**236370 תכנות מקבילי ומבוזר לעיבוד נתונים ולמידת מכונה**

**דוח תרגיל בית 2**

מרינה ינובסקי 324515659

קורן מעברי 207987314

**שאלה 1**

צילומי המסך נמצאים בהמשך. הפעלנו את main.py באמצעות 8, 16 ו-32 יחידות עיבוד (cores) עם הדגל -c <cores>. סיכום המדדים המרכזיים:

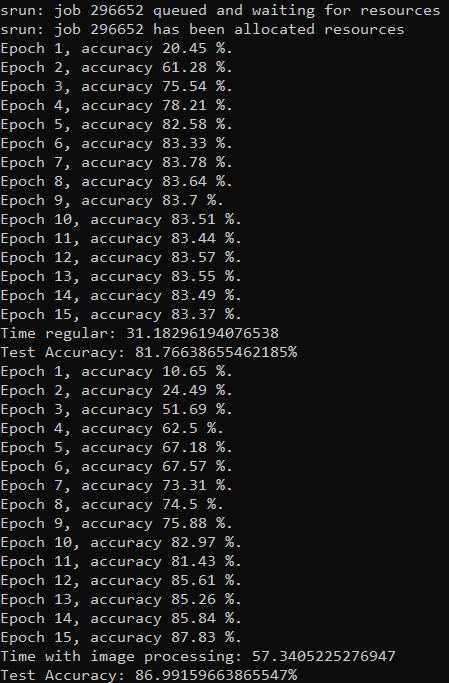
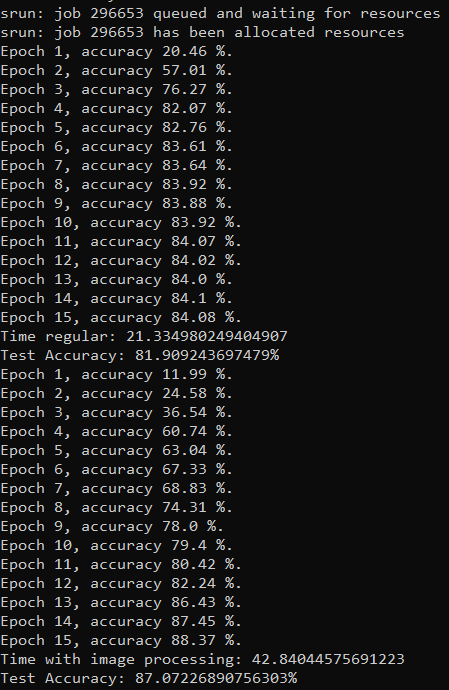
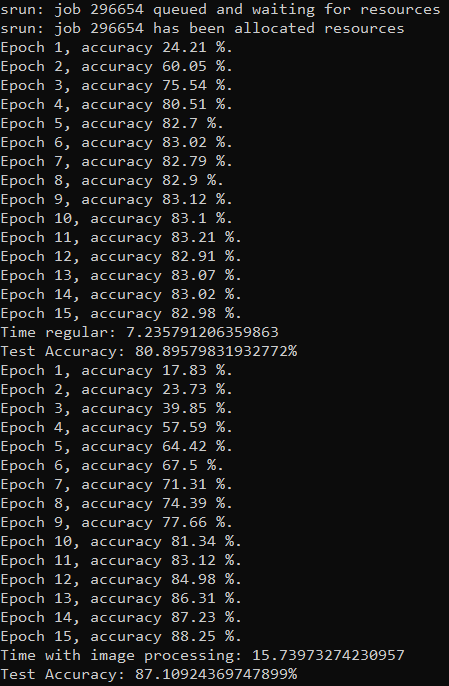
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| כמות ליבות | זמן רגיל (שניות) | דיוק רגיל (%) | זמן עם עיבוד תמונה (שניות) | דיוק עם עיבוד תמונה (%) |
| 8 | 31.18 | 81.77 | 57.34 | 86.99 |
| 16 | 21.33 | 81.91 | 42.84 | 87.07 |
| 32 | 7.23 | 80.90 | 15.74 | 87.11 |

זמן הריצה של שני המימושים, "רגיל" ו־"עם עיבוד תמונה", התקצר ככל שגדלה כמות הליבות, כפי שניתן לראות בטבלה.

