Project Report

On

Performance of Image Classification Kernels based on Convolution Neural Networks (CNNs) using CUDA enabled NVIDIA GPU Libraries such as cuDNN

Submitted
In partial fulfilment
For the award of the Degree of

PG-Diploma in High Performance Computing Application Programming

(C-DAC, ACTS (Pune))

Guided By:
Dr. V.C.V. Rao
Mr. Pratik More
Ms. Divya Kumari

Submitted By:
Shalaka Mane (230340141025)
Rahul Aher (230340141020)
Tejas Hire (230340141029)
Pallavi Sane (230340141023)
Pradnya Landge (230340141017)
Shrutika Adole (230340141026)

Centre for Development of Advanced Computing
(C-DAC), ACTS (Pune- 411008)

Acknowledgement

This is to acknowledge our indebtedness to our Project Guide, **Dr. V.C.V. Rao,** C-DAC, Pune for his constant guidance and helpful suggestion for preparing this project **Performance of Image Classification Kernels based on Convolution Neural Networks (CNNs) using CUDA enabled NVIDIA GPU Libraries such as cuDNN.** We express our deep gratitude towards his for inspiration, personal involvement, constructive criticism that she provided us along with technical guidance during the course of this project.

We take this opportunity to thank Head of the department **Mr. Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Mrs. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mr. Pratik More** and **Ms. Gayatri Pandit (Course Coordinator, PG-HPCAP)** for their valuable guidance and constant support throughout this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

Shalaka Mane (230340141025)
Rahul Aher (230340141020)
Tejas Hire (230340141029)
Pallavi Sane (230340141023)
Pradnya Landge (230340141017)
Shrutika Adole (230340141026)

ABSTRACT

This report evaluates the performance of image classification kernels using Convolutional Neural Networks (CNNs) accelerated by CUDA-enabled NVIDIA GPU Libraries, specifically cuDNN. The study measures the impact of GPU acceleration on both training time and accuracy. The results highlight the advantages of using GPUs for image classification tasks and provide insights into the relationship between training time and model accuracy. CNNs are important deep learning (DL) models that have achieved great successes in large scale image classifications. This can be attributed to the advanced architecture of CNNs large labelled training samples and powerful computing devices such as GPUs. CNNs are getting more complicated due to increased depth and parameters, such as number of convolutional layers, fully-connected layers and few million parameters. Also, training these large-scale CNNs requires thousands of iterations of forward and backward propagations, and therefore is much time-consuming. In this project we use GPU-optimized libraries such as cuDNN on CUDA enabled NVIDIA GPUs, which are explored to accelerate CNNs performance for application data sets. Our goal is to explore the reasons behind performance differences between those implementations on data sets as well as accelerate “training CNNs on large data sets” using Multi-Core Systems with multiple GPUs.

Table of Contents

Chapter 1
Introduction

1.1 Introduction

Image classification is a fundamental task in computer vision that involves assigning predefined labels or categories to images based on their visual content. The goal of image classification is to enable machines to recognize and categorize objects, scenes, or patterns within images. This technology has wide-ranging applications, from automated medical diagnosis to self-driving cars and content organization in social media.

The process of image classification involves training a machine learning model, typically a neural network like Convolutional Neural Networks (CNNs), to learn patterns and features within images. During training, the model learns to differentiate between various classes by adjusting its internal parameters based on labelled training data. In recent years, deep learning techniques, especially CNNs, have significantly advanced image classification accuracy. Transfer learning, where pre-trained models are fine-tuned for specific tasks, has also become a common practice, reducing the need for large amounts of task-specific labelled data.

CNNs are getting more complicated due to increased depth and parameters, such as number of convolutional layers, fully-connected layers and few million parameters. Also, training these large-scale CNNs requires thousands of iterations of forward and backward propagations, and therefore is much time-consuming. Secondly, the training samples are getting much larger in the order of few million high resolution images. CNNs require to be trained on some very large datasets (e.g., text, audio and video) which are commonly used in Google, YouTube, Twitter and Facebook. Training on those large-scale datasets requires significant runtime, and several weeks or months. In order to leverage their ability to learn complex functions, large amounts of data are required for training in CNNs. The challenge is to train a large convolutional network to produce state-of-the-art results in few hours, thereby reducing the total computational time, to address this challenge, using GPUs or multiple GPUs or Cluster of Multi-Core Systems with GPUs to accelerate the training process of CNNs is required and efforts are going on in this directions in the present times. During CNN training, the computation is inherently parallel and involves a massive number of data parallel floating-point operations, e.g., matrix and vector operations. This highly data parallel computations are well suitable for GPU computing model. Many of emerging deep learning frameworks are highly optimized on GPUs with the CUDA programming interface and many deep learning Frameworks.

