Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM
Khoa Công nghệ Thông tin
---o0o---

Báo cáo đồ án lập trình song song cải thiện thời gian chạy cho các lớp trong mạng CNN
(Phiên bản cuối cùng)

Giáo Viên Hỗ Trợ:
- **Phạm Trọng Nghĩa**
- **Nguyễn Trần Duy Minh**
- **Bùi Duy Đăng**

Nhóm: **M-team**

Sinh Viên:
- **Nguyễn Thế Hưng - 19127154**
- **Trần Minh Thiện - 19127281**
- **Lê Tâm Anh - 19127330**

Mục lục

1/ Link Github:.....3
2/ Bảng phân chia công việc:.....3
3 / Vấn đề đặt ra cho việc song song thuật toán CNN và dataset sử dụng trong thuật toán.....4
4/ Thiết lập song song hóa:.....4
4.1/ Lớp Convolution:.....4
4.1.1/ Các hàm được cài đặt song song:.....4
4.1.2/ Tính đúng đắn của các hàm song song đã cài đặt:.....6
4.1.3/ Đo thời gian chạy so với phiên bản tuần tự:.....7
4.1.3a/ Thời gian chạy của hàm song song so với hàm tuần tự:.....7
4.1.3b/ Thời gian chạy của lớp Convolution trước và sau khi sử dụng hàm song song:.....8
4.2/ Lớp Softmax:.....8
4.2.1/ Các hàm được cài đặt song song:.....8
4.2.2/ Tính đúng đắn của các hàm song song đã cài đặt:.....10
4.2.3/ Đo thời gian chạy so với phiên bản tuần tự:.....13
4.2.3a/ Thời gian chạy của hàm song song so với hàm tuần tự:.....13
4.2.3b/ Thời gian chạy của lớp Softmax trước và sau khi sử dụng hàm song song:.....16
4.3/ Lớp Max-Pooling:.....16
4.4/ Độ chính xác của mạng CNN:.....17
4.5/ So sánh thời gian chạy của các phiên bản:.....18
5/ Kết luận phiên bản tốt nhất:.....21

1/ Link Github:
https://github.com/thientran28/CNN-Parallel

2/ Bảng phân chia công việc:

Thành viên	Công việc	Tỉ lệ hoàn thành
19127330 - Lê Tâm Anh	- Nghiên cứu và cài đặt code tuần tự cho CNN - Song song hóa lớp max-pooling. - Song song hóa hàm nhân ma trận với ma trận trong lớp softmax. - So sánh thời gian chạy của các lớp tuần tự và khi áp dụng các cách song song khác nhau.	100%
19127281 - Trần Minh Thiện	- Nghiên cứu và cài đặt code tuần tự cho CNN - Song song hóa hàm nhân ma trận với vector trong lớp softmax. - Song song hóa hàm vector nhân ma trận trong lớp softmax. - Song song hóa hàm backward cho lớp convolutional. - Kiểm tra tính đúng đắn cho các hàm khi cài đặt song song.	100%
19127154 - Nguyễn Thế Hưng	- Nghiên cứu và cài đặt code tuần tự cho CNN. - Song song hóa hàm forward cho lớp convolutional. - Song song hóa hàm backward cho lớp convolutional. - Viết hàm predict và tính toán độ chính xác của thuật toán khi thực hiện song song hóa. - Kiểm tra tính đúng đắn cho các hàm khi cài đặt song song.	100%

3 / Vấn đề đặt ra cho việc song song thuật toán CNN và dataset sử dụng trong thuật toán

- A Convolutional Neural Network là một dạng deep neural network, thường được áp dụng cho việc phân loại hình ảnh. CNN sẽ bao gồm 3 layer chính đó là convolutional layer, max-pooling layer và fully connected layer và trong CNN ta phải thực hiện nhiều các phép tính giữa các ma trận với nhau nên có thể nói đây là điểm đặc biệt của thuật toán để ta khai thác và thực hiện song song hóa giảm tốc độ chạy cho network.

- Ở trong thuật toán CNN, ta có thể thực hiện song song hóa các hàm forward và backward trong các lớp cùng với việc cải thiện tốc độ của một số hàm tính toán hai ma trận trong lớp CNN để giảm dung lượng bộ nhớ và đạt được tốc độ cao lên với chi phí giảm độ chính xác thấp nhất có thể

- Tập dataset được sử dụng trong bài ở đây sẽ là tập MNIST dataset, một tập dataset chứa khoảng 60,000 các chữ số viết tay thường được sử dụng trong việc training các hệ thống xử lý hình ảnh.
- Trong đồ án này, ta sẽ sử dụng 10000 ảnh để train và kiểm thử cho việc cải thiện thời gian chạy cho thuật toán song song.

4/ Thiết lập song song hóa:

4.1/ Lớp Convolution:

4.1.1/ Các hàm được cài đặt song song:

- Song song hóa hàm forward_prop của lớp Convolution

Hàm forward_prop trước khi triển khai song song

```
def forward_prop(self, image):
    """
    Perform forward propagation for the convolutional layer.
    """
    # Extract image height and width
    image_h, image_w = image.shape
    # Initialize the convolution output volume of the correct size
    convolution_output = np.zeros((image_h-self.kernel_size+1, image_w-self.kernel_size+1, self.kernel_num))
    # Unpack the generator
    patches = self.patches_generator(image)
    for h in range(patches.shape[0]):
        for w in range(patches.shape[1]):
            # Perform convolution for each patch
            convolution_output[h,w] = np.sum(patches[h, w]*self.kernels, axis=(1,2))
    return convolution_output
```

CUDA kernel

```
@cuda.jit
def cnn_forward_kernel(patches, kernels, convolution_output):
    r, c = cuda.grid(2)

    if r < patches.shape[0] and c < patches.shape[1]:
        for k in range(kernels.shape[0]):
            sum = 0
            for i in range(kernels.shape[1]):
                for j in range(kernels.shape[2]):
                    sum += patches[r, c, i, j] * kernels[k, i, j]
            convolution_output[r, c, k] = sum
```

Hàm forward_prop sau khi triển khai song song hóa

```
def forward_prop(self, image):
    """
    Perform forward propagation for the convolutional layer.
    """
    # Extract image height and width
    image_h, image_w = image.shape
    # Initialize the convolution output volume of the correct size
    convolution_output = np.zeros((image_h-self.kernel_size+1, image_w-self.kernel_size+1, self.kernel_num))
    # Unpack the generator
    patches = self.patches_generator(image)
    block_size = (16, 16)
    grid_size = (math.ceil(convolution_output.shape[1] / block_size[0]),
                 math.ceil(convolution_output.shape[0] / block_size[1]))
    cnn_forward_kernel[grid_size, block_size](patches, self.kernels, convolution_output)
    cuda.synchronize()
    return convolution_output
```

- Song song hóa hàm back_prop của lớp Convolution

Hàm back_prop trước khi triển khai song song hóa

```
def back_prop(self, dE_dY, alpha):
    """
    Takes the gradient of the loss function with respect to the output and computes the gradients of the loss function with respect
    to the kernels' weights.
    dE_dY comes from the following layer, typically max pooling layer.
    It updates the kernels' weights
    """
    # Initialize gradient of the loss function with respect to the kernel weights
    dE_dk = np.zeros(self.kernels.shape)
    patches = self.patches_generator(self.image)
    for h in range(patches.shape[0]):
        for w in range(patches.shape[1]):
            for f in range(self.kernel_num):
                dE_dk[f] += patches[h, w] * dE_dY[h, w, f]
    # Update the parameters
    self.kernels -= alpha*dE_dk
    return dE_dk
```

CUDA kernel

```python
@cuda.jit
def cnn_backward_kernel(patches, dE_dY, dE_dk):
    x, y, z = cuda.grid(3)

    if x < dE_dk.shape[0] and y < dE_dk.shape[1] and z < dE_dk.shape[2]:
        temp = 0
        for h in range(patches.shape[0]):
            for w in range(patches.shape[1]):
                temp += patches[h, w, y, z] * dE_dY[h, w, x]
        dE_dk[x, y, z] = temp
```