תוצאה זו צפויה מכיוון שיותר ליבות מאפשרות עיבוד מקבילי יותר של משימות. בפרט, בעיבוד התמונה, מבוצעים מספר רב של חישובים (יצירת באצ'ים, וביצירת כל באצ', הפעלת כל הפונקציות של האוגמנטציה על כל תמונה) ולכן ככל שאנו מריצים על יותר ליבות זמינות, עומס העבודה מתחלק בין יותר WORKERS (שכמותם ככמות הליבות), דבר שמאיץ את הזמן הדרוש לביצוע. לכן, הרצה עם 32 ליבות נתנה את התוצאות הטובות ביותר.

כמו כן, אנו מקבלים דיוק טוב יותר עם עיבוד התמונה, שכן הוא יוצר דוגמאות חדשות ולכן יכול לבצע הכללה טובה יותר כי המידע עליו התאמן מגוון יותר.

מימין לשמאל: 8 ליבות, 16 ליבות, 32 ליבות.

A screenshot of a computer

Description automatically generated**שאלה 2**

|  |  |  |
| --- | --- | --- |
| אפוק | דיוק NeuralNetwork (%) | דיוק IPNeuralNetwork (%) |
| 1 | 24.21 | 17.83 |
| 2 | 60.05 | 23.73 |
| 3 | 75.54 | 39.85 |
| 4 | 80.51 | 57.59 |
| 5 | 82.7 | 64.42 |
| 6 | 83.02 | 67.5 |
| 7 | 82.79 | 71.31 |
| 8 | 82.9 | 74.39 |
| 9 | 83.12 | 77.66 |
| 10 | 83.1 | 81.34 |
| 11 | 83.21 | 83.12 |
| 12 | 82.91 | 84.98 |
| 13 | 83.07 | 86.31 |
| 14 | 83.02 | 87.21 |
| 15 | 82.98 | 88.25 |

דיוק במבחן של neuralNetwork: 80.89

דיוק במבחן של IpNeuralNetwork: 87.1

ניתן לראות שNEURAL כבר באפוק ה4 מגיע לדיוק של כ80%, בעוד שIP משפר את הדיוק שלו ככל שעולה כמות האפוקים עליהם הוא רץ.

IPNeuralNetwork עושה אוגמנטציה על הדוגמאות, ובכך מגדיל את הגיוון בדוגמאות, דבר שיוצר יותר גיוון בנתוני האימון. האימון איטי יותר, אך במבחן ניתן לעשות הכללה טובה יותר. לעומת זאת, NeuralNetwork מאמן מהר יותר והוא מתאמן על דוגמאות פחות מגוונות ולכן הדיוק שלו מעט נמוך יותר.

**שאלה 3**

מספר העובדים עבור ה־IPNeuralNetwork נקבע על פי מספר ליבות העיבוד שהוקצו להרצה, כפי שמוגדר במשתנה הסביבה SLURM\_CPUS\_PER\_TASK. זאת כי אנחנו רוצים לנצל את כל הליבות הנתונות לנו בצורה המיטבית ולהקצות לכל ליבה עובד, וכך לחלק את עומס העבודה בין יותר עובדים.

אם היינו בוחרים יותר עובדים מכמות הליבות המוקצות, הייתה לנו תקורה של החלפות הקשר בין כל העובדים ותקורה של scheduling.

אם היינו בוחרים בפחות עובדים מכמות הליבות המוקצות, לא היינו מבצאים utilization למשאבים הנתונים (כלומר לא מנצלים את כל החומרה שלרשותינו, כי לחלק מהCORES לא היה מוקצה עובד והם היו IDLE). דבר זה היה מוביל לכך שעומס העבודה מתחלק בצורה לא שוויונית בין החומרה, וזמן הביצוע היה גדל לעומת זמן הביצוע הפוטנציאלי אליו ניתן להגיע עם חלוקה טובה יותר (כפי שבפועל ביצענו).

