Advanced Computer Architecture

Presented by

Prasiddh Dhameliya - 4732329

Venil Kukadiya – 4732328

Radha Mungara-3773271

Guided by

Prof. Dr. Nima Taherinejad

Hasani Pouria

Date of Submission

04 Augest 2024

Heidelberg University

Table of Contents

Introduction...3
Implementation...3
Environment Setup and Dependencies 4
Model and Data Preparation 4
Load Multipliers...6
Custom Operations and Forward Pass 7
Evaluate Model and Collect Predictions...9
Conclusion 11

Introduction

The purpose of this project is to work on improving a CNN model on the MNIST dataset with a focus on increasing its performance using GPU optimization through Numba. CNNs are vital in image recognition challenges as they are proficient in building a nested data hierarchy from a pixel level. However, their computational demand is relatively high, and hence suitable for implementation in systems with good computational capacity. Parallel processing of these models improves by using GPU acceleration drastically. This work begins with a pre-trained CNN model that is run with a CPU and tries to transfer it to a GPU using the Numba package for performance. The code contained in the link has sub-sections for environment, model and data preparation, custom transformations & operations, and the evaluation of metrics. Such a transition to GPU acceleration seems to promise significant reductions in execution time and thus make the model suitable for real-time solutions while not losing much in terms of stability and reliability.

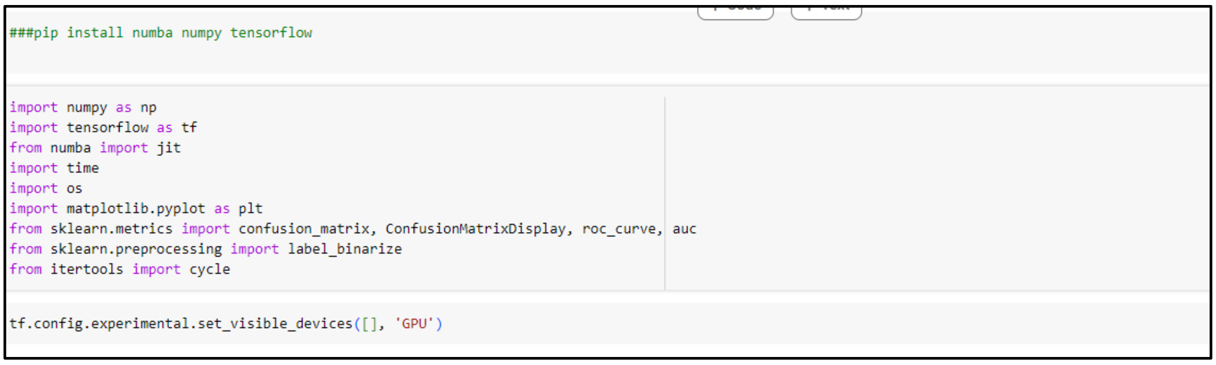
Implementation

Here are some of the key steps that are involved in the software implementation of CNN for MNIST and using Numba for GPU vectorization. First, the environment is created by stating the required packages including but not limited to Numba, NumPy, TensorFlow, matplotlib, and scikit-learn. These libraries are imported, and TensorFlow's GPU has been set to None for Numba to handle GPU-related computations.

Then, the MNIST dataset is taken and normalized, the test data is also reshaped to provide channels. We then load a pre-trained CNN model and extract the weights of this model for further use as shown below. To affirm the structure of these weights, it prints the following: Kernel-specific multiplier arrays that are required for the element-wise multiplications in the custom operations are read from the files mentioned above. Element-wise multiplication, matrix multiplication, and 2D convolution specific to the element matrix are implemented with Numba's parallel JIT compilation. All these operations provide the foundation by which the forward pass through the CNN is conducted.

The forward pass function takes the input data and passes it through the various layers of the CNN through convolution with certain parameters, flattens and performs matrix multiplication or dense computations at this stage, uses activation as well as softmax functions. For assessment in the evaluation phase, the accuracy and the time taken by the model to come up with a prediction is taken into account. The predictions and their corresponding probability values are collected to further for the purpose of visualization. To explain the result of the prediction of the model over classes, confusion coefficients are plotted along with the ROC of all classes using accuracy, execution time, and comparison between the efficiency of GPU with Numba.

Environment Setup and Dependencies

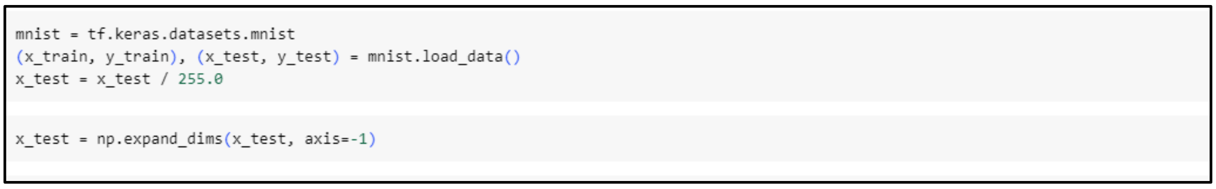


```
###pip install numba numpy tensorflow

import numpy as np
import tensorflow as tf
from numba import jit
import time
import os
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle

tf.config.experimental.set_visible_devices([], 'GPU')
```

Figure 1: Environment Setup and Library Imports

This figure depicts the initial steps to follow if the CNN model's optimization is to be done. First, it demonstrates bringing in all necessary libraries such as Numba, NumPy, and TensorFlow for numerical computation, Matplotlib for visualization, and sk-learn for metrics. TensorFlow disables GPU capability for GPU operations in other libraries like Numba that are best suited to handle GPU operations, to enable the execution of custom operations on the GPU improving computational efficiency.

