

תכנות מקבילי ומבוזר- תרגיל בית 2

איתי לביא 212147326 הראל ימין 213099492

Part 1

1. I ran it.
2. In the fit function we chose the number of processes to be the lowest between the number of batches and the core count, when we ran the simulation the number of batches was higher then 32 so we used more processes the more cores we got so the higher

```
(tf23-gpu) harel@lambda:~/MC2$ srun --gres=gpu:1 -c 8 --pty python3 ./main.py
Epoch 1, accuracy 26.5 %.
Epoch 2, accuracy 61.63 %.
Epoch 3, accuracy 72.97 %.
Epoch 4, accuracy 78.82 %.
Epoch 5, accuracy 81.89 %.
Epoch 6, accuracy 82.69 %.
Epoch 7, accuracy 83.1 %.
Epoch 8, accuracy 83.49 %.
Epoch 9, accuracy 83.6 %.
Epoch 10, accuracy 83.59 %.
Epoch 11, accuracy 83.66 %.
Epoch 12, accuracy 83.64 %.
Epoch 13, accuracy 83.66 %.
Epoch 14, accuracy 83.5 %.
Epoch 15, accuracy 83.49 %.
Time regular: 9.014626264572144
Test Accuracy: 81.58319327731093%
Epoch 1, accuracy 10.68 %.
Epoch 2, accuracy 40.35 %.
Epoch 3, accuracy 63.55 %.
Epoch 4, accuracy 69.69 %.
Epoch 5, accuracy 72.95 %.
Epoch 6, accuracy 77.42 %.
Epoch 7, accuracy 78.53 %.
Epoch 8, accuracy 82.04 %.
Epoch 9, accuracy 83.61 %.
Epoch 10, accuracy 84.86 %.
Epoch 11, accuracy 84.27 %.
Epoch 12, accuracy 84.43 %.
Epoch 13, accuracy 85.98 %.
Epoch 14, accuracy 86.76 %.
Epoch 15, accuracy 87.51 %.
Time with image processing: 19.217571258544922
Test Accuracy: 85.89336134453781%

(tf23-gpu) harel@lambda:~/MC2$ srun --gres=gpu:1 -c 16 --pty python3 ./main.py
Epoch 1, accuracy 9.61 %.
Epoch 2, accuracy 53.77 %.
Epoch 3, accuracy 74.45 %.
Epoch 4, accuracy 81.28 %.
Epoch 5, accuracy 83.82 %.
Epoch 6, accuracy 85.56 %.
Epoch 7, accuracy 85.52 %.
Epoch 8, accuracy 85.47 %.
Epoch 9, accuracy 85.55 %.
Epoch 10, accuracy 85.45 %.
Epoch 11, accuracy 85.5 %.
Epoch 12, accuracy 85.82 %.
Epoch 13, accuracy 85.93 %.
Epoch 14, accuracy 85.85 %.
Epoch 15, accuracy 85.93 %.
Time regular: 18.612130888355835
Test Accuracy: 81.87226898756383%
Epoch 1, accuracy 10.64 %.
Epoch 2, accuracy 31.3 %.
Epoch 3, accuracy 56.55 %.
Epoch 4, accuracy 69.9 %.
Epoch 5, accuracy 71.36 %.
Epoch 6, accuracy 75.15 %.
Epoch 7, accuracy 79.63 %.
Epoch 8, accuracy 80.05 %.
Epoch 9, accuracy 84.23 %.
Epoch 10, accuracy 84.92 %.
Epoch 11, accuracy 84.1 %.
Epoch 12, accuracy 86.51 %.
Epoch 13, accuracy 84.72 %.
Epoch 14, accuracy 87.17 %.
Epoch 15, accuracy 88.45 %.
Time with image processing: 18.487579584121704
Test Accuracy: 86.52941176470588%

(tf23-gpu) harel@lambda:~/MC2$ srun --gres=gpu:1 -c 32 --pty python3 ./main.py
Epoch 1, accuracy 25.53 %.
Epoch 2, accuracy 61.15 %.
Epoch 3, accuracy 75.98 %.
Epoch 4, accuracy 81.95 %.
Epoch 5, accuracy 82.11 %.
Epoch 6, accuracy 82.71 %.
Epoch 7, accuracy 83.17 %.
Epoch 8, accuracy 83.16 %.
Epoch 9, accuracy 83.9 %.
Epoch 10, accuracy 83.18 %.
Epoch 11, accuracy 83.14 %.
Epoch 12, accuracy 82.99 %.
Epoch 13, accuracy 82.95 %.
Epoch 14, accuracy 83.05 %.
Epoch 15, accuracy 83.01 %.
Time regular: 18.616498231887817
Test Accuracy: 81.1579831932773%
Epoch 1, accuracy 28.4 %.
Epoch 2, accuracy 27.53 %.
Epoch 3, accuracy 58.23 %.
Epoch 4, accuracy 65.57 %.
Epoch 5, accuracy 72.1 %.
Epoch 6, accuracy 76.03 %.
Epoch 7, accuracy 80.48 %.
Epoch 8, accuracy 82.99 %.
Epoch 9, accuracy 82.72 %.
Epoch 10, accuracy 85.99 %.
Epoch 11, accuracy 86.72 %.
Epoch 12, accuracy 86.83 %.
Epoch 13, accuracy 86.5 %.
Epoch 14, accuracy 86.01 %.
Epoch 15, accuracy 87.19 %.
Time with image processing: 17.341471672058185
Test Accuracy: 85.630551688834%
```

