אוניברסיטת בן גוריון בנגב – סמסטר סתיו תשע״ה

קורס מיומנויות מחקר – הפקולטה להנדסה

בניית קומפיילר לשפת GAP עם שימוש בGPU

בר בוכובזה

סטודנט שנה שלישית לתואר ראשון
בהנדסת מערכות מידע

מרצה מלווה : ד״ר רמי פוזיס,

המחלקה להנדסת מערכות מידע

22/1/2015

תוכן עניינים

תקציר 3
1. מבוא וסקירת ספרות 4
2. הקומפיילר המשופר – שיטות וחומרים 6
2.1 הקומפיילר הראשוני ב-C# 6
2.2 סיבות לשימוש ב-GPU 7
2.3 בניית המנגנון בשילוב עם ה-GPU 8
2.3.1 עקרונות עבודה עם מעבדים גרפיים – בפרט OpenCL 8
2.3.2 מבנה הקוד המחבר בין CPU ל-GPU 9
2.3.3 מגבלות OpenCL 10
2.3.4 התאמת הפסאודו קוד מ-Python ל-OpenCL 11
2.3.5 מבוא לשימוש בפונקציות באלגברה רלציונית על GPU 13
2.4 אלגוריתם "תחום ההגדרה של GAP" + שימוש ב-GPU 15
2.5 מימוש הפונקציות הרלציוניות להרצה ב-GPU 21
2.6 שימוש בתחום ההגדרה להרצת החוק 23
2.7 מנגנון ההרצה 24
3. תוצאות ומסקנות 26
3.1 בדיקת מהירות ה-GPU בהשוואה ל-CPU 26
3.2 בדיקת הרצת החוקים עד להתכנסות 28
3.3 מסקנות והצעות 29
3.4 קוד הפרויקט 30
4. ביבליוגרפיה 30

תקציר

במסגרת הבעיה אשר אותה הייתי צריך לחקור, הייתי צריך לקחת שפה מאוד פשוטה תחבירית הנקראת GAP (General Annotated Programming) המאפשרת מידול של מודלי דיפוזיה ביחד (Subrahmanian V.S., 2010) , ובנויה מפקודות (הנקראות בשפה זו "חוקים") הבנויים כתנאי לוגי "אם".

אם התנאי אשר מתקיים בחלק אחד של החוק (הנקרא body) על בסיס הנתונים אשר נמצאים לי במבנה הנתונים עבור פרדיקט/ים מסוים/ים, יתבצע שינוי במבני הנתונים בהתאם למה שנקבע בחלק השני של החוק (הנקרא header).

מטרת הפרויקט היא להדר כל חוק GAP לשפת Python, אולם השונה מפרויקטים קודמים הוא שהקוד אשר יצא מהמהדר ירוץ <u>גם</u> על המעבד הגרפי של המחשב (המכונה GPU – Graphic Processing Unit).

במסגרת חקירת הבעיה, אציג טענה שאותה אנסה לבדוק :

זמן הריצה של חוקי GAP עד להתכנסות מתחת לרף אפסילון מסוים <u>יקטן</u> במידה ואני <u>פוסל ככל הניתן</u> השמות לשמות משתני הארגומנטים, אשר קיימים בפרדיקטים של בלוקי ה-body (כאשר כל בלוק מורכב מפרדיקט המכיל ארגומנטים, ונוטציה מסוימת לאותו פרדיקט) אולם ה-body לא מתקיים עבורם. פסילת ההשמות שלא מקיימות את ה-body תתבצע על ה-GPU ואילו עדכון המידע על פי החוקים אחרי פסילת השמות יתבצע על גבי ה-CPU, באמצעות קוד מקומפל.

<u>סקירה כללית</u>

בחלק הראשון של הדו"ח אציג למעשה תיאור של מה שנעשה בעבר עבור GAP ומהם התוצאות אשר הגיעו אליהם עד אותה נקודה.

בחלק השני אציג את הקווים הכלליים של הקומפיילר, אשר משתמש ב-OpenCL על מנת לצמצם את זמן הריצה של תהליך ההרצה של החוקים עד להתכנסות.

בחלק השלישי אציג את מנגנון "תחום ההגדרה של GAP" שרץ על ה-OpenCL אשר מנצל את יכולות המקביליות של המעבד הגרפי כדי לפסול השמות של משתני ארגומנטים אשר לא מקיימים את ה-body של חוק.

בחלק הרביעי אראה את תוצאות ההרצה של הקומפיילר בהשוואה לקומפיילר הראשוני, ואת המסקנות הנובעות מהקומפיילר החדש.

1. מבוא וסקירת ספרות

שפת GAP (הידועה גם כ-General Annotated Logic Programming) הינה שפה אשר משמשת למידול של מודלים המתארים תהליכים של דיפוזיה (=פעפוע) ברשתות שונות (בעיקר לרשתות חברתיות) ומאפשרת לבדוק מספר פרמטרים שונים במקביל. (Subrahmanian V.S., 2010)

כל פקודה בשפה זו (הנקראת בשם "חוק"), בנויה משני חלקים :

- Body – תנאי "וגם" המורכב מבלוקים (כאשר כל בלוק מורכב מ[פרדיקט]([ארגומנטים]) :[נוטציה])
- Header – בלוק נוסף אשר ערך הפרדיקט בבלוק משתנה בהתאם לנוטציה.

כל חוק בנוי מצורת : $header \leftarrow body$. למעשה, אם ה-body של חוק מתקיים, אזי ה-header חייב להתקיים ועל כן יש לבצע שינוי בערך הפרדיקט בהתאם למה שהוגדר באותו בלוק header. אנחנו נבצע שינוי כאשר גודל השינוי יהיה חיובי וגדול מאפסילון שהוגדר מראש. (ברירת מחדל $\varepsilon = 10^{-6}$)

הוכח בעבר שאם אנחנו לוקחים קבוצת חוקים R ונבצע הרצה של אותם חוקים על פי הכללים שהוסברו בנוגע למבנה כל חוק – כמות הפעמים שיתבצע שינוי תתכנס עד ל-0 (שכן השינוי יתכנס עד למתחת אפסילון שהוגדר מראש).

לצורך ההרצה של החוקים, יש צורך לבנות מנוע אשר ידע כיצד לקרוא את החוקים, ויריץ אותם לפי דרישה (הרצה יחידה של כל החוקים פעם אחת בלבד, או הרצות של כל החוקים עד להתכנסות).

בתחילת העבודה על הקומפיילר שיבצע ריצה גם על ה-GPU, קיבלתי מד״ר פוזיס קומפיילר ראשוני אשר מבצע את הנדרש, אולם הקומפיילר הנ״ל הכיל מספר מגבלות :

1. הקומפיילר רץ אך ורק על ה-CPU וב-Single Thread – משמע שיש אזורים בקוד הקיים שאותם ניתן לבצע בצורה מקבילית ובכך לצמצם את זמן הריצה של הקומפיילר.
2. אנחנו מעוניינים להוסיף תמיכה בפעולות אריתמטיות נוספות בעתיד ובאותו קומפיילר התמיכה הייתה די מוגבלת.

לאורך העבודה על הקומפיילר המשופר, עלה בפי רעיון בו אשתמש בפונקציות שמתבצעות על רלציות (Join, Selection, Projection, Distinct), אולם רציתי להפעילן על ה-GPU.