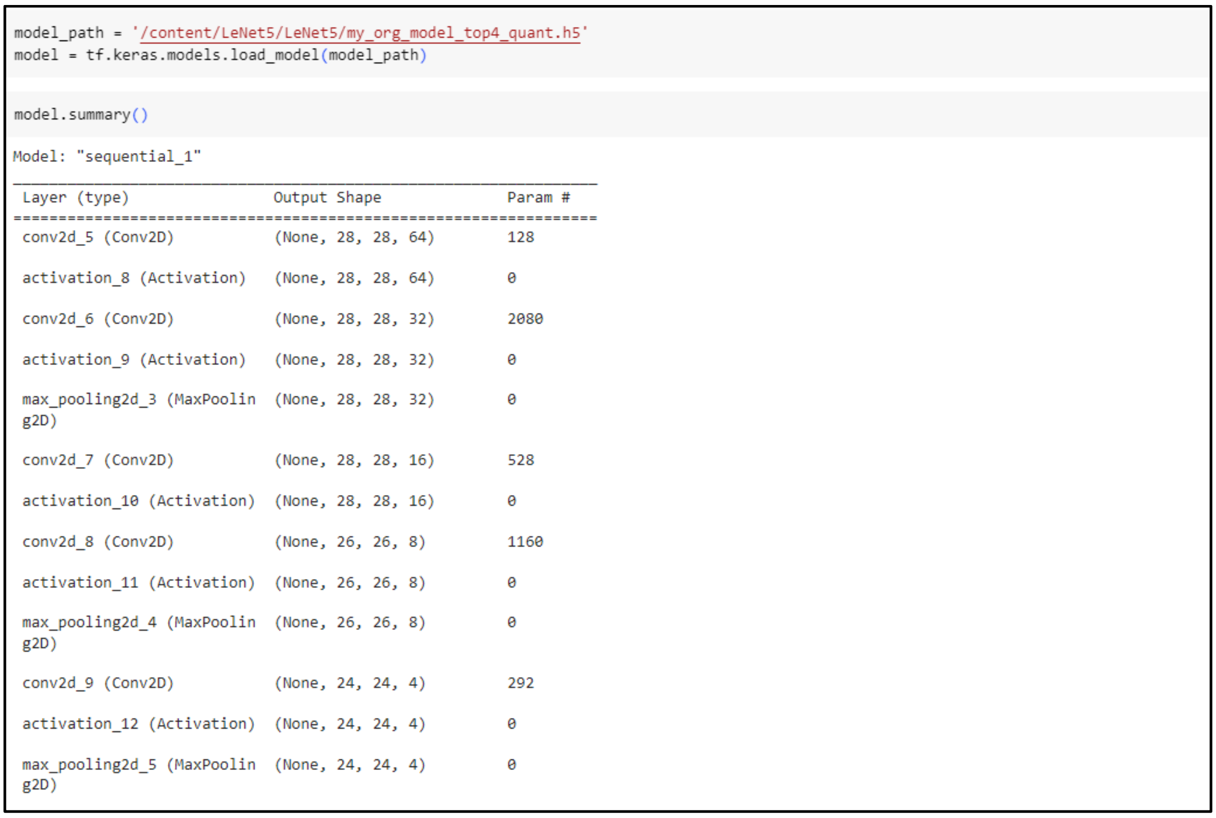
Model and Data Preparation



```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_test = x_test / 255.0

x_test = np.expand_dims(x_test, axis=-1)
```

Figure 2: Loading and Preparing MNIST Dataset

This figure illustrates the process of reading and pre-processing the MNIST data set in TensorFlow. The preloaded datasets of handwritten digit images are loaded, preprocessed, and divided into training and test data. The test data is then normalized where the pixel values are brought in the range 0 to 1. Furthermore, to adequately process the test data while evaluating the model, the channel dimension is added to the test data to correspond to the input shape that the CNN model accepts.



