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

**דוח תרגיל בית 2**

מרינה ינובסקי 324515659

קורן מעברי 207987314

**שאלה 1**

**ניתוח והסבר**

**סיכום התוצאות**

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

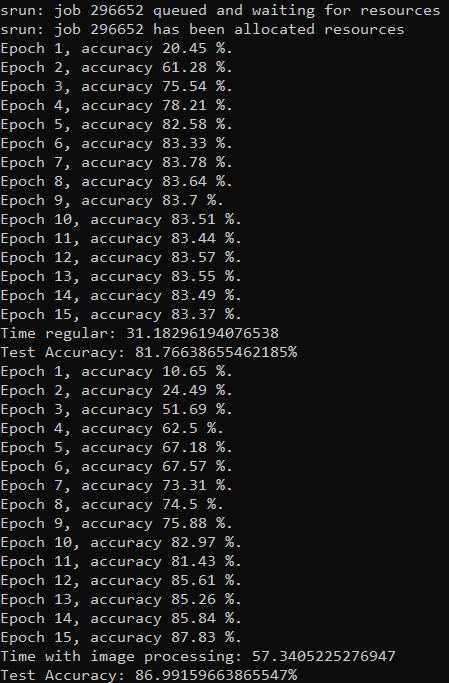
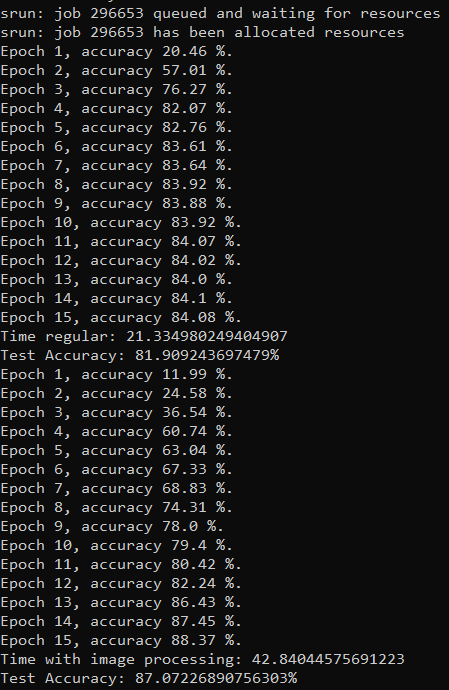
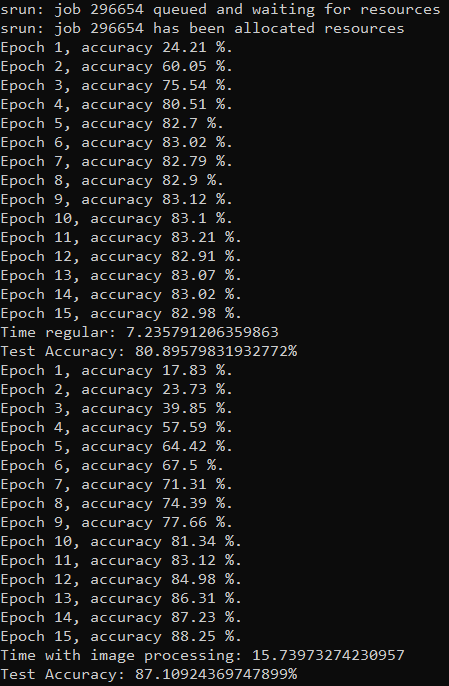
| **כמות ליבות** | **זמן רגיל (שניות)** | **דיוק רגיל (%)** | **זמן עם עיבוד תמונה (שניות)** | **דיוק עם עיבוד תמונה (%)** |
| --- | --- | --- | --- | --- |
| **8** | 31.18 | 81.77 | 57.34 | 86.99 |
| **16** | 21.33 | 81.91 | 42.84 | 87.07 |
| **32** | 7.23 | 80.90 | 15.74 | 87.11 |

