

236370 - תכנות מקבילי ומבוזר לעיבוד נתונים ולמידת מכונה

דוח תרגיל בית 2

טל בן עמי 212525257

קורן מעברי 207987314

דוח מסכם:

1. לא הצלחתי להריץ ביותר מ 4 ליבות את התכנית על השרת.
הרצתי לוקלית לכן התוצאות עלולות להשתנות
הריצות שביצעתי הן על 4,8,12 ליבות:

```
Epoch 1, accuracy 9.67 %.
Epoch 2, accuracy 64.8 %.
Epoch 3, accuracy 70.9 %.
Epoch 4, accuracy 80.77 %.
Epoch 5, accuracy 82.59 %.
Epoch 6, accuracy 83.13 %.
Epoch 7, accuracy 83.2 %.
Epoch 8, accuracy 83.09 %.
Epoch 9, accuracy 83.01 %.
Epoch 10, accuracy 83.19 %.
Epoch 11, accuracy 83.01 %.
Epoch 12, accuracy 82.99 %.
Epoch 13, accuracy 83.35 %.
Epoch 14, accuracy 83.24 %.
Epoch 15, accuracy 83.18 %.
Time regular: 2.5879921913146973
Test Accuracy: 81.34957983193277%
Creating 8 workers
Epoch 1, accuracy 22.08 %.
Epoch 2, accuracy 62.57 %.
Epoch 3, accuracy 73.67 %.
Epoch 4, accuracy 83.61 %.
Epoch 5, accuracy 84.39 %.
Epoch 6, accuracy 84.1 %.
Epoch 7, accuracy 86.29 %.
Epoch 8, accuracy 88.18 %.
Epoch 9, accuracy 89.07 %.
Epoch 10, accuracy 89.53 %.
Epoch 11, accuracy 90.17 %.
Epoch 12, accuracy 90.15 %.
Epoch 13, accuracy 90.34 %.
Epoch 14, accuracy 90.7 %.
Epoch 15, accuracy 89.83 %.
Time with image processing: 6.563018798828125
Test Accuracy: 88.70420168067227%
```

```
Epoch 1, accuracy 26.05 %.
Epoch 2, accuracy 63.81 %.
Epoch 3, accuracy 75.35 %.
Epoch 4, accuracy 80.08 %.
Epoch 5, accuracy 82.33 %.
Epoch 6, accuracy 83.21 %.
Epoch 7, accuracy 83.63 %.
Epoch 8, accuracy 83.8 %.
Epoch 9, accuracy 83.63 %.
Epoch 10, accuracy 83.78 %.
Epoch 11, accuracy 83.78 %.
Epoch 12, accuracy 83.85 %.
Epoch 13, accuracy 83.83 %.
Epoch 14, accuracy 83.89 %.
Epoch 15, accuracy 83.81 %.
Time regular: 2.719339370727539
Test Accuracy: 81.77310924369749%
Creating 4 workers
Epoch 1, accuracy 21.78 %.
Epoch 2, accuracy 61.95 %.
Epoch 3, accuracy 72.94 %.
Epoch 4, accuracy 81.27 %.
Epoch 5, accuracy 85.03 %.
Epoch 6, accuracy 86.58 %.
Epoch 7, accuracy 87.37 %.
Epoch 8, accuracy 87.8 %.
Epoch 9, accuracy 86.68 %.
Epoch 10, accuracy 89.82 %.
Epoch 11, accuracy 89.08 %.
Epoch 12, accuracy 90.33 %.
Epoch 13, accuracy 89.7 %.
Epoch 14, accuracy 90.51 %.
Epoch 15, accuracy 91.07 %.
Time with image processing: 6.109336853027344
Test Accuracy: 89.89243697478992%
```

```
(hw2) C:\programming\parallel\hw2_cdp>python main.py
Epoch 1, accuracy 21.98 %.
Epoch 2, accuracy 60.68 %.
Epoch 3, accuracy 74.91 %.
Epoch 4, accuracy 82.51 %.
Epoch 5, accuracy 83.69 %.
Epoch 6, accuracy 84.42 %.
Epoch 7, accuracy 84.36 %.
Epoch 8, accuracy 84.04 %.
Epoch 9, accuracy 83.97 %.
Epoch 10, accuracy 83.9 %.
Epoch 11, accuracy 83.79 %.
Epoch 12, accuracy 83.87 %.
Epoch 13, accuracy 83.79 %.
Epoch 14, accuracy 83.84 %.
Epoch 15, accuracy 83.86 %.
Time regular: 2.6298906803131104
Test Accuracy: 82.0%
Creating 12 workers
Epoch 1, accuracy 32.47 %.
Epoch 2, accuracy 61.43 %.
Epoch 3, accuracy 73.91 %.
Epoch 4, accuracy 85.41 %.
Epoch 5, accuracy 87.29 %.
Epoch 6, accuracy 86.21 %.
Epoch 7, accuracy 87.29 %.
Epoch 8, accuracy 89.19 %.
Epoch 9, accuracy 88.35 %.
Epoch 10, accuracy 89.51 %.
Epoch 11, accuracy 88.28 %.
Epoch 12, accuracy 90.5 %.
Epoch 13, accuracy 91.07 %.
Epoch 14, accuracy 89.41 %.
Epoch 15, accuracy 91.15 %.
Time with image processing: 7.716958284378052
Test Accuracy: 90.33445378151261%
```

