutional and pooling layers is depicted in Figure 4.6. First of all, upon execution, padding is performed on the input data. And if it is not required to perform padding, then directly the second kernel is called for execution. This second kernel can be either convolution or padding function depending on the defined flag. There are two further options in both convolution and pooling kernels. In case of convolution kernel, it is checked that either full convolution or matrix multiplication based convolution is required to be performed. Similarly, in case of pooling function, average or max function is invoked depending upon the defined flag.

4.5.2 Heterogeneously Accelerated ConvMM Layer

Matrix multiplication based convolutional layer is also realized using CUDA language. Heterogeneous resources of the embedded platform are fully utilized to compute the solution of the ConvMM layer. This layer is accelerated by offloading the appropriate computations between the CPU and the GPGPU of the embedded platform. The sequential tasks like the transformation of input maps and filters into two-dimensional matrices are assigned to the powerful CPU, and the concurrent matrix multiplication of this transformed data is performed using the GPU. In this transformation step, pixel values from the multi-dimensional input maps and filters are required to be placed on the distinct locations of matrices in a specific sequence. As CPUs are best suited for sequential computations, this sequence sensitive transformation can be performed efficiently using the CPU. An algorithm like this can benefits from the heterogeneous accelerators in terms of execution time [33, 62, 63].

Figure 4.7 illustrates the scheduling of workload between the GPU and the CPU for the full convolutional approach and heterogeneous ConvMM layer. As shown in Figure 4.7, fully convolution layer is executed homogeneously on the GPGPU, therefore, data is transferred two times between the device and the host memory, to copy the input data to the GPU and to provide the final result. GPU-accelerated padding followed by full convolution is performed on the input data to compute the desired solution. While in case of heterogeneous ConvMM layer data is transferred four times between the GPU and the CPU memories, first copied to the device memory to perform the concurrent padding, then transferred back to the CPU side to perform the sequential transformation, then again to the device memory to perform the GPU-based matrix multiplication and finally to the CPU memory.

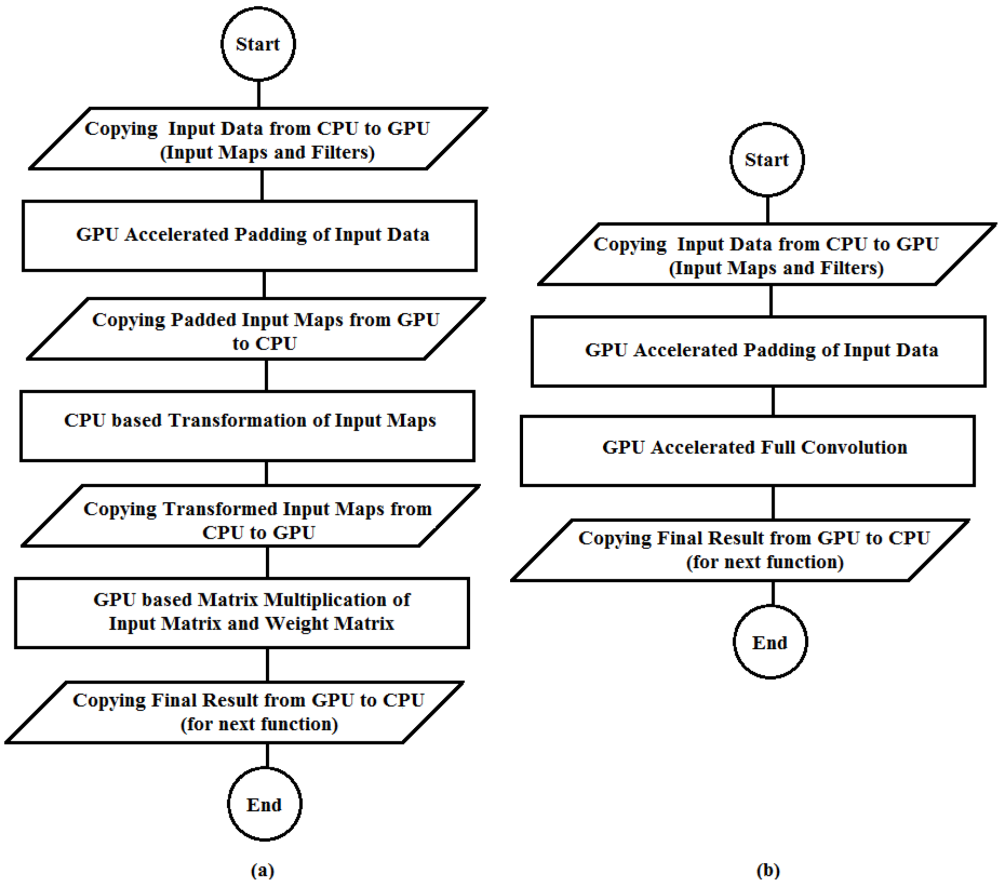

Start
Copying Input Data from CPU to GPU (Input Maps and Filters)
GPU Accelerated Padding of Input Data
Copying Padded Input Maps from GPU to CPU
CPU based Transformation of Input Maps
Copying Transformed Input Maps from CPU to GPU
GPU based Matrix Multiplication of Input Matrix and Weight Matrix
Copying Final Result from GPU to CPU (for next function)
End
(a)
Start
Copying Input Data from CPU to GPU (Input Maps and Filters)
GPU Accelerated Padding of Input Data
GPU Accelerated Full Convolution
Copying Final Result from GPU to CPU (for next function)
End
(b)

Fig. 4.7 (a) Flow of heterogeneous ConvMM layer; (b) Flow of full convolutional layer.

4.5.3 GPU-only ConvMM Layer

Instead of just comparing the execution time of the same kernel on the CPU and the GPU, cost of moving data should also be considered. Extra memory transfers across the PCI-express bus can break the performance of an application. The Number of data transfers should be minimized, even if the GPU version get little or no speedup compared to the CPU version.

Figure 4.7 (a) also depicts that the data is transferred four times between the GPU and the CPU memories to realize the heterogeneous ConvMM layer. Out of four, two transfers are done to perform the CPU-based transformation. These transfers can break the overall performance of the system in case of slower memory bandwidth. If this CPU-based transformation step of heterogeneous ConvMM layer is parallelized and performed using the GPU, so these extra memory transfers can be avoided [64]. Figure 4.8 shows the proposed flow of GPU-only ConvMM layer where both transformation and multiplication steps can be performed using the GPU.

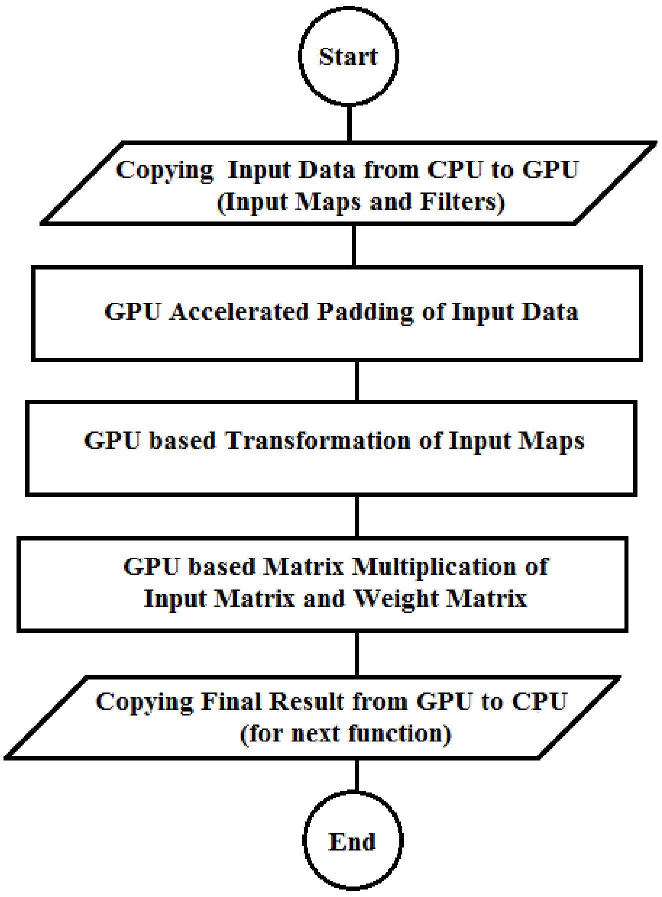

Start
Copying Input Data from CPU to GPU (Input Maps and Filters)
GPU Accelerated Padding of Input Data
GPU based Transformation of Input Maps
GPU based Matrix Multiplication of Input Matrix and Weight Matrix
Copying Final Result from GPU to CPU (for next function)
End

Fig. 4.8 Flow of GPU-only ConvMM layer.

Following formula can be used to transform the multi-dimensional input map into two-dimensional input matrix concurrently.

$$Input\_matrix_{(row,col)} = \sum_{i=1}^{o\_Row} \sum_{j=1}^{o\_Col} \left(\sum_{k=1}^{i\_Dim} \sum_{h=1}^{k\_h} \sum_{w=1}^{k\_w} Input\_map_{(i+h,j+w,k)} \right) \tag{4.3}$$

where $o\_Row = i\_Row - k\_h + 1\ ;$ $o\_Col = i\_Col - k\_w + 1;$

$row = i \times (o\_Col) + j\ ;$ $col = (k \times k\_w \times k\_h + (h \times k\_w) + w);$

In this formula, width and height of the filter are represented by k_w and k_h. i_Row, i_Col and i_Dim are representing the number of rows, columns and dimensions (channel) of the input image. Rows and columns of the output matrix are represented by the o_Row and o_Col.

4.5.4 GPU-Accelerated Max and Average Pooling Layers

Both max and average pooling layers are implemented to realize the different neural architectures on the embedded platform. The CUDA programming language is used to concurrently compute the solution of these sub-sampling approaches. The computational power of GPU is exploited to surpass the sequential version of pooling layers in terms of execution time. The CUDA-based padding function is also implemented and employed in these pooling layers to support the same options and architectures as provided by the Torch framework.

4.5.5 GPU-Accelerated Batch Normalization and Other Layers

All remaining layers (Fully connected, tangent hyperbolic unit, Threshold and ReLu) are also implemented in CUDA and accelerated using the three-dimensional grid of parallel blocks and threads. The fully connected layer is computed by the GPU-based concurrent matrix multiplication followed by a bias offset [65]. Trained weights and biases are used to perform this operation. The batch normalization layer also has trainable parameters (beta, gamma. variance and mean). These parameters are also imported from Torch-based trained models to normalize the feature maps using the CUDA-based version of batch normalization layer.

4.5.6 Neural Network Architectures implemented in CUDA

Three different deep classifiers are implemented using proposed CUDA-based scheme: (ResNet-34, OverFeat network and Alex Krizhevsky's network for CIFAR-10) [7, 23]. Performance of these deep classification networks is evaluated on two different embedded platforms for both convolution approaches. Torch computing framework is used to construct and train these classifiers. CIFAR-10 (Canadian Institute For Advanced Research 10) dataset is used to train the Alex's CIFAR-10 network, while other two networks are trained on ImageNet dataset. These different architectures of different depth are selected to verify the performance of proposed CUDA-based framework for embedded GPUs.

These deep networks have different architecture and components as listed in Table 4.1. All these functions and architectures are required to be replicated for image classification on an embedded GPU. These classifiers are composed of multiple layers and have different pooling and regularization functions. ResNet-34 has two additional layers than other architectures, batch normalization and average pooling. All these layers are implemented in CUDA to replicate the trained classifiers on embedded platforms.

Table 4.1 Neural Architectures used for implementation and evaluation.

Model	**No. of Layers**	**Required Functions**
Alex's CIFAR-10	5	Conv. + tanh + Max Pool
OverFeat	8	Conv. + ReLu + Max Pool
ResNet-34	34	Conv. + Max Pool + Batch Normalization + ReLu + Avg.Pooling

Trained parameters are imported from the Torch software, so can be used by our CUDA-based identical architectures for image classification. Size of the trained classifier (trainable parameters) is also an important factor to consider for successful realization.

As Embedded devices have limited storage capacity, so the sizes of trained models, frameworks (like Torch), computational packages and their operating system can affect the performance of the required network. By using our CUDA-based proposed framework, installation of additional computational packages and source framework can be avoided. The imported trained parameters include weights and biases from the convolutional and fully connected layers or values of mean, variance, gamma and beta for the batch normalization layers. Sizes of these parameters depend on the dimensions of features maps fed to the next layers and sizes of the filters of convolutional layers. The sizes of these parameters are listed in Table 4.2. These imported parameters are of double data type.

Table 4.2 Size of parameters imported form Torch Computing framework.

Model	**Size of File**
	Double
Alex's CIFAR-10	41 MB
OverFeat	1190 MB
ResNet-34	215 MB

4.6 Experiments and Results

Experiments are performed on two different types of embedded platforms. The first platform is Nvidia Jetson TX1 embedded board. This board has quad-core ARM Cortex A57 CPU, Nvidia Maxwell GPU with 256 CUDA cores and 4 GB of shared RAM. It provides 16 GB of embedded MultiMediaCard (eMMC) for storage. The second platform used for the performance evaluation is Nvidia Shield K1 tablet. This tablet is powered by the Nvidia Tegra K1 processor. It has Nvidia Kelper GPU with 192 CUDA cores and ARM Cortex A53 CPU. It has 2 GB of RAM memory.

4.6.1 Performance Evaluation of Proposed Scheme on Jetson TX1 Board

First of all, comparison of both convolutional layers is performed on the Jetson TX1 embedded board. This comparison is done to analyze the performance of both GPU-based traditional convolution and matrix multiplication based approaches. The matrix multiplication based approach (ConvMM) is also implemented using heterogeneous resources and pure CPU-based functions. In GPU- and CPU-only versions of ConvMM layer, both steps of transformation and multiplication are performed using homogeneous GPU or CPU systems while in heterogeneous ConvMM layer, the transformation is performed using the CPU and multiplication is computed using the GPU. All results are listed in Table 4.3.

Figure 4.9 depicts that for the smaller image sizes, traditional convolution approach performs well and outperforms the GPU-only ConvMM layer. However, as the size of image increases, the matrix multiplication based approach outperforms the fully convolution method and gains significant speedup. Thus, it concludes that the

GPU-only ConvMM layer would always show better performance where a greater number of output maps have to be computed, or convolution has to be performed on an input image with larger dimension.

Table 4.3 Comparison of different convolution layers under various computational loads on Jetson TX1 Board, best results are written in bold.

Required Output Maps = 16				
Image Size	**Fully Convolution**	**GPU-only ConvMM**	**Hetero ConvMM (CPU+GPU)**	**ConvMM (CPU-only)**
	Double (Milliseconds)			
CIFAR($32 \times 32 \times 3$)	**1.53**	5.12	8.03	11.17
ImageNet ($224 \times 224 \times 3$)	**33.23**	35.61	89.39	348.67
VGA ($640 \times 480 \times 3$)	275.23	**215.74**	375.02	2048.63
SVGA ($800 \times 600 \times 3$)	459.06	**369.91**	584.13	3212.65
SXGA ($1280 \times 1024 \times 3$)	1074.62	**582.35**	1294.56	8783.52

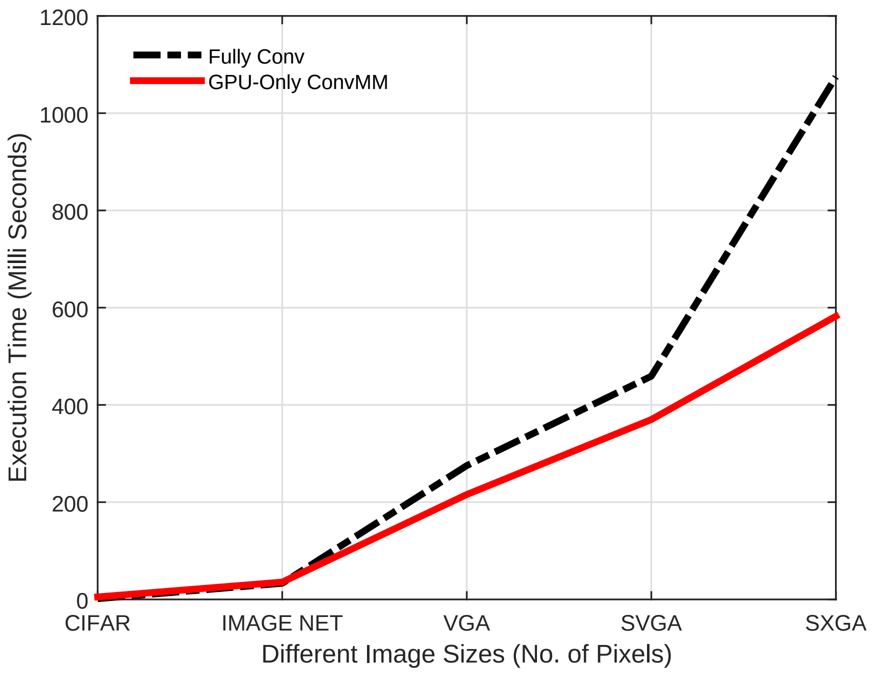


Fig. 4.9 Full convolution vs. GPU-only ConvMM as a function of input image size (Jetson TX1 Board).

The execution time of different versions of ConvMM layer can be visualized in Figure 4.10. Results depict that the heterogeneous and CPU-only ConvMM layers show poor performance and cannot match the computational capability of the GPU-only version. Furthermore, GPU-only ConvMM layer is approximately 2x faster than the heterogeneous approach and 10x faster than the CPU-only ConvMM layer.