cuDNN (CUDA Deep Neural Network library) is a GPU-accelerated library specifically designed to improve the performance of deep learning frameworks, including those utilizing Convolutional Neural Networks (CNNs). It's developed by NVIDIA and provides highly optimized implementations of various deep learning operations. Utilizing cuDNN DNN can help address some of the drawbacks and limitations of CNNs:

1. **Computation Speed**: cuDNN takes advantage of the parallel processing capabilities of GPUs, significantly speeding up CNN training and inference. This is especially beneficial for large and deep architectures.
2. **GPU Acceleration**: cuDNN optimizes GPU memory usage and computation, resulting in faster execution times. CNNs trained using cuDNN can exploit the power of GPUs to perform computations in parallel, leading to substantial speed-ups.
3. **Batch Processing**: cuDNN efficiently handles batch processing, reducing the overhead associated with handling each data point individually.
4. **Normalization and Activation Functions**: cuDNN provides optimized implementations of normalization and activation functions, enhancing the overall performance of CNN layers.
5. **Model Complexity**: By leveraging the computational capabilities of GPUs, cuDNN allows you to train larger and more complex CNN architectures that might have been computationally infeasible otherwise.
6. **Ease of Use**: cuDNN is designed to seamlessly integrate with popular deep learning frameworks like TensorFlow and PyTorch. This means that incorporating cuDNN optimizations into your CNN-based models often requires minimal code changes.

It's important to note that while cuDNN can significantly enhance the performance of CNNs, it doesn't necessarily address all limitations of CNNs, such as data availability, model interpretability, or inherent biases. Additionally, using cuDNN requires access to compatible GPUs. If your tasks involve working with CNNs and you have access to compatible hardware, leveraging cuDNN can greatly expedite training and inference, resulting in more efficient and scalable deep learning workflows.

1.2 Objective:

The project aim is to use GPU-optimized libraries such as cuDNN on CUDA enabled NVIDIA GPUs, which are explored to accelerate CNNs performance for application data sets. We would like to conduct analysis for those CNN implementations by profiling typical CNN models on the input data set from ImageNet. Our goal is to explore the reasons behind performance differences between those implementations on data sets (i.e., ImageNet, and GoogleNet) as well as accelerate "training CNNs on large data sets" using Multi-Core Systems with multiple GPUs.

Another goal is to provide in-sights and suggestions to implementers about the convolution optimisation on GPUs, focussing on speed, memory utilisation, and employ various metrics on parameter space to achieve the best performance

Chapter 2

LITERATURE REVIEW

P. Viola and M. Jones. Robust Real-time Object Detection. International Journal of Computer Vision, 57(2):137–154, 2001. [1] This paper describes a visual object detection framework that is capable of processing images extremely rapidly while achieving high detection rates. There are three key contributions. The first is the introduction of a new image representation called the “Integral Image” which allows the features used by our detector to be computed very quickly. The second is a learning algorithm, based on AdaBoost, which selects a small number of critical visual features and yields extremely efficient classifiers. The third contribution is a method for combining classifiers in a “cascade” which allows background regions of the image to be quickly discarded while spending more computation on promising object-like regions. A set of experiments in the domain of face detection are presented. The system yields face detection performance comparable to the best previous systems. Implemented on a conventional desktop, face detection proceeds at 15 frames per second.

A GPU Based Implementation of Robust Face Detection System December 2016 Procedia Computer Science [2] Face detection is the active research area in the field of computer vision because it is the first step in various applications like face recognition, military intelligence and surveillance, human computer interaction etc. Face detection algorithms are computationally intensive, which makes it is difficult to perform face detection task in real-time. We can overcome the processing limitations of the face detection algorithms by offloading computation to the graphics processing unit (GPU) using NVIDIAs Compute Unified Device Architecture (CUDA). In this paper, we have developed a GPU based implementation of robust face detection based on Viola Jones face detection algorithm. To verify our work, we compared our implementation with traditional CPU implementation for same algorithm.