Hàm back_prop sau khi triển khai song song hóa

```python
def back_prop(self, dE_dY, alpha):
    """
    Takes the gradient of the loss function with respect to the output and computes the gradients of the loss function with respect
    to the kernels' weights.
    dE_dY comes from the following layer, typically max pooling layer.
    It updates the kernels' weights
    """
    # Initialize gradient of the loss function with respect to the kernel weights
    dE_dk = np.zeros(self.kernels.shape)
    patches = self.patches_generator(self.image)
    block_size = (16, 4, 4)
    grid_size = (math.ceil(dE_dk.shape[2] / block_size[0]),
                 math.ceil(dE_dk.shape[1] / block_size[1]),
                 math.ceil(dE_dk.shape[0] / block_size[2]))
    cnn_backward_kernel[grid_size, block_size](patches, dE_dY, dE_dk)
    cuda.synchronize()
    # Update the parameters
    self.kernels -= alpha*dE_dk
    return dE_dk
```

4.1.2/ Tính đúng đắn của các hàm song song đã cài đặt:

Ta kiểm tra thử khi chạy phiên bản song song và tuần tự với một input mẫu để kiểm tra tính đúng đắn của thuật toán.

```
[ ] patches = patches_gen(image)
```

```
[ ] convolution_output_par = np.zeros((image_h-kernel_size+1, image_w-kernel_size+1, kernel_num))
    block_size = (16, 16)
    grid_size = (math.ceil(convolution_output_par.shape[1] / block_size[0]),
                 math.ceil(convolution_output_par.shape[0] / block_size[1]))
    cnn_forward_kernel[grid_size, block_size](patches, kernels, convolution_output_par)
    cuda.synchronize()
```

```
/usr/local/lib/python3.10/dist-packages/numba/cuda/dispatcher.py:488: NumbaPerformanceWarning: Grid size 4 will likely result in
  warn(NumbaPerformanceWarning(msg))
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:885: NumbaPerformanceWarning: Host array used in CUDA
  warn(NumbaPerformanceWarning(msg))
```

```
[ ] convolution_output_seq = np.zeros((image_h-kernel_size+1, image_w-kernel_size+1, kernel_num))
    for h in range(patches.shape[0]):
        for w in range(patches.shape[1]):
        # Perform convolution for each patch
            convolution_output_seq[h,w] = np.sum(patches[h, w]*kernels, axis=(1,2))
```

```
[ ] np.allclose(convolution_output_par, convolution_output_seq)
```

```
True
```

Tương tự với back_prop

```
dE_dY = np.random.random_sample((patches.shape[0], patches.shape[1], kernel_num))
```

```
[ ] dE_dk_par = np.zeros(kernels.shape)
    block_size = (16, 4, 4)
    grid_size = (math.ceil(dE_dk_par.shape[2] / block_size[0]),
                 math.ceil(dE_dk_par.shape[1] / block_size[1]),
                 math.ceil(dE_dk_par.shape[0] / block_size[2]))
    cnn_backward_kernel[grid_size, block_size](patches, dE_dY, dE_dk_par)
    cuda.synchronize()
```

```
/usr/local/lib/python3.10/dist-packages/numba/cuda/dispatcher.py:488: NumbaPerformanceWarning: Grid size 4 will likely result i
  warn(NumbaPerformanceWarning(msg))
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:885: NumbaPerformanceWarning: Host array used in CUDA
  warn(NumbaPerformanceWarning(msg))
```

```
[ ] dE_dk_seq = np.zeros(kernels.shape)
    for h in range(patches.shape[0]):
        for w in range(patches.shape[1]):
            for f in range(kernel_num):
                dE_dk_seq[f] += patches[h, w] * dE_dY[h, w, f]
```

Dùng allclose để kiểm tra 2 ma trận kết qua của 2 phương pháp song song với tuần tự là như nhau.

```
[ ] np.allclose(dE_dk_par, dE_dk_seq)
```

```
True
```

4.1.3/ Đo thời gian chạy so với phiên bản tuần tự:

4.1.3a/ Thời gian chạy của hàm song song so với hàm tuần tự:

Thời gian chạy khi thực hiện song song forward_prop là 18m 43s

```
+ Code + Text
18m
Step 8101. For the last 100 steps: average loss 0.47817934089265823, accuracy 86
Step 8201. For the last 100 steps: average loss 0.2969969171330879, accuracy 92
Step 8301. For the last 100 steps: average loss 0.31113038163372914, accuracy 94
Step 8401. For the last 100 steps: average loss 0.3684142366090459, accuracy 90
Step 8501. For the last 100 steps: average loss 0.42214231837673283, accuracy 89
Step 8601. For the last 100 steps: average loss 0.3541394384475026, accuracy 88
Step 8701. For the last 100 steps: average loss 0.3236827365226807, accuracy 93
Step 8801. For the last 100 steps: average loss 0.32751080632822377, accuracy 91
Step 8901. For the last 100 steps: average loss 0.35431208101171746, accuracy 91
Step 9001. For the last 100 steps: average loss 0.28164165882468467, accuracy 91
Step 9101. For the last 100 steps: average loss 0.3956557292312502, accuracy 87
Step 9201. For the last 100 steps: average loss 0.2499179132240109, accuracy 96
Step 9301. For the last 100 steps: average loss 0.21372544003443716, accuracy 92
Step 9401. For the last 100 steps: average loss 0.33604326211228885, accuracy 93
Step 9501. For the last 100 steps: average loss 0.2447549728965581, accuracy 95
Step 9601. For the last 100 steps: average loss 0.3393489329075653, accuracy 90
Step 9701. For the last 100 steps: average loss 0.29456058215146513, accuracy 90
Step 9801. For the last 100 steps: average loss 0.335957212628658, accuracy 89
Step 9901. For the last 100 steps: average loss 0.4387557664540549, accuracy 88
CPU times: user 18min 51s, sys: 12min 56s, total: 31min 47s
Wall time: 18min 43s
```

Thời gian chạy khi thực hiện song song back_prop là 6m 19s

```
+ Code + Text
6m [13]
Step 8401. For the last 100 steps: average loss 0.32135428673671873, accuracy 93
Step 8501. For the last 100 steps: average loss 0.20302299214592304, accuracy 94
Step 8601. For the last 100 steps: average loss 0.25064263283846944, accuracy 90
Step 8701. For the last 100 steps: average loss 0.25168506391257073, accuracy 93
Step 8801. For the last 100 steps: average loss 0.31686964678901997, accuracy 92
Step 8901. For the last 100 steps: average loss 0.30487912598888156, accuracy 93
Step 9001. For the last 100 steps: average loss 0.32228704940848707, accuracy 91
Step 9101. For the last 100 steps: average loss 0.32661668268652483, accuracy 91
Step 9201. For the last 100 steps: average loss 0.2607285402425165, accuracy 93
Step 9301. For the last 100 steps: average loss 0.2932870836898928, accuracy 93
Step 9401. For the last 100 steps: average loss 0.2899022394999317, accuracy 92
Step 9501. For the last 100 steps: average loss 0.3417074376776739, accuracy 90
Step 9601. For the last 100 steps: average loss 0.23353001082602012, accuracy 95
Step 9701. For the last 100 steps: average loss 0.195939625740269, accuracy 92
Step 9801. For the last 100 steps: average loss 0.3052640254754937, accuracy 89
Step 9901. For the last 100 steps: average loss 0.26257139127876816, accuracy 94
CPU times: user 6min 16s, sys: 4min 37s, total: 10min 53s
Wall time: 6min 19s
```

4.1.3b/ Thời gian chạy của lớp Convolution trước và sau khi sử dụng hàm song song:

Thời gian chạy của thuật toán sau khi implement cả 2 hàm forward_prop với back_prop song song là 4m 58s

```
+ Code  + Text
4m [19] Step 8401. For the last 100 steps: average loss 0.3167434093248779, accuracy 88
        Step 8501. For the last 100 steps: average loss 0.2960809023982045, accuracy 92
        Step 8601. For the last 100 steps: average loss 0.22397623768047434, accuracy 96
        Step 8701. For the last 100 steps: average loss 0.3112073283129049, accuracy 90
        Step 8801. For the last 100 steps: average loss 0.43514272188902936, accuracy 90
        Step 8901. For the last 100 steps: average loss 0.4702521896103024, accuracy 86
        Step 9001. For the last 100 steps: average loss 0.3876907122088437, accuracy 89
        Step 9101. For the last 100 steps: average loss 0.2584062851544872, accuracy 90
        Step 9201. For the last 100 steps: average loss 0.37311165205077357, accuracy 89
        Step 9301. For the last 100 steps: average loss 0.2615947182527604, accuracy 94
        Step 9401. For the last 100 steps: average loss 0.5447012105196578, accuracy 87
        Step 9501. For the last 100 steps: average loss 0.3681705952638081, accuracy 90
        Step 9601. For the last 100 steps: average loss 0.47641916087489844, accuracy 87
        Step 9701. For the last 100 steps: average loss 0.2682022835273742, accuracy 90
        Step 9801. For the last 100 steps: average loss 0.29382394247310356, accuracy 89
        Step 9901. For the last 100 steps: average loss 0.2366722760541514, accuracy 93
        CPU times: user 4min 55s, sys: 3min 38s, total: 8min 33s
        Wall time: 4min 58s
```

4.2/ Lớp Softmax:

4.2.1/ Các hàm được cài đặt song song:

- ***Hàm Dot Product giữa vector và ma trận:***
  Cài đặt hàm nhân giữa vector với ma trận với ý tưởng đơn giản là dùng một grid để duyệt ma trận và tiến hành nhân từng phần tử của vector cho phần tử tương ứng của ma trận rồi cộng chúng lại.

Vector-Matrix Dot Product function

```
[ ]  from numba import cuda, jit
     import numpy as np

     @cuda.jit
     def dot(a, b, c):
       col = cuda.grid(1)
       if (col < b.shape[1]):
         sum = 0.0
         for i in range(b.shape[0]):
           sum += a[i] * b[i, col]
         c[col] = sum
```

Gọi hàm nhân vector với ma trận thay cho hàm tuần tự ở bên trong hàm forward_prop() của lớp Softmax. Đồng bộ quá trình đọc ghi bộ nhớ giữa thiết bị và host bằng cách sử dụng các hàm cuda.to_device() và copy_to_host().

```
C = np.empty(10)
dA = cuda.to_device(image_flattened)
dB = cuda.to_device(self.weight)
dC = cuda.to_device(C)
dot[(self.weight.shape[0]+255)//256, 256](dA,dB,dC)
result = dC.copy_to_host()
first_output = result  + self.bias
```

- ***Hàm nhân ma trận với vector:***
  Cài đặt hàm với ý tưởng tương tự như hàm Vector nhân với ma trận ở phía trên (gọi grid với giá trị là 1 và sử dụng cho ma trận)

Matrix-vector multiply function

```
@cuda.jit
def cu_matrix_vector(A, b, c):
  row = cuda.grid(1)
  if (row < A.shape[0]):
    sum = 0.0
    for i in range(A.shape[1]):
      sum += A[row, i] * b[i]
    c[row] = sum
```

Gọi hàm nhân ma trận với vector thay cho hàm tuần tự ở bên trong hàm back_prop() của lớp Softmax. Đồng bộ quá trình đọc ghi bộ nhớ giữa thiết bị và host bằng cách sử dụng các hàm cuda.to_device() và copy_to_host(). *Ở phiên bản 2, ta sẽ hỉnh sửa số lượng blocks per grid và threads per block cho hàm ma trận nhân vector.*

```
# Matrix-vector multiply function
C = np.empty(dZ_dX.shape[0])
dA = cuda.to_device(dZ_dX)
dB = cuda.to_device(dE_dZ)
dC = cuda.to_device(C)
cu_matrix_vector[(dZ_dX.shape[0]+127)//128, 128](dA,dB,dC)
dE_dX = dC.copy_to_host()
```

Ver 1

```
# Matrix-vector multiply function
C = np.empty(dZ_dX.shape[0])
dA = cuda.to_device(dZ_dX)
dB = cuda.to_device(dE_dZ)
dC = cuda.to_device(C)
cu_matrix_vector[(dZ_dX.shape[0]+15)//16, 16](dA,dB,dC)
dE_dX = dC.copy_to_host()
```

Ver2

- ***Hàm nhân hai ma trận với nhau:***
  Sử dụng 2 grid để duyệt cho hàng(i) và cột(j) của ma trận kết quả, do ma trận có kích thước là (2701,1) và (1,10) nên ta có thể không sử dụng vòng lặp mà gán thẳng index cột của ma trận A là 0 và tương tự index hàng của ma trận B là 0.

```
@cuda.jit
def matmul(A,B,C):
  i,j = cuda.grid(2)
  if i < C.shape[0] and j < C.shape[1]:
    tmp = A[i,0] * B[0,j]
    C[i,j] = tmp
```

Gọi hàm nhân hai trận bên trong lớp hàm back_prop() của lớp softmax để thay thế cho hàm nhân hai ma trận tuần tự ban đầu.

```
# Gradient of loss with respect to weight, bias, input
C = np.empty((dZ_dw[np.newaxis].T.shape[0], dE_dZ[np.newaxis].shape[1]))
dA = cuda.to_device(dZ_dw[np.newaxis].T)
dB = cuda.to_device(dE_dZ[np.newaxis])
dC = cuda.to_device(C)
blockx = int(np.ceil(C.shape[0] / 16))
blocky = int(np.ceil(C.shape[1] / 16))
blockspergrid = (blockx, blocky)
matmul[blockspergrid, (16,16)](dA,dB,dC)
dE_dW = dC.copy_to_host()
```

4.2.2/ Tính đúng đắn của các hàm song song đã cài đặt:

Ta sẽ cần đảm bảo kết quả của thuật toán song song và thuật toán tuần tự là hoàn toàn giống nhau và đúng đắn. Do đó trong phần này, ta sẽ kiểm tra để đảm bảo kết quả của thuật toán song song là hoàn toàn trùng khớp so với thuật toán tuần tự trước đó.

- ***Hàm Dot Product giữa vector và ma trận:***

Trong hàm này, ta sẽ nhân vector image_flattened có kích thước (2704,) và ma trận weight có kích thước là (2704,10). Sau đó, ta sẽ kiểm tra bằng cách trừ kết quả của hai thuật toán với nhau xem có được một mảng chứa tất cả các phần tử là 0 hay không.

```
def forward_prop(self, image):
  self.original_shape = image.shape # stored for backprop
  # Flatten the image
  #print("image: ", image)
  image_flattened = image.flatten()
  #print("image_flattened: ", image_flattened)
  self.flattened_input = image_flattened # stored for backprop

  # Perform matrix multiplication and add bias
  C = np.empty(10)
  dA = cuda.to_device(image_flattened)
  dB = cuda.to_device(self.weight)
  dC = cuda.to_device(C)
  dot[(self.weight.shape[0]+255)//256, 256](dA,dB,dC)
  result = dC.copy_to_host()
  first_output = result  + self.bias
  print("parallel Vector-Matrix dot product", result)
  print("nympy Vector-Matrix dot product: ", np.dot(image_flattened, self.weight))
  print("checking result: ", result - np.dot(image_flattened, self.weight))
  #first_output = np.dot(image_flattened, self.weight) + self.bias
  self.output = first_output
  # Apply softmax activation
  softmax_output = np.exp(first_output) / np.sum(np.exp(first_output), axis=0)

  return softmax_output
```

