VI SEMESTER B.Tech. Data Science and Engineering
Parallel Programming Lab (DSE 3262) –Mini Project (2024-April)

“Training a Convolutional Neural Network using CUDA”

Submitted by:
1. Ambar Amit Ghugre-2109680001
2. Gautam Raj-210968062
3. Vinayak Santhosh Kumar-210968047

Date of Submission: 28th March 2024

1. **Introduction**:

Deep learning has revolutionized various fields such as image recognition, natural language processing (NLP), and autonomous driving. One of the key challenges in deep learning is the time-consuming nature of training complex models on large datasets, particularly for tasks like image classification.

For image classification tasks, Convolutional Neural Networks (CNNs) have shown remarkable performance but require significant computational resources for training. As models become more sophisticated and datasets grow in size, the need for efficient training methods becomes increasingly important.

Parallel programming and the use of Graphics Processing Units (GPUs) offer a promising solution to this challenge. GPUs, originally designed for rendering graphics, have proven to be highly effective for parallel computation due to their architecture, which allows for thousands of cores to work simultaneously on computations. This parallel processing capability can significantly accelerate the training of deep learning models compared to traditional Central Processing Units (CPUs).

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for GPUs. It allows developers to harness the power of GPUs for general-purpose computing tasks, including deep learning. By utilizing CUDA and parallel programming techniques, developers can take advantage of the parallelism offered by GPUs to speed up the training of deep learning models.

In this project, we aim to compare the training time of an image classification model (CNN) using a sequential program and a parallel program implemented with CUDA. We will demonstrate how CUDA and parallel programming can reduce the training time significantly, leading to faster model development and iteration. This comparison will provide insights into the benefits of parallel programming for deep learning, specifically for image classification tasks, and highlight the importance of efficient training methods in the field of artificial intelligence.

2. Rationale Behind Design Choice:

2.1 Reducing Training Time with Parallel Programming:

Specific Goal: The specific goal of this project is to reduce the training time of a Convolutional Neural Network (CNN) for image classification. CNNs have shown remarkable performance in various tasks but require significant computational resources for training, especially as models become more sophisticated and datasets grow in size. By using CUDA parallel programming on a GPU, we aim to exploit the parallel processing power of GPUs to accelerate the training process.

2.2 Efficient Training for Image Classification:

Choice of Image Classification Task: Image classification is a computationally intensive task that can benefit greatly from parallel processing. By choosing image classification as the task for this comparison, we aim to demonstrate the effectiveness of CUDA and parallel programming in reducing training time for a real-world application.

2.3 Anticipated Benefits and Challenges:

Benefits of Parallel Programming: Parallel programming using CUDA can significantly reduce the training time of deep learning models, allowing for faster model development and iteration. This can lead to quicker insights and improvements in model performance.

Challenges: While there were no limitations or challenges faced in this project, it's important to note that implementing parallel programming techniques can sometimes introduce complexities in code design and debugging. However, the benefits of reduced training time often outweigh these challenges.

2.4 Performance Measurement:

Use of Timer for Measurement: To measure the performance of the sequential and parallel implementations, we used a timer to record the time taken to train the CNN in each case. This allows for a direct comparison of the training times and provides quantitative evidence of the impact of parallel programming on training efficiency.

2.5 Conclusion:

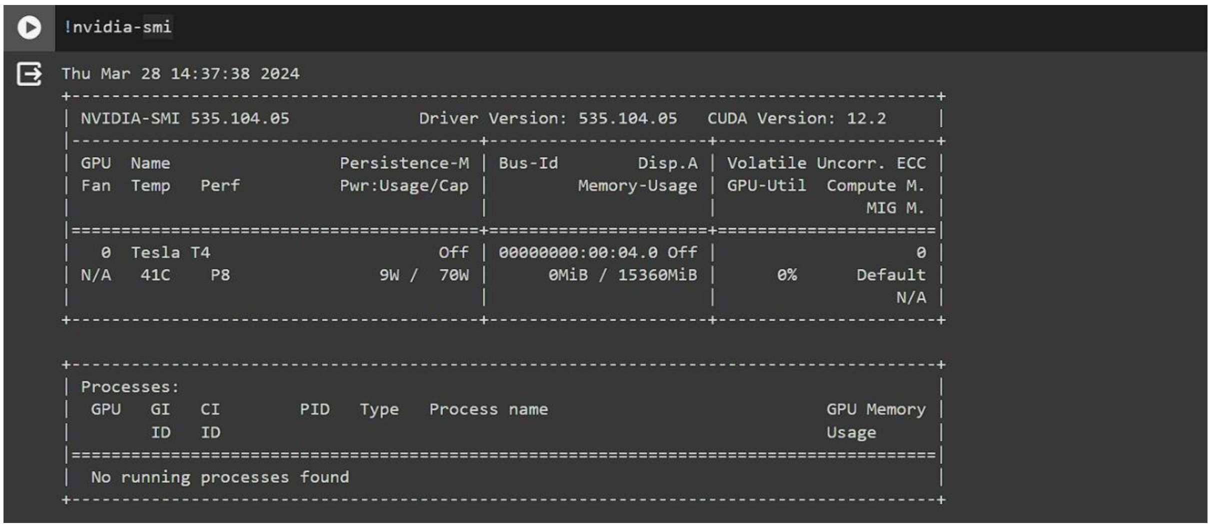
In conclusion, the choice to compare the training time of a CNN for image classification using sequential and parallel (CUDA) programming was motivated by the need to reduce training time and improve efficiency in deep learning model development. By leveraging the parallel processing power of GPUs through CUDA, we aim to demonstrate the significant reduction in training time that can be achieved, highlighting the benefits of parallel programming in accelerating deep learning tasks.