מבדיקה התברר שמחקר שבדק מימוש של פונקציות אלה על רלציות להרצה על ה-GPU, הראה שיפור בהרצה של פונקציית join (Diamos, 2012)ביותר מ-90% מזמן ההרצה על CPU, כמו כן גם הפעולות שצוינו לעיל שופרו בזמן הריצה שלהם ביותר ב-90%.

בעת קבלת הקומפיילר מהמרצה, קיבלתי למעשה קומפיילר ששופר כבר בעזרת אופטימיזציה על גבי מבנה הנתונים עצמו, אולם לא היה ידוע אם אופטימיזציה זו הייתה עוזרת לי בהרצה על גבי ה-GPU של החוקים.

2. הקומפיילר המשופר – שיטות וחומרים

2.1 הקומפיילר הראשוני ב-#C

בתחילת העבודה על המנוע בשלב הראשון, כדי להבין כיצד השפה עובדת טוב׳ התחלתי לכתוב את המנוע בשפת #C - ללא שימוש ב-GPU. בעצם מדובר בלא פחות במימוש נאיבי אשר מטרתו הייתה לעזור בהבנת שפת GAP. לא הייתי יכול להתחיל לעבוד על המנוע עם התאמות ל-GPU אם השפה לא הובנה כמו שצריך.

פירקתי כל חוק בשפה לאובייקטים של מחלקות אשר בניתי. בנייה זו הייתה מורכבת ברמה שלכל רכיב בחוק הייתה מחלקה משלו (מרשימת החוקים, דרך חוק, פרדיקט, ארגומנטים ונוטציה).

אחת השאלות העיקריות בעת העבודה על בניית הקומפיילר הראשוני הייתה כיצד להחזיק את המידע אשר אעבור עליו בהרצת החוקים. המבנה אשר מצאתי לנכון לעבוד אתו הוא ״עץ אינדקס״ מכיוון ש-

- כל פרדיקט בנוי משמו ומהארגומנטים אשר בונים אותו. הרבה פעמים קיים יותר מארגומנט אחד עבור אותו פרדיקט, וכן רציתי לצמצם ככל הניתן את האחזקה של המפתחות בזיכרון. עץ אינדקס מאפשר לי זאת בכך שהשורש בעץ מייצג את ערכי הארגומנט הראשון – ולא אצטרך לבצע חזרה על אותם ערכים שוב ושוב.
- העץ רץ על גבי קוד שפועל על Single Processing – כך שמירוצים הם דבר שנמנע בעת העבודה על גרסה זו, ולכן התאפשר גם השימוש במבנה זה.

לאחר בדיקה שהקוד אשר הוכן ב- #C תקין, ביצעתי בדיקות השוואה קלות של זמני הריצה של כל קוד בין שני המנועים – המנוע ב-Python אשר היה קיים לפני תחילת הפרויקט, ועבר מקצה שיפורים שכלל אופטימיזציות על מבנה הנתונים (שזהו למעשה הקומפיילר אשר קיבלתי בתחילת העבודה על הבעיה), והמנוע אשר בניתי ב-#C . שתי הבדיקות נעשו על מחשב ThinkPad T430. הבדיקות לא היו מחמירות אולם הם כן שימשו אותי להכוונה בנוגע לדרך העבודה שלי על הפרויקט.

המנוע שבניתי ב-C# רץ בממוצע כ-10 דקות להרצה בודדה (כאשר הוא הופעל במשך עשר פעמים) ואילו המנוע שכתוב ב-python רץ תוך פחות מדקה בממוצע (לכל עשרת הפעמים).

נעשתה בדיקה מעמיקה מהיכן מגיעה הבעיה בזמני הריצה האיטיים. הבעיה העיקרית נבעה מהעובדה כשבכל פעם שניגשתי לאיבר שלא היה קיים בעץ האינדקס, בניתי חוליה עם הערך 0. למעשה גררתי מצב בו החזקתי כמות אובייקטים גדולה (ידוע שב-GAP רוב הערכים האפשריים שיש לנו הם 0 למעט מה שצוין שלא (Subrahmanian V.S., 2010)), מה שהקשה על זיכרון המחשב, ויתרה מזאת – גם על זמני הריצה.

העדפתי שלא להמשיך ולעבוד על המנוע אשר בניתי, שכן –

- הוא לא יתאים לי לעבודה עם המעבד הגרפי. הוא הותאם להרצה על גבי Single Process. ברגע שאתחיל לעבוד בצורה מקבילית אשבש את ההרצה ואצטרך לשנות מבנים רבים בהתאם לכך.
- C# היא שפת התכנות החזקה אצלי ולכן העדפתי בשלב זה רק לפתח את הקומפיילר בשפה זו על מנת למקסם את הבנתי ככל הניתן בנוגע ל-GAP.
- בהתייעצות ולבקשת ד"ר פוזיס, הוחלט על מעבר ל-python (שזוהי השפה בה נכתב הקומפיילר שקיבלתי בתחילת העבודה). החלטה זו נבעה גם כדי לצמצם את הבדלי הריצה בין שתי שפות התכנות. כמו כן אם השפה בשני המנועים תהיה זהה – נוכל להוריד גורם רעש של שפת התכנות בעת חישוב התוצאות.

2.2 סיבות לשימוש ב-GPU

כדי להבין למה כדאי להשתמש במעבד הגרפי של הכרטיס הגרפי, צריך להבין קודם כל כיצד הוא בנוי.

המעבד הגרפי המודרני (נכון לכתיבת שורות אלה) מכיל בתוכו כ-8 ליבות חישוב מרכזיות. בכל ליבה יש בין 8 ל-16 יחידות חישוב משניות. הקוד לא חייב להיות זהה בין ליבות חישוב מרכזיות, אולם ביחידות החישוב המשניות אשר נמצאות בתוך ליבת חישוב מרכזית, הקוד יהיה זהה. על כן, יש לנו בין 64 ל-128 (יש מעבדים שמכילים גם 256 ו-512 ליבות חישוב בהתאם למבנה שלהם) יחידות עיבוד שונות

אשר רצות במקביל ויכולות לבצע חישובים מאוד מהירים. מכיל bus נתונים מהמהירים שיש היום בשוק וכמות ה-RAM שהמעבד מקבל גבוהה יותר מאשר ה-CPU, כתוצאה מכך שהמעבד אמור לבצע חישובים מורכבים בצורה מהירה ככל הניתן (בעקבות הגרפיקה).

הערה[בב1]: יש לרשום מהי המהירות

כוח חישוב שנמצא במעבד שכזה מאפשר לנו לבצע עבודה מקבילית אמתית (לא מקביליות-דמה כמו שה-CPU מבצע באמצעות חלוקת זמן החישוב בין התהליכים במחשב). בהינתן בעיה שניתנת להפוך אותה למקבילית (קרי יצירת חוסר תלות בין הגורמים השונים בבעיה), ניתן להקטין את זמני הריצה בצורה מובהקת.

2.3 בניית המנגנון בשילוב עם ה-GPU

בתחילת העבודה על המנוע החדש, היה חשוב להבין על איזו סביבה אני הולך לעבוד ולהשתמש בה. התכנון המקורי היה לעבוד עם CUDA (שהיא שפת תכנות שפותחה בידי NVidia בסוף 2006 עבור הכרטיסים הגרפיים שהיא מייצרת), אולם לצערי גיליתי שהמחשב שלי לא מכיל כרטיס גרפי של NVidia, אלא מכיל את הכרטיס הגרפי Intel Graphics HD 4000, והוא לא תומך ב-CUDA.