לכן, השתמשנו במשתנה הסביבה הזה, שכן ייתכן והעבודה תוקצה בסביבות שונות על כמות ליבות שונה, ונרצה לנצל זאת בצורה המיטבית ולהימנע מהבעיות שתיארנו.

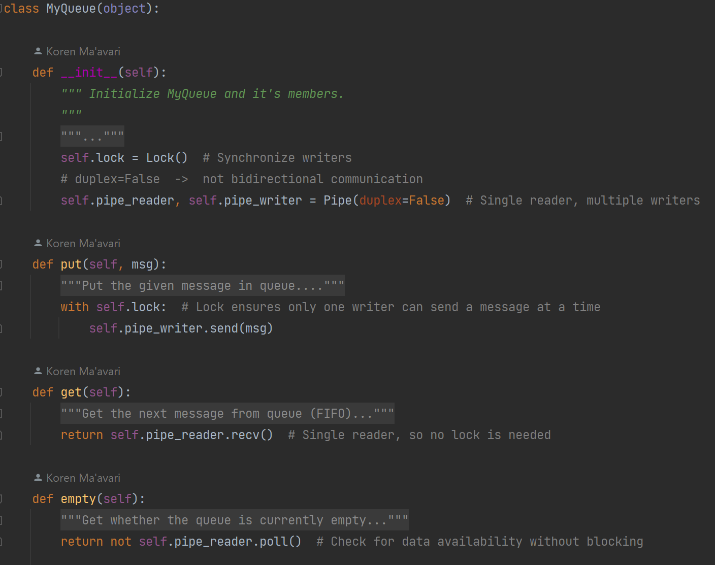
**שאלה 4**

עשינו שימוש בתהליכים נפרדים ולא בחוטים בתוך אותו תהליך. זאת מכיוון שכפי שראינו בתרגול 2, בפייתון קיים מנגנון GIL שמונע מחוטים של אותו תהליך להתבצע במקביל (שוב, כפי שראינו, יש מנעול על ה interpreter שקובע שהרצת bytecode של פייתון דורשת את נעילתו). בעוד שהייתרון לכך הוא מניעת דדלוק, דבר זה יוביל בפועל להרצה סדרתית של הקוד (אפקטיבית הקוד יהיה single threaded). מכיוון שנרצה לקבל האצה אמיתית בחישוב, ולחלק עומס עבודה, נשתמש בתהליכים שיכולים לרוץ במקביל ועל מספר ליבות משתנה, ולא כפופים למגבלה של GIL.

**שאלה 5**

האצה על ידי ביצוע functional decomposition: היינו יכולים לחלק את עיבוד התמונה למשל על פני תהליכים שונים כך שכל תהליך אחראי על ביצוע פעולה ספציפית בעיבוד, כאשר יש pipeline של משימות. כך עיבוד התמונות היה מסתיים מהר יותר, והיינו מקבלים גם נצילות גבוהה יותר בתהליכים, שבסיום כל חישוב של תמונה הם היו מתקדמים ישר לחישוב על התמונה הבאה.

האצה על ידי שימוש בGPU: הGPU יכול להריץ אלפי חוטים, ובכך לאפשר ביצוע מקבילי יותר שיאיץ עוד את החישוב. כרגע כל האימון ועיבודי התמונות מבוצעים על הCPU. בפרט שלבי האימון וBackpropagation, הכוללים כפל מטריצות והפעלת פונקציות, יכולים להיות מואצים באופן משמעותי על ידי שימוש ב־GPU שיכול לעשות הרבה מהפעולות הנל באופן מקבילי כפי שראינו בהרצאה. כלומר,אם היינו משתמשים בGPU היינו יכולים להאיץ עוד יותר את החישובים המקביליים ואת עיבודי התמונות.