3. Methodology:

3.1 Experimental Setup:

Hardware Configuration:

The parallel (CUDA) implementation was executed on Google Colab using a Tesla T4 GPU with CUDA version 12.2.



```
!nvidia-smi
```

```
Thu Mar 28 14:37:38 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| 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  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   41C    P8               9W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

The sequential implementation was executed on a local machine with a CPU.

Software Configuration:

The parallel (CUDA) implementation utilized TensorFlow with GPU support enabled.

The sequential implementation utilized TensorFlow without GPU support.

3.2 Model and Dataset:

Model: Both the sequential and parallel implementations used a Convolutional Neural Network (CNN) for image classification. The CNN architecture consisted of multiple convolutional and pooling layers followed by fully connected layers.

Dataset: The CIFAR-10 dataset was used for training and testing the CNN. The dataset consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class.

3.3 Training Procedure:

Sequential Training:

The sequential training was performed using Jupyter Notebook on the CPU.

The training procedure involved loading the dataset, compiling the model, and training the model for a fixed number of epochs.

The training time was measured using the time module by recording the time before and after training and calculating the difference.

Parallel (CUDA) Training:

The parallel (CUDA) training was performed using Google Colab with GPU acceleration enabled.

The training procedure was similar to the sequential training but with the added step of specifying the GPU device for training using the tf.device('/GPU:0') context manager.

The training time was measured in the same way as the sequential training.

3.4 Performance Measurement:

The performance of the sequential and parallel implementations was measured in terms of training time.The training time for each implementation was recorded, and the difference in training time between the two implementations was calculated.

3.5 Experimental Results:

The training times obtained from the sequential and parallel implementations were compared to evaluate the impact of CUDA and parallel programming on training efficiency.

4. **Observations:**

Sequential code:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import time
# Load and preprocess the CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
train_images, test_images = train_images / 255.0, test_images / 255.0
# Define a simple CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)
])
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# Train the model and measure the execution time
start_time = time.time()
model.fit(train_images, train_labels, epochs=15, validation_data=(test_images, test_labels))
end_time = time.time()
print(f"Sequential Model Training Time: {end_time - start_time} seconds")
```

