Tổng hợp CuPy và Numba

# CuPy

1. Các hàm cơ bản của CuPy:  
 Gọi thư viện : import cupy as cp  
1.1. Tạo mảng:  
- cp.array(object, dtype=None, copy=True, order='K'): Tạo mảng từ các dữ liệu có sẵn (ví dụ: list, tuple, NumPy array).  
 a = cp.array([1, 2, 3])  
  
- cp.zeros(shape, dtype=float, order='C'): Tạo mảng toàn bộ giá trị là 0.  
 a = cp.zeros((3, 3))  
  
- cp.ones(shape, dtype=float, order='C'): Tạo mảng toàn bộ giá trị là 1.  
 a = cp.ones((2, 2))  
  
- cp.empty(shape, dtype=float, order='C'): Tạo mảng chưa được khởi tạo, chứa giá trị rác.  
 a = cp.empty((3, 3))  
  
- cp.eye(N, M=None, dtype=float, order='C'): Tạo ma trận đơn vị.  
 a = cp.eye(3)  
  
- cp.full(shape, fill\_value, dtype=None, order='C'): Tạo mảng với tất cả các phần tử có giá trị bằng fill\_value.  
 a = cp.full((2, 3), 7)  
  
- cp.arange([start,] stop[, step], dtype=None): Tạo mảng với các giá trị cách đều từ start đến stop.  
 a = cp.arange(0, 10, 2)  
  
- cp.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None): Tạo mảng với các giá trị cách đều giữa start và stop.  
 a = cp.linspace(0, 1, 5)  
  
- cp.random.rand(d0, d1, ..., dn): Tạo mảng ngẫu nhiên với phân phối đồng đều trong khoảng [0, 1).  
 a = cp.random.rand(2, 3)  
  
- cp.random.randn(d0, d1, ..., dn): Tạo mảng ngẫu nhiên với phân phối chuẩn (Gaussian).  
 a = cp.random.randn(2, 3)  
  
- cp.random.randint(low, high=None, size=None, dtype=int): Tạo mảng ngẫu nhiên với giá trị nguyên trong khoảng [low, high).  
 a = cp.random.randint(0, 10, size=(3, 3))  
  
1.2. Các phép toán cơ bản:  
- Cộng mảng: c = a + b  
- Nhân mảng: c = a \* b  
- Tích vô hướng (dot product): c = cp.dot(a, b)  
- Phép chia: c = a / b  
- Chuyển mảng về NumPy array: c = a.get()  
  
1.3. Các phép toán với mảng:  
- Sắp xếp mảng: cp.sort(a)  
- Tính tổng: cp.sum(a)  
- Tính trung bình: cp.mean(a)  
- Tính max, min: cp.max(a), cp.min(a)  
- Tính chuẩn L2 (Euclidean norm): cp.linalg.norm(a)  
  
1.4. Sử dụng với bộ nhớ GPU:  
- Chuyển mảng NumPy sang CuPy: a\_np = np.array([1, 2, 3]); a\_cp = cp.array(a\_np)  
- Chuyển mảng CuPy về NumPy: a\_np = a\_cp.get()  
- Cấp phát bộ nhớ trực tiếp trên GPU: a\_gpu = cp.empty\_like(a)  
  
1.5. Các hàm liên quan đến đại số tuyến tính:  
- Tính ma trận nghịch đảo: cp.linalg.inv(a)  
- Tính định thức: cp.linalg.det(a)  
- Giải hệ phương trình tuyến tính: cp.linalg.solve(a, b)  
- Tính eigenvalues và eigenvectors: cp.linalg.eig(a)  
  
1.6. Các hàm thống kê:  
- Tính phương sai và độ lệch chuẩn: cp.var(a), cp.std(a)  
- Tính tích lũy: cp.cumsum(a), cp.cumprod(a)  
  
1.7. Các hàm chuyển đổi mảng:  
- Thay đổi hình dạng (reshape): cp.reshape(a, (2, 3))  
- Chuyển vị ma trận (transpose): cp.transpose(a)  
- Chuyển mảng thành một chiều: cp.ravel(a)  
- Chuyển mảng thành mảng 2D: cp.atleast\_2d(a)  
- Chuyển mảng thành mảng 3D: cp.atleast\_3d(a)  
  
1.8. Sử dụng với GPU (CUDA):  
- Chuyển dữ liệu vào bộ nhớ GPU: a\_gpu = cp.asarray(a)  
- Sử dụng cp.cuda để truy cập bộ nhớ CUDA: import cupy.cuda  
- Truy cập thông tin về GPU: cp.cuda.Device(0).mem\_info()  
  
1.9. Các hàm mở rộng với mảng CuPy:  
- Lưu mảng vào file: cp.save('array.npy', a)  
- Tải mảng từ file: a = cp.load('array.npy')