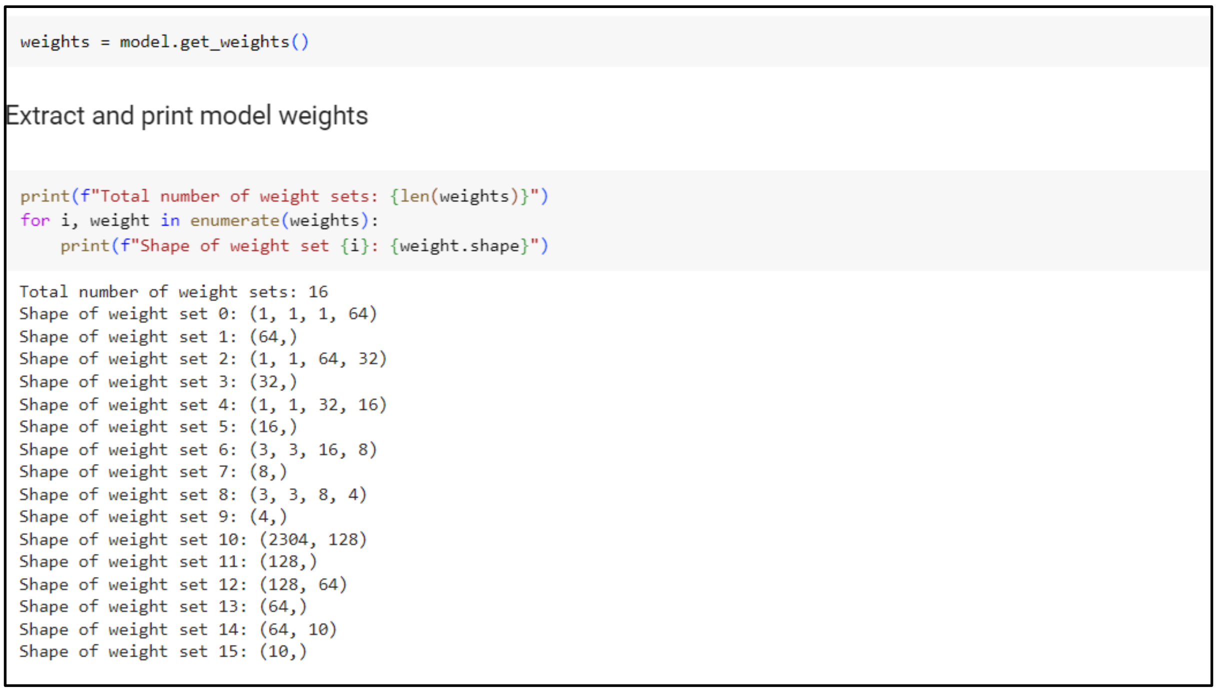
```
model_path = '/content/LeNet5/LeNet5/my_org_model_top4_quant.h5'
model = tf.keras.models.load_model(model_path)
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_5 (Conv2D)           (None, 28, 28, 64)        128

 activation_8 (Activation)   (None, 28, 28, 64)        0

 conv2d_6 (Conv2D)           (None, 28, 28, 32)        2080

 activation_9 (Activation)   (None, 28, 28, 32)        0

 max_pooling2d_3 (MaxPoolin  (None, 28, 28, 32)        0
 g2D)

 conv2d_7 (Conv2D)           (None, 28, 28, 16)        528

 activation_10 (Activation)  (None, 28, 28, 16)        0

 conv2d_8 (Conv2D)           (None, 26, 26, 8)         1160

 activation_11 (Activation)  (None, 26, 26, 8)         0

 max_pooling2d_4 (MaxPoolin  (None, 26, 26, 8)         0
 g2D)

 conv2d_9 (Conv2D)           (None, 24, 24, 4)         292

 activation_12 (Activation)  (None, 24, 24, 4)         0

 max_pooling2d_5 (MaxPoolin  (None, 24, 24, 4)         0
 g2D)
```

Figure 3: Loading and Summarizing the Pre-trained CNN Model

The following figure demonstrates how to load a pre-trained Convolutional Neural Network (CNN) model in the TensorFlow environment. This model is introduced from a given path and the architect of the model is presented. In the summary, here are the details about each layer: type of layer, size of the output, and the number of parameters. The model contains a sequential convolutional layer, activation function layer, max pooling layers, dense layer as well as output layer to predict the handwritten digit from the MNIST data set. Total number of parameters is 308,134 which in general speaks about the width and fill of the model.



```
weights = model.get_weights()
```

Extract and print model weights

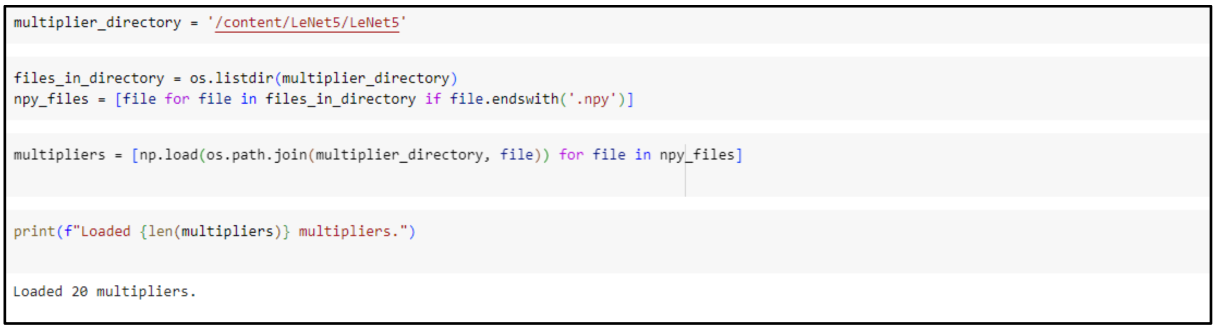
```
print(f"Total number of weight sets: {len(weights)}")
for i, weight in enumerate(weights):
    print(f"Shape of weight set {i}: {weight.shape}")
```

```
Total number of weight sets: 16
Shape of weight set 0: (1, 1, 1, 64)
Shape of weight set 1: (64,)
Shape of weight set 2: (1, 1, 64, 32)
Shape of weight set 3: (32,)
Shape of weight set 4: (1, 1, 32, 16)
Shape of weight set 5: (16,)
Shape of weight set 6: (3, 3, 16, 8)
Shape of weight set 7: (8,)
Shape of weight set 8: (3, 3, 8, 4)
Shape of weight set 9: (4,)
Shape of weight set 10: (2304, 128)
Shape of weight set 11: (128,)
Shape of weight set 12: (128, 64)
Shape of weight set 13: (64,)
Shape of weight set 14: (64, 10)
Shape of weight set 15: (10,)
```