Dưới đây là kết quả chạy thuật toán, các phần tử của hai thuật toán giống nhau và kết quả của việc trừ hai ma trận là một mảng chứa toàn 0. Từ đó, ta kết luận rằng thuật toán song song cho kết quả trùng với thuật toán tuần tự.

```
Epoch 1 ->
Step 1. For the last 100 steps: average loss 0.0, accuracy 0
parallel Vector-Matrix dot product [ 7.65091763e-04 -2.64196341e-03  1.52499905e-03 -3.33462567e-03
  2.00982050e-03  6.33632861e-03 -1.99527360e-03 -4.06859566e-03
 -1.67544370e-03  6.08582293e-05]
nympy Vector-Matrix dot product:  [ 7.65091763e-04 -2.64196341e-03  1.52499905e-03 -3.33462567e-03
  2.00982050e-03  6.33632861e-03 -1.99527360e-03 -4.06859566e-03
 -1.67544370e-03  6.08582293e-05]
checking result:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
parallel Vector-Matrix dot product [-0.20107429 -0.19821913 -0.19607213 -0.19764763 -0.19709109 -0.19369966
  1.78942994 -0.20366941 -0.20103229 -0.20235043]
nympy Vector-Matrix dot product:  [-0.20107429 -0.19821913 -0.19607213 -0.19764763 -0.19709109 -0.19369966
  1.78942994 -0.20366941 -0.20103229 -0.20235043]
checking result:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
parallel Vector-Matrix dot product [-0.21591121 -0.21555551 -0.21385159 -0.21321116 -0.21621343 -0.211181
  1.9233334  -0.21740663 -0.21382948 -0.21212719]
nympy Vector-Matrix dot product:  [-0.21591121 -0.21555551 -0.21385159 -0.21321116 -0.21621343 -0.211181
  1.9233334  -0.21740663 -0.21382948 -0.21212719]
checking result:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
parallel Vector-Matrix dot product [-0.23648389 -0.23402021 -0.23289614 -0.23690648 -0.23263084 -0.23474061
  1.05431099 -0.2349534   0.82004366 -0.23472026]
nympy Vector-Matrix dot product:  [-0.23648389 -0.23402021 -0.23289614 -0.23690648 -0.23263084 -0.23474061
  1.05431099 -0.2349534   0.82004366 -0.23472026]
```

- ***Hàm nhân ma trận với vector:***

  Cũng tương tự như trên ta sẽ kiểm tra kết quả của thuật toán song song và so sánh với thuật toán tuần tự.

```python
def back_prop(self, dE_dY, alpha):
  for i, gradient in enumerate(dE_dY):
    if gradient == 0:
      continue
    transformation_eq = np.exp(self.output)
    S_total = np.sum(transformation_eq)

    # Compute gradients with respect to output (Z)
    dY_dZ = -transformation_eq[i]*transformation_eq / (S_total**2)
    dY_dZ[i] = transformation_eq[i]*(S_total - transformation_eq[i]) / (S_total**2)

    # Compute gradients of output Z with respect to weight, bias, input
    dZ_dw = self.flattened_input
    dZ_db = 1
    dZ_dX = self.weight

    # Gradient of loss with respect ot output
    dE_dZ = gradient * dY_dZ

    # Gradient of loss with respect to weight, bias, input
    #dE_dw = dZ_dw[np.newaxis].T @ dE_dZ[np.newaxis]
    #dE_db = dE_dZ * dZ_db

    # Matrix-vector multiply function
    C = np.empty(dZ_dX.shape[0])
    dA = cuda.to_device(dZ_dX)
    dB = cuda.to_device(dE_dZ)
    dC = cuda.to_device(C)
    cu_matrix_vector[(dZ_dX.shape[0]+15)//16, 128](dA,dB,dC)
    dE_dX = dC.copy_to_host()
    print("dE_dX parallel: ",dE_dX)
    print("dE_dX: ", dZ_dX @ dE_dZ)

    # Update parameters
    self.weight -= alpha* (dZ_dw[np.newaxis].T @ dE_dZ[np.newaxis])
    self.bias -= alpha * (dE_dZ * dZ_db)
```

Dưới đây là một số kết quả khi ta giữa thuật toán song song và thuật toán tuần tự.

```
%%time
main()

Kết quả truyền trực tuyến bị cắt bớt đến 5000 dòng cuối.
dE_dX:  [ 5.73654403e-05  4.28543768e-04 -5.03350315e-04 ...  1.69639616e-03
  1.75231881e-03  9.87174668e-04]
dE_dX parallel:  [ 1.82915049e-04  8.79044514e-05  3.42047211e-04 ... -2.90612445e-04
 -2.39911939e-04  3.84792701e-04]
dE_dX:  [ 1.82915049e-04  8.79044514e-05  3.42047211e-04 ... -2.90612445e-04
 -2.39911939e-04  3.84792701e-04]
dE_dX parallel:  [-4.22057113e-04 -3.96464918e-04  1.20831329e-05 ...  1.02545903e-03
  1.47572758e-03  1.09729244e-04]
dE_dX:  [-4.22057113e-04 -3.96464918e-04  1.20831329e-05 ...  1.02545903e-03
  1.47572758e-03  1.09729244e-04]
dE_dX parallel:  [ 2.05032479e-05  4.08317906e-05 -4.99328110e-05 ... -1.68902188e-05
  4.90862751e-05 -1.24115944e-05]
dE_dX:  [ 2.05032479e-05  4.08317906e-05 -4.99328110e-05 ... -1.68902188e-05
  4.90862751e-05 -1.24115944e-05]
dE_dX parallel:  [ 8.83317660e-06 -5.70150324e-06  2.23977004e-05 ...  2.19182733e-06
 -2.14676624e-05  3.65084926e-07]
dE_dX:  [ 8.83317660e-06 -5.70150324e-06  2.23977004e-05 ...  2.19182733e-06
 -2.14676624e-05  3.65084926e-07]
dE_dX parallel:  [-1.05538891e-07 -1.11360690e-07  5.54984757e-08 ...  2.22128026e-07
  2.86962105e-07  6.62254528e-09]
dE_dX:  [-1.05538891e-07 -1.11360690e-07  5.54984757e-08 ...  2.22128026e-07
  2.86962105e-07  6.62254528e-09]
```

Do đó, có thể kết luận thuật toán song song nhân ma trận với vector hoàn toàn đúng đắn.

- ***Hàm nhân hai ma trận với nhau:***
  Sử dụng hàm np.allclose() để kiểm tra tất cả các phần tử trong ma trận kết quả của thuật toán song song có bằng với ma trận kết quả của thuật toán tuần tự.

```
[23]    # Compute gradients of output Z with respect to weight, bias, input
        dZ_dw = self.flattened_input
        dZ_db = 1
        dZ_dX = self.weight

        # Gradient of loss with respect ot output
        dE_dZ = gradient * dY_dZ

        # Gradient of loss with respect to weight, bias, input
        C = np.empty((dZ_dw[np.newaxis].T.shape[0], dE_dZ[np.newaxis].shape[1]))
        dA = cuda.to_device(dZ_dw[np.newaxis].T)
        dB = cuda.to_device(dE_dZ[np.newaxis])
        dC = cuda.to_device(C)
        blockx = int(np.ceil(C.shape[0] / 16))
        blocky = int(np.ceil(C.shape[1] / 16))
        blockspergrid = (blockx, blocky)
        matmul[blockspergrid, (16,16)](dA,dB,dC)
        dE_dW = dC.copy_to_host()

        # Checking correctness
        np_result = dZ_dw[np.newaxis].T @ dE_dZ[np.newaxis]
        print("Is parallel result equal to sequential result? :", np.allclose(dE_dW, np_result))
```

Dưới đây là kết quả khi kiểm tra hai ma trận có bằng nhau hay không, do kết quả khá nhiều nên chỉ được trình bày một phần.

```
Is parallel result equal to sequential result? : True
Is parallel result equal to sequential result? : True
Is parallel result equal to sequential result? : True
Is parallel result equal to sequential result? : True
Step 1301. For the last 100 steps: average loss 0.4954375949022086, accuracy 87
Is parallel result equal to sequential result? : True
Is parallel result equal to sequential result? : True
Is parallel result equal to sequential result? : True
Is parallel result equal to sequential result? : True
Is parallel result equal to sequential result? : True
Is parallel result equal to sequential result? : True
```

Dựa vào kết quả kiểm tra, ta có thể kết luận tính đúng đắn của thuật toán khi song song là hoàn toàn đúng khi kết quả trùng khớp với kết quả của thuật toán tuần tự.