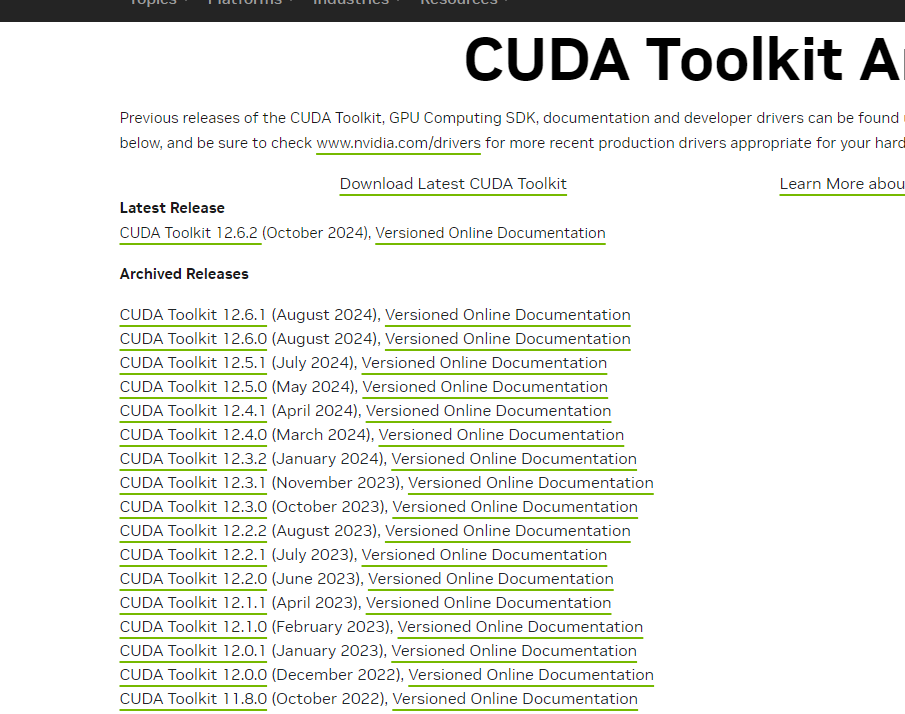
# Numba

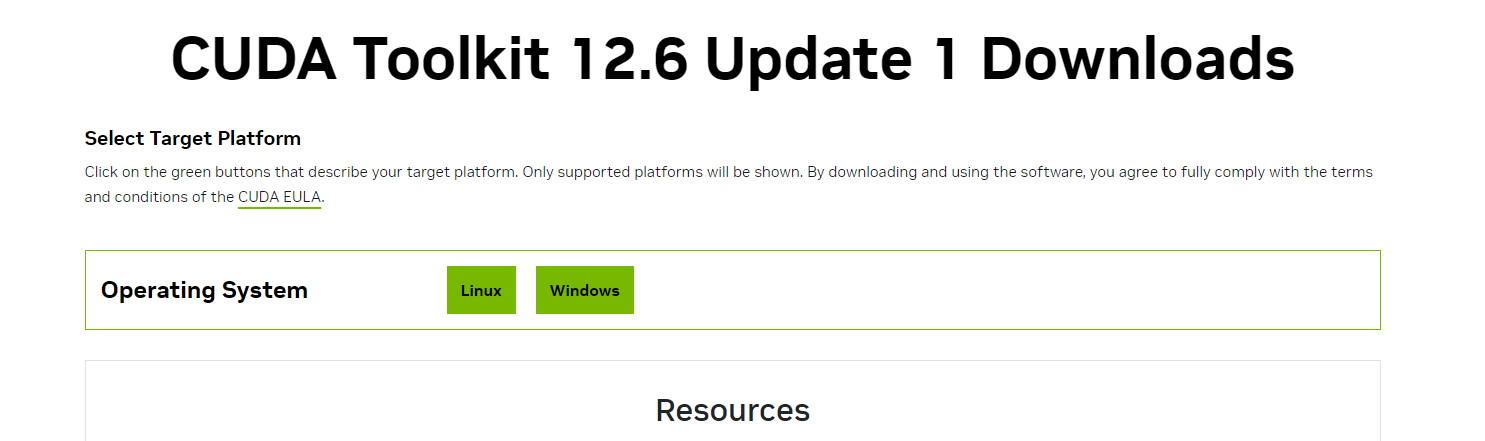
2. Các tính năng cơ bản của Numba:  
  
2.1. Sử dụng @jit để biên dịch mã Python thành mã máy:  
- @jit(nopython=True): Tối ưu hóa và đảm bảo mã chạy không dùng Python interpreter  
 from numba import jit  
 @jit(nopython=True)  
 def add(a, b):  
 return a + b  
  
- Các tham số quan trọng của @jit:  
 - nopython=True: Chỉ sử dụng mã máy thuần túy, không sử dụng tính năng Python.  
 - parallel=True: Cho phép chạy song song (một số phép toán có thể được tối ưu hóa cho GPU hoặc CPU đa lõi).  
 - cache=True: Lưu cache của các hàm đã biên dịch.  
  
2.2. Sử dụng @njit (No Python Mode):  
- @njit là một shortcut cho @jit(nopython=True), giúp biên dịch các hàm trong chế độ "no-Python" để tối ưu hóa tốc độ.  
 from numba import njit  
 @njit  
 def add(a, b):  
 return a + b  
  
2.3. Các hàm @vectorize, @guvectorize:  
- @vectorize: Chuyển một hàm sang dạng vector hóa (tức là áp dụng hàm cho tất cả phần tử trong mảng).  
 from numba import vectorize  
 @vectorize(["float32(float32, float32)"], target="cpu")  
 def add(x, y):  
 return x + y  
  
- @guvectorize: Được sử dụng khi bạn muốn hàm của mình hoạt động trên các mảng n-dimensional.  
 from numba import guvectorize  
 @guvectorize(["(float32, float32) -> (float32)"], "(n), (n) -> (n)")  
 def add(x, y, out):  
 for i in range(len(x)):  
 out[i] = x[i] + y[i]  
  
2.4. Biên dịch các hàm với @cuda.jit cho GPU:  
- @cuda.jit: Biên dịch mã GPU bằng Numba.  
 from numba import cuda  
 @cuda.jit  
 def add\_arrays\_gpu(a, b, c):  
 idx = cuda.grid(1) # Xác định chỉ mục của mỗi thread  
 if idx < a.size:  
 c[idx] = a[idx] + b[idx]  
  
- Cách gọi kernel:  
 import numpy as np  
 a = np.random.rand(1000)  
 b = np.random.rand(1000)  
 c = np.zeros\_like(a)  
  
 a\_device = cuda.to\_device(a)  
 b\_device = cuda.to\_device(b)  
 c\_device = cuda.device\_array\_like(a\_device)  
  
 threads\_per\_block = 256  
 blocks\_per\_grid = (a.size + (threads\_per\_block - 1)) // threads\_per\_block  
 add\_arrays\_gpu[blocks\_per\_grid, threads\_per\_block](a\_device, b\_device, c\_device)  
  
 c\_result = c\_device.copy\_to\_host()

**Các bước cài đặt CUDA với máy đã có Nvidia**

Bước 1: Cài Đặt CUDA Toolkit

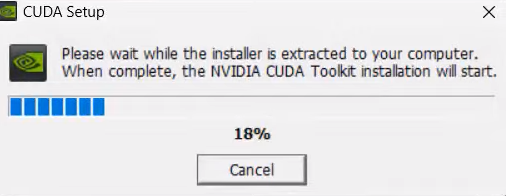
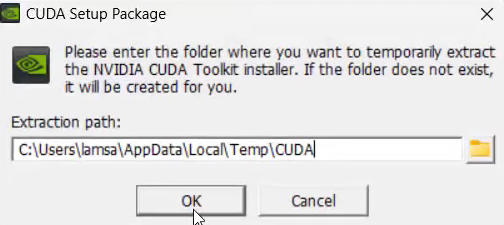
* Truy cập trang chính của CUDA Toolkit: <https://developer.nvidia.com/cuda-toolkit-archive>
* Chọn phiên bản CUDA phù hợp với hệ điều hành của bạn (Windows/Linux/macOS) và phiên bản driver NVIDIA bạn đang sử dụng.
* Tải về và cài đặt theo hướng dẫn của NVIDIA.

