Low Power MobileNets Acceleration In Cuda And OpenCL

Nikhil Lahoti
San Jose State University

A Writing Project Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

By

Nikhil Lahoti

Spring 2019

The Designated Project Committee Approves the Project Titled

Low Power MobileNets Acceleration In Cuda And OpenCL

By

Nikhil S. Lahoti

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

SPRING 2019

ABSTRACT

Convolutional Neural Network (CNN) has been used widely for the tasks of object recognition and facial recognition because of their remarkable results on these common visual tasks. In order to evaluate the performance of CNN for embedded devices effectively, it is essential to provide a comprehensive benchmark evaluation environment. Even though there are many benchmark suites available for use, but these benchmark suites require installation of various packages and proprietary libraries. This creates a bottleneck in using them in applications which are executed on resource constraint devices like embedded devices.

In this paper, we propose an evaluation platform which can be used for evaluation on any platform that supports Cuda and OpenCL. This evaluation platform was executed on Nvidia TX2 Jetson board embedded device and commodity hardware without needing any extra proprietary libraries to execute the model. We also achieved 4.5-fold gain in execution speed of the Cuda and OpenCL model. The model also exactly predicts images as the Python based with 100% accuracy. We also provide in-depth statistics about the CNN network execution pattern by executing the model on embedded devices and commodity hardware.

Index terms **- CNN, Benchmark Suite, Embedded devices, Cuda, OpenCL, Deep Neural Network**

ACKNOWLEDGEMENT

I would like to express my gratitude to my project advisor Dr. Robert Chun for his support and guidance. I would not have been able to complete this project without Dr. Chun's and Dr. Hyeran Jeon's valuable suggestions. I would also like to thank my committee members Dr. Hyeran Jeon and Dr. Thomas Austin for their suggestions and time.

I am also thankful to my friends and family for all the moral support and constant encouragement.

TABLE OF CONTENTS

TABLE OF FIGURES

I. Introduction

In recent years, the use of machine learning models especially the Deep Neural Network (DNNs) has been the prime research focus area for visual tasks. Convolutional Neural Network (CNNs), a type of DNN, have been able to achieve state of the art performance in common visual tasks like object detection and facial recognition. In the past years, three well-known CNN models – GoogLeNet, AlexNet, and VGGNet, which are all the winners of the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) organized by the Stanford University, have been able to achieve comparable results to a human being in the task of image recognition [1]. Image recognition has long been known to be a very difficult problem to solve for computers and computer systems, but humans can perform these tasks with ease. Such noteworthy results achieved by CNNs have been one of the reasons for their widespread use in all sorts of tasks from number plate detection [2] to car classification [3].

CNNs history goes way back in 1960 when researchers D. H. Hubel and T. N. Wiesel [4] proposed a new model that was inspired by the working of the visual system of mammals. Their model was based on the visual cortex model of cats and monkeys. [4] They showed that the visual system is made up of neurons and that the neurons are arranged in a layered architecture. They also showed that neurons directly respond to the other neurons in their direct environment. In 1980, researcher Fukushima [5] proposed a neural network model called "neocognitron" which was based on hierarchy. The model was able to recognize patterns in the image by identifying the different shapes inside of them. The working of the model was based on simple and complex neurons. Simple neurons are only activated when they see a certain shape or form at certain angles. Complex neurons have a much larger receptive field and hence are not sensitive to such specific requirements. Despite these advancements, none of the models was able to achieve reasonable results in the task of image recognition. In 1986, Yann Le Cun et al [6] were the first to achieve any noteworthy results, when they developed a model with the use of CNNs to recognize handwritten digits. Similar to the “neocognitron” model, this model was also arranged in a hierarchical manner. The problem with this model was its significant computations requirement to process, which at the time was impossible to employ. Since then there has been a lot of focus in finding ways to satisfy the requirements of the CNNs.

In 2012, Alex Krizhevsky et al [14] successfully parallelized the CNN computation on a GPU and were the first to utilize GPU in general purpose computing. They secured the first position in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 and achieved a top-5 error rate of 15.3%, lower by more 10.8 percent than the runner up [23]. Because of these amazing results, every model participating in the ImageNet competition since then have been based on CNN. Since then new CNN model structures have been proposed to make efficient CNN algorithms to make them applicable in day to day activities.

Other focus to make the CNN run efficiently and quickly have been in making better hardware and computer architectures. In order for computer architects to verify the newer computer architectures and optimizations, a comprehensive environment is needed for evaluation. Benchmark suites are commonly used for this purpose. Such benchmark suites have been really common for testing the effectiveness for the CPU architectures. For e.g. C. Cullinan et al [19] from MathWorks and SPEC [20] benchmark is commonly known for testing efficiency of CPU applications in the past.

In the DNN world, such benchmark suites have been coming up regularly[21][22]. These benchmarks, despite being comprehensive, have a few problems of their own. The problem with these frameworks is that they require lot of other third-party installations and packages to function properly (e.g. Keras [24], Tensorflow [25], and PyTorch [26]). Such computation and resource intensive packages cannot be installed on low compute devices like embedded devices and mobile phones due to their low constraints.

In addition to the computation requirement, these benchmarks also work with certain limited number of packages which are proprietary products of institutions and individuals (e.g. CUBLAS [27] and cuDNN [28] of Nvidia). Making optimizations and any modifications to these libraries thus becomes impossible because of the proprietary nature. This puts a limitation for testing newer frameworks and architectures which cannot be installed due to the resource limitations and compatibility issues. Hence there is a need for a new benchmark suite for testing applications, especially on embedded devices which suffer due to the resource requirements of the existing benchmark suites.

In this paper, we provide a new benchmark suite with the MobileNets model which can be executed on any architectures that can execute Cuda and OpenCL. Users can freely use the benchmark to test their applications, modifications and newer architectures to evaluate the results and get valuable information. The model is implemented without using any proprietary code, and hence, modifications to the model code can be done without any problem and any cost.

Section II discusses the history of the hardware implementation that are tried in the past to satisfy CNN requirement. In Section III, we talk about CNN & MobileNets architecture in depth to understand the model. Section IV talks about the implementation details, and in section V we discuss the results and experiments conducted.

II. Hardware Implementations

CNN model has a layered architecture. The output of one layer is then given as an input to the layer ahead of it in the architecture. Each layer of the CNN model performs the same mathematical computations over and over again. For the most part, these operations are a series of dot product operations between the matrices (the weight and the input matrices) to get the result matrix. The operations do not have any dependency within the same layer and hence they can be performed in parallel. To cope with the computation requirements of the CNNs, researchers have attempted to make use of various hardware implementations to satisfy the needs. Following are the approaches:

i) FPGA

Field Programmable Gate Arrays (FPGAs) [5] are semiconductor devices which are arranged in the form of a matrix. They are made up of configurable logic blocks (CLBs) which forms the matrix nodes, and the CLBs are connected via a programmable interconnect. They can be reprogrammed with many hardware languages like VHDL and Verilog to satisfy the needs of the application functionality. A single board can have millions of such CLBs to satisfy the programming needs. The CLBs can be grouped together to solve one set of tasks, while other CLBs can be grouped to solve other set of tasks and so on. This way, all the tasks can run and be executed in parallel. Fig. 1 shows the architecture of an FPGA.

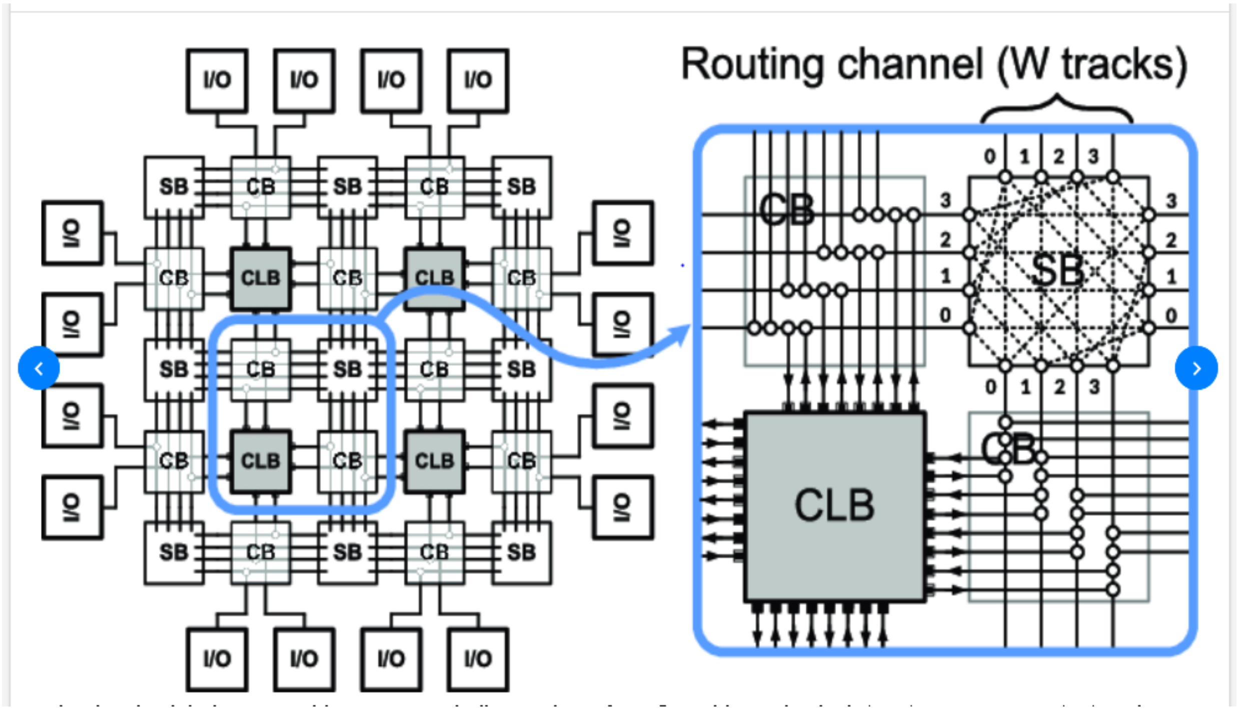


Figure 1: FPGA [11]

In 2002, Z. Nagy et al [11] made use of FPGAs for CNN computations. They used the distributed arithmetic approach which optimized the FPGA architecture and achieved smaller and faster arithmetic units than the conventional approach of multiplier cores and adder trees for state computation of CNN arrays. Another approach of parallelizing using FPGA was used by C. Huang et al [12] in 2017 when they optimized and deployed all the layers of the CNN model in a pipelined architecture on the Xilinx FPGA board. Many such approaches involving the use of FPGA have been proposed for CNN implementation.

ii) GPU

Graphical Processing Units (GPUs) [13] are hardware units which perform rapid mathematical calculations in parallel. They were primarily designed for rendering visuals onto the screen and to offload the computations which were earlier performed on the CPU. A GPU has thousands of single-purpose cores as opposed to the CPU which has limited multipurpose cores.