**שאלה 6**

Pipe(duplex=False) יוצר ערוץ תקשורת חד־כיווני עם שני קצוות:

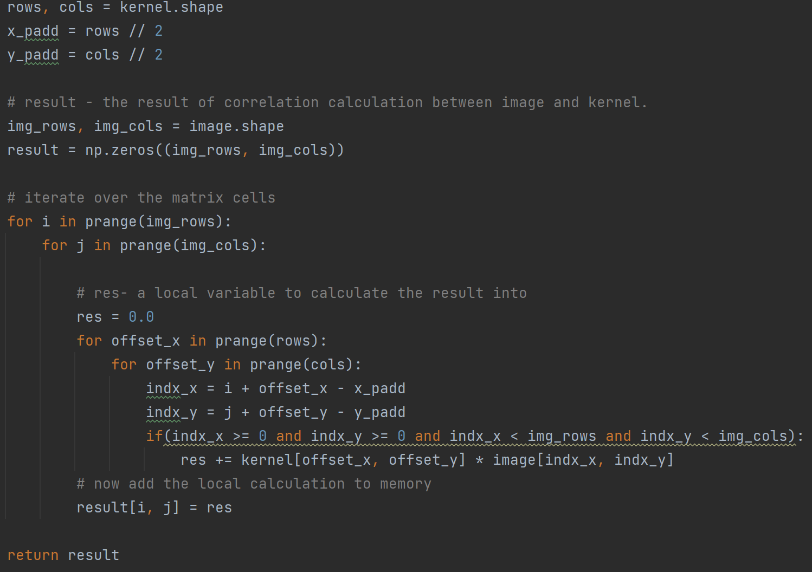
• pipe\_reader: משמש לקריאת הודעות מהתור

• pipe\_writer: משמש לכתיבת הודעות לתור

• ערך duplex נקבע ל־False מכיוון שהתקשורת חד־כיוונית (אין צורך בתקשורת דו־כיוונית)

Lock משמש לסנכרון כותבים, ומבטיח שרק כותב אחד יוכל לשלוח הודעה בכל רגע.

המימוש היה משתנה אם היו קוראים מרובים משום שהיה צורך לסנכרן בין קוראים מרובים כדי להבטיח שלא יקראו את אותה הודעה בו־זמנית (כלומר שכל הודעה תימסר לקורא אחד בלבד). בנוסף, היה נדרש שימוש ב־Lock או מנגנון סנכרון אחר לניהול הגישה ל־pipe\_reader.

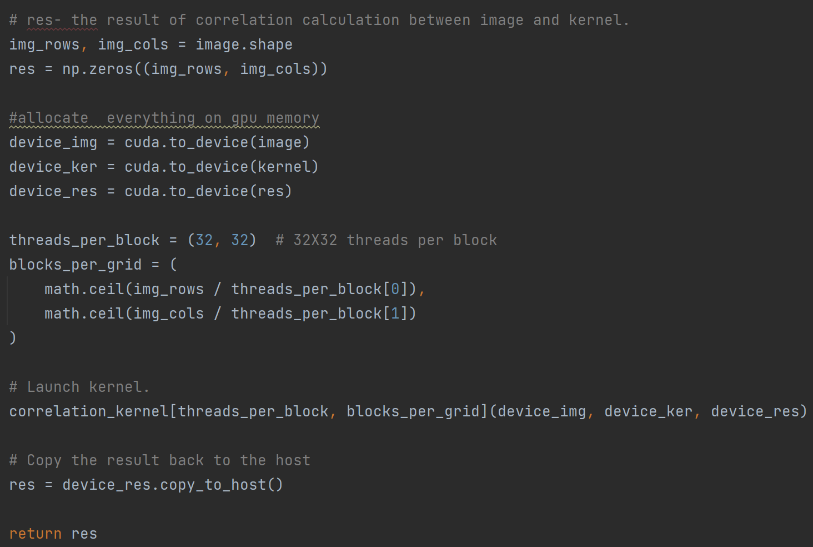
**שאלה 7**

מימוש עם NUMBA:

1. נשמור את מימדי התמונה והקרנל, וכן את הPADDING שמייצג את הגבולות של הקרנל יחסית למרכזו.
2. כעת, באיטרציות על מימדי התמונה (שהם מימדי הפלט), כך שi,j הינו התא שמייצג את הפיקסל עליו עובדים (פלט):
   1. נגדיר משתנה לוקלי אליו נשמור את החישוב כדי להפחית כתיבות לזיכרון
   2. באיטרציות על מימדי הקרנל: נוודא שהאינדקס איתו מחשבים מכפלה בתמונה חוקי. אם כן- נוסיף את המכפלה שתתקבל לסכימה.