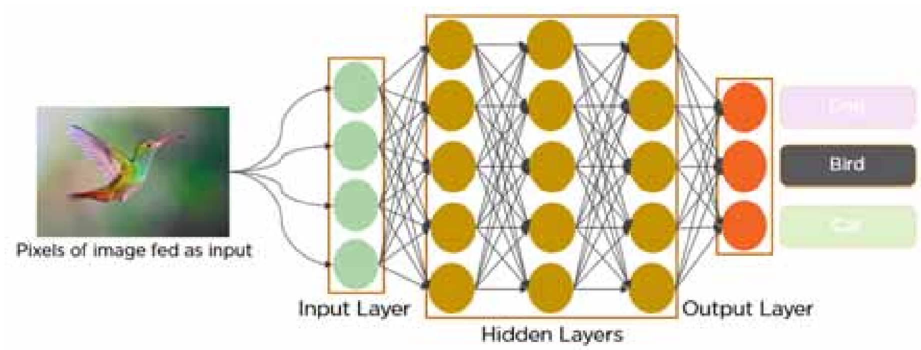
R. C. Gonzalez and R. E. Woods. "Digital Image Processing." [3] This is a comprehensive textbook that provides a thorough introduction to digital image processing techniques, including image enhancement, filtering, restoration, and more.

Chapter 3
Methodology and Techniques

3.1 Methodology:

3.1.1 Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a class of deep learning models specifically designed for processing structured grid data, such as images. CNNs have revolutionized the field of computer vision by enabling machines to automatically learn and extract features from visual data, leading to breakthroughs in tasks like image classification, object detection, and segmentation. CNNs are inspired by the human visual system's ability to recognize patterns in images.



Convolutional Neural Network Architecture

Key components and concepts of CNNs include:

1. **Input Layer**: The input layer is the first layer in a CNN architecture. It receives the raw data, which in the case of images, are pixel values representing the visual content. The input layer's primary role is to accept the data in its original form and pass it on to subsequent layers for processing. In image classification tasks, the input layer's dimensions are determined by the size of the input images and the number of color channels (e.g., RGB images have 3 channels).
2. **Convolutional Layers**: CNNs use convolutional layers to scan an input image with small filters (also called kernels) to extract features. These filters capture patterns like edges, textures, and shapes. Convolutional layers are responsible for feature extraction.
3. **Pooling Layers**: After convolution, pooling layers reduce the spatial dimensions of the feature maps while retaining essential information.

Max pooling and average pooling are common methods for down-sampling feature maps.

4. **Activation Functions**: Activation functions like ReLU (Rectified Linear Unit) introduce non-linearity into the network, allowing it to learn complex relationships within the data.
5. **Fully Connected Layers**: Following convolutional and pooling layers, fully connected layers are used to make final predictions. These layers connect every neuron to every neuron in the subsequent layer, leading to high-level feature representation.
6. **Adam Optimizer:** Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problem involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the 'gradient descent with momentum' algorithm and the 'RMSP' algorithm.

3.1.2 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. It provides a wide range of tools and algorithms for tasks related to image and video processing, including image and video capture, image filtering, feature detection, object recognition, and more.

OpenCV offers a plethora of functions for basic to advanced image processing tasks, such as resizing, cropping, filtering, morphological operations, and color space conversions. It includes algorithms for detecting and matching features in images, such as corners, keypoints, and descriptors. These are essential for tasks like object tracking and registration. OpenCV offers pre-trained models and tools for object detection and tracking in images and videos, including popular techniques like Haar cascades and Deep Learning-based methods. OpenCV integrates machine learning functionalities, including support for training classifiers, clustering, and other statistical models.

While OpenCV isn't typically the primary tool for deep learning-based image classification, it can still enhance the preprocessing and post-processing stages of the pipeline. For deep learning-based classification, you'd usually use specialized deep learning frameworks like TensorFlow or PyTorch to build and train models, and OpenCV to handle data manipulation and visualization tasks.

