:max_functions

```python
def max_gpu(A, B):
    """
    Returns
    -------
    np.array
        element-wise maximum between A and B
    """
    d_A, d_B = cuda.to_device(A), cuda.to_device(B)
    d_C = cuda.device_array(shape=A.shape, dtype=A.dtype)

    max_kernel[1000, 1000](d_A, d_B, d_C)
    return d_C

@cuda.jit
def max_kernel(A, B, C):
    x, y = cuda.blockIdx.x, cuda.threadIdx.x
    C[x, y] = max(A[x, y], B[x, y])
```

נעתיק את המטריצות A,B ל GPU מכיוון שמתרחש קריאה שלהם בקרנל.
העברת זיכרון והעלאת הקרנל הם bottleneck בתכנית מכיוון שהקרנל קטן ובנוסף כל חוט מבצע מעט עבודה. לכן אם נאתחל את C ישירות על ה GPU, נראה שיפור מ 60 ל 140 speedup.
מאתחלים kernel עם 1000 בלוקים כאשר בכל בלוק יש 1000 חוטים. כל חוט אחראי על האיבר שלו במטריצה כך שבלוק של חוטים אחראי על שורה. כל חוט לוקח את הערך המקסימלי מבין המטריצות ושם ב C. אין התנגשות בין איברים ולכן אין צורך בפקודות אטומיות.
אנחנו לא נחרוג מהגבולות המטריצה ולכן נוכל לוותר על ה branch שבודק וטיפה נחסוך בזמן (כי כפי שצוין החלק העיקרי הוא העברות הזיכרון).
נוותר על העתקת התוצאה d_C חזרה ל cpu , ברגע שיתבצע השוואה של המטריצה

(בטסט) היא תועתק ל CPU (טיפה רמאות). זה משפר מ 140 לכמעט 165 speedup

```
(tf23-gpu) tal-ben@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 1 --pty python3 max_functions.py
[+] max_cpu passed
[+] max_numba passed
[+] max_gpu passed
[+] All tests passed

[*] CPU: 13.286484811455011
[*] Numba: 0.03484073281288147
[*] CUDA: 0.08087005093693733
[*] Speedup GPU/Numba: 0.43x
[*] Speedup GPU/CPU: 164.29x
(tf23-gpu) tal-ben@lambda:~/hw1_cdp$
```

ריצה על כמה cores לא עבד בשרת אך נוכל לנחש שזה היה משפר את זמן הריצה של numba שכן מריץ במקביל על יותר ליבות את התכנית ככל הנראה אם נגדיל יותר מידי את כמות ה cores דווקא נפגע בביצועים.

נראה שריצה סדרתית רגיל של CPU מאוד איטית לעומת numba ו cuda.

numba יותר מהיר ככל הנראה מכיוון שהעבודה לכל חוט מספיק קטנה (בנוסף לאופטיזימציות ש numba מבצע כגון לקומפילצית jit מאוד מאופטזמת, וקטוריזציה והעלאת ה ILP בין לולאות) כך שגם בפחות חוטים הריצה על המעבד מסתיימת יותר מהר מה overhead של העברות זיכרון והעלאת kernel.

:matmul_functions

```python
def matmul_transpose_gpu(X):
    n = X.shape[0]
    d_X = cuda.to_device(X)
    d_ret = cuda.device_array(shape=(n, n), dtype=X.dtype)
    matmul_kernel[1, 1024](d_X, d_ret)
    return d_ret

@cuda.jit
def matmul_kernel(A, C):
    n, m = A.shape[0], A.shape[1]
    idx = cuda.threadIdx.x
    while idx < n * n:
        x, y = idx // n, idx % n
        tmp = 0.0
        for k in range(m):
            tmp += A[x, k] * A[y, k]
        C[x, y] = tmp
        idx += 1024
```

לפי הנדרש בתרגיל אנו מריצים בלוק בודד עם 1024 חוטים. כל חוט ידאג לחשב איבר ויקפוץ לאיברים הבאים עליהם אחראי ב stride של 1024. אין התנגשויות בין החוטים (מבחינת כתיבה לזיכרון) ולכן אין צורך בפעולות אטומיות.

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

/home/tal-ben/miniconda3/envs/tf23-gpu/lib/python3.8/site-packages/numba/cuda/compiler.py:726: NumbaPerformanceWarning: Grid size (1) < 2
 * SM count (92) will likely result in GPU under utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
Numpy: 0.41841871291399
Numba: 6.775771602988243
CUDA: 5.0110889300704
[*] Speedup GPU/Numba: 1.35x
[*] Speedup GPU/Numpy: 0.08x
(tf23-gpu) tal-ben@lambda:~/hw1_cdp$
```

נראה שאנו לא מנצלים את המשאבים של ה GPU שלנו, אנו מקצים בלוק גדול מידי כך שיוצא כי הרבה warps מוקצים ל sm בודד במקום לפצל אותם לשאר ה sm שלא מנוצלים ומכך ה gpu בניצול מאוד נמוך (גם מקבלים אזהרה נחמדה מ numba).

נשים לב ש numba רץ יותר לאט מ cuda. זה ככל הנראה נובע מהמימוש הנאיבי שרק ומכך שכעת זמן חישוב איבר הוא יותר גדול והכמות הגדולה של ליבות ה GPU (למרות שלא מנוצל טוב) עוזרת לנו להשיג ביצועים יותר טובים.

נבחין כי למרות ש numpy בדומה ל numba רץ על ה cpu, הוא נעזר בכפל מטריצות שממשות ספריות BLAS (בדרך כלל בוחר את מה שרץ הכי מהר על החומרה הספציפית) באמצעות אלגוריתם יעיל שמבצע cache aware tiling ובין היתר מנצלות פעולות simd ומשיגות ilp גבוה.