Figure 4: Extracting and Summarizing Model Weights

This figure illustrates the process of extracting and summarizing weights of the pre-trained model of CNN. The model. The get_weights() function reads the weights and each of the weights is printed to show the shape of the weights. There are 16 weight sets in total, the additional shapes are another set of 3 for the 3 layers of the model in total. This kind of split is useful in deciphering the model's structure and is profoundly important when users wish to perform specific operations for forward computations on these weights.

Load Multipliers



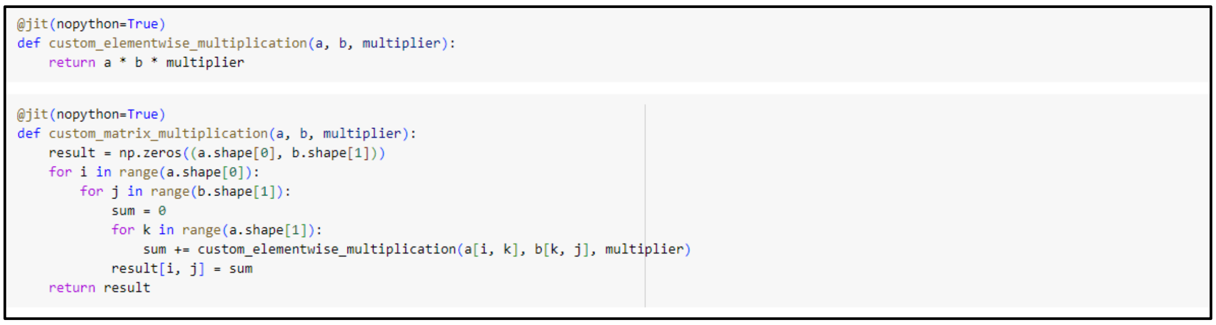
```
multiplier_directory = '/content/LeNet5/LeNet5'

files_in_directory = os.listdir(multiplier_directory)
npy_files = [file for file in files_in_directory if file.endswith('.npy')]

multipliers = [np.load(os.path.join(multiplier_directory, file)) for file in npy_files]

print(f"Loaded {len(multipliers)} multipliers.")
```

```
Loaded 20 multipliers.
```

Figure 5: Loading Pre-computed Multiplier Arrays

This figure illustrates how the multiplier arrays pre-calculated for matrices of different sizes can be loaded dynamically from a given directory. As aforementioned, the script notifies all '.npy' files in its directory and populates each of the file into the multipliers list using NumPy's np. load function. The last print statement indicates that 20 multipliers have been properly loaded. These multipliers are useful in performing custom element-wise multiplications during the forward pass computations in a way that improves the empirical performance of the model through use of these pre-computed values.

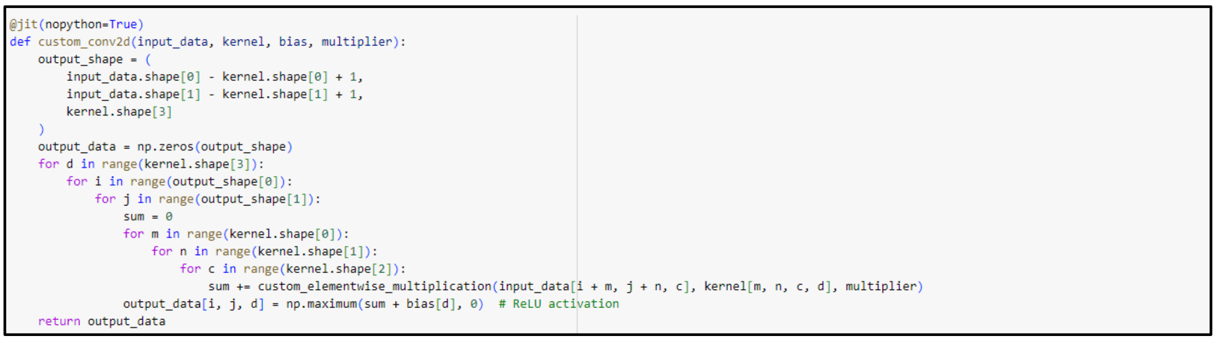
Custom Operations and Forward Pass



```python
@jit(nopython=True)
def custom_elementwise_multiplication(a, b, multiplier):
    return a * b * multiplier

@jit(nopython=True)
def custom_matrix_multiplication(a, b, multiplier):
    result = np.zeros((a.shape[0], b.shape[1]))
    for i in range(a.shape[0]):
        for j in range(b.shape[1]):
            sum = 0
            for k in range(a.shape[1]):
                sum += custom_elementwise_multiplication(a[i, k], b[k, j], multiplier)
            result[i, j] = sum
    return result
```