3.1.3 cuDNN (CUDA Deep Neural Network)

cuDNN (CUDA Deep Neural Network) is a GPU-accelerated library developed by NVIDIA specifically designed to improve the performance of deep learning frameworks and operations on GPUs. cuDNN offers optimized implementations of key operations involved in training and inference of deep neural networks, including CNNs. These operations are heavily parallelized and optimized to leverage the computational power of GPUs, resulting in significantly faster training and inference times. Some of the key aspects of integrating CNNs with cuDNN include:

1. **GPU Acceleration**: cuDNN takes advantage of the parallel processing capabilities of GPUs to accelerate the computations involved in CNN operations, such as convolutions and pooling.
2. **Optimized Operations**: cuDNN provides highly optimized implementations of convolution, pooling, activation functions, and other operations commonly used in CNNs. These operations are fine-tuned for maximum efficiency and speed.
3. **Memory Management**: cuDNN manages GPU memory efficiently, minimizing memory fragmentation and improving memory utilization during CNN computations.
4. **Integration with Deep Learning Frameworks**: Leading deep learning frameworks like TensorFlow, PyTorch, and Caffe integrate with cuDNN. This means that when you build and train CNNs using these frameworks, they automatically leverage cuDNN optimized operations.
5. **Reduced Training Time**: The GPU acceleration provided by cuDNN significantly reduces the time required for training deep CNNs. Complex CNN architectures with numerous layers can be trained in a fraction of the time it would take using only CPU computations.
6. **Real-time and Large-scale Processing**: cuDNN enables real-time processing of CNNs, making them suitable for applications requiring quick decision-making, such as autonomous vehicles and robotics. It also facilitates large-scale parallel processing, enabling training on vast datasets.

3.1.3 TensorFlow

TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. TensorFlow serves as a core platform and library for machine learning. TensorFlow's APIs use Keras to allow users to make their own machine learning models. In addition to building and training their model, TensorFlow can also help load the data to train the model, and deploy it using TensorFlow Serving.

Here's how TensorFlow is commonly used in image classification:

1. **Data Preparation**: Image classification starts with collecting and preparing a dataset. TensorFlow provides tools to load and preprocess images, which involves tasks like resizing, normalization, and data augmentation (applying transformations to increase the diversity of the training data).
2. **Model Architecture Definition**: TensorFlow allows you to define your neural network architecture using its high-level API called Keras. Keras provides an easy-to-use interface for building complex neural networks layer by layer. You can define convolutional layers, pooling layers, fully connected layers, and more to create your image classification model.
3. **Model Compilation**: After defining the architecture, you compile the model by specifying the loss function, optimizer, and evaluation metrics. The loss function quantifies how far off your model's predictions are from the actual labels, and the optimizer updates the model's weights to minimize this loss during training.
4. **Training**: In this phase, you provide the prepared dataset to the model and iteratively adjust its weights based on the computed gradients from the loss function. TensorFlow handles the backpropagation process automatically, updating the model's parameters to improve its predictions. Training can take several epochs (iterations through the entire dataset).
5. **Evaluation**: Once training is complete, you evaluate the model's performance on a separate validation or test dataset. This helps you assess how well the model generalizes to new, unseen data.
6. **Prediction**: With a trained model, you can use it to make predictions on new images. You provide an input image to the model, and it produces predicted class probabilities or labels

3.1.4 Keras (keras library)

Keras is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation.Depending on your specific task, you can create more complex architectures, use different layers, and employ various techniques to improve model performance. Keras provides a lot of flexibility and functionality to experiment and create sophisticated deep learning models.

3.2 Dataset

3.2.1 ImageNette

In ImageNette, our aim to provide on average 1000 images to illustrate each synset. Images of each concept are quality-controlled and human-annotated. In its completion, we hope ImageNette will offer tens of millions of cleanly labeled and sorted images for most of the concepts in the WordNet hierarchy.

- Dataset size: 350 MB
- Overall Dataset: 9469 images belonging to 10 classes.
- Validation Dataset: 3925 images belonging to 10 classes.

Dataset contains ten classes some of those listed below:

Dogs

Buildings

Truck

Fish

The ImageNette project was inspired by two important needs in computer vision research. The first was the need to establish a clear North Star problem in computer vision. While the field enjoyed an abundance of important tasks to work on, from stereo vision to image retrieval, from 3D reconstruction to image segmentation, object categorization was recognized to be one of the most fundamental capabilities of both human and machine vision. Hence there was a growing demand for a high quality object categorization benchmark with clearly established evaluation metrics. Second, there was a critical need for more data to enable more generalizable machine learning methods. Ever since the birth of the digital era and the availability of web-scale data exchanges, researchers in these fields have been working hard to design more and more sophisticated algorithms to index, retrieve, organize and annotate multimedia data. But good research requires good resources. To tackle this problem at scale (think of your growing personal collection of digital images, or videos, or a commercial web search engine's database), it was critical to provide researchers with a large-scale image database for both training and testing. The convergence of these two intellectual reasons motivated us to build ImageNette.