* האינדקס בתמונה איתו מחשבים מכפלה = אינדקס התא של התוצאה + padding + האינדקס המתאים של התא בקרנל איתו נכפיל.
* דילוג על הכפלה של ערכים לא חוקיים, שקול להכפלה ב0 והוספה לתוצאה, ולכן אין טעם לבצע חישוב זה.

בנוסף, בחרנו לבצע איטרציות עם prange כדי שניתן יהיה לבצע חישוב מקבילי בהוספת דגל parallel=true.

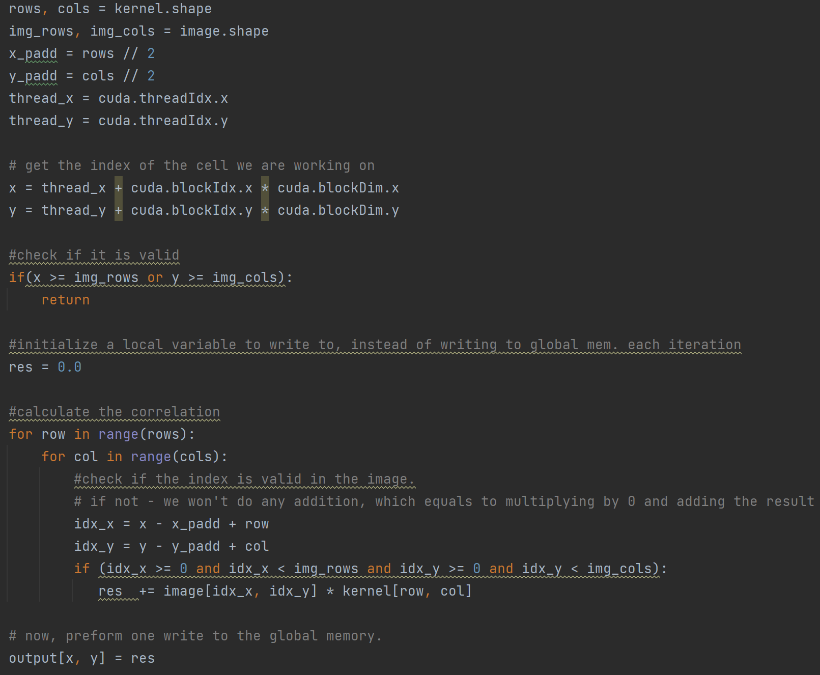
מימוש עם CUDA:

נעתיק לdevice את הפרמטרים המתאימים, כולל את המטריצה של הפלט.

מכיוון שאין הנחה על גודל התמונה, נרצה להקצות את הthreads ואת הבלוקים בהתאם. נגדיר גודל דו מימדי מגודל לכל היותר של 32\*32 כמימד החוטים בבלוק (סה״כ לכל היותר 1024 חוטים, כפולה של 32 שהוא גודל של warp וכן מביא לנצילות של חומרה כי עדיף להריץ כמה שיותר חוטים בבלוק, במקרה שלנו- כמות מקסימלית, כדי לנצל אותו באופן המיטבי). הגדרנו באופן דו מימדי מטעמי נוחות בחישוב (ניתן לקבל ערכי x ו-y בחוטים עליהם נעשה חישוב לאינדקסים).

כעת נגדיר גם כן כמות בלוקים בגודל דו מימדי. נרצה שהכמות הזו תתאים בדיוק לחלוקה של התמונה לתתי בלוקים בגודל 32\*32, כדי לחשב את התוצאה עבור כל התאים בתמונה.

נקרא לקרנל עם המימדים שהגדרנו, ונחזיר את התוצאה שהוא חישב לאחר שהעתקנו אותה חזרה לhost.

מימוש הקרנל:

נשמור את מימדי הקלטים, וכן את הpadding ששימושו הוסבר במימוש הקודם.