core count the

lower the runtime, and the results are as expected. the accuracy varies because the randomness in our preprocessor functions. But it is still better than the default.

3. We ran the main again with 8 cores(it shouldn't matter for the accuracy) and got the results below. As we can see the normal NN accuracy improved fast at the start but at around the 5th epoch it stopped improving significantly. Probably a result of the data not being diverse. On the contrary, our NN gave different data every time, because of the randomness of the workers augmentation. So the accuracy grows slower but kept growing till the end.

NeuralNetwork	IPNeuralNetwork	Epoch
26.5	10.68	1
61.63	40.35	2
72.97	63.55	3
78.82	69.69	4

תכנות מקבילי ומבוזר- תרגיל בית 2

איתי לביא 212147326 הראל ימין 213099492

81.89	72.95	5
82.69	77.42	6
83.1	78.53	7
83.49	82.04	8
83.6	83.61	9
83.59	84.86	10
83.66	84.27	11
83.64	84.43	12
83.66	85.98	13
83.5	86.76	14
83.49	87.51	15

```
tr23@gpu) harely@lambda:~/MC23$ run --gres=gpu:1 -c 8 -pty python3 ./main.py
Epoch 1, accuracy 26.5 %
Epoch 2, accuracy 61.63 %
Epoch 3, accuracy 72.97 %
Epoch 4, accuracy 78.82 %
Epoch 5, accuracy 81.89 %
Epoch 6, accuracy 82.69 %
Epoch 7, accuracy 83.1 %
Epoch 8, accuracy 83.49 %
Epoch 9, accuracy 83.6 %
Epoch 10, accuracy 83.59 %
Epoch 11, accuracy 83.66 %
Epoch 12, accuracy 83.64 %
Epoch 13, accuracy 83.66 %
Epoch 14, accuracy 83.5 %
Epoch 15, accuracy 83.49 %
Time regular: 9.034625264572144
Test Accuracy: 81.58319327731893%
Epoch 1, accuracy 10.68 %
Epoch 2, accuracy 40.35 %
Epoch 3, accuracy 63.55 %
Epoch 4, accuracy 69.69 %
Epoch 5, accuracy 72.95 %
Epoch 6, accuracy 77.42 %
Epoch 7, accuracy 78.53 %
Epoch 8, accuracy 82.04 %
Epoch 9, accuracy 83.61 %
Epoch 10, accuracy 84.86 %
Epoch 11, accuracy 84.27 %
Epoch 12, accuracy 84.43 %
Epoch 13, accuracy 85.98 %
Epoch 14, accuracy 86.76 %
Epoch 15, accuracy 87.51 %
Time with image processing: 19.217571258544922
Test Accuracy: 86.80336134453781%
```

4. We are using processes and not threads because as we learned in the tutorials, python can't run threads simultaneously. And that won't help us get the multiprocessing result we want.
5. One idea is to use the GPU to calculate the images augmentation, instead of the cpu. A second one is maybe using a different worker distribution, instead of using functional decomposition, use domain decomposition, or maybe if the communication between processes is the bottle neck we can use ring all reduce like we seen in the tutorial.

Part 2

6. We initialized a Lock , Pipe and an integer counter which is a mp.Value type so it will be shared between processes(we tested len without it and it didn't work). Every time a thread called put, we locked the Lock and first we increased the counter and then added the msg to the pipe, so if the size will be read in the time between the two instructions it will be to the higher side. And ofcourse we unlocked the Lock. On the get we didn't need to lock to use pipe recv because there is only one reader and locking is not needed in this case, and the decreased the counter so if the size will be read in the time between the two instructions it will be to the higher side. In length we only read the counter so no lock is needed.

תכנות מקבילי ומבוזר- תרגיל בית 2

איתי לביא 212147326 הראל ימין 213099492

The locking is only important when you enter messages to the pipe, because there is only one reader and mp.Value is not needed to be synchronized.