**הסבר התוצאות**

1. **השוואת זמני ריצה**
   * זמן הריצה של שני המימושים, "רגיל" ו־"עם עיבוד תמונה", התקצר בצורה משמעותית ככל שגדלה כמות הליבות:
     + עם **8 ליבות**, המימוש "עם עיבוד תמונה" לקח **57.34 שניות**.
     + עם **16 ליבות**, הזמן ירד ל־**42.84 שניות**.
     + עם **32 ליבות**, הזמן ירד עוד יותר ל־**15.74 שניות**.
   * תוצאה זו צפויה מכיוון שיותר ליבות מאפשרות עיבוד מקבילי של משימות, כגון **הרחבת נתונים** (data augmentation) או **עיבוד מיני-אצוות** (mini-batch processing), שהן משימות כבדות חישובית.
2. **השוואת דיוק**
   * הדיוק נשאר יציב בכל הקונפיגורציות עבור המימוש "עם עיבוד תמונה" (כ־87%) ללא קשר לכמות הליבות.
   * עבור המימוש "רגיל" (ללא עיבוד תמונה), הדיוק ירד מעט ככל שכמות הליבות גדלה:
     + **8 ליבות: 81.77%**
     + **16 ליבות: 81.91%**
     + **32 ליבות: 80.90%**
   * ירידה זו בדיוק יכולה לנבוע מהבדלים בסדר האימון או בהתנהגות ההתכנסות (convergence) כתוצאה מהאופן שבו המשימות חולקו בין הליבות.
3. **הקונפיגורציה הטובה ביותר**
   * קונפיגורציית **32 ליבות** נתנה את זמן הריצה הטוב ביותר תוך שמירה על הדיוק הגבוה ביותר עבור המימוש "עם עיבוד תמונה" (**87.11%** בזמן של **15.74 שניות** בלבד).
   * מימוש "עם עיבוד תמונה" מפיק תועלת רבה יותר מעיבוד מקבילי מכיוון שהטרנספורמציות (כגון **סיבוב** (rotate), **הזזה** (shift), **הוספת רעש** (add\_noise)) עצמאיות לכל תמונה וניתנות לחלוקה בין הליבות.
4. **מדוע הביצועים משתפרים עם יותר ליבות?**
   * מימוש IPNeuralNetwork משתמש ב־multiprocessing לעיבוד והרחבת תמונות. כל ליבה מטפלת בתת־קבוצה של משימות (כגון הרחבת אצוות תמונות), ולכן זמן הריצה מתקצר משמעותית כשמספר הליבות גדל.
   * עבור מספר קטן של ליבות, המשימות ממתינות בתור ומעובדות בסדר עוקב, מה שיוצר צווארי בקבוק.

**צילומי מסך**

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

**שאלה 2**

**השוואה בין NeuralNetwork ל־IPNeuralNetwork לאורך אפוקים**

**סיכום תוצאות:**

| **אפוק** | **דיוק NeuralNetwork (%)** | **דיוק IPNeuralNetwork (%)** |
| --- | --- | --- |
| 1 | 20.99 | 19.39 |
| 2 | 63.46 | 20.28 |
| 3 | 71.3 | 42.94 |
| 4 | 79.81 | 61.15 |
| 5 | 81.33 | 66.68 |
| 6 | 83.09 | 65.49 |
| 7 | 83.34 | 70.17 |
| 8 | 83.29 | 74.73 |
| 9 | 83.45 | 78.39 |
| 10 | 83.35 | 79.66 |
| 11 | 83.45 | 82.29 |
| 12 | 83.39 | 85.13 |
| 13 | 83.64 | 84.87 |
| 14 | 83.28 | 87.00 |
| 15 | 83.35 | 86.21 |

