HW1 Report

תכנות מקבילי ומבוזר לעיבוד נתונים
236370

מיכל עוזרי 325719052
גיא סודאי 214300550

חלק 1

```
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 2 --pty python3 main.py
Epoch 1, accuracy 91.81 %.
Epoch 2, accuracy 95.24 %.
Epoch 3, accuracy 96.24 %.
Epoch 4, accuracy 96.62 %.
Epoch 5, accuracy 97.0 %.
Time to train using np.matmul: 23.219115257263184 seconds
Test Accuracy: 96.72%
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$
```

חלק 2

הרצה עם ליבה אחת:

```
michal.ozeri@132.68.39.159:22 - Bitvise xterm - michal.ozeri@lambda: ~/hw1_cdp
(tf23-gpu) michal.ozeri@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: 12.886113958898932
[*] Numba: 0.03010775800794363
[*] CUDA: 0.08389257686212659
```

הסבר על מימוש max_gpu ו-max_kernel:

בפונקציה max_gpu שולחים את 2 מערכי הקלט- A ו-B לgpu. יוצרים מערך פלט C ישירות על הgpu כדי לחסוך זמן. שכן יצירתו על הcpu הייתה גוררת את העברתו לgpu ללא צורך- אין בו מידע קודם שיש להעביר מהcpu.

לפי הדרישה, נקצה 1000 בלוקים כך שכל בלוק מכיל 1000 חוטים ונריץ את max_kernel עם dev_A, dev_B, dev_C. בחזרה מ-max_kernel נחזיר לcpu את מערך הפלט C נחזיר אותו ונסיים.

בפונקציה max_kernel, מכיוון שמובטח לנו שהמטריצות A ו-B הן ממימדים 1000×1000 ומכיוון שיש לנו 1000 בלוקים ובכל בלוק 1000 חוטים, נגדיר שכל חוט בכל בלוק אחראי לעדכן את התא $C[bx, tx]$ כך ש-bx הוא הבלוק בו החוט נמצא ו-tx זה מזהה החוט בבלוק. בכך נמנע גישות משותפות לתאים ונבטיח נכונות וניצול מקסימלי של משאבים.

הרצה עם כמה ליבות:

```
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 2 --pty python3 max_functions.py
[+] max_cpu passed
[+] max_numba passed
[+] max_gpu passed
[+] All tests passed

[*] CPU: 12.814635635819286
[*] Numba: 0.029517433140426874
[*] CUDA: 0.08398842392489314
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 4 --pty python3 max_functions.py
[+] max_cpu passed
[+] max_numba passed
[+] max_gpu passed
[+] All tests passed

[*] CPU: 12.8831029930152
[*] Numba: 0.022172821685671806
[*] CUDA: 0.08307844772934914
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 8 --pty python3 max_functions.py
[+] max_cpu passed
[+] max_numba passed
[+] max_gpu passed
[+] All tests passed

[*] CPU: 12.748099931050092
[*] Numba: 0.012952613178640604
[*] CUDA: 0.08278299774974585
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$
```

חישוב ה$speedup$:

cores / speedup of _	$\frac{max_gpu}{max_numba}$	$\frac{max_gpu}{max_cpu}$
1	$\frac{0.08389}{0.03010} \approx 2.787$	$\frac{0.08389}{12.886} \approx 6.51E-3 \approx \frac{1}{153}$
2	$\frac{0.08398}{0.02951} \approx 2.846$	$\frac{0.08398}{12.814} \approx 6.55E-3 \approx \frac{1}{152}$
4	$\frac{0.08307}{0.02217} \approx 3.747$	$\frac{0.08307}{12.883} = 6.45E-3 \approx \frac{1}{155}$
8	$\frac{0.08278}{0.01295} \approx 6.392$	$\frac{0.08278}{12.748} \approx 6.49E-3 \approx \frac{1}{154}$

נבחין כי כשהעלנו את מספר ה cores, ריצת max_cpu לא השתנתה. הדבר צפוי כיוון שהפעולה ממומשת באופן סדרתי לחלוטין, לכן לא ניתן למקבל אותה עם מספר cores. כלומר אין ניצול של ה cores הנוספים, והריצה על core יחיד, זמני הריצה זהים.
גם ריצת max_gpu לא השתנתה, זאת כיוון שהגדלת ה - cores ב *cpu* לא תגדיל את כוח החישוב ב gpu. לכן, חישוב המקסימום המתבצע בgpu יפעל באופן זהה. (יש פעולות הקורות גם על המעבד כמו שליחת הנתונים, וקבלתם, אך גם הן לא יכולות להפיק תועלת משמעותית מריבוי cores, מהסברים דומים לmax_cpu).

עד כה, הסברנו שזמני הריצה של max_cpu, max_gpu לא משתנים בהגדלת מספר ה cores ולכן הspeedup נשאר קבוע. הסיבה ל speedup זה היא יש המוני כניסות בלתי תלויות שיש לחשב, וה gpu משתמש ב 1000X1000 רכיבים מקביליים לחשבם (אחד לכל כניסה), ככה שההאצה בחישוב המקבילי משמעותית בהרבה מהתקורה של התקשורת עם ה gpu, והפעולה מהירה יותר ב gpu.

כאשר העלנו את מספר ה cores עבור max_numba, זמן החישוב התקצר. הדבר הגיוני כיוון שבשימוש בnumba עם prange, יהיו לנו יותר cores לחלק ביניהם את עבודת ה range, וכך יותר עבודה נעשתה במקביל.