לאחר בירור מעמיק, גיליתי שקיימת שפה שהחלו בפיתוחה בין השנים 2008-2009 אשר נקראת OpenCL והיא למעשה שפת אחות של CUDA שפועלת באותם תחומים של תכנות על גבי המעבד הגרפי, והיא כן נתמכת בידי הכרטיס הגרפי אשר נמצא במחשבי האישי. בתיאום עם ד"ר פוזיס נקבע כי אעבוד עם OpenCL.

הערה[בב2]: בדוק גם את זה !

2.3.1 עקרונות עבודה עם מעבדים גרפיים – בפרט OPENCL

OpenCL היא שפה אשר קוד ההרצה מורכב משני חלקים:

1. קוד בשפת c, כאשר c קיבלה תוספות מיוחדות עבור העבודה עם ה-GPU, והקוד שמור בקבצים בעלי סיומת cl. זהו למעשה הקוד אשר **רץ על גבי ה-GPU.** הקוד בנוי כך שבתחילתו אנחנו מקבלים מליבת ההרצה את מספר ה-thread עליו אנחנו עובדים (באמצעות פונקציה הנקראת get_global_id), ופרמטרים אשר יועברו לנו מקוד מספר 2 (מאתחל ההרצה של הקוד על גבי

ה-GPU). חשוב לדעת לתכנן כמו שצריך את הקוד הנ״ל, שכן קשה מאוד לבצע debug במידה ויש באג בקוד.

2. קוד בשפה אחרת (כאשר מדובר ב-c או כל שפת תכנות אשר הוסיפו לה תמיכה ב-OpenCL, כמו python) אשר תפקידו להכין את ה-GPU לקראת ההרצה. קוד זה הינו קוד קריטי מהרגיל כי כל דבר שלא נעביר בצורה נכונה עלול להשתבש והמידע יהיה לא תקין. כמו כן, קוד זה הוא ארוך מהרגיל בגלל סדרת הפעולות בו, ויש צורך בריכוז רב בעת כתיבתו, אולם זהו קוד בעל מבנה כמעט קבוע בכל מצב בו נצטרך להריץ קוד על ה-GPU.

2.3.2 מבנה הקוד המחבר בין CPU ל-GPU

את הקוד ניתן לחלק לשני חלקים :

1. <u>קוד גלובאלי לכל הפעולות אשר מכינות את ה-GPU להרצת פונקציה</u> (רץ פעם אחת בלבד)
   a. שומרים בזיכרון platform – כאשר מדובר ברשימה של devices (יפורט בסעיף הבא) כאשר ה-Vendor (היצרן) של כל המכשירים הוא המכנה המשותף שלהם. לדוגמה : אם אני רוצה platform של intel, אני אקבל את רשימת ה-devices של intel אשר קיימים לי במחשב.
   b. שומרים בזיכרון device – device הוא למעשה רכיב חומרה אשר בנוי לעבוד עם OpenCL – בין אם זה בדרייברים אשר מותקנים במחשב או שפיזית בנו אותו כדי לאפשר זאת. דוגמה : מתוך ה-platform ששמרנו קודם נדרוש את כל ה-devices אשר הם GPU. (גם CPU מוגדר כ-device בשפה זו, אולם לצורך פרויקט זה, מידע זה הינו זניח).
   c. יוצרים context על ה-device – זהו למעשה אחד הרכיבים הקריטיים בקוד שלנו – תפקידו לנהל את כל הפעולות השונות שאנחנו עושים בשביל להריץ כל קוד על ה-GPU.

d. שמירת תור הרצות – זהו תור אליו אנחנו מכניסים את הקוד שלנו להרצה וכאשר המעבד הגרפי פנוי הוא יתחיל להריץ את הקוד ששלחנו בהתאם למה שקיבל בארגומנטים.

e. בניית program מקובץ / מחרוזת שבנינו תוך כדי ריצה / קובץ בינארי – יש לקחת את הקוד שכתבנו בקובץ ה-cl ופשוט להדר אותו כדי שיהיה ניתן להריצו.

2. <u>קוד אשר מבוצע לפני הפעלת ה-GPU</u> (רק לפני כל הרצה על ה-GPU)

a. שמירת kernel – כאשר kernel זוהי פונקציה בקובץ ה-cl אשר מיועדת לרוץ על ה-GPU. היא מסומנת באמצעות הדגל __kernel בתחילת כותרת הפונקציה.

b. יצירת באפרים – לכל מערך שאנחנו נבנה, אנחנו חייבים לבנות באפר שידע להעביר ל-GPU את המידע אשר אנחנו רוצים לשלוח לאותו ה-kernel אשר שמרנו בסעיף הקודם. במידה ואנחנו רוצים להעביר שדה גלובלי לכל הליבות, נעביר אותו כמערך בגודל 1.

c. הגדרת ארגומנטים – יש להגדיר את הארגומנטים של הפונקציה לפני הרצה. במידה ומדובר בשדה אטומי , נהפוך אותו למערך מאותו סוג טיפוס בגודל 1 ונעביר אותו ישירות ל-kernel (מערכים כאלה אין טעם להעביר ב-buffer כי המנגנון יודע אוטומטית להעביר אותם לשם, עקב גודלם הקבוע).

d. שליחת ה-kernel לתור ההרצות.

e. המתנה עד לקבלת event מתור ההרצות שפעולת הקוד הסתיימה.

f. קריאת המערכים אליהם כתבנו את התוצאות של ההרצה.

2.3.3 מגבלות OPENCL

- אין הקצאות דינאמיות בקוד אשר נמצא בקובץ cl.

  <u>משמעות :</u> אין ADTs אלא אם כן הם ממומשים באמצעות מערכים.

- הטיפוסים עצמם בקוד הם <u>טיפוסים פרימיטיביים</u> , למעט מערכים אולם הם רק בנויים מאיברים שהם מטיפוסים פרימיטיביים בלבד.

הערה : struct הינו מבנה קביל ב-OpenCL, רק אם הוא מכיל שדות אטומיים בלבד, ללא מצביעים.

- מערכים בעלי יותר מממד אחד, יקראו ב-GPU כמערכים חד ממדיים.

משמעות : נצטרך להעביר את גודל הממדים של כל מערך דו/תלת ממדי על מנת שנוכל לעבוד עליו בממדים אלה.

- פונקציית kernel לא יכולה להחזיר דבר. לכן אם נרצה להחזיר תוצאה נשים אותה בשדה אותו נשלח באמצעות באפר.
- אם שדה אטומי הוא שדה קבוע, קרי - שערכו לא משתנה, נעביר אותו בהתאם למה שנאמר קודם. אחרת, בנה באפר עבור השדה שאותו אתה רוצה להעביר וטען אותו ל-kernel.