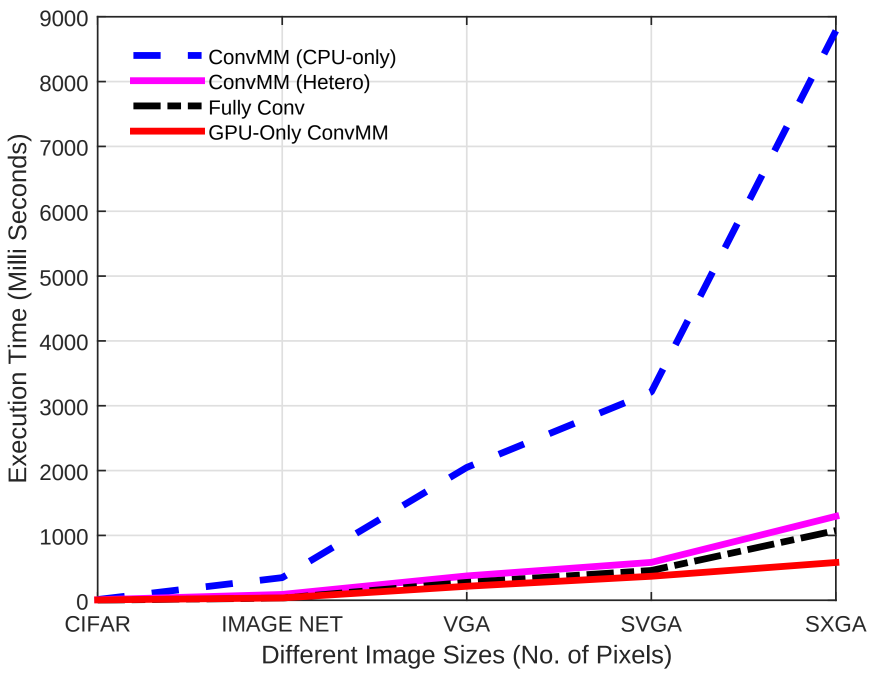


Fig. 4.10 Comparison of different versions of convolution operation as a function of the input image size (Jetson TX1 Board).

After this, the performance comparison of rectifier linear unit (ReLU) is performed for the GPU-based parallelized version and CPU-based sequential version. The sequential version is computed using a single thread of embedded CPU, while the GPU-based version is computed using the two-dimensional and three-dimensional grids, which is an important constraint for the GPU-based implementations. Results of this experiment are listed in Table 4.4.

Table 4.4 Execution time of rectifier linear unit under various computational loads on Jetson TX1 Board, best results are written in bold.

Image Size	**CPU**	**GPU (2D Grid)**	**GPU (3D Grid)**
		Double	
		(Milliseconds)	
CIFAR (32 × 32 × 16)	5.16	3.30	**1.42**
ImageNet (224 × 224× 16)	46.12	25.45	**18.06**
VGA (640 × 480 × 16)	271.72	128.81	**106.83**
SVGA (800 × 600 × 16)	477.23	172.62	**161.43**
SXGA (1280 ×1024 × 16)	2559.91	400.57	**373.52**

Figure 4.11 shows that the GPU-based Rectifier Linear unit is approximately 7x faster than the CPU-based version. Furthermore, there is not a significant performance difference in case of three-dimensional grid-based implementation over the two-dimensional grid-based version. So an embedded GPU supporting only 2D-grids can also be used for the realization of these CUDA-based functions.

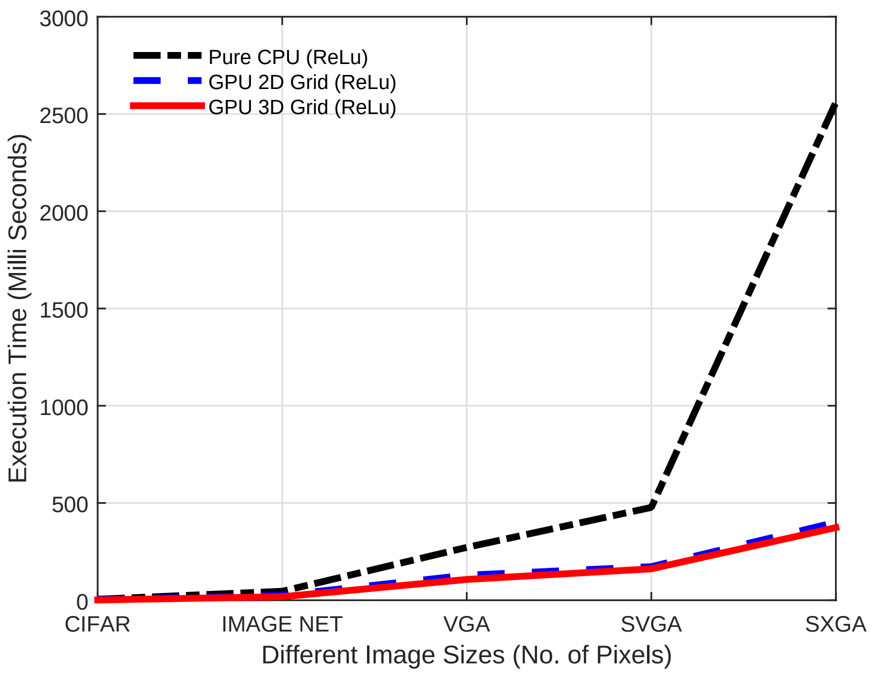


Fig. 4.11 Performance comparison of rectifier linear unit on Jetson TX1 board as a function of input image size.

Table 4.5 and Figure 4.12 show the execution time of max pooling layer. The GPU-based accelerated version is approximately 4x faster than the sequential version.

Table 4.5 Execution time of both versions of pooling layers under various computational loads (Jetson TX1 Board), best results are written in bold.

Image Size	**CPU**	**GPU**
	Double	
	(Milliseconds)	
CIFAR (32 × 32 × 16)	2.33	**2.03**
ImageNet (224 × 224× 16)	53.12	**18.51**
VGA (640 × 480 × 16)	306.53	**93.24**
SVGA (800 × 600 × 16)	469.67	**162.53**
SXGA (1280 ×1024 × 16)	1892.31	**621.82**

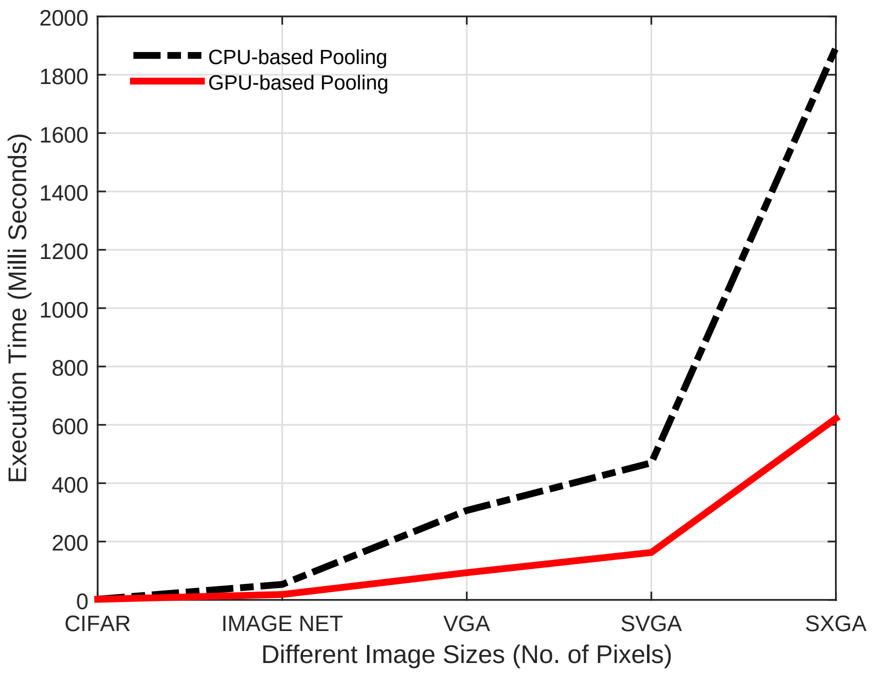


Fig. 4.12 Performance comparison of pooling layers on Jetson TX1 board as a function of input image size.

After evaluating individual functions, three trained deep classifiers (ResNet-34, OverFeat network and Alex Krizhevsky's network for CIFAR-10) are implemented using discussed layers. Performance of these CUDA-based replicated networks can be compared using Table 4.6.

Table 4.6 Classification time of deep models on Jetson TX1 board, best results are written in bold.

Model	Layers	ConvMM (CPU-only)	Fully Conv (GPU)	Hetero ConvMM (CPU+GPU)	GPU-only ConvMM	Speedup over Sequential
		Double (Milliseconds)				
Alex's CIFAR-10	5	7814.25	149.43	129.90	**107.03**	73×
OverFeat	8	254,285.12	3250.57	2492.94	**1924.41**	132×
ResNet-34	34	190,054.39	2838.01	2361.18	**1217.33**	156×

Results show that the GPU-accelerated deep classifiers are hundred times faster than the CPU-based sequential versions. Results also point out that even the heterogeneous ConvMM layer-based models are faster than the traditional convolution based classifiers and can further be accelerated with faster CPU. In case of CUDA-based realizations, amount and number of memory transfers are the lower bounds on the expected gain, as results show that the GPU-only ConvMM layers-based networks are significantly faster than the heterogeneous versions because of eliminated data transfers required by sequential transformation.

It can be noted that the trainable parameters imported from the Torch computing framework are of double data type and discussed results are of same data type due to the double precision arithmetic operations performed on these parameters. Performance of these deep classifiers can be further improved using single precision data storage and arithmetic. This conversion does not affect the accuracy of the trained networks. Some research works have already proved that the high accuracy can be achieved with extremely low precision neural networks as the fixed-point representation or even only 2 bits per weight are enough to improve the performance and efficiency of the convolutional classifiers without impacting their accuracy [66–69]. A research group from IBM experimented with rounding schemes, their results showed that the neural classifiers can be trained using low precision (16-bit wide) fixed-point representation and can achieve the same performance as that obtained using 32-bit floating point representation [68]. Another research group from Intel Labs experimented with Residual Networks and by training a ResNet-34 network on the ImageNet dataset with 2-bit weights achieved similar accuracy (90.37 % Top-5) than the full precision version (91.26 % Top-5) [69]. The deep neural classifiers are numerically robust, so that small differences in precision do not affect the classification accuracy. The differences introduced by low-precision representations are well within the tolerances a neural network has learned to deal with.

Execution time of double- and single-precision ConvMM layers can be compared using Table 4.7 and Figure 4.13. Results show that the single-precision ConvMM layer is approximately 2× faster than the double-precision version.

Table 4.7 Comparison of double- and single-precision ConvMM layers on Jetson TX1 board, best results are written in bold.

Image Size	**GPU-only ConvMM**	**GPU-only ConvMM**
	Double	**Single**
	(Milliseconds)	
CIFAR (32 × 32 × 16)	5.12	**3.20**
ImageNet (224 × 224× 16)	35.61	**21.63**
VGA (640 × 480 × 16)	215.74	**119.17**
SVGA (800 × 600 × 16)	369.91	**162.80**

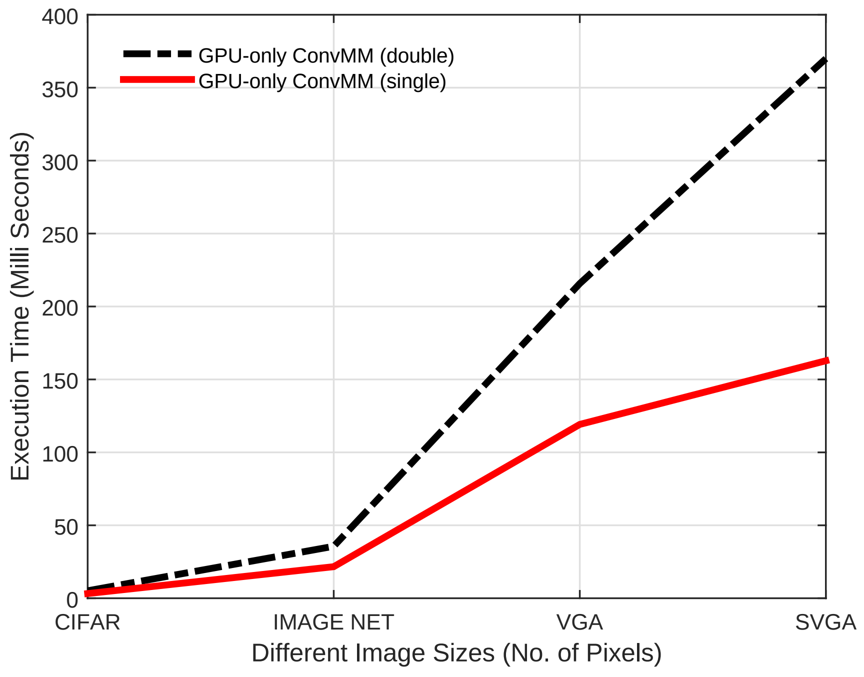


Fig. 4.13 Performance comparison of double- and single-precision ConvMM layers on Jetson TX1 Board as a function of input image size.

Table 4.8 further validates that the single-precision models are significantly faster than the original versions. This conversion is also profitable in terms of storage requirement that is an important constraint for the embedded devices.

Table 4.8 Classification time of double- and single-precision deep models on Jetson TX1 Board, best results are written in bold.

Model	Layers	GPU-only ConvMM Double (Milliseconds)	GPU-only ConvMM Single (Milliseconds)
Alex's CIFAR-10	5	107.03	**76.36**
OverFeat	8	1924.41	**1212.74**
ResNet-34	34	1217.33	**887.53**

4.6.2 Performance Evaluation of Proposed Scheme on Mobile Shield K1 Tablet

The second platform used for measuring the performance of our CUDA-based proposed scheme is Nvidia Shield K1 tablet. This embedded device is powered by Tegra K1 GPU having 192 CUDA cores. Same experiments are performed to verify the previous results and measure the performance of realized deep classifiers.

First of all, the performance comparison of different versions of convolution layers is performed on this second embedded GPU. All parameters are converted to single-precision. This is done to reduce the arithmetical complexity and storage requirement. The execution time of all convolutional layers can be compared using Table 4.9 and Figure 4.14. Results show that the GPU-only ConvMM layer is significantly faster than all other layers. It can also be noted that for smaller workload, all versions including the GPU-only ConvMM layer are outperformed by the fully convolution approach, while this traditional approach is much slower for larger workloads that even heterogeneous ConvMM layer can outperform it.

Figure 4.14 also shows that the GPU-only ConvMM layer is approximately 2× faster than the heterogeneous ConvMM layer. This speedup in execution time is due to the elimination of extra memory transfers required by the sequential transformation step of Heterogeneous ConvMM layer.

Table 4.9 Comparison of different convolution layers under various computational loads on Nvidia Shield K1 tablet, best results are written in bold.

Required Output Maps = 16				
Image Size	**Fully Convolution**	**GPU-only ConvMM**	**Hetero ConvMM (CPU+GPU)**	**ConvMM (CPU-only)**
	Single			
	(Milliseconds)			
CIFAR(32 × 32 × 3)	**2.31**	3.95	4.20	37.21
ImageNet (224 × 224 × 3)	28.23	**28.01**	62.27	1570.98
VGA (640 × 480 × 3)	170.94	**154.32**	327.53	10,055.24
SVGA (800 × 600 × 3)	276.16	**231.95**	519.08	15,852.86
SXGA (1280 × 1024 × 3)	1960.57	**495.19**	1040.48	44,969.71

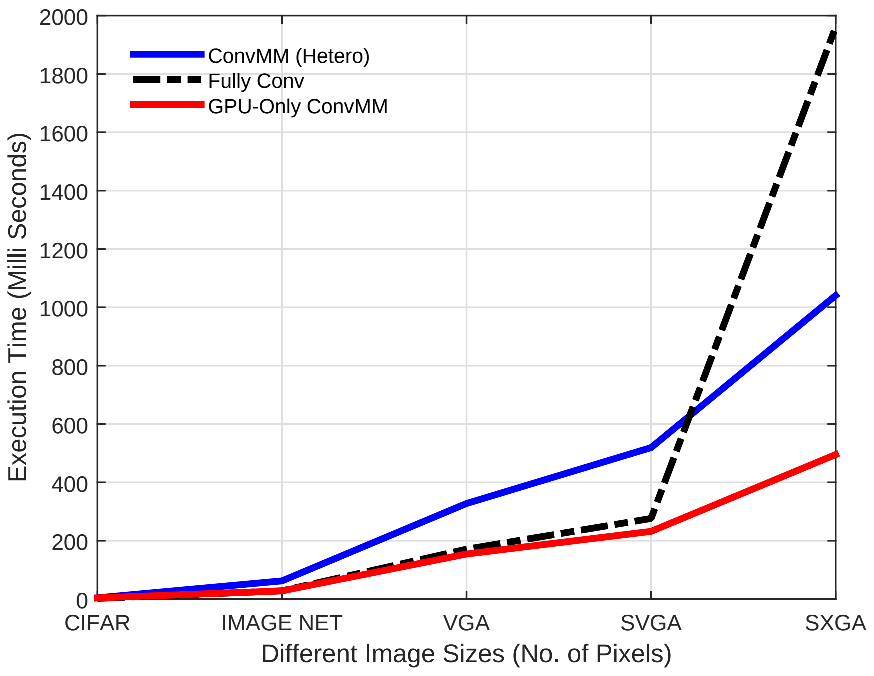


Fig. 4.14 Comparison of different versions of convolution operation as a function of input image size (Nvidia Shield Tablet).

After this, the performance of sequential and CUDA-based Rectifier Linear layer is measured and results are listed in Table 4.10. As mentioned previously, the sequential version is computed using a single thread of embedded CPU while the three-dimensional grid of blocks is exploited to parallelize the GPU-based version. Figure 4.15 shows that the GPU-based version is approximately 13× faster than the CPU-based version.