4.2.3/ Đo thời gian chạy so với phiên bản tuần tự:

4.2.3a/ Thời gian chạy của hàm song song so với hàm tuần tự:

- ***Hàm Dot Product giữa vector và ma trận:***

```
[42] Step 6801. For the last 100 steps: average loss 0.34133726498655553, accuracy 90
     Step 6901. For the last 100 steps: average loss 0.3114706253050864, accuracy 92
     Step 7001. For the last 100 steps: average loss 0.3095490927220985, accuracy 91
     Step 7101. For the last 100 steps: average loss 0.34867940371507034, accuracy 90
     Step 7201. For the last 100 steps: average loss 0.1586437198456132, accuracy 96
     Step 7301. For the last 100 steps: average loss 0.18549820355366164, accuracy 97
     Step 7401. For the last 100 steps: average loss 0.37372247921815904, accuracy 88
     Step 7501. For the last 100 steps: average loss 0.3302007857745093, accuracy 88
     Step 7601. For the last 100 steps: average loss 0.44998373969700495, accuracy 87
     Step 7701. For the last 100 steps: average loss 0.34775599732556495, accuracy 94
     Step 7801. For the last 100 steps: average loss 0.13989958951175657, accuracy 97
     Step 7901. For the last 100 steps: average loss 0.4038726168022598, accuracy 88
     Step 8001. For the last 100 steps: average loss 0.37443172656739326, accuracy 91
     Step 8101. For the last 100 steps: average loss 0.30532044314112416, accuracy 93
     Step 8201. For the last 100 steps: average loss 0.31210625699644035, accuracy 93
     Step 8301. For the last 100 steps: average loss 0.27384219300675444, accuracy 92
     Step 8401. For the last 100 steps: average loss 0.2618412726841744, accuracy 92
     Step 8501. For the last 100 steps: average loss 0.23729419599256574, accuracy 90
     Step 8601. For the last 100 steps: average loss 0.34080981151473777, accuracy 88
     Step 8701. For the last 100 steps: average loss 0.2811696807162289, accuracy 90
     Step 8801. For the last 100 steps: average loss 0.2663684023793971, accuracy 95
     Step 8901. For the last 100 steps: average loss 0.39408234120336344, accuracy 92
     Step 9001. For the last 100 steps: average loss 0.3683145849252073, accuracy 89
     Step 9101. For the last 100 steps: average loss 0.3818133244553232, accuracy 88
     Step 9201. For the last 100 steps: average loss 0.44723299624579654, accuracy 86
     Step 9301. For the last 100 steps: average loss 0.2887981989935755, accuracy 89
     Step 9401. For the last 100 steps: average loss 0.25943401680523087, accuracy 94
     Step 9501. For the last 100 steps: average loss 0.23237186927592723, accuracy 95
     Step 9601. For the last 100 steps: average loss 0.3378612803198382, accuracy 90
     Step 9701. For the last 100 steps: average loss 0.45333508603279127, accuracy 86
     Step 9801. For the last 100 steps: average loss 0.19267790318814007, accuracy 93
     Step 9901. For the last 100 steps: average loss 0.30465399900933343, accuracy 89
     CPU times: user 12min 8s, sys: 8.01 s, total: 12min 16s
     Wall time: 12min 16s
```

So với phiên bản tuần tự thì hàm song song nhân ma trận với vector sẽ giảm được thời gian chạy khoảng 7 phút.

- ***Hàm nhân ma trận với vector:***

```
     Step 7201. For the last 100 steps: average loss 0.42357617739958797, accuracy 88
     Step 7301. For the last 100 steps: average loss 0.33259091751883807, accuracy 89
     Step 7401. For the last 100 steps: average loss 0.49416984546133563, accuracy 86
     Step 7501. For the last 100 steps: average loss 0.7914401394044664, accuracy 81
     Step 7601. For the last 100 steps: average loss 0.12611518933239535, accuracy 98
     Step 7701. For the last 100 steps: average loss 0.3208726886780635, accuracy 90
     Step 7801. For the last 100 steps: average loss 0.3832891846034779, accuracy 89
     Step 7901. For the last 100 steps: average loss 0.4513910168717827, accuracy 85
     Step 8001. For the last 100 steps: average loss 0.4477879172048789, accuracy 89
     Step 8101. For the last 100 steps: average loss 0.37737809850383663, accuracy 89
     Step 8201. For the last 100 steps: average loss 0.3925308034598663, accuracy 87
     Step 8301. For the last 100 steps: average loss 0.37666190506634606, accuracy 87
     Step 8401. For the last 100 steps: average loss 0.3056741287689698, accuracy 91
     Step 8501. For the last 100 steps: average loss 0.604593329656226, accuracy 87
     Step 8601. For the last 100 steps: average loss 0.4688214376302015, accuracy 86
     Step 8701. For the last 100 steps: average loss 0.38319506493215577, accuracy 89
     Step 8801. For the last 100 steps: average loss 0.2690205995948458, accuracy 91
     Step 8901. For the last 100 steps: average loss 0.37501051946002845, accuracy 91
     Step 9001. For the last 100 steps: average loss 0.36710701181507716, accuracy 88
     Step 9101. For the last 100 steps: average loss 0.3134025043886621, accuracy 93
     Step 9201. For the last 100 steps: average loss 0.37187152396407314, accuracy 89
     Step 9301. For the last 100 steps: average loss 0.3326038005908487, accuracy 85
     Step 9401. For the last 100 steps: average loss 0.3003302998204991, accuracy 87
     Step 9501. For the last 100 steps: average loss 0.14306124762719427, accuracy 95
     Step 9601. For the last 100 steps: average loss 0.43749248375351096, accuracy 86
     Step 9701. For the last 100 steps: average loss 0.2791372623666859, accuracy 90
     Step 9801. For the last 100 steps: average loss 0.24560403291035166, accuracy 93
     Step 9901. For the last 100 steps: average loss 0.3248755684976662, accuracy 92
     CPU times: user 18min 29s, sys: 13min 4s, total: 31min 33s
     Wall time: 17min 59s
```

So với phiên bản tuần tự thì hàm song song nhân ma trận với vector sẽ tối ưu được thời gian chạy khoảng gần 2 phút.