2.3.4 התאמת הפסאודו קוד מ-PYTHON ל-OPENCL

לצורך הסבר חלק זה של המסמך, ניקח את הקוד אשר יצא כפלט מהקומפיילר הבסיסי עבור החוק הבא :

$$g1_member(x): a * b * c * d$$
$$\leftarrow q(x): a \& g1_member(y): b \& \ p(y): c \& q(x): d$$

להלן הקוד אשר יצא כפלט עבור אותו חוק :

```
wasachange=0
for (Y,) in g1_member.keys(()):
 for (X,) in friend.keys((Y,)):
  b=g1_member[(Y,)]
  c=p[(Y,)]
  a=friend[(Y,X,)]
  d=q[(X,)]
  if 1  and q[(X,)]>=d  and p[(Y,)]>=0.25  and friend[(Y,X,)]>=a  and
g1_member[(Y,)]>=b :
   _temp = a*b*c*d;
   if _temp>0.000001+g1_member[(X,)]:
    g1_member[(X,)]=_temp;
    wasachange+=1
return wasachange
```

כפי שניתן לראות, בלולאות אשר נמצאות בתחילת הקוד אנחנו לוקחים את רשימת הערכים האפשרית לכל משתנה, ואנחנו מצמצמים את הערכים עליהם אנחנו נרוץ.

לאחר מכן אנחנו טוענים בנוטציות את הערכים שכל אחד מקבל ממבנה הנתונים על סמך ערכי משתני הארגומנטים שלנו בחוק.

מתבצעת בדיקה לראות האם ה-body של החוק מתקיים. במידה וכן – אנחנו מחשבים את נוטציית בלוק ה-header ואנחנו רואים האם יש שינוי בין הערכים של הנוטציה לערך הפרדיקט במבנה הנתונים, שהוא גדול יותר מאפסילון אשר בחרנו מראש (במקרה שלנו $\varepsilon = 10^{-6}$), ורק אז נעדכן את ערך הפרדיקט בהתאם.

מניתוח הקוד היוצא מהקומפיילר ניתן לקבוע ש :

1. בתנאי הראשון בקוד – עבור כל פרדיקט מסוג predicat(x):a אין טעם שיהיה תנאי a>=predicat[(x,)] מכיוון שלפני כן אנחנו ממילא עושים a=predicat[(x,)].
2. **למה עבורנו כדאי לבצע השמה לנוטציות לפני הבדיקה שה-body מתקיים? הרי אם התנאי לא מתקיים, אנחנו לא מבצעים שום דבר.**

המשפט השני הינו משפט שעליו כל העבודה בהמשך תסתמך. למעשה, אין טעם לבדוק השמות למשתני הארגומנטים אם ה-body לא מתקיים בכלל עבור ההשמות הללו. יתרה מזאת – אם אין לנו בכלל השמות אפשריות – ניתן פשוט לדלג על הרצת החוק.

על כן, נשאלת השאלה שעסקתי בה במשך הקורס והיא נבעה מאותה מסקנה : **עד כמה ניתן לשפר את הרצת חוקי ה-GAP עד להתכנסות, אם אנחנו מבצעים תהליך של צמצום השמות ככל הניתן למשתני הפרדיקטים ?**

אולם גם נשאלת שאלה אחרת : **איך בעיה זו יכולה להיפתר באמצעות שימוש במקביליות שמציע לי ה-GPU ?** התשובה מבוססת על כללי תכנות שקיימים כבר (ויפורטו בחלק הבא של המאמר), ובעקבות כללים אלה מבנה הקוד יהיה ניתן להתאמה ל-GPU.

2.3.5 מבוא לשימוש בפונקציות באלגברה רלציונית על GPU

בהתייעצות אשר נעשתה בתחילת העבודה על הקומפיילר המשופר שתומך גם בהרצה על GPU, עם ד"ר פוזיס, נקבע כי כדי לשמור על פשטות אעבוד עם רלציות להחזקת הערכים של כל פרדיקט.

לאחר זיהוי המסקנה מהעמוד הקודם, זיהיתי כי ניתן לבצע צמצום השמות **דווקא בגלל המבנה הרלציוני של מבני הנתונים של הפרדיקטים**.

הפתרון : שימוש בפונקציות שבסיסן לקוח מתחום האלגברה הרלציונית (projection, selection, inner join and distinct).[1]

בשלב הנוכחי של העבודה עניין אותי כיצד אני מצמצם את ההשמות האפשריות שמקיימות את ה-body – אולם בו בעת גם לצמצם את זמן הריצה של הליך הצמצום.

אנחנו רוצים להחזיק מבנה נתונים שיכיל השמות של משתני הארגומנטים, ללא כפל משתנים. ניתן לקבוע כי אם יש לנו את משתני הארגומנטים x,y,z בחוק מסוים, אזי מבנה הנתונים יכיל וקטורים שמכילים ערכים עבור (x,y,z,). אנחנו לא רוצים כפל ערכים למשתנה - גם כדי לצמצם נפח בזיכרון, וגם כדי לשמור על עקיבות – יכול להתקיים במצב הנ"ל שעבור אותו משתנה אנחנו נחזיק ערכים לא שווים – אז במה נבחר ? לשם כך –

הגדרות :

- רלציית ההשמות - רלציה בה כל השדות ביחד מהווים מפתח ראשי. למעשה לא נאפשר שתהיה את אותה ההשמה פעמיים ברלציה זו. שדות הרלציה הם למעשה שמות משתני הארגומנטים.
- תמונת המשתנים של פרדיקט / רלציה – בהינתן פרדיקט / רלציה המכיל/ה ארגומנטים, תמונת המשתנים היא למעשה רשימה בה מצוין כל ארגומנט. קיימים 2 סוגים של תמונות משתנים :
    - תמונת משתנים **וירטואלית** – בהינתן פרדיקט וארגומנטים המוצמדים לאותו פרדיקט / רלציה, ומילון שנותן לכל ארגומנט

[1] ידוע ש-distinct היא פונקציה שלא קיימת בתחום האלגברה הרלציונית בגלל שבאלגברה זו אין חזרות על רשומות, אולם היא תופעל מיד אחרי projection על מנת למנוע כפילויות. כמו כן, זוהי פונקציה לכל דבר שתפקידה לבצע פעולה על גבי רלציה.

מספר סידורי – תמונת המשתנים הווירטואלית היא למעשה רשימת הארגומנטים, אולם מתורגמים למספרים.

דוגמה : עבור g(x,y) ו-x=1,y=0 נקבל (1,0)

דוגמה : עבור p(x,y,x) ואותו מילון מקודם נקבל (1,0,1)

- תמונת משתנים **פיזית** – בהינתן פרדיקט / רלציה וארגומנטים המוצמדים לאותו פרדיקט / רלציה, ומילון שם ארגומנט למספר סידורי (כמו בסעיף הקודם), אנחנו יוצרים tuple בגודל k (כמות האיברים במילון), כאשר מקום i ב-tuple מייצג את שם הארגומנט ש-i הוא המספר הסידורי של אותו ארגומנט. ערך המקום ה-i בתוך ה-tuple הוא למעשה המקום הפיזי של אותו ארגומנט ברשימת הארגומנטים / רלציה. במידה והארגומנט לא קיים ברשימה נציב מספר שלילי (ערך ברירת מחדל 1-). כל הערכים בתמונת משתנים זו חייבים להיות מספרים שלמים.

לדוגמה : g(x,y) ו-מילון x=1.y=2,z=0 נקבל (-1,0,1) .

- במקום 0 שמייצג את z לא קיים לנו ברשימת הארגומנטים ערך ועל כן נציב 1-
- במקום 1 שמייצג את x קיים לנו מופע יחיד של x במקום 0. על כן נציב 0.
- במקום 2 שמייצג את z קיים לנו מופע יחיד של y במקום 1. על כן נציב 1.

