

## **236370 תכנות מקבילי וUMBRELLA לעיבוד**

### **נתונים ולמידת מכונה**

**דוח תרגיל בית 1**

**מариינה ינובסקי 324515659**

**קורן מועברי 207987314**

## חלק 1:

תוצאות האימון, עם דיקט סופי גובה מ% 95% כנדרש:

```
Epoch 1, accuracy 92.37 %.  
Epoch 2, accuracy 95.11 %.  
Epoch 3, accuracy 95.88 %.  
Epoch 4, accuracy 96.62 %.  
Epoch 5, accuracy 97.17 %.  
Time to train using np.matmul: 17.583032369613647 seconds  
Test Accuracy: 96.73%
```

## חלק 2:

### שימוש max\_gpu - הסבר:

נקזה מטריצת פלט.

מבצע העתקה של הפרמטרים הדרושים ל-device.

נקרא לKERNEL: נפעילו עם 1000 בלוקים, כאשר בכל בלוק 1000 חוטים. הפרמטרים לKERNEL אלו הפרמטרים שהעתקנו לזכרון device בשלב הקודם.

בסיום עבודת הkernel, נעתיק את התוצאה שנכתבה לאחד הפרמטרים בזיכרון של הdevice חזרה לזכרון host, כדי שהhost יוכל לגשת אליה ולקרוא אותה.

### שימוש הkernel - הסבר:

נתון שגודל מטריצות הקלט הוא  $1000 \times 1000$ . ישנו 1000 בלוקים, ובכל אחד מהם 1000 חוטים שמריצים את הkernel.

לכן ניתן לסתכל על זה בצורה הבאה: לחישוב של כל שורה, יהיה אחראי בלוק אחד. לחישוב תא בתוור שורה, יהיה אחראי חוט בבלוק. סך הכל - בלוק x בgrid מתאים לשורה הx, וחוט y בבלוק מתאים לעמודה הy, כלומר על ביצוע מקסימום בין אינדקסים y,x במטריצות A,B אחראי החוט הx בבלוק x.

נקבל את האינדקס של הבלוק ע"י הפקודה `cuda.blockIdx.x` (אנו משתמשים בגריד בעל מיד אחד ולכן מעניין אותנו הערך x. בלבד).

נקבל את האינדקס של החוט בבלוק ע"י הפקודה `cuda.threadIdx.x` (אנו משתמשים בבלוקים בעלי מיד אחד ולכן מעניין אותנו הערך x. בלבד).

סך הכל - נכתב במטריצת הפלט במיקום הרצוי המקסימום בין האינדקסים המתאים, לפי ההסבירים לעיל.

```
def max_gpu(A, B):  
    """  
    Returns  
    -----  
    np.array  
        element-wise maximum between A and B  
    """  
  
#A and B are given matrixes of size(1000,1000) with integer values of range [0,255]  
# Define grid and block size  
threads_per_block = 1000  
blocks = 1000  
  
C = np.zeros_like(A)  
  
# Allocate device memory  
device_A = cuda.to_device(A)  
device_B = cuda.to_device(B)  
device_C = cuda.to_device(C)  
  
# Launch kernel  
max_kernel[blocks, threads_per_block](device_A, device_B, device_C)  
  
# Copy the result back to the host  
C = device_C.copy_to_host()  
  
return C  
  
@cuda.jit  
def max_kernel(A, B, C):  
    """  
    CUDA kernel for calculating the element-wise maximum of two matrices.  
    Parameters:  
        A (cuda.device_array): First input matrix (device array).  
        B (cuda.device_array): Second input matrix (device array).  
        C (cuda.device_array): Output matrix to store the element-wise maximum.  
    """  
  
    tx = cuda.threadIdx.x # Thread ID within a block  
    bx = cuda.blockIdx.x # Block ID within the grid  
    # We have 1000*1000 matrix.  
    # We have total of 1000 * 1000 threads (1000 blocks, 1000 threads in a block).  
    # Meaning: blockIdx will represent the matrix row, threadIdx will represent the matrix column.  
  
    C[bx, tx] = max(A[bx, tx], B[bx, tx])
```

```
[+] max_cpu passed
[+] max_numba passed
[+] max_gpu passed
[+] All tests passed

==Job was executed on 1 cores==
[*] CPU: 12.072056457400322
[*] Numba: 0.028749946504831314
[*] CUDA: 0.2776772454380989
speed-up between the CPU runtime to the GPU run time: 43.47515761292868
----Time Comparison:----
max_gpu/max_numba = 9.658356943079403
max_gpu/max_cpu = 0.02300165232145508
max_numba/max_cpu = 0.002381528499827984
```