Table 4.10 Execution time of rectifier linear unit under various computational loads on Nvidia Shield K1 tablet, best results are written in bold.

Image Size	**CPU**	**GPU**
	Single	
	(Milliseconds)	
CIFAR (32 × 32 × 16)	5.75	**1.47**
ImageNet (224 × 224× 16)	39.73	**13.02**
VGA (640 × 480 × 16)	294.84	**36.09**
SVGA (800 × 600 × 16)	496.16	**97.14**
SXGA (1280 ×1024 × 16)	4059.09	**313.85**

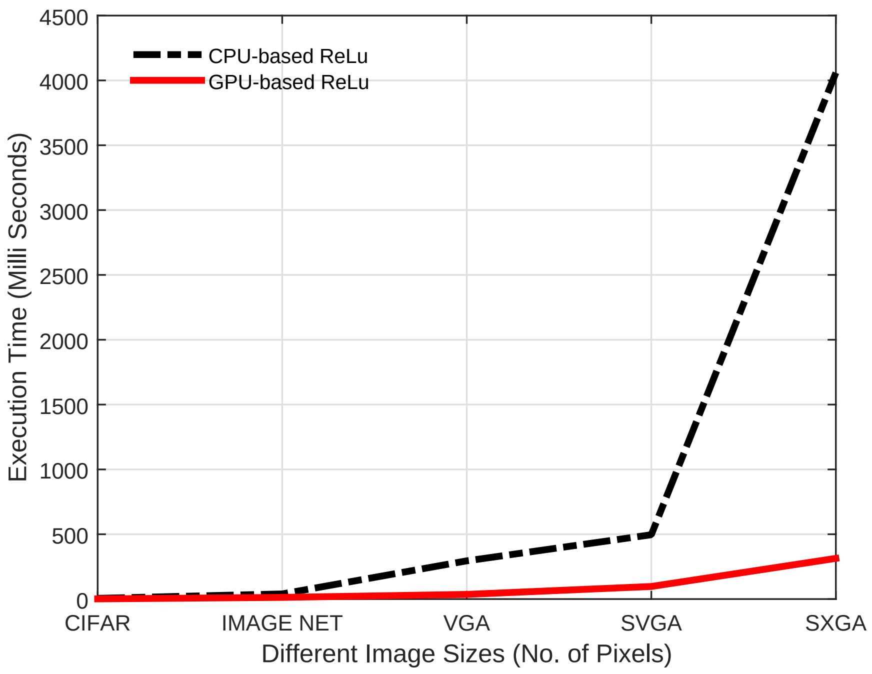


Fig. 4.15 Performance comparison of rectifier linear unit on Nvidia Shield K1 tablet as a function of input image size.

Table 4.11 and Figure 4.16 show the execution time of max pooling layer. The GPU-accelerated max pooling layer is approximately 4x faster than the CPU-based version.

Table 4.11 Execution time of both versions of pooling layers under various computational loads (Nvidia Shield K1 tablet), best results are written in bold.

Image Size	**CPU**	**GPU**
	Single	
	(Milliseconds)	
CIFAR (32 × 32 × 16)	1.83	**1.10**
ImageNet (224 × 224× 16)	45.72	**10.57**
VGA (640 × 480 × 16)	275.43	**68.58**
SVGA (800 × 600 × 16)	413.85	**101.83**
SXGA (1280 ×1024 × 16)	1231.20	**329.03**

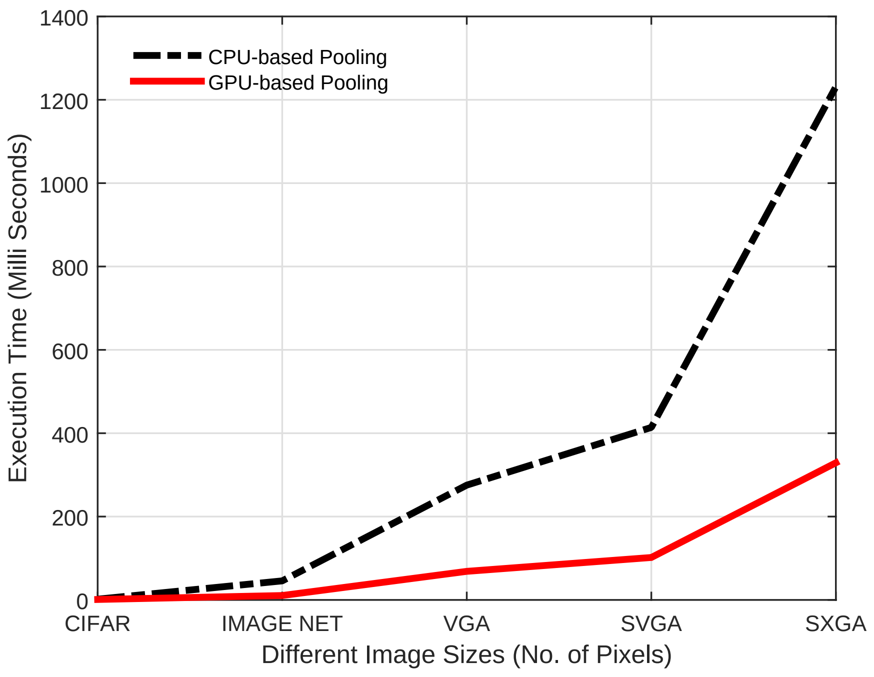


Fig. 4.16 Performance comparison of pooling layers on Nvidia Shield K1 tablet as a function of the input image size.

Table 4.12 tabulates the performance of the different version of convolutional layers over the same deep classifiers. Results show that the GPU-only ConvMM layer boosts the performance of deep classifiers and outperforms the other versions.

The GPU-only ConvMM layer based deep classifiers are hundreds of times faster than the CPU-based sequential versions.

Table 4.12 Classification time of deep models on Nvidia Shield K1 tablet, best results are written in bold.

Model	Layers	ConvMM (CPU-only)	Fully Conv (GPU)	Hetero ConvMM (CPU+GPU)	GPU-only ConvMM	Speedup over Sequential
		Single				
		(Milliseconds)				
Alex's CIFAR-10	5	10,449.21	584.75	419.67	**144.23**	73×
OverFeat	8	440,631.27	12,582.54	5792.54	**3899.43**	113×
ResNet-34	34	252,955.27	11,481.74	3319.34	**2545.83**	99×

Furthermore, a hand-held mobile device is powered by a rechargeable battery which is limited in capacity due to its smaller size, while the energy consumption goes up with the usage of computational resources and can reduce the life of a battery. Therefore, the power consumption is also an important constraint to consider when performing image classification using a mobile device. However, GPUs can compute the solution of a given task in less time than a CPU-based version by utilizing the computational resources for a shorter period; it can significantly reduce the energy consumption. The Nvidia Shield K1 tablet is powered by a battery of 19.75 Wh and 5192 mAh. A software profiler is used to measure the energy consumption of realized deep classifiers. Table 4.13 lists the energy consumption per image frame by the android application through hardware.

Table 4.13 Energy consumed by different deep classifier on Nvidia Shield K1 tablet (joule), best results are written in bold.

Model	Sequential ConvMM	Fully Convolution	GPU-only ConvMM	Improvements over Sequential
Alex's CIFAR-10	16.0	0.430	**0.373**	43×
OverFeat	850.8	13.2	**2.13**	399×
ResNet-34	480.0	2.5	**1.2**	400×

Results show that the CPU-based sequential versions consume more energy than the GPU-based concurrent versions because of longer execution times. Namely, the ConvMM-based classifiers are consuming hundred of times lesser energy than the sequential versions and are more energy efficient than the fully convolution based networks. The proposed framework can be further accelerated and optimized by employing an efficient memory transfer scheme and exploiting the hardware-dependent resources of an embedded GPU. These techniques and proposed optimized scheme are presented in the next chapter.

Chapter 5

Optimization of Deep Neural Classifiers for Embedded GPUs

A developed framework needs to be highly optimized to run the deep classifiers on the embedded devices. It is only possible by overcoming the limiting constraints and improving the overall results including energy efficiency, storage requirement and inference performance. This chapter discusses the optimization of our proposed framework where a set of techniques are employed and hardware capabilities of embedded GPUs are exploited to improve the performance of deep neural networks, making them applicable to embedded applications[1]. The flow of our CUDA-based framework is optimized using unified memory scheme and hardware-dependent matrix multiplication approach. The performance of proposed optimized networks is measured, and results are compared with Torch computing framework.

5.1 Introduction

In a CPU-GPU environment, different important factors and parameters (like amount and number of data transfers between the host and the device, etc.) can make or break the performance of an application. The performance of a GPU-based framework can be improved by following few general guidelines. First, the number of data transfers

[1]Part of the work described in this chapter has been previously published in **Applied Sciences** Journal as "Rizvi, S.T.H.; Cabodi, G.; Francini, G. Optimized Deep Neural Networks for Real-Time Object Classification on Embedded GPUs. Appl. Sci. 2017, 7, 826.

between the CPU and the GPU should be minimized, even if that means running kernels on the GPU that get little or no speedup compared to running them on the CPU. Furthermore, higher bandwidth can be achieved between host and device by using pinned or page-locked memory. By loading the required parameters or constants at the start of an application, the transfer overheads can also be avoided.

As discussed in **Chapter 4**, there are different approaches to compute the convolution operation and matrix multiplication based solution is one of them. Our proposed framework uses this matrix multiplication based convolution approach where the transformed input and filter matrices can be multiplied efficiently using the concurrent resources of a GPU. There are various software libraries that also provide high-performance matrix multiplication subroutines; however, it still remains a challenge to utilize these computational packages having the near-optimal performance for the realization of deep classifiers on the embedded platforms.

In this work, an optimized and accelerated scheme for image classification is proposed for the embedded platforms [70]. The results show that the proposed scheme significantly improve the inference performance, storage requirements and energy efficiency. Only CUDA-based functions and libraries are used to optimize the framework presented in the previous chapter. Unified memory transfer scheme is employed to optimize the flow of our GPU-only ConvMM layer and all other required functions (Pooling, batch normalization, etc.). By using this data-transfer scheme, factors like double allocation of parameters, memory access latency and the explicit data movements are eliminated. Additionally, the proposed framework is further accelerated and optimized using the hardware-dependent selection of resources and matrix-multiplication operation for the convolution layers of deep architectures.

5.2 Optimized Data transfer schemes

In this section, various data transfer schemes are discussed and unified memory scheme is proposed to optimize the architectures of all CUDA-based layers presented in **Chapter 4**.

5.2.1 Data Allocations using Pinned Memory

For CUDA-based neural classifiers proposed in the previous chapter, the input data to be processed (filters, input images etc) must be copied from the host (CPU) memory to the device (GPU) memory and the output data (final results) must be retrieved from the GPU memory to the CPU memory. These memory-transfers may be required several times depending on the architecture of a neural network and can affect the performance of the application. Host allocations are pageable by default, and the GPU cannot get data directly from the pageable memory. Therefore whenever a data transfer is requested, CUDA driver must copy the data from the pageable memory to a temporary host array in the pinned memory and then data can be finally transferred to the Device memory via PCIe (Peripheral Component Interconnect-Express) bus as shown in Figure 5.1. The pinned memory acts as a staging area between the host and device memories.

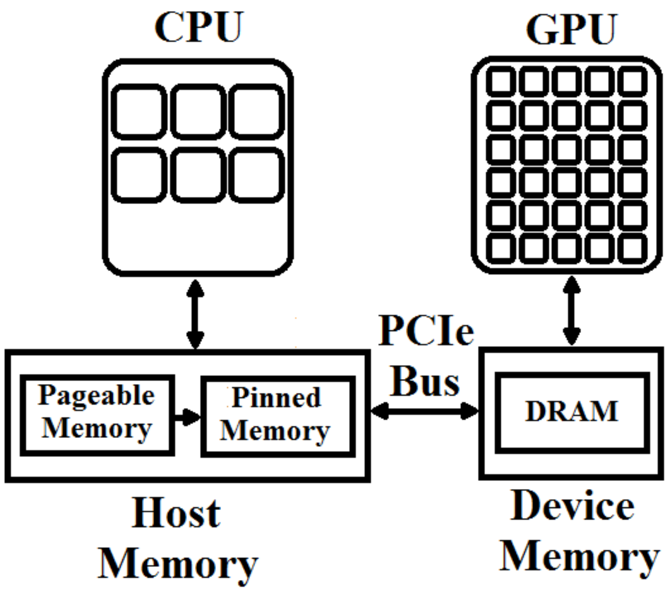


Fig. 5.1 Pinned and pageable memories on the host (CPU) side.

The overhead caused by this extra memory transfer between the host memories can be avoided by allocating the input data in the pinned memory. CUDA provides the feature to directly allocate the host data into the pinned memory as shown in Figure 5.2.

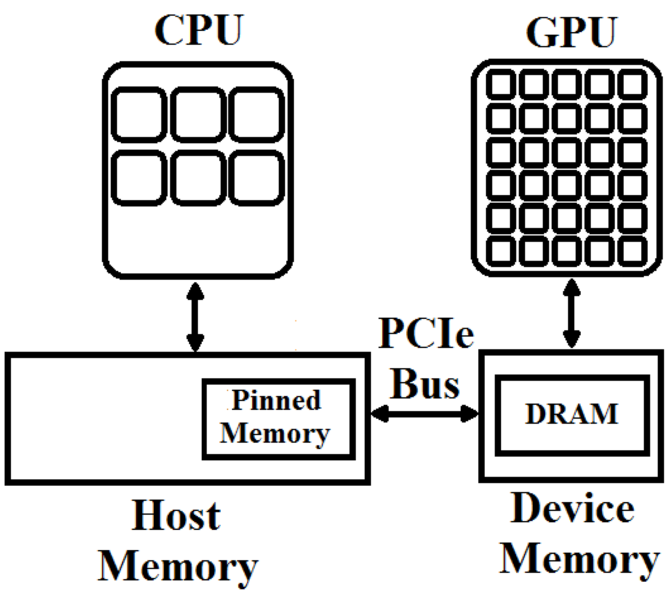


Fig. 5.2 Data transfer directly from the Pinned memory.

It is a common practice to utilize the pinned memory for acceleration of GPU-based desktop and server frameworks [71]. However, some constraints (like the size of input data, output data, and trainable parameters) need to be considered before using pinned memory on an embedded platform. Pinned memory should not be over-allocated because it can break the performance of the used system by reducing the amount of physical memory (Random Access Memory) to its operating system and other programs. Unlike pageable memory, pinned memory cannot be paged out by the Operating system. Therefore, usage of pinned memory on embedded platforms can affect the performance of neural networks because embedded devices have limited storage capacity.

5.2.2 Proposed flow using Unified Memory-based Allocations

In a typical desktop workstation or GPU server, host (CPU) and device (GPU) memories are physically separated by the PCIe bus as shown in Figure 5.3. Data must be allocated in both memories and explicitly transferred between them. As discussed earlier, these data transfers are very frequent, and speed of these transactions is dependent on the PCIe bus. These data transfers add memory overhead and increase the structural complexity of an algorithm. Figure 4.8 shows the flow of GPU-only

ConvMM layer and it can be noted that at least two data transfers are required for every layer, to copy the input data from the CPU to the GPU for the required computations, and to retrieve the output data from the GPU memory for the next layer or operation.

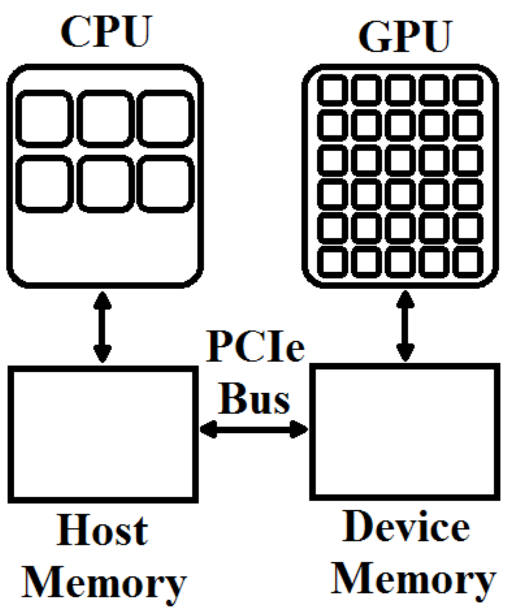


Fig. 5.3 Physically separate Host and device memories.

Unified memory is introduced from the CUDA version 6.0, which creates a pool of shared memory between the host (CPU) and device (GPU) as shown in Figure 5.4. This shared memory is accessible to both host (CPU) and device (GPU) using a single pointer so that the same data can be used by a CPU function as well as a GPU kernel according to the requirement of the program. By migrating the data to a shared pool, unified memory offers the performance of local data on the GPU and can enhance the performance of an algorithm [72]. However, shared data needs to be synchronized after execution of every GPU kernel.

Most of the embedded devices have physically unified memories, so this includes both CPU as well as GPU memory requirements. Before CUDA v6.0, two copies of input data were required in both CPU and GPU memories. However, by using unified memory scheme, only a single copy of data elements is needed that can be allocated to a shared pool of memory. This single allocation of parameters can be beneficial for the realization of deep neural networks on low storage embedded platforms.