In 2012, A. Krizhevsky [14] made the first use of GPUs in general purpose computing when they implemented a CNN model using the Cuda parallel platform. Cuda [15] is a parallel computing platform provided by Nvidia corporation to program and run computations on a GPU. After this, all the CNN models which have won the image recognition competitions have all been based on the training on a GPU.

In 2017, S. Oh et al [16] made use of the GPU on the embedded devices to train the CNN models. They were able to achieve about 65% accuracy from the model trained on a GPU than the model trained on a CPU, but the model only consumed about 1.2 % of the total energy used by the CPU model. This was a significant result for the CNN training on a GPU. Below is the architecture of the GPU.

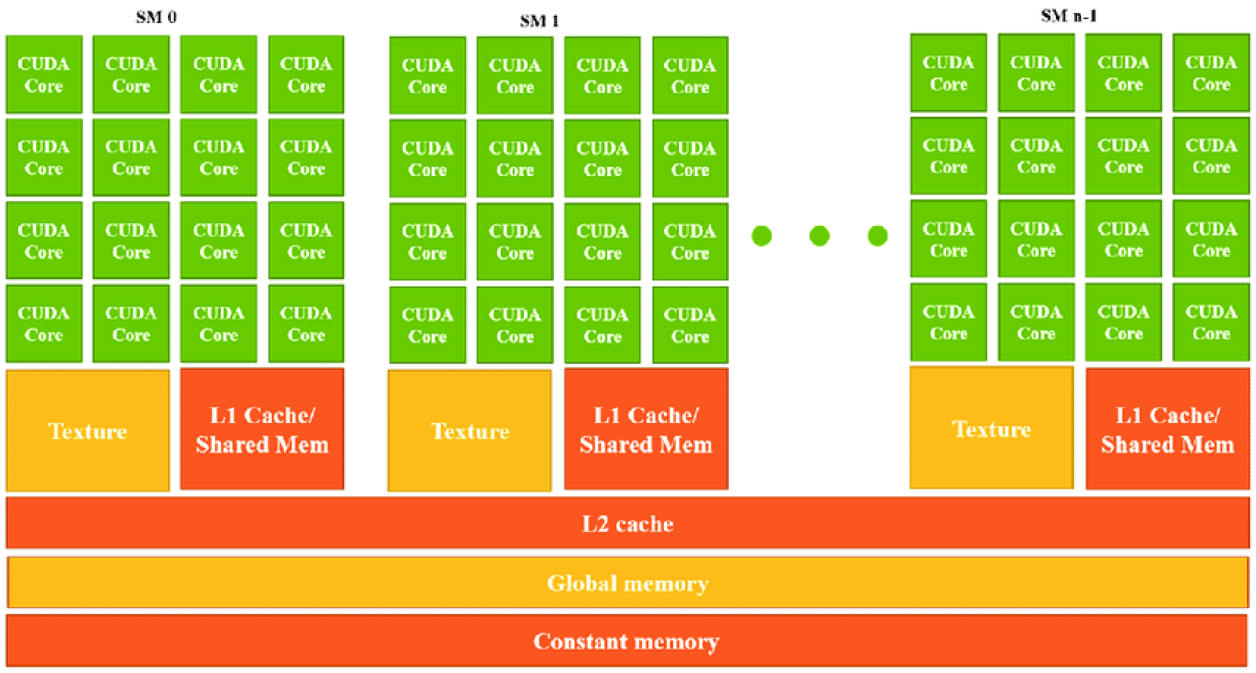


Figure 2: GPU Architecture [13]

In recent years, Graphical Processing Units (GPUs) have been used in training CNN models. GPUs are made up of 1000’s of cores, each of which is capable of working independently and performing mathematical operations in parallel. This is different from the Central Processing Unit (CPU) which have a limited number of cores and which were used in the past for CNN computations. As these operations can be performed in parallel, using a GPU has been incorporated. Embedded devices and mobile phones are generally equipped with a GPU, and hence, advantage can be taken by performing CNN computations on these GPUs.

III. Convolutional Neural Networks

CNNs are a type of Deep Learning model which are made of neurons arranged in the form of layers. This layered architecture is based on the working of the human brain where neurons are connected to one another and pass information amongst them. CNNs are like the deep neural networks except that the neurons in one layer are connected to only a few neurons in the previous layer [7] rather than being connected to all the neurons. Neurons consist of weights and biases which are the learnable parameters of the network. Below is the diagram of the architecture of the convolutional neural network.

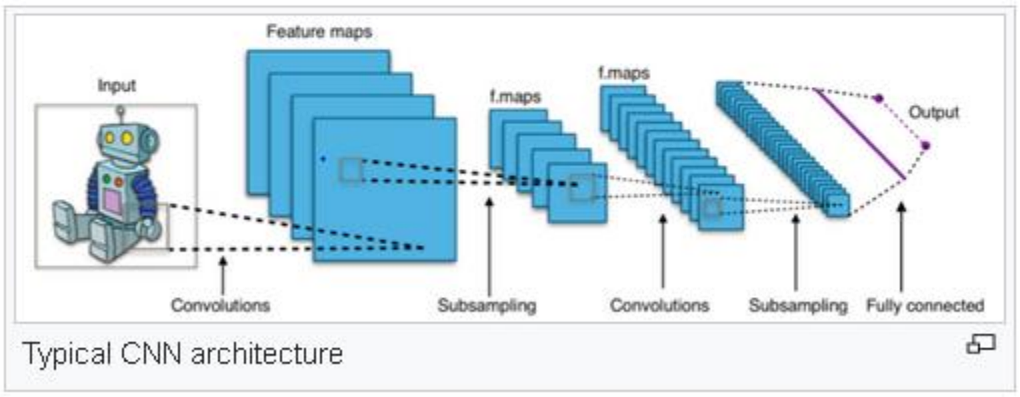


Figure 3: Architecture of CNN[15]

As shown in the diagram, the initial layers are connected to only a few neurons of the previous layer. The architecture also consists of a Pooling and Subsampling layer in the middle which is used to exclude trivial information learned by the network. CNNs usually have a fully connected layer at the end of the network just before the output layer.

MobileNets Architecture

MobileNets are lightweight CNN model developed by A. Howard et al **[**17**]** at Google and was published in 2017. They have achieved tremendous results in spite of the model having substantially fewer parameters than the other well-known model. The MobileNets model [17], has various versions available for use based on the hyperparameters selected by the user. The largest MobileNets model has only 4.2 million parameters as opposed to the VGGNet which has 138 million parameters. A. Howard [17] showed that despite the reductions in the number of parameters, the model performed strongly against the well-known models on the ImageNet classification. MobileNets was able to achieve a top-1 error rate of 70.6 % against the VGGNet which achieved 71.2 % accuracy. The top-5 error rate performance of the MobileNets is 88.4% [17] while that of the VGGNet is 89.2%. The MobileNets model performed well and consumes much less computation power and resources than the other well-known CNN models. Thus, this model perfectly represents the kind of load and computation requirement that can be executed on an embedded device.

Depthwise Separable Convolution lies at the heart of the model and hence we discuss about it in this section.

Depthwise Separable Convolution

Depthwise Separable convolution divides a standard convolution operation into multiple steps, namely depthwise convolution and pointwise convolution. It was first proposed by L. Sifre [18] in this Ph.D. thesis work. A standard convolution applies the convolution operation across spatial and temporal dimension, all in the same step, i.e. the same kernel is used to combine the operations across multiple channels. Depthwise Separable convolution performs a two-step operation. In the first step, it applies a single kernel operation across a single channel called as the depthwise operation (operating on each depth separately), and in the second step, it combines the results from multiple different channels to the final output. Overall, it achieves the same effect on the input matrix but divides it into multiple steps, achieving a tremendous reduction in the number of parameters in the process as we will see later.

A depthwise convolution takes as an input a matrix of size D_F x D_F x M, where D_F is the width and height of the input image and M is the number of channels in the input image (typically there are 3 channels), N kernels of size D_K x D_K is applied to get D_G x D_G output matrix. Each of the N kernels is applied separately on each channel. This is different from the standard convolution where the kernel is applied across all the channels. The operation of the depthwise separable convolution is as shown below:

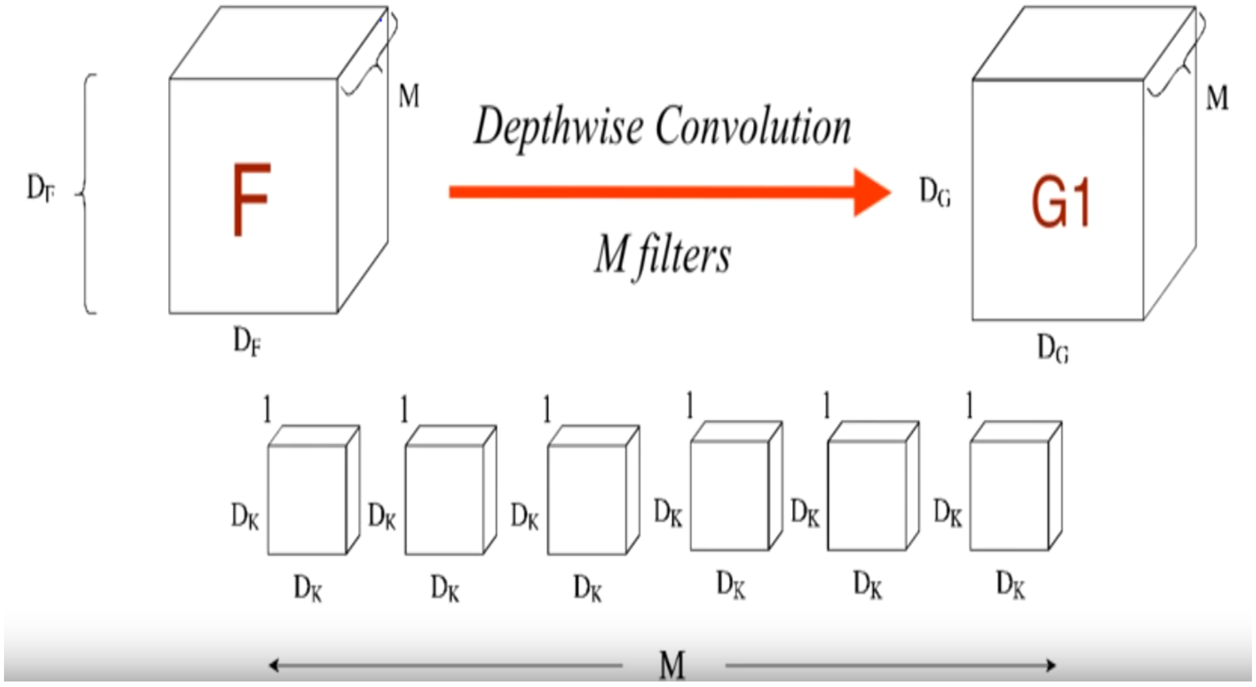


Figure. 4: Depthwise Separable Convolution [29]