כיוון שזמן הריצה של max_gpu לא משתנה בהגדלת מספר ה cores, ואילו של max_numba כן, קורה השינוי בspeedup כשמגדילים את מספר ה cores. הסיבה שבאופן כללי numba יותר מהיר מה gpu, היא שתקורת העברת הנתונים לgpu וקבלתם חזרה, לא משתלמת עבור מספר רכיבים מקביליים גדול יותר, ואילו למקבל על ה cpu אשר הזיכרון כבר מוכן,אין את תקורת העברה זו, וה cores מהירים יותר (וגם מעטים יותר) עדיפה.

חלק 3

הסבר על מימוש matmul_transpose_gpu ו-matmul_kernel:

בפונקציה matmul_transpose_gpu שולחים את מערך הקלט- X לgpu. יוצרים מערך פלט C במימדים $rows \times rows$ (נובע מכך שמימדי X הם $rows \times cols$ ומימדי X^T הם $cols \times rows$ לכן מימדי XX^T הם $rows \times rows$) ישירות על הgpu כדי לחסוך זמן. שכן יצירתו על הcpu הייתה גוררת את העברתו לgpu ללא צורך- אין בו מידע קודם שיש להעביר מהcpu.

לפי הדרישה, נקצה בלוק אחד המכיל 1024 חוטים ונריץ את matmul_kernel עם dev_X, dev_C. בחזרה מ-matmul_kernel נחזיר לcpu את מערך הפלט C נחזיר אותו ונסיים.

בפונקציה matmul_kernel, מכיוון שיש לנו 1024 חוטים, נגדיר שכל חוט אחראי לעדכן את התאים $C[i, j]$ כך ש-

$$(i * num_columns_of_C + j) \bmod 1024 = tx$$

$$iff, \quad (i * rows \ + j) \bmod 1024 = tx$$

כך ש-tx זה מזהה החוט בבלוק. כלומר, כל חוט יתחיל בעדכון התא $C[tx // rows, tx\%rows]$ ויעדכון את התאים הבאים בקפיצות של 1024. בכך נמנע גישות משותפות לתאים ונבטיח נכונות וניצול מקסימלי של משאבים (כלומר, חלוקה שווה בקירוב של העומסים).

```
michal.ozeri@132.68.39.159:22 - Bitvise xterm - michal.ozeri@lambda: ~/hw1_cdp
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 1 --pty python3 matmul_functions.py
[+] matmul_transpose_trivial passed
[+] matmul_transpose_numba passed
[+] matmul_transpose_gpu passed
[+] All tests passed

Numpy: 0.4209525021724403
Numba: 5.80656470824033
CUDA: 6.987547372933477
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 2 --pty python3 matmul_functions.py
[+] matmul_transpose_trivial passed
[+] matmul_transpose_numba passed
[+] matmul_transpose_gpu passed
[+] All tests passed

Numpy: 0.4210456367582083
Numba: 5.805208188015968
CUDA: 6.983507165219635
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 4 --pty python3 matmul_functions.py
[+] matmul_transpose_trivial passed
[+] matmul_transpose_numba passed
[+] matmul_transpose_gpu passed
[+] All tests passed

Numpy: 0.2318826108239591
Numba: 3.120263021904975
CUDA: 6.987735772039741
(tf23-gpu) michal.ozeri@lambda:~/hw1_cdp$ srun --gres=gpu:1 -c 8 --pty python3 matmul_functions.py
[+] matmul_transpose_trivial passed
[+] matmul_transpose_numba passed
[+] matmul_transpose_gpu passed
[+] All tests passed

Numpy: 0.1492176498286426
Numba: 1.8676453148946166
CUDA: 6.988251808099449
```

נסביר את התוצאות:

numpy:

ב numpy ממומש אלגוריתם יעיל מאוד לחישוב מכפלת מטריצות (יותר יעיל מהמימוש הטריוויאלי הלוקח $O(n^3)$)

מקומפל מראש, היודע לנצל מספר cores (ממקבל את החישוב).

לכן ריצתו מהירה מאוד, ומשתפרת עם העלת ה cores.

numba:

ב numba ממומש האלגוריתם הטריוויאלי לחישוב מכפלת מטריצות (הלוקח $O(n^3)$), היודע לנצל מספר cores (ממקבל את החישוב עם prange).

אין תקורת העברת נתונים כי המידע הדרוש כבר נמצא בזיכרון המעבד.

לכן ריצתו משתפרת עם העלת ה cores, אך איטית יותר מ numpy בגלל הבדלי מימוש האלגוריתם.

gpu:

ב gpu ממומש האלגוריתם הטריוויאלי לחישוב מכפלת מטריצות (הלוקח $O(n^3)$), ובנוסף לא מסוגל לנצל ריבוי cores ב cpu (רכיב חומרה אחר, והפעולות הכרוכות ב cpu כמו בהעברת המידע לא "נהנות" מריבוי cores).

בנוסף, יש תקורת העברת נתונים כי המידע הדרוש נמצא בזיכרון המעבד, ויש להעבירו לזיכרון ה gpu לצורך החישוב, וכן להחזיר את התוצאה למעבד.

פעולות החישוב עצמה מתבצעת באופן מקבילי על מספר גדול של threads, אך לא בצורה המשתלמת עבור תקורת ה gpu, בהשוואה להרצה מקבילית על ה cpu כמו numba.

לסיכום, ריצתו אינה משתפרת עם העלת ה cores, וכן איטית יותר מ numba בגלל שהאלגוריתם, אך תקורת העברת הנתונים משמעותיות יותר מאופן המיקבול.