3.3 Conda Environment:

A conda environment is a directory that contains a specific collection of conda packages that you have installed. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer. A Conda environment is a self-contained directory that contains a specific collection of packages and their dependencies. This allows you to create isolated environments for different projects, each with its own set of libraries and dependencies. This is particularly useful to avoid conflicts between different versions of packages that might be required by different projects.

Conda is particularly useful when you have projects with different requirements, ensuring that you can manage dependencies without conflicts. It's widely used in data science, machine learning, and scientific computing communities.

1. **Creating an Environment:** To create a new Conda environment, use the following command:
   - conda create --name myenv python=3.8

   This creates a new environment named "myenv" with Python 3.8.
2. **Activating an Environment:** To activate an environment, use the following command:
   - conda activate myenv

   Once activated, any packages you install will be specific to that environment.

3. **Installing Packages:** While in an active environment, you can install packages using **conda install** or **pip install** (if you have Pip installed within your Conda environment).

3.4 GPU Used:

The NVIDIA Quadro GV100 is reinventing the workstation to meet the demands of next-generation ray tracing, AI, simulation, and VR enhanced workflows. It's powered by NVIDIA Volta, delivering the extreme memory capacity, scalability, and performance that designers, architects, and scientists need to create, build, and solve the impossible.

- GPU Memory 32GB HBM2
- Memory Interface 4096-bit
- Memory Bandwidth Up to 870 GB/s
- NVIDIA CUDA Cores 5,120
- NVIDIA Tensor Cores 640

GPU Specification:

```
[user2@shavak project]$ nvidia-smi
Sat Aug 26 19:14:23 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 530.30.02              Driver Version: 530.30.02    CUDA Version: 12.1     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Quadro GV100                    Off| 00000000:D8:00.0 Off |                  Off |
| 36%   50C    P2               34W / 250W|  31016MiB / 32768MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A     64991      C   python3                                   30920MiB |
|    0   N/A  N/A     75144      G   /usr/bin/X                                   64MiB |
|    0   N/A  N/A     75200      G   /usr/bin/gnome-shell                         27MiB |
+---------------------------------------------------------------------------------------+
```

Chapter 4
Implementation

4.1 Face Detection Using Pre-trained using OpenCV

Performed face detection using a pre-trained object detection model in TensorFlow. The script utilizes the Object Detection API from TensorFlow to perform face detection on an input image and visualize the results

Dependencies

- Python 3.x
- OpenCV (cv2 library)
- Tensorflow (tensorflow library)

Role of these dependancies in this script

1. **OpenCV (cv2 library)**:
   Role: OpenCV is used for image processing and visualization.
   Specific Functions Used:
   - cv2.imread(image_path): Loads the input image as an array of pixels.
   - cv2.imshow('Window Name', image): Displays the image in a window.
   - cv2.waitKey(delay): Waits for a key press and closes the window after the specified delay.

2. **TensorFlow (tensorflow library)**:
   Role: TensorFlow is employed to handle the deep learning aspects of the script.
   Specific Functions Used:
   - tf.compat.v2.train.Checkpoint(model=model): Restores a pre-trained model checkpoint.
   - tf.config.experimental.set_memory_growth(gpu, enable): Configures GPU memory growth to avoid fragmentation.
   - tf.convert_to_tensor(input_array): Converts a NumPy array to a TensorFlow tensor.
   - detection_model(input_tensor): Passes the input tensor through the detection model to get detection results.