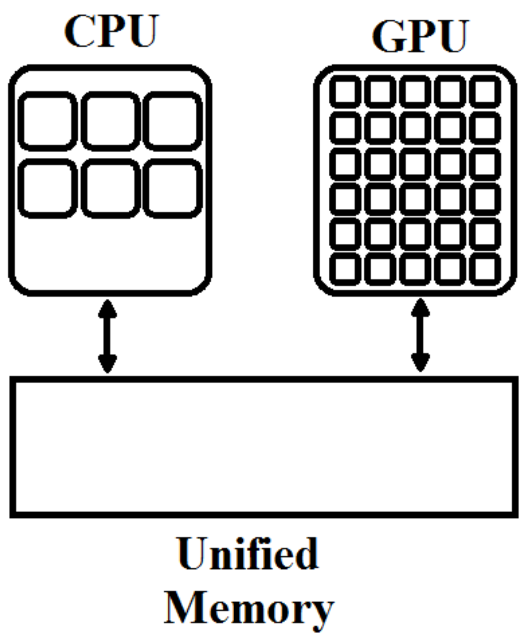


Fig. 5.4 Shared pool of Unified memory.

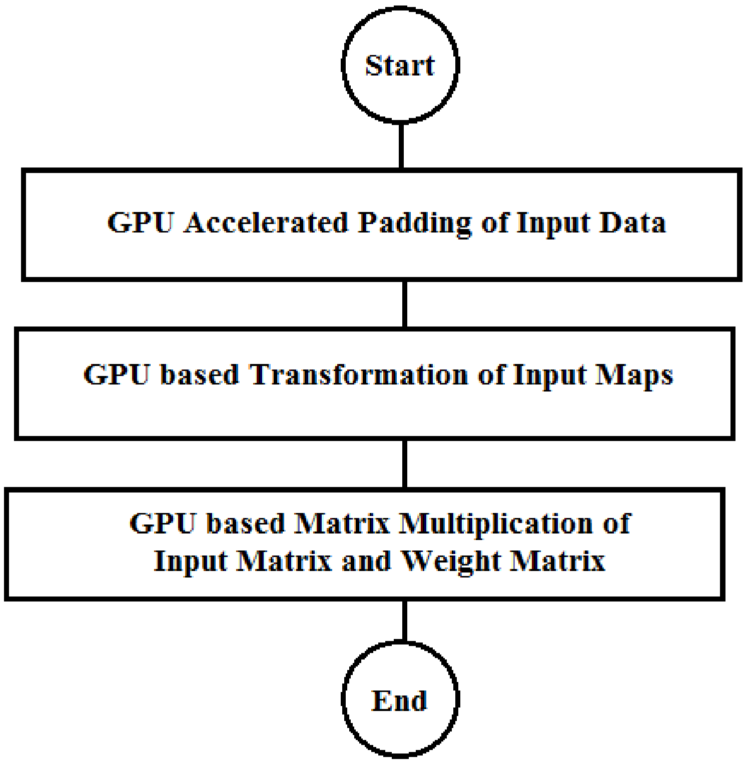

Start
GPU Accelerated Padding of Input Data
GPU based Transformation of Input Maps
GPU based Matrix Multiplication of Input Matrix and Weight Matrix
End

Fig. 5.5 The flow of GPU-only Unified ConvMM Layer.

This unified memory scheme is employed to optimize the flow of our CUDA-based neural architectures. Using unified memory, the flow of all required layers and functions is simplified and optimized. Figure 5.5 shows the flow of GPU-only Unified ConvMM layer. All trainable parameters, inputs and outputs are allocated in the unified memory space to achieve this simplified flow. By using this scheme, the output of one layer can be directly fed to the next layer without any explicit data transfer.

5.3 Accelerated Matrix Multiplication for GPU-only Unified ConvMM layer

The proposed framework can be further optimized by improving the performance of matrix multiplication operation required by the convolutional layer. In this section, two different approaches to accelerate the matrix multiplication operation are presented for real-time image classification. Our proposed framework selects one of the accelerated matrix multiplication (AMM) scheme depending on the compute capability of the used embedded device as shown in Figure 5.6. Compute capability (C.C) defines the available features and restrictions of a GPU hardware. These accelerated matrix multiplication approaches are used to exploit the hardware-dependent resources of the embedded platforms.

5.3.1 cuBLAS-Accelerated Matrix Multiplication Convolution

The CUDA toolkit includes a set of high-performance libraries which provide standard mathematical functions. CUDA Basic Linear Algebra Subroutines (cuBLAS) is one of them. The cuBLAS library provides the various GPU-accelerated subroutines and shows tremendous speedup over other available libraries [73, 74]. It also provides a wide range of General Matrix-Matrix Multiply (GEMM) subroutines that support different type of operands (double-precision, single-precision, half-precision, etc) that can be useful to speed-up different algorithms [75, 76]. The performance of convolution operation can also be improved using cuBLAS-Accelerated Matrix Multiplication (CAMM), but some important aspects need to be taken into consideration before using this library.

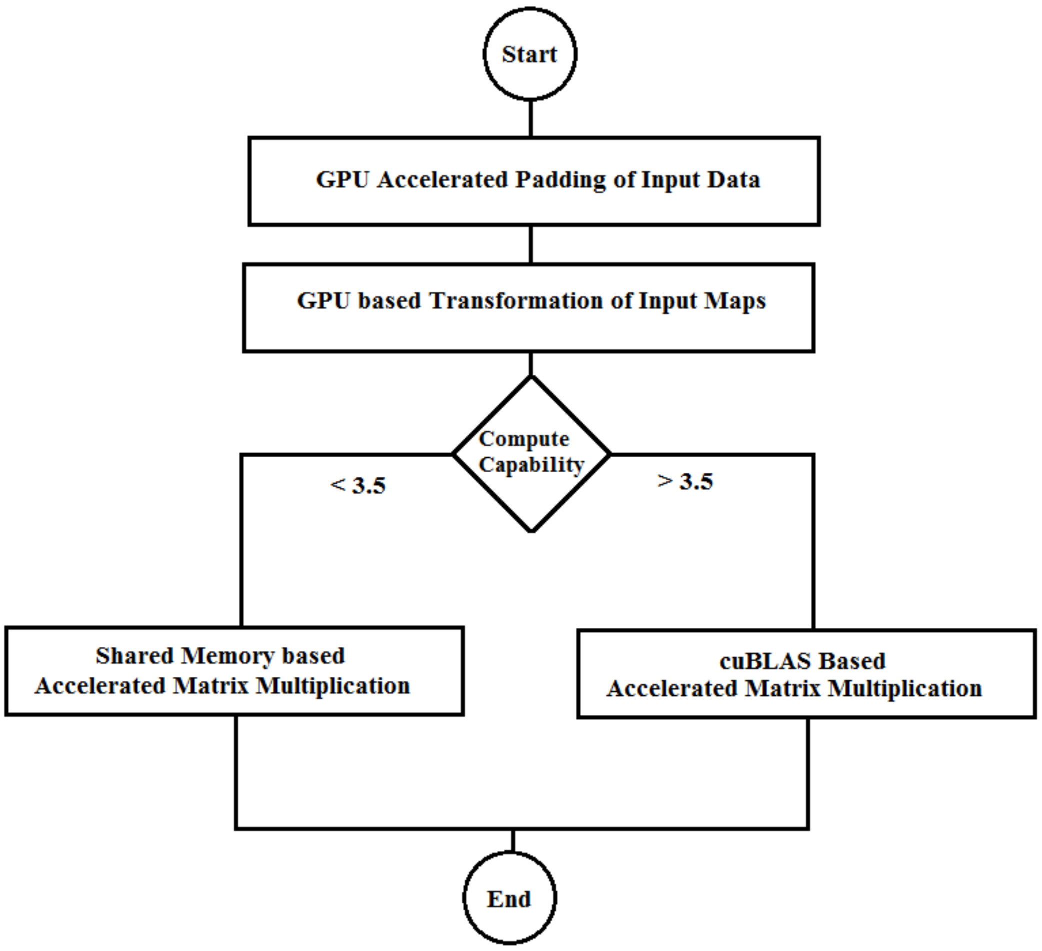

Start
GPU Accelerated Padding of Input Data
GPU based Transformation of Input Maps
Compute Capability
< 3.5
> 3.5
Shared Memory based Accelerated Matrix Multiplication
cuBLAS Based Accelerated Matrix Multiplication
End

Fig. 5.6 Flow of the accelerated matrix multiplication based ConvMM layer with unified memory.

The first constraint is the compute capability (C.C) of the used GPU platform, as the computational power of cuBLAS subroutines cannot be exploited on a GPU hardware having compute capability less than 3.5.

The storage layout of the cuBLAS library is the next important constraint to take into consideration. The data storage layout of commonly used C/C++ languages is row-major order while the matrix multiplication subroutines of cuBLAS library follow the column-major order. So, a data transformation step is essential to covert the data from row-major layout to column-major order for using every cuBLAS subroutine. This transformation between different layouts can reduce the performance of an algorithm instead of improving it.

In the proposed framework, the GEMM subroutine of the cuBLAS library is employed to accelerate the matrix multiplication of convolutional layer (for GPU platforms having C.C greater than 3.5). A comparison of different GEMM subroutines (single- and half-precision) is performed to analyze the performance gain. This cuBLAS-Accelerated Matrix Multiplication (CAMM) is employed in the proposed flow of neural architectures.

The issue of storage layout is resolved for the required cuBLAS subroutines. Since the matrix multiplication routines follow the column-major layout and always suppose that the matrices would be structured and passed in the same format, so the input and output matrices are transformed inherently before forwarding to the multiplication subroutine and after retrieving the resultant matrix. This does not cause a buffer overrun and efficaciously transposes the data without actually moving the matrices around in the memory. Thus, the issue of storage layout is resolved by passing the input matrices in the reverse order (To compute C= A x B, matrices are passed as $C^{'} = B^{'} \times A^{'}$), and as the resultant matrix would always be transposed implicitly by the cuBLAS library before passing to main application, the resultant matrix ($(C^{'})^{'}$ =C) is achieved in row-major layout without any extra transformation.

5.3.2 Shared Memory-based Matrix Multiplication Convolution

As discussed earlier, CUDA devices have different types of memory spaces which offer various performance characteristics. The current version of GPU-only ConvMM layer shown in Figure 4.8 is using the global memory which is much slower than the shared memory space. In this section, shared memory based matrix multiplication (SAMM) is proposed to accelerate the GPU-only Unified convolutional layer (for devices having compute capability less than 3.5). By dividing the global data into tiles or blocks, these subsets can be copied into shared memory to reduce the global access latency and achieve the memory level parallelism.

In naive matrix multiplication approach, many redundant global memory accesses can be reduced using shared memory space as shown in Figure 5.7. Every element of matrix A and B is required to be fetched N times from the global memory in naive approach. It can be noted that the same row and column elements of input matrices A and B are needed to compute every row and column element of output matrix (C) respectively.

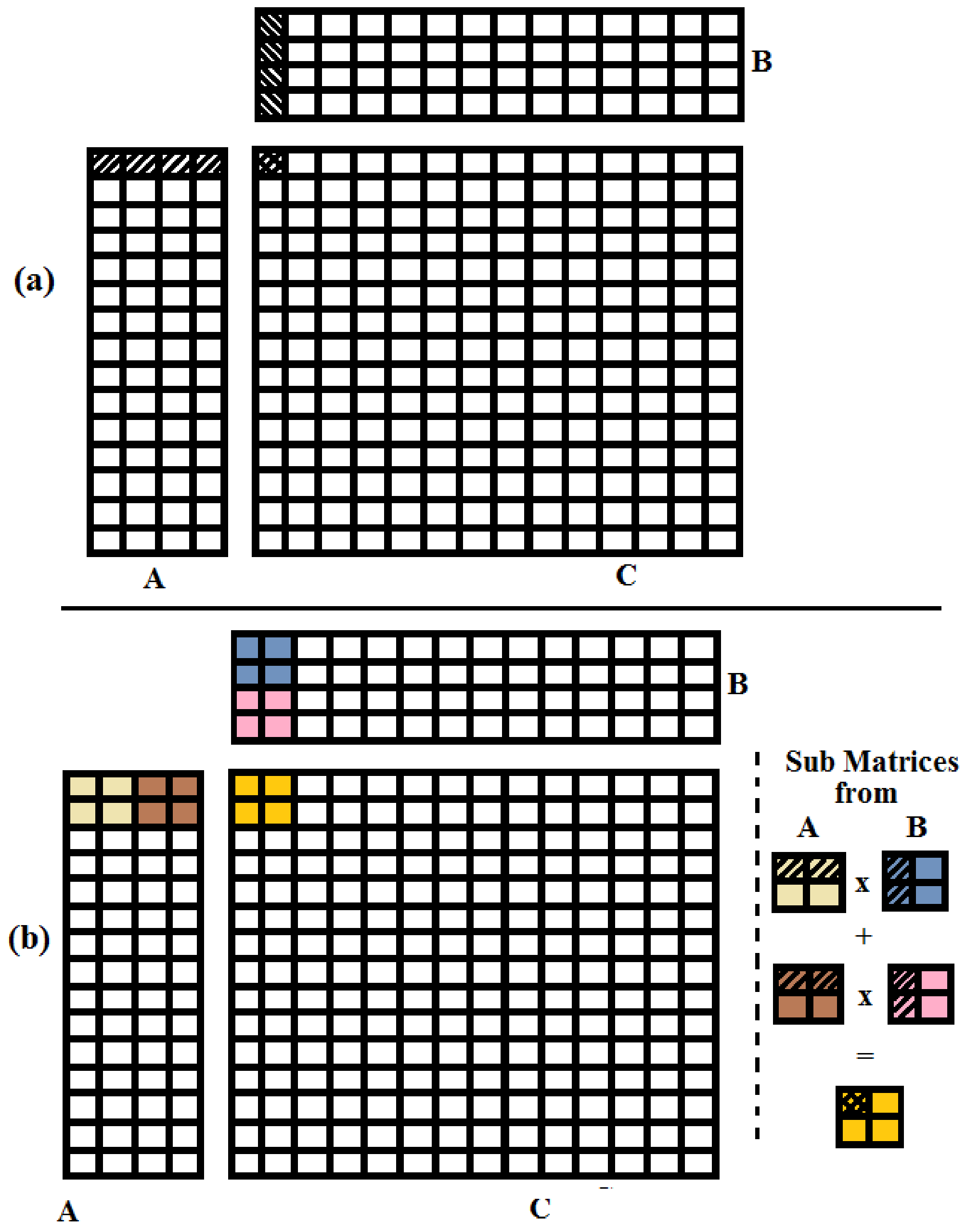


Fig. 5.7 (a) Naive matrix multiplication approach; (b) Shared memory-based matrix multiplication approach.

In shared memory-based proposed scheme, input matrices (A and B) are divided into the sub-matrices (tiles) to assign to the thread blocks, and then these tiles are multiplied independently as shown in Figure 5.7(b). Finally, the elements of the resultant matrix (C) are computed by summing up the results of these multiplications.

Data residing in shared memory can be accessed faster in a concurrent manner as compared to global memory; therefore, shared memory-based matrix multiplication is profitable to avoid the memory latency by reusing the shared data. In addition, the selection of tile size is an important performance factor for this shared memory-based approach which is kept variable in the proposed flow.

5.4 GPU architecture based Exploitation

Recent GPU architectures support the half-precision floating point (FP16) data storage and arithmetic operations that can be useful to increase the training and inference performance of a deep classifier [38, 77, 78]. The main advantage of using half-precision data format over 32-bits single-precision format is that it requires half the storage and bandwidth at the expense of no or marginal loss in accuracy. The composition of different floating point representations can be visualized using Figure 5.8.

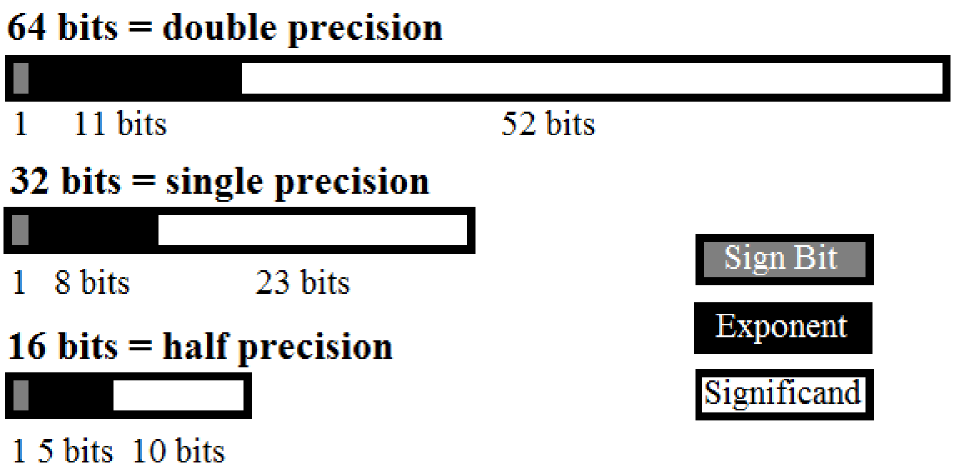


Fig. 5.8 Format of floating point representation (IEEE754).

Nvidia recently introduced support for the FP16 storage and arithmetic into their Pascal GPUs [79]. With the introduction of these architectures, Nvidia is expanding the set of libraries and tools available for mixed-precision computing. Embedded boards like Jetson TX1 and TX2 also support FP16 arithmetic and storage. By converting the complete proposed framework and variables (input, output and trainable parameters) to half-precision format, arithmetic complexity and storage requirements are further reduced for the embedded platforms having Pascal GPUs.

5.5 Experiments and Results

In this work, only CUDA-based function and libraries are used to optimize and accelerate the deep neural classifiers discussed in the previous chapter. All required layers and functions to construct the neural classifiers are optimized using Unified memory scheme. Same three classification networks (ResNet-34, OverFeat model and Alex Krizhevsky's network for CIFAR-10) are used to evaluate the performance of proposed flow. Experiments are performed on Nvidia Jetson TX1 embedded board and Nvidia Shield K1 tablet. Compute capability of Jetson TX1 embedded board and Nvidia Tegra K1 GPUs are 5.3 and 3.2 respectively.

