תרגיל בית מספר 3 מחשוב מקבילי סמסטר א 2022

בתרגיל יש שילוב של MPI, OpenMP ו- CUDA.

יש להגיש קובץ zip הכולל את כל הקבצים שכתבתם וגם makefile. לקובץ ההרצה של התכנית יש לקרוא histogram.

יש לכתוב תכנית שמחשבת היסטוגרמה (histogram) של ערכים המופיעים בקלט. כל ערך הוא מספר שלם בין 0 ל- 255. במילים אחרות, עבור כל ערך יש לחשב את מספר המופעים שלו בקלט. לדוגמא אם הקלט הוא

10 10 3 10 150 150 180 3 150 10 אז הפלט יהיה

```
3: 2
10: 4
150: 3
180: 1
```

כלומר המספר 3 מופיע בקלט פעמיים, המספר 10 מופיע 4 פעמים וכן הלאה. ניתן להשמיט מהפלט מספרים שאינם מופיעים בקלט.

השורה הראשונה בקלט כוללת מספר N ובהמשך מופיעים N מספרים (בטווח 0-255) המופרדים ע"י whitespace (רווחים, טאבים או newlines).

התוכנית תקרא את הקלט מה- standard input. היא תכתוב את הפלט ל- standard output.

מבנה התוכנית.

יהיו בתוכנית שני תהליכים של MPI. תהליך 0 יקרא את הקלט לתוך מערך של מספרים וישלח חצי מהמערך לתהליך השני. כל תהליך יחשב את ההיסטוגרמה של אחד מחצאי המערך. התהליך השני ישלח את ההיסטוגרמה שחישב לתהליך 0 שימזג אותה עם ההיסטוגרמה שהוא עצמו חישב ויכתוב את הפלט. אם למשל באחת ההיסטוגרמות למספר 7 יש 10 מופעים ובהיסטוגרמה השניה למספר 7 יש 25 מופעים אז בהיסטוגרמה הממוזגת למספר 7 יש 35 מופעים.

תהליך 0 (של MPI) ישתמש ב- OpenMP (בלי CUDA) כדי לחשב את ההיסטוגרמה שלו. התהליך השני של MPI ישתמש גם ב- OpenMP וגם ב- CUDA כדי לחשב את ההיסטוגרמה שלו: בעזרת OpenMP תחושב ההיסטוגרמה של חצי מערכי הקלט

שהוא אחראי עליהם (שהם רבע מהערכים בקלט המקורי) ובעזרת CUDA תחושב

ההיסטוגרמה של החלק הנותר. לאחר חישוב 2 ההיסטוגרמות האלו, התהליך השני ימזג אותן להיסטוגרמה אחת (את זה ניתן לעשות בקוד סדרתי) ואת ההיסטוגרמה הממוזגת ישלח לתהליך 0 (שכאמור ימזג אותה עם ההיסטוגרמה שהוא עצמו חישב. גם

את המיזוג הזה ניתן לעשות באופן סדרתי).

<u>פרטים אודות המימוש</u>

ניתן לייצג היסטוגרמה בעזרת מבנה נתונים פשוט:

int histogram[256]; ואז histogram[0] יהיה מספר המופעים

של המספר 0, histogram[1] יהיה מספר המופעים של המספר 1 וכן הלאה.

<u>CUDA</u>

יש להשתמש במספר בלוקים שבכל אחד מהם יהיו 256 threads.

את מספר הבלוקים יש לחשב כך שיתאים לגודל הקלט באופן שכל thread יטפל בערך אחד שמופיע בקלט.

<u>דרך פשוטה לחשב את ההיסטוגרמה (אל תשתמשו בדרך זו):</u>

ההיסטוגרמה תאוחסן בזיכרון הגלובלי (שכל ה- threads בכל הבלוקים יכולים לגשת אליו) וכל thread יהיה אחראי על קריאת ערך אחד ממערך הקלט ועדכון הכניסה

המתאימה בהיסטוגרמה "הגלובלית".

