

HW2 Report

תכנות מקבילי וմבוזר לעיבוד נתונים
236370

מיכל עזרי 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 נקלט זמן של: 37.27 seconds

עבור 16 יחידות עיבוד נקלט זמן של: 24.14 seconds

עבור 32 יחידות עיבוד נקלט זמן של: 21.43 seconds

כפי שציפינו, זמן החישוב ב大妈ת רידה ככל שמספר יחידות העיבוד עולה. הסיבה היא שככל שיש יותר יחידות

עיבוד, כך יש יותר תהליכי המריצים את עיבוד התמונות (הגדreno שמספר הוראות Workers הוא מספר יחידות

העיבוד) וממלאים את תור התוצאות יותר מהר, מה שמאיץ את פעולה `.create_batches`.

לגביה אוחז הדיק בtest של IPneuralnetwork, אין הבדל משמעותי בין מספר ליבות שונה, וכך נסיק שהדיק

בtest לא תלוי במספר יחידות העיבוד.

שאלה 2:

לשם ההשוואה נבחר בהרצה עם 16 ליבות (צלום מס' 2 משאלת 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 לדיק גבוה יותר מאשר *IPNeuralNetwork*. אך עם התקדמות epochs ניתן לראות *IPNeuralNetwork* מגיע לדיק גבוה יותר ונמצא בוגמת עלייה ממש. לעומת זאת *NeuralNetwork* כנ"ד מ Dick יותר בכל שמתקדמים בepochs, אך החל מepoch מס' 10 לא משתפר בDick האימון בepochs הבאים ונסחר בערך על Dick של ~82%. ניתן להסיק מכך, שהאגמנטציה והרנדומיות שבבחירה הפרמטרים לאוגמנטציה, משפרת את הדיק ה-*test* באמצעות epochs שונים באימון והן ב-*test* (ניתן לראות את ההבדל בצלום- *NeuralNetwork* מושג 80.2% ב-*IPNeuralNetwork* מושג 7.4% - הפרש של 87.6%)

שאלה 3:

באופן מקבילי, בכל יחידה מקבילתית, علينا לקרוא ממאגר משותף, לבצע מספר עיבודי תמונה, ולכתוב לתור משותף.

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

אבל, נבחן בשני דברים:
1. הפעולה עצמה היא מאוד CPU-bound (המון פעולות חישוב לכל פיקסל, לכל תמונה)

2. ב python, בגלל ה GUI אין הרצה מקבילתית אמיתית של threads, אלא רק אחד רץ בכל זמן נתון. מכך נסיק, שבשימוש ב threads בpython, במצב אפקטיבי יהיה קל להרצת process אחד (כל פעם רק

thread אחד, וכל שהוא מבצע זה המשך של פעולות חישוב) ואילו בשימוש ב processes, נקבל ריצה מקבילתית אמיתית של מספר תהליכים, אבל עם התקורה של העברת התקשורת של העבודות והטוצאות. אבל הפעולה הנדרשת היא CPU-bound כך שהתקורה מינורית והאצה משמעותית.

שאלה 4:

* נבצע את מקבול אוגמנטציה התמונות ב GPU: ב-alignment, במימוש הנוכחי, יש לעבור פיקסל באופן סדרתי לצורך חישוב כל פעולה סטטistica שהגדנו לחישוב התמונה החדשה.

אבל כל פעולה כזו של חישוב פיקסל אינה תליה בחישוב האחרים, ולכן ניתן לקבל אותה בצורה טובה ב GPU.

אפשר להציג ב GPU חוט לחישוב כל פיקסל בתמונה הנוכחי, ונחשב תמונות שונות במקביל על פני בלוקים שונים.

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

* מקבל את האימון עצמו:

למදנו בהרצה ובתרגול על 2 שיטות למקבול אימון רשת נירוניים:

הachat לפ'י חלוקת data ליחידות מקבילות - כל יחידה תחזיק את כל הרשות.

והשניה חלוקה לפי שכבות הרשות - כל יחידה תחזיק שכבה אחת ותעבד על כל data.

לפי אחת-methodologies אלה ניתן לקבל את האימון ובכך להשיג ביצועים טובים יותר.

שאלה 5:

:attributes

Pipe- נממש את התור באמצעות `seek` עם צד קורא הצד כתוב.
Lock- המועד לכותבים משומם שנותן שייהה לו רק קורא אחד והרבה כותבים لكن אין צורך בסyncron הצד הכותב.

:get

פשוט נחזיר את האיבר הראשון בSEQUENCE. אין צורך לסינכרן כי יש רק קורא אחד.

:put

גישת הכותבים צריכה להיות מסונכנת כי יש הרבה כותבים. לכן נרצה לעטוף את ההכנסה לSEQUENCE ברכישת מנעול וחרור מנעול.