5.5.1 Performance Evaluation on Jetson TX1 embedded board (For GPUs having C.C. > 3.5)

First of all, the performance of GPU-only Unified ConvMM layer is measured on Jetson TX1 embedded board, and a comparison is performed with the previous versions of ConvMM layers (presented in Chapter 4). Table 5.1 lists the execution speeds of different convolutional layers.

Table 5.1 Comparison of different versions of ConvMM layers under various computational loads on Jetson TX1 Board, best results are written in bold.

Required Output Maps = 16				
Image Size	**ConvMM (CPU-only)**	**Hetero ConvMM (CPU+GPU)**	**GPU-only ConvMM**	**GPU-only Unified ConvMM**
	Double		**Single**	
	(Milliseconds)			
CIFAR($32 \times 32 \times 3$)	11.17	8.03	3.20	**2.53**
ImageNet ($224 \times 224 \times 3$)	348.67	89.39	21.63	**9.67**
VGA ($640 \times 480 \times 3$)	2048.63	375.02	119.17	**38.76**
SVGA ($800 \times 600 \times 3$)	3212.65	584.13	169.80	**59.38**
SXGA ($1280 \times 1024 \times 3$)	8783.52	1294.56	582.35	**112.26**

Results show that the there is a significant gain in performance when the flow of GPU-only ConvMM layer is optimized by combining it with unified memory scheme. Figure 5.9 shows that this optimized ConvMM layer is 5× faster than the GPU-only version and is approximately 12× faster than the heterogeneous ConvMM layer.

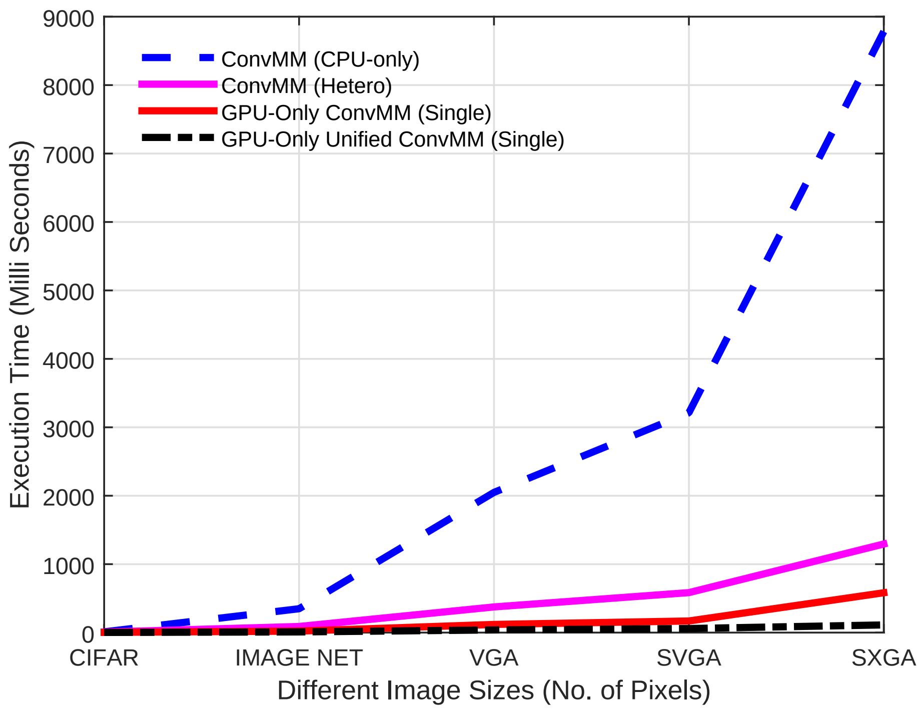


Fig. 5.9 Performance Comparison of GPU-only Unified ConvMM layer and other versions of Convolutional layer on Jetson TX1 Board as a function of input image size.

Table 5.2 presents the performance comparison of this GPU-only Unified memory based classifiers with previously discussed versions. Results validate that a significant gain in inference performance is achieved by using proposed memory optimization scheme.

Table 5.2 Performance comparison of different ConvMM layers based deep models (Jetson TX1 embedded board), best results are written in bold.

Model	**Layers**	**ConvMM (CPU-only)**	**Hetero ConvMM (CPU+GPU)**	**GPU-only ConvMM**	**GPU-only Unified ConvMM**
		Double		**Single**	
		(Milliseconds)			
Alex's CIFAR-10	5	7814.25	129.90	76.36	**41.56**
OverFeat	8	254,285.12	2492.94	1212.74	**512.40**
ResNet-34	34	190,054.39	2361.18	887.53	**652.28**

After this memory-optimized flow, the cuBLAS library is used to further accelerate the proposed scheme. cuBLAS-accelerated matrix multiplication (CAMM) is employed to compute the matrix multiplication operation required by the convolutional layer. As the compute capability (C.C) of TX1 embedded board is higher than 3.5, the cuBLAS library can be used to exploit the computational power of its embedded GPU.

As mentioned earlier, there are various General Matrix-Matrix Multiply (GEMM) subroutines in the cuBLAS library to support different data types and representations. In this work, single-precision (SGEMM) routine is employed in GPU-only ConvMM layer because the complete framework and related variables(input maps, output results, and trainable parameters) are converted to and now in single-precision format to reduce the memory bandwidth and storage requirements. Table 5.3 compares the inference performance of this cuBLAS-based ConvCAMM layer over same deep neural architectures. It can be noted that the ConvCAMM layer is approximately 2$\times$ faster than the GPU-only ConvMM layer.

Table 5.3 Performance comparison of ConvCAMM layers on the deep architectures (Jetson TX1 embedded board), best results are written in bold.

Model	**Layers**	**GPU-only ConvMM**	**GPU-only Unified ConvMM**	**ConvCAMM**
		Single		
		(Milliseconds)		
Alex's CIFAR-10	5	76.36	41.56	**33.53**
OverFeat	8	1212.74	512.40	**496.83**
ResNet-34	34	887.53	652.28	**577.66**

After testing the performance of ConvCAMM layer, the unified memory scheme is employed to further optimize and accelerate the flow of this cuBLAS-accelerated layer. Results of this experiment are listed in Table 5.4. Results show that the Unified ConvCAMM layer is tens of times faster than the heterogeneous (CPU-GPU) version and achieves a speedup of hundreds of times over the CPU-based sequential version.

Table 5.4 Performance Comparison of Unified ConvCAMM layer over deep classifier (Jetson TX1 Board), best results are written in bold.

Model	Layers	ConvMM (CPU-only)	Hetero ConvMM (CPU+GPU)	Unified ConvCAMM	Speedup Over	
		Double		Single	Sequential	Hetero
		(Milliseconds)				
Alex's CIFAR-10	5	7814.25	129.90	**19.13**	408×	7×
OverFeat	8	254,285.12	2492.94	**122.90**	2069×	20×
ResNet-34	34	190,054.39	2361.18	**423.86**	448×	6×

As Jetson TX1 board has Pascal GPU and supports the half-precision storage and arithmetic operations; so by converting the proposed framework to FP16 format, the half-precision General Matrix-Matrix Multiply (HGEMM) subroutine can be exploited to gain some additional speedup in performance. For this, all imported parameters and allocations are converted from float- to half-precision data types. Table 5.5 presents the comparison of different versions of GPU-only ConvMM layers. The half-precision GPU-only ConvMM layer is approximately 4× faster than the double-precision version and shows a speed-up of 2× over the single-precision scheme as shown in Figure 5.10.

Table 5.5 Comparison of double-, single- and half-precision ConvMM layers on Jetson TX1 Board, best results are written in bold.

Image Size	GPU-only ConvMM Layer		
	Double	Single	Half
	(Milliseconds)		
CIFAR ($32 \times 32 \times 16$)	5.12	3.20	**2.76**
ImageNet ($224 \times 224 \times 16$)	35.61	21.63	**17.20**
VGA ($640 \times 480 \times 16$)	215.74	119.17	**52.03**
SVGA ($800 \times 600 \times 16$)	369.91	162.80	**96.26**

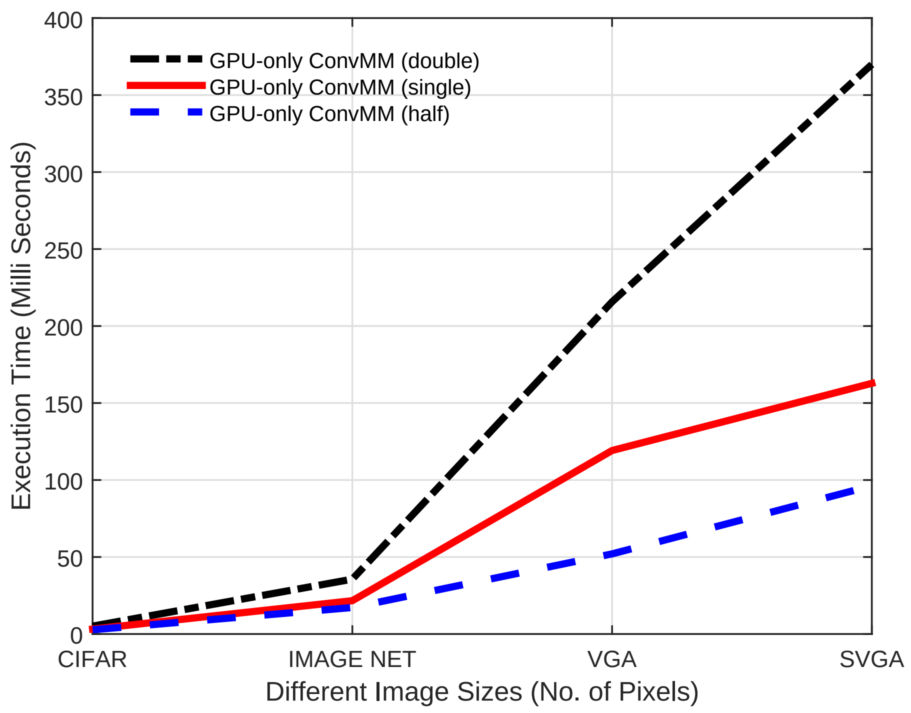


Fig. 5.10 Performance Comparison of double-, single- and half-precision ConvMM layers on Jetson TX1 Board as a function of input image size.

Table 5.6 further validates that the half-precision deep classifiers are significantly faster than the other versions and can be used to accelerate the performance of proposed framework.

Table 5.6 Classification time of double-, single- and half-precision deep models on Jetson TX1 Board, best results are written in bold.

Model	Layers	GPU-only ConvMM Layer Double (Milliseconds)	GPU-only ConvMM Layer Single (Milliseconds)	GPU-only ConvMM Layer Half (Milliseconds)
Alex's CIFAR-10	5	107.03	76.36	**50.41**
OverFeat	8	1924.41	1212.74	**836.03**
ResNet-34	34	1217.33	887.53	**724.47**

The flow of these half-precision classifiers is further optimized using cuBLAS-accelerated matrix multiplication (CAMM)-based convolution approach and proposed data transfer scheme. Table 5.7 tabulates the results achieved by half-precision GEMM routine and unified memory for all three classification networks. Results show that the half-precision models are faster than the single-precision scheme, and can also be useful to reduce the storage consumption and memory bandwidth which are the crucial constraints for embedded platforms.

Table 5.7 Performance of ConvCAMM and Unified ConvCAMM layers on the deep models (Jetson TX1 Board), best results are written in bold.

Model	**Layers**	**ConvCAMM**		**Unified ConvCAMM**	
		Single	**Half**	**Single**	**Half**
		(Milliseconds)			
Alex's CIFAR-10	5	35.53	26.37	19.13	**12.85**
OverFeat	8	496.83	435.3	122.9	**88.02**
ResNet-34	34	577.66	474.90	423.86	**362.76**

This support for FP16 storage and arithmetic on Pascal GPUs is mainly beneficial as a storage format that provides a significant reduction in data transfer time from the host to device side. However as the unified memory is proposed and employed in our embedded framework, this speedup in transactions is not contributing enough to improve the inference performance. This half-precision feature can be further exploited using some specific techniques [80].

5.5.2 Performance Evaluation on Nvidia Shield K1 tablet (For GPUs having C.C. < 3.5)

Performance of proposed scheme is also evaluated on the second embedded platform, Nvidia Shield K1 tablet. This embedded device has Kepler K1 GPU with 192 CUDA cores, and its compute capability is 3.2. Same experiments are performed to measure the performance of proposed framework for embedded devices having compute capability less than 3.5.

First of all, the performance of memory-optimized ConvMM layer is compared with its sequential, heterogeneous and GPU-only versions. Results of this experiment are listed in Table 5.8. It can be noted that for more extensive workloads, GPU-only Unified ConvMM layer is significantly faster than all other ConvMM layers.

However, this memory-based optimized layer is outperformed by the GPU-only and heterogeneous ConvMM layers for smaller workloads due to lengthy context initializations of the unified scheme as shown in Figure 5.11.

Table 5.8 Performance Comparison of different versions of convolutional layer under various computational loads on Nvidia Shield tablet, best results are written in bold.

Required Output Maps = 16				
Image Size	**ConvMM (CPU-only)**	**Hetero ConvMM (CPU+GPU)**	**GPU-only ConvMM**	**GPU-only Unified ConvMM**
	Single			
	(Milliseconds)			
CIFAR(32 × 32 × 3)	37.21	4.20	**3.95**	12.52
ImageNet (224 × 224 × 3)	1570.98	62.27	28.01	**26.59**
VGA (640 × 480 × 3)	10,055.24	327.53	154.32	**116.85**
SVGA (800 × 600 × 3)	15,852.86	519.08	231.95	**181.85**

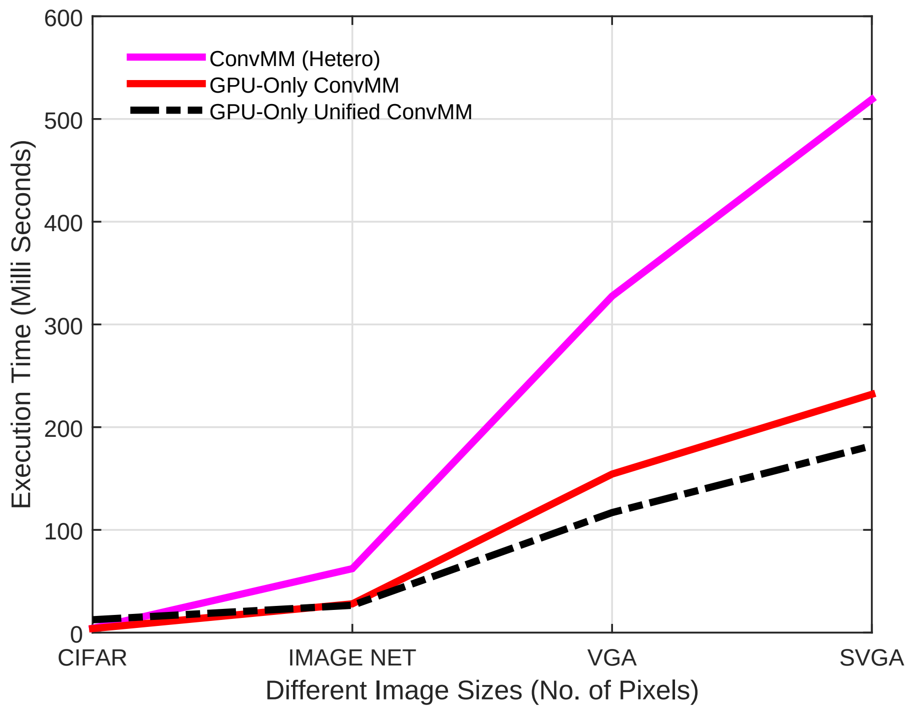


Fig. 5.11 Comparison of Hetero, GPU-only and memory optimized ConvMM layers on Nvidia Shield K1 tablet as a function of input image size.

As stated previously, cuBLAS-accelerated ConvCAMM layer cannot be used on Nvidia Shield K1 tablet because its compute capability is less than 3.5. Therefore, Shared memory-based matrix multiplication (SAMM) is proposed to be employed and exploited on these types of embedded devices. The flow of this shared memory-based ConvSAMM layer is further simplified and accelerated using unified memory scheme. Results of this experiment are summarized in Table 5.9. It can be noted that the GPU-only ConvMM layer is outperformed by the ConvSAMM layer for each case of workload and its performance for higher computational loads can be further improved by using unified memory scheme.

Table 5.9 Performance Comparison of different versions of accelerated and optimized ConvMM layers (Nvidia Shield tablet), best results are written in bold.

Required Output Maps = 16			
Image Size	**GPU-only (ConvMM)**	**ConvSAMM**	**Unified ConvSAMM**
		Single	
		(Milliseconds)	
CIFAR(32 × 32 × 3)	3.95	**3.20**	29.15
ImageNet (224 × 224 × 3)	28.01	**19.91**	34.75
VGA (640 × 480 × 3)	154.32	127.32	**62.98**
SVGA (800 × 600 × 3)	231.95	143.45	**102.69**

Tables 5.10 and 5.11 list the inference performance of shared memory-based ConvSAMM layer and its memory-optimized version (Unified ConvSAMM layer) over the same three classification models.

Table 5.10 Classification time of optimized deep models on Nvidia Shield K1 tablet, best results are written in bold.

Model	**Layers**	**ConvMM (CPU-only)**	**Hetero ConvMM (CPU+GPU)**	**GPU-only ConvMM**	**ConvSAMM**	**Unified ConvSAMM**
				Single		
				(Milliseconds)		
Alex's CIFAR-10	5	10,449.21	419.67	144.23	74.20	**38.57**
OverFeat	8	440,631.27	5792.54	3899.43	1784.24	**632.91**
ResNet-34	34	252,955.27	3319.34	2545.83	1210.16	**767.53**

Results show that the Unified ConvSAMM layer-based deep classifiers outperform the other versions and achieve significant speedup in inference performance of image classifiers.