מספר יחידות העיבוד שנתן את התוצאה הטובה ביותר: בהתבסס על זמני הריצה, השימוש ב-4 יחידות עיבוד (Workers) הניב את הזמן המהיר ביותר (6.109 שניות). ככל שהגדלנו את מספר ה-Workers (ל-8 ול-12), זמן הריצה הכולל לא השתפר אלא דווקא התארך.

זה נובע מ:

(1) תקורת הנעילות שמבצעים על התורים דרך תורים דורשת זמן יקר של סנכרון. כאשר פעולת העיבוד עצמה (האוגמנטציה) היא יחסית קצרה, הזמן שלוקח לנהל את התקשורת עולה על הזמן שנחסך בחישוב המקבילי.

(2) יצירה וניהול של מספר רב של תהליכים צורכים משאבי מערכת וזמן שמושקע ב context switching.

(3) חוק אמדל (Amdahl's Law): ישנו חלק סדרתי בתוכנית שאינו ניתן למקבול. הוספת מעבדים משפרת רק את החלק המקבילי, אך התקורה הנוספת גורמת בסופו של דבר להאטה (Parallel Slowdown) כאשר עוברים את נקודת האיזון האופטימלית עבור גודל המשימה הספציפי הזה.

```
• (tf23-gpu) tal~ben@lambda:~/hw2_cdp$ srun --gres=gpu:1 -c 2 --pty python3 main.py
Epoch 1, accuracy 20.3 %.
Epoch 2, accuracy 49.49 %.
Epoch 3, accuracy 73.72 %.
Epoch 4, accuracy 78.96 %.
Epoch 5, accuracy 82.77 %.
Epoch 6, accuracy 83.21 %.
Epoch 7, accuracy 83.55 %.
Epoch 8, accuracy 83.9 %.
Epoch 9, accuracy 83.9 %.
Epoch 10, accuracy 83.88 %.
Epoch 11, accuracy 84.03 %.
Epoch 12, accuracy 84.03 %.
Epoch 13, accuracy 84.01 %.
Epoch 14, accuracy 83.95 %.
Epoch 15, accuracy 83.61 %.
Time regular: 9.45073676109314
Test Accuracy: 82.0%
Creating 2 workers
Epoch 1, accuracy 20.55 %.
Epoch 2, accuracy 64.35 %.
Epoch 3, accuracy 76.09 %.
Epoch 4, accuracy 83.32 %.
Epoch 5, accuracy 86.89 %.
Epoch 6, accuracy 85.99 %.
Epoch 7, accuracy 88.43 %.
Epoch 8, accuracy 89.13 %.
Epoch 9, accuracy 88.26 %.
Epoch 10, accuracy 88.63 %.
Epoch 11, accuracy 89.6 %.
Epoch 12, accuracy 90.61 %.
Epoch 13, accuracy 90.45 %.
Epoch 14, accuracy 91.18 %.
Epoch 15, accuracy 90.55 %.
Time with image processing: 45.93725109100342
Test Accuracy: 89.26218487394958%
```

