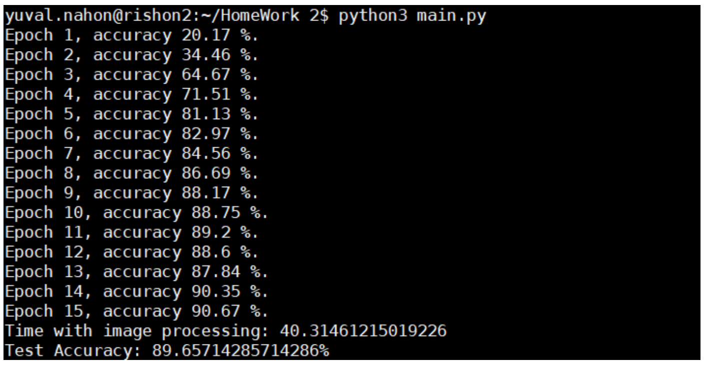
Parallel and distributed programing – HW 2 Report

Submitters:
Yuval Nahon – 206866832
Sahar Cohen - 206824088

Part 1:

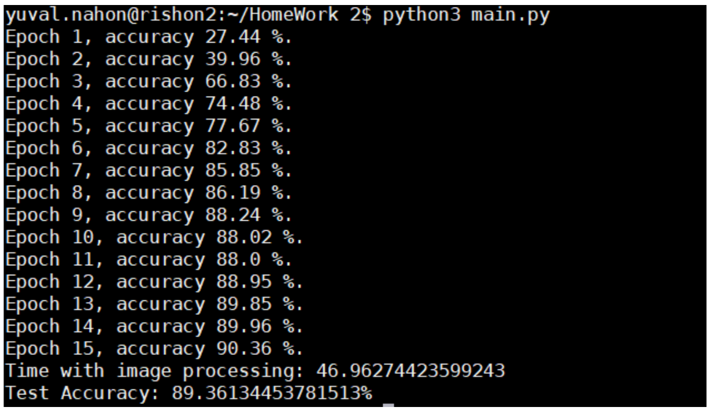
2)

32 cores:



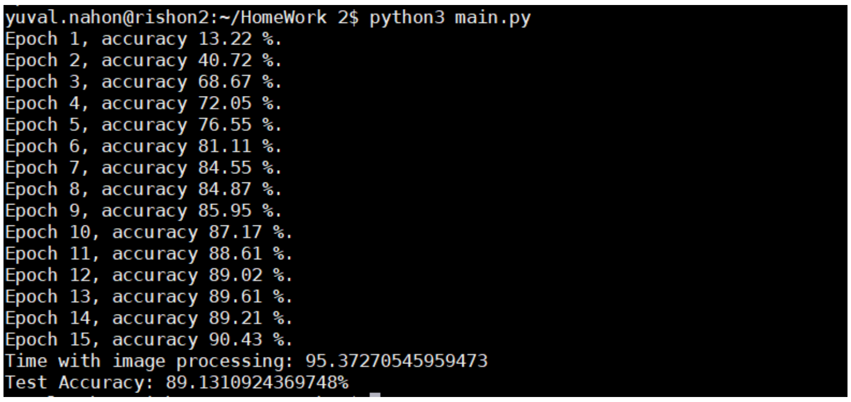
```
yuval.nahon@rishon2:~/HomeWork 2$ python3 main.py
Epoch 1, accuracy 20.17 %.
Epoch 2, accuracy 34.46 %.
Epoch 3, accuracy 64.67 %.
Epoch 4, accuracy 71.51 %.
Epoch 5, accuracy 81.13 %.
Epoch 6, accuracy 82.97 %.
Epoch 7, accuracy 84.56 %.
Epoch 8, accuracy 86.69 %.
Epoch 9, accuracy 88.17 %.
Epoch 10, accuracy 88.75 %.
Epoch 11, accuracy 89.2 %.
Epoch 12, accuracy 88.6 %.
Epoch 13, accuracy 87.84 %.
Epoch 14, accuracy 90.35 %.
Epoch 15, accuracy 90.67 %.
Time with image processing: 40.31461215019226
Test Accuracy: 89.65714285714286%
```