<u>תרחיש : מה קורה בהינתן g(x,y,x) ?</u>

תמונת הפנים הפיזית תהיה זהה כמו מהדוגמה שבעמוד הקודם (אנחנו שומרים את המיקום הראשון בו הארגומנט מופיע), אולם אנחנו מחזיקים רשימה נוספת שנקראת **"רשימת ההתאמות"** והיא מכילה את כל הזוגות שתואמים אחד לשני.

במקרה שלנו האיבר במקום ה-0 שווה לאיבר במקום ה-2. על כן נוסיף (0,2) לרשימה. רשימה זו תהיה הכרחית לאלגוריתם "תחום ההגדרה" אשר יוגדר בהמשך הדו"ח.

2.4 אלגוריתם "תחום ההגדרה של GAP" + שימוש ב-GPU

האלגוריתם אשר יפורט בחלק זה של הדו"ח, מהווה את החלק העיקרי בעת הרצת חוק GAP . אלגוריתם זה מורכב מסדרה של פונקציות אשר מחוברות בעת הרצת האלגוריתם, שנקרא "תחום ההגדרה של GAP" (או בקיצור "תחום ההגדרה")

האלגוריתם בנוי משלוש שכבות :

1. אלגוריתם השאילתה (בקר השליטה) – למעשה מדובר בשכבה העליונה של האלגוריתם – המעטפת הכללית אשר קוראת לפונקציות שנמצאות בשכבה התחתית (שתפורט בסעיף הבא). אלגוריתם זה למעשה מפעיל את הפונקציות השונות על פי הסדר אשר יפורט בצורה מסודרת בהמשך הדו"ח.
2. פונקציות רלציוניות - זוהי למעשה השכבה **הקריטית** של האלגוריתם, והחיבור הפיזי ל-OpenCL נעשה בפונקציות אלה. הפונקציות הללו מגדירות ממשק ב-python לקוד אשר רץ באמצעות OpenCL, כך שהקוד בעת זימונו יהיה נקי ככל הניתן. פונקציות אלה מממשות פונקציונליות על רלציות :
   - Join – יצירת רלציה Inner Join בין 2 רלציות.
   - Selection –
     - Selection – Select Above של ערך גדול ממספר c קבוע.
     - Selection – Filter של שוויון בין שדות ברלציה.
   - Projection – יצירת טבלה בעלת חלק מהשדות של הטבלה שמתקבלת כפרמטר.
   - Distinct (מבוצע מיד אחרי Projection) – מצמצם רשומות כפולות בטבלה.
3. פונקציות עזר – משמשות את הפונקציות הרציונליות בעיקר לצורך חישובי "תמונות משתנים" (גם תמונות וירטואליות וגם תמונות פיזיות).

על מנת שנוכל להפעיל את הפונקציות הרציונליות, יש את "אלגוריתם השאילתה" שמקבל סדרה של בלוקים (מה-body) ומשתמש בתבנית קבועה של עץ שאילתה כדי לצמצם ככל הניתן השמות למשתני הארגומנטים אשר לא מקיימים לפחות את אחד הבלוקים.

הסיבה למה יש תבנית קבועה ולא קוד אשר מתקמפל ומתאים את עצמו לתרחיש נובע מכך שבבירור מעמיק התגלו בעיות בזיכרון, אשר פשוט לא היה בו מקום להחזיק יותר איברים. התבנית הקבועה פתרה את בעיית הזיכרון.

לאורך כל האלגוריתם נחזיק רשימה שנקראת "רשימת הרלציות" והיא תכיל איברים שמורכבים משני חלקים : 1. הרלציה עצמה, 2. תמונת המשתנים הפיזית של אותה רלציה (שמירתה קריטית כי רוב הפעולות ב-OpenCL מסתמכות על תמונה זו).

כל עץ שאילתה מורכב מ-3 חלקים (כאשר כל חלק תלוי בתנאים המקדימים לו) :

חלק ראשון : Filter

בחלק זה אנחנו נפטרים מבלוקים בעלי פרדיקטים בהם קיימים ארגומנטים שחוזרים על עצמם יותר מפעם אחת ברשימת הארגומנטים. G(x,y,x) הינו פרדיקט שיופעל עליו חלק זה.

פרדיקטים שעבורם רשימת הארגומנטים היא רשימה בה כל ארגומנט מופיע ברשימה פעם אחת בלבד תעבור ישירות לחלק ב', באמצעות רשימת הרלציות.

פסאודו קוד

1. לכל בלוק ב-body [תזכורת :בלוק =<פרדיקט>(<ארגומנטים>) :<נוטציה>]
    a. צור תמונת משתנים וירטואלית ושמור אותה
    b. צור תמונת משתנים פיזית ושמור אותה
    c. אם קיימים איברים ברשימת ההתאמות של הבלוק :
        i. הרץ אלגוריתם FILTER. (זהו אלגוריתם שכתוב בקוד, אולם ב-OpenCL חוויתי אתו בעיות, ולכן זהו אחד משני האלגוריתמים היחידים אשר כתובים ב-python).
        ii. בתום ההרצה נקבל טבלה בה כל שדה מופיע פעם אחת ונפסלו רשומות שהשדות מ"רשימת ההתאמות" לא היו תואמים.

לקראת החלק השני של הקוד, אנחנו חייבים לוודא שאין לנו כפילויות של ארגומנטים. על כן, חלק זה פותר מאתנו את הכפילויות, וגם פוסל השמות שתנאי השוויון לא מתקיים עבורם. רלציה שעברה חלק זה תוכנס לרשימת הרלציות להמשך עבודה.

דוגמה : בהינתן מפתחות הארגומנטים הבאים עבור פרדיקט q בזיכרון :

$$[(12,13,),(12,12,),(13,11,),(24,24,),(13,24,)]\ \&\ q(a,a)$$
$$\rightarrow [(12,),(24,)]$$

הרשימה השמאלית היא למעשה רשימת המפתחות של טבלה q שהוטענה בזיכרון.

לאחר הפעלת filter נקבל רשימה רק של הערכים שמקיימים את השוויון. כמו כן, כפילות מוסרת כתוצאה מחוקי חלק א׳.

תמונת המשתנים הפיזית תישאר זהה לפני ואחרי ביצוע חלק זה, אולם התמונה הווירטואלית תשתנה, אולם אנחנו פחות נותנים חשיבות לחלק זה.

חלק ב׳ – Join

חלק זה של ההרצה הוא החלק העיקרי.

לאחר שיש לנו את הרלציות שלקחנו מהזיכרון ועשינו filter בהתאם לכללים,

אנחנו רוצים לבצע join על הטבלאות עד שנקבל טבלה שמכילה את כל שמות משתני הארגומנטים כשדות הטבלה, וכל רשומה מהווה **השמה לאותם משתני ארגומנטים**.

<u>פסאודו קוד</u>

1. כל עוד יש יותר מרלציה אחת ב״רשימת הרלציות״ :
   a. צור ״רשימת רלציות״ חדשה וזמנית
   b. אם יש כמות שווה או גדולה יותר מ-2 טבלאות :
      i. הוצא את שתי הרלציות הראשונות מהרשימה והפעל עליהן Join. (על OpenCL)
      ii. אם הרצליה שחזרה ריקה מרשומות – החזר רלציה ריקה, ותמונת משתנים פיזית של אותה רלציה – מידית.
      iii. אחרת, הכנס לרשימה זמנית את הרלציה ואת תמונת המשתנים הפיזית של אותה רלציה.
   c. אחרת, אם יש רלציה אחת :
      i. הכנס את הרלציה לרשימה הזמנית (כולל תמונת המשתנים הפיזית ששמורה לצידה)

d. אחרת, רוקן את ה"רשימת הרלציות" הזמנית ל"רשימת הרלציות" הבסיסית.