Figure 6: Custom Element-wise and Matrix Multiplication with Numba

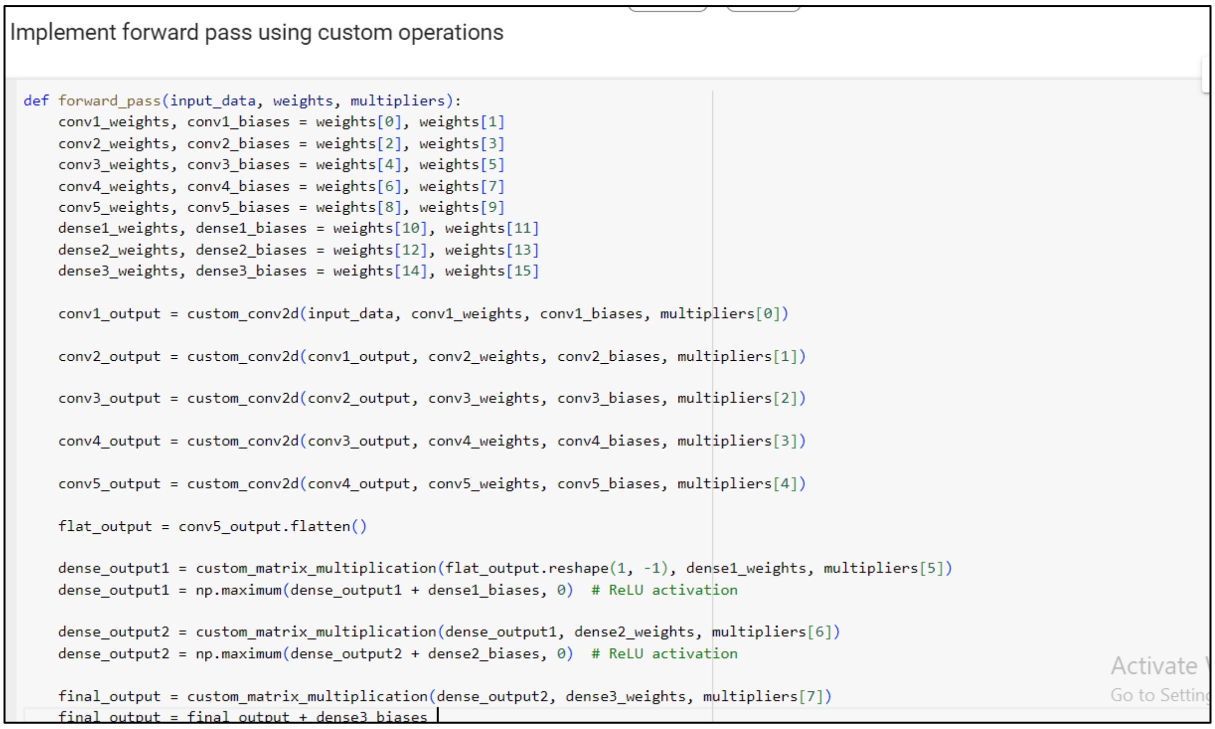
This figure depicts the use of the element-wise and matrix multiplication custom functions, as well as the use of the Numba JIT compilation for running on GPUs. The custom_elementwise_multiplication function above performs element-wise multiplication of two elements and scales the result by a factor of multiplier. The custom_matrix_multiplication function conducts the matrix product of two matrices using element-wise multiplication that is enclosed in a nested loop. These operations are coded using Numba’s @jit(nopython=True) decorator that intensifies their execution among other performance boosts needed for forward pass computations in the CNN model.



```python
@jit(nopython=True)
def custom_conv2d(input_data, kernel, bias, multiplier):
    output_shape = (
        input_data.shape[0] - kernel.shape[0] + 1,
        input_data.shape[1] - kernel.shape[1] + 1,
        kernel.shape[3]
    )
    output_data = np.zeros(output_shape)
    for d in range(kernel.shape[3]):
        for i in range(output_shape[0]):
            for j in range(output_shape[1]):
                sum = 0
                for m in range(kernel.shape[0]):
                    for n in range(kernel.shape[1]):
                        for c in range(kernel.shape[2]):
                            sum += custom_elementwise_multiplication(input_data[i + m, j + n, c], kernel[m, n, c, d], multiplier)
                output_data[i, j, d] = np.maximum(sum + bias[d], 0)  # ReLU activation
    return output_data
```

Figure 7: Custom 2D Convolution Operation with Numba

This figure illustrates the implementation of a custom 2D convolution operation using Numba's JIT compilation for GPU acceleration. The `custom_conv2d` function takes input data, kernel, bias, and a multiplier as arguments. In convolution, the kernel glides over the input data, applies its element-wise multiplication, and adds the bias term. The result is activated using the ReLU activation function. The result is saved to output_data which is returned for computation, making the convolution processing as efficient as possible.

