

HW2 Report

תכנות מקבילי ומבוזר לעיבוד נתונים
236370

מיכל עוזרי 325719052
גיא סודאי 214300550

שאלה 1:

```
● (tf23-gpu) michal.ozeri@lambda:~/hw2_cdp$ srun --gres=gpu:1 -c 8 --pty python3 main.py
Epoch 1, accuracy 10.66 %.
Epoch 2, accuracy 61.14 %.
Epoch 3, accuracy 72.24 %.
Epoch 4, accuracy 79.55 %.
Epoch 5, accuracy 82.36 %.
Epoch 6, accuracy 83.12 %.
Epoch 7, accuracy 83.27 %.
Epoch 8, accuracy 83.39 %.
Epoch 9, accuracy 83.59 %.
Epoch 10, accuracy 83.46 %.
Epoch 11, accuracy 83.53 %.
Epoch 12, accuracy 83.37 %.
Epoch 13, accuracy 83.17 %.
Epoch 14, accuracy 83.35 %.
Epoch 15, accuracy 83.33 %.
Time regular: 8.759744882583618
Test Accuracy: 81.29411764705883%
Epoch 1, accuracy 10.9 %.
Epoch 2, accuracy 29.01 %.
Epoch 3, accuracy 64.04 %.
Epoch 4, accuracy 70.39 %.
Epoch 5, accuracy 72.62 %.
Epoch 6, accuracy 75.14 %.
Epoch 7, accuracy 80.44 %.
Epoch 8, accuracy 84.71 %.
Epoch 9, accuracy 85.45 %.
Epoch 10, accuracy 85.57 %.
Epoch 11, accuracy 85.59 %.
Epoch 12, accuracy 87.29 %.
Epoch 13, accuracy 88.52 %.
Epoch 14, accuracy 89.03 %.
Epoch 15, accuracy 87.62 %.
Time with image processing: 37.27217364311218
Test Accuracy: 86.19495798319328%
```

```
● (tf23-gpu) michal.ozeri@lambda:~/hw2_cdp$ srun --gres=gpu:1 -c 16 --pty python3 main.py
Epoch 1, accuracy 26.88 %.
Epoch 2, accuracy 55.06 %.
Epoch 3, accuracy 69.76 %.
Epoch 4, accuracy 79.08 %.
Epoch 5, accuracy 81.85 %.
Epoch 6, accuracy 82.12 %.
Epoch 7, accuracy 82.58 %.
Epoch 8, accuracy 82.55 %.
Epoch 9, accuracy 82.47 %.
Epoch 10, accuracy 82.48 %.
Epoch 11, accuracy 82.4 %.
Epoch 12, accuracy 82.45 %.
Epoch 13, accuracy 82.57 %.
Epoch 14, accuracy 82.29 %.
Epoch 15, accuracy 82.37 %.
Time regular: 8.602523565292358
Test Accuracy: 80.21512605042017%
Epoch 1, accuracy 20.1 %.
Epoch 2, accuracy 29.16 %.
Epoch 3, accuracy 66.31 %.
Epoch 4, accuracy 68.36 %.
Epoch 5, accuracy 73.02 %.
Epoch 6, accuracy 78.26 %.
Epoch 7, accuracy 80.57 %.
Epoch 8, accuracy 82.88 %.
Epoch 9, accuracy 83.45 %.
Epoch 10, accuracy 84.43 %.
Epoch 11, accuracy 86.03 %.
Epoch 12, accuracy 84.96 %.
Epoch 13, accuracy 88.17 %.
Epoch 14, accuracy 87.65 %.
Epoch 15, accuracy 89.06 %.
Time with image processing: 24.144654989242554
Test Accuracy: 87.6%
```

```

(tf23-gpu) michal.ozeri@lambda:~/hw2_cdp$ srun --gres=gpu:1 -c 32 --pty python3 main.py
Epoch 1, accuracy 17.28 %.
Epoch 2, accuracy 61.36 %.
Epoch 3, accuracy 74.45 %.
Epoch 4, accuracy 81.55 %.
Epoch 5, accuracy 82.77 %.
Epoch 6, accuracy 83.21 %.
Epoch 7, accuracy 83.43 %.
Epoch 8, accuracy 83.47 %.
Epoch 9, accuracy 83.56 %.
Epoch 10, accuracy 83.65 %.
Epoch 11, accuracy 83.64 %.
Epoch 12, accuracy 83.8 %.
Epoch 13, accuracy 83.7 %.
Epoch 14, accuracy 83.7 %.
Epoch 15, accuracy 83.7 %.
Time regular: 10.736645221710205
Test Accuracy: 81.96974789915966%
Epoch 1, accuracy 24.15 %.
Epoch 2, accuracy 29.99 %.
Epoch 3, accuracy 65.09 %.
Epoch 4, accuracy 71.97 %.
Epoch 5, accuracy 72.12 %.
Epoch 6, accuracy 75.27 %.
Epoch 7, accuracy 82.48 %.
Epoch 8, accuracy 82.83 %.
Epoch 9, accuracy 84.37 %.
Epoch 10, accuracy 85.32 %.
Epoch 11, accuracy 86.15 %.
Epoch 12, accuracy 87.42 %.
Epoch 13, accuracy 87.17 %.
Epoch 14, accuracy 88.22 %.
Epoch 15, accuracy 88.4 %.
Time with image processing: 21.433645248413086
Test Accuracy: 87.11932773109243%