2. ניתן לראות שאחוזי הדיוק ב-IPNeuralNetwork (הכוללת אוגמנטציות) גבוהים יותר הן ב-Train והן ב-Test לעומת הרשת הרגילה. הסיבה לכך היא שהאוגמנטציות מייצרות דוגמאות אימון מגוונות (סיבובים, רעש, הזזות) מתוך המידע הקיים. גיוון זה מקשה על המודל לשנן את המידע (Overfitting) ומאלץ אותו ללמוד מאפיינים כלליים יותר של הספרות, מה שמוביל ליכולת הכללה טובה יותר על נתונים שלא ראה (Test Set).

3. בחרנו את מספר ה workers כמספר הליבות בהן התהליך מוגדר לרוץ בשרת. אך אם היינו בוחרים יותר מידי workers אז הזמן overhead של אתחולם וסנכרון יהיה דומיננטי מזמן העבודה שייקח לליבות לסיים באופן כזה שסכום הזמנים יהיה גדול מהזמן העבודה שיקח לליבה אחת לסיים ולכן נקבל ירידה בביצועים. אם היינו בוחרים יותר workers מ cores אז היינו גם מקבלים תקורת context switching בין התהליכים שיוגדרו על אותה ליבה. אם היינו בוחרים מעט מידי workers אז הפעם overhead האתחול וסנכרון יהיה יותר קטן וזמן

העבודה יהיה יותר גדול כך שסכום הזמנים גם כן יהיה יותר גדול מזמן עבודה שיקח לליבה אחת לסיים.

4. python משתמש במנעול גלובלי (GIL) שנועל את כל משאבי ה runtime של ה interpreter שבהם חוט יכול לגעת בכדי למנוע data races בזמן אינטרפרטציה של תכנית python. (לאחרונה יש דיבור על החלפת המימוש כך שיתאפשר ההסרה של המנעול). מכאן שאנחנו משתמשים בספריית multiprocessing שבעצם עושה שימוש בתהליכים שונים שלא משתפים את אותו מרחב הזיכרון ב HEAP כך שאין חשש מ data races פנימיים של ה interpreter (אך כן יש צורך לדאוג בקוד עצמו שאין גישות בעייתיות למשאבים משותפים של התהליכים כפי שראינו בתרגיל)

5. ניתן לבצע את האוגמנטציות ב GPU ולהעביר את כל משקלי המודל והדוגמאות המקוריות גם כן ל GPU כך שכל החישובים שמתרחשים בתהליך האימון יתבצעו ב GPU במקום במעבד. ניתן לייצר ולשמור את האוגמנטציות כך שיהיה ניתן ישר להשתמש בהן בריצות הבאות במקום ליצור אותם במהלך ה- run time.

6. ממשנו את התור באמצעות multiprocessing.Pipe. השפעת הנחת קורא יחיד: המימוש שלנו משתמש ב-Lock אך ורק בפונקציית ה-put (כדי לסנכרן בין כותבים מרובים שעלולים לנסות לכתוב ל-Pipe בו זמנית). בפונקציית ה-get, אנו קוראים ישירות (`self.recv_conn.recv()`) ללא מנעול. לולא הנחה זו: אם היו מספר תהליכים קוראים (Readers), היינו חייבים להוסיף מנעול (Lock) גם בפונקציית ה-get. ללא סנכרון בקריאה, שני תהליכים היו עלולים לנסות לקרוא מה-Pipe בו-זמנית, מה שהיה מוביל למצב שבו הודעה 'נקרעת' או שגיאות בגישה למשאב המשותף.