16 cores:



```
yuval.nahon@rishon2:~/HomeWork 2$ python3 main.py
Epoch 1, accuracy 27.44 %.
Epoch 2, accuracy 39.96 %.
Epoch 3, accuracy 66.83 %.
Epoch 4, accuracy 74.48 %.
Epoch 5, accuracy 77.67 %.
Epoch 6, accuracy 82.83 %.
Epoch 7, accuracy 85.85 %.
Epoch 8, accuracy 86.19 %.
Epoch 9, accuracy 88.24 %.
Epoch 10, accuracy 88.02 %.
Epoch 11, accuracy 88.0 %.
Epoch 12, accuracy 88.95 %.
Epoch 13, accuracy 89.85 %.
Epoch 14, accuracy 89.96 %.
Epoch 15, accuracy 90.36 %.
Time with image processing: 46.96274423599243
Test Accuracy: 89.36134453781513%
```

8 cores:



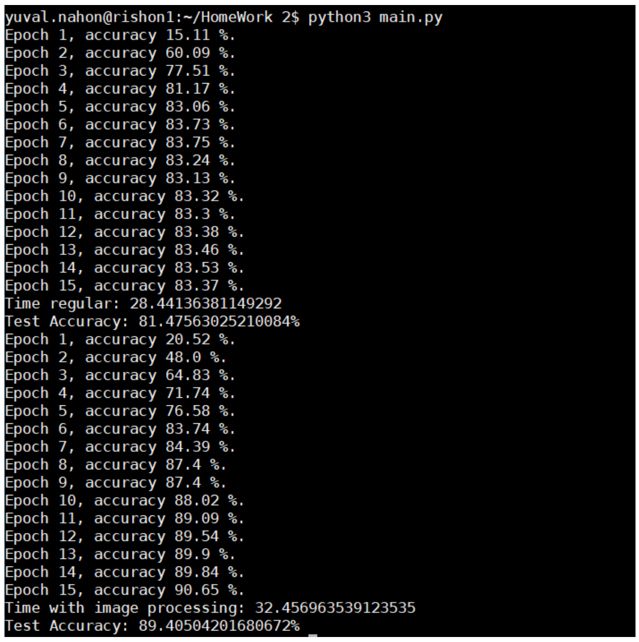
```
yuval.nahon@rishon2:~/HomeWork 2$ python3 main.py
Epoch 1, accuracy 13.22 %.
Epoch 2, accuracy 40.72 %.
Epoch 3, accuracy 68.67 %.
Epoch 4, accuracy 72.05 %.
Epoch 5, accuracy 76.55 %.
Epoch 6, accuracy 81.11 %.
Epoch 7, accuracy 84.55 %.
Epoch 8, accuracy 84.87 %.
Epoch 9, accuracy 85.95 %.
Epoch 10, accuracy 87.17 %.
Epoch 11, accuracy 88.61 %.
Epoch 12, accuracy 89.02 %.
Epoch 13, accuracy 89.61 %.
Epoch 14, accuracy 89.21 %.
Epoch 15, accuracy 90.43 %.
Time with image processing: 95.37270545959473
Test Accuracy: 89.1310924369748%
```

The best performance was given by running with 32 cores because more workers could work simultaneously.

3)

	NeuralNetwork	**IPNeuralNetwork**
Epoch 1	15.11%	20.52%
Epoch 2	60.09%	48.0%
Epoch 3	77.51%	64.83%
Epoch 4	81.17%	71.74%
Epoch 5	83.06%	76.58%
Epoch 6	83.73%	83.74%
Epoch 7	83.75%	84.39%
Epoch 8	83.24%	87.4%
Epoch 9	83.13%	87.4%
Epoch 10	83.32%	88.02%
Epoch 11	83.3%	89.09%
Epoch 12	83.38%	89.54%
Epoch 13	83.46%	89.9%
Epoch 14	83.53%	89.84%
Epoch 15	83.37%	90.65%
Time	28.44136	32.45696
Accuarcy	81.475630%	89.405042%