Pointwise Separable Convolution

The output of the depthwise phase is given as an input to the pointwise phase. In the pointwise phase, the output from all the channels is combined together to get the resultant matrix. The input is of size D_G x D_G x M and is applied to N kernels each of size 1 x 1 x M to get the output as D_G x D_G x N. The 1 x 1 operation combines different channels and hence the number of channels in the kernel is the same as the number of channels in the input matrix. The entire operation can be seen below.

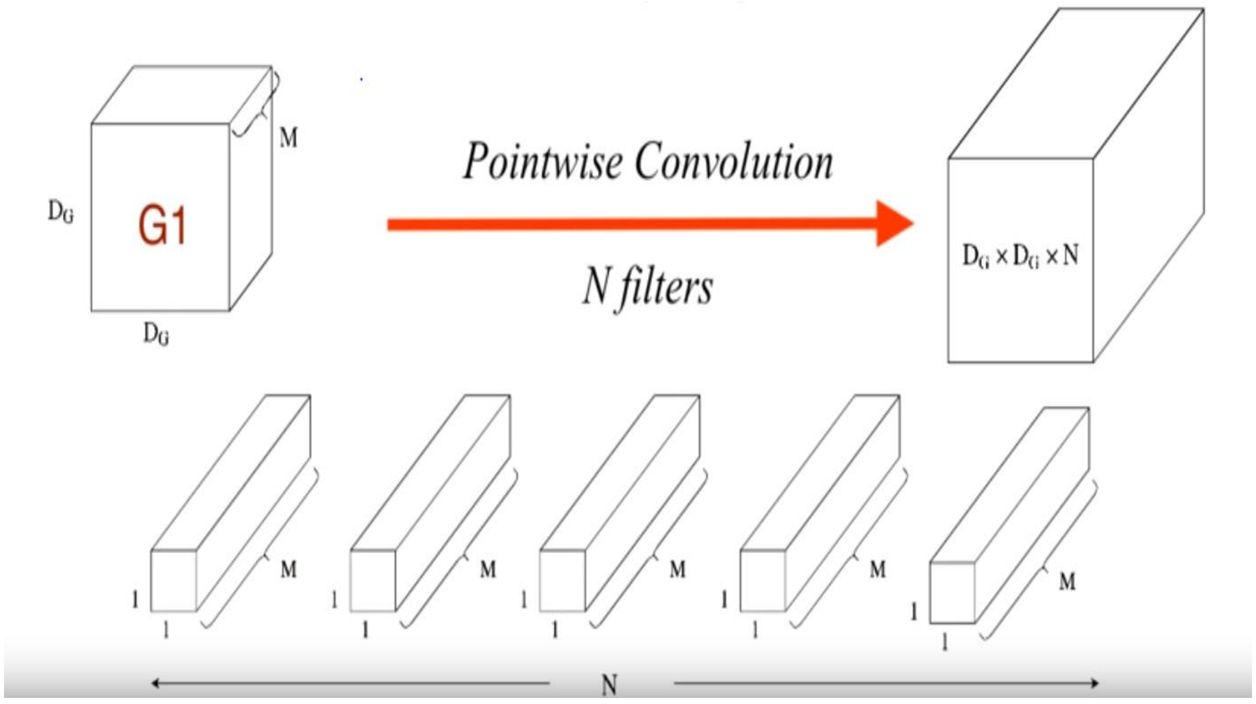


Figure. 5: Pointwise Separable Convolution [29]

Fig 6 shows the comparison between the standard convolution operation and the depthwise separable convolution. One standard convolution is broken down into two steps which is able to significantly reduce the number of parameters needed by the model.

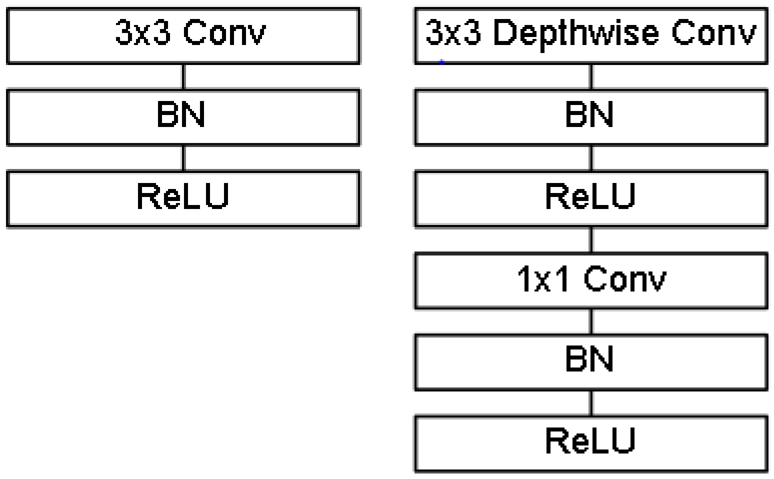
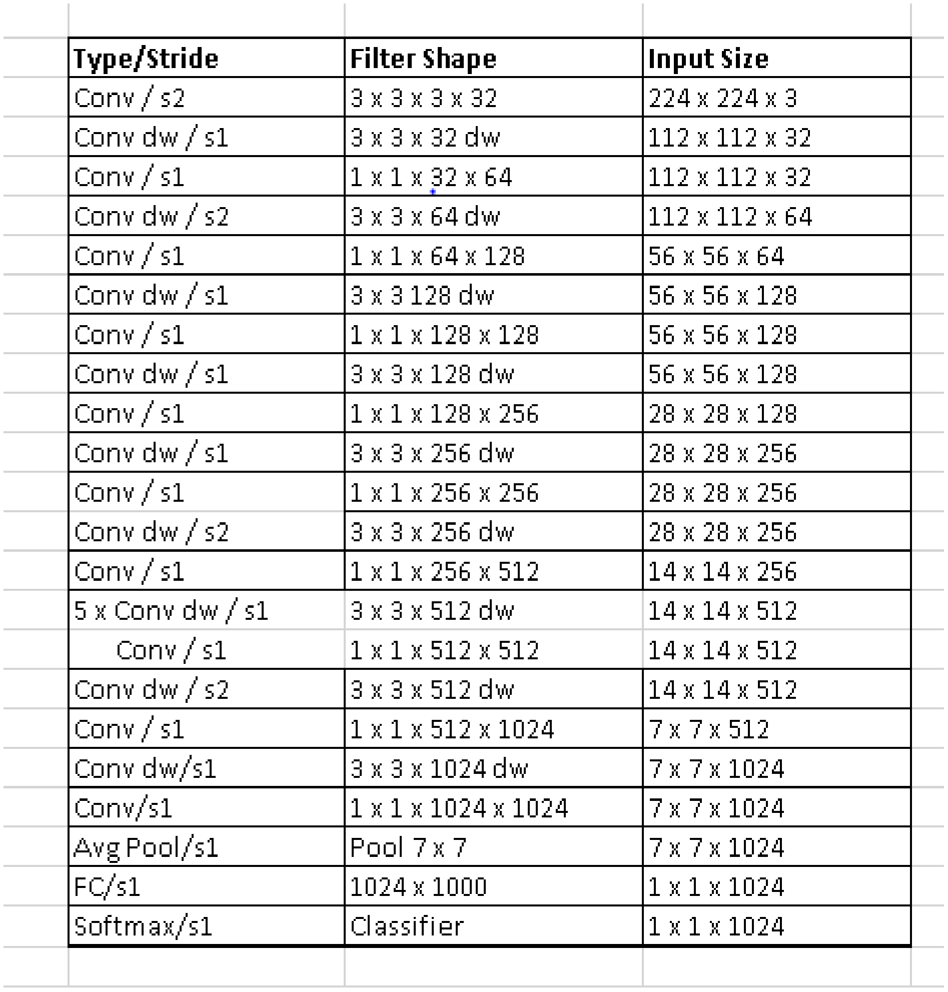


Figure 6: Standard Convolution Vs Depthwise Separable Convolution [17]

The MobileNets model has 28 layers in total. The first and the final layers in the model are the standard convolution fully connected layers where neurons on one layer are connected to all the neurons in the previous layer. The middle layers consist of the depthwise and pointwise convolution layers arranged one after the other. The model also contains a max pooling layer which is used to remove the trivial information that the model has learned so far.

After each pointwise layer, the model also has a batch normalization layer. Batch normalization is used so that each layer can learn the input and the relations a little bit by itself rather than completely relying on the other layers in the network.

The entire MobileNets architecture is as shown in Fig 7. There are in total 28 layers, with 2 Fully connected layers, 13 depthwise and 13 pointwise layers. Each layer also contains the stride parameter which is shown in the first column.