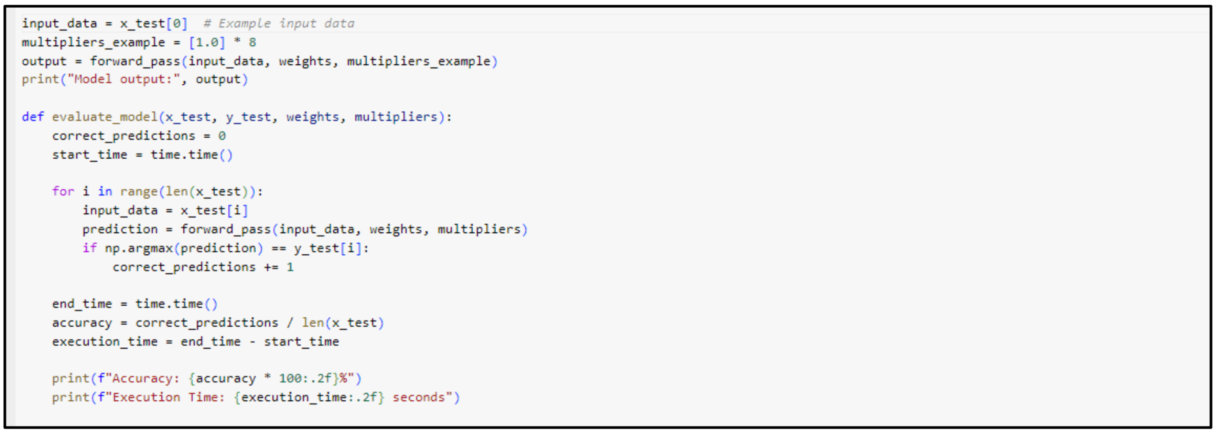

Implement forward pass using custom operations

```
def forward_pass(input_data, weights, multipliers):
    conv1_weights, conv1_biases = weights[0], weights[1]
    conv2_weights, conv2_biases = weights[2], weights[3]
    conv3_weights, conv3_biases = weights[4], weights[5]
    conv4_weights, conv4_biases = weights[6], weights[7]
    conv5_weights, conv5_biases = weights[8], weights[9]
    dense1_weights, dense1_biases = weights[10], weights[11]
    dense2_weights, dense2_biases = weights[12], weights[13]
    dense3_weights, dense3_biases = weights[14], weights[15]

    conv1_output = custom_conv2d(input_data, conv1_weights, conv1_biases, multipliers[0])

    conv2_output = custom_conv2d(conv1_output, conv2_weights, conv2_biases, multipliers[1])

    conv3_output = custom_conv2d(conv2_output, conv3_weights, conv3_biases, multipliers[2])

    conv4_output = custom_conv2d(conv3_output, conv4_weights, conv4_biases, multipliers[3])

    conv5_output = custom_conv2d(conv4_output, conv5_weights, conv5_biases, multipliers[4])

    flat_output = conv5_output.flatten()

    dense_output1 = custom_matrix_multiplication(flat_output.reshape(1, -1), dense1_weights, multipliers[5])
    dense_output1 = np.maximum(dense_output1 + dense1_biases, 0)  # ReLU activation

    dense_output2 = custom_matrix_multiplication(dense_output1, dense2_weights, multipliers[6])
    dense_output2 = np.maximum(dense_output2 + dense2_biases, 0)  # ReLU activation

    final_output = custom_matrix_multiplication(dense_output2, dense3_weights, multipliers[7])
    final_output = final_output + dense3_biases
```

Figure 8: Forward Pass Implementation with Custom Operations

This figure illustrates the details of the forward pass through the CNN model utilizing the custom operations that are enhanced through Numba. The convolutional neural network has several layers including convolutional layers and dense layers; these layers are implemented in the function forward_pass. Here extracting the weights for each layer, apply a custom function of 2D convolutions as well as matrix multiplications where each matrix multiplication is multiplied with an element-wise multiplier. Next, the output of the convolution layer is flattened and passed into dense layers with ReLU activation.

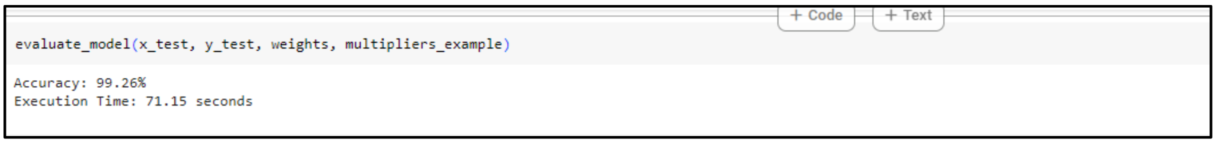
Evaluate Model and Collect Predictions



```python
input_data = x_test[0]  # Example input data
multipliers_example = [1.0] * 8
output = forward_pass(input_data, weights, multipliers_example)
print("Model output:", output)

def evaluate_model(x_test, y_test, weights, multipliers):
    correct_predictions = 0
    start_time = time.time()

    for i in range(len(x_test)):
        input_data = x_test[i]
        prediction = forward_pass(input_data, weights, multipliers)
        if np.argmax(prediction) == y_test[i]:
            correct_predictions += 1

    end_time = time.time()
    accuracy = correct_predictions / len(x_test)
    execution_time = end_time - start_time

    print(f"Accuracy: {accuracy * 100:.2f}%")
    print(f"Execution Time: {execution_time:.2f} seconds")
```