By using the shared memory-based accelerated matrix multiplication and optimized unified memory scheme, the proposed Unified ConSAMM layer is hundreds of times faster than the sequential version and 4×-11× faster than the heterogeneous version.

Table 5.11 Performance gain achieved by proposed optimized scheme on Nvidia Shield K1 tablet, best results are written in bold.

Model	Layers	Speedup over	
		Sequential	Hetero
Alex's CIFAR-10	5	271×	11×
OverFeat	8	696×	9×
ResNet-34	34	330×	4×

Furthermore, energy consumption of proposed flow is also evaluated as it is an important performance metric of mobile devices. A comparison is performed with sequential and heterogeneous versions, and results are listed in Table 5.12. Results show that the proposed Unified ConvSAMM layer is consuming up to 75% less energy than the heterogeneous approach. It validates that the proposed scheme significantly reduces the energy consumption per image frame.

Table 5.12 Power consumption of optimized scheme on Nvidia Shield K1 tablet (joule), best results are written in bold.

Model	Layers	ConvMM (CPU-only)	Hetero ConvMM (CPU+GPU)	Unified ConvSAMM	Efficiency over	
					Sequential	Hetero
Alex's CIFAR-10	5	16.0	0.410	**0.202**	79×	2×
OverFeat	8	850.8	3.2	**0.529**	1608×	6×
ResNet-34	34	480.0	1.8	**0.432**	1111×	4×

5.5.3 Performance Comparison with Torch framework

All these experiments are conducted on three different neural architectures (ResNet-34, OverFeat model and Alex Krizhevsky's network for CIFAR-10). These deep classifiers are trained using Torch computing framework and the trainable parameters are imported to be used by the proposed scheme. Table 5.13 presents a performance comparison of Torch-based models and proposed optimized networks utilizing the trained parameters imported from the former models. This comparative analysis is performed on Jetson TX1 embedded board which has 4 GB of RAM, a Maxwell GPU having 256 CUDA cores and 16 GB of embedded MultiMediaCard storage.

The performance of Torch's cuDNN (Nvidia CUDA Deep Neural Network) library is evaluated for the same three deep classifiers. The cuDNN is a GPU-based library of highly tuned primitives for neural networks.

The results show the comparable performance of proposed scheme and torch-based library for OverFeat and Alex's CIFAR-10 networks, while the proposed scheme is slower than the cuDNN library for ResNet-34 architecture having small filters. However, the proposed framework can also be accelerated for such networks by using Winograd's filtering algorithm.

Table 5.13 Comparison of proposed scheme with torch computing framework on Jetson TX1 embedded board.

Model	Layers	Input Image Size	Proposed Unified ConvCAMM Flow	Torch (cuDNN)
			(Milliseconds)	
Alex's CIFAR-10	5	(1,3,32,32)	12.85	37.11
OverFeat	8	(1,3,231,231)	88.02	140.78
ResNet-34	34	(1,3,224,224)	362.76	102.10

As Embedded platforms have limited storage capacity; so the size of trained models, computing framework and required computational packages must be considered as limiting constraints for the realization of neural classifiers on embedded devices. Table 5.14 tabulates and compares the size of trained models imported from the torch computing language and converted parameters for the proposed framework.

It can be noted that the installation of computing framework, its libraries (in this particular case, 828 MB is consumed by the Torch and its packages on Jetson TX1 embedded board) and a complete model with trained parameters need a significant amount of storage space. While just using the trained parameters on the embedded platform, proposed CUDA-based optimized scheme can provide comparable inference performance without installing any extra memory consuming computational package.

Table 5.14 Size of Model/Parameters.

Model	**Layers**	**Trained Parameters (float)**	**Torch**
		(Megabytes)	
Alex's CIFAR-10	5	19	41
OverFeat	8	556	1190
ResNet-34	34	83	215

All these results illustrate that the proposed optimized scheme can perform image classification in real time on embedded platforms with significant improvement in overall results including inference performance, energy efficiency and storage requirements over the sequential and heterogeneous versions. This suggests promising future work towards the realization of useful visual analysis tasks on the embedded platforms [10, 29, 81, 82]. The next chapter discusses the realization of neural network-based automatic license plate recognition (ALPR) system on a mobile platform [83].

Chapter 6

Automatic License Plate Recognition System on Mobile Platform

The realization of trained neural networks on the embedded platforms can open a wide range of applications, especially in the computer vision field. This chapter presents the realization of neural network-based automatic license plate recognition (ALPR) system on a mobile platform[1]. Trained parameters of a highly precise Italian license plate detection and recognition system are imported to be used by our CUDA-based framework. This ALPR system is realized on a mobile platform by simplifying the flow of trained deep architecture developed for computationally powerful desktop and server environments.

6.1 Introduction

The principle responsibility of an ALPR system is to recognize an on-road vehicle using its license plate. A complete ALPR system is commonly subdivided into various computational blocks that execute sequentially, such as (i) acquisition of image and its preprocessing, (ii) detection and localization of license plate and (iii) character segmentation and recognition. The detection and recognition of license plate is an important research area in the intelligent transportation system (ITS) and

[1]Part of the work described in this chapter has been previously published in **Future Internet** Journal as "Rizvi, S.T.H.; Patti, D.; Björklund, T.; Cabodi, G.; Francini, G. Deep Classifiers-Based License Plate Detection, Localization and Recognition on GPU-Powered Mobile Platform. Future Internet 2017, 9, 66.

is widely employed in numerous practical applications, such as electronic payment systems, traffic surveillance, etc [84, 85].

Owing to its significance, several computer vision algorithms and techniques have been proposed for the detection and recognition of a license plate and its contents [86–91]. Although the deep classifiers can provide outstanding results in detection, localization and recognition of a license plate like other computer vision tasks, but it would be challenging to realize a computationally intensive deep architecture-based ALPR system on an embedded platform.

As discussed earlier, the requirements of training and inference phases of a deep classifier are slightly different. The training phase cannot be performed on an embedded platform due to a huge number of input samples, parallel classifications and numerous iterations required for learning. The state-of-the-art deep architectures cannot be trained without support of powerful desktop stations and GPU clusters. However, a trained network usually needs to classify only the single images in its inference phase. Thus, it is possible to realize a neural network-based application on a mobile platform by utilizing the computational resources of its embedded GPU and simplifying the flow of a trained deep architecture developed for computationally powerful environments (like a desktop workstation or GPU cluster). By importing the trained parameters of a neural network and replicating the same flow using an optimized framework, a mobile device can perform image classification in real-time when there is no network connectivity or back-end support.

6.2 Related Work

Many approaches have been proposed recently for detection and recognition of license plates [86–91]. However, there are very few realizations of these automatic license plate recognition approaches on the embedded platforms. Several memory-consuming computational libraries and packages are required to realize an image classifier on an embedded platform. Realization of these approaches is a challenging task due to limited computational power and storage resources of an embedded device [92]. In addition, an approach must be efficient and effective to satisfy the computational and memory constraints of an embedded device.

A license plate recognition system for embedded platforms is proposed in [93]. A simple neural classier "AlexNet" is used and trained for recognition of Korean License plates. This system is realized on Jetson TX1 embedded board and shows high recognition accuracy. However, the Korean license plate contains only digits that are easier to recognize than the alphanumeric characters.

A digital signal processor (DSP) kit-based real-time embedded ALPR system is presented in [94]. Different hardware modules are interfaced with a digital signal processor to perform the real-time detection and recognition of license plates. However, results such as inference performance and recognition accuracy are not discussed that can be used for comparison with other embedded platform-based realizations.

An embedded plate recognition system for Brazilian vehicle identification is presented in [95]. This system is implemented using C language and can be deployed on the embedded platforms having Linux operation system. This real-time recognition system has some limitations for challenging imaging conditions, such as perspective distortion, light intensity, etc.

Some research work has also been done on deep classifier-based license recognition systems, but these systems require computationally powerful desktop workstations and GPU servers to perform the real-time recognition [10, 96]. The realization of a neural network-based automatic license plate recognition system is not well explored for a mobile platform.

In this chapter, we presented an embedded platform based Italian license plate recognition system. This deep neural network-based ALPR system is realized on a mobile platform by simplifying its flow [10]. The CUDA-based framework presented in previous chapters is used to replicate the neural classifiers. Trained parameters are imported to be used by the replicated network. By exploiting the computational resources of an embedded GPU, visual analysis tasks can be performed on a handheld device without any need of network connectivity. Other resources (e.g, the camera, Bluetooth and WiFi) of a mobile device can also be utilized for acquisition of input data and communication of results.

6.3 Plate Recognition System Developed for Desktop and Server Environments

Training of a fully convolutional deep neural network is performed using a synthetic dataset having images of a broad diversity [10]. The accuracy of this network is tested on a dataset of real images, it is done to assure that the learned features are robust to the various image capturing conditions. The neural network-based architecture for an Italian license plate recognition system is constructed and trained using Torch framework. The detection and localization of plates and characters are performed using two different classifiers as shown in Figures 6.1 and 6.2.

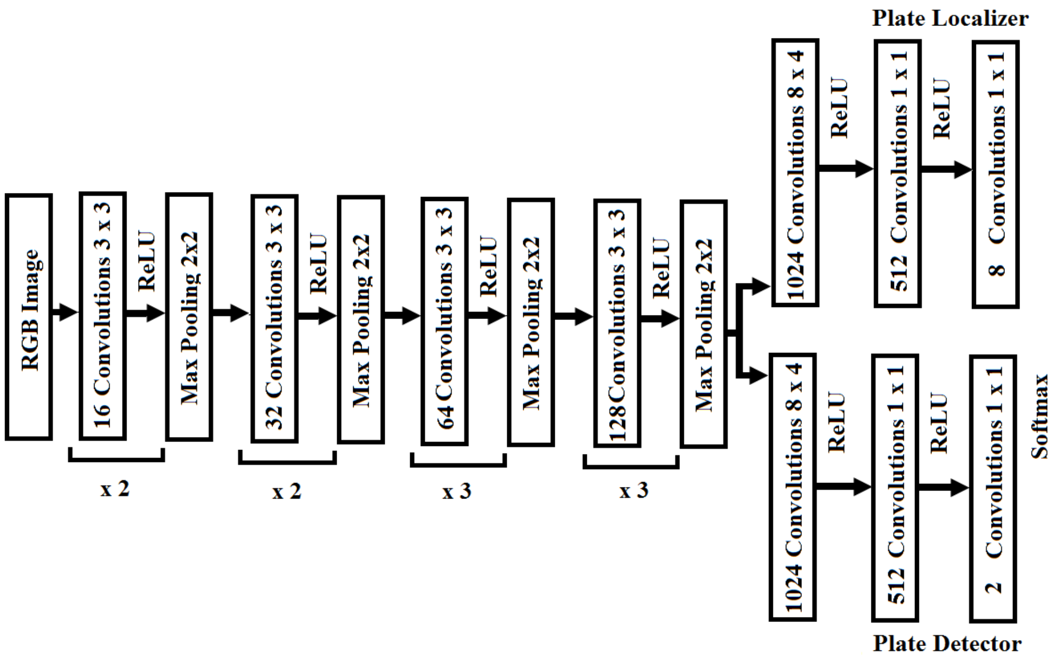


Fig. 6.1 Italian plate detector and localizer.

The task of detection and localization can be performed simultaneously for both networks (for plates and characters) using shared connections of convolutional layers. The same neural classifiers with minor modifications can be trained on a new dataset to recognize the multi-style license plates [97].

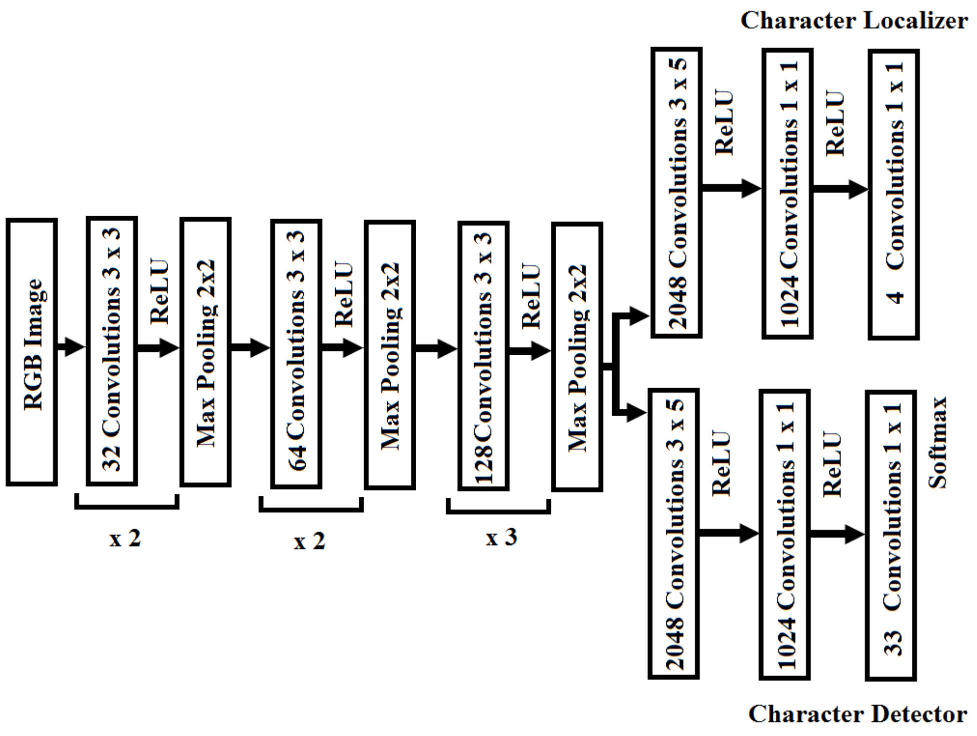


Fig. 6.2 Character detector and localizer.

These neural networks attach weights in a convolutional pattern instead of using fully connected layers for classification stage. These fully convolutional layers with max-pooling functions keep only the strongest features, by decreasing the spatial resolution and relevance. An input image of any size can be classified by reorganizing the fully connected layers to fully convolutional layers. The output of this network becomes a matrix where the classification of each window in the original image corresponds to a specific element of the output matrix.

Figure 6.3 shows the flow of a deep neural architecture-based Italian plate recognition system developed for running on computationally powerful desktop and server environments. The accuracy of this system is 98% on real images. This highly precise ALPR system is composed of numerous computationally intensive image processing blocks that perform different important operations, such as multi-scale detection and localization of a single input frame for the license plates and then characters. This complete architecture is highly complex and demands high computational power. Realization of a computationally demanding deep neural architecture on a mobile platform is very challenging due to limited computational power and storage resources of an embedded device.

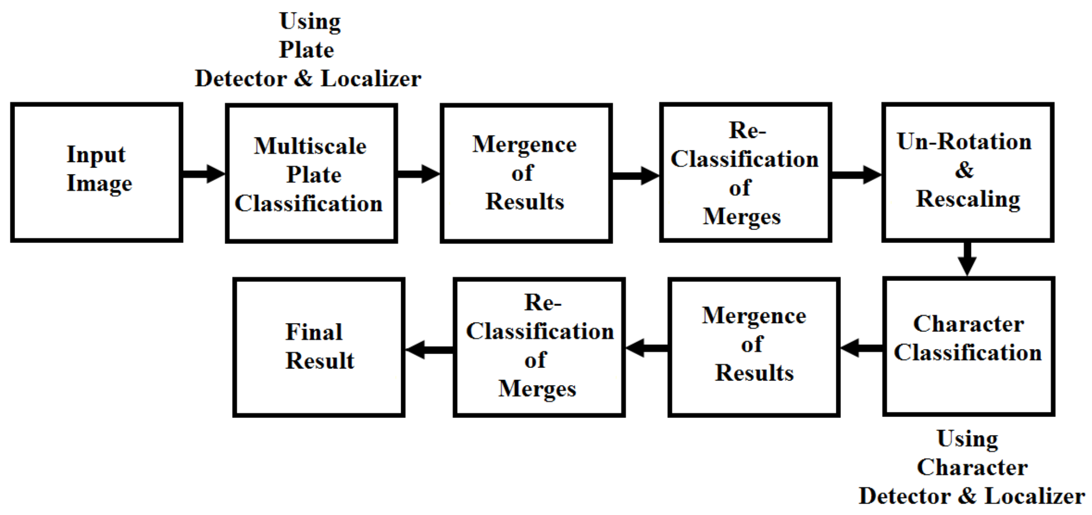


Fig. 6.3 Flow of neural network-based license plate recognition system developed for desktop and server environments.

The computationally demanding architectures like these can be realized on a low-power mobile platform by simplifying the flow of the trained architecture and exploiting an optimized library for the neural networks. However, there would be a direct trade-off between the classification accuracy and the inference performance for the simplified architecture.

6.4 Simplified Flow for Mobile Platform

This section presents our simplified flow of the Italian license plate recognition system for the low-power mobile platforms. Trained parameters are imported from the plate and character networks discussed in the previous section. These parameters are imported to be used by the identical classifiers realized using our CUDA-based framework. The GPU of the mobile platform is exploited using the CUDA-based functions to accelerate the detection and localization of plate and characters.