2. החזר את הרלציה היחידה שנשארה ברשימה ואת תמונת המשתנים הפיזית של אותה הרלציה.

בתום חלק זה נקבל רלציה שמכילה השמות למשתני הארגומנטים, ואת תמונת המשתנים הפיזית שלה.

חלק ג' – Select Above

בחלק זה אנחנו מסתכלים על בלוקים מתוך ה-body שבנויים בצורה הבאה :

$$< predicat > (< arguments >) : < Number >$$

למעשה, בלוקים בתבנית זו מכילים נוטציה של מספר <u>קבוע ולא משתנה</u>. לפי חוקי GAP, רק פרדיקטים שמכילים ערכים גבוהים או שווים לאותו מספר קבוע למעשה מחזירים ערך בוליאני חיובי. אנחנו למעשה מעוניינים להסתכל אך ורק על ההשמות שמקיימים בלוקים מסוג זה.

<u>שאלה</u> : למה Select Above נעשה <u>בסוף</u> ולא <u>בתחילת ההרצה</u> ?

<u>תשובה</u> : להלן דוגמה שתוכיח שחלק זה לא יכול להיות בתחילת ההרצה.

להלן חוק GAP : $\boxed{\boldsymbol{p(x,y): a * b \leftarrow q(x): a \cap q(y): 0.5 \cap q(y): b}}$

נניח שקיימים לנו בטבלת הנתונים של פרדיקט q את הערכים הבאים :

$$q \rightarrow [(22,): 0.6, (12,): 0.3, \dots]$$

אם אנחנו מפעילים את הבלוק q(y):0.5 אנחנו מסירים את (12,) מטבלת ההשמות.

אבל אם נבצע השמה $x = 12, y = 22$ ה-body <u>כן</u> מתקיים ! זהו מצב שאנחנו רוצים <u>להימנע ממנו</u> ככל הניתן, ועל כן חלק זה <u>נעשה בסוף ההרצה</u>.

פסאודו קוד

1. אם קיימים בלוקים שמקיימים את תנאי חלק ג׳ :
   a. צור ״רשימת רלציות״ זמנית ריקה
   b. לכל בלוק שמקיים את תנאי חלק ג׳ (בדיקה זו נעשית כבר בתום חלק א׳) :
      i. קח את רשימת המשתנים הווירטואלית של הארגומנטים בבלוק, ואת רלציית ה-join הסופית מסוף חלק ב׳ (כולל תמונת משתנים פיזית של הטבלה) והפעל את הפונקציה Projection (על OpenCL)
      ii. את הרלציה שחוזרת מה-Projection יש לקחת ולהפעיל פונקציית Distinct לפסילת כפילויות של רשומות (רץ על python – לוגית, האלגוריתם מורכב מדי לביזור בגלל התלות במידע הקודם). האלגוריתם גם יטען לתוך רלציה בעלת שדה אחד וכמות רשומות זהה את ערכי כל מפתח הפרדיקט ישירות לרלציה זו. רלציה זו לצורך פשטות תיקרא ״רלציית הערכים״
      iii. לאחר Distinct, נפעיל Select Above (על OpenCL) כאשר התנאי הוא שרק אם האיבר במקום ה-i ב״רלציית הערכים״ גדול או שווה לערך הנוטציה בבלוק, נעתיק לרלציית התוצאה של האלגוריתם את הרשומה במיקום ה-i.
      iv. את התוצאה מ-Select Above נכניס לתוך ״רשימת הרלציות״ הזמנית.
   c. הפעל את האלגוריתם של חלק ב׳ על ״רשימת הרלציות״ הזמנית. לרלציה המוחזרת נקרא בשם "Select Above Join Relation" (או בקיצור SAJR).
   d. הפעל Join אחרון (על OpenCL) בין SAJR לבין רלציית ה-Join הסופית שקיבלנו בחלק ב׳.

דוגמה :

נניח וקיים עבורנו אחרי חלק ב' רלציית ההשמות של הארגומנטים של החוק :

x	y	z
12	13	11
12	13	11
12	14	14
13	15	14
14	32	12
15	21	15
11	7	12
8	6	10

נניח וקיים לנו ב-body של החוק את הבלוק הבא :$p(x,y)$: 0.3

נניח ש-x=2,y=0,z=1 ממילון הארגומנטים שיצרנו לחוק בעת הליך הקומפילציה.

1. יש לנו בלוק שמקיים את תנאי חלק ג' –
   a. עבור הבלוק p(x,y):0.3 –
      i. נעשה projection כאשר התמונה הווירטואלית של הבלוק היא (2,0) ונקבל את הרלציה הבאה :

x	y
12	13
12	13
12	14
13	15
14	32
15	21
11	7
8	6

      ii. נפעיל distinct ונקבל את הרלציות הבאות (עם "רלציית הערכים") :

      iii. נפעיל Select Above עם minValue=0.3 ונקבל את הרלציה הבאה :

x	y	value
12	13	0.2
12	14	0.5
13	15	0.3
14	32	0.4
15	21	0.8
11	7	0.1
8	6	0.14

   b. נכניס את הרלציה הנ"ל לתוך רשימת הרלציות הזמנית.

X	Y
12	14
13	15
14	32
15	21

   c. נפעיל Join על הרלציה היחידה ברשימה וברלציה אשר קיבלנו בתחילת חלק זה (זוהי אותה רלציה שחזרה מחלק ב')

ועל כן - נקבל את הרלציה הבאה :

x	y	z
12	14	14
13	15	14
14	32	12
15	21	15

לסיכום, כך מורכב אלגוריתם "תחום ההגדרה של GAP".

2.5 מימוש הפונקציות הרלציוניות להרצה ב-GPU

חלק זה של הדו"ח בעצם מציג כיצד אני מממש חלק מהפונקציות הרלציוניות כדי שהן תרוצנה על גבי ה-GPU.

האלגוריתמים על גבי ה- GPU מתחלקים בפרויקט זה לשני סוגים עיקריים :

1. <u>אלגוריתמים מקבילים בצורה מוחלטת</u> –

מדובר למעשה בכל אלגוריתם אשר לא תלוי בפעולות של הליבות האחרות של ה-GPU. למשל – Projection ימומש בצורה כזו מכיוון שכפי שיוצג מיד, האלגוריתם לא תלוי באיברים אחרים של הטבלה. גם Cartesian ממומש בצורה זו.

2. <u>אלגוריתמי Buffer בעלי תלות הרצה</u> –

מדובר ב**פרדיגמה** שלמה לקבוצת אלגוריתמים אשר עובדים בעיקרון זהה ומאפשרים עבודה עם buffer שהכתיבה אליו נעשית בצורה "דמה-סדרתית". בין הפונקציות שממומשות בפרדיגמה זו : Join, Select Above.

<u>מבנה הפרדיגמה :</u>

0. לפני הרצה (ב-python) , הגדר משתנה גלובלי current = 0
1. אם הרשומה הנוכחית שעליה אתה עובד, תקנית לפי התנאים אשר נקבעו מראש :
   a. בצע בצורה **<u>אטומית</u>** (נעשה באמצעות atomic_add) –
      i. Int curr = current
      ii. Current += 1
   b. בצע כתיבה בטבלת ה-output באמצעות curr כאינדקס.