**תובנות עיקריות:**

1. **דיוק התחלתי (אפוקים 1-3):**
   * המודל **NeuralNetwork** משיג דיוק גבוה יותר באפוקים הראשונים (למשל, 20.99% מול 19.39% באפוק 1).
   * הפער נובע מכך ש־**IPNeuralNetwork** מוסיף עיבוד תמונה אקראי (הגדלת נתונים), שמגדיל את השונות בנתוני האימון ומאט את ההתכנסות בהתחלה.
2. **אפוקים אמצעיים (אפוקים 4-10):**
   * באפוק 4, **IPNeuralNetwork** מתחיל לסגור את הפער ועוקף את **NeuralNetwork** באפוק 9.
   * הגדלת הנתונים ב־**IPNeuralNetwork** משפרת את יכולת ההכללה, ומאפשרת למודל ללמוד תכונות יציבות ועמידות מהנתונים.
3. **אפוקים מאוחרים (אפוקים 11-15):**
   * באפוק 15, **IPNeuralNetwork** מגיע לדיוק גבוה משמעותית (87.00% מול 83.35%).
   * האסטרטגיות להגדלת נתונים, כגון סיבוב, רעש ועיוותים, מאפשרות ל־**IPNeuralNetwork** להתמודד טוב יותר עם שונות בנתונים לא נראים, וכך לשפר את הביצועים.
4. **דיוק במבחן:**
   * דיוק המבחן של **NeuralNetwork**: **81.55%**
   * דיוק המבחן של **IPNeuralNetwork**: **84.56%**
   * הפער בדיוק המבחן מדגיש את היתרון של שימוש בהגדלת נתונים לשיפור ההכללה.
5. **זמן ביצוע:**
   * זמן האימון עם **IPNeuralNetwork** ארוך יותר (56.87 שניות) בהשוואה ל־**NeuralNetwork** (29.21 שניות), בשל התקורה הנוספת הנגרמת על ידי עיבוד הנתונים שמבוצע על ידי העובדים.

**הסבר:**

**IPNeuralNetwork** משתמש בטכניקות להגדלת נתונים כגון סיבוב, הזזה והוספת רעש, שמגדילות את הגיוון בנתוני האימון. בעוד שטכניקות אלו מאטות את האימון, הן מובילות להכללה טובה יותר ולדיוק גבוה יותר באפוקים המאוחרים. לעומת זאת, **NeuralNetwork** מאמן מהר יותר אך אינו מנצל נתונים מוגדלים, ולכן הדיוק שלו בנתונים לא נראים מעט נמוך יותר.

**שאלה 3**

**בסיס לבחירת מספר העובדים:**

מספר העובדים עבור ה־IPNeuralNetwork נקבע על פי מספר ליבות העיבוד הזמינות, כפי שמוגדר במשתנה הסביבה SLURM\_CPUS\_PER\_TASK. אם משתנה זה אינו זמין, ברירת המחדל היא מספר הליבות הכולל במכונה (os.cpu\_count()).

**רציונל:**

1. **מקסום השימוש במשאבים**:
   * המטרה היא לנצל את כל הליבות הזמינות לעיבוד מקבילי באופן יעיל.
   * כל עובד מתאים לליבה אחת, ומבטיח שהעומס יחולק באופן שווה בלי להשאיר ליבות שאינן פעילות.
2. **הימנעות מעומס יתר**:
   * הקצאת יותר מדי עובדים עלולה להוביל לעומס יתר, כמו מעברי הקשר תכופים, תחרות על זיכרון וקונפליקטים בתזמון, מה שעלול לפגוע בביצועים.
3. **התאמה למבנה של ריבוי תהליכים (Multiprocessing)**:
   * מכיוון שהעובדים מבצעים משימות עצמאיות (למשל עיבוד תמונות), הגיוני להקצות עובד לכל ליבה, כך שהמערכת תתאים למבנה העבודה המקבילי.

**השפעות של בחירת מספר עובדים גדול מדי או קטן מדי:**

**יותר מדי עובדים:**

1. **ירידה בביצועים**:
   * העובדים יתחרו על אותם משאבי CPU, מה שיגרום למעברי הקשר תכופים ולעומס על מתזמן המערכת.
   * הדבר עלול להאריך את זמני הביצוע של המשימות ולהגביר את צריכת הזיכרון.
2. **תחרות על משאבים**:
   * אם המערכת סובלת ממגבלות זיכרון, מספר רב מדי של עובדים יכול לגרום להחלפת זיכרון (Memory Swapping), מה שמוביל לפגיעה נוספת בביצועים.
3. **תשואה פוחתת**:
   * מעבר למספר הליבות הפיזיות, עובדים נוספים לא מספקים יתרון מקבילי משמעותי, מכיוון שהזמן מתחלק בין יותר תהליכים מהיכולת של החומרה.
4. **הגברת השהיה**:
   * תחרות בתור המשותף בין מספר רב מדי של עובדים יכולה להוביל לעיכובים בגישה למבנים משותפים כמו JoinableQueue.

