I. Thành phần của cuDNN

cuDNN (CUDA Deep Neural Network) là một thư viện của NVIDIA được tối ưu hóa cho các tác vụ học sâu (deep learning) trên GPU. Các thành phần chính của cuDNN bao gồm:

1. Tensor Operations: cuDNN cung cấp các hàm cho các phép toán tensor cơ bản, chẳng hạn như phép cộng, phép nhân, phép chia và phép trừ. Các phép toán này được tối ưu hóa để sử dụng tối đa hiệu suất của GPU.
2. Convolution: Đây là một trong những thành phần quan trọng nhất của cuDNN. cuDNN cung cấp các hàm để tính toán các phép chập (convolution), sử dụng cả chập 2D và 3D, thường được dùng trong các mô hình học sâu như mạng nơ-ron tích chập (CNN).
3. Activation Functions: cuDNN hỗ trợ các hàm kích hoạt phổ biến trong học sâu như ReLU, Sigmoid, Tanh, Leaky ReLU, v.v. Các hàm này giúp tăng khả năng học của mạng nơ-ron.
4. Normalization: cuDNN hỗ trợ các phép chuẩn hóa như Batch Normalization, giúp cải thiện tốc độ và sự ổn định của việc huấn luyện các mô hình học sâu.
5. Pooling: CuDNN cũng cung cấp các phép toán pooling như max pooling và average pooling, giúp giảm kích thước của đầu ra trong các mạng nơ-ron.
6. RNN Operations: cuDNN cung cấp các hàm cho các phép toán của mạng nơ-ron hồi tiếp (RNN) như LSTM và GRU. Các phép toán này đặc biệt hữu ích trong các tác vụ xử lý chuỗi (sequence processing).
7. Forward/Backward Propagation: cuDNN tối ưu hóa các phép tính lan truyền xuôi (forward) và lan truyền ngược (backward), giúp tăng tốc việc huấn luyện các mô hình học sâu.
8. Autotuning: cuDNN cung cấp tính năng tự động điều chỉnh (autotuning), giúp xác định cách thức tính toán tối ưu nhất cho từng tác vụ cụ thể trên các loại GPU khác nhau.

Nhờ vào các thành phần này, cuDNN giúp tăng tốc các phép toán học sâu trên GPU, giảm thiểu thời gian huấn luyện và cải thiện hiệu suất mô hình.

II. Các hàm cơ bản

1. Tạo Tensor

- `tf.constant(value)`: Tạo một tensor với giá trị cố định.
- `tf.zeros(shape)`: Tạo một tensor với tất cả giá trị bằng 0.

- `tf.ones(shape)`: Tạo một tensor với tất cả giá trị bằng 1.
- `tf.random.normal(shape, mean, stddev)`: Tạo một tensor với giá trị ngẫu nhiên theo phân phối chuẩn.
- `tf.random.uniform(shape, minval, maxval)`: Tạo một tensor với giá trị ngẫu nhiên theo phân phối đều.

2. Duyệt Tensor

- `tf.shape(tensor)`: Trả về hình dạng (shape) của tensor.
- `tf.rank(tensor)`: Trả về bậc (rank) của tensor.
- `tf.size(tensor)`: Trả về số lượng phần tử trong tensor.

3. Thao tác với tensor

- `tf.reshape(tensor, shape)`: Đổi hình dạng của tensor.
- `tf.transpose(tensor)`: Chuyển vị của tensor.
- `tf.expand_dims(tensor, axis)`: Thêm một chiều vào tensor tại vị trí chỉ định.
- `tf.squeeze(tensor)`: Loại bỏ các chiều có kích thước 1.

