אוניברסיטת אריאל בשומרון
המחלקה להנדסת חשמל ואלקטרוניקה

האצת אלגוריתם סריקה בשלב ההסקה עם עיבוד מקבילי בGPU

פרויקט גמר המהווה חלק מהדרישות לתואר B.Sc.

מוגש ע"י:

נריה אליה

ת.ז. : 206452716

מנחה אקדמי: פרופ' יואל רצאבי

אוקטובר 2025

תקציר

בפרויקט הזה בוצע מימוש ואופטימיזציה לאלגוריתם cuconv[1] להאצת חישובי קונבולוציה בGPU בעזרת CUDA, וגם עם הCPU עבור שלב ההסקה בCNN.

המימוש הראשוני על הGPU, שהתבסס על שני קרנלים מפוצלים, נותח באמצעות Nsight Compute.
הניתוח זיהה חוסר יעילות בגישה לזיכרון הDRAM ואי שימוש בזיכרון משותף.

בעקבות הממצאים, פותחה גרסה מאוחדת לGPU המשתמשת בקרנל יחיד, זיכרון משותף וטעינת נתונים יעילה כדי להפחית את צווארי הבקבוק.

כצפוי, בדיקות השוואתיות הראו יתרון לGPU על פני הCPU (עד פי 9508).

הגרסה המאוחדת הייתה מהירה יותר בקונבולוציות סטנדרטיות (כמו 3x3), והגרסה המקורית (המפוצלת) שמרה על יתרון בפילטרים של 1x1 בזכות טיפול נקודתי למקרה הזה.

הכרת תודה

לפרופ׳ יואל רצאבי המנחה האקדמי, על ההדרכה המקצועית, ההכוונה המדויקת והעברת הידע שסייעו לי בביצוע ובהבנת הפרויקט.
תודה על התמיכה, הסבלנות ועל הנכונות הבלתי מתפשרת להקדיש מזמנו בכל שלב בתהליך העבודה, גם בשאלות המורכבות והמתישות ביותר.

תוכן עניינים

1. מבוא..5

1.1. תיאור כללי של רשתות נוירונים קונבולוציוניות...5

1.1.1. חשיבות המהירות בשלב ההסקה בביצועי רשתות נוירונים.......................................5

1.1.2 הצורך באופטימיזציה של חישובי קונבולוציה 6

1.2. מבוא לGPU..7

1.2.1. מבוא לCUDA...8

1.3. אלגוריתם CUCONV[1]...9

2. מטרת הפרוייקט 13

2.1. דרישות מערכת...13

3. תיאור מערכת 14

3.1. תכנון חומרה...14

3.1.1 תכונות הרכיב 14

3.2. תיאור תוכנה..15

3.2.1. רכיבי תוכנה 15

3.2.2. תאימות בין הרכיבים 15

3.2.3. דרישות ותלויות 15

3.2.4. מבנה התוכנה...16

4. מימוש המערכת..17

4.1. קובץ main.cpp עבור הGPU..17

4.2. מבנה ספריית cuconv והקשר בין cuconv_api.h לcuconv_lib.cu.........................24

4.3. cuconv_lib.cu 26

4.3.1. חיבור הAPI...26

4.3.2. קרנל scalar_prods_kernel...31

4.3.3. הקרנל sum_kernel..35

4.4. הרצת טורית על הCPU...37

4.5. ניתוח קרנלים באמצעות NSIGHT COMPUTE...47

4.6. גרסה מאוחדת של האלגורתם 50

4.7. ניתוח הקרנל המאוחד באמצעות NSIGHT COMPUTE 54

5. ניסויים 56

5.1. קונפיגורציות המאמר...57

5.2. קונפיגורציות להמחשת יתרון הGPU מול הCPU...58

6. מסקנות...59

7. רשימות 60

7.1. רשימת איורים 60

7.2. רשימת טבלאות..60

8. מקורות..61

8.1. מקורות ספרותיים 61
8.2. מקורות עזר חזותיים 61
9. נספחים 62
9.1 קוד מקור לפרוייקט 62

1. מבוא

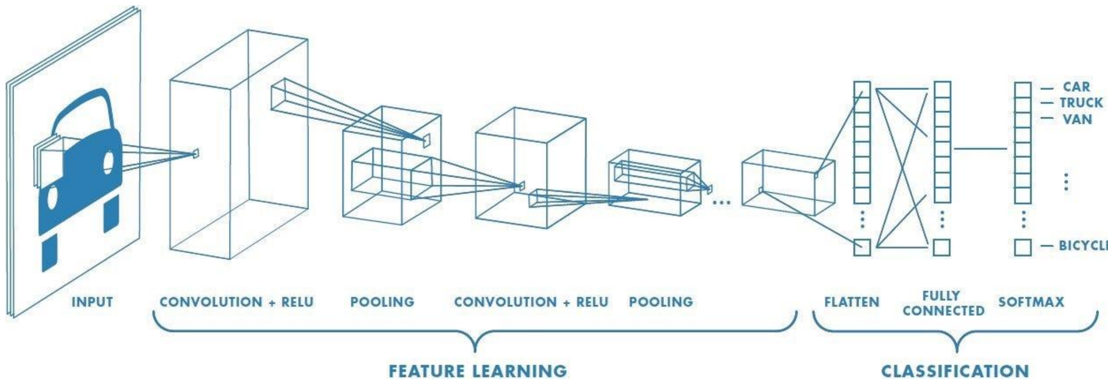
1.1. תיאור כללי של רשתות נוירונים קונבולוציוניות.

רשתות נוירונים קונבולוציוניות CNN הן אחת מהטכנולוגיות שמאפשרות למחשבים לזהות תמונות, להבין פנים, ואפילו לאבחן מחלות. אם פעם מחשבים ראו תמונות כסתם מספרים, היום הם מסוגלים לזהות תבניות קווים, טקסטורות וצורות עד שהם מבינים מה מופיע בתמונה.

בתחילת התהליך, הרשת הזאת מזהה אלמנטים פשוטים כמו קווים וקצוות.
ככל שהיא מעמיקה, היא מתחילה להבחין בפרטים מורכבים יותר, כמו חלקים של אובייקטים, ובסופו של דבר היא מבינה את התמונה כולה. בדיוק כמו שמוח לומד לזהות פנים של אנשים עם הזמן, אז רשתות CNN משתפרות ככל שהן נחשפות ליותר דוגמאות.

היתרון הגדול של CNN הוא שהיא לא צריכה שיגידו לה מה לחפש.
במקום שנגדיר לה כל פרט מראש, היא לומדת לבד מתוך אינספור תמונות. השימושים של CNN נמצאים בכל מקום.
נגיד כשהטלפון מזהה את פנים ופותח את הנעילה, זאת בעצם רשת CNN שעובדת מאחורי הקלעים. ברכבים אוטונומיים, CNN מנתחת את הדרך, מזהה תמרורים ומכשולים, ועוזרת למכונית "לראות".

בעולם הרפואה, היא מסייעת באיתור מחלות על סמך צילומי רנטגן וMRI.
אפילו בחיפוש תמונות בגוגל או בGoogle Photos, המערכת מזהה חפצים ואנשים בתמונות.



איור 1 : מבנה כללי של CNN

1.1.1. חשיבות המהירות בשלב ההסקה בביצועי רשתות נוירונים.

שלב ההסקה ברשתות CNN הוא השלב שבו הרשת מסווגת נתונים חדשים באמצעות מודל שכבר אומן. בשלב זה, הרשת מקבלת כקלט feature maps שהן תוצר של עיבוד הקלט הגולמי בשכבות קודמות.

לאחר שלב הקונבולוציה, המידע המופשט עובר דרך שכבות fully connected שממירות אותו לתחזית סופית. כמות החישובים הגדולה שנדרשת בשלב ההסקה הופכת אותו למאתגר, ולכן נעשה שימוש בכרטיסי GPU שמאפשרים ביצוע מקבילי של חישובים רבים.

המהירות והדיוק בשלב הזה חיוניים במיוחד ביישומים כמו זיהוי תמונות, ונהיגה אוטונומית.
עיכוב, אפילו קצר עלול להוביל לתוצאות לא רצויות.