נחשב את המיקום המוחלט של תא הפלט לפי מיקום החוט בבלוק ולפי מיקום הבלוק בgrid.

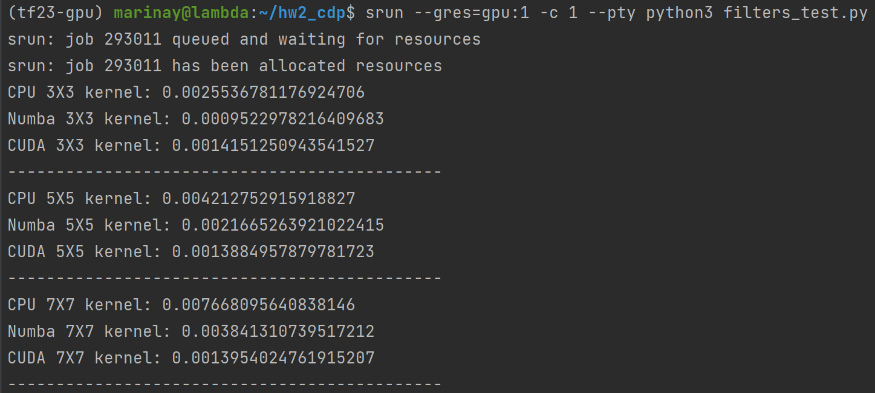
נוודא שהמיקום הנ"ל לא חורג מגודל המימדים המותר.

נאתחל משתנה מקומי אליו נכתוב תוצאות ביניים, וכך נחסוך גישות לזיכרון.

כעת, באיטרציות על מימדי kernel: נוודא שהאינדקס איתו מחשבים מכפלה בתמונה חוקי. אם כן- נוסיף את המכפלה שתתקבל לסכימה

הערות באופן דומה למימוש הקודם:

* האינדקס בתמונה איתו מחשבים מכפלה = אינדקס התא של התוצאה + padding + האינדקס המתאים של התא בקרנל איתו נכפיל.
* דילוג על הכפלה של ערכים לא חוקיים, שקול להכפלה ב0 והוספה לתוצאה, ולכן אין טעם לבצע חישוב זה.

**שאלה 8**

התמונה עליה מתבצע החישוב בגודל 215\*215.

תחילה, הCPU מבצע הרצה רגילה של קוד פייתון: כל שורה מתורגמת ע"י הinterpreter, ומורצת. יהיו 215\*215\* (size\_of\_kernel) איטרציות סדרתיות. לכן ניתן לראות בבירור שזמן הביצוע על הCPU בכל ההרצות הוא הכי איטי, מכיוון שהוא מבצע חישובים רבים באופן סדרתי . לכן נתמקד בהשוואה בין הGPU לNUMBA שמבצעים הרצה מקבילית, דבר שישפר את זמן ההרצה ביחס לCPU.

עבור הקרנל בגודל 3\*3:

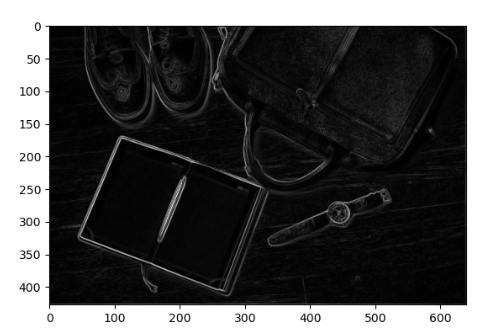
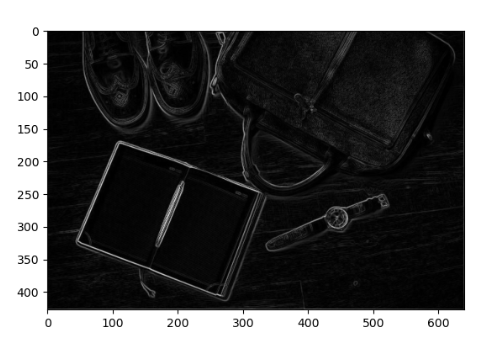
ניתן לראות כי NUMBA, שמבצע הרצה מקבילית על הCPU מהיר יותר מאשר הרצה על הGPU. זאת מכיוון שיש תקורה על העתקת כל המידע לGPU וממנו, וההעתקה זו לא משתלמת ביחס לכמות החישובים הנדרשים לביצוע. בנוסף באיטרציה הראשונה הקוד עובר קמפול (JIT), דבר שמאיץ את החישוב באיטרציות הבאות ולכן נקבל ביצוע מהיר יותר ביחס לCPU הסדרתי האיטי.