Type/Stride	Filter Shape	Input Size
Conv / s2	3 x 3 x 3 x 32	224 x 224 x 3
Conv dw / s1	3 x 3 x 32 dw	112 x 112 x 32
Conv / s1	1 x 1 x 32 x 64	112 x 112 x 32
Conv dw / s2	3 x 3 x 64 dw	112 x 112 x 64
Conv / s1	1 x 1 x 64 x 128	56 x 56 x 64
Conv dw / s1	3 x 3 128 dw	56 x 56 x 128
Conv / s1	1 x 1 x 128 x 128	56 x 56 x 128
Conv dw / s1	3 x 3 x 128 dw	56 x 56 x 128
Conv / s1	1 x 1 x 128 x 256	28 x 28 x 128
Conv dw / s1	3 x 3 x 256 dw	28 x 28 x 256
Conv / s1	1 x 1 x 256 x 256	28 x 28 x 256
Conv dw / s2	3 x 3 x 256 dw	28 x 28 x 256
Conv / s1	1 x 1 x 256 x 512	14 x 14 x 256
5 x Conv dw / s1	3 x 3 x 512 dw	14 x 14 x 512
Conv / s1	1 x 1 x 512 x 512	14 x 14 x 512
Conv dw / s2	3 x 3 x 512 dw	14 x 14 x 512
Conv / s1	1 x 1 x 512 x 1024	7 x 7 x 512
Conv dw/s1	3 x 3 x 1024 dw	7 x 7 x 1024
Conv/s1	1 x 1 x 1024 x 1024	7 x 7 x 1024
Avg Pool/s1	Pool 7 x 7	7 x 7 x 1024
FC/s1	1024 x 1000	1 x 1 x 1024
Softmax/s1	Classifier	1 x 1 x 1024

Figure 7: MobileNets Architecture [17]

VI. Implementation

A) Implementation in Cuda

All the 28 layers of the MobileNets model are implemented in Cuda. Each layer requires the values for its weight matrices in order to correctly predict the output. For this purpose, we collected the weight matrix values into separate files, and these weight files are partitioned into their appropriate folders. E.g. all the weight files required by the first layer are stored at data/FirstLayer path relative to the root folder. The input files include the weight and batch normalization parameter files (One input file each for Standard Deviation, Epsilon, Beta & mean).

Each Cuda kernel is given an input of the above-mentioned parameters. For each output value, we launch one thread which processes that portion of the output matrices. For the most part, each layer in the model is launched with one kernel. However, we do have certain kernel layers for which launching the required amount of thread is not possible because of the limitation in the maximum amount of threads. To process such kernels, we divide our layer into multiple kernels. For e.g. for the first layer of the model which is a fully connected layer, the input of this layer is of size (224 x 224 x 3) (the input image), and we perform a 3D Convolution with the filter size of (3 x 3 x 3) and having 32 such filters giving an output of (114 x 114 x 32). Hence, launching kernels of size (114 x 114 x 32) is not possible in a single go. We, therefore, divide the layer processing into 3 kernels, each processing a different part of the output matrix as shown below. The blue part covers portion from 0-96 on x-axis and 0-96 on y-axis, the grey from 96 -112 on the x-axis and 0 – 112 on the y-axis, and the green from 0 - 96 on the x-axis and 96 – 112 on the y-axis. The output for the next layer needs to perform padding on the input matrix, so we decide to handle the padding in this layer itself improving the performance and reducing the effort to relaunch new kernels to achieve the same operation and results.

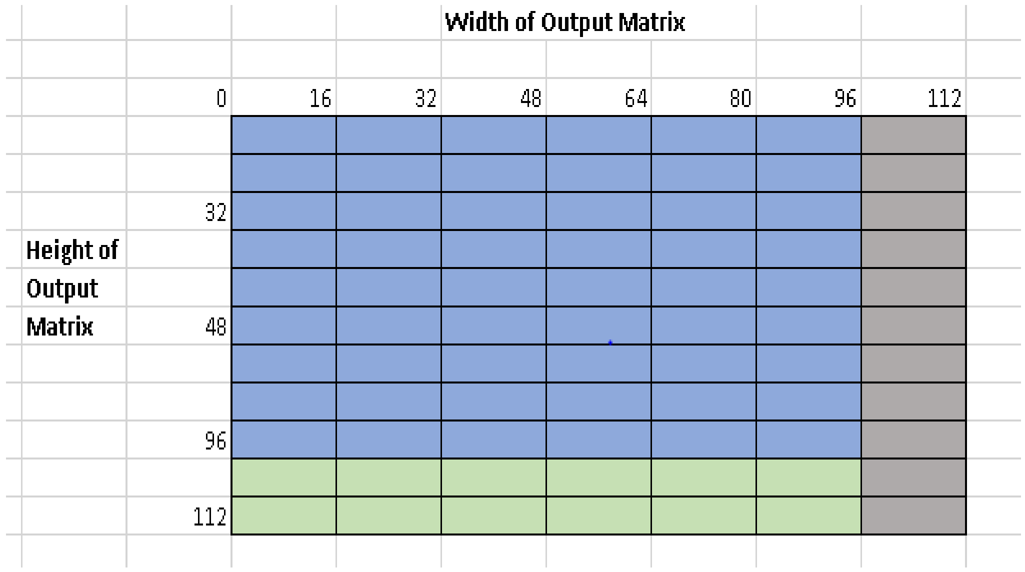


Figure 8: Kernel Execution for First Layer

The reason for launching multiple kernels for processing a single layer was not only due to the maximum thread limitation but was also to get the best performance of the underlying architecture and following the best-known practices. It is commonly known for each kernel to have threads in a multiple of 32 to get the best performance of the Nvidia Cuda architecture due to the reason that threads in Nvidia Cuda architecture are launched in hardware in batches of 32. Thus, to get the maximum performance and optimize our implementation to give the best results, we also divided certain layers into multiple kernel calls. For example. consider the fourth kernel which has an input of size (113 x 113 x 64) with the 2-dimensional convolution of a filter of size (3 x 3) with 64 such filters, giving an output of size (56 x 56 x 64). We divide the kernel call into 4 parts each having the thread count in the multiple of 32 to get the best performance. The execution pattern followed for the layer is as mentioned below in Fig. 9. As mentioned before, we manage the padding code in this layer itself to get the matrix right for the next layer.

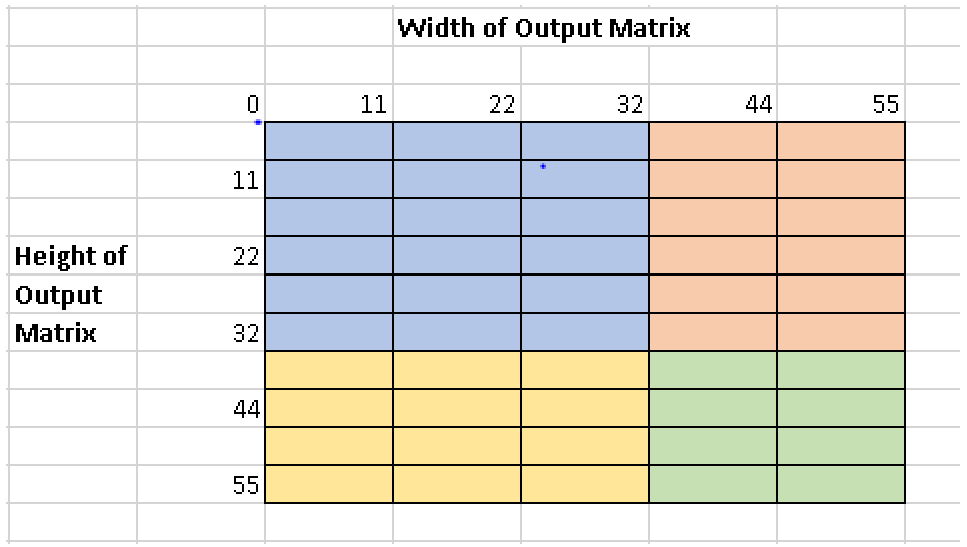


Figure 9: Kernel Execution for Fourth Layer

The number of kernel call divisions for each layer are shown in the Fig. 10 below:

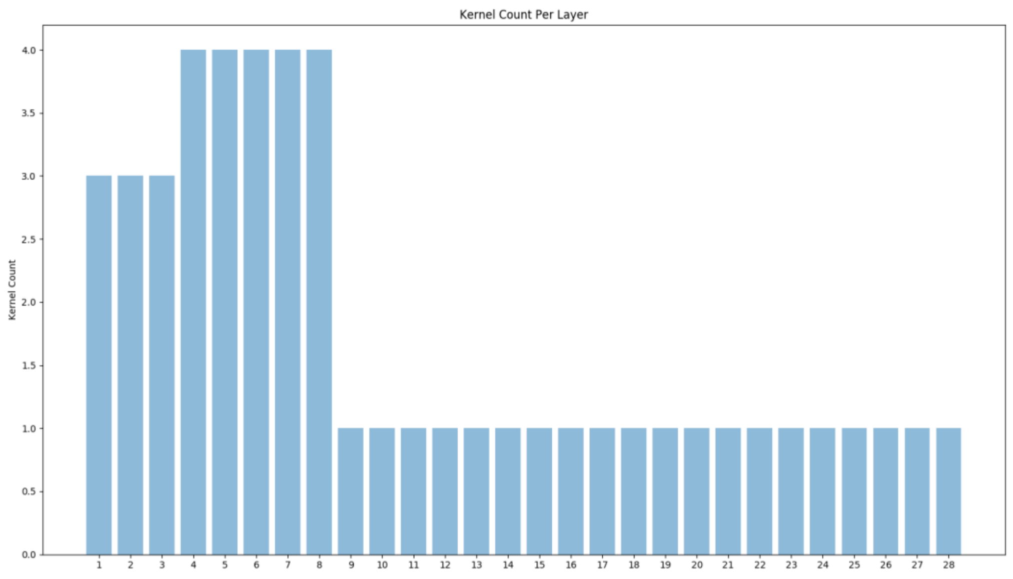


Figure 10. Kernel Calls per Layer

Threads in Cuda and other parallel programming platforms are launched in grids and blocks. Similar to the construct of a single process in a CPU, where threads within a single process share resources and communicate with each other, threads within the same block are also able to share resources and are able to communicate with one another. Blocks of threads are launched in the form of a grid. Launching the threads in the grid and block format helps in indexing the threads really effectively. The Cuda kernel configurations of each individual layer such as grid dimension and block dimension are also shown below.