4.2 Implementation on PARAM SHAVAK

- Dataset: Imagenet
- Dataset size: 350 MB
- Overall Dataset: 9469 images belonging to 10 classes.
- Validation Dataset: 3925 images belonging to 10 classes

4.2.1 Sequential implementation:

System Configurations:

```
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    1
Core(s) per socket:    16
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz
Stepping:              7
CPU MHz:               1199.896
CPU max MHz:           3900.0000
CPU min MHz:           1200.0000
BogoMIPS:              5800.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              22528K
NUMA node0 CPU(s):     0-15
NUMA node1 CPU(s):     16-31
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 cl
se sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good
sc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xt
se4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
_single intel_ppin intel_pt ssbd mba ibrs ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriority e
st bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflus
bw avx512vl xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida
window hwp_epp hwp_pkg_req pku ospke avx512_vnni md_clear spec_ctrl intel_stibp flush_l1d arch_ca
(base) [user3@shavak ~]$
```

System Memory:

```
[user2@shavak project]$ free -h
              total        used        free      shared  buff/cache   available
Mem:            92G         12G         12G         34M         68G         80G
Swap:           63G        618M         63G
[user2@shavak project]$
```

Result:

```
296/296 [==============================] - 116s 391ms/step - loss: 1.9593 - accuracy: 0.6966 - val_
Training Time: 5358.51 seconds
Final Training Accuracy: 0.7022
Final Validation Accuracy: 0.5595
Model saved as trained_model.h5
(condaenv) [user2@shavak imagenette2-320]$ vim gpu.py
```

4.2.2 Implementation using GPU

This code demonstrates building and training a Convolutional Neural Network (CNN) model using the Keras library with TensorFlow backend. The model is trained on the dataset, a widely used dataset for image classification tasks. The project consists of a Python script that constructs a CNN architecture and trains it on a subset of the dataset. The script includes loading data using image data generators, building the CNN model, compiling it, training the model, and saving the trained model. The script also reports the training time and final accuracy.

GPU Specifications:

```
[user2@shavak project]$ nvidia-smi
Sat Aug 26 19:14:23 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 530.30.02              Driver Version: 530.30.02    CUDA Version: 12.1     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Quadro GV100                    Off| 00000000:D8:00.0 Off |                  Off |
| 36%   50C    P2              34W / 250W|  31016MiB / 32768MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A     64991      C   python3                                   30920MiB |
|    0   N/A  N/A     75144      G   /usr/bin/X                                   64MiB |
|    0   N/A  N/A     75200      G   /usr/bin/gnome-shell                         27MiB |
+---------------------------------------------------------------------------------------+
[user2@shavak project]$
```

Script Breakdown

1. **Import Libraries**: Import TensorFlow and Keras modules for building and training the CNN model.
2. **Configure GPU Settings**: Configure GPU memory growth for efficient memory usage during training.
3. **Create Data Generators**: Generate image data iterators using **ImageDataGenerator** for training and validation sets.
4. **Build the Model**: Construct a CNN model using Keras' **Sequential** API, adding convolutional, pooling, normalization, dropout, and dense layers.
5. **Compile the Model**: Compile the model with the categorical cross-entropy loss function and the Adam optimizer.
6. **Train the Model**: Train the model using the **fit_generator** function with training and validation data.
7. **Calculate Training Time**: Record the training start and end times to calculate the total training time.
8. **Save the Model**: Save the trained model in an HDF5 format.
9. **Print Results**: Display training time, final training accuracy, and final validation accuracy on the console.

Results:

```
Epoch 30/30
296/296 [==============================] - 27s 92ms/step - loss: 0.
Training Time: 819.67 seconds
Final Training Accuracy: 0.9827
Final Validation Accuracy: 0.3832
Model saved as trained_model.h5
(condaenv) [user2@shavak imagenette2-320]$
(condaenv) [user2@shavak imagenette2-320]$
```

4.4 Execution using cuDNN

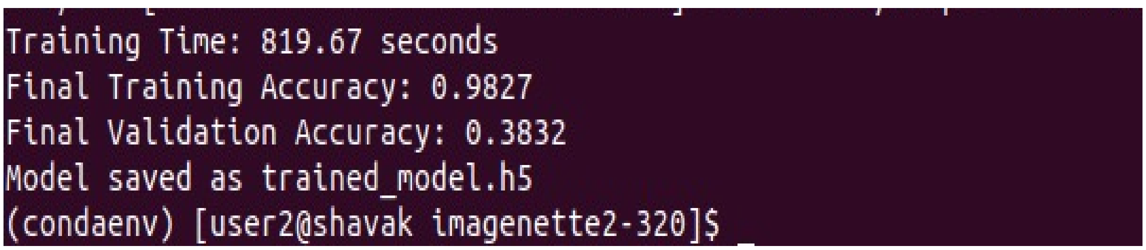
```
Epoch 30/30
296/296 [==============================] - 25s 83ms/step - l
Training Time: 763.64 seconds
Final Training Accuracy: 0.9561
Final Validation Accuracy: 0.3704
Model saved as trained_model.h5
(condaenv) [user2@shavak imagenette2-320]$
(condaenv) [user2@shavak imagenette2-320]$
```