7. מימוש correlation_numba :

השתמשנו בדקורטור `@njit(parallel=True)` המימוש נועד להאיץ את חישוב הקורלציה על המעבד באמצעות מהדר jit של ספריית Numba. השתמשנו בדקורטור זה כדי לקמפל את פונקציית הפייתון לקוד מכונה יעיל בזמן ריצה. הפרמטר `parallel=True` מאפשר ל-Numba לבצע אופטימיזציה של לולאות באופן מקבילי אוטומטי (Auto-parallelization) על גבי ליבות המעבד השונות.

הכנת הנתונים (Padding):

לפני ביצוע הקורלציה, ביצענו ריפוד (Padding) של התמונה באפסים מסביב לשוליה. גודל הריפוד (`pad_h, pad_w`) מחושב כחצי מגודל הקרנל. שלב זה הכרחי כדי שנוכל לחשב את הקורלציה גם עבור הפיקסלים הנמצאים בקצוות התמונה (אנו לא משפיעים על התוצאות בגלל האפסים) המקורית מבלי לחרוג מגבולות המערך.

לולאת החישוב (prange):

הלולאה החיצונית משתמשת ב-prange (במקום range רגיל). זוהי פקודה ספציפית ל-Numba המורה למהדר לחלק את האיטרציות של הלולאה (במקרה זה, השורות של התמונה) בין התהליכים (Threads) של המעבד ולבצע אותן במקביל.

בתוך הלולאה הפנימית, עבור כל פיקסל (`i, j`), חילצנו את האזור הרלוונטי מהתמונה המרופדת (region) בגודל הקרנל וביצענו כפל איבר-באיבר (Element-wise multiplication) בין האזור לבין הקרנל, וסכמנו את התוצאה (`np.sum`). הערך המתקבל נשמר במטריצת התוצאה.

חלק ב': מימוש correlation_cuda (רצה על ה-GPU)

המימוש מחולק לשתי פונקציות: פונקציית מעטפת ("Host") המכילה את הנתונים, ופונקציית הליבה ("Kernel") הרצה על הכרטיס הגרפי.

1. הפונקציה `correlation_gpu` (ה-Host Code):
פונקציה זו אחראית על ניהול הזיכרון והזנקת התהליכים ב-GPU.

הכנת נתונים והמרת טיפוסים: בדומה למקודם מבצעים ריפוד (Padding) של התמונה.

העברה ל-Device: שימוש ב-`cuda.to_device` כדי להעתיק את התמונה המרופדת ואת הקרנל מזיכרון המחשב (Host) לזיכרון ה-GPU (Device).

חישוב גריד החוטים (Grid & Block Dimensions):

נקבע גודל בלוק של 16×16 תהליכונים (threadsperblock).

מספר הבלוקים (blockspergrid) מחושב כך שיכסה את כל פיקסלי התמונה (גובה ורוחב התמונה חלקי גודל הבלוק, עם עיגול כלפי מעלה).

הפעלת הקרנל: קריאה ל-`correlation_kernel` עם תצורת הגריד והפרמטרים הדרושים.

קבלת התוצאה: העתקת התוצאה חזרה מה-GPU ל-CPU באמצעות `copy_to_host`.

2. הפונקציה `correlation_kernel` (ה-Device Code):
זוהי הפונקציה שרצה במקביל על אלפי תהליכונים ב-GPU (דקורטור `cuda.jit@`).

זיהוי הפיקסל (`cuda.grid`): כל תהליכון ("Thread") מזהה על איזה פיקסל (i, j) הוא אחראי באמצעות `cuda.grid(2)`. זהו מיפוי ישיר בין מיקום התהליכון בגריד לבין הקואורדינטות בתמונה.

בדיקת גבולות (Boundary Check): מכיוון שמספר הבלוקים הוא כפולה של 16, ייתכנו תהליכונים שחורגים מגודל התמונה האמיתי. התנאי `if i < i_h and j < i_w` מבטיח שרק תהליכונים רלוונטיים יבצעו חישוב.