<u>היתרונות בפרדיגמה זו :</u>

- צמצום זמן המתנה בין הליבות לקטע הקוד הקריטי ל-2 פקודות של העתקה והעלאת ערך ב-1. סימון קוד הכתיבה כקטע קוד אטומי ב-OpenCL יגרור האטה של ההרצה בגלל שנאבד חלק מהמקביליות.

- מאפשר כתיבה בצורה מקבילית ל-output בגלל חוסר התלות בין רשומות היעד בכל טבלה וטבלה.
- מאפשר אחרי תום ההרצה לבצע קיזוז של טבלת ה-output לגודל המתאים – דבר שיחסוך מקום בזיכרון.

<u>חסרונות בשיטה זו :</u>

- גודל הבאפר צריך להוות חסם עליון של כמות הרשומות האפשרית לכתיבה ב-output – משמע גודל הזיכרון שיידרש יהיה גדול, על אף שברוב המקרים לא נצטרך יותר מחצי משטח זה.
- אם היינו יכולים להוציא להרצה בליבות נפרדות את הכתיבה, המבנה לא מאפשר זאת – משמע פגיעה במקביליות המקסימלית של הקוד (יש טכניקה אחרת שלא בוצעה במסגרת פרויקט זה, והיא תתואר בסיכום הדו"ח)
- חייב להתקיים התנאי בו גודל כל רשומה במערך ה-output קבוע ולא משתנה בין רשומות !

להלן הסבר של פונקציה מתוך כלל הפונקציות שרצות על ה-GPU באמצעות OpenCL. (רק יוסבר הקוד בשפת c אשר רץ על ה-GPU)

<u>אלגוריתם Projection</u>

מדובר באלגוריתם שהמטרה שלו פשוטה – נתונים לי m עמודות ו-n רשומות בטבלה ואני מעוניין באותה הטבלה, אולם עם $m \leq k$ עמודות (שהגדרתי מראש ברשימה מסודרת)

לצורך הסבר העבודה להלן קוד הפונקציה Projection :

```
__kernel
void PROJECTION (__global const int* array, const int array_col, __global
int* result, const int result_col, __global const int* places) {

                int x = get_global_id(0), y = get_global_id(1);
                result[x*result_col+y] = array[x*array_col + places[y]];
}
```

כל קוד ב-OpenCL חייב כדי לרוץ בצורה מקבילית לבצע את השורה המסומנת באפור. למעשה שורת פקודה זו מבקשת מהליבה אשר הקוד רץ עליה את מיקום

ההרצה עליו הליבה הולכת לעבוד. (x,y) ביחד הם מפתח הרצה עבור הליבה. הוא יאפשר לקבוע לנו כיצד לפעול בהמשך.

פונקציית ה-projection רצה על 2 אינדקסים שונים :

- X – רץ על רשומות הטבלה
- Y – רץ על עמודות הטבלה אשר אנו מעוניינים בהם ב-Projection.

השורה השנייה פשוט עושה העתקה של התא מהמקום המתאים מהטבלה הראשית, לטבלת ה-output, במקום המתאים.

כפי שניתן לראות, העתקה עצמה יכולה להתבצע בצורה מקבילית על גבי הליבות.

2.6 שימוש בתחום ההגדרה להרצת החוק

לאחר חישוב תחום ההגדרה, יש לבצע את הרצת החוק – הרי זאת הסיבה למה תחום ההגדרה קיים, ויתרה מזאת – המטרה היא להריץ את החוקים, לא רק את "תחומי ההגדרה" של כל חוק.

הקוד אשר יבצע את הרצת החוק יהיה בשפת python ישופר בצורה הבאה :

- הקוד לא יכיל את התנאי אשר היה בקוד שקיבלנו במנוע הבסיסי אשר קיבלתי בתחילת העבודה (עמוד 10)
- הקוד יזהה מתי יש הוספה של איבר למבנה הנתונים או רק שינוי של איבר.

מבנה הקוד

1. לכל השמה בתחום ההגדרה, בצע :
   a. הצב במשתנה a_{i} (כאשר i זהו מספר סידורי מ-0 עד k לא כולל) את ערכו של האיבר ה-i בהשמה.
   b. לכל בלוק body אשר קיימת בו נוטציה שהיא משתנה בצע :
      i. שם משתנה = מבנה נתונים["שם פרדיקט"][(tuple*)]
      (כאשר tuple* זהו tuple שמכיל משתני a_{i} בצורת תמונת המשתנים הווירטואלית של הבלוק).

דוגמה : עבור הבלוק p(x,y):b ותמונה וירטואלית (1,0)

נקבל את הפקודה b=MainDict["p"][(a_1,a_0,)

c. עבור בלוק ה-header :

i. אם לא קיים המפתח הנתון בפרדיקט בלוק ה-header :

1. הוסף את הנוטציה למבנה הנתונים.

ii. אם ערך הנוטציה של בלוק ה-header גדול או שווה לפרדיקט בלוק ה-header בהינתן מפתח נתון לפי רשימת משתני ארגומנטים + אפסילון :

1. עדכן את ערך המפתח עבור אותו פרדיקט במבנה הנתונים.

ב-Command Line אשר פיתחתי (בקובץ pygaplus.py) קיימת פונקציה בשם ExportCode שתציג על המסך לאחר הטענת חוקים למנוע, את הקוד המקומפל בהתאם. ניתן להכניס כל חוק תקני ל-Command Line ולקבל קוד מקומפל.

2.7 מנגנון ההרצה

מנגנון ההרצה אשר נמצא ב-pygaplus, מכיל מספר פונקציות מרכזיות :

1. Load_Rules מקבל קובץ gap וטוען את החוקים בו למנגנון.
2. Load_Data מקבל קובץ csv וטוען את המידע למבנה הנתונים המתאים
3. Add מאפשר להכניס חוק gap ישירות למנגנון באמצעות ה-CommandLine
4. Run מבצע הרצה בודדה של כל החוקים במנגנון על גבי המידע ששמור במנגנון.
5. Run_FixPoint מבצע הרצות של כל החוקים עד להתכנסות.
6. Export מאפשר הוצאה של המידע ששמור במנגנון לקובץ.
7. ExportCode מאפשר הוצאה של הקוד המקומפל להרצה לקובץ.

בפונקציות Run ו-Run_FixPoint יש פונקציה שרצה לפני הרצת החוקים בפעם הראשונה. פונקציה זו נקראת PreRun ותפקידה להכין את המנגנון לאיפוס וקמפול קוד לפני הרצה. הקמפול נספר כחלק מההרצה הראשונה.

בכל הרצה אנחנו בוחנים שני דברים :

- 0 הוספות בהרצת כל החוקים בהרצה הקודמת – אם התשובה היא כן, לא נעדכן לפני הרצת כל חוק את תחום ההגדרה של אותו חוק. אין הוספות = תחום ההגדרה זהה.
- 0 שינויים + הוספות בהרצת כל החוקים – הגענו לנקודת התכנסות, ועל כן נפסיק להריץ את החוקים.

3. תוצאות ומסקנות

3.1 בדיקת מהירות ה-GPU בהשוואה ל-CPU

קודם כל כדי לבחון האם האלגוריתם "תחום ההגדרה" עובד טוב יותר ב-GPU מאשר ה-CPU, אנחנו צריכים לבחון את זמן הריצה של אלגוריתם זה. אם הוא לא יעבוד מהר יותר מה-CPU או שההבדל בין הזמנים לא יהיה מובהק – אין טעם להשתמש ב-GPU.