**פחות מדי עובדים:**

1. **תת־ניצול של משאבים**:
   * אם מספר העובדים נמוך משמעותית ממספר הליבות, חלק מליבות המעבד יישארו לא פעילות, וכך פוטנציאל הביצועים לא ממומש.
2. **זמני עיבוד ארוכים יותר**:
   * קבוצה קטנה של עובדים תעבד אצוות (Batches) באופן סדרתי או עם מקביליות מינימלית, מה שיוביל להארכת זמן הביצוע הכולל.
3. **חלוקת עומסים לא מאוזנת**:
   * עם מספר נמוך של עובדים, העומס על כל עובד גדל, מה שעלול לגרום לעובד אחד להפוך לצוואר בקבוק.

**גישה אידיאלית:**

היישום שבחרנו קובע באופן דינמי את מספר העובדים האידיאלי בהתבסס על הסביבה (SLURM\_CPUS\_PER\_TASK) או על החומרה (os.cpu\_count()):

* **מבטיח מקביליות מקסימלית** על ידי התאמת מספר העובדים למספר הליבות הפיזיות.
* **מצמצם עומס יתר** על ידי מניעת מצב שבו יש יותר עובדים מהיכולת של המערכת.
* **פשטות בפריסה (deployment)** בסביבות שונות, מכיוון שהמספר מתעדכן אוטומטית.

גישה זו מאוזנת היטב בין יתרונות המקביליות לבין מניעת עומסים, ומבטיחה שימוש יעיל במשאבים תוך התאמה לתשתיות חומרה שונות.

**שאלה 4**

ב**חלק א'** נעשה שימוש בתהליכים נפרדים ולא בחוטים (threads) בתוך אותו תהליך. הבחירה נעשתה בעיקר כדי ליישם את פרדיגמת ה־multiprocessing ולמקסם את הביצועים המקבילים.

**סיבות לשימוש בתהליכים נפרדים:**

1. **Global Interpreter Lock (GIL):**
   * ה־GIL בגרסת Python מונע מחוטים מרובים להריץ קוד של פייתון במקביל בתוך אותו תהליך.
   * שימוש בחוטים בפייתון לא ינצל בצורה מלאה את כל ליבות המעבד, מכיוון שה־GIL יגרום לריצת קוד פייתון בצורה סדרתית, גם אם מדובר באפליקציה מרובת חוטים.
2. **ביצועים מקבילים אמיתיים:**
   * באמצעות תהליכים נפרדים, כל תהליך מקבל מפרש (interpreter) פייתון ומרחב זיכרון משלו.
   * גישה זו מאפשרת לתהליכים לפעול באופן עצמאי על ליבות מעבד מרובות, מה שמבטיח ניצול מלא של מעבדים מרובי ליבות מודרניים.
3. **עומס חישובי כבד:**
   * משימות כמו עיבוד תמונה (למשל סיבוב, הזזה, הוספת רעש) דורשות חישובים אינטנסיביים. משימות אלו נהנות באופן משמעותי מהרצה מקבילית על פני תהליכים נפרדים.
4. **בידוד זיכרון:**
   * לכל תהליך מרחב זיכרון נפרד, מה שמקטין את הסיכון למרוצי נתונים או השפעות לא מכוונות שנובעות משיתוף זיכרון (כפי שקורה בחוטים).
   * זה הופך את המימוש לפשוט ובטוח יותר, מכיוון שמנגנוני סנכרון (כגון מנעולים) נדרשים רק עבור משאבים משותפים כמו תורים, ולא עבור הזיכרון הכללי.
5. **תמיכת ספריית multiprocessing:**
   * ספריית multiprocessing של פייתון מספקת אבסטרקציות ברמה גבוהה, כגון Process, Queue ו־JoinableQueue, מה שמקל על מימוש מקביליות באמצעות תהליכים נפרדים.
   * אבסטרקציות אלו מסייעות בניהול תקשורת בין תהליכים וסנכרון בצורה יעילה.