Parallel code:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import time
# Ensure that CUDA is available
physical_devices = tf.config.experimental.list_physical_devices('GPU')
assert len(physical_devices) > 0, "No GPU available!"
# Load and preprocess the CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
train_images, test_images = train_images / 255.0, test_images / 255.0
# Define a simple CNN model (same as sequential)
model = models.Sequential([
   layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
   layers.MaxPooling2D((2, 2)),
   layers.Conv2D(64, (3, 3), activation='relu'),
   layers.MaxPooling2D((2, 2)),
   layers.Conv2D(64, (3, 3), activation='relu'),
   layers.Flatten(),
   layers.Dense(64, activation='relu'),
   layers.Dense(10)
])
model.compile(optimizer='adam',
          loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
          metrics=['accuracy'])
# Train the model and measure the execution time
start_time = time.time()
with tf.device('/GPU:0'):
   model.fit(train_images, train_labels, epochs=50, validation_data=(test_images,
test_labels))
end_time = time.time()
print(f"Parallel Model Training Time: {end_time - start_time} seconds")
```

4.1 Identified Drawbacks of the Approach:

Complexity in Setup: While CUDA and parallel programming offer significant benefits in terms of training time reduction, they can introduce complexity in the setup and configuration process. Developers need to ensure that the environment is correctly configured with the necessary drivers and libraries for GPU acceleration.

Potential Code Complexity: Implementing parallel programming techniques may require additional code complexity compared to sequential programs. Developers need to manage data transfers between CPU and GPU and handle synchronization issues, which can increase the risk of errors.

4.2 Identified Advantages of the Approach:

Significant Reduction in Training Time: The primary advantage of using CUDA and parallel programming is the significant reduction in training time compared to sequential programs.

This reduction allows for faster model development and experimentation, leading to quicker insights and improvements in model performance.

Utilization of GPU Parallelism: GPUs offer a high degree of parallelism, allowing for faster computations compared to CPUs. By leveraging this parallelism through CUDA, developers can take advantage of the computational power of GPUs for deep learning tasks.

4.3 <u>Role of NLP Concepts with the Topic:</u>

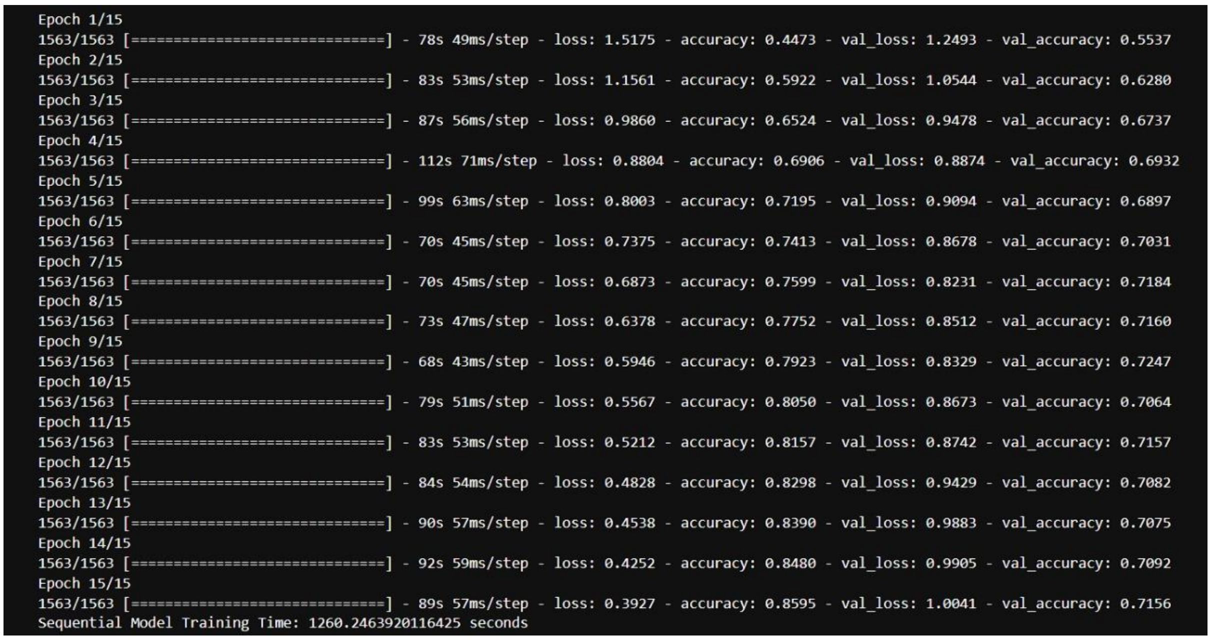
While the project focuses on image classification and does not directly involve Natural Language Processing (NLP), the concepts of parallelism and optimization are fundamental to both fields. In NLP, parallel processing techniques are used to speed up tasks such as language modeling and machine translation, similar to how CUDA is used to accelerate training in deep learning models.

5. **<u>Conclusion:</u>**

The comparison between the sequential and parallel (CUDA) implementations of a Convolutional Neural Network (CNN) for image classification has provided valuable insights into the impact of CUDA and parallel programming on training efficiency in deep learning.

<u>Sequential training:</u>

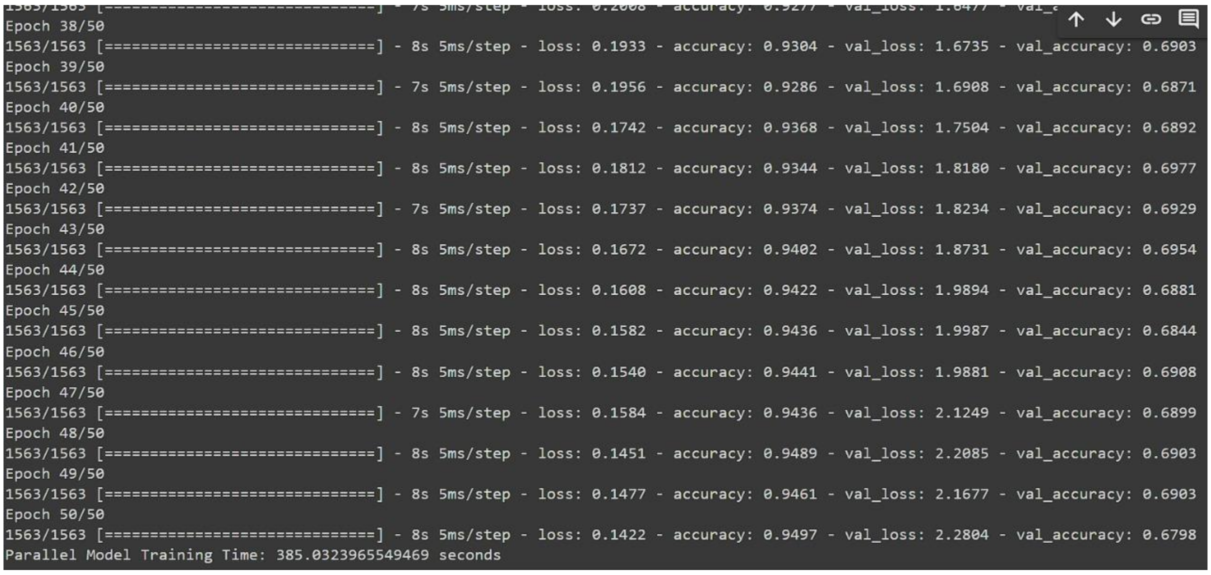
We can see that the time taken to train the sequential model for 15 epochs is 1260.24 seconds