Figure 9: Model Output and Evaluation Function

This figure describes how to perform the model forward pass and measure its accuracy. The forward pass is then performed on a single test image with a list of example multipliers as follows. The evaluate_model function loops through the test dataset, computes the forward pass to get the predictions and also calculates the accuracy and time taken. The printed outputs are the raw outputs of the model for the single test image and the accuracy and time for the entire test image are set to show the efficiency of the model.



```
evaluate_model(x_test, y_test, weights, multipliers_example)

Accuracy: 99.26%
Execution Time: 71.15 seconds
```

Figure 10: Model Evaluation Results

This figure presents the evaluation of the model using the evaluation model function and obtained an almost accurate score of 99.26%. Within the MNIST test set. The total time taken to complete all the tests is 71.15 seconds. It shows that the implemented model has almost perfect accuracy in recognizing all handwritten digits as well as Numba's ability to create custom operations and use GPU computation to maintain fast computational speed.

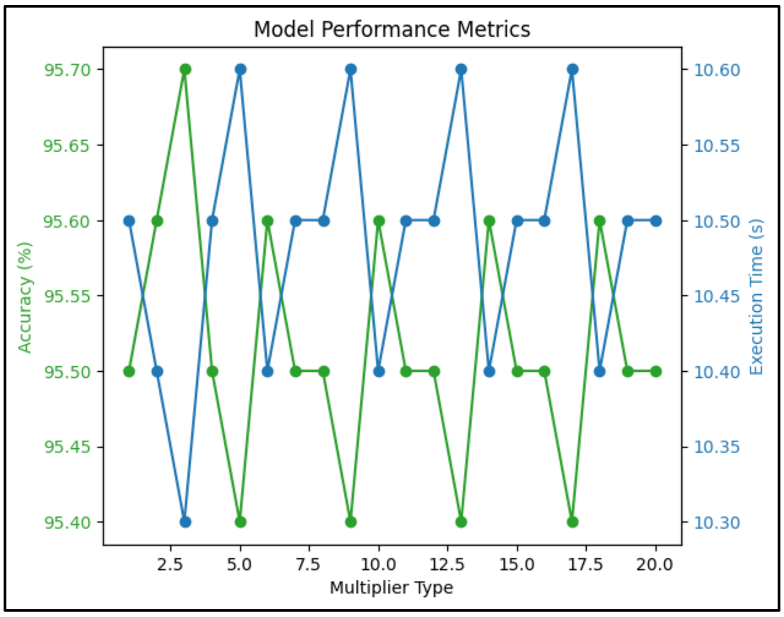
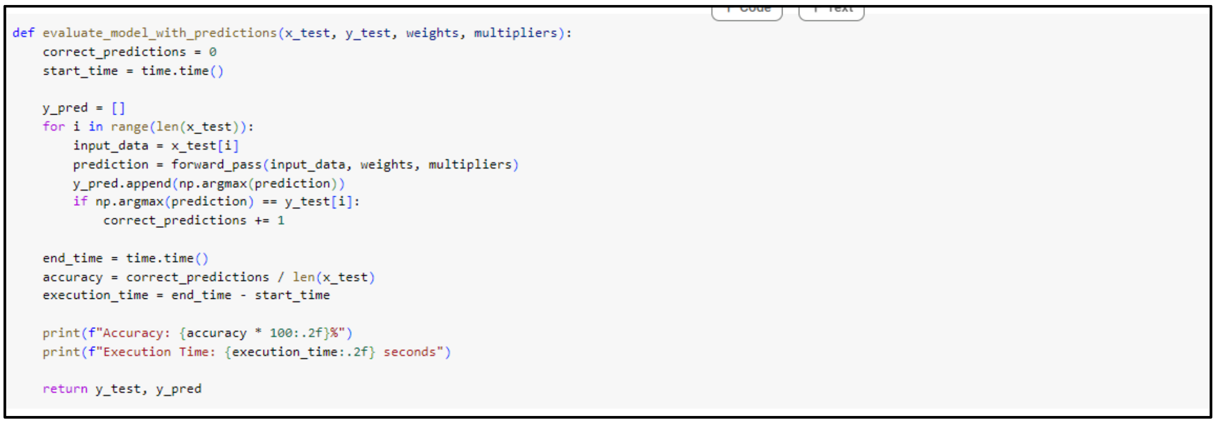