Table 1. Grid and Block Dimensions for Cuda model

	Grid Dimensions	***Block Dimensions***
First Layer – A	(32, 3, 3)	(32, 32)
First Layer – B	(32, 7)	(16, 16)
First Layer – C	(32, 6)	(16, 16)
Second Layer – A	(32, 3, 3)	(32, 32)
Second Layer – B	(32, 7)	(16, 16)
Second Layer – C	(32, 6)	(16, 16)
Third Layer – A	(64, 3, 3)	(32, 32)
Third Layer – B	(64, 7)	(16, 16)
Third Layer – C	(64, 6)	(16, 16)
Fourth Layer – A	(64, 1, 1)	(32, 32)
Fourth Layer – B	(64, 1, 1)	(32, 24)
Fourth Layer – C	(64, 1, 1)	(24, 32)
Fourth Layer – D	(64, 1, 1)	(24, 24)
Fifth Layer – A	(128, 1,1)	(32, 32)
Fifth Layer – B	(128, 1,1)	(32, 24)
Fifth Layer – C	(128, 1,1)	(24, 32)
Fifth Layer – D	(128, 1,1)	(24, 24)
Sixth Layer – A	(128, 1,1)	(32, 32)
Sixth Layer – B	(128, 1,1)	(32, 24)
Sixth Layer – C	(128, 1,1)	(24, 32)

Sixth Layer – D	(128, 1,1)	(24, 24)
Seventh Layer – A	(128, 1,1)	(32, 32)
Seventh Layer – B	(128, 1,1)	(32, 24)
Seventh Layer – C	(128, 1,1)	(24, 32)
Seventh Layer – D	(128, 1,1)	(24, 24)
Eighth Layer	(128, 1,1)	(28, 28)
Ninth Layer	(256, 1,1)	(28, 28)
Tenth Layer	(256, 1,1)	(28, 28)
Eleventh Layer	(256, 1,1)	(28, 28)
Twelfth Layer	(256, 1,1)	(14, 14)
Thirteenth Layer	(512, 1,1)	(14, 14)
Fourteenth Layer	(512, 1,1)	(14, 14)
Fifteenth Layer	(512, 1,1)	(14, 14)
Sixteenth Layer	(512, 1,1)	(14, 14)
Seventeenth Layer	(512, 1,1)	(14, 14)
Eighteenth Layer	(512, 1,1)	(14, 14)
Nineteenth Layer	(512, 1,1)	(14, 14)
Twenty Layer	(512, 1,1)	(14, 14)
Twenty-One Layer	(512, 1,1)	(14, 14)
Twenty-Two Layer	(512, 1,1)	(14, 14)
Twenty-Three Layer	(512, 1,1)	(14, 14)
Twenty-Four Layer	(512, 1,1)	(14, 14)
Twenty-Five Layer	(1024, 1,1)	(7,7)
Twenty-Six Layer	(1024, 1,1)	(7,7)
Twenty-Seven Layer	(1024, 1,1)	(7,7)
Twenty-Eight Layer	(1, 1,1)	(32,32)

The operations were performed on three different types of hardware. The configurations of the three hardware's are as shown below in Table 2. The three hardware include a GPU server, a commodity hardware laptop, and an embedded Nvidia Jetson board hardware unit.

Table 2: GPU Architectures

Parameters	Nvidia Jetson board	Commodity Hardware	GPU Server
Architectures	Tegra	Maxwell	Kepler
Global Memory	4 GB	8 GB	24 GB
Cuda Cores	256	640	2880
Shared/L1D	48KB	64KB per block	128 KB per block
OS	Ubuntu 14.04.1 Intel Xeon E52623	Ubuntu 14.04.3 LTS ARM Cortex – A57	Ubuntu 16.04 LTS

B) Implementation in OpenCL

Similar to the Cuda model, kernels in the OpenCL model are also launched in blocks and grids of threads. But, the way in which these blocks and grids of threads is defined are different from the Cuda version. We also took efficiency into consideration, and similar to the Cuda version launched the layer operations in multiple kernels wherever required. Following are the grid and block dimensions used in the OpenCL version.

Table 3: Grid and Block dimension OpenCL model

	Grid Dimensions	***Block Dimensions***

First Layer – A	(32, 96, 96)	(1,32,32)
First Layer – B	(32, 112, 16)	(1,16, 16)
First Layer – C	(32, 96, 16)	(1,16, 16)
Second Layer – A	(32, 96, 96)	(1,32,32)
Second Layer – B	(32, 112, 16)	(1,16, 16)
Second Layer – C	(32, 96, 16)	(1,16, 16)
Third Layer – A	(64, 96, 96)	(1,32, 32)
Third Layer – B	(64, 112, 16)	(1,16, 16)
Third Layer – C	(64, 96, 16)	(1,16, 16)
Fourth Layer – A	(64, 32, 32)	(1,32, 32)
Fourth Layer – B	(64, 32, 24)	(1,32, 24)
Fourth Layer – C	(64, 24, 32)	(1,24, 32)
Fourth Layer – D	(64, 24, 24)	(1,24, 24)
Fifth Layer – A	(128, 32, 32)	(1,32, 32)
Fifth Layer – B	(128, 32, 24)	(1,32, 24)
Fifth Layer – C	(128, 24, 32)	(1,24, 32)
Fifth Layer – D	(128, 24, 24)	(1,24, 24)
Sixth Layer – A	(128, 32, 32)	(1,32, 32)
Sixth Layer – B	(128, 32, 24)	(1,32, 24)
Sixth Layer – C	(128, 24, 32)	(1,24, 32)
Sixth Layer – D	(128, 24, 24)	(1,24, 24)
Seventh Layer – A	(128, 32, 32)	(1,32, 32)
Seventh Layer – B	(128, 32, 24)	(1,32, 24)
Seventh Layer – C	(128, 24, 32)	(1,24, 32)
Seventh Layer – D	(128, 24, 24)	(1,24, 24)
Eighth Layer	(128, 28, 28)	(1,28, 28)
Ninth Layer	(256, 28, 28)	(1,28, 28)
Tenth Layer	(256, 28, 28)	(1,28, 28)
Eleventh Layer	(256, 28, 28)	(1,28, 28)
Twelfth Layer	(256, 14,14)	(1,14, 14)
Thirteenth Layer	(512, 14,14)	(1,14, 14)
Fourteenth Layer	(512, 14,14)	(1,14, 14)
Fifteenth Layer	(512, 14,14)	(1,14, 14)

Sixteenth Layer	(512, 14,14)	(1,14, 14)
Seventeenth Layer	(512, 14,14)	(1,14, 14)
Eighteenth Layer	(512, 14,14)	(1,14, 14)
Nineteenth Layer	(512, 14,14)	(1,14, 14)
Twenty Layer	(512, 14,14)	(1,14, 14)
Twenty-One Layer	(512, 14,14)	(1,14, 14)
Twenty-Two Layer	(512, 14,14)	(1,14, 14)
Twenty-Three Layer	(512, 14,14)	(1,14, 14)
Twenty-Four Layer	(512, 14,14)	(1,14, 14)
Twenty-Five Layer	(1024, 7, 7)	(1,7, 7)
Twenty-Six Layer	(1024, 7, 7)	(1,7, 7)
Twenty-Seven Layer	(1024, 7, 7)	(1,7, 7)
Twenty-Eight Layer	(1, 32,32)	(1,32,32)

VI. Experiments

A. Experimental Setup

Following are the steps for the software setup of the experiment

1. Install gcc compiler for compiling c programs
2. Install Python
3. Install Anaconda [https://www.anaconda.com/download/] and add Conda to the global path variable
4. Install Nvidia Cuda [https://developer.nvidia.com/]
5. Create a new Anaconda virtual environment

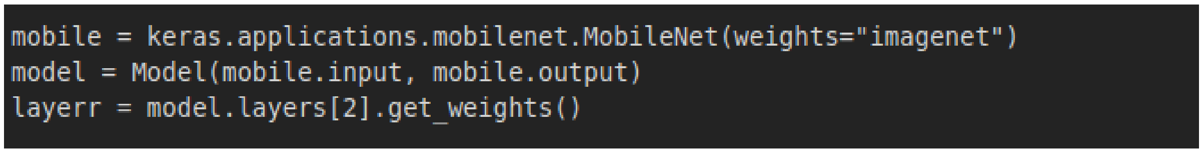
   a. conda create -n env_name
   b. activate env_name
   c. pip install –ignore-installed –upgrade tensorflow / tensorflow-gpu
   d. pip install –ignore-installed –upgrade keras

6. Following packages should be installed when installing anaconda. If they are not installed, you can install using the following commands
   a. Numpy: pip install numpy
   b. Pandas: pip install pandas
   c. Jupyter Notebook: pip install Jupyter-notebook

B. Preprocessing

The input to the model is an image of size (224 x 224 x 3). We convert our input image in Python and then save that as text files which are then read in our Cuda and OpenCL models. In order for our model to accurately predict, we normalize the input image. The process of normalization is a trivial task in Python and hence we decided to do the task in Python. We make use of the Keras and Numpy packages to achieve this task.

The Cuda and OpenCL models also need the learned weights and biases of the MobileNets which contains all the information needed by the model to recognize the objects from the input image. To get these weights, we use the model from the Keras package. Keras package provides a way to save these weights in text files and similar to the input image, these weights are then read by our Cuda and OpenCL models. Below is the syntax of the code to get the weights from the Keras model:



```
mobile = keras.applications.mobilenet.MobileNet(weights="imagenet")
model = Model(mobile.input, mobile.output)
layerr = model.layers[2].get_weights()
```

Figure 11. Saving Keras Weights

Similar to the weight and biases, the model also needs the batch normalization parameters to be saved in text file. These parameters are also saved into text files and are given as input to the models.

C. Experiment

As a part of our implementation, we decided to work with 5 different images of common objects and things to compare the accuracy of both the models in predicting the output. We decided to work with image of a Car, a Cat, a Coffee, a Beagle Dog and a Lizard to compare the predictions. Only the top 5 predictions for all the models are considered as we believe it is a very fair comparison.