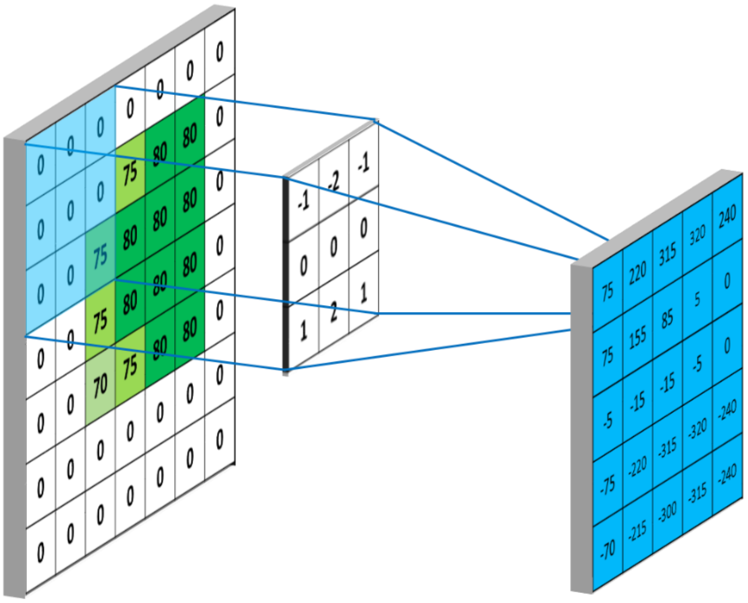
1.1.2 הצורך באופטימיזציה של חישובי קונבולוציה

מהי בכלל קונבולוציה?

באופן מתמטי קונבולוציה היא שילוב של שתי פונקציות ליצירת פונקציה חדשה באמצעות הזזה וכפל. בתהליך הקונבולוציה ברשתות CNN, מתבצעת "סריקה" של התמונה באמצעות פילטרים .

כל פילטר הוא בעצם מטריצה קטנה של משקולות שנעה על פני כל האזורים בתמונה כדי להפיק מידע רלוונטי כמו קצוות, מרקמים, וצורות בסיסיות.

התוצאה של התהליך הזה נקראת feature map והיא מדגישה אזורים משמעותיים בתמונה.
השלב הזה מאפשר לרשת CNN ללמוד מאפיינים ויזואליים ולהשתמש בהם מאוחר יותר בשלב ההסקה.



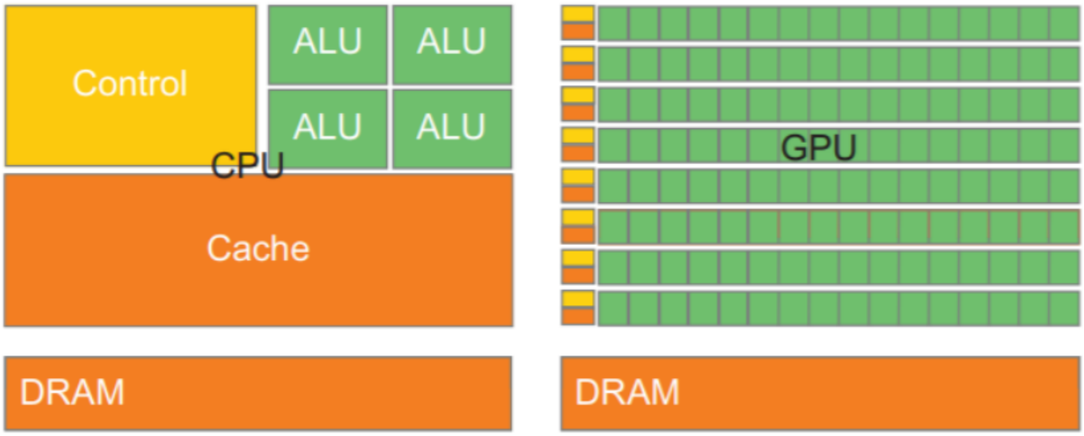
איור 2 : המחשה של פעולת הקונבולוציה בתמונת קלט

1.2. מבוא לGPU

GPU הוא סוג של מעבד שנמצא בכרטיס הגרפי של המחשב, והתפקיד המרכזי שלו הוא לבצע חישובים בצורה מקבילית. כלומר, הרבה חישובים בבת אחת.
אחד היתרונות הגדולים שלו הוא רוחב פס גבוה מאוד של זיכרון, הרבה יותר מזה של הCPU (המעבד המרכזי של המחשב), מה שמאפשר לו לטפל בכמויות גדולות של מידע בצורה מהירה ויעילה במקביל.

הGPU בנוי אחרת לגמרי מהCPU, במקום כמה ליבות חזקות כמו שיש לCPU הGPU מורכב מאלפי ליבות קטנות שפועלות בו זמנית במהירות נמוכה יותר לעומת הCPU.
והמבנה הזה נותן לו יתרון אדיר במשימות שבהן צריך לבצע המון פעולות דומות על הרבה מאוד נתונים כמו למשל בעיבוד תמונה, וידאו, או רשתות נוירונים בלמידת מכונה.

הCPU נועד לביצוע רצפים מורכבים של פקודות במהירות ודיוק, והוא טוב מאוד בהפעלה של כמה עשרות ת׳רדים במקביל.
לעומת זאת, הGPU נועד להפעיל אלפי ת׳רדים במקביל ,גם אם כל אחד מהם רץ קצת יותר לאט ובסופו של דבר, זה מה שמאפשר לו להגיע לתפוקה כוללת הרבה יותר גבוהה במצבים מסוימים.



איור 3: מבנה מעבד GPU לעומת CPU

1.2.1. מבוא לCUDA

CUDA היא ארכיטקטורה שפיתחה חברת NVIDIA במטרה לאפשר כתיבה והרצה של תוכניות מחשב ישירות על גבי הGPU כלומר, במקום שהמעבד המרכזי הCPU יעשה את כל העבודה, אפשר "לגייס" גם את המעבד הגרפי GPU .

הכתיבה בקוד מתבצעת בשפת CUDA for C שהיא בעצם הרחבה לשפת C הרגילה, עם תוספות שמאפשרות שליטה בGPU.

מה שמיוחד בCUDA זה שהיא נותנת למתכנת גישה ישירה לזיכרון של כרטיס המסך ,וגם לסט הפקודות שמריצות את הקוד על הליבות של הGPU .
כל זה נועד בעיקר לטפל במשימות עיבוד כבדות שמצריכות המון חישובים במקביל כמו בעיבוד תמונה, למידת מכונה, סימולציה של פיזיקה ועוד.

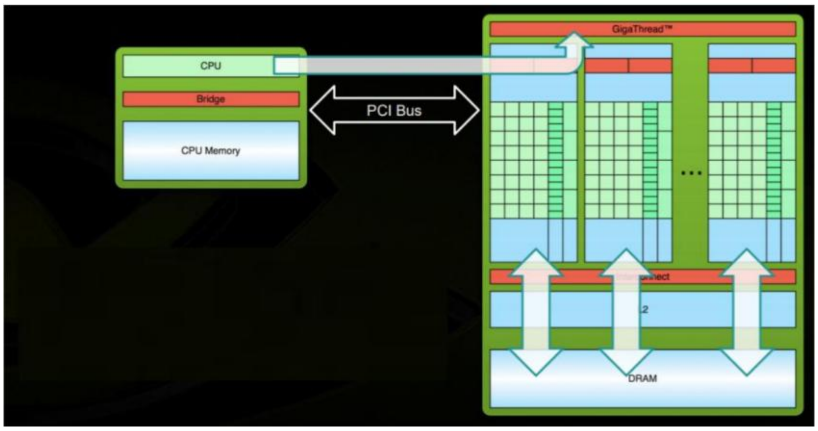
בתכנות עם CUDA יש שני מושגים עיקריים שצריך להכיר :

- Host - הכוונה למחשב הראשי, כלומר הCPU והזיכרון הרגיל שלו.
- Device - הכוונה לGPU ולזיכרון של כרטיס המסך.

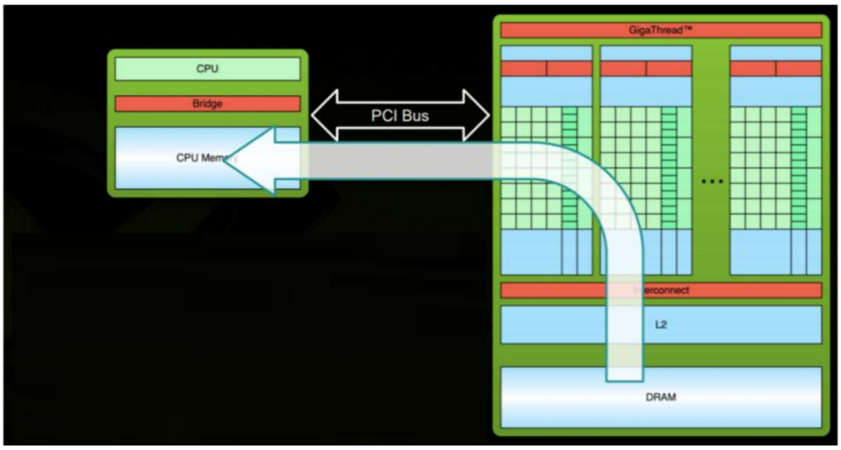
איך זה עובד בפועל?