```

עבור 8 יחידות עיבוד נקבל זמן של: *37.27 seconds*

עבור 16 יחידות עיבוד נקבל זמן של: *24.14 seconds*

עבור 32 יחידות עיבוד נקבל זמן של: *21.43 seconds*

כפי שציפינו, זמן החישוב במגמת ירידה ככל שמספר יחידות העיבוד עולה. הסיבה היא שככל שיש יותר יחידות עיבוד, כך יש יותר תהליכים המריצים את עיבוד התמונות (הגדרנו שמספר הWorkers הוא מספר יחידות העיבוד) וממלאים את תור התוצאות יותר מהר, מה שמאיץ את פעולת `create_batches`.
 לגבי אחוז הדיוק בtest של IPneuralnetwork, אין הבדל משמעותי בין מספר ליבות שונה, ולכן נסיק שהדיוק בtest לא תלוי במספר יחידות העיבוד.

שאלה 2:

לשם ההשוואה נבחר בהרצה עם 16 ליבות (צילום מסך 2 משאלה 1):

<i>epoch</i>	<i>NeuralNetwork</i>	<i>IPNeuralNetwork</i>
1	26.88	20.1
2	55.06	29.16
3	69.76	66.31
4	79.08	68.36
5	81.85	73.02
6	82.12	78.26
7	82.58	80.57
8	82.55	82.88
9	82.47	83.45
10	82.48	84.43
11	82.4	86.03
12	82.45	84.96
13	82.57	88.17
14	82.29	87.65
15	82.37	89.06

ניתן לראות שבepochs הראשונים, *NeuralNetwork* מגיע לדיוק גבוה יותר מאשר *IPNeuralNetwork*. אך עם התקדמות הepochs ניתן לראות ש*IPNeuralNetwork* מגיע לדיוק גבוה יותר ונמצא במגמת עלייה ממש. לעומתו *NeuralNetwork* כן מדייק יותר ככל שמתקדמים בepochs, אך החל מepoch מסוים הוא לא משתפר בדיוק האימון בepochs הבאים ונשאר בערך על דיוק של 82%. ניתן להסיק מכך, שהאוגמנטציה והרנדומיות שבבחירת הפרמטרים לאוגמנטציה, משפרת את הדיוק הן לאורך epochs שונים באימון והן בtest (ניתן לראות את ההבדל בצילום- *NeuralNetwork* משיג 80.2% בtest ו*IPNeuralNetwork* משיג 87.6% - הפרש של 7.4%)

שאלה 3:

באופן מקבילי, בכל יחידה מקבילית, עלינו לקרוא ממאגר משותף, לבצע מספר עיבודי תמונה, ולכתוב לתור משותף.

על פניו, מאחר שישנן פעולות זיכרון משותף רבות, היינו מעדיפים שימוש ב threads על פני processes. (כי ב threads הזיכרון משותף והיינו עם overhead נמוך בתקשורת משל processes).

אבל, נבחין בשני דברים:

1. הפעולה עצמה היא מאוד CPU-bound (המון פעולות חישוב לכל פיקסל, לכל תמונה)
2. ב python, בגלל ה GIL אין הרצה מקבילית אמיתית של threads, אלא רק אחד רץ בכל זמן נתון. מכך נסיק, שבשימוש ב threads בpython, בצורה אפקטיבית יהיה שקול להרצת process אחד (כל פעם רץ thread אחד, וכל שהוא מבצע זה המשך של פעולות חישוב) ואילו בשימוש ב processes, נקבל ריצה מקבילית אמיתית של מספר תהליכים, אבל עם התקורה של העברת התקשורת של העבודות והתוצאות. אבל הפעולה הנדרשת היא CPU-bound כך שהתקורה מינורית והאצה משמעותית.