:empty

נשאל באמצעות `poll` האם הSEQUENCE ריק.

ניתן להניח שהמתודה `empty` תיקרא רק על ידי התהילה הקורא. בנוסף, מותר שהמתודה תחזיר שהטור ריק גם אם קיימת הודעה שכרגע נשלה – המתודה צריכה להחזיר שהטור אינו ריק רק אם אכן קיימת הודעה שנשלה וגם שהטהילה ששלחו אותו סיימם לבצע את תוכן המתודה ולכן ניתן לא לסינכרן את הפעולה `poll`.

שאלה 6:

:correlation gpru

ראשית, נגדיר מערך קלט ומערך פלט על המחשב המקומי ונעביר אותם לGPU.
מערכי הקלט הם: `dev_image` - מכיל את התמונה מרופדת בשורות ובעמודות אפסים. ובפירוט, אם מטריצת הקernal היא מסדר `dev_image` $k_{rows} \times k_{columns}$ אז `dev_image` על `rows` על `image` מלמעלה ומלמטה ועד $\left\lceil \frac{k_{columns}}{2} \right\rceil$ שורות אפסים ו- $\left\lceil \frac{k_{rows}}{2} \right\rceil$ עמודות אפסים שני צדדיה. `dev_kernel` - מטריצת הקernal כפי שקיבלנו אותה.

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

אם מטריצת התמונה המקורית היא מסדר `rows` \times `columns` אז נגדיר `rows = num of blocks` ובכל בлок, `columns = num of threads` כך שכל thread יקבל תא במטריצת הפלט לחשב. נקרא `apply_kernel` עם הפרמטרים הנ"ל שתופעל בGPU. ואז כל חוט יבצע את החישוב על התא המוגדר על ידי `ax, bx, cx, dx` כאשר `dx` זה מזהה החוט בבלוק ו`bx` זה מזהה הבלוק של החוט.

:correlation numba

כמו בימוש של GPU, ניצור מהתמונה מטריצה המכילה אותה ומרופדת באפסים. נאתחל מטריצת קלט מאותו סדר כמו של התמונה המקורית. נróż בלולאות תוך שימוש `prange` על מנת שאמבעת יוכל לקבל אותן (לשם כך נוסיף באנטציה: `parallel = true`). לכל תא i, j במטריצת הפלט נבצע את החישוב: לכל תא j , במטריצת הקernal:

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

זהה יחשב לנו את הממוצע המשווקל עברו התא i, j 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 הkernel המימוש של CUDA לוקח אותו הזמן בערך אך ניתן לראות מגמת עלייה כל שסדר מטריצת הkernel גדול יותר. ניתן לצפות זאת כיון שייצרנו חוטים כמספר תאי המטריצה של התמונה כאשר כל חוט אחראי על חישוב תא אחד. ככל שמטריצת הkernel גדולה יותר, כך יש יותר חישובים לכל חוט לעשوت.

שנית, נבחן כי CUDA מהיר יותר Numba ומהיר יותר `fft2Dn` של `convolve` לכל מטריצות הkernel -

עבור המטריצה 3×3 מהיר יותר בהפרש ≈ 1 כלומר $speedup \approx 1$.

עבור המטריצה 5×5 מהיר פי $2 \approx 2$ $speedup$.

עבור המטריצה 7×7 מהיר פי $3 \approx 3$ מהימוש של Numba ומהיר פי $4 \approx 4$ מהימוש של `scipy`.

שאלה 8:

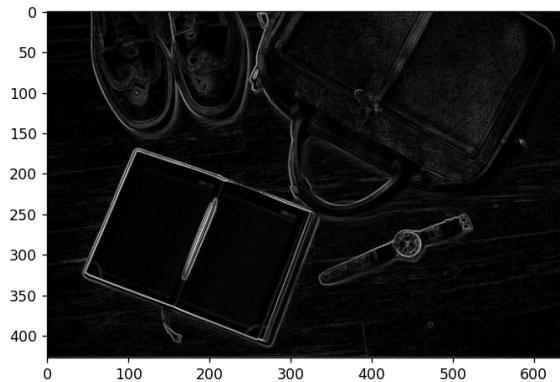
אם היינו משתמשים במטריצת `kernel` גדולה יותר בחלק ג' אז היינו מקבלים העצמה של הפעולה שהתכוונו לעשות על התמונה. למשל מטריצת kernel שהיתה אמרורה לטשטש את התמונה, תטשטש יותר ומטריצת kernel שאמורה לחדד את התמונה תחזר אותה יותר, בשני המקרים נקבל יותר איבוד מידע. שכן, בטשטוש, ככל שמטריצה גדולה יותר, כך ערך התוצאה בכל פיקסל דומה יותר לשכני. ובוחידוד, ככל שמטריצת הkernel גדולה יותר, כך השונות בין כל פיקסל לשכנים גדלה.