חלק 3-

:correlaiton numba

על מנת לבצע קורלציה, יצרנו מטריצה חדשה padded שהיא העתק של התמונה המקורית רק שריפדנו את השוליים שלה (למעלה למטה וצדדים) באפסים. מלמעלה ולמטה ריפדנו בkernel_row שורות של אפסים ובצדדים ריפדנו בkernel_col עמודות אפסים. עשינו זאת כדי שכאשר נחרוג מגבולות התמונה (בחישוב השכנים) לא נצטרך לבדוק גבולות ונוכל להתייחס לפיקסל ה"חריג" כאל 0 כפי שרצינו. בעזרת prange רצנו על הפיקסלים המקוריים (שהיו בתמונה, לא האפסים שהוספנו) וחישבנו את המטריצה pixels, שהיא מטריצת השכנים של הפיקסל הנוכחי (הפיקסל הנוכחי במרכז), וצורתה היא kernel. (עשינו זאת על ידי slicing numpy) לאחר מכן הכפלנו איבר-איבר את kernel וpixels וסכמנו על כל הערכים שהתקבלו על מנת לקבל סכום משוקלל כרצוי. שמרנו את הסכומים המשוקללים במטריצה שמייצגת את התמונה שמתקבלת מהפעלת הקורלציה והחזרנו אותה.

:correlation gpu

יצרנו מטריצה מרופדת באפסים באופן זהה ל-correlation_numba. העתקנו לזיכרון של gpu את המטריצה המרופדת ומטריצת הקרנל. יצרנו מטריצה חדשה new_image שבה נמלא את התוצאה שאמורה להתקבל והעתקנו אותה ל-gpu. לאחר מכן יצרנו גריד של image_row (מס' השורות בתמונה הנתונה) בלוקים image_col (מס' העמודות בתמונה הנתונה) חוטים. כל חוט אחראי על עדכון הפיקסל ה-[block_num, thread_num] במטריצת התוצאה. (מריץ את פונק' הקרנל update_pixel_kernel) הוא עושה זאת באופן כמעט זהה (עד כדי פונקציות שאינן נתמכות בcuda) לאופן שעשינו בפונקציית numba. לאחר שכל החוטים סיימו אנחנו מעתיקים את מטריצת התוצאה new_image בחזרה לcpu ומחזירים אותה.

תכנות מקבילי ומבוזר- תרגיל בית 2

איתי לביא 212147326 הראל ימין 213099492

הערכים שקיבלנו מהרצת הטסט:

```
● (tf23-gpu) itaylavi@lambda:~/CDPDML/hw2$ srun --gres=gpu:1 -c 1 --pty python3 filters_test.py
CPU 3X3 kernel: 0.001539725111797452
Numba 3X3 kernel: 0.006048291921615601
CUDA 3X3 kernel: 0.0013470340054482222
-----
CPU 5X5 kernel: 0.003257359843701124
Numba 5X5 kernel: 0.008361044805496931
CUDA 5X5 kernel: 0.0013586480636149645
-----
CPU 7X7 kernel: 0.005994366016238928
Numba 7X7 kernel: 0.011194897815585136
CUDA 7X7 kernel: 0.001388315111398697
```

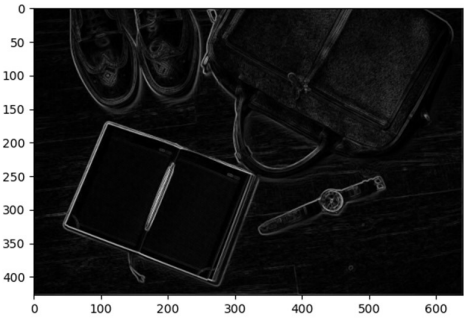
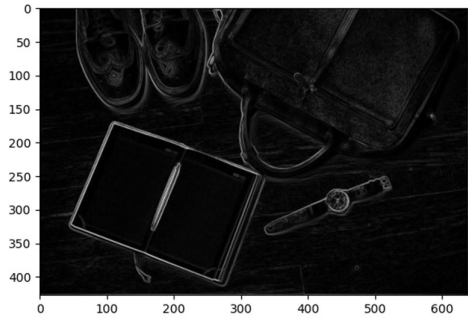
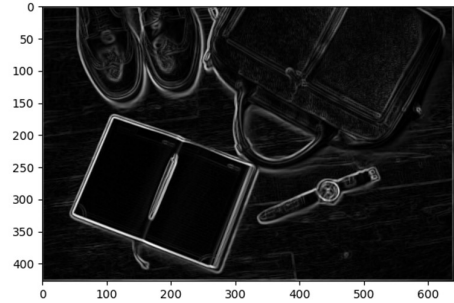
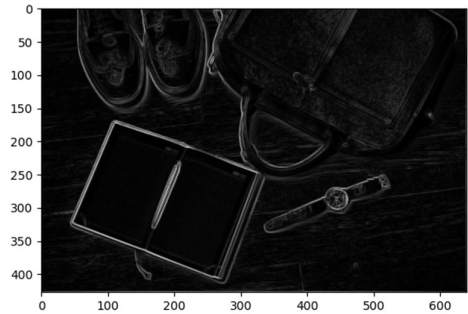