Figure 6.4 shows the flow of the simplified architecture. The trained network for the plate detection and localization is realized in its original form, this is done to correctly localize a single license plate in a complex scene. In order to reduce the memory consumption and improve the inference performance, the trained parameters are converted from double to float data type. However, this conversion does not affect the accuracy of the plate detection. Since the original network discussed in the previous section demands multiple classifications for plate and character recognition that can be computationally and memory intensive for an embedded platform, so the multiple classifications are avoided to simplify the flow of the original network by sacrificing some classification accuracy. An error correction block is also introduced in the simplified flow to improve the recognition performance. This error correction scheme is based on the position of the detected characters, as the Italian license plates have a specific format, so some errors can be corrected by verifying the position of the characters in the license plate.

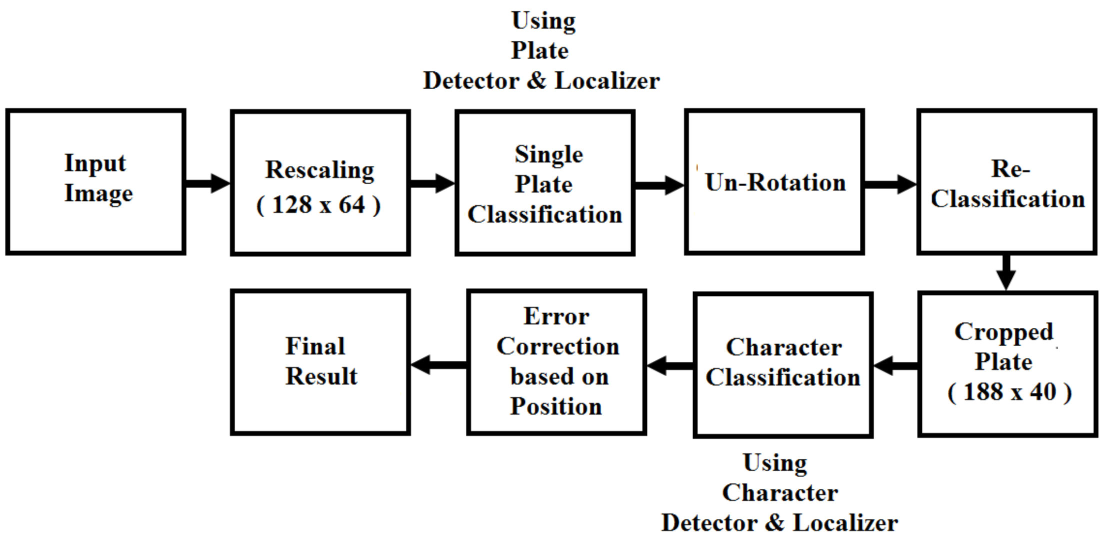


Fig. 6.4 Simplified Flow of neural network-based license plate recognition system.

The original network discussed in Section 6.3 is entirely scale-independent by carrying out the multiple classifications through scaling down the input image in steps by a factor of 2, while the simplified network for the mobile platform is limited to a scale factor of 2 between the minimal and maximal recognizable scale. It significantly reduces the computational complexity but also restricts the user to capture the correct image by applying the appropriate zoom. The license plate should span about 30 to 90% of the captured image and may be rotated up to 45 degrees. Light conditions must be such that the license plate should be readable in captured image by a human. Figure 6.5 shows some examples of such challenging imaging conditions.

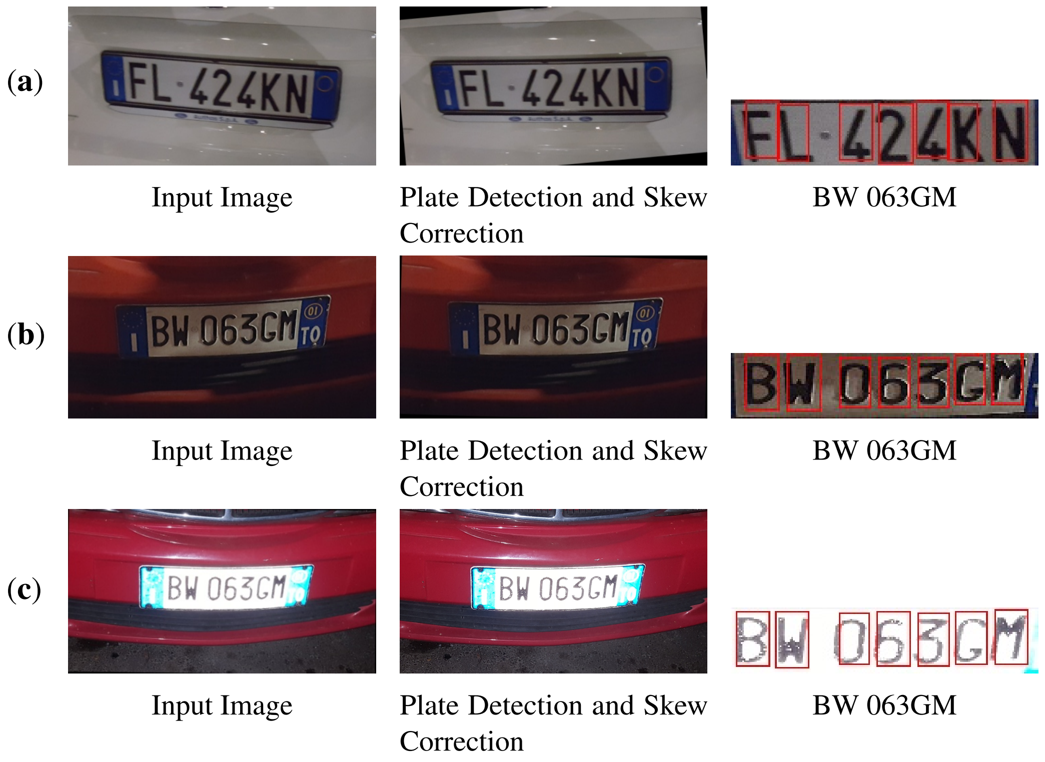

(a)
Input Image
Plate Detection and Skew Correction
BW 063GM
(b)
Input Image
Plate Detection and Skew Correction
BW 063GM
(c)
Input Image
Plate Detection and Skew Correction
BW 063GM

Fig. 6.5 Images captured at night in different imaging conditions: (a) image captured with perspective distortion; (b) image captured with insufficient illumination; (c) image captured with over-exposure and reflections due to speed-light.

6.4.1 Simplified flow for Plate Classification

First of all, the size of the input image is scaled down to simplify the flow of the trained deep architecture discussed in Section 6.3. The input image is resized to 128 $\times$ 64 to allow a single license plate classification per image as shown in Figure 6.6. Then, the license plate classification is performed to detect the plate and find its position inside the input image. This detection and localization network consists of 16 convolutional layers as shown in Figure 6.1. Out of 16 convolutional layers, the first 10 layers are to provide shared connections for simultaneous license plate detection and localization. The other 6 layers are to perform either plate detection or localization (3 convolutional layers for each task). The plate detector has 2 outputs while the plate localizer provides 8 values. The output of plate detector indicates that a license plate is detected or not, while the output of localizer represents the x- and y- coordinates of 4 points that enclose the detected license plate.

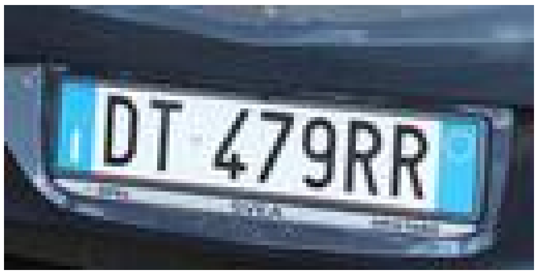


Fig. 6.6 Rescaled input image.

The output of plate localizer normally generates a simple quadrilateral (parallelogram). The skewness of license plates negatively affects the performance of subsequent characters detection and localization steps. Besides other factors, the performance of character classification also depends on the skew correction of detected plate and can be improved using a reliable approach [98]. Thus, the input image is un-rotated to convert this parallelogram to a perfect rectangle for accurate character classification.

After skew correction, it is essential to acquire the updated coordinates of the detected plate, to obtain better results from character detection and localization. It can be achieved either by reclassifying the license plate or by computing the updated coordinates based on the rotation (skew correction). The plate reclassification helps to obtain more accurate license plate localization but also increases the overall computational time. So, the plate reclassification is performed, and detected plate is cropped from the original input image based on the updated localization results.

6.4.2 Simplified flow for Character Classification

The next part of the simplified automatic license recognition system is the classification of characters in the cropped image. The character classifier shown in Figure 6.2 is composed of 12 convolutional layers and can also perform simultaneous detection and localization of characters as the plate classifier. The character detector yields 33 outputs per classification that represent the possibility of different alpha-numeric characters for Italian license plates, while the character localizer provides 4 values per classification that represent the x- and y-coordinates of the top-left and the bottom-right vertices of the rectangle that encloses the detected character.

First of all, the cropped image from the plate classification stage is resized to 188 × 40. This is done to acquire the appropriate classifications of characters. It can be noted that the dimension of this crop is higher than the input image used for the plate detection and localization because higher resolution of characters is not required for plate classification. The character detection and localization network has 3 max-pooling layers and the stride between the classifications are 8 pixels. The complexity of network is significantly reduced by rescaling the crop to a height of 40 pixels that reduces the classifications to a single line. This yields a crop of 188 × 40 pixels due to the aspect ratio and provides 21 partly overlapping slots as shown in Figure 6.7.

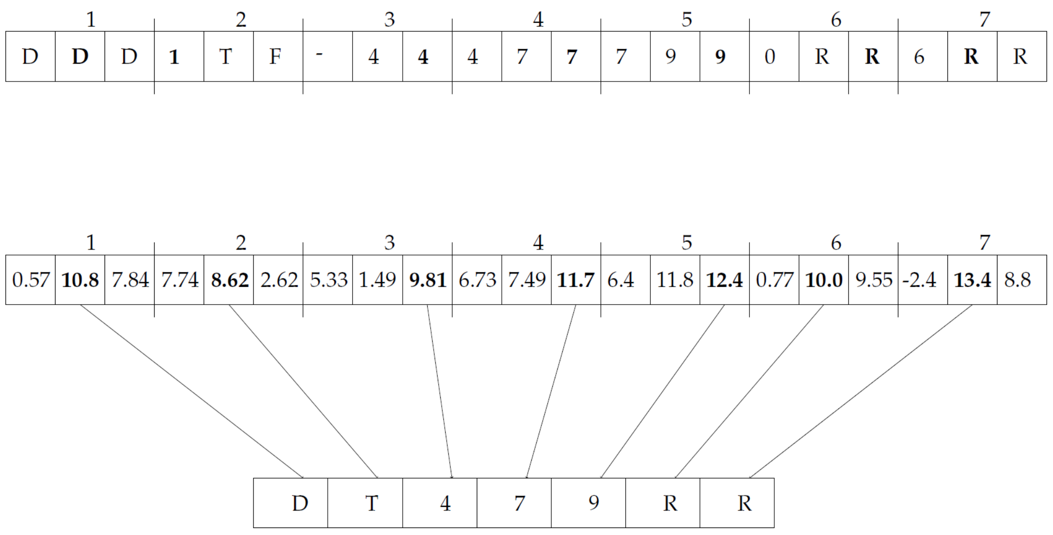


Fig. 6.7 Overlapping classifications and selection of alpha-numeric characters for final result.

A single slot or classification represents a 24 × 40 portion of the input image that may contain an alpha-numeric character. The highest output value of the character detector yields the final result out of 33 values representing the Italian alphabets (22), digits (0-9) and a null (background) value.

There are total 7 characters in an Italian license plate as shown in Figure 6.8. Out of 21 partly overlapping classifications, the simplified Italian plate recognition system must select 7 characters (4 alphabets and 3 digits). A heuristic approach is adopted to find the final result. The 21 slots (classifications) are divided into 7 small clusters each having 3 slots as shown in Figure 6.7. The highest output values within each cluster are selected for the final result.

Furthermore, the output of character detector is directly affected by the size of the cropped image. The number of classifications and the accuracy of character classifier can be varied by selecting an offset on plate localization step defining the character input crop. It can be visualized from Figure 6.8 that the cropped license plate with an offset of 4 pixels yields exactly 7 classifications of an Italian license plate.

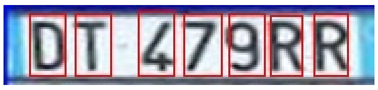


Fig. 6.8 Detected Characters in output image.

6.5 Experiments and Results

First of all, the performance of original network discussed in Section 6.3 is measured on two different types of GPU platforms. The first platform is a powerful desktop workstation equipped with Intel Core i7 processor at 3.40 GHz and 16 GB of RAM memory. This workstation is connected to a Nvidia Quadro K2200 graphic card having 640 CUDA cores and 4GB of GPU memory via a PCI express bus. The second platform is powerful Nvidia Jetson TX1 embedded board. As mentioned earlier, this embedded board comes with ARM Cortex A-57 processor, 4 GB of RAM and a Maxwell GPU having 256 CUDA cores. The execution time of Torch-based original network is examined on both platforms for performance analysis.

An input image of size 640 $\times$ 480 is selected for evaluation of multi-scale classification of Italian license plates. As listed in Table 6.1, there can be two cases in the original ALPR system. It can be visualized from Figure 6.3 that if a license plate is not detected in the input image, then there is no need to carry out the remaining processes of plate and character classification. This restriction significantly reduces the inference time, as can be observed from Table 6.1. Conversely, execution of all processing blocks like reclassification and merging of results would be performed in case of a detected license plate. As mentioned previously, the accuracy of original plate recognition system is 98% on real photos. However, it can be noted that in case of positive classification, the execution time of the original ALPR system is higher (1.45 + 3.24 = 4.69 seconds) on a powerful embedded platform like Jetson TX1 board.

Table 6.1 Performance analysis of original and simplified networks.

Network	Quadro K2200		Jetson TX1		Nvidia Shield K1
	Original	**Simplified**	**Original**	**Simplified**	**Simplified**
			(Milliseconds)		
Plate Classification Time	150	26	1450	244	300
Character Classification Time	510	27	3240	248	250

The inference performance of the original network can be much worse on the mobile platforms that have limited computational resources than the powerful desktop workstations and embedded boards. At expense of some recognition accuracy, the inference performance of an ALPR system can be improved significantly by simplifying the original flow and eliminating the computationally intensive processing blocks.

The simplified flow of ALPR system proposed in Section 6.4 is first evaluated on same GPU platforms, Quadro K2200 and Jetson TX1 board. This is done to compare the inference performance of proposed simplified flow with the original network. Results show that the execution time is reduced significantly due to elimination of intensive tasks like merging of results and reclassifications in the simplified flow. Then the performance of this proposed flow is evaluated on Nvidia Shield K1 tablet that is equipped with Tegra K1 GPU having 192 CUDA cores. For an input image of size 128×64, the plate detection and localization is performed in around 300 milliseconds. Additionally, the character classification of a cropped license plate of size 188×40 is performed approximately in 250 milliseconds. Results show that the inference performance of a license plate recognition system can be significantly improved by using the input images of smaller dimensions and single-scale classification.

As mentioned earlier, the size of trained models, computing framework and required computational packages must be considered as limiting constraints for the realization of neural classifiers on the low storage embedded platforms like a mobile device. A complete model with trained parameters needs a significant amount of storage space. By importing the trained parameters and replicating a neural network-based system using our CUDA-based scheme, there is no need of any other memory consuming framework or computational packages. Table 6.2 compares the size of trained parameters imported from the torch computing language and converted parameters for the simplified flow of automatic license recognition system. It can be noted that the proposed flow is consuming half storage space as compared to the original network, because the trained parameters are converted to float data type to reduce the memory consumption and improve the inference performance.

Table 6.2 Size of Parameters.

Network	**Trained Parameters**	**Converted Parameters**
	Double	**Float**
	(Megabytes)	
Plate Classifier	75.70	37.85
Character Classifier	65.63	32.81

However, the simplification of architecture indeed affects the recognition performance. Table 6.3 lists the classification accuracy of full plate detection (all 7 alpha-numeric characters) and a single character detection. The testing dataset is comprised of real-world images of Italian license plates downloaded from the Internet [99]. Total 788 crops of aspect ratio 2:1 roughly centered around license plates are used to test the performance of simplified slow.

Results listed in Table 6.3 show that the classification accuracy can be increased by selecting a suitable offset value. After finding the best offset value (=4), the coordinates generated by the plate localizer (for cropping the input images) are moved some pixels in both horizontal directions to analyze the effect of shifting on the classification accuracy. Different offset and shifting values are tested and the results show that the best classification accuracy is obtained by cropping the detected license plate with offset 4 outside the localization values and shifting 2 pixels to the right.

Table 6.3 Classification accuracy of character detection and localization.

Offset for Cropping Detected Plates	**Full Plate Accuracy**	**Single Character Accuracy**
0	17%	68%
2	28%	78%
4	30%	82%
8	25%	79%
4 (+2) (21 detected characters)	54%	90%
4 (+2) + Error correction	**61%**	**92%**
Original Network	**94%**	**98%**

Furthermore, a new block for error correction is employed in the simplified flow to increase the classification accuracy. As Italian license plates have a specific format that can be visualized using Figure 6.8, an error correction scheme based on the position of characters is applied. Table 6.4 tabulates some rules defined for correction of characters based on their position in the Italian license plate. The classification accuracy after error correction is also listed in Table 6.3. It can be noted that there is an increase of 13 % in full plate accuracy and an increase of 2 % in single character accuracy after position-based error correction.

Table 6.4 Error correction based on position.

Original Character	**Correction**
B	8
D	0
Z	2
J	1
G	6

The accuracy of this mobile device-based ALPR system can be further improved by increasing the resolution of the input image and employing the multi-scale classification and other eliminated processing blocks from the deep neural architecture developed for desktop and server environments. The execution time of these computational blocks and neural architectures can be significantly improved by compressing or reducing the computational complexity of convolutional layers.