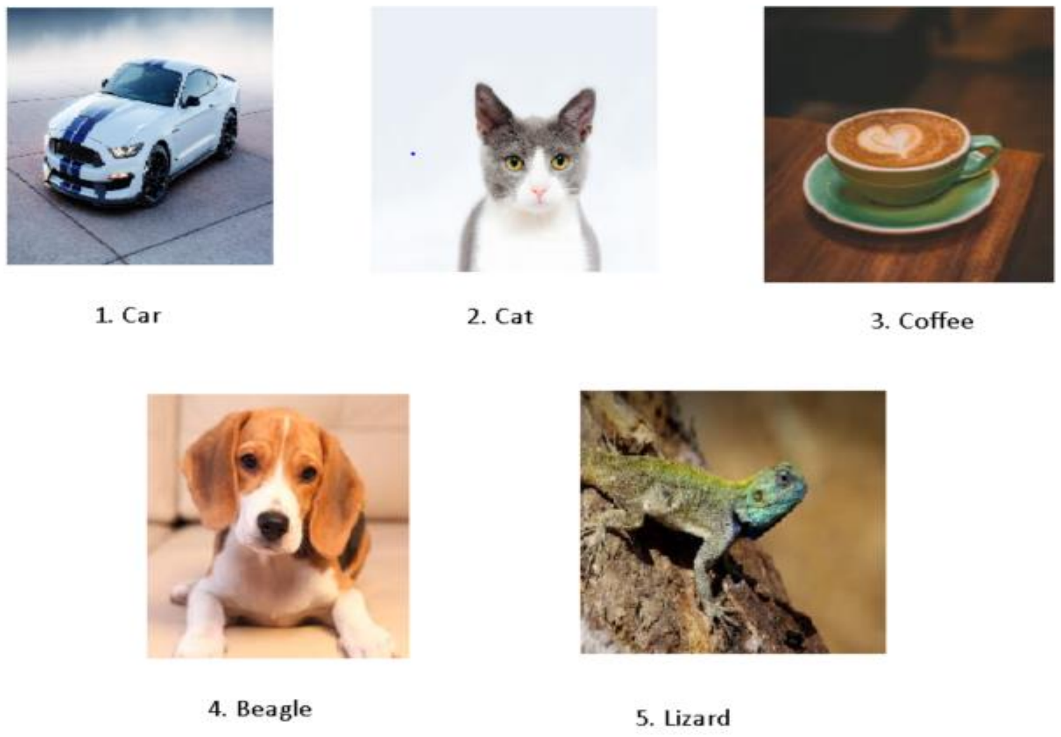


Figure 12. Experiment Images

i) Predictions of Cuda model

Table 4: Python Vs Cuda Model Prediction

	Python Code	Cuda Code
Beagle Dog	1 beagle 2 English_foxhound 3 Walker_hound 4 redbone 5 bluetick	1 beagle 2 English_foxhound 3 Walker_hound 4 redbone 5 bluetick
Car	1 convertible 2 sports_car 3 car_wheel 4 minivan 5 grille	1 convertible 2 sports_car 3 car_wheel 4 minivan 5 grille

Cat	1 tabby 2 nipple 3 Egyptian_cat 4 bow_tie 5 tiger_cat	1 tabby 2 nipple 3 Egyptian_cat 4 bow_tie 5 tiger_cat
Coffee	1 cup 2 espresso 3 consomme 4 eggnog 5 strainer	1 cup 2 espresso 3 consomme 4 eggnog 5 strainer
Lizard	1 agama 2 banded_gecko 3 African_chameleon 4 whiptail 5 frilled_lizard	1 agama 2 banded_gecko 3 African_chameleon 4 whiptail 5 frilled_lizard

As we can see from the table above, our Cuda model is able to predict the top-5 labels with 100% accuracy. The results match exactly with the keras based Python model as can be seen from the table 4.

Performance Analysis

In order for our model to represent effective benchmark standards, we measure various statistics of the model. The comparison of different statistics is as shown in the Fig. 12.

a) Execution Time comparison

We compared the execution time of our Cuda model against the Python model to check if the model matches the execution time of the currently used industry models. The time taken by each model against the five different images is as shown below in Fig. 13. As we can see, the Cuda model takes approximately 5 times less time to execute the Python model.

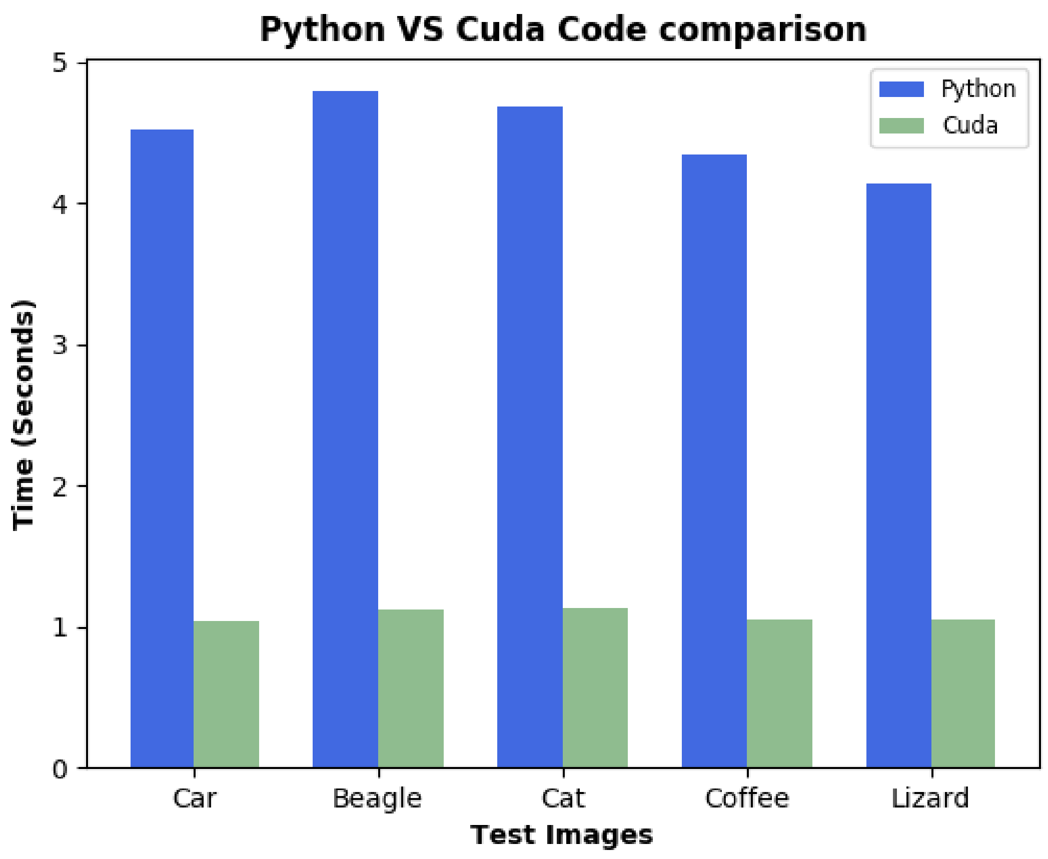


Figure 13: Timing comparison between Python and Cuda models

b) Overall Performance

To measure the performance of the model, we measure the performance of the different types of operations performed in each layer. The percentage of the time taken to perform each operation is shown below in Fig 14. As we can see from the result, most of the time is taken by the Pointwise Separable Convolution (PSC) layer with around 94.7% of the total time. This result of the operations correctly match with the results of the model as proposed by A. Howard et al [17] where the major focus was to shift the overall weight and calculations on the Pointwise Separable Convolution layers while keeping the other part of the network compact.

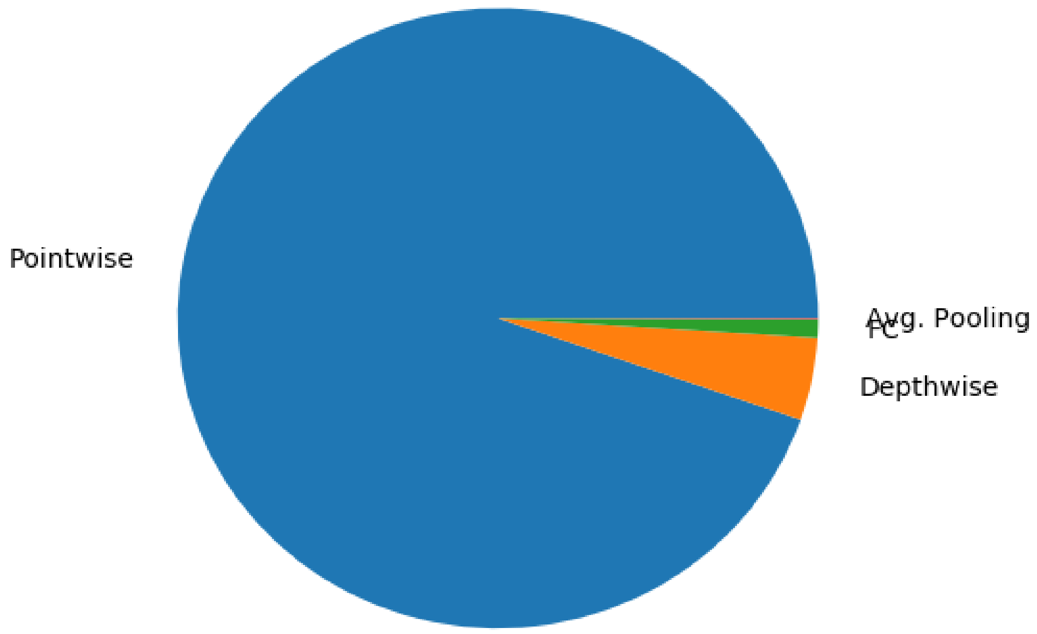


Figure 14. Percentage of the total operations

c) **Memory Stall Reason**

To measure the performance bottleneck in the model execution, we calculated the memory stalls reasons for each layer's execution. We used nvprof [30] profiler provided by Nvidia and the results were executed on the Nvidia 960M Maxwell architecture GPU.

Memory stalls help in identifying the performance bottlenecks caused in the execution of the different types of operations like standard convolution, depthwise convolution, and pointwise convolution. Memory stall information can be used by the computer architects to improve upon the performance of the CNN network like developing a specialized highly streamlined application. The profiler provides memory stall information for many issues, but we only highlight the significant operations only.

Fig. 16 shows the memory stall reasons for the pointwise separable convolution layers. Most of the memory stall reasons are due to Other factors and Data Request. Other factors are problems that are caused due to the compiler or hardware reasons. A developer does not have any control over these stall problems.

As pointwise apply convolution on each channel of the input matrix, it is understandable that the Data Request would take significant time. The GPU needs to fetch the data from each channel and hence as expected constituted most of the memory stall reason.