1. קודם כל מעבירים את הנתונים מהHost אל הDevice כלומר מהזיכרון של המחשב לזיכרון של הGPU.

2. אחר כך טוענים את הקוד שצריך לרוץ על הGPU, והוא מתחיל לעבוד עליו.



3. העתקת תוצאות הDevice לHost.



4. מחיקת זיכרון בDevice.

1.3. אלגוריתם CUCONV[1]

המאמר מציג את אלגוריתם cuconv שהוא בעצם מימוש של פעולת הקונבולוציה עבור רשתות CNN על הGPU, שמטרתו לשפר היעילות של הגישה לזיכרון ולהשוות ביצועים מול ספריית cuDNN של NVIDIA.

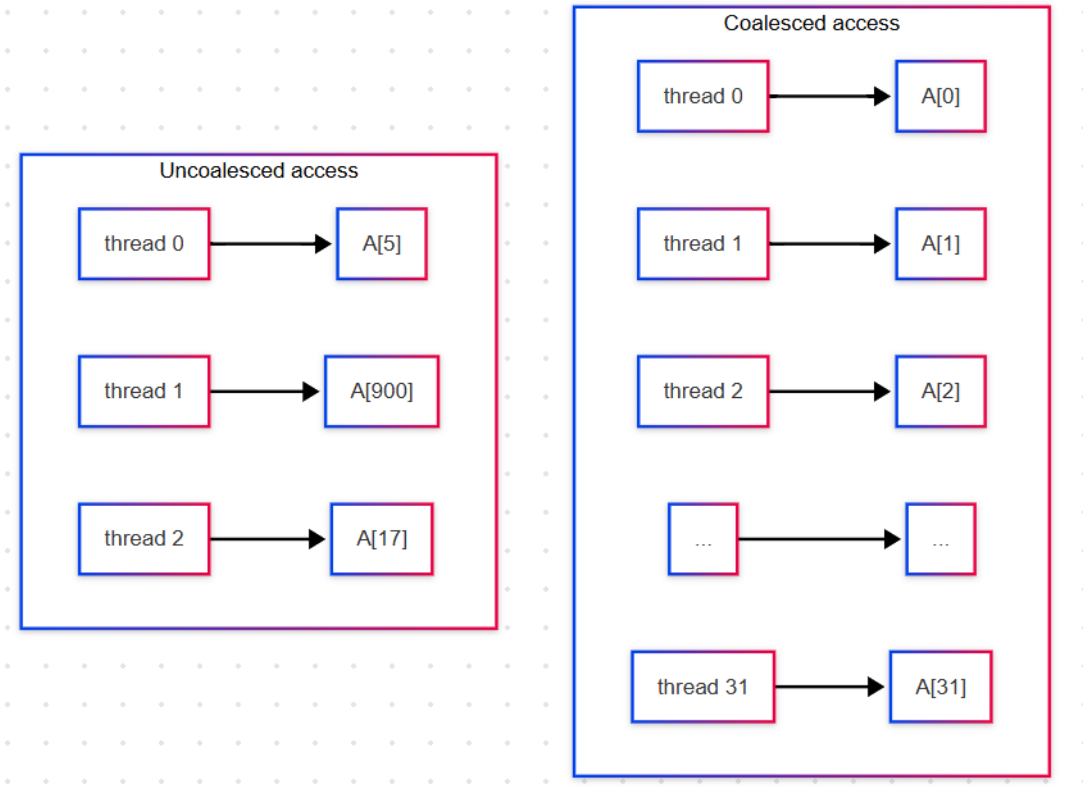
בפרויקט הזה לא נעשה שימוש ישיר בקוד של המאמר, אלא יושם מימוש עצמאי שהתבסס על הרעיונות והעקרונות שתוארו בו (לא מצורף במאמר קוד מימוש).

תחילה פותחה גרסה בסיס שבה פעולת הקונבולוציה חולקה לשני קרנלים נפרדים, ולאחר מכן פותחה גרסה מאוחדת שמשלבת את השלבים לקרנל אחד לאחר סקירת ביצועים בNsight Compute לגרסה המקורית.

השלבים והתהליכים המרכזיים של אלגוריתם cuconv :

1. שימוש בMemory Coalescing :

המימוש מתמקד בcoalesced accesses בכדי לנצל בצורה טובה את רוחב הפס של הזיכרון הגלובלי (Global Memory) של הGPU משמעות הדבר היא שת'רדים רצופים מבצעים גישות לזיכרון רציף(כלומר לכתובות אחת ליד השניה), וככה מונעים בזבוז רוחב פס ומקטינים השהיות.



איור 4 : השוואה בין גישות זיכרון Coalesced לעומת Uncoalesced

2. האלגוריתם מחולק לשני שלבי חישוב נפרדים:

א. חישוב מכפלות סקלריות: בשלב הזה מחושבות כל המכפלות הסקלריות בין שורות העומק (כלומר ערוצים, ציר Z) של הקלט (שהוא בעצם התוצר של השכבות הקודמות ברשת) לבין שורות העומק של הפילטרים, השלב הזה יוצר מטריצות החלקיות המכילים את תוצאות ביניים.

החישוב של המטריצה החלקית:

$$P = \sum_{C=0}^{C-1} X\left[c, y - \text{pad}_h + f_y(k), x - \text{pad}_w + f_x(k)\right] W\left[m, c, f_y(k), f_x(k)\right]$$

P - Partial Result[m, k, y, x] - המטריצות החלקיות.
X – הוא הקלט עצמו בערך ספציפי.
C - האינדקס של הערוץ בקלט.
K – הוא המספר הסידורי שממפה כל מיקום **בפילטר** למספר יחיד.
y - מיקום הפלט בגובה.
x- מיקום הפלט ברוחב.
Pad – כמה "ריפוד" התווסף לכל שורה או עמודה (כלומר אפסים למסגרת).
W- המשקל של הפילטר במיקום הספציפי (כלומר הערך של הפילטר).
m -אינדקס הפילטר.
$f_y(k), f_x(k)$ – אלו פונקציות שמחזירות את השורה והעמודה של מיקום הפילטר מתוך k.

גודל כל מטריצה חלקית היא:

$$H_p = H - H_f + 1$$
$$W_p = W - W_f + 1$$

H, W – הממדים של הקלט (האורך והרוחב).

H_p, W_p – הממדים של המטריצה החלקית (האורך והרוחב).

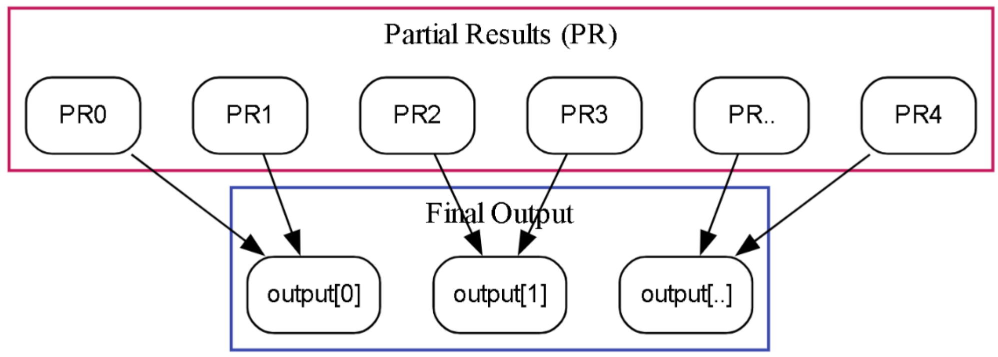
H_f, W_f – הממדים של הפילטר (אורך ורוחב).

בשלב הזה נוצרת קבוצת המטריצות החלקיות בגודל $H_p \times W_p$, וסה"כ יש:

$$H_f \times W_f \times N \times M$$

כל אחת מהמטריצות החלקיות האלה מייצגת את התוצאה של המכפלה הסקלרית בין האזור בקלט לבין השורת העומק בפילטר, עבור תמונה אחת ופילטר אחד מתוך M הפילטרים.

ב. שקלול תוצאות: בשלב זה, המטריצות החלקיות שנוצרו בשלב הראשון מאוחדים יחד כדי לקבל את תוצאות הקונבולוציה הסופיות.



איור 5 : שילוב מטריצות חלקיות לקבלת פלט סופי בתהליך הקונבולוציה

גודלן של המטריצות הסופיות לאחר הסכימה:

$$H_{out} = \frac{H + 2 \cdot pad_y - H_f}{stride_y} + 1$$

$$W_{out} = \frac{W + 2 \cdot pad_x - W_f}{stride_x} + 1$$

H, W - מימדי הקלט.