- ***Hàm nhân hai ma trận với nhau:***

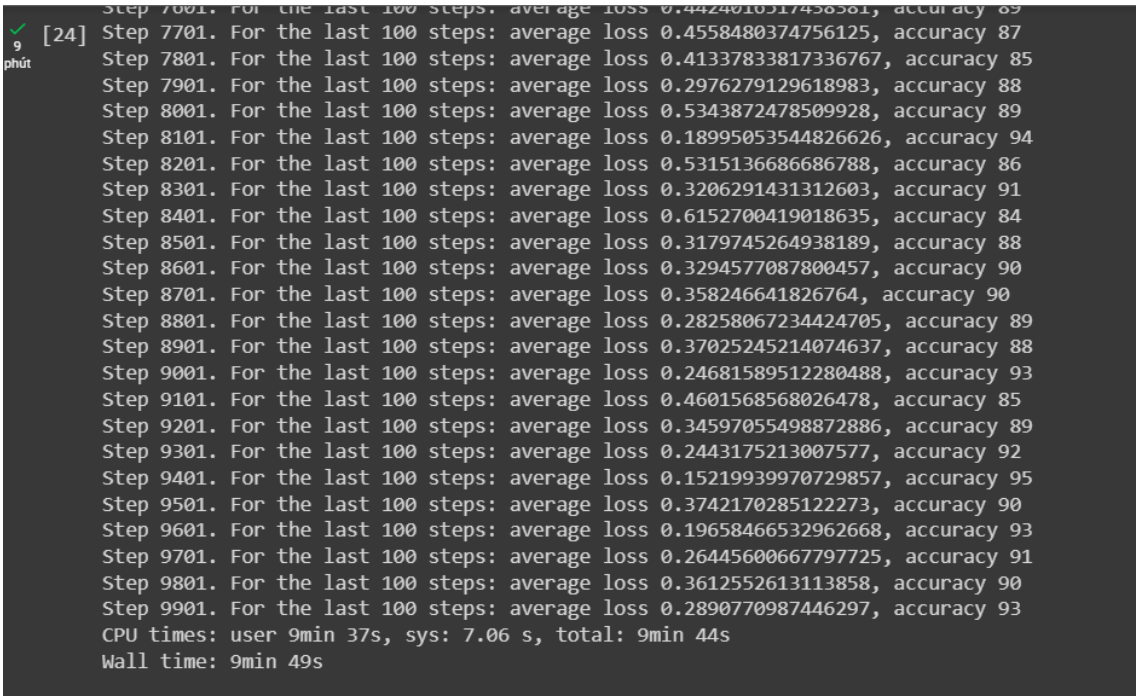
  Ta sẽ thay thế phép nhân hai ma trận thông thường bằng phương pháp song song ở trong phiên bản tuần tự và đo thời gian chạy.

```
            continue
        transformation_eq = np.exp(self.output)
        S_total = np.sum(transformation_eq)

        # Compute gradients with respect to output (Z)
        dY_dZ = -transformation_eq[i]*transformation_eq / (S_total**2)
        dY_dZ[i] = transformation_eq[i]*(S_total - transformation_eq[i]) / (S_total**2)

        # Compute gradients of output Z with respect to weight, bias, input
        dZ_dw = self.flattened_input
        dZ_db = 1
        dZ_dX = self.weight

        # Gradient of loss with respect ot output
        dE_dZ = gradient * dY_dZ

        # Gradient of loss with respect to weight, bias, input

        C = np.empty((dZ_dw[np.newaxis].T.shape[0], dE_dZ[np.newaxis].shape[1]))
        dA = cuda.to_device(dZ_dw[np.newaxis].T)
        dB = cuda.to_device(dE_dZ[np.newaxis])
        dC = cuda.to_device(C)
        blockx = int(np.ceil(C.shape[0] / 16))
        blocky = int(np.ceil(C.shape[1] / 16))
        blockspergrid = (blockx, blocky)
        matmul[blockspergrid, (16,16)](dA,dB,dC)
        dE_dw = dC.copy_to_host()

        #dE_dw = dZ_dw[np.newaxis].T @ dE_dZ[np.newaxis]
        dE_db = dE_dZ * dZ_db
        dE_dX = dZ_dX @ dE_dZ

        # Update parameters
        self.weight -= alpha*dE_dw
        self.bias -= alpha*dE_db

        return dE_dX.reshape(self.original_shape)
```

```
+ Mã  + Văn bản
16 phút
Step 7001. For the last 100 steps: average loss 0.4269263529400053, accuracy 86
Step 7101. For the last 100 steps: average loss 0.30687569559995587, accuracy 92
Step 7201. For the last 100 steps: average loss 0.28468985286021636, accuracy 88
Step 7301. For the last 100 steps: average loss 0.37426643258853703, accuracy 91
Step 7401. For the last 100 steps: average loss 0.42380212751830015, accuracy 85
Step 7501. For the last 100 steps: average loss 0.4830761054115144, accuracy 87
Step 7601. For the last 100 steps: average loss 0.17148154002339352, accuracy 94
Step 7701. For the last 100 steps: average loss 0.28512200452320124, accuracy 91
Step 7801. For the last 100 steps: average loss 0.3178547004199781, accuracy 91
Step 7901. For the last 100 steps: average loss 0.39312142174199727, accuracy 92
Step 8001. For the last 100 steps: average loss 0.3269254238684755, accuracy 90
Step 8101. For the last 100 steps: average loss 0.36901348400532474, accuracy 86
Step 8201. For the last 100 steps: average loss 0.18680973530632194, accuracy 94
Step 8301. For the last 100 steps: average loss 0.16132262653640478, accuracy 92
Step 8401. For the last 100 steps: average loss 0.37260130817890497, accuracy 91
Step 8501. For the last 100 steps: average loss 0.30438634343185406, accuracy 89
Step 8601. For the last 100 steps: average loss 0.26406272273069137, accuracy 88
Step 8701. For the last 100 steps: average loss 0.30688824845109924, accuracy 91
Step 8801. For the last 100 steps: average loss 0.4670109412431248, accuracy 91
Step 8901. For the last 100 steps: average loss 0.2859089875771763, accuracy 88
Step 9001. For the last 100 steps: average loss 0.46867478079561814, accuracy 86
Step 9101. For the last 100 steps: average loss 0.33866664008217984, accuracy 88
Step 9201. For the last 100 steps: average loss 0.2860156681596204, accuracy 93
Step 9301. For the last 100 steps: average loss 0.22764784407432068, accuracy 90
Step 9401. For the last 100 steps: average loss 0.2043702190806686, accuracy 95
Step 9501. For the last 100 steps: average loss 0.38551260619021055, accuracy 85
Step 9601. For the last 100 steps: average loss 0.22008779333495415, accuracy 90
Step 9701. For the last 100 steps: average loss 0.4051198799558361, accuracy 86
Step 9801. For the last 100 steps: average loss 0.37218676751189705, accuracy 84
Step 9901. For the last 100 steps: average loss 0.19097095441729017, accuracy 94
CPU times: user 16min 16s, sys: 11min 38s, total: 27min 55s
Wall time: 16min 7s
```

Kết quả chạy hàm song song cho phép nhân hai ma trận có vẻ là khá tốt khi ta tối ưu được đến khoảng 3 phút.

4.2.3b/ Thời gian chạy của lớp Softmax trước và sau khi sử dụng hàm song song:



Sau khi kết hợp cả 3 hàm song song vào trong lớp Softmax, ta giảm thời gian chạy của toàn bộ thuật toán CNN xuống còn 9 phút 49 giây, nhanh hơn 10 phút so với thuật toán ban đầu.

4.3/ Lớp Max-Pooling:

- Ở lớp max pooling, do tính chất của lớp này chỉ có update lại các trọng số từ lớp trước và không có tính toán gì về matrix nên gần như việc song song cũng không làm giảm thời gian chạy của thuật toán và không hiệu quả như chạy tuần tự. Ngoài ra ta cũng tính toán được lớp max pooling chiếm thời gian chạy thấp nhất so với 2 lớp còn lại.

```
Epoch 1 ->
Step 1. For the last 100 steps: average loss 0.0, accuracy 0
forward  <__main__.ConvolutionLayer object at 0x7ad3fa6a0ca0>
Processing time: 0.012261152267456055 s
forward  <__main__.MaxPoolingLayer object at 0x7ad3fa6a1e70>
Processing time: 0.004868268966674805 s
forward  <__main__.SoftmaxLayer object at 0x7ad3fa6a35b0>
Processing time: 0.01004338264465332 s


backprob  <__main__.SoftmaxLayer object at 0x7ad3fa6a35b0>
Processing time: 0.0007891654968261719 s
backprob  <__main__.MaxPoolingLayer object at 0x7ad3fa6a1e70>
Processing time: 0.00013947486877441406 s
backprob  <__main__.ConvolutionLayer object at 0x7ad3fa6a0ca0>
Processing time: 0.04870915412902832 s
complete one image

forward  <__main__.ConvolutionLayer object at 0x7ad3fa6a0ca0>
Processing time: 0.01158761978149414 s
forward  <__main__.MaxPoolingLayer object at 0x7ad3fa6a1e70>
Processing time: 0.0017118453979492188 s
forward  <__main__.SoftmaxLayer object at 0x7ad3fa6a35b0>
Processing time: 0.009320974349975586 s


backprob  <__main__.SoftmaxLayer object at 0x7ad3fa6a35b0>
Processing time: 0.0002713203430175781 s
backprob  <__main__.MaxPoolingLayer object at 0x7ad3fa6a1e70>
Processing time: 0.00014829635620117188 s
backprob  <__main__.ConvolutionLayer object at 0x7ad3fa6a0ca0>
Processing time: 0.07295560836791992 s
complete one image
```

- Như hình trên ta có thể thấy là Hai hàm forward và backward của max pooling layer luôn cho ra thời gian nhanh hơn so với hai lớp còn lại, theo sau là lớp softmax và cuối cùng lớp chiếm thời gian chạy nhiều nhất là convolutional với thời gian chạy ở cả hai function đều từ 0.01 trở lên đồng thời hàm backward là hàm chiếm thời gian nhiều nhất trong lớp này.