4. Phép toán cơ bản

- `tf.add(tensor1, tensor2)`: Phép cộng 2 tensor.
- `tf.subtract(tensor1, tensor2)`: Phép trừ 2 tensor.
- `tf.multiply(tensor1, tensor2)`: Phép nhân 2 tensor.
- `tf.divide(tensor1, tensor2)`: Phép chia 2 tensor.
- `tf.pow(tensor, exponent)`: Lũy thừa của tensor.
- `tf.sqrt(tensor)`: Căn bậc hai của tensor.

5. Phép toán thống kê

- `tf.reduce_sum(tensor)`: Tính tổng các phần tử trong tensor.
- `tf.reduce_mean(tensor)`: Tính giá trị trung bình của các phần tử trong tensor.
- `tf.reduce_max(tensor)`: Tìm giá trị lớn nhất trong tensor.
- `tf.reduce_min(tensor)`: Tìm giá trị nhỏ nhất trong tensor.

6. Cập nhật giá trị trong tensor

- `tf.assign(variable, value)`: Cập nhật giá trị của một biến.

- `tf.scatter_add(tensor, indices, updates)`: Cập nhật giá trị của tensor tại các chỉ số cụ thể.

7. So sánh

- `tf.equal(tensor1, tensor2)`: Kiểm tra xem 2 tensor có bằng nhau không.
- `tf.not_equal(tensor1, tensor2)`: Kiểm tra xem 2 tensor có khác nhau không.
- `tf.greater(tensor1, tensor2)`: Kiểm tra xem tensor1 có lớn hơn tensor2 không.
- `tf.less(tensor1, tensor2)`: Kiểm tra xem tensor1 có nhỏ hơn tensor2 không.

- Convolution
- Convolution 2D:

```
import tensorflow as tf

# Tensor đầu vào

input_tensor = tf.random.normal([1, 32, 32, 3])  # Batch size, Height, Width, Channels

filter_tensor = tf.random.normal([3, 3, 3, 64])  # Filter size (3x3), Input channels (3), Output channels (64)

# Tích chập

conv_output = tf.nn.conv2d(

    input=input_tensor,

    filters=filter_tensor,

    strides=[1, 1, 1, 1],

    padding='SAME'

)
```

```
print(conv_output.shape)
```

- Pooling
- Max Pooling:

```
max_pool_output = tf.nn.max_pool2d(
    input=input_tensor,
    ksize=[1, 2, 2, 1],    # Kích thước cửa sổ
    strides=[1, 2, 2, 1], # Bước nhảy
    padding='SAME'
)
print(max_pool_output.shape)
```

- ReLU Activation

```
relu_output = tf.nn.relu(conv_output)
```

- Batch Normalization:

```
bn_output =
tf.keras.layers.BatchNormalization()(conv_output)
```

2. PyTorch với CuDNN

- Convolution 2D:

```
import torch
import torch.nn as nn
# Tích chập 2D
conv = nn.Conv2d(in_channels=3, out_channels=64,
kernel_size=3, stride=1, padding=1)
input_tensor = torch.randn(1, 3, 32, 32) # Batch size,
Channels, Height, Width
output_tensor = conv(input_tensor)
```

```
print(output_tensor.shape)
```

- Pooling:
- Max Pooling:

```
pool = nn.MaxPool2d(kernel_size=2, stride=2)

pooled_output = pool(output_tensor)

print(pooled_output.shape)
```

- Batch Normalization:

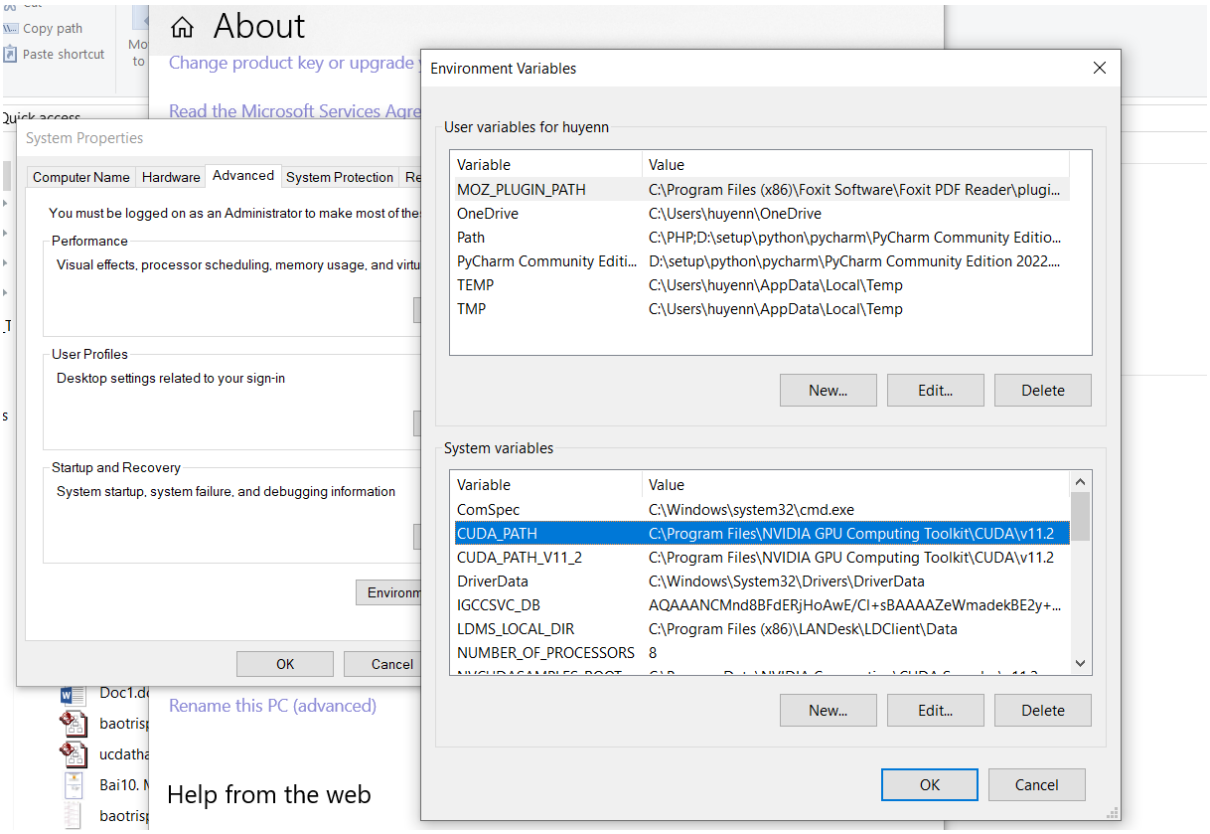
```
bn = nn.BatchNorm2d(num_features=64)

bn_output = bn(output_tensor)
```

III. Cách cài đặt cuDNN

1. Có GPU

1. Cài đặt Cuda Toolkit phù hợp với hệ điều hành, kiến trúc CPU (x86_64), và phiên bản driver GPU tương ứng
   Đảm bảo đã thêm đường dẫn CUDA vào PATH



2. Cài đặt CuDNN
Đăng nhập bằng tài khoản NVIDIA Developer.
Tải phiên bản CuDNN phù hợp với phiên bản CUDA đã cài đặt.
Giải nén file đã tải xuống.
Sao chép nội dung các file bin, include, lib từ thư mục CuDNN vào thư mục cài đặt CUDA.
3. Cài đặt và sử dụng framework hỗ trợ GPU như TensorFlow hoặc PyTorch.