H_f, W_f- מימדי הפילטר.

H_{out}, W_{fout} - מימדי הפלט לאחר הסכימה (רוחב ואורך).

pad_y, pad_x – הוספת 0 לכל צד בציר בקצה.

$stride_y, stride_x$– גודל הצעד (קפיצה) של הפילטר.

$Hout, Wout$- מימדי מטריצות המוצא.

3. **ניצול יעיל של זיכרון משותף (shared memory):**

באלגוריתם cuconv, שורות העומק של הפילטר נטענות קודם מהזיכרון הגלובלי האיטי של הGPU ואז נשמרות בתוך הזיכרון המשותף הפנימי של כל הליבות CUDA שקיימות באזור הזה.

הפעולה הזאת מתבצעת רק פעם אחת בתחילת הריצה של הבלוק, וזה מבטיח זמינות מהירה יותר של נתוני הפילטר לכל הת׳רדים שנמצאים באותו הבלוק.
וככה, כל ת׳רד יכול לגשת לערכים (במקרה שלנו שורות העומק של הפילטר הרלוונטיות) מתוך הזיכרון המשותף במקום לבצע גישה חוזרת לזיכרון הגלובלי.

הגישה הזאת מאפשרת שימוש חוזר יעיל יותר בנתונים של הפילטר, במיוחד כשמתבצעות הרבה קונבולוציות על אותו סט של נתונים.
בגלל ההעתקה לזיכרון המשותף יש האצה של זמן ההרצה וניצול טוב יותר של רוחב הפס הפנימי של הGPU.

2. מטרת הפרוייקט

מטרת הפרויקט היא לממש את האלגוריתם cuconv כפי שהוצג במאמר[1]

ולבצע Profiling לקרנלים באמצעות Nsight Compute ולשפר את הקרנל בהתאם לממצאים שהתקבלו.
ולאחר מכן תתבצע השוואה מול הרצה על CPU כדי להמחיש את היתרון של GPU בחישובים מקביליים ביחס לCPU.

2.1. דרישות מערכת

כדי לבצע את הניסויים ולממש את המערכת נצטרך את הדברים הבאים:

1. GPU- RTX 3060 Laptop 130w TPG
2. NVIDIA Nsight Compute 2024.3.2
3. CPU - AMD Phenom II X6 1055T

3. תיאור מערכת

המערכת הינה סדרת ניסויים המבוססים על האלגוריתם cuconv.
אשר רץ באופן מקבילי, ושינוי הקוד CUDA בהתאם לדוחות ביצועים מNsight Compute
ולאחר מכן השוואתם לביצועים על הCPU .

3.1. תכנון חומרה

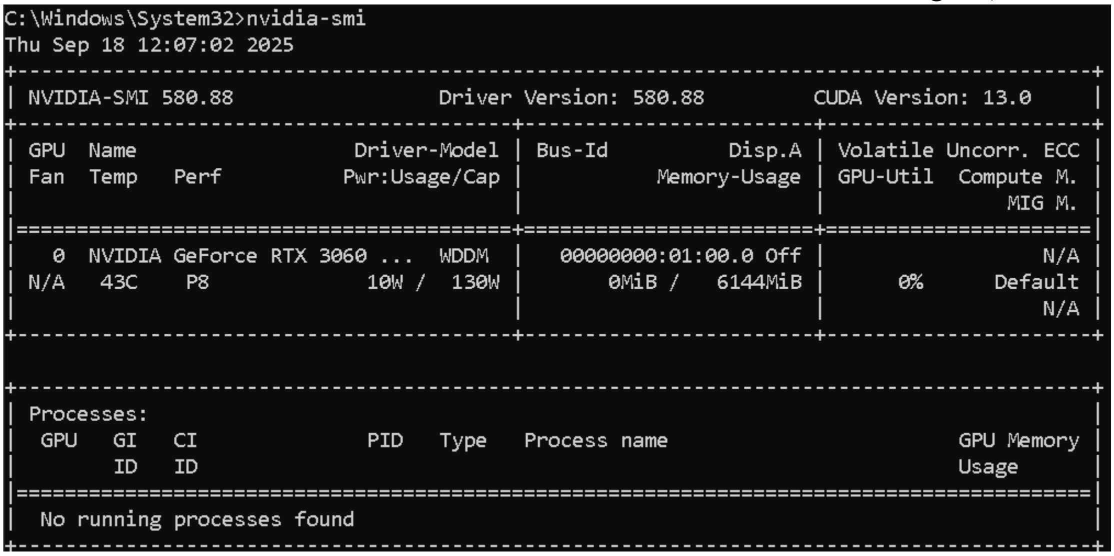
המערכת ממומשת על גבי GPU מדגם RTX 3060 של חברת NVIDIA .

רכיב RTX 3060 הוא רכיב GPU (Graphics Processing Unit) השייך לחברת Nvidia .

3.1.1 תכונות הרכיב

- ארכיטקטורה : שבב GA106 מבוסס Ampere
- כמות ליבות CUDA : 3,840
- תדרי עבודה : 1425 MHz (Base) 1425 / 1702 MHz (Boost)
- זיכרון : 6GB מסוג GDDR6
- רוחב ממשק זיכרון (Bus Width): 192 bit
- רוחב פס לזיכרון: עד כ 336GB/s
- צריכת הספק מקסימלית : TPG 130W
- חיבור ללוח האם : PCI Express (מולחם, לא מודולרי).
- יכולות עיבוד Compute Capability 8.6 : Tensor Cores, RT Cores וכו.