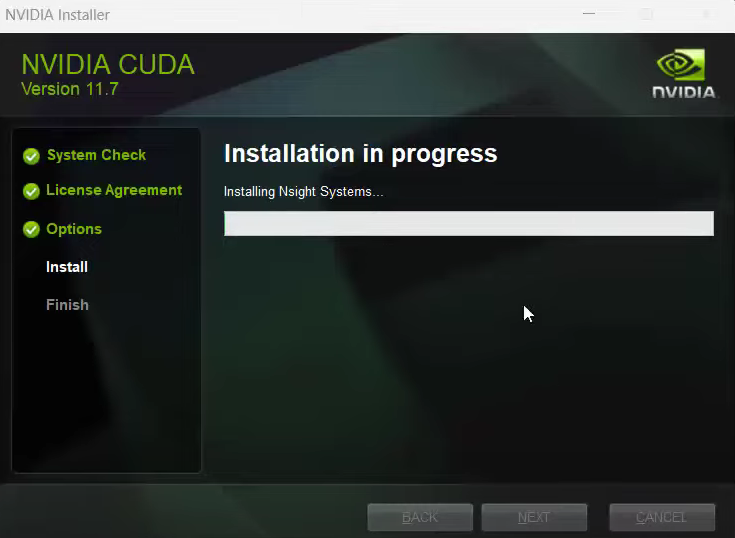


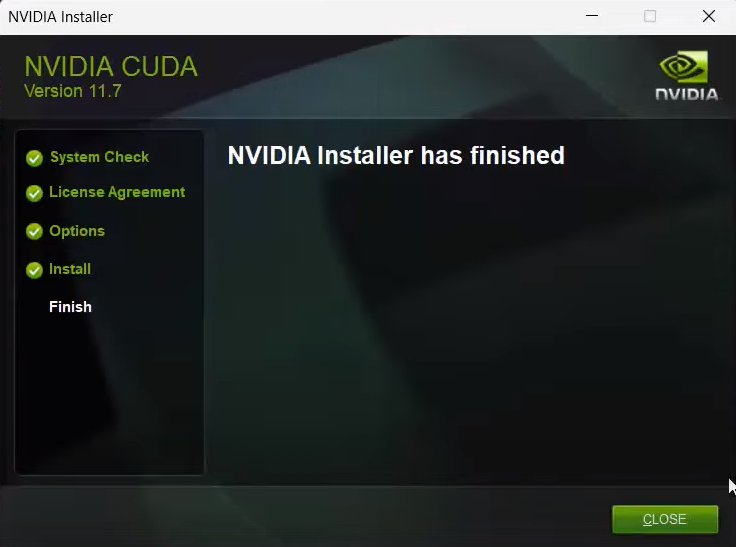


* Mở file và cài đặt:



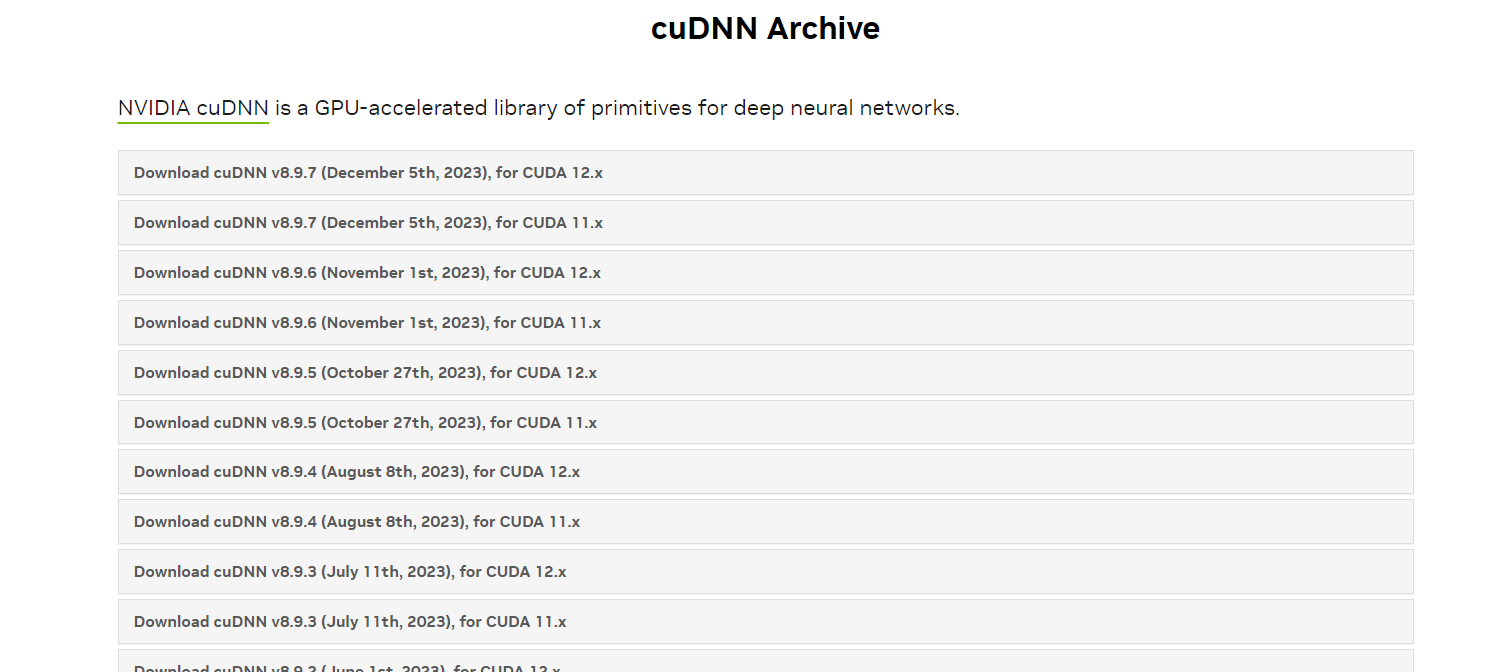


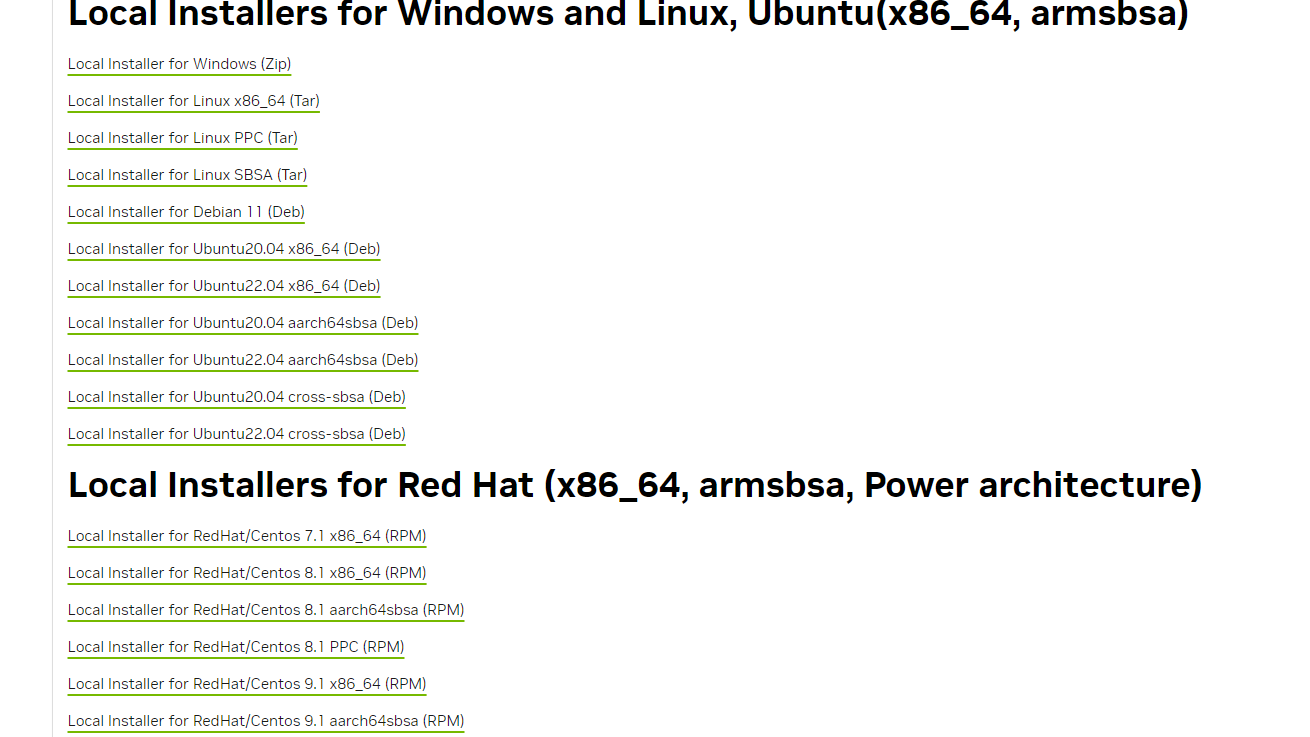




Bước 2: Cài Đặt cuDNN (CUDA Deep Neural Network library)

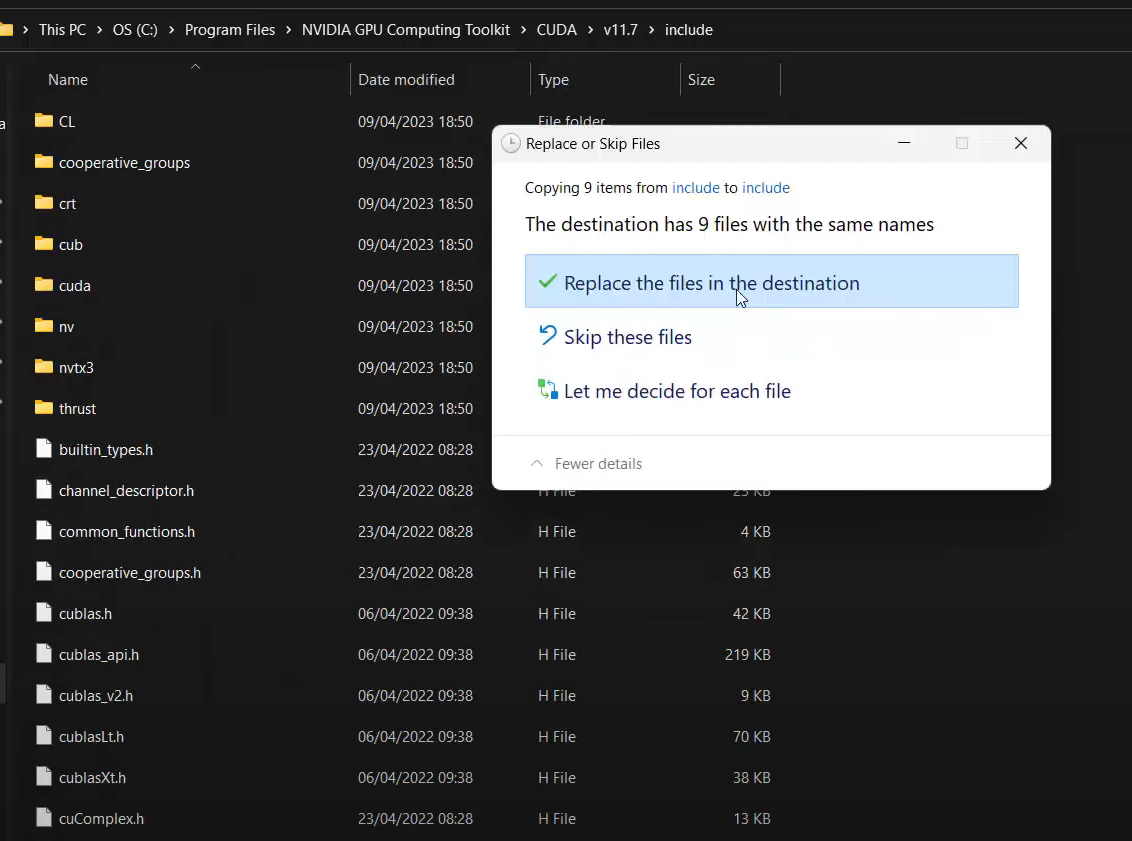
* Truy cập trang tải CuDNN: <https://developer.nvidia.com/rdp/cudnn-archive>
* Chọn phiên bản cuDNN phù hợp với phiên bản CUDA mà bạn đã cài đặt.