בנוסף, עבור מטריצת kernel גדולה יותר, נקבל ביצועים פחות טובים (זמן ריצה גבוהה יותר). מ שאלה 7 נסיק, כי

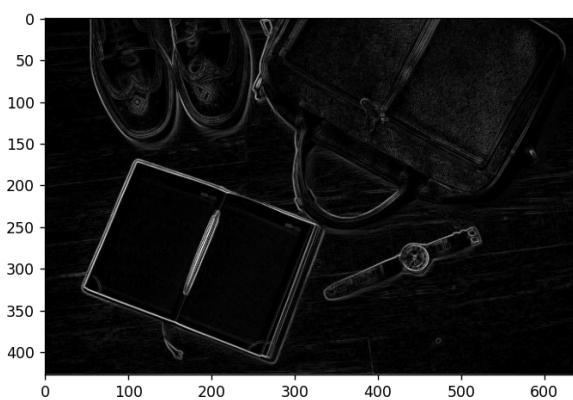
עבור המימוש של CUDA ההפחטה ביצועים תהיה הרבה פחות משמעותית מההפחטה ביצועים עבור המימוש של Numba ו-`scipy`.

שאלה 9

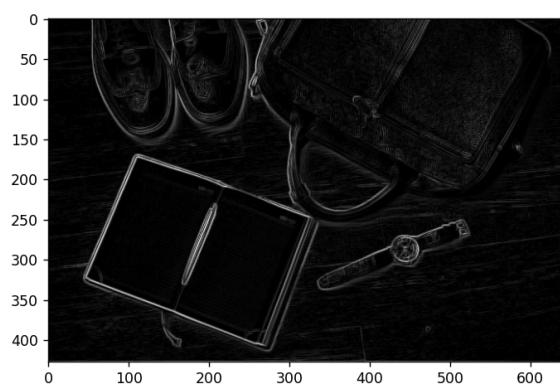
sobel_filter



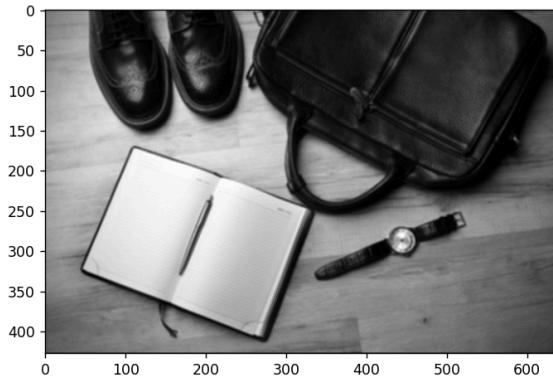
kernel1



kernel2



kernel3



תחילה, נסביר את הדימויין בין $sobel$, $kernel1$, $kernel2$ ו- $kernel3$.
נסתכל על המטריצות המתארות אותן:

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

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

נבחן כי בכללן, ערכי העמודה הראשונה, זרים ובסימן הפוך לעמודה השלישית, והעמודה השנייה יכולה אפסים. בחישוב הקורלציה עבור פיקסל מסוים, אנחנו סוכמים את הפיקסלים בסביבה בהתאם למשקלים, וכך:
 1. אם ערכי הפיקסלים העמודה הראשונה והשלישית דומים/קרובים בערךם, בסכימה נוספת נוסיף וחסיר את אותם הערךם ונקבל פיקסל הקרוב בערךו ל 0 (כלומר שחור).
 2. אם הערךם שונים מאוד מהשני בצבעם, אז בסכימה.cn נקבל הפרש שכבר אינו אפס והתמונה והפיקסל המחשב יקבל צבע בולט.

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

השוני בין $sobel$, $kernel1$, $kernel2$ ו- $kernel3$:
 הסברנו שאופן העבודה של הקרים דומה, אבל רואים שהמשקלים הניתנים לכל פיקסל שונים, ותווחה הפיקסלים שמסתכלים עליו שונה ביןיהם, לכן כן נצפה להבדלים שלהם. ואכן ניתן לראות שבין הקרים השונים מקבלים אזוריים לבנים יותר וללבנים פחות, מקומות בהם קרナル אחד נתן חשיבות לשינוי צבע יותר מאשר לקרナル השני, בהתאם למשקלים ולסביבה.

:kernel3

שונה מאוד מהאחרים, המבנה שלו הוא צזה שמשכלל בחיבור כל פיקסל מהסביבה. לכל פיקסל, קובעים את ערכו להיות סכום הפיקסלים השכנים שלו. כלומר אנו "מערבים" את הצלעים השכנים לערכו של הנוכחי, וכך מקבלים אפקט של טשטוש בתמונה.