**השוואה לחוטים (Threads):**

* **חוטים:**
  + משתפים זיכרון בתוך אותו תהליך, דבר שדורש מנגנוני סנכרון נוספים לגישה בטוחה במקביל.
  + מוגבלים על ידי ה־GIL במשימות שמצריכות שימוש כבד במעבד, ולכן לא מנצלים את המעבד בצורה אופטימלית.
* **תהליכים:**
  + מרחבי זיכרון נפרדים מפחיתים את הצורך בסנכרון מורכב.
  + מאפשרים ניצול של ליבות מעבד מרובות בצורה אפקטיבית, ללא מגבלות ה־GIL.

**מסקנה:**

בחלק א', נעשה שימוש בתהליכים נפרדים מכיוון שהם מאפשרים ביצוע מקביל אמיתי של משימות עיבוד התמונות, מנצלים בצורה מלאה את ליבות המעבד, ונמנעים מהמגבלות של ה־GIL של פייתון. גישה זו מתאימה במיוחד למשימות שמצריכות שימוש כבד במעבד כמו עיבוד תמונות, והיא נתמכת ישירות על ידי ספריית multiprocessing, מה שהופך אותה לבחירה אידיאלית עבור תרחיש זה.

**שאלה 5**

**1. שימוש בהאצת GPU**

**הסבר:**

היישום הנוכחי של IPNeuralNetwork משתמש בריבוי תהליכים מבוססי CPU לצורך הגדלת נתונים. למרות שזה יעיל, שלבי האימון והחישוב לאחור (Backpropagation), הכוללים כפל מטריצות והפעלת פונקציות, יכולים להיות מואצים באופן משמעותי על ידי שימוש ב־**GPU**. ספריות כמו **NumPy** או **SciPy** אינן מותאמות לחישובי GPU, אך ספריות כמו **PyTorch** או **TensorFlow** כן. העברת תהליך האימון לשימוש ב־GPU תאיץ את:

* ההעברה קדימה (כפלי מטריצות, פונקציות אקטיבציה).
* החישוב לאחור (חישוב נגזרות).
* עדכון המשקלים (פעולות על איברי המטריצות).

**מימוש:**

1. החלף את פעולות המטריצה במחלקות NeuralNetwork ו־IPNeuralNetwork בפעולות מתוך ספריות תומכות GPU כמו PyTorch או TensorFlow.
2. העבר את נתוני האימון לזיכרון ה־GPU (לדוגמה, באמצעות torch.Tensor.to('cuda')).
3. ודא שהגדלת הנתונים (Data Augmentation) נשארת על ה־CPU כדי להימנע מעומס מיותר על משאבי ה־GPU.

**היגיון:**

מעבדי GPU מותאמים לחישובים מקביליים בקנה מידה גדול ויכולים להתמודד עם חישובי כפל מטריצות כבדים הנדרשים במהלך האימון הרבה יותר מהר ממעבדי CPU.

**2. שימוש בצינורות (pipelines) עיבוד נתונים עם ביצוע אסינכרוני**

**הסבר:**

כרגע, התהליכים בקובץ preprocessor.py מעבדים תמונות בצורה סדרתית בכל תהליך. למרות שזה מקבילי בין תהליכים שונים, הגדלת הנתונים עדיין מהווה צוואר בקבוק, במיוחד עבור מערכי נתונים גדולים או הגדלות נתונים מורכבות. הכנסת **צינורות אסינכרוניים** תאפשר עיבוד מקביל של נתונים ואימון בו־זמנית.

**מימוש:**

1. **צינור נתונים**:
   * השתמשו בספריות כמו **TensorFlow Data API** או **PyTorch DataLoader** כדי ליצור צינור עיבוד נתונים שיכול לאחזר ולהכין נתונים באסינכרוניות.
   * עבדו מנות (batches) נתונים במקביל בזמן שהמודל מתאמן על המנה הנוכחית.
2. **שמירה במטמון בזיכרון**:
   * השתמש בכלים כמו **Joblib** או **Dask** כדי לשמור נתונים שעובדו בזיכרון או על הדיסק (במקרה שהזיכרון מוגבל). כך ניתן להימנע מהגדלה מחדש של אותן תמונות באפוקים עתידיים.
3. **חפיפת הגדלה ואימון**:
   * התחל להגדיל את הנתונים של המנה הבאה בזמן שה־GPU/CPU מאמן את המנה הנוכחית. ניתן להשיג זאת באמצעות תבניות **Producer-Consumer** או ספריות כמו **Ray**.