TensorFlow:

Cài đặt phiên bản hỗ trợ GPU:

```
pip install tensorflow==<version>
```

Kiểm tra:

```
import tensorflow as tf
print(tf.config.list_physical_devices('GPU'))
```

PyTorch:

Cài đặt phiên bản hỗ trợ CUDA:

```
pip install torch torchvision torchaudio --index-
url https://download.pytorch.org/whl/cuXX
```

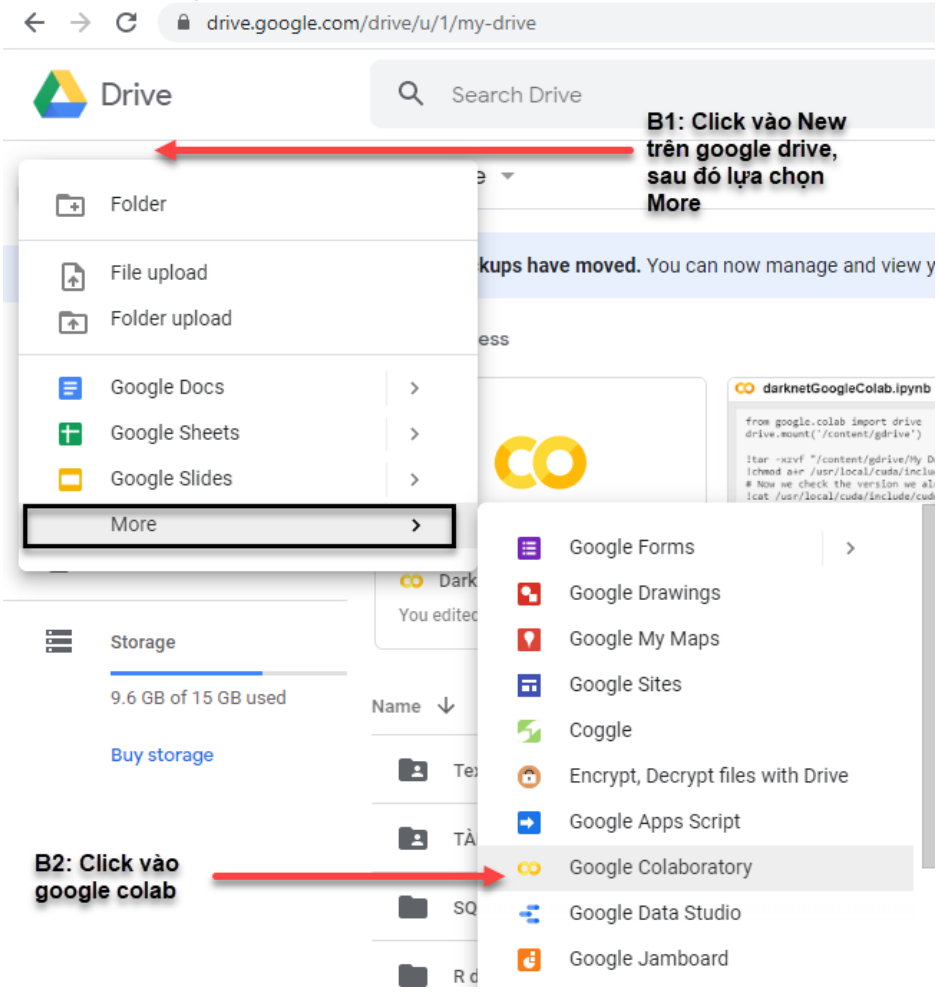
Kiểm tra

```
import torch
```

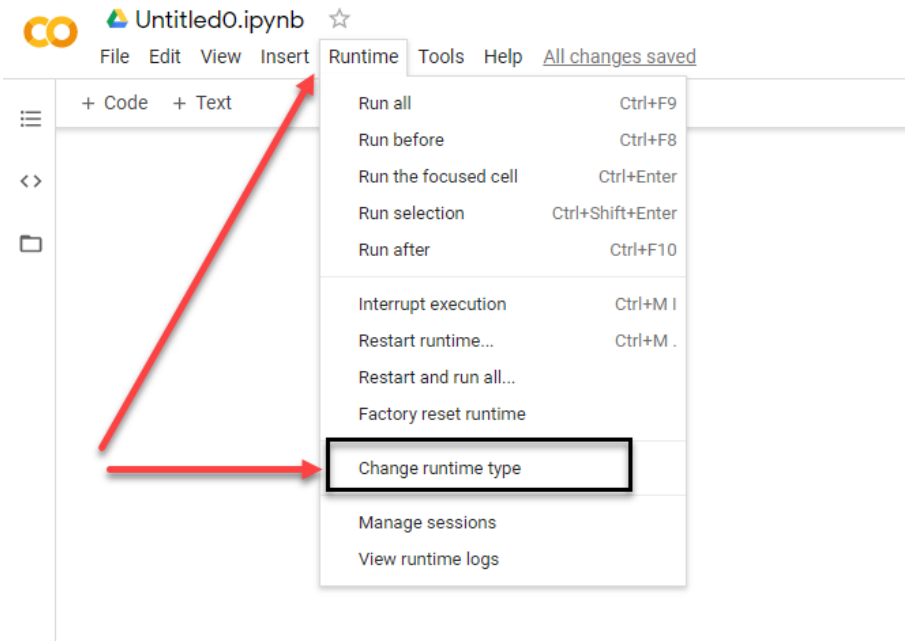
```
print(torch.cuda.is_available())
```

2. Với Google Colab

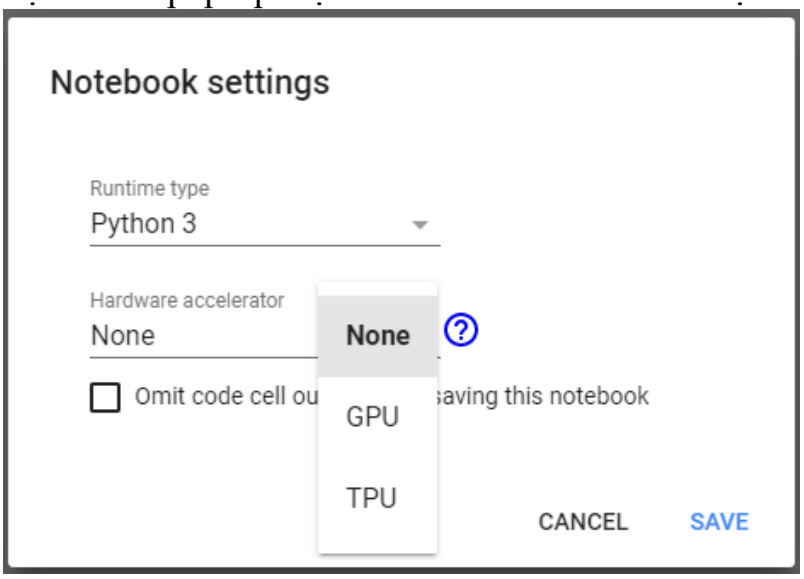
- tạo một google colab. Các bạn vào google drive, sau đó click vào New > More > Google colab.



- Cho phép GPU trên google colab



Tại cửa sổ pop-up mục Hardware accelerator ta lựa chọn GPU và save.



IV. Khái niệm về GPU

GPU (Graphics Processing Unit) là một thiết bị phần cứng chuyên dụng, được thiết kế để xử lý đồ họa và các tác vụ tính toán yêu cầu tính toán song song, đặc biệt là trong các hệ thống tính toán hiệu năng cao với GPGPU (General-Purpose Graphics Processing Unit). CUDA (Compute Unified Device Architecture) là một nền tảng

và mô hình lập trình được NVIDIA phát triển, giúp tận dụng GPU để thực hiện các phép tính không chỉ liên quan đến đồ họa mà còn có thể áp dụng trong nhiều lĩnh vực khác như AI, học máy, khoa học dữ liệu, và tính toán khoa học.