מאחר וכל ה- threads מעדכנים את ההיסטוגרמה נוצר race condition כשיותר מ- thread אחד מנסה לעדכן את אותה הכניסה בהיסטוגרמה באותו הזמן.

לכן כדי לעדכן כניסה בהיסטוגרמה יהיה צורך להשתמש ב- atomicAdd.

(החתימה: int atomicAdd(int *address, int val). זה מוסיף

באופן אטומי את הערך val לערך שמאוחסן בכתובת address)

אם מספר threads ינסו בו זמנית לעדכן (בעזרת atomicAdd) את אותה כניסה בהיסטוגרמה, העדכונים יעשו זה אחר זה באופן סדרתי ולא בו זמנית. זה מבטיח שהעדכונים יעשו בצורה נכונה אבל מאט את התוכנית. תופעה זו של threads שמנסים באותו זמן לעדכן את אותה כניסה צפויה להתרחש לעיתים קרובות אם

הרבה מאוד threads יגשו לאותה היסטוגרמה.

פתרון טוב יותר (את זה יש לממש)

ניתן לעבוד בשני שלבים:

בשלב ראשון כל בלוק של threads יבנה היסטוגרמה נפרדת שתאוחסן ב- shared memory (נזכיר שלכל בלוק יש shared memory פרטי שכל ה- threads בבלוק (ורק הם) יכולים לגשת אליו. הגישה ל- shared memory מהירה בהרבה מהגישה ל- global memory). גם כאן יהיה צורך להשתמש ב- `atomicAdd` (כי כל ה-threads בבלוק

יעדכנו את ההיסטוגרמה של הבלוק) אבל מאחר ובבלוק אחד יש פחות threads מאשר בכל הבלוקים, מספר המקרים בהם שני threads (או יותר) יגשו בו זמנית לאותה כניסה צפוי להיות קטן יותר.

.

בשלב שני ממזגים את כל ההיסטוגרמות של כל הבלוקים להיסטוגרמה אחת שתשב בזיכרון הגלובלי. (היא צריכה להיות בזיכרון הגלובלי כדי שניתן יהיה להעתיקה

לזיכרון של ה- host. ל- host אין גישה ל- shared memory של ה- GPU) .

כל אחד מהבלוקים יהיה אחראי למזג את ההיטוגרמה "המקומית" שלו להיסטוגרמה

"הגלובלית".

לדוגמא אם כניסה עם אינדקס 7 בהיסטוגרמה המקומית של בלוק מסוים

מכילה את הערך 5 אז אותו בלוק (ליתר דיוק, אחד ה- threads ששייכים לבלוק) יוסיף 5 לכניסה עם אינדקס 7 בהיסטוגרמה הגלובלית.

שימו לב שהשימוש ב- shared memory עשוי לחסוך הרבה גישות לזיכרון הגלובלי.

בהמשך לדוגמא הנ"ל: במקום לגשת לכניסה עם אינדקס 7 בזיכרון הגלובלי חמש

פעמים (ובכל פעם לקדם את המונה באחד) – ניגשים לכניסה פעם בודדת ואז

מקדמים את הערך בחמש.

OpenMP

גם כאן ניתן לחלק את העבודה בין threads. בשלב ראשון כל thread יבנה

היסטוגרמה פרטית ובשלב שני ההיסטוגרמות הפרטיות ימוזגו להיסטוגרמה אחת.

כל thread יכול לעדכן את היסטוגרמה המאוחדת בהתאם לערכים שמופיעים בהיסטוגרמה הפרטית שלו.

יש להגן על כל עדכון של ההיסטוגרמה המאוחדת ע"י שימוש ב-

`#pragma omp critical` או ב- `#pragma omp atomic`

אבל אז הגישות להיסטוגרמה יעשו באופן סדרתי.

לכן את מיזוג ההיסטוגרמות <u>יש לעשות בדרך אחרת.</u>

כאמור למעלה, שני תהליכי ה- MPI אמורים להשתמש ב- OpenMP.

למען התרגול, כל אחד מהם יטפל במיזוג ההיסטוגרמות (שנוצרו תוך שימוש ב- OpenMP) בדרך אחרת:

תהליך 0 ישתמש ב- reduction כדי למזג את ההיסטוגרמות הפרטיות.