* Giải nén file tải về và sao chép các thư mục bin, include, và lib vào thư mục cài đặt CUDA:





Bước 3: Cài Đặt Thư Viện Python

* Để sử dụng GPU với Python, bạn cần cài đặt các thư viện như TensorFlow hoặc PyTorch, tùy thuộc vào nhu cầu của bạn.
  + Cài đặt TensorFlow GPU: pip install tensorflow
  + Cài Đặt PyTorch với GPU: pip install torch torchvision torchaudio

Bước 4: Kiểm Tra Cài Đặt GPU

* Sau khi cài đặt xong CUDA, cuDNN và các thư viện Python, bạn có thể kiểm tra xem các thư viện Python có nhận diện GPU đúng cách không.
* Kiểm Tra với TensorFlow: Mở Python và nhập:
  + import tensorflow as tf
  + print("Num GPUs Available: ", len(tf.config.experimental.list\_physical\_devices('GPU')))

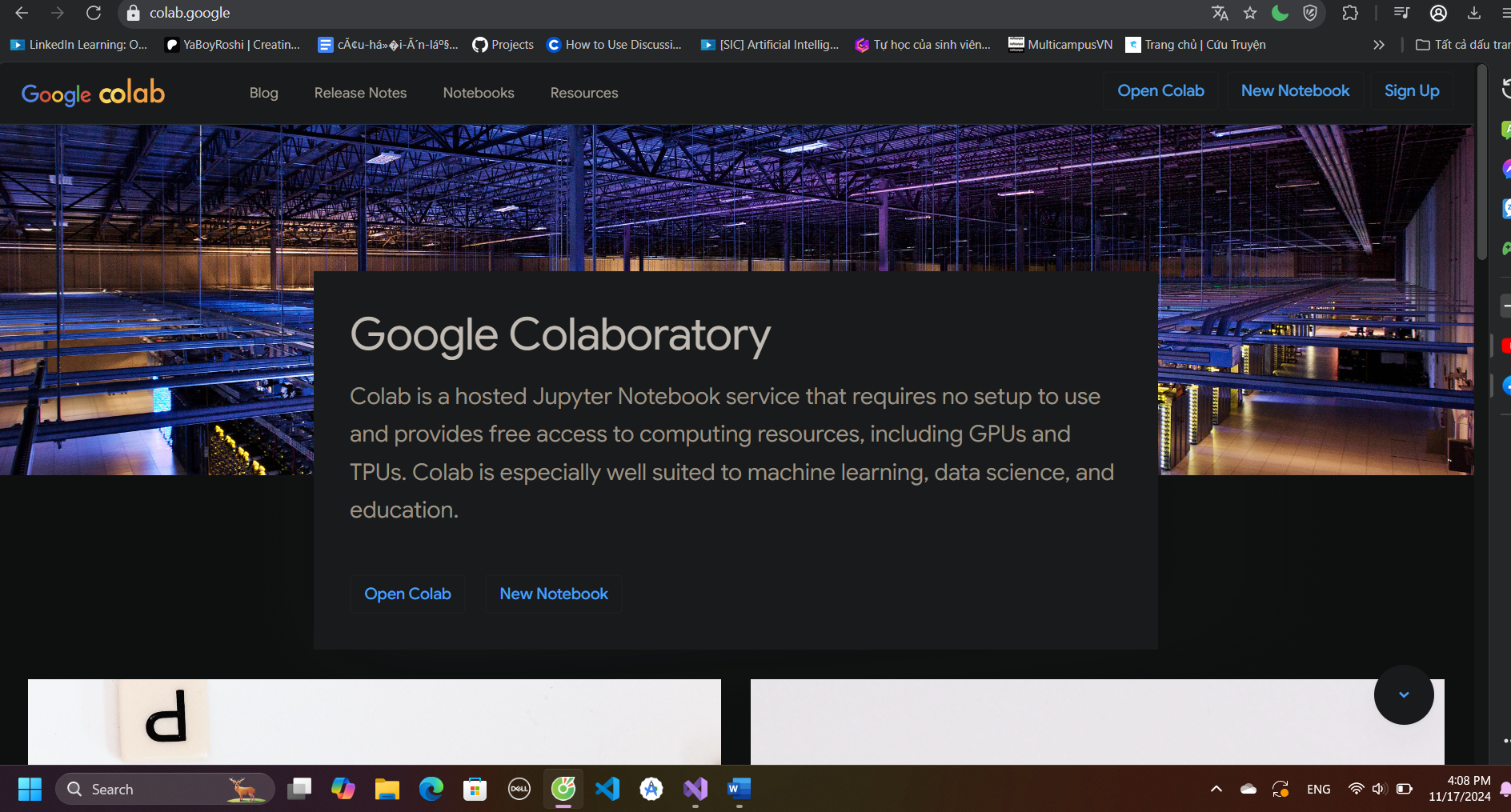
Nếu trả về số lượng GPU > 0, tức là TensorFlow đã nhận diện GPU của bạn.

* Kiểm Tra với PyTorch: Mở Python và nhập:
  + import torch
  + print(torch.cuda.is\_available()) # True nếu có GPU CUDA, False nếu không