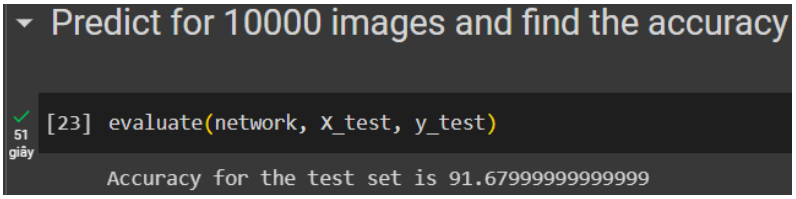
- Từ hình trên ta có thể kết luận lớp convolutional là lớp trọng điểm có thể dùng để song song hóa và tối ưu thời gian chạy trong thuật toán CNN.

4.4/ Độ chính xác của mạng CNN:

- **<u>Độ chính xác:</u>**
    - Cài đặt các hàm predict() để dự đoán cho 10000 ảnh trong tập dữ liệu và hàm evaluate() để đánh giá độ chính xác cho mạng CNN được song song.

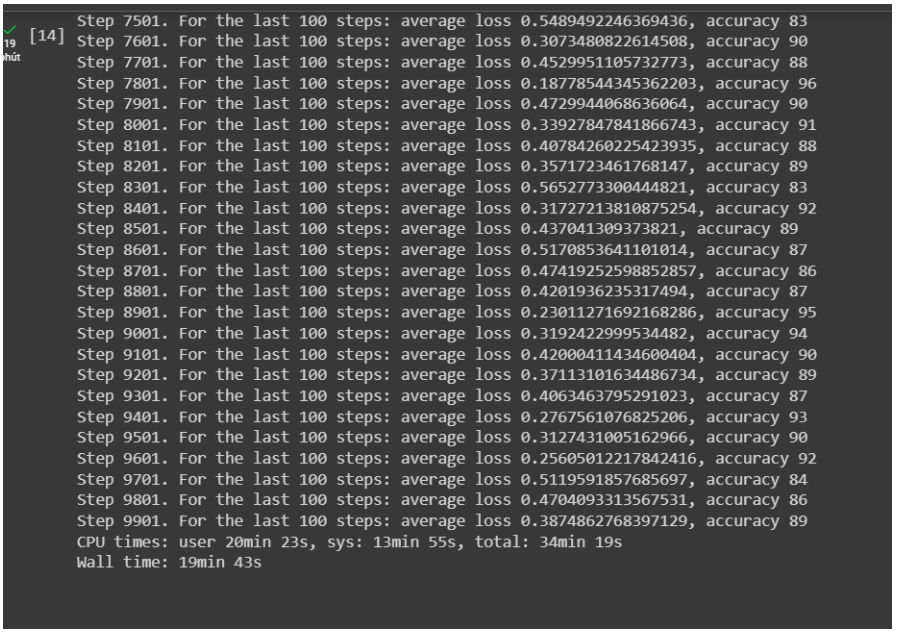
```python
def predict(network, image):
    output = image/255.
    for layer in network:
        output = layer.forward_prop(output)
    return np.argmax(output) # return a number
def evaluate(network, X_test, y_test):
    correct = 0
    for x, y in zip(X_test, y_test):
        pred = predict(network, x)
        if y == pred:
            correct += 1
    acc = correct / y_test.shape[0]
    print(f'Accuracy for the test set is {acc *100}')
```

- Chạy thử nghiệm và ta thu được kết quả khá tốt khi ***accuracy là gần 92%.***



4.5/ So sánh thời gian chạy của các phiên bản:

- ***Phiên bản tuần tự:***
  - Thời gian chạy: ***19 phút 43 giây***



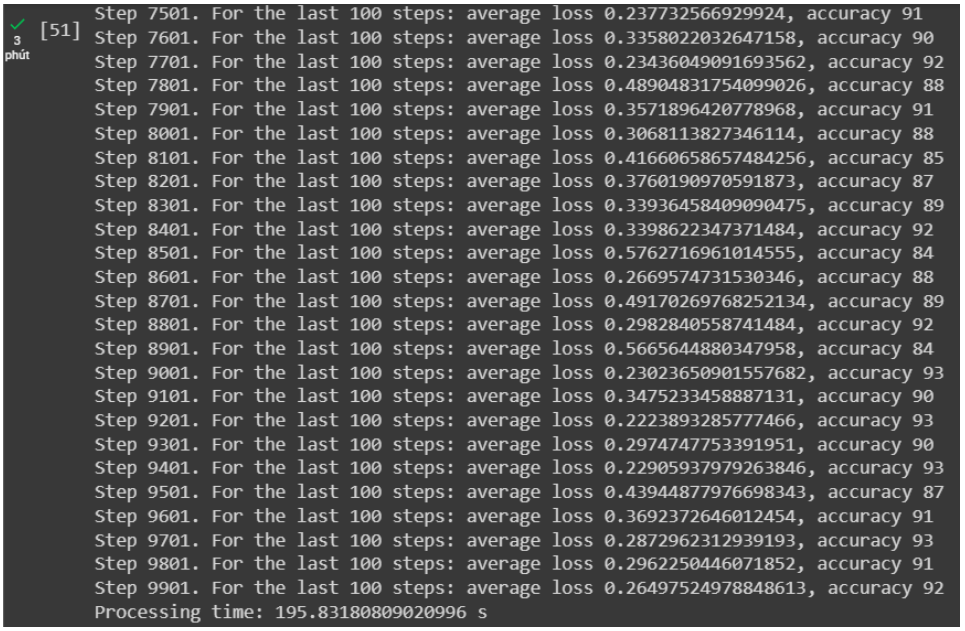
Thời gian chạy của phiên bản tuần tự

- ***Phiên bản 1:***
    - Lớp Convolution: Sử dụng hàm song song cho forward_prop() và back_prop().
    - Lớp Softmax: Sử dụng 2 hàm song song là nhân ma trận với vector và nhân vector với ma trận.
    - Thời gian chạy: ***1 phút 51 giây***

```
Step 7501. For the last 100 steps: average loss 0.39681454523861587, accuracy 88
Step 7601. For the last 100 steps: average loss 0.26814663092869373, accuracy 94
Step 7701. For the last 100 steps: average loss 0.20720597294670232, accuracy 98
Step 7801. For the last 100 steps: average loss 0.35185205592342345, accuracy 91
Step 7901. For the last 100 steps: average loss 0.2838049292758561, accuracy 92
Step 8001. For the last 100 steps: average loss 0.1700826421957802, accuracy 94
Step 8101. For the last 100 steps: average loss 0.35810547204103593, accuracy 90
Step 8201. For the last 100 steps: average loss 0.3650215637598439, accuracy 91
Step 8301. For the last 100 steps: average loss 0.2816223955980213, accuracy 91
Step 8401. For the last 100 steps: average loss 0.26348909294782713, accuracy 92
Step 8501. For the last 100 steps: average loss 0.19421031706580938, accuracy 96
Step 8601. For the last 100 steps: average loss 0.3054666976641127, accuracy 92
Step 8701. For the last 100 steps: average loss 0.26124304193159564, accuracy 92
Step 8801. For the last 100 steps: average loss 0.16403678715632128, accuracy 94
Step 8901. For the last 100 steps: average loss 0.35331597843619783, accuracy 93
Step 9001. For the last 100 steps: average loss 0.27159438528351876, accuracy 91
Step 9101. For the last 100 steps: average loss 0.262875324330881, accuracy 94
Step 9201. For the last 100 steps: average loss 0.18679469844538021, accuracy 95
Step 9301. For the last 100 steps: average loss 0.3165981349430608, accuracy 92
Step 9401. For the last 100 steps: average loss 0.1658358303079629, accuracy 96
Step 9501. For the last 100 steps: average loss 0.38446389934803776, accuracy 88
Step 9601. For the last 100 steps: average loss 0.25845057629249985, accuracy 92
Step 9701. For the last 100 steps: average loss 0.34775867138297434, accuracy 92
Step 9801. For the last 100 steps: average loss 0.3454054707254099, accuracy 90
Step 9901. For the last 100 steps: average loss 0.21349389141487218, accuracy 94
Processing time: 111.18359184265137 s
```