```
Epoch 1/15
1563/1563 [==============================] - 78s 49ms/step - loss: 1.5175 - accuracy: 0.4473 - val_loss: 1.2493 - val_accuracy: 0.5537
Epoch 2/15
1563/1563 [==============================] - 83s 53ms/step - loss: 1.1561 - accuracy: 0.5922 - val_loss: 1.0544 - val_accuracy: 0.6280
Epoch 3/15
1563/1563 [==============================] - 87s 56ms/step - loss: 0.9860 - accuracy: 0.6524 - val_loss: 0.9478 - val_accuracy: 0.6737
Epoch 4/15
1563/1563 [==============================] - 112s 71ms/step - loss: 0.8804 - accuracy: 0.6906 - val_loss: 0.8874 - val_accuracy: 0.6932
Epoch 5/15
1563/1563 [==============================] - 99s 63ms/step - loss: 0.8003 - accuracy: 0.7195 - val_loss: 0.9094 - val_accuracy: 0.6897
Epoch 6/15
1563/1563 [==============================] - 70s 45ms/step - loss: 0.7375 - accuracy: 0.7413 - val_loss: 0.8678 - val_accuracy: 0.7031
Epoch 7/15
1563/1563 [==============================] - 70s 45ms/step - loss: 0.6873 - accuracy: 0.7599 - val_loss: 0.8231 - val_accuracy: 0.7184
Epoch 8/15
1563/1563 [==============================] - 73s 47ms/step - loss: 0.6378 - accuracy: 0.7752 - val_loss: 0.8512 - val_accuracy: 0.7160
Epoch 9/15
1563/1563 [==============================] - 68s 43ms/step - loss: 0.5946 - accuracy: 0.7923 - val_loss: 0.8329 - val_accuracy: 0.7247
Epoch 10/15
1563/1563 [==============================] - 79s 51ms/step - loss: 0.5567 - accuracy: 0.8050 - val_loss: 0.8673 - val_accuracy: 0.7064
Epoch 11/15
1563/1563 [==============================] - 83s 53ms/step - loss: 0.5212 - accuracy: 0.8157 - val_loss: 0.8742 - val_accuracy: 0.7157
Epoch 12/15
1563/1563 [==============================] - 84s 54ms/step - loss: 0.4828 - accuracy: 0.8298 - val_loss: 0.9429 - val_accuracy: 0.7082
Epoch 13/15
1563/1563 [==============================] - 90s 57ms/step - loss: 0.4538 - accuracy: 0.8390 - val_loss: 0.9883 - val_accuracy: 0.7075
Epoch 14/15
1563/1563 [==============================] - 92s 59ms/step - loss: 0.4252 - accuracy: 0.8480 - val_loss: 0.9905 - val_accuracy: 0.7092
Epoch 15/15
1563/1563 [==============================] - 89s 57ms/step - loss: 0.3927 - accuracy: 0.8595 - val_loss: 1.0041 - val_accuracy: 0.7156
Sequential Model Training Time: 1260.2463920116425 seconds
```

Parallel training time:

We can see that the time taken to train the parallel model for 50 epochs is just 385 seconds which is quarter the time taken to train just 15 epochs sequentially.