חישוב הקורלציה:

כל תהליכון מבצע לולאה כפולה קטנה על מימדי ה-`kernel (ki, kj)`.

הוא ניגש לפיקסלים המתאימים בתמונה המרופדת (`padded_image`) ומבצע כפל-סכום (Multiply-Accumulate) עם ערכי הקרנל.

מכיוון שכל תהליכון כותב לתא זיכרון נפרד במטריצת התוצאה (`result[i, j]`), אין צורך במנעולים או סנכרון מורכב (Race Condition) בכתיבה.

```

(tf23-gpu) tal-ben@lambda:~/hw2_cdp$ srun --gres=gpu:1 -c 1 --pty python3 filters_test.py
CPU 3X3 kernel: 0.0017057880759239197
Numba 3X3 kernel: 0.0015553124248981476
CUDA 3X3 kernel: 0.0012050382792949677
-----
CPU 5X5 kernel: 0.0037094131112098694
Numba 5X5 kernel: 0.0025892816483974457
CUDA 5X5 kernel: 0.0011866390705108643
-----
CPU 7X7 kernel: 0.0071222372353076935
Numba 7X7 kernel: 0.004271987825632095
CUDA 7X7 kernel: 0.0012408047914505005
-----
(tf23-gpu) tal-ben@lambda:~/hw2_cdp$

```

8.

ניתן לראות שיפור בביצועים במעבר מ-CPU ל-Numba, ושיפור נוסף במעבר ל-CUDA.

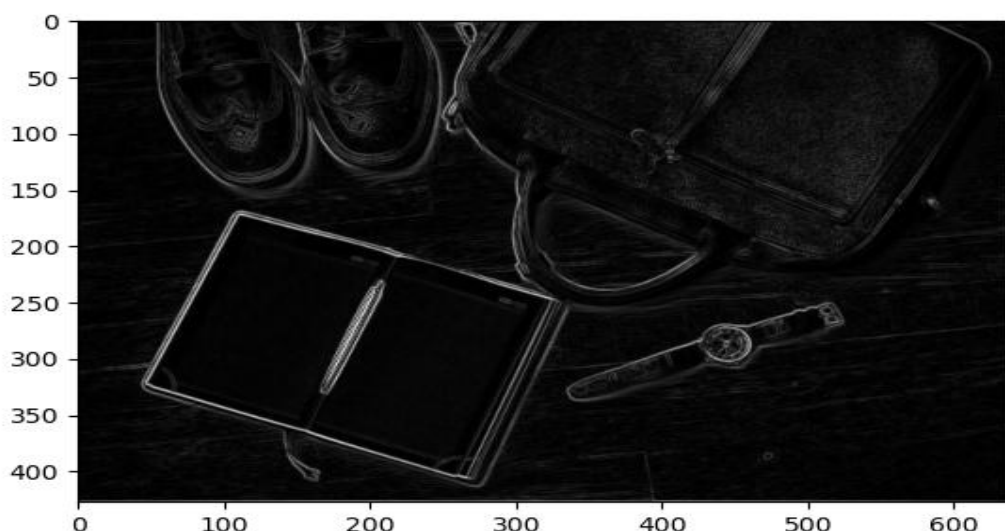
מימוש `scipy`: תהליך אחד שמריץ את הקוד וסובל מ"תקורת המפרש" (Interpreter Overhead) ואינה מנצלת היטב את המעבד לחישובים מקביליים ברמת הפיקסל.

Numba (njit): מהיר יותר מה-CPU הרגיל למרות שגם כן רץ על CPU עם core בודד. המהדר (JIT Compiler) הופך את קוד הפייתון לקוד מכונה אופטימלי בזמן ריצה. זה חוסך את התקורה של המפרש ומאפשר אופטימיזציות של המעבד (כגון וקטוריזציה).