The difference between the neural networks is that the IPNeuralNetwork makes some modifications to the images by allowing some shift, rotate, step and skew functions to be applied to the them (the functions modify the images in such way where they are still labeled the same). Therefore, IPNeuralNetwork is learning the true nature of the way digits are handwritten instead of learning from fixed examples.



```
yuval.nahon@rishon1:~/HomeWork 2$ python3 main.py
Epoch 1, accuracy 15.11 %.
Epoch 2, accuracy 60.09 %.
Epoch 3, accuracy 77.51 %.
Epoch 4, accuracy 81.17 %.
Epoch 5, accuracy 83.06 %.
Epoch 6, accuracy 83.73 %.
Epoch 7, accuracy 83.75 %.
Epoch 8, accuracy 83.24 %.
Epoch 9, accuracy 83.13 %.
Epoch 10, accuracy 83.32 %.
Epoch 11, accuracy 83.3 %.
Epoch 12, accuracy 83.38 %.
Epoch 13, accuracy 83.46 %.
Epoch 14, accuracy 83.53 %.
Epoch 15, accuracy 83.37 %.
Time regular: 28.44136381149292
Test Accuracy: 81.47563025210084%
Epoch 1, accuracy 20.52 %.
Epoch 2, accuracy 48.0 %.
Epoch 3, accuracy 64.83 %.
Epoch 4, accuracy 71.74 %.
Epoch 5, accuracy 76.58 %.
Epoch 6, accuracy 83.74 %.
Epoch 7, accuracy 84.39 %.
Epoch 8, accuracy 87.4 %.
Epoch 9, accuracy 87.4 %.
Epoch 10, accuracy 88.02 %.
Epoch 11, accuracy 89.09 %.
Epoch 12, accuracy 89.54 %.
Epoch 13, accuracy 89.9 %.
Epoch 14, accuracy 89.84 %.
Epoch 15, accuracy 90.65 %.
Time with image processing: 32.456963539123535
Test Accuracy: 89.40504201680672% _
```

4) Threads in python can't really run simultaneously and processes can.

Since we want to make the workers work at the same time, we used processes.

Part 2:

5) We used a pipe as the data structure for the class. We used Pipe because it's process-safe.

Functions explanation:

__init__ (self): Creates a lock for the queue and saves it in the self.lock field.

Creates a Pipe and saves it in the self.pusher and self.reader fields respectively.

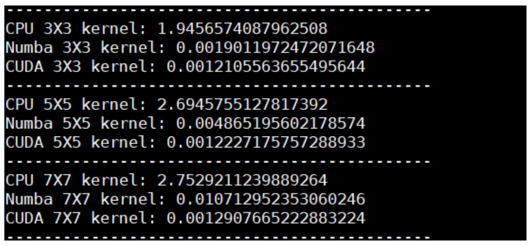
put (self, msg): pushes a new element to the pipe through the self.pusher field.

get (self): locks with self.lock and reads the top element from self.reader.

We lock the queue in the get function because only one reader can exist at any time. We don't lock the queue in the put function because there is no limit on the number of writers to the queue (a lot of messages can be written to the queue concurrently).

6) **Convolution numba**: creates a flipped kernel. Creates a padded version of the image (with 0's) so calculating the result's entries at the edges of the image can be done. Afterwards, we iterate over every index in the result matrix and calculate its value by cropping only the relevant part of the image, multiplying it element-wise with the flipped kernel and summing the results.

Covolution gpu: implemented with the exact same logic as numba, only this time we created image_width * image_height threads and assigned an index for each thread to calculate in the result matrix.



```
---------------------------------------------
CPU 3X3 kernel: 1.9456574087962508
Numba 3X3 kernel: 0.0019011972472071648
CUDA 3X3 kernel: 0.0012105563655495644
---------------------------------------------
CPU 5X5 kernel: 2.6945755127817392
Numba 5X5 kernel: 0.004865195602178574
CUDA 5X5 kernel: 0.0012227175757288933
---------------------------------------------
CPU 7X7 kernel: 2.7529211239889264
Numba 7X7 kernel: 0.010712952353060246
CUDA 7X7 kernel: 0.0012907665222883224
---------------------------------------------
```

9) If we use a larger kernel:

The gpu's speedup will get lower because each thread will get a larger workload. Since each gpu thread by its own has an inferior computational power, a larger kernel will slow it down a lot.

Numba's speedup will get lower **significantly** faster compared to the gpu's speedup decrease because the number of cores is very limited while the number of cuda threads is in a magnitude of tens of thousands (usually). Eventually, the inner loop of the function will be a substantial computational process that will diminish numba's outer loop parallelism.

In both cases, increasing the kernel result in a decrease of speedup. That is to be expected, because both functions rely on the image's dimensions to create workers for paralyzing the task - not the kernel's.