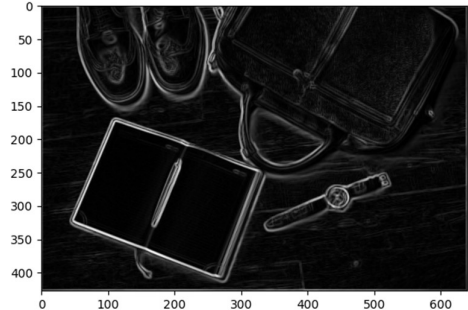
	3x3	5x5	7x7
<i>numba_speedup</i>	$\frac{153972}{604829} \approx 0.254$	$\frac{325736}{836104} \approx 0.389$	$\frac{59943}{111949} \approx 0.535$
<i>gpu_speedup</i>	$\frac{153972}{134703} \approx 1.143$	$\frac{325736}{135865} \approx 2.397$	$\frac{599436}{138831} \approx 4.317$

אילו היינו משתמשים במטריצת קרנל גדולה יותר, היינו מבזבזים זיכרון על שמירה של הרבה אפסים מיותרים שאמנם עוזרים לנו לא לחרוג מגבולות המערך אבל יוצרים תקורה גדולה מאוד של זיכרון. בנוסף, אם גודל הקרנל היה גדול מגודל התמונה, היינו כופלים המון אפסים מיותרים בכל איטרציה של עדכון פיקסל ולכן הייתה תקורה גם של פעולות כפל וסכימה מיותרות.

תכנות מקבילי ומבוזר- תרגיל בית 2

איתי לביא 212147326 הראל ימין 213099492

חלק 4- התוצאות שהתקבלו מהרצת הפילטרים:

פילטר 2	פילטר 1
	
פילטר 4	פילטר 3
	
התמונה המקורית	תוצאת הקורלציה
	

תכנות מקבילי ומבוזר- תרגיל בית 2

איתי לביא 212147326 הראל ימין 213099492

הסבר: כאשר אופרטור Sobel מופעל על תמונה, הוא מזהה את הקצוות שבה. הוא עושה זאת על ידי "גזירת" כל פיקסל בתמונה ומציאת הגרדיאנט שלו. הגרדיאנט יעיד לנו על מידת השינוי של הפיקסל הנוכחי לעומת השכנים שלו. ההבדל בין 4 הפילטרים בהם השתמשנו הוא הקרנל איתו הפעלנו את האופרטור. ה-shape של הקרנל קובע את אזור הפיקסלים מהם נשקלל את המידע והערכים בקרנל קובעים את המשקלות שיהיו לכל פיקסל בחישוב הסכום המשוקלל. ככל שהערכים בקרנל יהיו גדולים יותר (בערכם המוחלט), כך הקצוות שנקבל בתוצאה יהיו יותר בולטים וחדים.

לדוגמה: פילטר 1 משתמש בקרנל 3×3 עם ערכים 1,2,1 ופילטר 2 משתמש בקרנל מאותה צורה עם ערכים 3,10,3. לכן התמונה שתתקבל בפילטר 2 תבליט יותר את הקצוות בתמונה מאשר פילטר 1. לעומת זאת, בגלל שגודל מטריצת הקרנל בשני הפילטרים מאוד קטנה, זיהוי הקצוות יהיה מאוד חלש ולכן כמעט ולא נראה הבדל. ניתן לראות שבשימוש בפילטר 4 שמכיל את אותם הערכים כמו בפילטר 1, הקצוות שמתקבלים הרבה יותר חדים ומפורטים מפני שמטריצת הקרנל שלו יותר גדולה. (5×5) בנוסף, בפילטר 4 ישנן 3 עמודות של אפסים באמצע המטריצה, לכן אנחנו בעצם משתמשים בפיקסלים רחוקים יותר על מנת לזהות את הקצוות. לכן לעומת פילטר מס' 3, פילטר 4 מזהה הרבה יותר טוב קצוות. לכן ככל שגודל הקרנל והערכים שבו גדלים, כך זיהוי הקצוות יהיה חזק ומפורט יותר.