**היגיון:**

על ידי חפיפת שלבי עיבוד הנתונים והאימון, זמן ההמתנה של המודל יופחת, מה שיוביל לשימוש יעיל יותר במשאבי המחשוב.

**הערות נוספות:**

* **שילוב שני הרעיונות**:
  + השתמש בהאצת GPU לאימון ובצינורות עיבוד נתונים לצורך הגדלת הנתונים. עבד את הנתונים על ה־CPU והעבר את החישובים של האימון ל־GPU.
* **אופטימיזציה של גודל המנה**:
  + נסה גדלי מנות גדולים יותר כדי למקסם את השימוש ב־GPU. מנות גדולות יותר מפחיתות את מספר שלבי עדכון המשקלים, מה שמאיץ את האימון הכולל, בתנאי של־GPU יש מספיק זיכרון.

**סיכום:**

השילוב בין האצת GPU לצינורות עיבוד נתונים אסינכרוניים יכול להפחית באופן דרמטי את הזמן הדרוש לאימון, ולאפשר איטרציות מהירות יותר ויעילות גבוהה יותר.

**שאלה 6**

**הסבר על המימוש שלך עבור חלק ב'**

המימוש שלך של MyQueue נועד לשמש **תור FIFO (ראשון נכנס, ראשון יוצא)** תוך שימוש ב־**multiprocessing.Pipe** לתקשורת בין תהליכים וב־**Lock** לסנכרון כותבים מרובים. להלן הסבר מעמיק על כל רכיב ומתודה:

**1. אתחול (\_\_init\_\_)**

* 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.

**2. מתודת put**

* משתמשת ב־self.lock לסנכרון גישה ל־pipe\_writer. זה מבטיח שרק כותב אחד ישלח הודעה בכל רגע, ומונע פגיעות בנתונים.
* קוראת ל־pipe\_writer.send(msg) כדי להכניס את ההודעה לתור.

**3. מתודת get**

* קוראת ל־pipe\_reader.recv() כדי לשלוף את ההודעה הבאה.
* אין צורך ב־Lock מכיוון שהמטלה מבטיחה שקיים רק קורא אחד.

**4. מתודת empty**

* קוראת ל־pipe\_reader.poll() כדי לבדוק אם יש הודעות זמינות לקריאה.
* מחזירה True אם התור ריק, אחרת False.

**האם המימוש היה שונה ללא ההנחה של "קורא אחד לכל היותר"?**

כן, המימוש היה משתנה באופן משמעותי אם היו **קוראים מרובים**. הנה למה ואיך המימוש היה משתנה:

**1. סנכרון לקוראים**

* היה צורך לסנכרן בין קוראים מרובים כדי להבטיח שלא יקראו את אותה הודעה בו־זמנית.
* היה נדרש שימוש ב־Lock או מנגנון סנכרון אחר (למשל Condition או Semaphore) לניהול הגישה ל־pipe\_reader.

**2. הבטחת מסירה של הודעות**

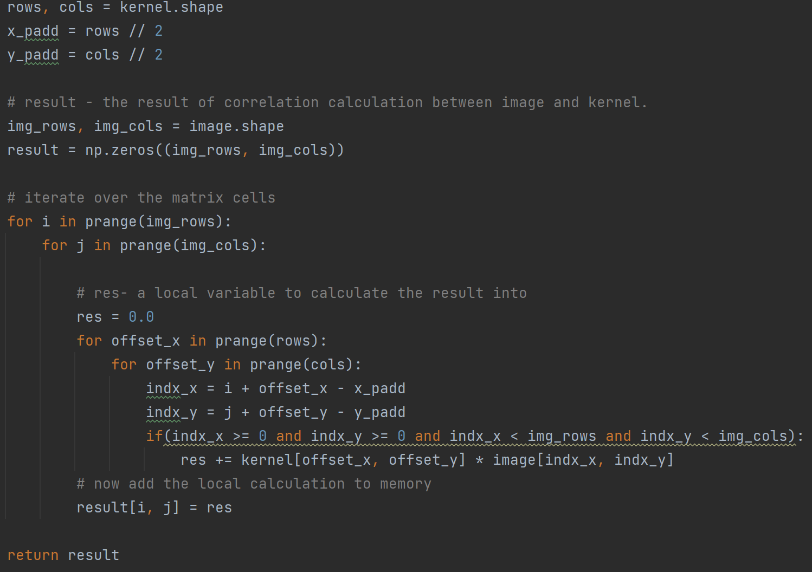
* הודעות עשויות להיקרא על ידי מספר קוראים אלא אם ייעשה שימוש במנגנונים נוספים (למשל מימוש של Queue) שיבטיחו שכל הודעה תימסר לקורא אחד בלבד.