הבדיקה היא לגבי ריצת רכיב ה-OpenCL בקוד שמתממשק עם ה-GPU.

לצורך הפשטות ובדיקת ה-GPU ככל הניתן, החוק שהוכנס לבדיקה לא כלל בלוק בו קיים פרדיקט עם רשימת ארגומנטים אשר צריכה לעבור Filter (נכון לעכשיו זהו אלגוריתם ב-python ולא ב-OpenCL בגלל בעיה שלא נפתרה עד לכתיבת שורות אלה). המידע שהוכנס היה זהה בשתי הבדיקות. על גבי המחשב רק הופעל ה-IDE שאתו אני עובד (PyCharm 4). המחשב הוא מדגם ThinkPad T430 של Lenovo + כרטיס גרפי Intel Graphics HD 4000 – וישמש לכל הבדיקות בדו"ח זה.

הבדיקה הורצה 100 פעמים על "אלגוריתם תחום ההגדרה של GAP", אולם ההבדל בין הבדיקות היה במימוש הפונקציות אשר מבצעות פעולות ישירות על הרלציות ("פונקציות רלציוניות" – עמ' 15) : המימוש הראשון שנבדק הוא מימוש שכתוב להרצה על גבי ה-GPU (באמצעות OpenCL ו-Wrapper לשפה זו ב-Python), ואילו המימוש השני שנבדק הוא מימוש שכתוב רק ב-Python ומיועד לרוץ על ה-CPU בלבד.

ולהלן התוצאות (בטבלת ניתוח שונות לבדיקת חוסר הומוגניות לפחות באחת התוחלות – מכיוון ש-k=2 לא צריך לעשות בדיקה פרטנית, כי נובע ששתי התוחלות של כל טיפול לא שווים) :

מקור השונות	SS	MS	df	F*
טיפולים	16883.27	16883.27	1	19385.15
שגיאות	172.4458	0.870938	198	
סה"כ	17055.72		199	

לפי ערך ה-F הסטטיסטי שנובע מטבלת ניתוח השונות, גם עבור רמת מובהקות של 0.01% אנחנו דוחים את השערת האפס – ($F_{1,198,0.99995} = 20.5591441$) משמע שתי התוחלות שיצאו לנו לא שוות באופן מובהק.

ממוצע זמן הרצת האלגוריתם על גבי ה-CPU בלבד עומד על 18.62966 שניות, בעוד שממוצע זמן הרצת האלגוריתם בשילוב עם ה-GPU עומד על 0.253988 שניות – צמצום זמן הריצה של האלגוריתם בכ-98.6366% מזמן הריצה של המימוש שרץ אך ורק על ה-CPU.

לכן ניתן לקבוע כי עבור האלגוריתם "תחום ההגדרה של GAP" נעדיף מימוש בשילוב עם ה-GPU.

3.2 בדיקת הרצת החוקים עד להתכנסות

אולם, בפועל מה שבאמת נבדק במשך הקורס הוא עד כמה ניתן לצמצם והאם ניתן בכלל לצמצם את זמן הריצה בפועל של מנגנון ההרצה.

שני המנגנונים אשר נבדקים הם :

- מנגנון הקומפילציה וההרצה שקיבלתי בתחילת העבודה, עם אופטימיזציה שנעשתה על מבנה הנתונים אשר מחזיק את המידע עליו אנחנו עובדים.
- מנגנון הקומפילציה וההרצה שבניתי

בבדיקה אנחנו מעוניינים לראות כמה זמן ייקח לכל חוקי ה-GAP (כאשר מדובר באותם חוקים בשני המנגנונים) לגרום לנתונים במבנה הנתונים (אשר בכל פעם מכיל נתונים התחלתיים זהים לחלוטין, גם בין המנגנונים וגם לפני כל הרצה).

ולהלן התוצאות :

ממוצע זמן הריצה עד להתכנסות במנגנון שקיבלתי מראש עמד על כ-3.009043 שניות, ואילו ממוצע זמן הריצה עד להתכנסות במנגנון שפיתחתי עמד על 31.65289 שניות.

בחישוב של טבלת ניתוח שונות אנחנו מקבלים :

מקור השונות	SS	MS	DF	F*
טיפולים	28716.46	28716.46	1	465651.9
שגיאות	8.510372	0.061669	138	
סה"כ	28724.97		139	

לפי ערך ה-F הסטטיסטי שנובע מטבלת ניתוח השונות, גם עבור רמת מובהקות של 0.01% אנחנו דוחים את השערת האפס – ($F_{1,138,0.99995} = 21.03875119$) משמע שתי התוחלות שיצאו לנו לא שוות באופן מובהק. לצערנו, ממוצע זמן הריצה של המנגנון עם ה-GPU איטי בהרבה מאשר המנגנון הבסיסי.

3.3 מסקנות והצעות

1. מנגנון הקומפילציה החדש שנבנה היה איטי בהרבה מאשר המנגנון שהתקבל בתחילת הקורס. עיקר הבעיה נובעת מפיתוח של "תחום ההגדרה" לכל חוק עד כאשר ואין הוספות יותר של איברים למבני הנתונים של הפרדיקטים.
2. ה-GPU משפר זמני ריצה בקטעי קוד בהם מבוצעת פעולה חישובית מורכבת. עבור הפרויקט, זמני הריצה שופרו בממוצע יותר מ-99% מזמן הריצה הבסיסי.
3. כל אופטימיזציה שניסיתי לבצע בשביל לשפר זמני ריצה (גם כאלו שלא הוצגו בגלל נפילתן בזמן הפרויקט), במקום לשפר רק הזיקו לזמן הריצה.
4. קיימת אופציה לממש פונקציות ב-OpenCL בגישה לה אני קורא "הגישה הדו שלבית", שלצערי לא היה לי זמן לעבוד עליה, ובה אנחנו מפרקים את הפרדיגמה שצוינה בעמוד 21 לשתי פונקציות שרצות אחת אחרי השנייה, אולם הן פחות מבזבזות זמן ריצה ומנצלות את המשאבים בצורה טובה יותר. הייתי שמח לנסות ולעבוד בגישה זו כדי לנסות אף לשפר יותר את זמן הריצה של "אלגוריתם תחום ההגדרה".

5. הייתי מציע לממש למעשה את כל הקוד המקומפל ב-OpenCL כדי לראות האם זמני הריצה ישופרו. כדי להקל על הליבות יש למיין את מבני הנתונים, ולבצע חיפוש בינארי שהוכיח יעילות טובה יותר בעת מימוש פונקציית Join (Diamos, 2012).

3.4 קוד הפרויקט

את קוד הפרויקט ניתן למצוא תחת הקישור הבא ב-GitHub, או בחיפוש ב-GitHub של GAPlus. הקוד כתוב ב-Python 3.4 אולם קיימת תיקייה ב-repository בשם py27 המכילה את אותו הקוד אולם מותאם ל-Python 2.7.

הקישור ל-repository : **https://github.com/barboko/GAPlus.git**

4. ביבליוגרפיה

Diamos, G' W1) .'st Feb 2012 .(Efficient relational algebra algorithms and data structures for GPU.14 .

Matthew, S .(2012) .'*OpenCL in Action* .Manning Publications CO.

Subrahmanian V.S., C' M .(2010) .'Using GAPs to solve social network optimization problems.52 .