Các đặc điểm của GPU:

- **Xử lý song song**: GPU có hàng ngàn lõi xử lý nhỏ, cho phép xử lý nhiều tác vụ cùng lúc, khác biệt so với CPU chỉ có vài lõi.
- **Hiệu năng cao**: GPU rất mạnh trong việc xử lý các phép toán đơn giản nhưng đồng thời trên số lượng lớn dữ liệu, giúp tăng tốc độ tính toán.
- **Bộ nhớ chuyên dụng**: GPU có bộ nhớ riêng biệt, như VRAM, giúp xử lý dữ liệu đồ họa và các phép toán nhanh chóng hơn.
- **Ứng dụng đa dạng**: GPU không chỉ dùng cho đồ họa mà còn được sử dụng trong các tác vụ như deep learning, xử lý video, và khai thác tiền điện tử (cryptocurrency mining).

1. Kiến Trúc GPU

GPU có một cấu trúc phức tạp với các thành phần sau:

- **Streaming Multiprocessor (SM)**:
  - **SM** là đơn vị xử lý song song chính trên GPU. Mỗi **SM** có các lõi xử lý (cores), bộ nhớ nhanh (register file), bộ nhớ chia sẻ (shared memory), và một bộ lập lịch (warp scheduler).
  - **SM** điều phối việc thực thi các **warp**, là nhóm 32 thread xử lý đồng thời.
- **Lõi xử lý (Cores)**:
  - Các lõi xử lý trong **SM** thực hiện các phép toán đơn giản nhưng có thể xử lý đồng thời hàng nghìn thread.
- **Warp**:
  - Một **warp** là một nhóm 32 thread chạy đồng thời trên GPU. GPU thực thi các thread trong warp một cách đồng bộ, tức là tất cả các thread trong một warp thực hiện cùng một lệnh tại một thời điểm.
- **Thread**:
  - Là đơn vị xử lý nhỏ nhất trên GPU. Mỗi thread có bộ nhớ riêng (register) và thực hiện một công việc độc lập.
- **Bộ nhớ (Memory)**: GPU có nhiều cấp bộ nhớ để tối ưu hóa hiệu suất, bao gồm:

- **Global Memory**: Bộ nhớ chia sẻ giữa tất cả các thread của GPU, dung lượng lớn nhưng truy cập chậm.
- **Shared Memory**: Bộ nhớ tốc độ cao, dung lượng nhỏ, chia sẻ giữa các thread trong cùng một block.
- **Register**: Bộ nhớ nhanh nhất, dung lượng nhỏ, mỗi thread có một bộ nhớ riêng.
- **Local Memory**: Thực chất là bộ nhớ trong global memory, dung lượng dành cho các biến không đủ chứa trong register.
- **Constant Memory**: Bộ nhớ chỉ đọc, thích hợp cho dữ liệu không thay đổi trong suốt quá trình chạy.

2. Cách GPU Xử Lý Công Việc

- **SIMT (Single Instruction, Multiple Threads)**:
  - GPU thực thi các lệnh theo mô hình **SIMT**, nghĩa là một nhóm 32 thread (warp) thực hiện cùng một lệnh, mặc dù mỗi thread có thể xử lý dữ liệu khác nhau.
- **Đồng bộ và giao tiếp giữa các thread**:
  - Các thread trong cùng một block có thể đồng bộ với nhau thông qua lệnh __syncthreads() và chia sẻ dữ liệu qua **shared memory**.

3. Hiệu Suất và Tối Ưu Hóa

- **Xử lý song song**:
  - GPU có khả năng thực thi hàng nghìn thread đồng thời, giúp xử lý các tác vụ tính toán nặng với tốc độ cao, đặc biệt trong các ứng dụng như học máy, xử lý hình ảnh và video, và tính toán khoa học.
- **Tài nguyên hạn chế**:
  - Mỗi **SM** có số lượng register, shared memory và thread nhất định. Sự tối ưu hóa việc phân phối tài nguyên giữa các thread và block rất quan trọng để tận dụng tối đa khả năng của GPU.