הרצה על core אחד:

אכן ניתן לראות שאנו מקבלים האצה של פי 40 לפחות בגין זמן הביצוע סדרתי של הCPU לעומת זמן הביצוע מקביל של GPU.

מוספקים בצלום המספר כמו כן חישובים של הקטועה בין הריצות השונות, כנדרש.

הסבר של התוצאות:

הឧמן ביצוע הריצה רגילה של קוד פ'יתון: כלומר כל שורה מתוגמת על ידי interpreter, ומורצת. במימוש הנאיי שלנו, יהיו  $1000^*1000$  איטרציות סדרתיות, דבר שיקח זמן רב.

הឧמן ביצוע הריצה במקביל, הוא יבצע את הפעולה באופן מקבילי, וכך יתנו תוצאות טובות יותר ביחס לCPU. התוכנית בCPU היא `id-bound compute`, ומבצעת את אותה הפעולה הרבה פעמים. לכן, המקובל (בפרט מקובל) שמאפשר ביצוע מספר כה רב של פעולות במקביל) נותן תוצאה טובה יותר מאשר הריצה על CPU.

הריצה עם `numba`: מכיוון שהקוד עובר באיטרציה ה-1 קומפלול (DO), בשאר האיטרציות הוא יירוץ מהר יותר בהשוואה לinterpreter ולהרצה CPU הרגילה. בנוסף, ישנה תקורת העברת מידע GPU וחזרה ממנו, ותקורת ניהול הניהול שלו, ולכן הריצה זו תהיה מהירה יותר גם במקרה בהשוואה לGPU.

הריצה על כמה מוגנות של `cores`:

```
==Job was executed on 2 cores, number of numba threads: 2==
[*] CPU: 11.679155480116606
[*] Numba: 0.03070126101374626
[*] CUDA: 0.2820624113082886
speed-up between the CPU runtime to the GPU run time: 41.40628106363142
----Time Comparison:----
max_gpu/max_numba = 9.187323321410062
max_gpu/max_cpu = 0.024150925277815757
max_numba/max_cpu = 0.00262872269082676473
```

נשים לב כי ההבדל המהותי הוא עבור הריצה לגרסה של `numba`. הסיבה להבדלים בהרצות אלו לכמות `cores` הנקוד `cores` היא שהקוד מ被执行 ב-`parallel loop`, כלומר יש כמה תהליכיים שמריצים את הקוד שבולאה בצדקה מקבiliar, דבר שמספק שיפור ביצועים ככל שיש יותר `cores` שמאפשרים הריצה עליהם (מוספיקים יותר מקובל).

```
==Job was executed on 4 cores, number of numba threads: 4==
[*] CPU: 12.023526832461357
[*] Numba: 0.02035994827747345
[*] CUDA: 0.27193670719861984
speed-up between the CPU runtime to the GPU run time: 44.21443120468284
----Time Comparison:----
max_gpu/max_numba = 13.356453734192177
max_gpu/max_cpu = 0.022617049971098306
max_numba/max_cpu = 0.0016933424411301064
```

עבור גרסה הCPU התוכנית מבוצעת סדרתי, ועבור גרסה CUDA הריצה על GPU ולא מושפעת מהגדלת הילבות. לכן הגדלת כמה מוגנות `cores` לא משפיעה על מהלך הביצוע של התוכנית ברגשות אלן ובפרט גם לא משפיעה על התוצאות שלהן.

```
==Job was executed on 8 cores, number of numba threads: 8==
[*] CPU: 11.956670485436916
[*] Numba: 0.014528211206197739
[*] CUDA: 0.27241046726703644
speed-up between the CPU runtime to the GPU run time: 43.89211106824366
----Time Comparison:----
max_gpu/max_numba = 18.75044789759293
max_gpu/max_cpu = 0.022783137462793607
max_numba/max_cpu = 0.0012150716392069956
```