**3. שימוש ב־Pipe**

* Pipe לבדו לא יספיק מכיוון שהוא אינו תומך באופן מובנה בקוראים מרובים.
* מנגנון חלופי כמו multiprocessing.Queue, שתומך ביצרנים וצרכנים מרובים, יהיה מתאים יותר.

**סיכום השינויים ללא ההנחה של "קורא אחד":**

1. **סנכרון לקוראים:** שימוש במנגנון להבטחת אבטחת שרשור לקוראים מרובים.
2. **מנגנון תור:** החלפת Pipe ב־Queue לתמיכה מובנית בקוראים וכתבים מרובים.
3. **השפעה על הביצועים:** שימוש ב־Queue עשוי להוסיף מעט תקורה לעומת Pipe, אך מפשט את הסנכרון עבור קוראים מרובים.

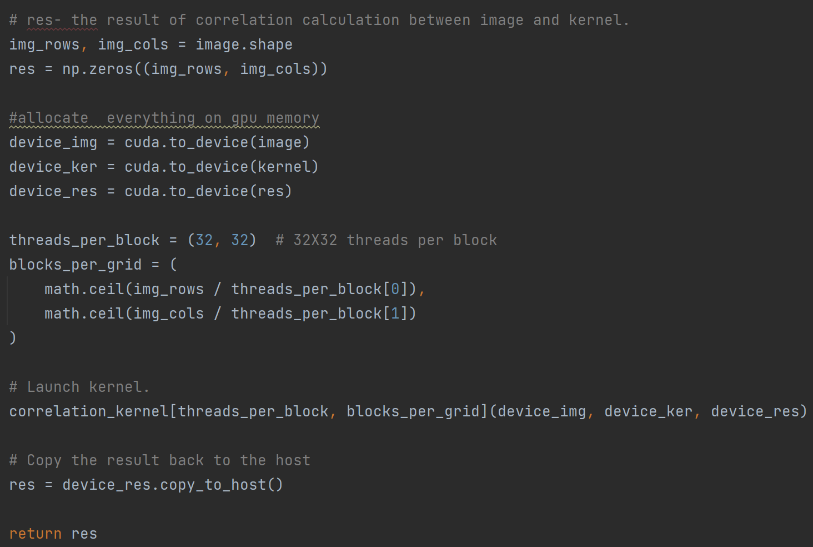
**שאלה 7**

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.

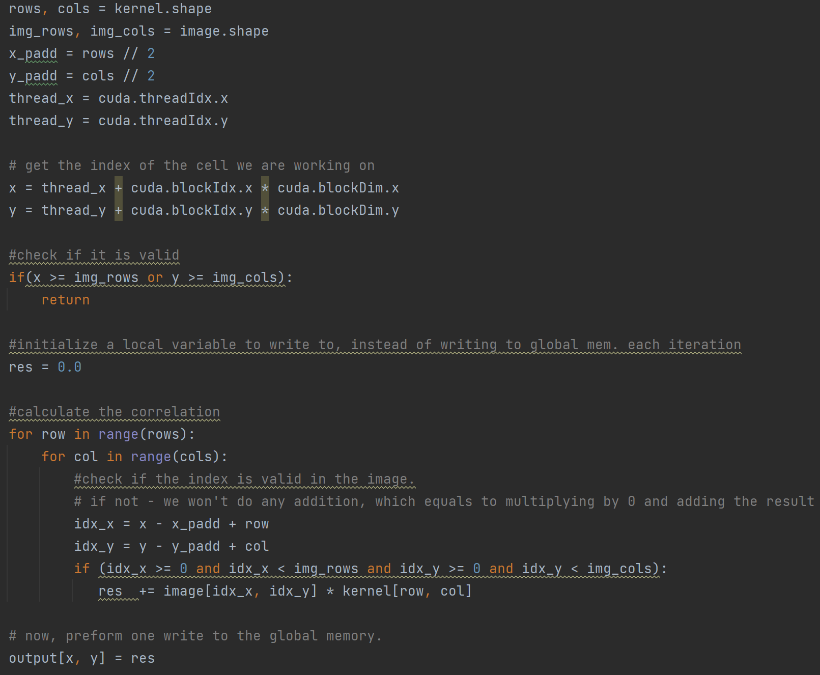
מימוש עם CUDA:

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.