4. So Sánh với CPU

- **CPU**: Có ít lõi (thường từ 4 đến 16) nhưng mạnh mẽ trong xử lý các tác vụ tuần tự, đa nhiệm, và xử lý các lệnh phức tạp.
- **GPU**: Có hàng ngàn lõi xử lý nhỏ, mạnh mẽ trong việc xử lý song song nhiều tác vụ đơn giản, thích hợp cho các tác vụ tính toán quy mô lớn như đồ họa 3D, học sâu (deep learning), và khai thác tiền điện tử.

5. Ứng Dụng GPU

- **Đồ họa và Gaming**: GPU giúp render các hình ảnh 3D, video, và hiệu ứng đồ họa trong game với chất lượng cao.
- **AI và Machine Learning**: GPU tăng tốc quá trình huấn luyện các mô hình học máy, đặc biệt là với các mạng neural sâu (deep neural networks).
- **Xử lý Video**: GPU cũng hỗ trợ tăng tốc độ xử lý và chuyển mã video (video encoding/decoding).
- **Khoa học và Tính toán khoa học**: GPU được sử dụng trong các phép tính khoa học phức tạp, mô phỏng, và phân tích dữ liệu lớn.
- **Khai thác Tiền điện tử**: GPU là công cụ chính trong việc khai thác các loại tiền điện tử như Bitcoin và Ethereum nhờ khả năng xử lý song song.

6. Tính Mở Rộng và Phân Tán

- **Sử dụng nhiều GPU**: Trong các hệ thống tính toán hiệu năng cao, nhiều GPU có thể được kết nối với nhau để tăng tốc khả năng xử lý, ví dụ trong các cụm máy chủ hoặc siêu máy tính.
- **OpenCL và CUDA**: GPU có thể lập trình qua các nền tảng như CUDA (cho GPU NVIDIA) và OpenCL (cho các GPU khác), giúp mở rộng khả năng tính toán và ứng dụng đa dạng.

- **Giao tiếp giữa host và device**

Host: Là máy tính với CPU và bộ nhớ chính (RAM), chịu trách nhiệm điều khiển toàn bộ quá trình tính toán. Host sẽ quản lý việc chuyển dữ liệu từ bộ nhớ chính sang bộ nhớ của GPU (Device), khởi tạo và điều phối các kernel (hàm tính toán song song) trên GPU, và nhận kết quả tính toán từ GPU để xử lý tiếp.
Device: Là GPU, có khả năng thực hiện các phép toán song song rất nhanh nhờ vào hàng nghìn luồng xử lý. Device có bộ nhớ riêng (VRAM) dùng để lưu trữ dữ liệu trong quá trình tính toán. Các kernel (hàm tính toán) được thực thi trên GPU, xử lý dữ liệu song song và trả kết quả về Host.

Quá Trình Hoạt Động:

1. **Host** chuyển dữ liệu từ bộ nhớ chính (RAM) lên bộ nhớ GPU (VRAM) để GPU có thể truy xuất và xử lý.
2. **Host** khởi tạo kernel và chỉ định cho **Device** thực thi kernel trên GPU. Kernel sẽ được chia thành các block và thread.
3. **Device** thực thi kernel song song trên các thread và block của GPU.

4. Sau khi tính toán hoàn tất, kết quả sẽ được **Device** trả về **Host**, nơi **Host** tiếp tục xử lý hoặc xuất kết quả.