שימו לב שמשתנה הרדוקציה ב- OpenMP יכול להיות גם מערך (או חלק רציף

של מערך ((זה נקרא array section) אבל לזה לא נזדקק). במקרה כזה הרדוקציה

נעשית על איברים תואמים של המערכים הפרטיים (יש רקדוקציה של כל האיברים

עם אינדקס 0, רדוקציה של כל האיברים עם אינדקס 1 וכן הלאה).

התהליך השני של MPI יבצע את מיזוג ההיסטוגרמות בדרך שונה:

כל אחד מה- OpenMP threads ייצר היסטוגרמה "פרטית" כמו מקודם אבל

טכנית, ההיסטוגרמות לא תהיינה private אלא תהיינה נגישות לכל ה- threads

לצורך המיזוג שלהן: כל thread יהיה אחראי על חישוב מספר כניסות בהיסטוגרמה המאוחדת ואז לא נוצרrace condition כי אין מצב בו מספר threads ניגשים לאותה כניסה בהיסטוגרמה.

ניתן לייצג את ההיסטוגרמיות "הפרטיות" בעזרת מבנה נתונים כזה:

```
int *histograms[10];
```

כאן מניחים שיש בסך הכל עשרה threads. הכניסה histogram[t] מצביעה להיסטוגרמה הפרטית של thread מספר t. ההיסטוגרמה הזאת היא בעצמה מערך של 256 אלמנטים מטיפוס int.

אבל בתכנית שלכם אין להניח מראש מה מספר ה- threads אלא לברר אותו בעזרת קריאה ל- `omp_get_num_threads()`. יש להקצות זיכרון למבנה הנתונים

הנ"ל באופן דינמי כלומר תוך שימוש ב- malloc (גם calloc שמאפס את הזיכרון

שהוא מקצה עשוי להיות שימושי).

<u>שילוב OpenMP, MPI ו- CUDA באפליקציה אחת.</u>

הנה דרך פשוטה אחת לעשות את זה:

בקובץ אחד (קובץ C עם סיומת נקודה c) כתבו את הקוד שמשתמש ב- MPI ואת הקוד שמשתמש ב- OpenMP. באופן כללי, תהליך שמשתמש ב- MPI יכול (כמו

כל תהליך אחר) ליצור threads (בעזרת OpenMP למשל).

באותו קובץ קראו לפונקציה (קריאה רגילה לפונקציה בשפת C) שתוגדר בקובץ אחר (עם סיומת נקודה cu) שבו תכתבו את הקוד שמשתמש ב- CUDA. חלק מהקוד הזה ירוץ על ה- host וחלק על ה-device.

כמובן שגם קוד שמשתמש ב- OpenMP (או ב- MPI) ניתן לשים בפונקציות (שיוגדרו בקובץ הראשי או בקבצים אחרים). וכרגיל ניתן להשתמש בקבצי include לפי הצורך.

<u>הערה על standard input ו- standard output</u>

תזכורת: את ה- standard input ואת ה- standard output ניתן לכוון לקבצים

ע"י שימוש בסימונים > ו- < בפקודה שנותנים ל- shell.

למשל אם לקובץ ההרצה של תכנית קוראים myprog אז ניתן לתת פקודה כזאת:

```
./myprog < foo.txt > bar.txt
```

ואז כאשר התכנית תקרא מה- standard input (למשל ע"י שימוש ב- scanf) היא בעצם תקרא מהקובץ foo.txt

וכאשר היא תכתוב ל- standard output (למשל ע"י שימוש ב- printf) היא בעצם תכתוב לקובץ bar.txt.

בתור ברירת מחדל, מה שנכתב ל- standard output מגיע למסך ומה שנקרא

מה- standard input נקרא מהטרמינל (כלומר ממה שהמשתמש כותב בלוח המקשים).

למשל אם נרשום פקודה

```
./myprog < foo.txt
```

אז מה שהתכנית תכתוב ל- standard output יגיע למסך (היא עדיין תקרא קלט מ- foo.txt)