ה GPU נקי מ Processing :



```
C:\Windows\System32>nvidia-smi
Thu Sep 18 12:07:02 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.88                 Driver Version: 580.88         CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3060 ...  WDDM  |   00000000:01:00.0 Off |                  N/A |
| N/A   43C    P8             10W /  130W |       0MiB /   6144MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

איור 6: הרכיב GPU שרץ על Windows הוא Dedicated GPU .

3.2. תיאור תוכנה

המערכת מממשת אלגוריתם קונבולוציה בעזרת CUDA, שנמצאת בספרייה cuconv_lib.cu הנקשרת לאפליקציית ההדגמה.
האלגוריתם ממומש בשני קרנלים:

1. scalar_prods_kernel- חישוב מכפלות סקלריות לאורך עומק הקלט (הערוצים של פיקסל יחיד) לבין שורות העומק של הפילטרים.
2. sum_kernel- סכימת התוצאות החלקיות לכל פילטר ליצירת מפת פלט סופית בגודל $Ho \times Wo$.
   במקרים של קונבולוציה בגודל 1×1 שלב הסכימה מתבטלת.

3.2.1. רכיבי תוכנה

cuconv_lib.cu - כולל את הקרנלים עצמם (scalar_prods_kernel, sum_kernel) ואת עטיפות הHost האחראיות להגדרת גריד/בלוק, הקצאות זיכרון.

cuconv_api.h- הוא ממשק "ציבורי" פשוט להפעלת האלגוריתם:

cuconv_conv_forward- פונקציה שאחראית על ביצוע הקונבולוציה בפורמט NCHW עם פרמטרים עבור קלט, פילטרים, Padding וStride.

main.cpp - קובץ הדגמה האחראי על יצירת נתוני הקלט והפילטרים, קריאה לAPI של הספרייה, וקריאה להרצת האלגוריתם בפועל.

3.2.2. תאימות בין הרכיבים

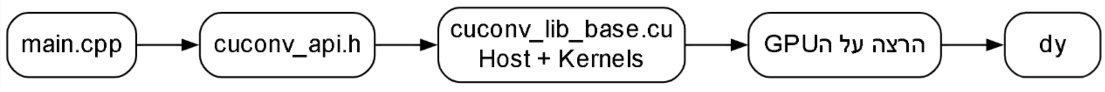
- הקרנלים בcuconv_lib.cu ממומשים ככה שכל הקריאות זיכרון יהיו מאוחדות (coalesced).
- הטיפול בPadding נעשה לוגית, כשגישה מחוץ לתחום מתורגמת לערך אפס.

3.2.3. דרישות ותלויות

- CUDA 11 ומעלה.
- דרייבר תואם לNVIDIA.
- Nsight Compute לביצוע ניתוח ביצועים ופרופיילינג.

3.2.4. מבנה התוכנה

הדיאגרמה הזאת מתארת את מבנה הזרימה של התוכנה, ומציגה כיצד הנתונים עוברים בין חלקי התוכנה:



איור 7: תהליך זרימת התוכנה

1. main.cpp - זהו הקובץ הראשי בפרויקט. הוא האחראי על הכנת הנתונים (כמו קלט, פרמטרים וקונפיגורציות) ועל קריאה לפונקציות מהAPI של הספרייה.

2. cuconv_api.h - קובץ הכותרת (הHeader) שמגדיר את הממשק הציבורי של הספרייה. יש בו את הצהרה על פונקציות הHost שנמצאות בספריית CUDA.

3. Host + Kernels , cuconv_lib.cu - זה הקובץ המרכזי שמממש את האלגוריתם.

   - חלק הHost כולל קוד שרץ על המעבד (CPU) ומכין את הזיכרון בGPU, מעתיק נתונים, ומבצע קריאות לקרנלים
   - חלק הKernels כולל את הקוד שרץ על הGPU עצמו ומבצע את פעולות החישוב.

4. הרצה על הGPU- השלב זה מייצג את ביצוע הקרנלים על גבי הGPU בפועל. כאן מתבצעת הפעולה המקבילית על אלפי ליבות חישוב.

5. dy - זה מצביע לפלט החישוב בזיכרון של הGPU. הפלט נשמר בפורמט NCHW, שזה הפורמט שהוא די נפוץ ברשתות CNN .

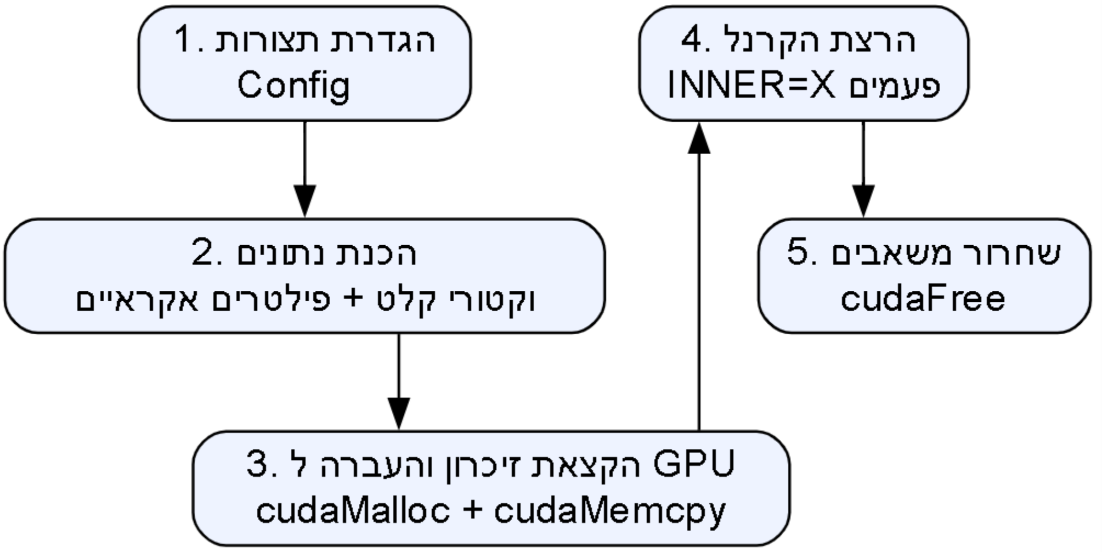
4. מימוש המערכת

4.1. קובץ main.cpp עבור הGPU

הקובץ main.cpp מהווה בעצם את נקודת ההפעלה של המערכת והמטרה העיקרית שלו היא להריץ סדרת ניסויים לביצוע פעולות קונבולוציה על הGPU כשהוא גם מבצע מדידת זמן הריצה במקביל.
זהו בעצם קובץ עצמאי שמרכז את כל שלבי ההרצה : יצירת קלטים, קריאה למימוש בGPU, מדידה והצגת תוצאות. התוכנית עצמה בנויה ככה שתהיה אפשרות לבחון מספר קונפיגורציות קבועות מראש שהם בדר״כ קיימות בשלבי ההסקה ברשתות CNN נפוצות.
כגון : Resnet50, Googlenet וכו'.

עבור התצורות האלו נוצרים וקטורי קלט ומשקולות אקראיים, ומבוצעת קונבולוציה על ידי הקריאה לממשק cuconv_conv_forward.

מבנה הפעולה המרכזי :



איור 8 : תהליך פעולת קובץ main.cpp

1. **הגדרת תצורות**
כל תצורה (Config) כוללת פרטים על גודל התמונה, מספר הפילטרים, גודל הפילטר, ועומק הקלט.
ההרצה מוגבלת אך ורק למקרים בהם, (Batch)N=1, stride=1, groups=1 והpadding סימטרי.

2. **הכנת נתונים**
עבור כל תצורה, נוצרים וקטורים של קלט ופילטרים, עם ערכים אקראיים אחידים לטובת השוואה הוגנת בין ריצות.

3. **הקצאת זיכרון והעברה לGPU**
מוקצה זיכרון בGPU לקלט, פילטרים ופלט.
הנתונים מועברים מהHost לDevice.

4. **הרצת הקרנל**
הקרנל מופעל מספר פעמים (INNER = X) ברציפות, והממוצע מוצג בסיום כל תצורה.

5. **שחרור משאבים**
לאחר ההרצה, הזיכרון משוחרר .

הגדרת התצוגה:
הוגדר מבנה בשם Config שמטרתו לרכז בצורה מסודרת את כל הפרמטרים הנדרשים להרצת ניסוי יחיד.
שימוש במבנה זה מאפשר לנהל בקלות סדרה של תצורות שונות מבלי לפזר משתנים רבים מידי בקוד, וככה בעצם לשמור על בהירות וקריאות של הקוד .

```
struct Config {
    const char* table;
    const char* label;
    int N, H, R, M, C;
};
```

פירוט השדות במבנה:

table – זה בעצם מצביע למחרוזת המזהה את סוג הפילטר או גודל הקונבולוציה.
label – תווית קצרה (כמו "A", "B", "C") שמשמשת לזיהוי של תצורה.
N – מספר הדוגמאות עבור ההרצה, כאן מוגבל ל1.
H – גובה התמונה (כאשר הרוחב W שווה לH), כלומר תמונה ריבועית.
R – גודל של הפילטר, כלומר הרוחב והאורך שהם גם ריבועיים.
M – מספר הפילטרים (מספר הערוצים בפילטר).
C – מספר הערוצים בקלט .

באמצעות המבנה הזה ניתן להגדיר בצורה ברורה וקומפקטית את כל מאפייני הניסוי, ולאפשר לקוד הראשי לעבור על רשימת תצורות מוכנה מראש.

תוכנית הMAIN:

```
int INNER = X;
```

הגדרת משתנה מסוג INT בגדול X.
גודל זה מתאר את כמות הפעמים שהאלגורתם ירוץ על המכשיר, כלומר הGPU .

```
vector<Config> cases = {
        {"T3-1x1","A", 1,  7,1,  256, 832},
        {"T3-1x1","B", 1, 14,1, 1024, 256},
        {"T3-1x1","C", 1, 27,1,  256,  64},
        {"T4-3x3","A", 1,  4,3,  384, 192},
        {"T4-3x3","B", 1, 13,3,  384, 384},
        {"T5-5x5","A", 1,  7,5,  128,  348},
};
```

במערך cases מוגדרות שישה תצורות ניסוי, שכל אחת מהן משלבת שם פילטר (1×1, 3×3, 5×5), תווית זיהוי (A,B,C), כמה תמונות נכנסות בכל הרצה (N) גודל תמונה (H), גודל פילטר (R), מספר פילטרים (M) ומספר ערוצים בקלט (C).
הערכים נבחרו ככה שהם ישקפו תרחישים אופייניים בקונבולוציות של רשתות CNN כמו שמתואר במאמר[1] .

ביצוע סדרת התצורות:

בקטע קוד הזה מתבצעת הקריאה לריצה שלפני הריצה על הGPU.
כל התצורות נשלחות בצורה מסודרת לפונקציית run לפי הגדרת Config עד שנגמר התצורות לשליחה.

```
for (const auto& cfg : cases) {
        run(cfg, INNER);
}
return 0;
```

הפונקציה run היא ההרצה של ניסוי בודד.

```
static float run(const Config& cfg, int inner_runs)
```

היא מקבלת תצורה אחת ואת מספר האיטרציות שצריך לבצע, ומכינה את כל מה שנדרש כדי להריץ את הקרנל: חישוב גדלים, הקצאת זיכרון, יצירת נתוני קלט אקראיים.

שליפת פרמטרי התצורה והגדרת פרמטרי הקונבולוציה:

```
int N = cfg.N, C = cfg.C, H = cfg.H, W = cfg.H;
```

כאן נשלפים מתוך התצורה הערכים הבסיסיים: מספר הדוגמאות (N) .
מספר הערוצים בקלט (C) .
גובה התמונה (H) והרוחב (W) כאשר במקרה שלנו מוגדר שהרוחב שווה לגובה.