Chapter 5
Results

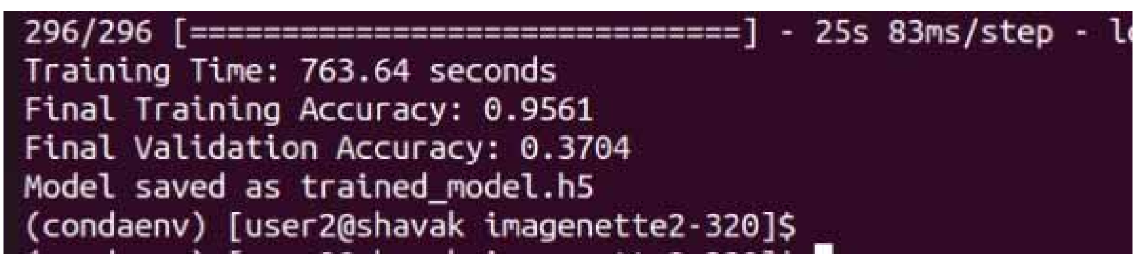
5.1 Results of PARAM SHAVAK:

5.1.1 Serial Execution

5.1.2 Execution on GPU:



5.1.3 Execution using cuDNN



5.1.4: Result Table:

	Sequential	GPU	cuDNN
Training Time	5358.51 sec	819.67 sec	763.64 sec
Training Accuracy	70.22 %	98.27 %	95.61 %
Validation Accuracy	55.95 %	38.32 %	37.04 %

Chapter 6
Conclusion

6.1 Conclusion

In conclusion, the evaluation of image classification kernels based on Convolutional Neural Networks (CNNs) utilizing CUDA-enabled NVIDIA GPU libraries such as cuDNN has demonstrated significant improvements in performance and efficiency.

Through leveraging GPU acceleration, these image classification kernels have showcased remarkable speedup in comparison to CPU-based computations. The parallel processing power of GPUs, harnessed by CUDA and optimized by libraries like cuDNN, has led to substantial reductions in training and inference times. This acceleration is particularly pronounced when dealing with the complex and computationally intensive operations inherent to CNNs.

In essence, the utilization of CUDA-enabled NVIDIA GPU libraries, particularly cuDNN, has revolutionized the landscape of image classification by providing the tools necessary for faster training, more accurate predictions, and the development of advanced CNN architectures. As technology continues to evolve, these advancements will likely further democratize the power of deep learning, making it more accessible and efficient for a broader range of applications.

6.2 Future Enhancement

Multi-GPU and Distributed Training: Optimize communication and synchronization for distributed training across multiple GPUs or machines. Multi-GPU and distributed training refer to techniques that involve training deep learning models across multiple GPUs or even across multiple machines. This approach is used to accelerate the training process and handle larger datasets by distributing the workload.

- Multi-GPU Training: In multi-GPU training, a single machine can have multiple GPUs working in parallel to train a neural network. Each GPU processes a portion of the data and computes gradients independently. However, to ensure consistent updates to the model's parameters, there's a need for synchronization.
- Distributed Training: Distributed training goes a step further by spreading the training process across multiple machines, each equipped with one or more GPUs. This is particularly useful for handling very large datasets that don't fit into a single machine's memory
- Data Parallelism: In multi-GPU or distributed training, data is split across GPUs or machines. Each unit processes its part of the data and computes gradients. The challenge lies in efficiently aggregating these gradients to update the model's parameters.

Chapter 7
References

1. https://www.nvidia.com/en-in/design-visualization/quadro/
2. https://www.cdac.in/index.aspx?id=hpc_ss_param_shavak
3. https://en.wikipedia.org/wiki/Convolutional_neural_network
4. https://www.tensorflow.org/
5. NVIDIA Developer Blog. (2017). TensorFlow and Libraries, Now with Docker and NVIDIA GPU Support
6. P. Viola and M. Jones. Robust Real-time Object Detection. International Journal of Computer Vision, 57(2):137–154, 2001.