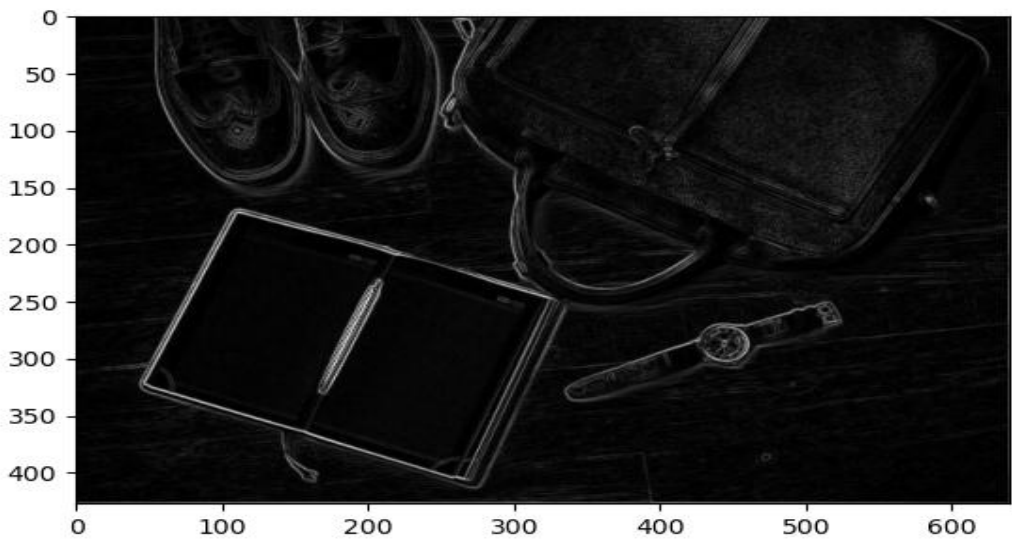
CUDA: המהיר ביותר (במקרה זה). ה-GPU הוא מעבד המיועד לעיבוד מקבילי מסיבי. בפעולת קונבולוציה, כל פיקסל מחושב באופן בלתי תלוי, מה שמאפשר לאלפי ליבות ה-GPU לעבוד במקביל. החישוב כולל את הפעולה הבנויה `fused multiply-add (FMA)` שרץ מאוד מהר על `gpu`.

9. נרצה להשתמש ב `numba` עבור חישוב קורולוציה של `kernels` קטנים ונרצה כמובן להריץ את `numba` עם יותר מעבדים. זה מכיוון שנראה מהתוצאות שככל שה `kernel` יותר קטן הזמן חישוב של `numba` קטן מכיוון שמשימת החישוב קטנה ככל שמקטינים את ה `kernel` אז המעבד מסיים את עבודתו יותר מהר. נשים לב שזמני `cuda` לא משתנים, ככל הנראה זה נובע מכך שמשימות החישוב קטנות מידי כך ש `cuda` הוא `memory bound` עבור הקרנלים הקטנים.