```
int M = cfg.M, R = cfg.R, S = cfg.R;
```

כאן נשלפים מספר הפילטרים (M) וגודל הפילטר (R) הערך S מוגדר שווה לR.
כלומר הפילטר ריבועי (גובה ורוחב שווים).

```
int pad = (R - 1) / 2, stride = 1, dil = 1, groups = 1;
```

כאן מוגדרים פרמטרים נוספים להפעלת הקונבולוציה:

- pad- חישוב הpadding ככה שהקונבולוציה תהיה בעצם "סימטרית" ותשמור על ממדי הקלט כמעט זהים.
- stride = 1 - ההזזה של הפילטר אל מול הפיקסל קלט מתבצעת צעד אחד בכל פעם.
- dil = 1 - ללא דילול (כלומר צפיפות פילטר אחידה).
- groups = 1 - כל הערוצים מחוברים יחד ללא חלוקה לקבוצות

חישוב גדלי המערכים בקונבולוציה:

```
size_t xin = (size_t)N * C * H * W;
```

חישוב מספר הערכים בקלט: מספר דוגמאות (N) כפול מספר ערוצים (C) כפול גובה (H) כפול רוחב (W) .

```
size_t win = (size_t)M * C * R * S;
```

חישוב מספר הערכים במשקולות: מספר פילטרים (M) כפול מספר ערוצים (C) כפול גובה פילטר (R) כפול רוחב פילטר (S) .

```
int Ho = H + 2 * pad - R + 1;
```

מחשב את גובה הפלט לאחר הוספת padding ויישום הפילטרים.

```
int Wo = W + 2 * pad - S + 1;
```

מחשב את רוחב הפלט באותו צורה.

```
size_t yout = (size_t)N * M * Ho * Wo;
```

מחשב את מספר הערכים בפלט: מספר דוגמאות כפול מספר פילטרים כפול ממדי הפלט החדשים.
בקטע הזה מתבצעת ההכנה של נתוני הקלט והפילטרים לפני ההרצה על הGPU.
נוצרות שלוש מערכות נתונים בHost (בCPU):

```
vector<float> hx(xin), hw(win), hy(yout, 0.0f);
fill_random(hx);
fill_random(hw);
```

hx (xin) – זהו המערך שמייצג את נתוני הקלט, בגודל התלוי במספר הדוגמאות, הערוצים והמידות המרחביות.

hw (win) – מערך שמייצג את משקולות הפילטרים, בגודל התלוי במספר הפילטרים, ערוצי הקלט וגודל הפילטר.

hy (yout, 0.0f) – מערך לפלט של ההרצה, מאתחל את כולו לערכים אפסיים כדי לשמור על עקביות בניסויים.

לאחר יצירת המערכים של נתוני הקלט ומשקולות הפילטרים, שתי המערכים הראשונים (hx וhw) מתמלאים בערכים אקראיים אחידים באמצעות הפונקציה fill_random הבאה:

```
static void fill_random(vector<float>& v) {
    mt19937 rng(42);
    uniform_real_distribution<float> U(-1.f, 1.f);
    for (auto& x : v) x = U(rng);
}
```

נוצר גם שימוש בseed קבוע (42) שיבטיח שכל ריצה תחזור על אותם ערכים רנדומליים.

הקצאת זיכרון בGPU והעברת נתונים מהHost לDevice:

בקטע הקוד הבא, מתבצע השלב שבו מוגדר הזיכרון בGPU ומועברים אליו הנתונים מהHost .
השלב הזה מאפשר לקרנלים לרוץ על נתוני הקלט והפילטרים מתוך הזיכרון של הGPU .

```
float* dx = nullptr, * dw = nullptr, * dy = nullptr;
```

המצביעים מוגדרים כnull בהתחלה.
ובהמשך הם בעצם ישמשו כדי להחזיק את כתובות הזיכרון של הקלט (dx) והפילטרים (dw) והפלט (dy) על הGPU.

```
cudaMalloc(&dx, xin * sizeof(float));
```

הקצאת זיכרון בGPU עבור מערך הקלט, בגודל שמתאים למספר הערכים הנדרש (xin) .

```
cudaMalloc(&dw, win * sizeof(float));
```

הקצאת זיכרון עבור הפילטרים (המשקולות), בהתאם למספר הערכים שנדרש(win) .

```
cudaMalloc(&dy, yout * sizeof(float));
```

הקצאת זיכרון עבור הפלט, כלומר המקום שבו יאוחסנו תוצאות החישוב (yout) .

```
cudaMemcpy(dx, hx.data(), xin * sizeof(float),
cudaMemcpyHostToDevice);
```

העתקת הנתונים מהHost (זיכרון CPU) אל הזיכרון בGPU עבור הקלט.

```
cudaMemcpy(dw, hw.data(), win * sizeof(float),
cudaMemcpyHostToDevice);
```

העתקת הנתונים מהHost (זיכרון CPU) אל הזיכרון בGPU עבור הפילטר.

הוראה לביצוע קונבולוציה:

לאחר שהוקצו הזיכרונות והנתונים הועברו אל הGPU מתבצעת לולאה של הרצות חוזרות.
כל הרצה מפעילה מחדש את הקרנלים שמבצעים את פעולת הקונבולוציה על הנתונים שהוטענו מראש לGPU.

```
For (int I = 0; I < inner_runs; i++) {
cuconv_conv_forward(dx, dw, dy, N, C, H, W, M, R, S,
pad, pad, stride, stride, dil, dil, groups);
}
```

ניתן לראות שהלולאה רצה כמספר הפעמים שהיא קיבלה,
כדי שיתקבל זמן ביצוע מדויק יותר שמבוסס על כמות גדולה של ריצות על הGPU ולא מקרה יחיד שיכול להיות מושפע מגורמים אחרים.
ובכל איטרציה מופעלת מחדש הפונקציה cuconv_conv_forward (בהמשך יהיה פירוט על אופן פעולתה), שהיא בעצם המימוש של הקונבולוציה על הGPU .
הפרמטרים שנשלחו כוללים בתוכם את כתובות הזיכרון (dx dy dw) ואת כל ההגדרות והגדלים הרלוונטיים.

```
cudaFree(dx); cudaFree(dw); cudaFree(dy);
```

שחרור הזיכרון שהוקצה על הGPU עבור מצביעי הקלט, הפילטרים והפלט.
הפעולה הזאת מבטיחה שהזיכרון בGPU יחזור להיות פנוי.

4.2. מבנה ספריית cuconv והקשר בין cuconv_api.h לcuconv_lib.cu

הפרויקט בנוי סביב ספרייה משותפת בשם cuconv, שנועדה לבצע קונבולוציה על GPU באמצעות CUDA.
ממשק הAPI שבעצם מספקת הספרייה כולל את הפונקציה המרכזית cuconv_conv_forward שתוכננה ככה שתוכל להיקרא מתוך קוד בשפת C או C++, ותומכת בפורמט נתונים מסוג NCHW כמו שהוגדר לפני במבנה בשם Config .
והקובץ cuconv_api.h הוא מהווה רק כהצהרה על הפונקציה cuconv_conv_forward.
לא את מה שהיא עושה בפועל.
המימוש האמיתי על הGPU יקרה דרך קובץ cuconv_lib.cu אשר יפורט בהמשך.

זהו החלק בקוד שמטפל בהגדרות כלליות של הקובץ ובדרך שבה הספרייה תיוצא או תיובא בפרויקט.

```
#pragma once
#include <cstdint>
```

זאת פקודה ששומרת שהקובץ יהיה רק פעם אחת בזמן הקומפילציה, גם אם יש כמה #include אליו. זה בעצם מונע כפילויות ושגיאות קישור.
ובנוסף, ספריה להגדרת טיפוסי נתונים.

```
#ifdef CUCONV_EXPORTS
  #define CUCONV_API __declspec(dllexport)
#else
  #define CUCONV_API __declspec(dllimport)
#endif
```

בWindows צריך להבדיל בין מצב שבו הספרייה נבנית לבין מצב שבו רק משתמשים בה.
בשביל זה נכנס הקטע קוד הזה.
כי אם מוגדר כבר CUCONV_EXPORTS, אז הפונקציות יסומנו לייצוא החוצה מה DLL (ספרייה דינמית לשיתוף פונקציות).
ואם לא מוגדר, אז הן יסומנו כמיובאות פנימה לתוך תוכנית אחרת.
בצורה הזאת אותה כותרת של הקובץ מתאימה לשני המצבים, גם כשהספרייה נבנית וגם כשהיא בשימוש .

הקטע הבא מגדיר בעצם את הממשק של הפונקציה המרכזית במערכת, פונקציה שמבצעת קונבולוציה על הGPU.
זה כאילו שער הכניסה לכל קריאה חיצונית אל הספרייה.