In this work, the computational power of embedded GPU is utilized to accelerate the image classification on a hand-held device. A Graphics processing unit (GPU) is not specialized for only Visual analysis tasks, the computational resources of an embedded GPU can also be exploited to accelerate the other general purpose (non-computer vision) applications [12, 100].

Chapter 7

Conclusions and Future Work

In this thesis, we presented an optimized scheme to deploy trained neural architectures on embedded platforms. The first part of this thesis was related to the development of a CUDA-based framework to perform image classification via deep classifiers on embedded devices. Required networks were trained using the desktop workstation or GPU clusters depending on the required computational power. Then, the trained parameters were imported and fed to the identical deep architectures constructed using our CUDA-based implemented functions without any precision loss. The CUDA computing language was used to implement and accelerate all required functions, including matrix multiplication based convolutional (ConvMM) layer. The performance of realized networks is evaluated on two different type of embedded platforms: on Jetson TX1 embedded board and Nvidia Shield K1 tablet. Results showed that the GPU-accelerated ConvMM layer based deep classifiers were hundreds of times faster than the CPU-based sequential versions.

The second part of this thesis is the extension of first work, where we optimized the proposed scheme to deliver higher inference performance and energy efficiency with lesser memory requirements. A set of optimization techniques were proposed and employed in our embedded framework to bridge the gap towards the real-time image classification on the hand-held devices. Optimized data transfer scheme, hardware dependent matrix multiplication and GPU architecture based exploitation of resources significantly improved the performance of proposed framework. Results illustrated that the proposed scheme can perform real-time image classification on embedded platforms.

These results motivated us to explore the other possible image classification problems that can benefit from a portable mobile device. An automatic license plate recognition (ALPR) system was realized on the mobile platform by simplifying the flow of a trained neural architecture developed for running on the desktop and server environments. The performance of already trained networks was measured on a powerful desktop workstation equipped with Quadro K2200 GPU card and Jetson TX1 board. Results showed that this neural network-based ALPR system cannot be realized on a low power mobile platform in its original forms. By reducing the architectural complexity of network and resolution of the input image, the task of license plate recognition was performed on Nivida Shield K1 tablet. However, this reduction in architectural complexity directly affected the recognition accuracy. A comparative analysis of arithmetical complexity and inference performance was performed. Experimental results concluded that the even computationally complex neural network can be deployed on the embedded platforms by sacrificing some recognition performance.

All these results suggest promising future directions towards the realization of neural network-based image recognition and classification problems on the embedded platforms. The performance of developed framework can be further improved by reducing the computational complexity of the convolution operation. Different approaches like Winograd's minimal filtering technique can be adopted to minimize the arithmetic complexity of the convolution operation over small tiles. The Winograd technique minimizes the complexity by performing the element-wise multiplication instead of matrix multiplication and reduces the number of arithmetic operations compared to traditional convolutions approach. This technique can further improve the performance of proposed framework for small filter and batch sizes as used by the recent deep architectures.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[3] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pages 580–587, Washington, DC, USA, 2014. IEEE Computer Society.

[4] R. Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, Dec 2015.

[5] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, June 2014.

[6] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, June 2015.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[8] M. B. López, H. Nykänen, J. Hannuksela, O. Silvén, and M. Vehviläinen. Accelerating image recognition on mobile devices using gpgpu. *Parallel Processing for Imaging Applications*, 7872:78720R, 2011.

[9] Y. Wang, S. Li, and A. C. Kot. Deepbag: Recognizing handbag models. *IEEE Transactions on Multimedia*, 17(11):2072–2083, Nov 2015.

[10] T. Björklund, A. Fiandrotti, M. Annarumma, G. Francini, and E. Magli. Automatic license plate recognition with convolutional neural networks trained

on synthetic data. In *Proceedings of the IEEE 19th International Workshop on Multimedia Signal Processing (MMSP 2017), Luton, UK*, October 2017.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, October 2008.

[12] Syed Tahir Hussain Rizvi, Gianpiero Cabodi, Denis Patti, and Muhammad Majid Gulzar. A general-purpose graphics processing unit (gpgpu)-accelerated robotic controller using a low power mobile platform. *Journal of Low Power Electronics and Applications*, 7(2), 2017.

[13] Torch computing framework. http://torch.ch. Accessed: 2017-10-30.

[14] Caffe: Deep learning framework. http://caffe.berkeleyvision.org/. Accessed: 2017-10-30.

[15] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[16] cuda-convnet. https://code.google.com/archive/p/cuda-convnet/. Accessed: 2017-12-02.

[17] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1915–1929, Aug 2013.

[18] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.

[19] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, June 2014.

[20] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[21] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[22] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[23] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

[24] Brian Chu, Daylen Yang, and Ravi Tadinada. Visualizing residual networks. *CoRR*, abs/1701.02362, 2017.

[25] David Kirk and Wen mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, 2012.

[26] R. Tsuchiyama, T. Nakamura, T. Iizuka, and A. Asahara. *The OpenCL Programming Book*. Fixstars Corporation, 2010.

[27] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and opencl. *CoRR*, abs/1005.2581, 2010.

[28] Junli Gu, Yibing Liu, Yuan Gao, and Maohua Zhu. Opencl caffe: Accelerating and enabling a cross platform machine learning framework. In *Proceedings of the 4th International Workshop on OpenCL*, IWOCL '16, pages 8:1–8:5, New York, NY, USA, 2016. ACM.

[29] Y. Huang, R. Wu, Y. Sun, W. Wang, and X. Ding. Vehicle logo recognition system based on convolutional neural networks with a pretraining strategy. *IEEE Transactions on Intelligent Transportation Systems*, 16(4):1951–1960, Aug 2015.

[30] A. H. Abdulnabi, G. Wang, J. Lu, and K. Jia. Multi-task cnn model for attribute prediction. *IEEE Transactions on Multimedia*, 17(11):1949–1959, Nov 2015.

[31] S. T. H. Rizvi, G. Cabodi, D. Patti, and M. M. Gulzar. Comparison of gpgpu based robotic manipulator with other embedded controllers. In *2016 International Conference on Development and Application Systems (DAS)*, pages 10–15, May 2016.

[32] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 126–131, Aug 2009.

[33] V. H. Naik and C. S. Kusur. Analysis of performance enhancement on graphic processor based heterogeneous architecture: A cuda and matlab experiment. In *2015 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pages 1–5, Feb 2015.

[34] M. T. Satria, S. Gurumani, W. Zheng, K. P. Tee, A. Koh, P. Yu, K. Rupnow, and D. Chen. Real-time system-level implementation of a telepresence robot using an embedded gpu platform. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1445–1448, March 2016.

[35] N. A. Vandal and M. Savvides. Cuda accelerated illumination preprocessing on gpus. In *2011 17th International Conference on Digital Signal Processing (DSP)*, pages 1–6, July 2011.

[36] S. Raghav, M. Ruggiero, A. Marongiu, C. Pinto, D. Atienza, and L. Benini. Gpu acceleration for simulating massively parallel many-core platforms. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1336–1349, May 2015.

[37] A. R. Baek, K. Lee, and H. Choi. Speed-up image processing on mobile cpu and gpu. In *2015 Asia Pacific Conference on Multimedia and Broadcasting*, pages 1–3, April 2015.

[38] Loc Nguyen Huynh, Rajesh Krishna Balan, and Youngki Lee. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. In *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, WearSys '16, pages 25–30, New York, NY, USA, 2016. ACM.

[39] P. K. Tsung, S. F. Tsai, A. Pai, S. J. Lai, and C. Lu. High performance deep neural network on low cost mobile gpu. In *2016 IEEE International Conference on Consumer Electronics (ICCE)*, pages 69–70, Jan 2016.

[40] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pages 1201–1205, New York, NY, USA, 2016. ACM.

[41] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4820–4828, June 2016.

[42] Andrew Lavin. Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308, 2015.

[43] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530, 2015.

[44] Zejia Zheng, Zhu Li, A. Nagar, and Kyungmo Park. Compact deep neural networks for device based image classification. In *2015 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, pages 1–6, June 2015.

[45] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. Enet: A deep neural network architecture for real-time semantic segmentation. *CoRR*, abs/1606.02147, 2016.

[46] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[47] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061, 2016.

[48] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[49] Nazri Mohd Nawi, Walid Hasen Atomi, and M.Z. Rehman. The effect of data pre-processing on optimized training of artificial neural networks. *Procedia Technology*, 11(Supplement C):32 – 39, 2013. 4th International Conference on Electrical Engineering and Informatics, ICEEI 2013.

[50] M. D. Zeiler and R. Fergus. Stochastic Pooling for Regularization of Deep Convolutional Neural Networks. *ArXiv e-prints*, January 2013.

[51] Jason Cong and Bingjun Xiao. *Minimizing Computation in Convolutional Neural Networks*, pages 281–290. Springer International Publishing, Cham, 2014.

[52] S. Puri K. Chellapilla and P. Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, October 2006.

[53] Michaël Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *CoRR*, abs/1312.5851, 2013.

[54] J. Wang, J. Lin, and Z. Wang. Efficient convolution architectures for convolutional neural network. In *2016 8th International Conference on Wireless Communications Signal Processing (WCSP)*, pages 1–5, Oct 2016.

[55] W. Jiang, Y. Chen, H. Jin, B. Luo, and Y. Chi. A novel fast approach for convolutional networks with small filters based on gpu. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 278–283, Aug 2015.

[56] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 67–76, Aug 2016.

[57] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zero and data reuse-aware fast convolution for deep neural networks on gpu. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES '16, pages 33:1–33:10, New York, NY, USA, 2016. ACM.

[58] K. Li, H. Shi, and Q. Hu. Complex convolution kernel for deep networks. In *2016 8th International Conference on Wireless Communications Signal Processing (WCSP)*, pages 1–5, Oct 2016.

[59] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition*, pages 346–361. Springer International Publishing, Cham, 2014.

[60] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In David Blei and Francis Bach, editors, *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 448–456. JMLR Workshop and Conference Proceedings, 2015.

[61] K. Hara, D. Saito, and H. Shouno. Analysis of function of rectified linear unit used in deep learning. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2015.

[62] Syed Tahir Hussain Rizvi, Gianpiero Cabodi, Denis Patti, and Gianluca Francini. Gpgpu accelerated deep object classification on a heterogeneous mobile platform. *Electronics*, 5(4), 2016.

[63] E. Torti, G. Danese, F. Leporati, and A. Plaza. A hybrid cpu-gpu real-time hyperspectral unmixing chain. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(2):945–951, Feb 2016.

[64] S. T. H. Rizvi, G. Cabodi, and G. Francini. Gpu-only unified convmm layer for neural classifiers. In *4th International Conference on Control, Decision and Information Technologies*, April 2017.

[65] S. T. H. Rizvi, G. Cabodi, A. Arif, M. Y. Javed, and M. M. Gulzar. Gpgpu based concurrent classification using trained model of handwritten digits. In *2016 International Conference on Open Source Systems Technologies (ICOSST)*, pages 142–146, Dec 2016.

[66] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

[67] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[68] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[69] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. Accelerating deep convolutional networks using low-precision and sparsity. *CoRR*, abs/1610.00324, 2016.

[70] Syed Tahir Hussain Rizvi, Gianpiero Cabodi, and Gianluca Francini. Optimized deep neural networks for real-time object classification on embedded gpus. *Applied Sciences*, 7(8), 2017.

[71] Chao Liu, Janki Bhimani, and Miriam Leeser. Using high level gpu tasks to explore memory and communications options on heterogeneous platforms. In *Proceedings of the 2017 Workshop on Software Engineering Methods for Parallel and High Performance Applications*, SEM4HPC '17, pages 21–28, New York, NY, USA, 2017. ACM.

[72] K. Balhaf, M. A. Alsmirat, M. Al-Ayyoub, Y. Jararweh, and M. A. Shehab. Accelerating levenshtein and damerau edit distance algorithms using gpu with unified memory. In *2017 8th International Conference on Information and Communication Systems (ICICS)*, pages 7–11, April 2017.

[73] A. Barberis, G. Danese, F. Leporati, A. Plaza, and E. Torti. Real-time implementation of the vertex component analysis algorithm on gpus. *IEEE Geoscience and Remote Sensing Letters*, 10(2):251–255, March 2013.

[74] S. Bernabé, S. Sánchez, A. Plaza, S. López, J. A. Benediktsson, and R. Sarmiento. Hyperspectral unmixing on gpus and multi-core processors: A comparison. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 6(3):1386–1398, June 2013.

[75] Y. Yang, M. Feng, and S. Chakradhar. Hppcnn: A high-performance, portable deep-learning library for gpgpus. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 582–587, Aug 2016.

[76] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka. Tensor contractions with extended blas kernels on cpu and gpu. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 193–202, Dec 2016.

[77] He Ma, Fei Mao, and Graham W. Taylor. *Theano-MPI: A Theano-Based Distributed Training Framework.* Springer International Publishing, Cham, 2017.

[78] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed Precision Training. *ArXiv e-prints*, October 2017.

[79] IEEE Std 754-2008. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[80] Nhut-Minh Ho and Weng-Fai Wong. Exploiting half precision arithmetic in nvidia gpus. In *21st International Conference on High Performance Extreme Computing Conference*, September 2017.

[81] Y. Qian, M. Bi, T. Tan, and K. Yu. Very deep convolutional neural networks for noise robust speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(12):2263–2276, Dec 2016.

[82] Peter Christiansen, Lars N. Nielsen, Kim A. Steen, Rasmus N. Jørgensen, and Henrik Karstoft. Deepanomaly: Combining background subtraction and deep learning for detecting obstacles and anomalies in an agricultural field. *Sensors*, 16(11), 2016.

[83] Syed Tahir Hussain Rizvi, Denis Patti, Tomas Björklund, Gianpiero Cabodi, and Gianluca Francini. Deep classifiers-based license plate detection, localization and recognition on gpu-powered mobile platform. *Future Internet*, 9(4), 2017.

[84] S. Lawlor, T. Sider, N. Eluru, M. Hatzopoulou, and M. G. Rabbat. Detecting convoys using license plate recognition data. *IEEE Transactions on Signal and Information Processing over Networks*, 2(3):391–405, Sept 2016.

[85] A. Suryatali and V. B. Dharmadhikari. Computer vision based vehicle detection for toll collection system using embedded linux. In *2015 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015]*, pages 1–7, March 2015.

[86] Y. Yuan, W. Zou, Y. Zhao, X. Wang, X. Hu, and N. Komodakis. A robust and efficient approach to license plate detection. *IEEE Transactions on Image Processing*, 26(3):1102–1114, March 2017.

[87] S. Issaoui, R. Ejbeli, T. Frikha, and M. Abid. Embedded approach for edge recognition: Case study: Vehicle registration plate recognition. In *2016 13th International Multi-Conference on Systems, Signals Devices (SSD)*, pages 336–341, March 2016.

[88] C. Gou, K. Wang, Y. Yao, and Z. Li. Vehicle license plate recognition based on extremal regions and restricted boltzmann machines. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1096–1107, April 2016.

[89] M. Wafy and A. M. M. Madbouly. Efficient method for vehicle license plate identification based on learning a morphological feature. *IET Intelligent Transport Systems*, 10(6):389–395, 2016.

[90] G. Lofrano Corneto, F. Assis da Silva, D. R. Pereira, L. L. de Almeida, A. Olivete Artero, J. P. Papa, V. H. Costa de Albuquerque, and H. Molina Sapia. A new method for automatic vehicle license plate detection. *IEEE Latin America Transactions*, 15(1):75–80, Jan 2017.

[91] Y. Chen, D. Zhao, L. Lv, and C. Li. A visual attention based convolutional neural network for image classification. In *2016 12th World Congress on Intelligent Control and Automation (WCICA)*, pages 764–769, June 2016.

[92] J. A. F. Calderon, J. S. Vargas, and A. Pérez-Ruiz. License plate recognition for colombian private vehicles based on an embedded system using the zedboard. In *2016 IEEE Colombian Conference on Robotics and Automation (CCRA)*, pages 1–6, Sept 2016.

[93] S. Lee, K. Son, H. Kim, and J. Park. Car plate recognition based on cnn using embedded system with gpu. In *2017 10th International Conference on Human System Interactions (HSI)*, pages 239–241, July 2017.

[94] X. Luo and M. Xie. Design and realization of embedded license plate recognition system based on dsp. In *2010 Second International Conference on Computer Modeling and Simulation*, volume 2, pages 272–276, Jan 2010.

[95] Edson Cavalcanti Neto, Samuel Luz Gomes, Pedro Pedrosa Rebouças Filho, and Victor Hugo C. de Albuquerque. Brazilian vehicle identification using a new embedded plate recognition system. *Measurement*, 70(Supplement C):36 – 46, 2015.

[96] Hui Li and Chunhua Shen. Reading car license plates using deep convolutional neural networks and lstms. *CoRR*, abs/1601.05610, 2016.

[97] A. Elbamby, E. E. Hemayed, D. Helal, and M. Rehan. Real-time automatic multi-style license plate detection in videos. In *2016 12th International Computer Engineering Conference (ICENCO)*, pages 148–153, Dec 2016.

[98] C. T. Nguyen, T. B. Nguyen, and S. T. Chung. Reliable detection and skew correction method of license plate for ptz camera-based license plate recognition system. In *2015 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1013–1018, Oct 2015.

[99] Platesmania : Largest internet project dedicated to vehicle registration plates. http://platesmania.com/. Accessed: 2017-07-13.

[100] J. A. Belloch, J. M. Badía, F. D. Igual, A. Gonzalez, and E. S. Quintana-Ortí. Optimized fundamental signal processing operations for energy minimization on heterogeneous mobile devices. *IEEE Transactions on Circuits and Systems I: Regular Papers*, PP(99):1–14, 2017.