עבור שאר הקרנלים: הGPU מהיר יותר מכיוון שכל החישובים בו מתבצעים בצורה מקבילית על כל הCORES שלו, לעומת הרצה מקבילית על הCPU בה ניתן להשתמש בכמות מוגבלת של CORES עליהם מריצים במקביל (הרצה שתתאפשר עם דגל parallel=true עם NUMBA). לפיכך, למרות שאנחנו משלמים מחיר על העתקת המידע לGPU וממנו, אנחנו מקבלים שיפור בביצועים כי יש חישובים רבים שנדרש לבצע, והחישובים מבוצעים במקביליות רבה (החישוב הינו compute bound ולכן הGPU נותן שיפור בביצועים).

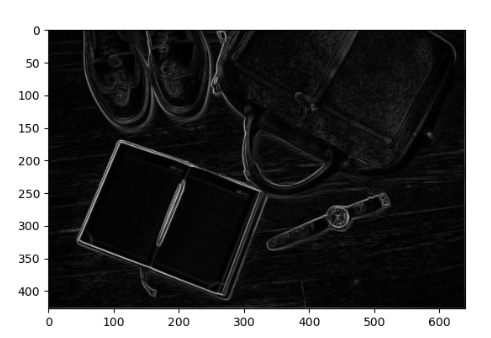
**שאלה 9**

נעדיף להשתמש בnjit כאשר יש מעט חישובים ביחס להעתקת המידע שיש לבצע לGPU ובחזרה (memory bound), ונשתמש בcuda כאשר יש חישובים רבים שיש לבצע, ואנחנו נרוויח מהמקבול על הGPU ולא נפסיד על ההעתקה (compute bound).

**שאלה 10**

הרצה עם הקרנל הנתון: הרצה עם קרנל 1:

A notebook and a watch on a wooden floor

Description automatically generatedהרצה עם קרנל 2: הרצה עם קרנל 3:

**קרנל 1:**

מבנה הפילטר: הפילטר מוגבר על ידי משקלים, כאשר השורה המרכזית מכילה מקדמים גדולים יותר (10 ו 10-) כדי להדגיש את הציר האופקי.

השפעה על התוצאות: קצוות אופקיים מזוהים בצורה בולטת יותר, במיוחד במקומות שבהם יש שינויים חזקים בעוצמה, וקצוות עם הבדלים עדינים מועצמים בשל המשקלים החזקים.

**קרנל 2:**

מבנה הפילטר: הפילטר הזה מאריך את הגובה (המימד האנכי) וכך הוא מכסה יותר שורות.

השפעה על התוצאות: קצוות אנכיים מזוהים יותר, והוא רגיש למעברים על פני מספר שורות במקום שינויים חדים ומקומיים.

**קרנל 3:**

מבנה הפילטר: הפילטר ממצע אחיד עם דגש קל על הפיקסל המרכזי.

השפעה על התוצאות: הפילטר הזה לא מזהה קצוות אלא מבצע סוג של טשטוש או החלקה, מה שגורם לתמונה להיראות רכה יותר. זה קורה כי הוא מחשב ממוצע של עוצמות הפיקסלים, מה שמפחית רעשים ומטשטש את הקצוות.

**הפילטר המקורי:**

מבנה הפילטר: מזהה קצוות על ידי הדגשת גרדיאנטים בכיוונים אופקיים ואנכיים.

השפעה על התוצאות: מפיק תוצאה מאוזנת לקצוות גם בכיוון האופקי וגם בכיוון האנכי.