```
1563/1563 [==============================] - 7s 5ms/step - loss: 0.2008 - accuracy: 0.9277 - val_loss: 1.8477 - val_a
Epoch 38/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1933 - accuracy: 0.9304 - val_loss: 1.6735 - val_accuracy: 0.6903
Epoch 39/50
1563/1563 [==============================] - 7s 5ms/step - loss: 0.1956 - accuracy: 0.9286 - val_loss: 1.6908 - val_accuracy: 0.6871
Epoch 40/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1742 - accuracy: 0.9368 - val_loss: 1.7504 - val_accuracy: 0.6892
Epoch 41/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1812 - accuracy: 0.9344 - val_loss: 1.8180 - val_accuracy: 0.6977
Epoch 42/50
1563/1563 [==============================] - 7s 5ms/step - loss: 0.1737 - accuracy: 0.9374 - val_loss: 1.8234 - val_accuracy: 0.6929
Epoch 43/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1672 - accuracy: 0.9402 - val_loss: 1.8731 - val_accuracy: 0.6954
Epoch 44/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1608 - accuracy: 0.9422 - val_loss: 1.9894 - val_accuracy: 0.6881
Epoch 45/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1582 - accuracy: 0.9436 - val_loss: 1.9987 - val_accuracy: 0.6844
Epoch 46/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1540 - accuracy: 0.9441 - val_loss: 1.9881 - val_accuracy: 0.6908
Epoch 47/50
1563/1563 [==============================] - 7s 5ms/step - loss: 0.1584 - accuracy: 0.9436 - val_loss: 2.1249 - val_accuracy: 0.6899
Epoch 48/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1451 - accuracy: 0.9489 - val_loss: 2.2085 - val_accuracy: 0.6903
Epoch 49/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1477 - accuracy: 0.9461 - val_loss: 2.1677 - val_accuracy: 0.6903
Epoch 50/50
1563/1563 [==============================] - 8s 5ms/step - loss: 0.1422 - accuracy: 0.9497 - val_loss: 2.2804 - val_accuracy: 0.6798
Parallel Model Training Time: 385.0323965549469 seconds
```

5.1 Key Findings:

Significant Reduction in Training Time: The parallel (CUDA) implementation demonstrated a significant reduction in training time compared to the sequential implementation. This reduction in training time is crucial for accelerating model development and improving productivity in deep learning tasks.

Efficiency of GPU Parallelism: The use of CUDA and GPU parallelism allowed for the efficient utilization of GPU resources, leading to faster computations and improved training efficiency. The parallel implementation leveraged the parallel processing capabilities of GPUs, resulting in a more efficient training process compared to the sequential implementation on a CPU.

Scalability and Generalizability: The findings suggest that the benefits of CUDA and parallel programming are scalable and can be generalized to other deep learning tasks beyond image classification. As models become more complex and datasets grow in size, the use of CUDA and parallel programming becomes increasingly important for achieving efficient training times.

5.2 Implications and Future Directions:

Practical Applications: The results of this study have practical implications for the development of deep learning models in various fields, including computer vision, natural language processing, and robotics. The use of CUDA and parallel programming can lead to faster and more efficient model training, enabling the development of more advanced AI systems.

Further Optimization: While the parallel (CUDA) implementation showed significant improvements in training time, further optimization and tuning of the parallel programming techniques could potentially lead to even greater performance gains. Exploring advanced CUDA features and optimizations could further enhance the efficiency of GPU parallelism in deep learning tasks.

Exploration of Other Parallelization Techniques: Future research could explore the comparison of CUDA-based parallel programming with other parallelization techniques, such as multi-threading or distributed computing, to determine the most effective approach for different deep learning tasks and scenarios.

5.3 Conclusion:

In conclusion, the comparison between the sequential and parallel (CUDA) implementations highlights the significant impact of CUDA and parallel programming on reducing training time for deep learning models. The results demonstrate the importance of efficient training methods in improving productivity and advancing the field of artificial intelligence. By leveraging the parallel processing capabilities of GPUs through CUDA, developers can accelerate model development and achieve faster insights, ultimately leading to more advanced and efficient AI systems.