שאלה 4:

* נבצע את מקבול אוגמנטציית התמונות ב GPU:

ב-multicore, במימוש הנוכחי, יש לעבור פיקסל פיקסל באופן סדרתי לצורך חישוב כל פעולה סטטית שהגדרנו לחישוב התמונה החדשה.

אבל כל פעולה כזו של חישוב פיקסל אינה תלויה בחישוב האחרים, ולכן נוכל למקבל אותה בצורה טובה ב GPU.

נוכל להגדיר ב GPU חוט לחישוב כל פיקסל בתמונה הנוכחית, ונחשב תמונות שונות במקביל על פני בלוקים שונים.

כך, נאיץ את שלב החישוב הכבד באוגמנטציה (חישובים פר פיקסל בצורה מקבילית על פני סדרתית) ונצפה לתוצאות טובות יותר.

* נמקבל את האימון עצמו:

למדנו בהרצאה ובתרגול על 2 שיטות למקבול אימון רשת ניורונים:

האחת לפי חלוקת datan ליחידות מקביליות - כל יחידה תחזיק את כל הרשת.

והשנייה חלוקה לפי שכבות הרשת - כל יחידה תחזיק שכבה אחת ותעבוד על כל datan.

לפי אחת משיטות אלו ניתן למקבל את האימון ובכך להשיג ביצועים טובים יותר.

שאלה 5:

attributes:

Pipe - נממש את התור באמצעות pipe עם צד קורא וצד כותב.

Lock - המיועד לכותבים משום שנתון שיהיה לנו רק קורא אחד והרבה כותבים לכן אין צורך בסנכרון הצד הכותב.

get:

פשוט נחזיר את האיבר הראשון בpipe. אין צורך לסנכרן כי יש רק קורא אחד.

put:

גישת הכותבים צריכה להיות מסונכרנת כי יש הרבה כותבים. לכן נרצה לעטוף את ההכנסה לpipe ברכישת מנעול ושחרור מנעול.

empty:

נשאל באמצעות poll האם הpipe ריק.

ניתן להניח שהמתודה empty תיקרא רק על ידי התהליך הקורא. בנוסף, מותר שהמתודה תחזיר שהתור ריק גם אם קיימת הודעה שכרגע נשלחת – המתודה צריכה להחזיר שהתור אינו ריק רק אם אכן קיימת הודעה שנשלחה וגם שהתהליך ששלח אותו סיים לבצע את תוכן המתודה ולכן ניתן לא לסנכרן את הפעולה poll.

שאלה 6:

correlation gpu:

ראשית, נגדיר מערכי קלט ומערך פלט על המחשב המקומי ונעביר אותם לGPU.

מערכי הקלט הם: dev_image - מכיל את התמונה מרופדת בשורות ובעמודות אפסים. ובפירוט, אם מטריצת הקרנל היא מסדר $k_rows \times k_columns$ אז dev_image תוסיף על image מלמעלה ומלמטה עוד

$\left\lfloor \frac{k_rows}{2} \right\rfloor$ שורות אפסים ו- $\left\lfloor \frac{k_columns}{2} \right\rfloor$ עמודות אפסים משני צדדיה. dev_kernel - מטריצת הקרנל

כפי שקיבלנו אותה.

מערך הפלט - מערך מלא באפסים מאותו סדר של התמונה המקורית.

אם מטריצת התמונה המקורית היא מסדר $rows \times columns$ אז נגדיר $num\ of\ blocks = rows$ ובכל בלוק, $num\ of\ threads = columns$ כך שכל thread יקבל תא במטריצת הפלט לחשב.

נקרא ל'apply_kernel' עם הפרמטרים הנ"ל שתופעל בGPU. ואז כל חוט יבצע את החישוב על התא המוגדר על ידי tx, bx כאשר tx זה מזהה החוט בבלוק וbx זה מזהה הבלוק של החוט.

correlation numba:

כמו במימוש של GPU, ניצור מהתמונה מטריצה המכילה אותה ומרופדת באפסים. נאתחל מטריצת קלט מאותו סדר כמו של התמונה המקורית. נרוץ בלולאות תוך שימוש בprange על מנת שnumba יוכל למקבל אותן (לשם כך נוסיף באנוטציה: parallel = true). לכל תא x,y במטריצת הפלט נבצע את החישוב: לכל תא i,j במטריצת הקרנל:

$$C[x, y] += zero_padded_image[x + i, y + j] * kernel[i, j]$$

וזה יחשב לנו את הממוצע המשוקלל עבור התא x,y במטריצת הפלט.