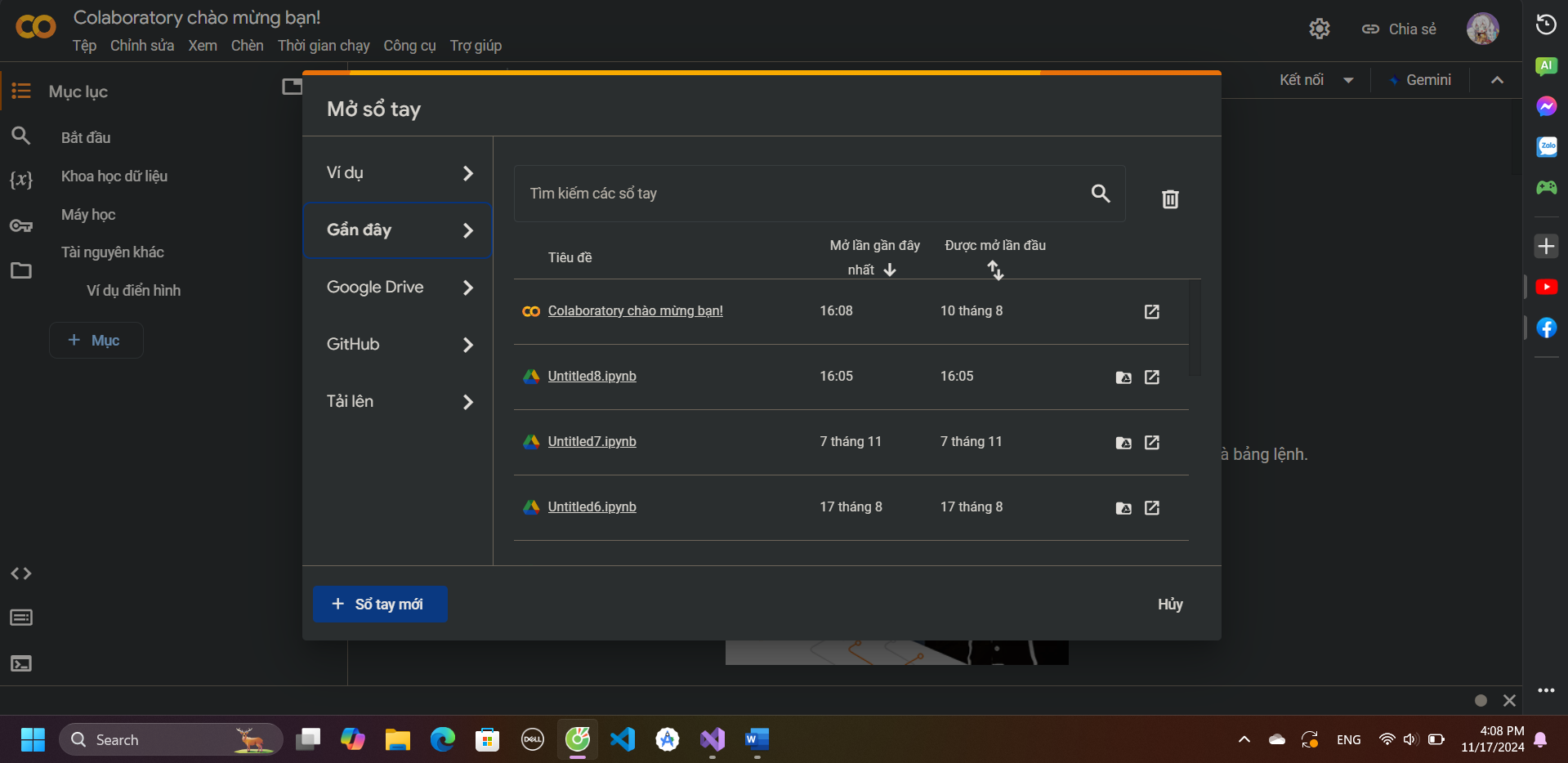
Nếu trả về True, điều này có nghĩa là PyTorch nhận diện GPU của bạn và sẵn sàng sử dụng.

**Các bước cài đặt CUDA trên Colab**

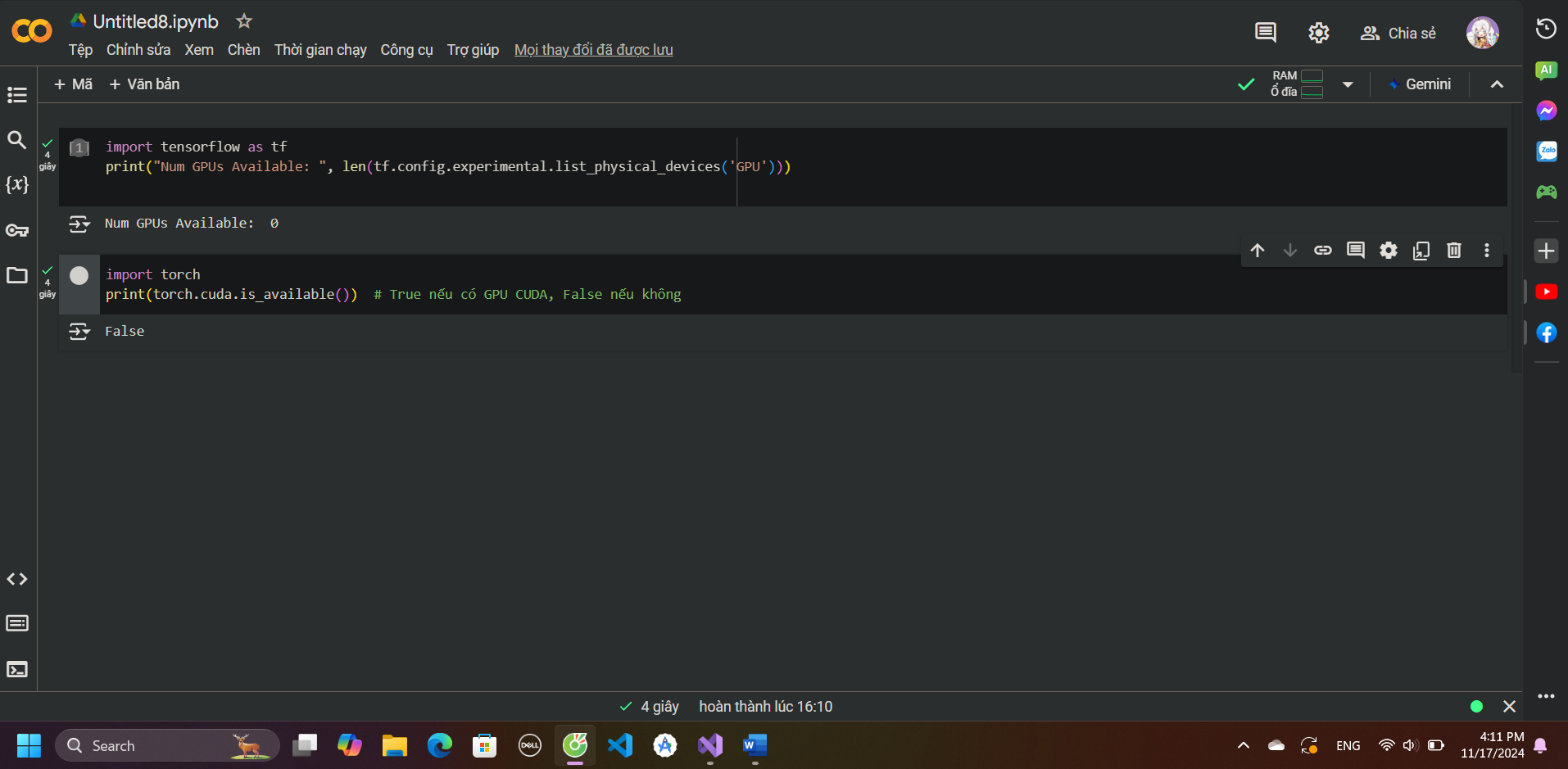
Bước 1: Vô trang chủ colab và Open Colab