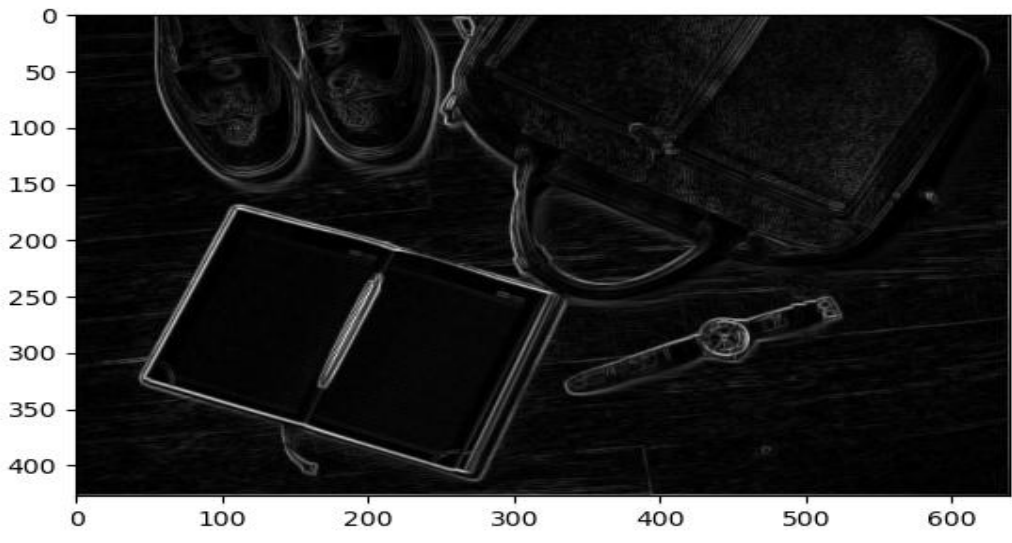
10. `sobel_filter`

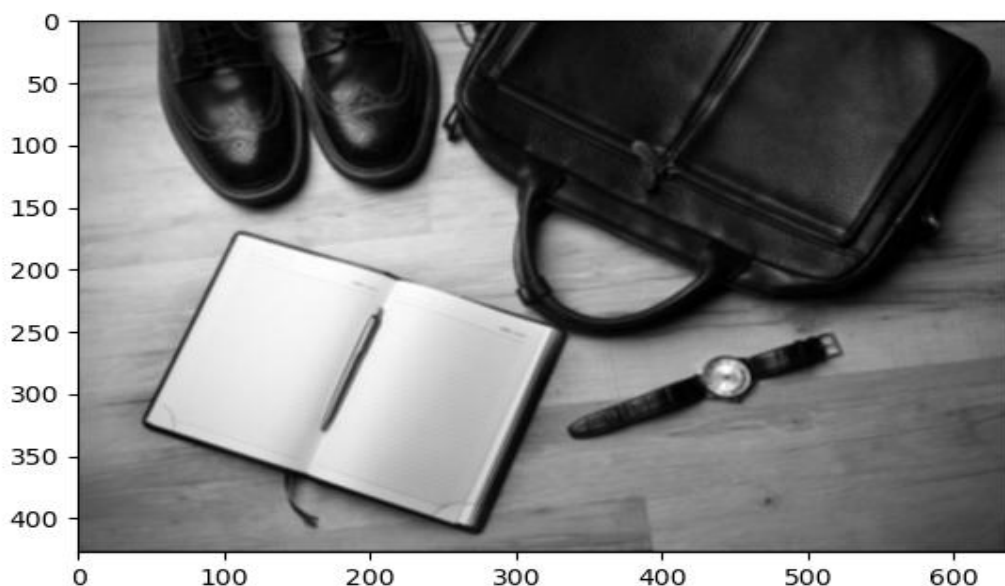


kernel1:



kernel2:





:sobel

התוצאה: זיהוי קצוות סטנדרטי

הסבר: המטריצה מכילה ערכים חיוביים בצד שמאל ושיליים בצד ימין כך שחישוב קורולוציה עם מטריצה זו גורם לקצוות שמאליים בתמונה להיות יותר בהירים וקצוות ימניים להיות יותר כהים.

: Kernel 1

התוצאה: האפקט זהה למקודם רק שהוא יותר חזק יותר (בעל ניגודיות יותר גדולה)

הסבר: המטריצה נותנת משקל גבוה יותר למרכז (10 במקום 2) ולערכים הקיצוניים (3 במקום 1), מה שמגביר את הרגישות לקצוות.

Kernel 2 (מטריצה מוארכת $\times 35$):

התוצאה: זיהוי קצוות "חלק" יותר.

הסבר: המטריצה הגבוהה מבצעת החלקה (Smoothing) על פני 5 פיקסלים אנכיים, מה שמפחית רעשים אנכיים אך שומר על הקצוות הראשיים.

: Kernel 3

התוצאה: תמונה מטושטשת ובהירה (לא זיהוי קצוות).

הסבר: המטריצה מכילה רק ערכים חיוביים. היא מבצעת סכימה/ממוצע של פיקסלים ולא חיסור. לכן, במקום לקבל הפרשים (קצוות), מקבלים אפקט של טשטוש (Blur) בהיר על התמונה המקורית.