```
extern "C" CUCONV_API int cuconv_conv_forward(
```

extern נכתב כדי למנוע שינוי שמות פונקציות בזמן הקומפילציה של C++.
בגלל extern השם של הפונקציה נשמר פשוט וברור, וזה מה שמאפשר קישור נוח גם מקוד בשפת C או מספרייה דינמית (DLL/so).

CUCONV_API- זה המאקרו שמוגדר בקובץ הAPI.
בWindows הוא מתורגם ל(declspec(dllexport או (declspec(dllimport בהתאם למצב הבנייה, (ובלינוקס הוא נשאר ריק).
ואז הוא מבטיח שהפונקציה תיוצא או תיובא כראוי.

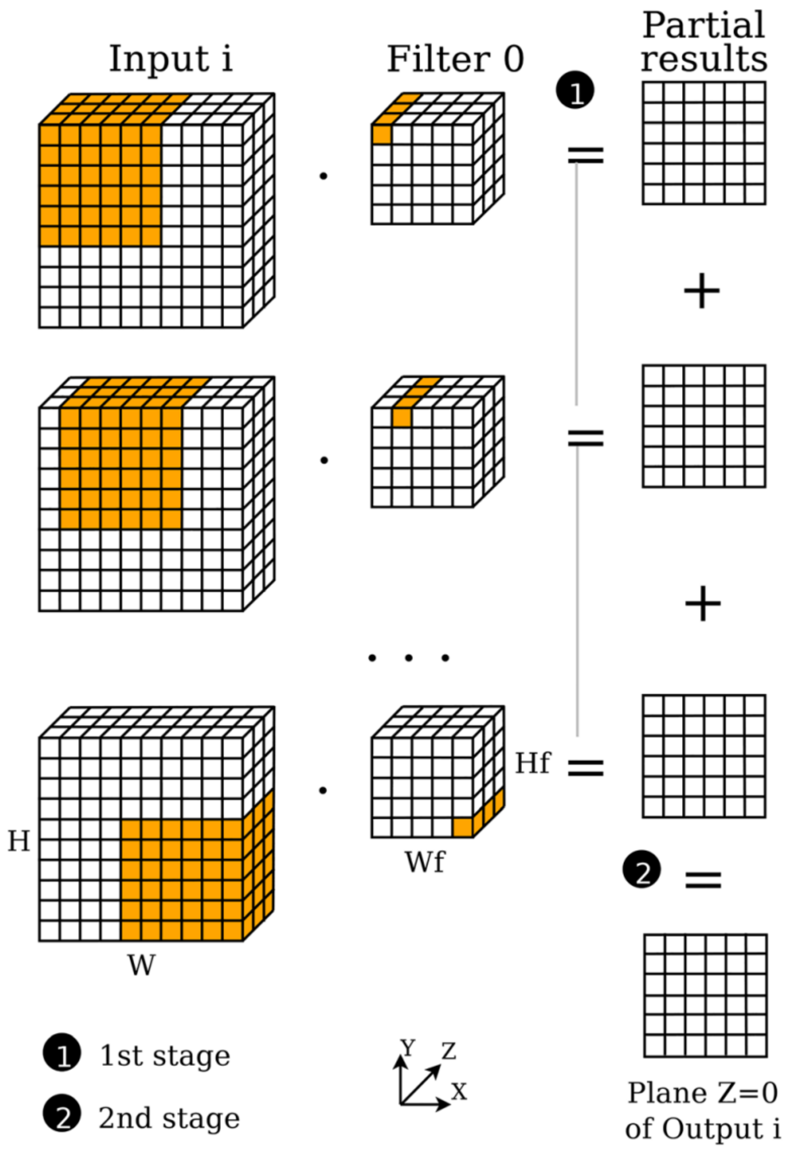
```
const float* x, const float* w, float* y,
int N, int C, int H, int W,
int M, int R, int S,
int pad_h, int pad_w,
int stride_h, int stride_w,
int dil_h, int dil_w,
int groups);
```

אלו שאר המשתנים שהפונקציה run שלחה בתוכנית הmain.cpp אל הפונקציה החיצונית cuconv_conv_forward.

4.3. cuconv_lib.cu

4.3.1. חיבור הAPI

במימוש הAPI הפונקציה cuconv_conv_forward משמשת בעצם כשכבת חיבור בין הקוד החיצוני (main.cpp) לבין הקרנלים שרצים על הGPU.
היא מקבלת מצביעים לנתוני הקלט, הפילטרים והפלט ביחד עם כל הפרמטרים של הממדים (N, C, H, W, M, R, S וכו').
היא גם מבצעת בדיקות בסיסיות, מחשבת את ממדי הפלט, מקצה גם זיכרון זמני על הGPU, ואז מפעילה ברצף שני קרנלים: scalar_prods_kernel שמחשב את המכפלות החלקיות ולאחר מכן את sum_kernel שסוכם אותם לתוצאה הסופית.



איור 9 : פעולת הקונבולוציה מתבצעת בשני שלבים, בשלב הראשון מחושבות מכפלות סקלריות ליצירת תוצאות חלקיות, ובשלב השני מתבצעת סכימתן לקבלת הפלט הסופי.

הספריות של הקובץ cuconv_lib.cu

```
<cuda_runtime.h>
```

זאת ספרייה לCUDA שמספקת את כל הפונקציונליות של הCUDA Runtime API
השימושים בקוד :

- הקצאת ושחרור זיכרון במכשיר : cudaMalloc, cudaFree.
- טיפוסים מובנים : dim3 .
- השקת קרנלים בסינטקס **<<<grid, threads>>>** .

```
<cstdlib>
```

זאת ספריית שירותי מערכת כלליים לC .
שימושים בקוד :

- נדרשת עבור טיפוסים סטנדרטיים כמו size_t .

```
"cuconv_api.h"
```

זהו קובץ הכותרת (header) של הAPI הציבורי שנוצר למערכת.
השימושים בקוד :

- הוא מכיל בתוכו את ההצהרה לפונקציה cuconv_conv_forward עם extern "C" כדי לשמור על שם יציב ולאפשר קישור מC או טעינה דינמית.
- הוא מגדיר את המקרו CUCONV_API שבעצם תומך בייצוא דינמי של הפונקציה לקובץ DLL .

קריאה לפונקציה

הפונקציה cuconv_conv_forward מוגדרת כנקודת הכניסה הראשית אל תוך הקוד שרץ על הGPU.
התפקיד שלה הוא לגשר בין הקריאה מבחוץ (במקרה המערכת, מקובץ main.cpp או מספריית דינמית) לבין ההרצה הפנימית של הקרנלים בCUDA.
היא אחראית לקלוט את כל פרמטרי ההרצה:
נתוני קלט, פילטרים, ממדים, Padding ועוד.
ולנהל את זרימת הביצוע: מהקצאת זיכרון והפעלת הקרנלים ועד ניקוי המשאבים.
המבנה הזה מבודד את קוד הCUDA מפרטי הקריאה החיצונית, ומאפשר שימוש יעיל, כללי ומודולרי בפונקציונליות של הספרייה.

```
extern "C" int cuconv_conv_forward(
    const float* x, const float* w, float* y,
    int N, int C, int H, int W,
    int M, int R, int S,
    int pad_h, int pad_w,
    int stride_h, int stride_w,
    int dil_h, int dil_w,
    int groups)
```

כאן זה המידע שהפונקציה אמורה לקבל כמתואר ממה שנשלח מהקוד הראשי של המערכת main.cpp מפונקציית run .

```
if (N != 1 || groups != 1) return -1;
if (stride_h != 1 || stride_w != 1) return -2;
if (dil_h != 1 || dil_w != 1) return -3;
```

החלק הזה מבצע בדיקות תנאי, כדי לוודא שההרצה תואמת למגבלות של האלגוריתם.
הוא מחזיר קוד שגיאה אם אחד מהתנאים הבאים מתקיים:

- אם מספר הדגימות (N) שונה מ1 או שיש קבוצות פילטרים.
- אם ערכי הצעדים שונים מ1.

```
int Ho = H + 2*pad_h - R + 1;
int Wo = W + 2*pad_w - S + 1;
```

שתי השורות האלה מחשבות את ממדי הפלט של הקונבולוציה (Ho, Wo) בציר האנכי והאופקי.
החישוב הזה מבוסס על נוסחת הקונבולוציה (מתוך המאמר) עם padding סימטרי, stride = 1, וdilation = 1.
זה בעצם קובע כמה פיקסלים יהיו במפת הפלט לאחר הרצת הפילטרים על הקלט.