Thời gian chạy của phiên bản 1 khi kết hợp tất cả các hàm song song

- ***Phiên bản 2:***
    - Lớp Convolution: Sử dụng hàm song song cho forward_prop() và back_prop().
    - Lớp Softmax: Sử dụng 3 hàm song song, trong đó có 2 hàm cũ ở phiên bản 1 là nhân ma trận với vector và nhân vector với ma trận. Cài đặt thêm hàm nhân hai ma trận chạy song song để cải thiện tốc độ cho lớp softmax.
    - Chỉnh sửa số lượng blocks per grid và threads per block cho hàm ma trận nhân vector.
    - Thời gian chạy: ***3 phút 16 giây***

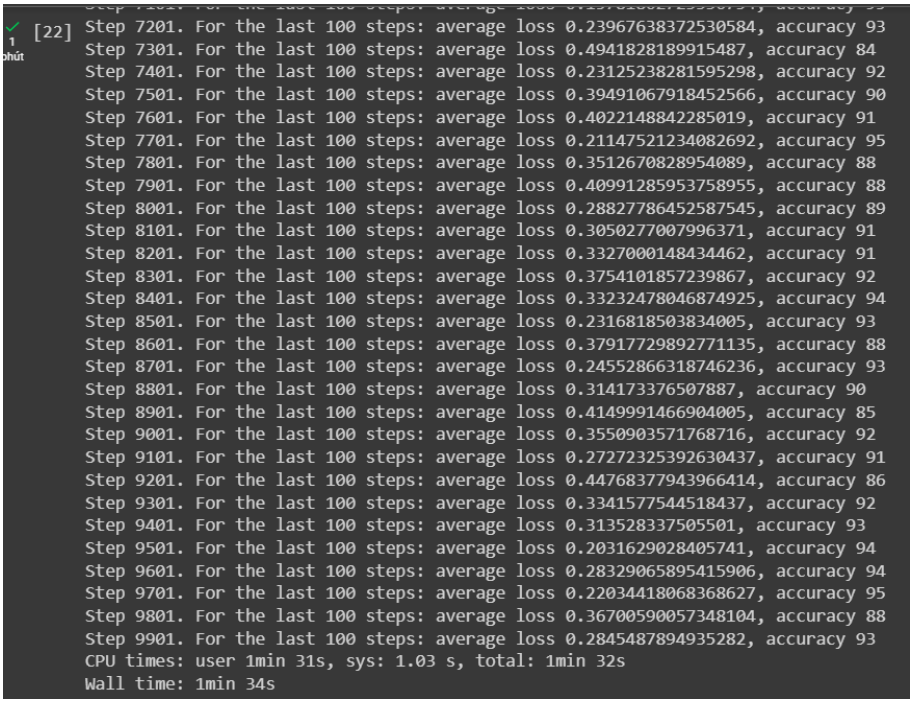


Thời gian chạy của phiên bản 2 khi kết hợp tất cả các hàm song song

- ***Phiên bản cuối cùng:***
  - Lớp Convolution: Sử dụng hàm song song cho forward_prop() và back_prop().
  - Lớp Softmax: Sử dụng 2 hàm song song ở phiên bản 1 là nhân ma trận với vector và nhân vector với ma trận. Đồng thời, chỉnh sửa số lượng blocks per grid và threads per block cho hàm ma trận nhân vector.

```
# Matrix-vector multiply function
C = np.empty(dZ_dX.shape[0])
dA = cuda.to_device(dZ_dX)
dB = cuda.to_device(dE_dZ)
dC = cuda.to_device(C)
cu_matrix_vector[(dZ_dX.shape[0]+15)//16, 16](dA,dB,dC)
dE_dX = dC.copy_to_host()
```

  - Thời gian chạy: ***1 phút 34 giây***



Phiên bản cuối cùng

Kết quả tốt nhất của thuật toán song song mà ta thu được có thời gian chạy chỉ là 1 phút 34 giây so với thuật toán tuần tự là 19 phút 43 giây.

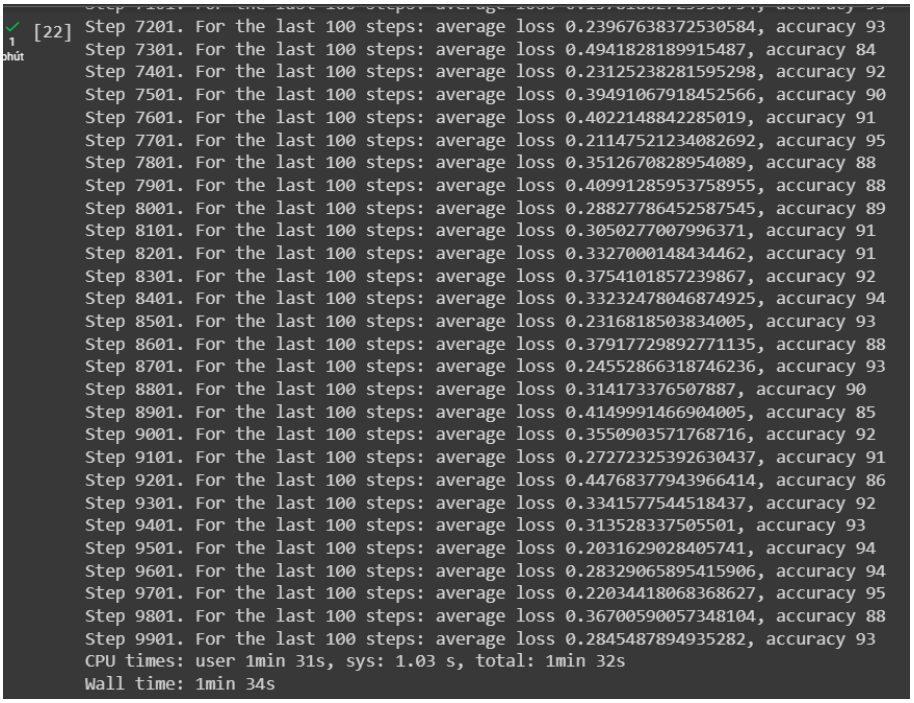
5/ Kết luận phiên bản tốt nhất:

- ***Bảng thống kê thời gian chạy của các phiên bản:***

Phiên bản	**Phiên bản tuần tự**	**Phiên bản 1**	**Phiên bản 2**	**Phiên bản cuối cùng**
Thời gian chạy	19 phút 43 giây	1 phút 51 giây	3 phút 16 giây	***1 phút 34 giây***

- ***Phiên bản tốt nhất:***
  - Lớp Convolution: Sử dụng hàm song song cho forward_prop() và back_prop().
  - Lớp Softmax: Sử dụng 2 hàm song song ở phiên bản 1 là nhân ma trận với vector và nhân vector với ma trận. Đồng thời, chỉnh sửa số lượng blocks per grid và threads per block cho hàm ma trận nhân vector.

  - Thời gian chạy: ***1 phút 34 giây***



Phiên bản cuối cùng

- ***Kết luận:*** Kết quả tốt nhất của thuật toán song song mà ta thu được có thời gian chạy chỉ là **1 phút 34 giây** so với thuật toán tuần tự là **19 phút 43 giây** (nhanh hơn gấp 12 lần so với phiên bản tuần tự).