### חלק 3:

```
def matmul_kernel(A, C):
    # note: we only have 1024 threads!
    # then - each thread will handle a part of the elements in the result matrix!

    threadIdx = cuda.threadIdx.x
    total_threads = cuda.blockDim.x
    rows, cols = A.shape

    # Compute how many elements each thread should handle
    total_elements = rows * rows # Total number of elements in the result matrix
    elems_per_thread = (total_elements + total_threads - 1) // total_threads # Divide work among threads

    for i in range(elems_per_thread):
        element_id = threadIdx + i * total_threads
        if element_id < total_elements: # Ensure we're within bounds
            # Map element_id to a specific (row, col) in the result matrix
            row = element_id // rows
            col = element_id % rows

            # Compute the value of C[row, col]
            temp = 0
            for k in range(cols):
                temp += A[row, k] * A[col, k]

            C[row, col] = temp
```

ימושה הkernel המטראיצ'ות :matmul\_kernel

אין לנו הנחות כלשהן על גודלי המטריצ'ות שאנו מכפילים, ולכן נרצה שכל thread יבצע כמה עבודה זהה (נרצה לחלק את workload באופן שוויוני). העבודה שתבוצע: הכפלת שורה בעמודה המתאימה לה במטריצה .transpose.

תחליה נחשב את גודל chunk עלינו כל thread יבצע עבודה (גודל chunk הינו מספר האלמנטים שנחשב עבור מטריצ'ת התוצאה). גודלו יחושב על ידי כמה אלמנטים הכולת במטריצ'ת התוצאה הרצiosa (כמה איברי המטריצ'ה) לחילק במספר threads.

cut, בולולאה על גודל chunk:

נחשב את המיקום היחסי עליו מעוניינים לבצע את העבודה. נזודא בפרט שהמיקום הנ"ל לא חורג מגבולות הבעה (שכן תיתכן שארית לחישוב).

לאחר מכן, נבצע הכפלה של וקטור השורה בוקטור העמודה המתאים במטריצ'ת transpose. הערך המתתקבל הוא התוצאה הרצiosa לאלמנט אחד בchunk. נשמר את התוצאה במטריצ'ת הפלט, אליו יכתבו כל החוטים במקביל. נציין כי אין צורך בביטוי סכירותון, שכן אמןם כל החוטים כתובים לאוטומאטיות (בזיכרונו), אך האינדקסים אליו הם החוטים כתובים הם "חודים".

ניתוח התוצאות:

```
[+] matmul_transpose_trivial passed
[+] matmul_transpose_numba passed
[+] matmul_transpose_gpu passed
[+] All tests passed
```

Numpy: 0.44993598386645317

Numba: 4.352094490081072

CUDA: 5.778981540352106

לספריה numpy- להכפלת המטריצ'ות הספריה משתמשת בקוד לו בוצעו אופטימיזציות רבות עבור הפעולה.

השוואה בין ביצועי numba לביוצעי -cuda-

- שניהם מרים קוד בצורה מקבילית.
- Cuda מרים קוד על GPU, ודורש העתקה של הקלט לdevice ושל הפלט מהdevice, דבר שדורש זמן ומשפיע על המדיידות.
- GPU לא מנצל בצורה המיטבית וכן ה שימוש בו לא משללים:
  - גודל הבעיה הנבדקת לא מספיק גדול כדי לנצל את חומרת GPU.
  - אם מרים את הקernel רק עם בלוק אחד ו-1024 חוטים, דבר שגורם לרבות cores של GPU להיות במצב idle ולא מנצלים. אם נרצה לשפר ביצועים- נצטרך להגדיל את סך החוטים הכלול שMRIIZ את הבעיה כדי לבצע יותר utilization GPU.
  - תקורה גבוהה של העתקת המידע בגין לרווח מההרצה המקבילית על GPU.

לפיך, הרצה מקבילית על GPU שמספקת numba, בשילוב עם הקומפיילר של CUDA, מספקים לנו ביצועים טובים יותר 😊