Bước 2: Tạo một sổ tay mới:

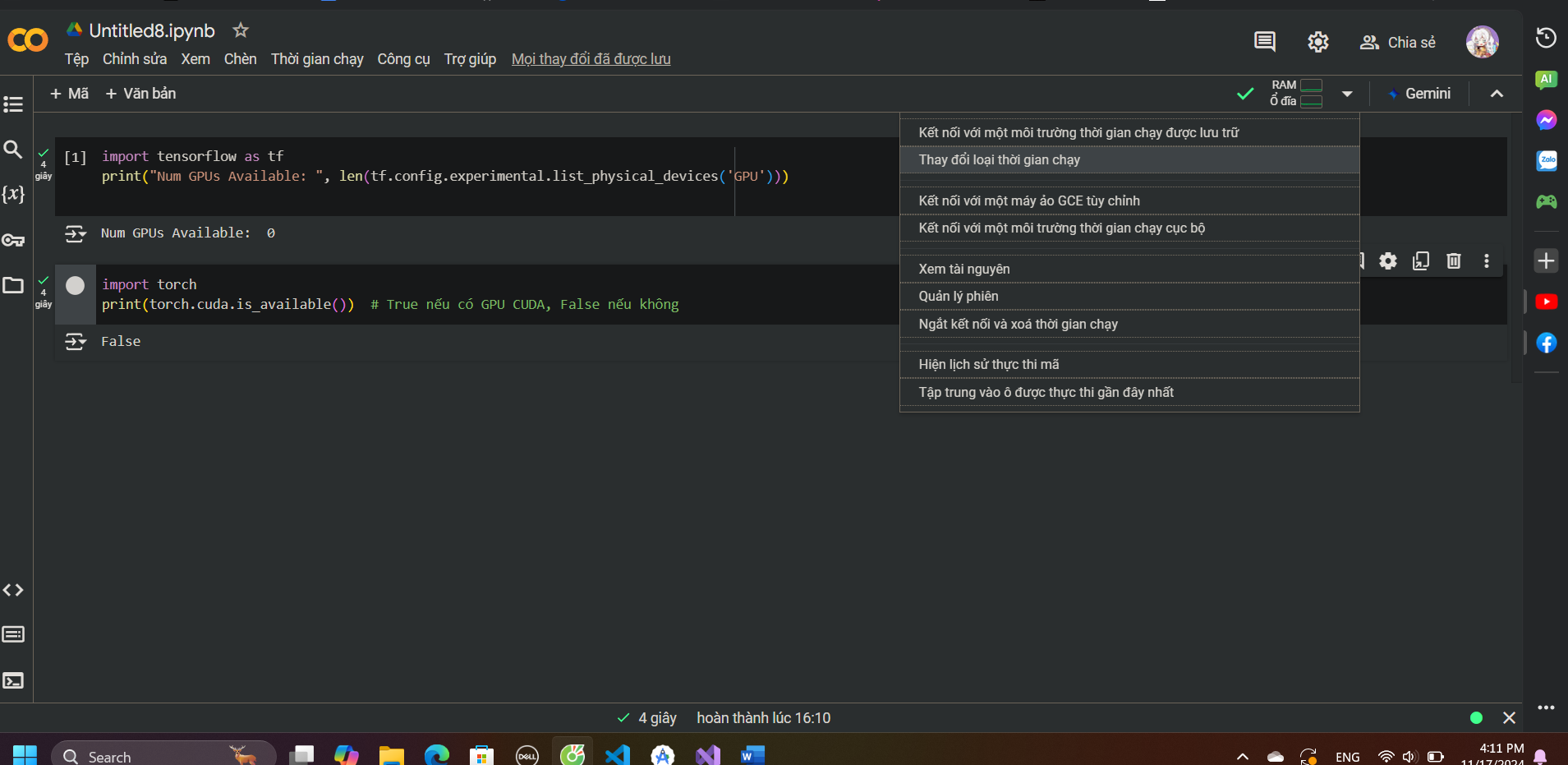


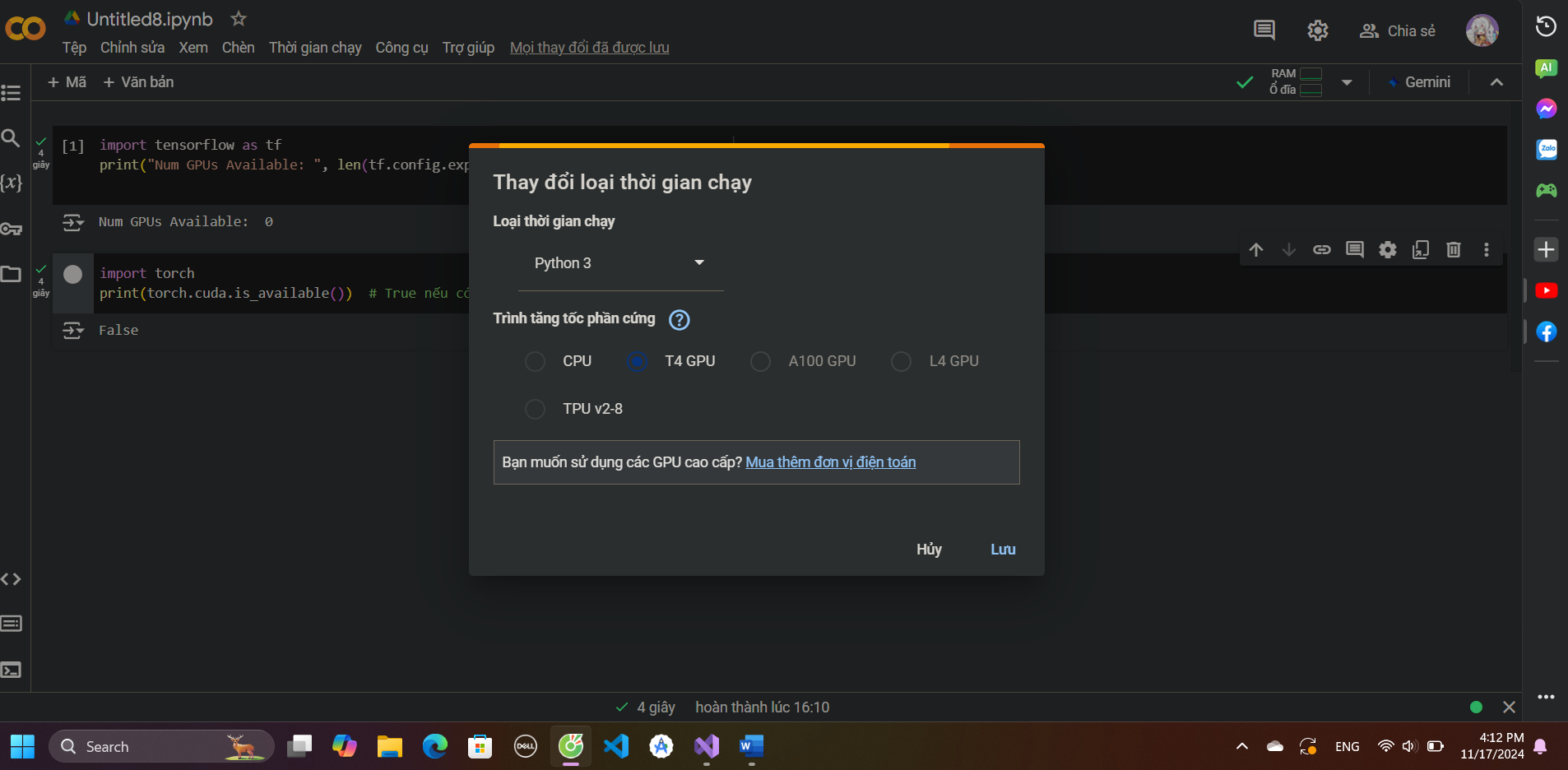
Bước 3: Vì trong colab đã tích hợp sẵn các thư viện nên ta sẽ chạy thử code nhận diện xem có GPU không:



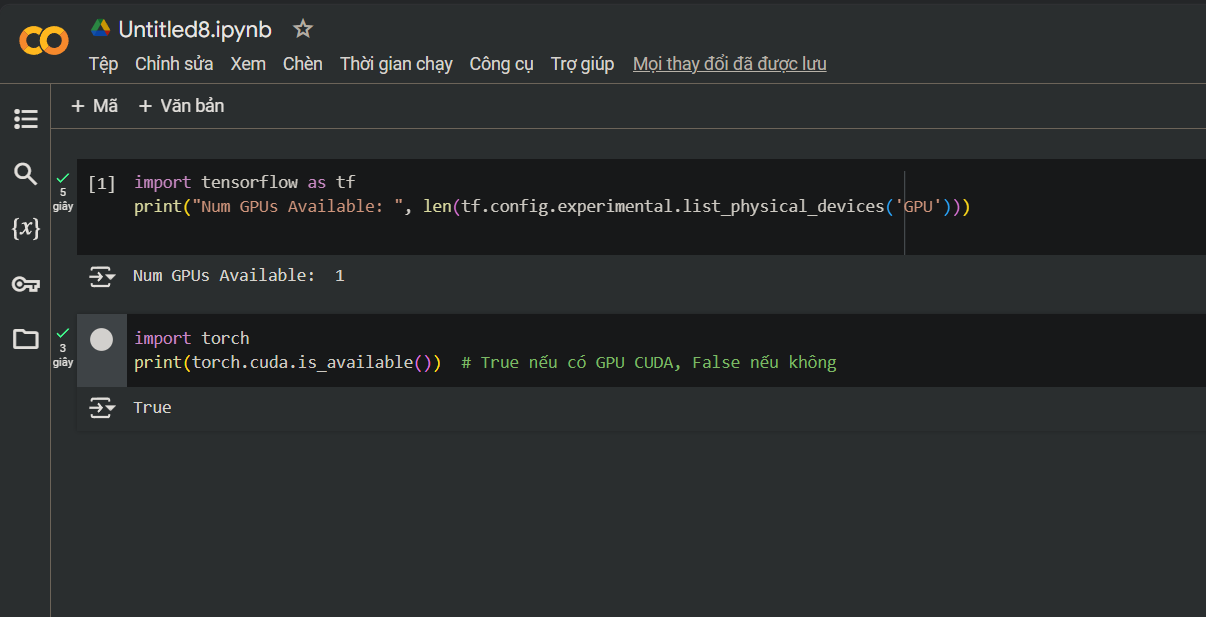
Không có GPU được nhận diện

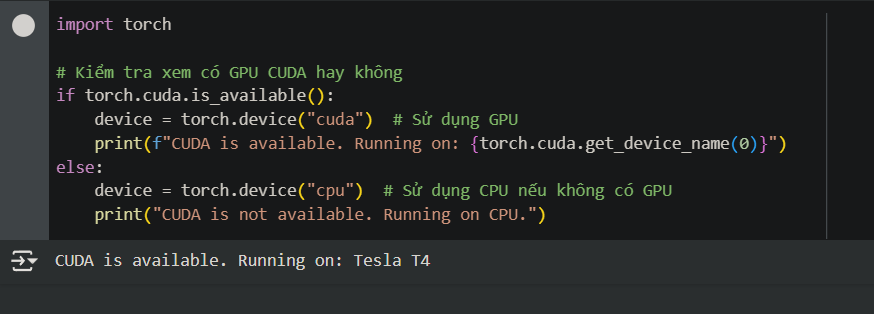
Bước 4: Thay đổi loại thời gian chạy và chuyển sang GPU và lưu





Như vậy chúng ta đã nhận diện được GPU



****