שאלה 7:

```
(tf23-gpu) michal.ozeri@lambda:~/hw2_cdp$ srun --gres=gpu:1 -c 1 --pty python3 filters_test.py
CPU 3X3 kernel: 0.001532992348074913
Numba 3X3 kernel: 0.002076755277812481
CUDA 3X3 kernel: 0.0015040002763271332
-----
CPU 5X5 kernel: 0.0032616527751088142
Numba 5X5 kernel: 0.0030905595049262047
CUDA 5X5 kernel: 0.001506982371211052
-----
CPU 7X7 kernel: 0.006451036781072617
Numba 7X7 kernel: 0.004479540511965752
CUDA 7X7 kernel: 0.0015171775594353676
-----
```

ניתן לראות כי לכל מטריצות הקרנל המימוש של CUDA לוקח אותו הזמן בערך אך ניתן לראות מגמת עלייה ככל שסדר מטריצת הקרנל גדול יותר. ניתן לצפות זאת כיוון שיצרנו חוטים כמספר תאי המטריצה של התמונה כאשר כל חוט אחראי על חישוב תא אחד. ככל שמטריצת הקרנל גדולה יותר, כך יש יותר חישובים לכל חוט לעשות.

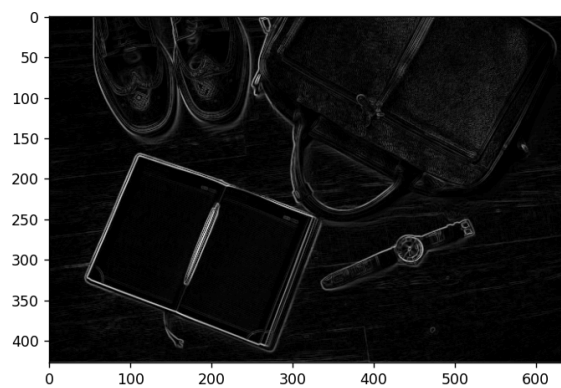
שנית, נבחין כי CUDA מהיר יותר מNumba ומהיר יותר מconvolve2D של scipy לכל מטריצות הקרנל- עבור המטריצה 3x3 מהיר יותר בהפרש זעיר כלומר $speedup \approx 1$. עבור המטריצה 5x5 מהיר פי 2 $speedup \approx 2$. ועבור המטריצה 7x7 מהיר פי 3 $speedup \approx 3$ מהימוש של Numba ומהיר פי 4 $speedup \approx 4$ מהימוש של scipy.

שאלה 8:

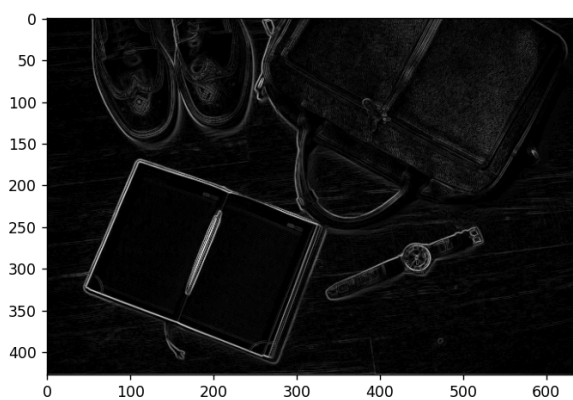
אם היינו משתמשים במטריצת kernel גדולה יותר בחלק ג אז היינו מקבלים העצמה של הפעולה שהתכוונו לעשות על התמונה. למשל מטריצת קרנל שהייתה אמורה לטשטש את התמונה, תטשטש אותה יותר ומטריצת קרנל שאמורה לחדד את התמונה תחדד אותה יותר, בשני המקרים נקבל יותר איבוד מידע. שכן, בטשטוש, ככל שהמטריצה גדולה יותר, כך ערך התוצאה בכל פיקסל דומה יותר לשכניו. ובחידוד, ככל שמטריצת הקרנל גדולה יותר, כך השונות בין כל פיקסל לשכניו גדלה. בנוסף, עבור מטריצת קרנל גדולה יותר, נקבל ביצועים פחות טובים (זמן ריצה גבוה יותר). משאלה 7 נסיק, כי עבור המימוש של CUDA ההפחתה בביצועים תהיה הרבה פחות משמעותית מההפחתה בביצועים עבור המימוש של Numba ו-scipy.