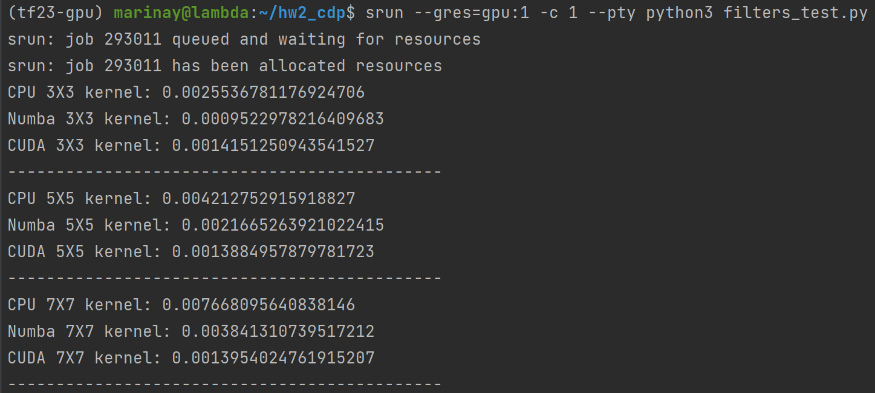
נוודא שהמיקום הנ"ל לא חורג מגודל המימדים המותר.

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:

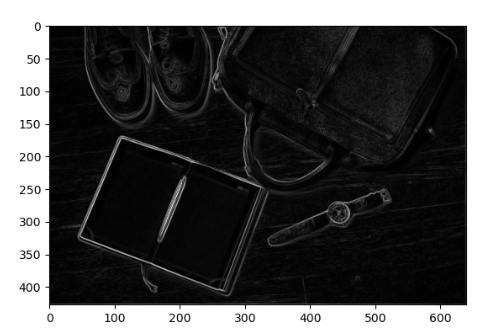
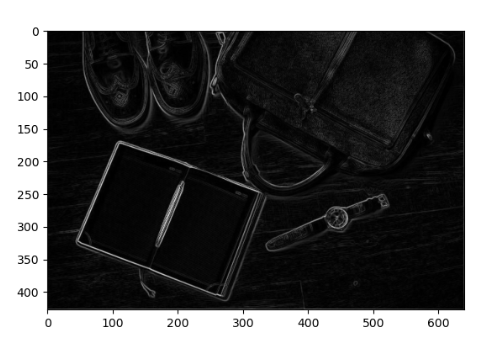
ניתן לראות כי NUMBA, שמבצע הרצה מקבילית על הCPU מהיר יותר מאשר הרצה על הGPU. זאת מכיוון שיש תקורה על העתקת כל המידע לGPU וממנו, וההעתקה זו לא משתלמת ביחס לכמות החישובים הנדרשים לביצוע. בנוסף באיטרציה הראשונה הקוד עובר קמפול (JIT), דבר שמאיץ את החישוב באיטרציות הבאות ולכן נקבל ביצוע מהיר יותר ביחס לCPU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

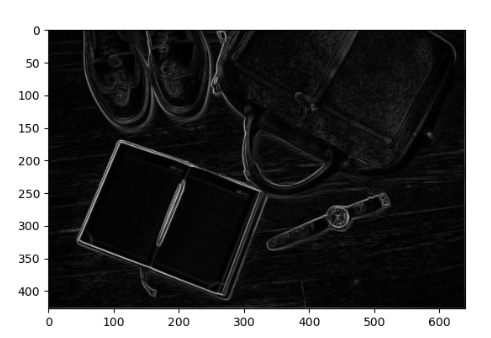
**שאלה 9**

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.