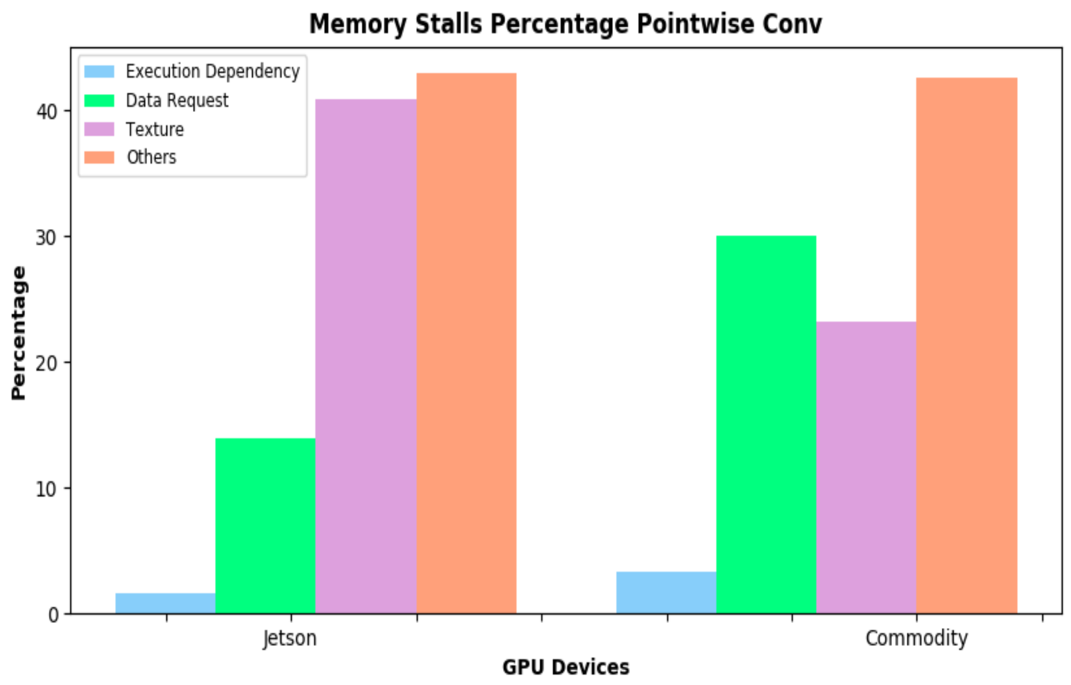


Figure 15. Memory Stall Reason for Pointwise Convolution

Fig. 17 shows the memory stall reasons for the depthwise convolutions. Most of the stall issue with depthwise convolutions is related to the Data Dependency issue and Other reasons. This result matches with the expected outcome because depthwise operations are performed on the same input channels and hence there can be a lot of dependencies to perform the operation between different kernel threads. As all the kernels operate in the same channel, Data Request stall reason is very low for the depthwise convolution layers.

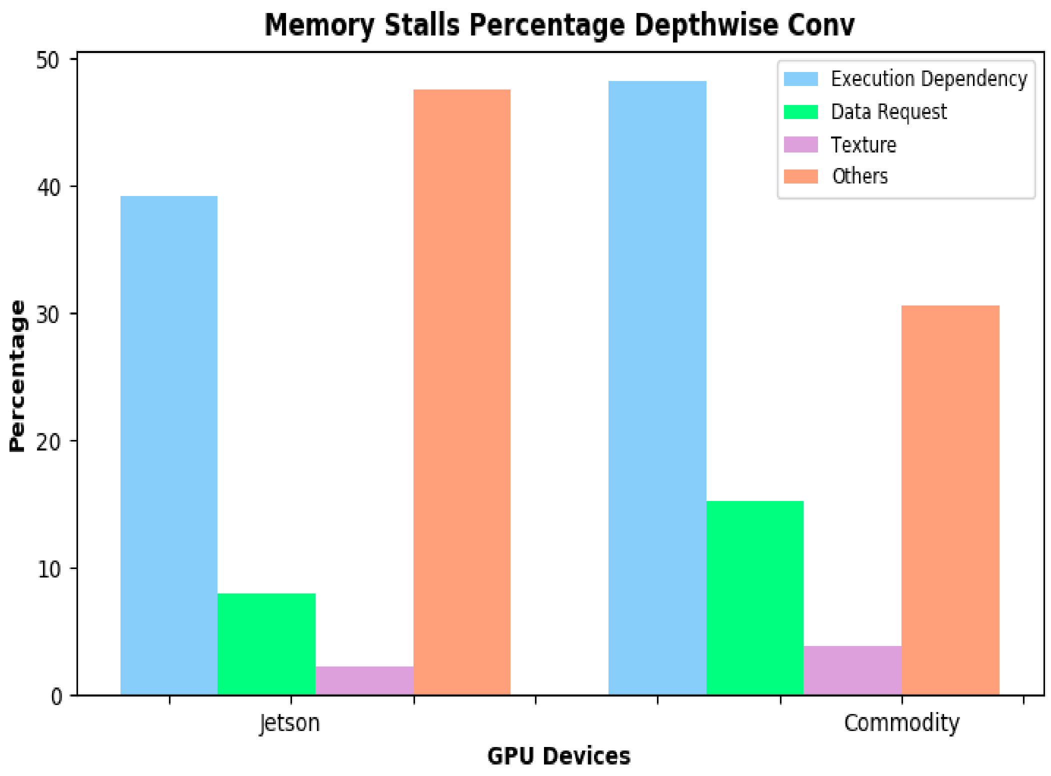


Figure 16: Memory Stall Reason for Depthwise Conv

The memory stall for the Average pooling layer is as shown in Fig. 18. Most of the issues related to Avg. Pooling layer is related to the Texture and Other reasons, while Data Request and Execution Dependency constitutes minimal portion.

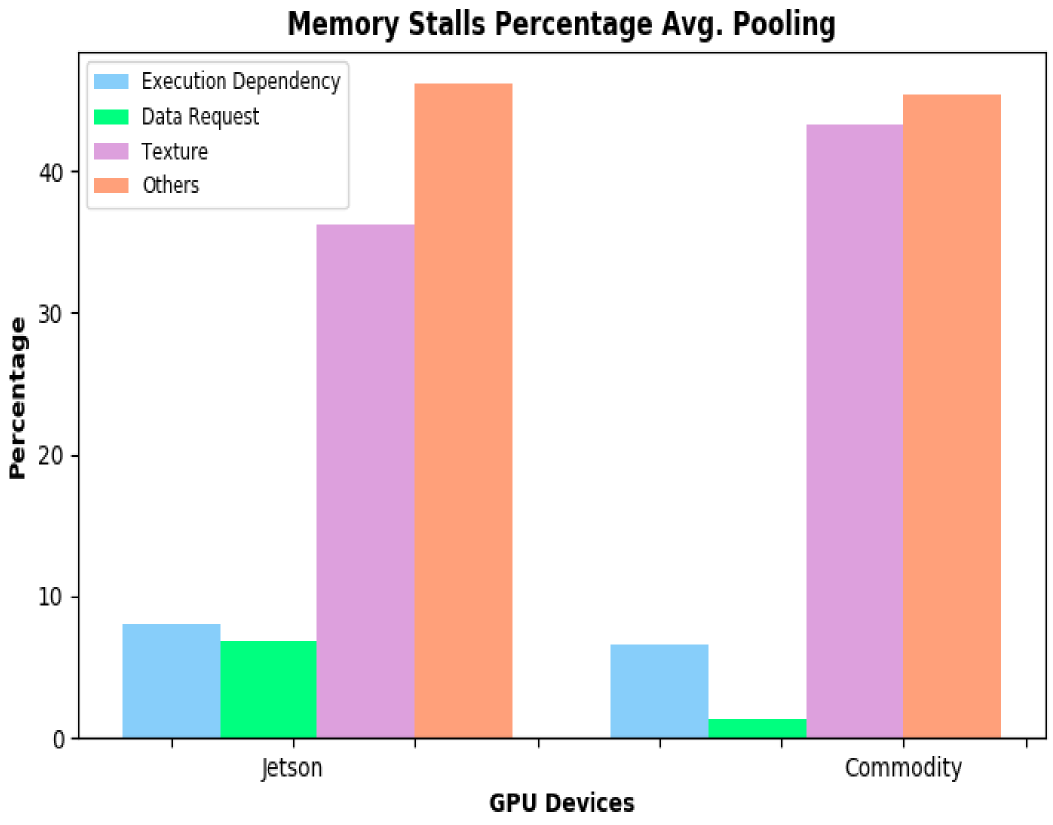


Figure 17: Memory Stall Reason Average Pooling Operation

d) Instruction Breakdown

We also collected the information regarding the instructions that take the most time to execute by the CNN model. This information is shown in the Fig. 19.

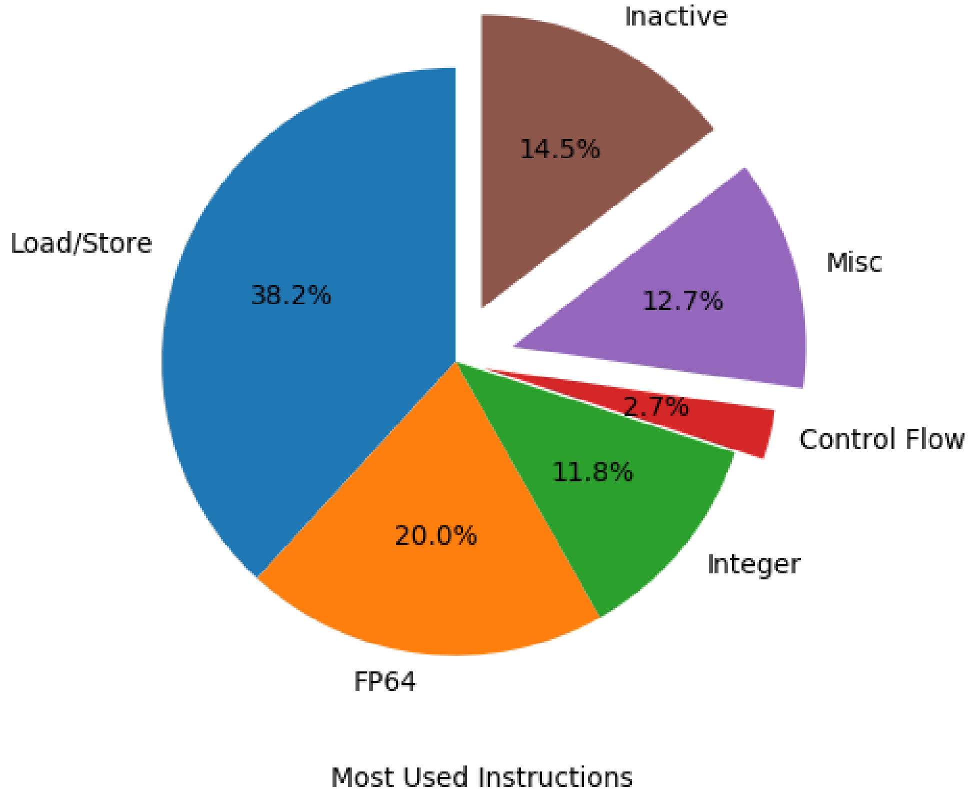


Figure 18: Execution Time of Instructions

ii) Predictions of OpenCL model

The OpenCL model, similar to the Cuda model was also tested with the Python model. We were able to achieve similar results for where the top-5 predictions for both matched exactly. Predictions of the OpenCL model are as shown in the Table 5 below.