שאלה 9:

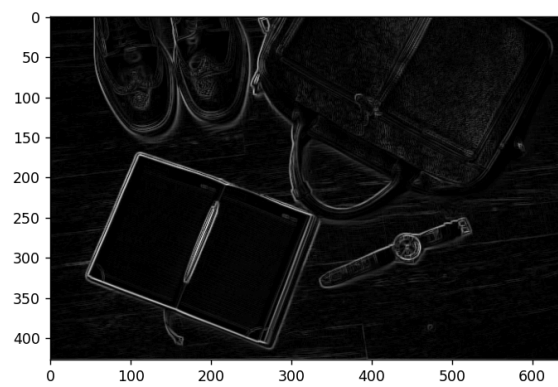
sobel_filter



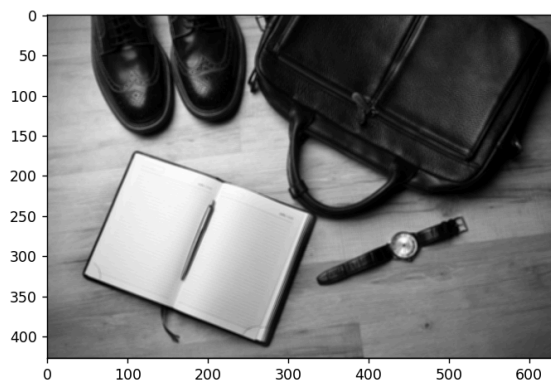
kernel1



kernel2



kernel3



תחילה, נסביר את הדימיון בין $sobel$, $kernel1$, $kernel2$.
נסתכל על המטריצות המתארות אותן:

$$sobel_filter = \begin{pmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{pmatrix}$$

$$kernel_1 = \begin{pmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{pmatrix}, kernel_2 = \begin{pmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{pmatrix},$$

נבחין כי בכולן, ערכי העמודה הראשונה, זהים ובסימן הפוך לעמודה השלישית, והעמודה השנייה כולה אפסים. בחישוב הקורלציה עבור פיקסל מסוים, אנחנו סוכמים את הפיקסלים בסביבה בהתאמה למשקלים, ולכן:

1. אם ערכי הפיקסלים העמודה הראשונה והשלישית דומים/קרובים בערכם, בסכימה נוסף ונחסיר את אותם הערכים ונקבל פיקסל הקרוב בערכו ל 0 (כלומר שחור).
2. אם הערכים שונים מאוד אחד מהשני בצבעם, אז בסכימה כן נקבל הפרש שכבר אינו אפס והתמונה והפיקסל המחושב יקבל צבע בולט.

מסקנה: קווי המתאר של אובייקטים בתמונה ובמעברים חדים בצבע, אשר בהם יש הפרש דרסטי בצבעי הפיקסלים בסביבה (שכן עוברים בין אובייקטים שצבעם שונה) נראה צבעים בוהקים, בעוד ש בצבעים קרובים יהיה שחור.
מכאן נובע הדימיון הרב שלהם.

השוני בין $sobel$, $kernel1$, $kernel2$:
הסברנו שאופן העבודה של הקרנלים דומה, אבל רואים שהמשקלים הניתנים לכל פיקסל שונים, וטווח הפקסלים שמסתכלים עליו שונה ביניהם, לכן כן נצפה להבדלים כלשהם. ואכן ניתן לראות שבין הקרנלים השונים מקבלים אזורים לבנים יותר ולבנים פחות, מקומות בהם קרנל אחד נתן חשיבות לשינוי צבע יותר מאשר לקרנל השני, בהתאמה למשקלים ולסביבה.

:kernel3

שונה מאוד מהאחרים, המבנה שלו הוא כזה שמשכלל בחיוב כל פיקסל מהסביבה.
לכל פיקסל, קובעים את ערכו להיות סכום הפיקסלים השכנים שלו. כלומר אנו "מערבבים" את הצבעים השכנים
לערכו של הנוכחי, ולכן מקבלים אפקט של טשטוש בתמונה.