1. Chuyển dữ liệu từ Host (CPU) sang Device (GPU):

```
def send_to_gpu(data):
    with tf.device('/GPU:0'):
        tensor_gpu = tf.Variable(data)
    return tensor_gpu
```

2. Chuyển dữ liệu từ Device (GPU) về Host (CPU):

```
def send_to_host(tensor_gpu):
    with tf.device('/CPU:0'):
        data_host = tensor_gpu.numpy()
    return data_host
```

3. Chuyển dữ liệu giữa các vùng bộ nhớ của Device (GPU):

```
def copy_device_to_device(tensor_gpu):
    with tf.device('/GPU:0'):
        tensor_copy = tf.identity(tensor_gpu)
    return tensor_copy
```

4. Chuyển dữ liệu giữa các vùng bộ nhớ của Host (CPU):

```
def copy_host_to_host(data):
    return data.copy()
```

- **Cấu trúc chương trình CuDNN**

1. Cài đặt và chuẩn bị môi trường

- Cài đặt **CUDA** và **cuDNN**.
- Cài đặt **Python** và các thư viện liên quan: pycuda, cupy, hoặc tensorflow-gpu.

2. Khởi tạo cuDNN

- Tạo **cuDNN handle** để giao tiếp với cuDNN.
- Khởi tạo các **descriptors** cho dữ liệu, bộ lọc, và lớp.

3. Cấu hình dữ liệu và bộ lọc

- Cấp phát bộ nhớ trên GPU cho dữ liệu đầu vào và đầu ra.
- Thiết lập **tensor descriptors** cho dữ liệu (input, output, filter).

4. Định nghĩa các phép toán

- Cấu hình các lớp như **convolution**, **activation**, hoặc **pooling**.
- Xác định các tham số cần thiết cho các phép toán (ví dụ: padding, stride, dilation).

5. Thực thi phép toán

- Chạy các phép toán như **convolution** hoặc **activation** với cuDNN.
- Tính toán và lưu trữ kết quả trên GPU.

6. Dọn dẹp tài nguyên

- Giải phóng bộ nhớ GPU.
- Hủy các **descriptors** và **handle**.

7. Tích hợp cuDNN với các framework

- Nếu sử dụng **TensorFlow/PyTorch**, chỉ cần đảm bảo rằng các phép toán sử dụng cuDNN đã được tích hợp sẵn và sẽ tự động tối ưu hóa cho GPU.

8. Kiểm tra và tối ưu hóa

- Kiểm tra xem cuDNN có được sử dụng chính xác trên GPU không.
- Tối ưu hóa các tham số (ví dụ: kích thước batch, kiểu dữ liệu) để cải thiện hiệu suất.

V. Chương trình ví dụ

- Bài toán nhân ma trận

```
import tensorflow as tf
import time

print("GPU is", "available" if
tf.config.list_physical_devices('GPU') else "NOT
AVAILABLE")
```

```
matrix1 = tf.random.normal([10000, 10000])
matrix2 = tf.random.normal([10000, 10000])  # Thêm dấu
cách giữa '1000, 1000'

def matmul_on_cpu(matrix1, matrix2):
    with tf.device('/CPU:0'):
        start_time = time.time()
        result = tf.matmul(matrix1, matrix2)
        end_time = time.time()
    return result, end_time - start_time

def matmul_on_gpu(matrix1, matrix2):
    with tf.device('/GPU:0'):
        start_time = time.time()
        result = tf.matmul(matrix1, matrix2)
        result.numpy()  # Không cần 'numpy()' nếu chỉ
muốn kiểm tra thời gian
        end_time = time.time()
    return result, end_time - start_time

result_cpu, cpu_time = matmul_on_cpu(matrix1, matrix2)
print(f"Thời gian thực hiện trên CPU: {cpu_time:.4f}
giây")

if tf.config.list_physical_devices('GPU'):
    result_gpu, gpu_time = matmul_on_gpu(matrix1,
matrix2)
    print(f"Thời gian thực hiện trên GPU:
{gpu_time:.4f} giây")
else:

    print("Không có GPU khả dụng để so sánh.")
```

Kết quả:

```
GPU is available

Thời gian thực hiện trên CPU: 20.4766 giây
```

```
Thời gian thực hiện trên GPU: 1.0831 giây
```