Table 5: Python Vs OpenCL Model Prediction

	Python Code	OpenCL Code
Beagle Dog	1 beagle 2 English_foxhound 3 Walker_hound 4 redbone 5 bluetick	1 beagle 2 English_foxhound 3 Walker_hound 4 redbone 5 bluetick
Car	1 convertible 2 sports_car 3 car_wheel 4 minivan 5 grille	1 convertible 2 sports_car 3 car_wheel 4 minivan 5 grille
Cat	1 tabby 2 nipple 3 Egyptian_cat 4 bow_tie 5 tiger_cat	1 tabby 2 nipple 3 Egyptian_cat 4 bow_tie 5 tiger_cat
Coffee	1 cup 2 espresso 3 consomme 4 eggnog 5 strainer	1 cup 2 espresso 3 consomme 4 eggnog 5 strainer
Lizard	1 agama 2 banded_gecko 3 African_chameleon 4 whiptail 5 frilled_lizard	1 agama 2 banded_gecko 3 African_chameleon 4 whiptail 5 frilled_lizard

Performance Analysis

a) Execution Time comparison

The Fig. 20 below shows the comparison between the Python and the OpenCL model. As we can see, the performance is similar to the Cuda version and the model is able to achieve almost 4 times better performance than the Python version of the model.

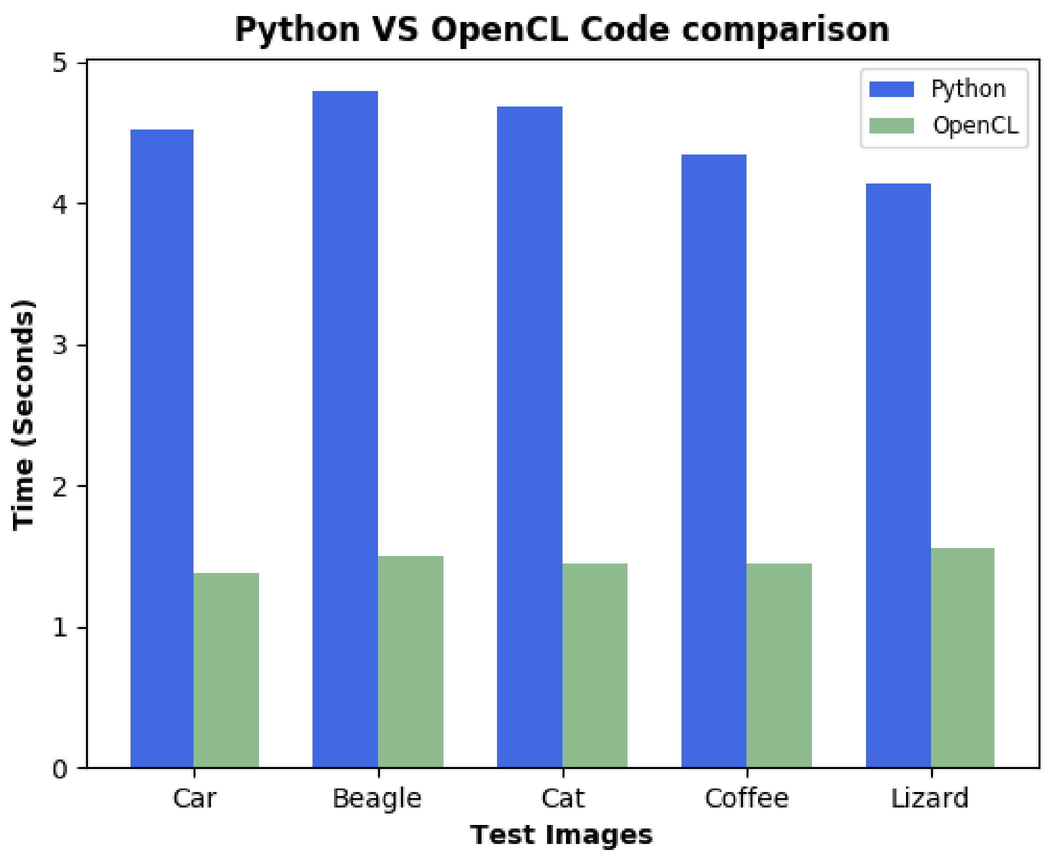


Figure 19: Execution Time comparison between Python and OpenCL model

We do not have any OpenCL profiler which gives the statistics related to the performance of the OpenCL model on a GPU. Hence, collecting those statistics were not possible.

VI. Conclusion

In this project, we provide a benchmark suite which can be used to evaluate the performance of any architecture which can be executed on the OpenCL and Cuda platforms. The benchmark suite can be executed without any of the proprietary DNN libraries or installing resource intensive convolution neural network frameworks. We tested the model on 3 different hardware's and collected different statistics regarding the execution of the CNN model.

We also provided various statistics which can be really useful for CNN accelerators. With the Cuda and OpenCL version, we were also able to significantly reduce the execution time of the model achieving a nearly 5-fold gain in execution speed than the Python model.

As there was no compiler available to collect the statistics for the OpenCL version, we would want to collect that in the future. We would also want to collect statistics from GPU simulators like GPGPU-Sim which can provide valuable statistics & results on the execution pattern of the CNN models. We would also want to support FPGA platforms in the future so that the CNN model execution can be tested on those architectures as well.

We currently only have MobileNets model implemented in Cuda and OpenCL, but in the future, we would want to add a few other CNN models to the benchmark suite. This would provide a comprehensive repository and allow more insights into the execution pattern for researchers for different types of workloads.

VII. References

[1] R. Geirhos, “Comparing deep neural networks against humans: object recognition when the signal gets weaker”. Available: https://arxiv.org/pdf/1706.06969.pdf

[2] S. Lee, K. Son, H. Kim and J. Park, "Car plate recognition based on CNN using embedded system with GPU," *2017 10th International Conference on Human System Interactions (HSI)*, Ulsan, 2017, pp. 239-241.doi: 10.1109/HSI.2017.8005037

[3] J. Liang, X. Chen, M. He, L. Chen, T. Cai and N. Zhu, "Car detection and classification using cascade model," in *IET Intelligent Transport Systems*, vol. 12, no. 10, pp. 1201-1209, 12 2018.
doi: 10.1049/iet-its.2018.5270

[4] D. H. Hubel and T. N. Wiesel, “Receptive fields of single neurons in the cats straite cortex” in J. Physiol. (1959) I48, 574-591 publication date: April, 1959. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1359523/pdf/jphysiol01247-0121.pdf

[5] Fukushima, “Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”, Biol. Cybernetics 36, 193 202 (1980

[6] Y. LeCun et al, “Backpropagation Applied to Handwritten Zip Code recognition” http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf

[7] Convolutional Neural Network, https://en.wikipedia.org/wiki/Convolutional_neural_network

[8] K. Simonyan et al, "Very Deep Convolutional Networks for Large Scale Image Recognition", Available: https://arxiv.org/pdf/1409.1556.pdf

[9] C. Szegedy et al, "Going Deeper with Convolutions", Available: https://arxiv.org/abs/1409.4842

[10] Field Programmable Gate Arrays, https://en.wikipedia.org/wiki/Field-programmable_gate_array

[11] Z. Nagy et al, "Configurable multi-layer CNN-UM emulator on FPGA using distributed arithmetic" Available: https://ieeexplore.ieee.org/document/1046481

[12] W. Xie, "An Energy-Efficient FPGA-Based Embedded System for CNN Application", Available: https://ieeexplore.ieee.org/document/8487057

[13] Graphical Processing Unit (GPUs), Available: https://www.researchgate.net/figure/Schematic-of-NVIDIA-GPU-architecture-where-SM-refers-to-streaming-multiprocessor_fig2_321958738

[14] A. Krizhevsky, "ImageNet Classification with Deep Convolutional Neural Networks", Available: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[15] Cuda, https://www.geforce.com/hardware/technology/cuda

[16] S. Oh et al, "Investigation on performance and energy efficiency of CNN-based object detection on embedded device", in IEEE, Bali, Indonesia, conference date: 8-10 Aug 2017

[17] A. Howard et al, “MobileNets” Available: https://arxiv.org/abs/1704.04861

[18] L. Sifre, “Rigid-Motion scattering for Image Classification” Available: https://www.di.ens.fr/data/publications/papers/phd_sifre.pdf

[19] C. Cullinan et al, “Computing Performance Benchmarks among CPU, GPU, and FPGA”, Available: https://web.wpi.edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking_Final.pdf

[20] Spec CPU Benchmark, https://www.spec.org/cpu/

[21] Baidu DNN Benchmark, https://github.com/baidu-research/DeepBench

[22] Dawn Benchmark, https://dawn.cs.stanford.edu/benchmark/

[23] AlexNet Wikipedia, https://en.wikipedia.org/wiki/AlexNet

[24] Keras Library, https://keras.io/

[25] Tensorflow Library, https://www.tensorflow.org/

[26] PyTorch Library, https://pytorch.org/

[27] Cublas Library, https://developer.nvidia.com/cublas

[28] cuDNN Library, https://developer.nvidia.com/cudnn

[29] CodeEmporium YouTube Channel, https://www.youtube.com/watch?v=T7o3xvJLuHk

[30] Nvprof, https://docs.nvidia.com/cuda/profiler-users-guide/index.html