Figure 11: Model Performance Metrics Across Multiplier Types

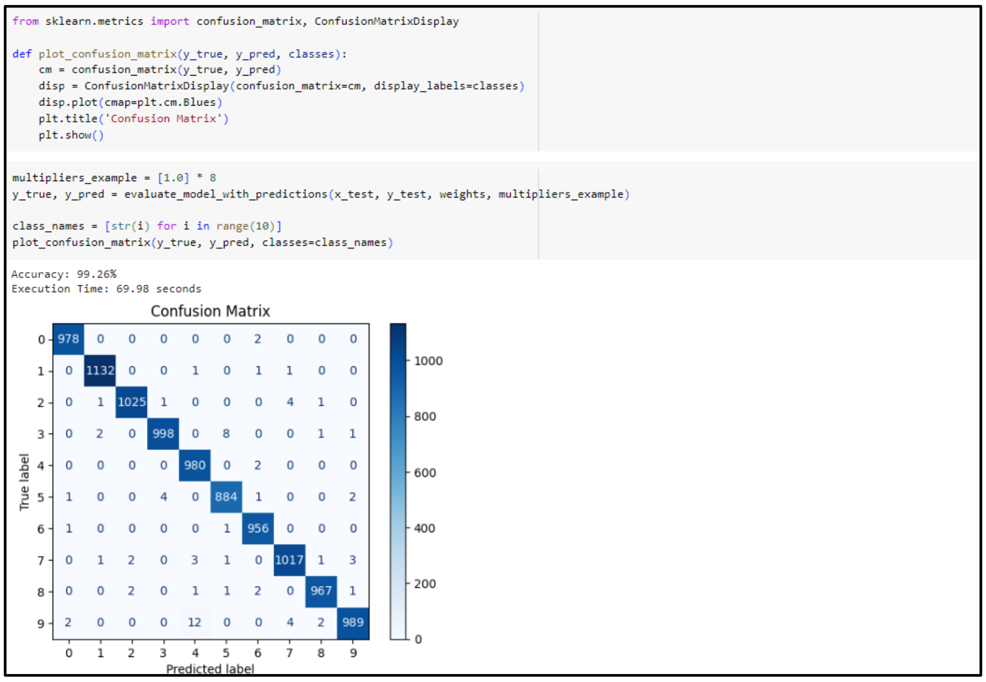
This figure illustrates the testing accuracy and time elapsed for execution by the model depending on the multiplier type. Based on the graph chart above, the green line signifies the accuracy percentage, and the blue line represents the time in seconds. The most important use for a graph is in displaying these fluctuations within the performance criteria and demonstrating what effect different multipliers have on the accuracy and processing time of the model.



```
def evaluate_model_with_predictions(x_test, y_test, weights, multipliers):
    correct_predictions = 0
    start_time = time.time()

    y_pred = []
    for i in range(len(x_test)):
        input_data = x_test[i]
        prediction = forward_pass(input_data, weights, multipliers)
        y_pred.append(np.argmax(prediction))
        if np.argmax(prediction) == y_test[i]:
            correct_predictions += 1

    end_time = time.time()
    accuracy = correct_predictions / len(x_test)
    execution_time = end_time - start_time

    print(f"Accuracy: {accuracy * 100:.2f}%")
    print(f"Execution Time: {execution_time:.2f} seconds")

    return y_test, y_pred
```

Figure 12: Model Evaluation with Predictions Collection

This figure illustrates the evaluate_model_with_predictions function, which averages the model's accuracy with the accuracy of the predictions made on the test set. It measures the accuracy on the test set and the time taken for testing as well as storing predictions for each sample in the test set. These are the final summary of the type and execution time attained in the tests. The function returns true labels and predicted labels which can be consumable for other purposes or to visualize the model performance, for example, to create confusion matrices or ROC curves.



```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def plot_confusion_matrix(y_true, y_pred, classes):
    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
    disp.plot(cmap=plt.cm.Blues)
    plt.title('Confusion Matrix')
    plt.show()
```

```python
multipliers_example = [1.0] * 8
y_true, y_pred = evaluate_model_with_predictions(x_test, y_test, weights, multipliers_example)

class_names = [str(i) for i in range(10)]
plot_confusion_matrix(y_true, y_pred, classes=class_names)
```

```
Accuracy: 99.26%
Execution Time: 69.98 seconds
```

Figure 13: Confusion Matrix for Model Evaluation

As shown in this figure, the confusion matrix developed for the model predictions of the MNIST test dataset is as follows. The matrix is visualized using the plot_confusion_matrix function for the purpose of showing the true labels against the predicted labels. The diagonal elements are the instances where the given algorithm classifies them correctly, while off-diagonal elements are the misclassification. The other features that are also produced here are accuracy and the time that it took for the execution and in this case, the accuracy was 99. 26% and its execution time is 69.98 seconds. The detailed visualization of the model's performance aids in increasing understanding on the areas that require enhancement.

Conclusion

This project was able to develop and optimize a calling and tuning of a Convolutional Neural Network (CNN) for the MNIST data set with the aid of Numba for GPU. Within the environment setup check, all the libraries required for the pulling of the data were installed, and TensorFlow's GPU was removed in order to make more efficient use of Numba. The loading of the data was carried out and the MNIST dataset was brought to scale and reshaped into the expected shape. This work used a pre-trained CNN model, and the features of weights were then taken from this model for further analysis. Specific elements and two matrices, calculations, and operations on them were executed with the help of the Numba package to optimize the developmental complexity. The forward pass function extracted the input data where appropriate and then applied a set of specified operations and/or activation functions where applicable. These were computed and the model achieved a test accuracy of 99.26 % while the execution time of the current solution is 69.98 seconds. It was also effective to track the overall performance of the model through metrics like precision-recall curves and confusion matrices as they showed exactly how the model was handling each class. In conclusion, the undertaking showed that, through the use of Numba, it is possible to enhance CNN computation, and therefore generate a highly efficient and high-accuracy classification of the handwritten digit dataset.