```
const int kernelSize = R * S;
const bool is1x1 = (kernelSize == 1);
```

1x1 זה מצב פרטי שבו אין צורך לקריאה לקרנל השני כי התוצאות החלקי של פעולת הכפל הנקודתי הזה הם גם המוצא, כי אין "שכנים" לפיקסל הזה.
לכן נוצר משתנה לתנאי הזה, כדי שהקרנל השני לא ירוץ סתם.

```
float* d_partial = nullptr;
if (!is1x1) {
    size_t partial_sz = (size_t)kernelSize * M * Ho * Wo;
    cudaMalloc(&d_partial, partial_sz * sizeof(float));
            }
float* out_or_partial = is1x1 ? y : d_partial;
```

כאשר הפילטר שונה מ1X1 אז הקטע הזה מהקוד מחשב את הגודל של הזיכרון הדרוש לאחסון של התוצרים החלקיים של הקונבולוציה (partial_sz).
ומקצה עבורם זיכרון על הGPU עם הפונקציה cudaMalloc, כאשר הגודל נקבע לפי מספר התוצרים החלקיים הדרושים בכפל עם טיפוס של flaot, והזיכרון הזה משמש לאחסון תוצאות הביניים של הקרנל הראשון.
והמצביע (d_partial) מאותחל כדי להחזיק את כתובת אותו אזור.
זה שלב ההכנה לפני הפעלת הקרנלים שימלאו את התוצרים החלקיים .

```
{
     int threads = 256;
     int blocks_x = (Ho * Wo + threads - 1) / threads;
     dim3 grid(blocks_x, M, R * S);
     scalar_prods_kernel<<<grid, threads>>>(
            x, w, d_partial,
            H, W,
            pad_h, pad_w,
            R, S,
            C, M,
            Ho, Wo
     );
     cudaGetLastError();
}
```

הקטע הזה בעצם מייצג את השלב הראשון בתהליך של הקונבולוציה.
כאן גם מוגדר התצורה של ההרצה של הקרנל הראשון scalar_prods_kernel .
ונקבע מספר התרדים בכל בלוק (256).
וגם מחושב מספר הבלוקים בציר הx על הגריד לפי הממדים של הפלט (כלומר כמה "חתיכות עבודה" צריך כדי שכל הפיקסלים במפת הפלט יחושבו במקביל על הGPU).

ישנה גם רשת תלת ממדית של הגריד שבה הצירים הם :

- ציר x מחלק את העבודה על פני פיקסלי הפלט.
- ציר y מייצג את מספר הפילטרים (M).
- ציר z מייצג את כל המיקומים בפילטר (R×S).

הקרנל שמוזנק בשלב זה מבצע את חישובי הdot products עבור כל פילטר וכל מיקום בפלט שלו.
ואז שומר את התוצאות הזמניות במערך d_partial של המצביע שאותחל על הGPU.

ופונקציית `cudaGetLastError` נמצאת כדי לוודא שהקרנל בוצע כראוי ללא תקלות.

```
if (!is1x1) {
    int threads = 256;
    int blocks_x = (Ho * Wo + threads - 1) / threads;
    dim3 grid(blocks_x, M, 1);
    int kernelSize = R * S;
    sum_kernel<<<grid, threads>>>(
    d_partial, y, Ho, Wo, M, kernelSize
    );
    cudaGetLastError();
}
```

הקטע הזה מייצג את השלב השני והמסכם בתהליך של האלגורתם.
כלומר, לאחר שהקרנל הראשון הפיק לכל מיקום פלט את התוצרים החלקיים מכל מיקום בפילטר, כאן מוגדרים הפרמטרים של ההרצה (כמות תרדים ובלוקים).
וגם ערכו של ציר Z נקבע ל1 כדי שלא יהיה שכפול מיותר של רשת הבלוקים בממד השלישי.
ואז מופעל הקרנל sum_kernel (רק עבור מקרים שבהם הפילטר הוא לא 1X1) שהתפקיד שלו הוא לאחד את כל התוצאות החלקיות למפת פלט סופית אחת.
ככה שתיסגר שרשרת העיבוד.

ופונקציית `cudaGetLastError` נמצאת כדי לוודא שהקרנל בוצע כראוי ללא תקלות.

```
if (!is1x1)cudaFree(d_partial);
```

לאחר שפעולת שני הקרנלים בוצעה למימוש האלגורתם, המצביע שנשמר בזיכרון הGPU באמצעות cudaMalloc משוחרר מהזיכרון של הGPU .

4.3.2. קרנל scalar_prods_kernel

הקרנל הזה הוא שלב החישוב הראשון באלגוריתם CUCONV.
הוא מתמקד בביצוע הdot products בין קטעים מהקלט לבין המשקלים של הפילטרים עבור כל מיקום אפשרי של הפילטר על התמונה.
המטרה בשלב הזה היא לא לייצר את הפלט הסופי, אלא היא לבנות את התוצרים החלקיים שזה בעצם מערך גדול שבו לכל פילטר ולכל מיקום (fx, fy) בפילטר ולכל פיקסל פלט, נשמרת התוצאה של המכפלה הנקודתית לאורך ממד העומק.
השלב הזה מאפשר למערכת להפריד בין חישוב המכפלות עצמם לבין הסכימה שלהם, שתתבצע אחר כך בקרנל השני.
וככה מנוצלת המקביליות של הGPU, כלומר, כל ת׳רד מתמקד בפיקסל פלט אחד, עבור פילטר מסוים ומיקום פילטר מסוים, ומבצע עבורו את החישוב המלא.

```
__global__ void scalar_prods_kernel(
const float* __restrict__ input,
const float* __restrict__ filters,
float* __restrict__ partial_outputs,
int H, int W, int pad_h, int pad_w, int filterH, int
filterW, int depth, int numFilters,
int outH, int outW)
```

הקרנל מוגדר באמצעות __global__, כלומר הוא בעצם קרנל CUDA שרץ על הGPU ונקרא מהHost.
הוא מקבל מצביע לנתוני קלט input במבנה (N=1, C * H * W).
ולפילטרים מצביע filters במבנה (M * C * R * S).
כאשר כל פילטר כולל (C) ערוצים בגודל (R * S).
והתוצאות החלקיות נכתבות למצביע partial_outputs בגודל של (R * S) * M * Ho * Wo .
ככה שלכל מיקום בפילטר (fy,fx) שמורה מפת פלט נפרדת.
שאר הפרמטרים כוללים את ממדי הקלט (H, W).
את הpadding האנכי והאופקי (pad_h, pad_w).
את גודל הפילטר (filterH, filterW) שממנו נגזרים fy, fx .
את עומק הקלט שהוא מספר הערוצים.
את מספר הפילטרים (numFilters) שמשויכים לblockIdx.y .
ואת ממדי הפלט (outH, outW) המשמשים לחישוב אינדקס הפיקסל (outY, outX) מתוך אינדקס שטוח.

```
int fyfx = blockIdx.z;
if (fyfx >= filterH * filterW) return;
int fy = fyfx / filterW;
int fx = fyfx % filterW;
```

הקטע בקוד ממפה את האינדקס blockIdx.z למיקום דו ממדי בתוך הפילטר (fy, fx) שבו מתבצעת הפעולה הנוכחית של הthread block.
בגלל שפילטר בגודל filterW * filterH כולל מספר של filterW * filterH מיקומים, המשתנה fyfx מייצג את המיקום שלהם.
לאחר בדיקת חריגה, המערכת מחשבת את מיקום השורה fy = fyfx / filter .
ואת העמודה fx = fyfx % filterW.
לדוגמה, אם הפילטר הוא בגודל 3×3 (filterH = filterW = 3) והאינדקס fyfx = 5, אז fy = 5 / 3 = 1 וfx = 5 % 3 = 2, כלומר זה המיקום בשורה הראשונה והעמודה השנייה של הפילטר.
כלומר כל בלוק של ת׳רדים מתמקד במיקום אחד כזה בפילטר.

```
int filterIdx = blockIdx.y;
if (filterIdx >= numFilters) return;
```

כאן, כל בלוק בGPU מקבל את המשימה לחשב את התרומה של פילטר אחד מסוים לתמונת הפלט.
האינדקס filterIdx קובע איזה פילטר מתוך M שייך לבלוק הנוכחי.
אם הGPU יצר יותר בלוקים ממספר הפילטרים בפועל, הפקודה return מונעת בעצם מהבלוק הזה לבצע חישובים מיותרים.
ואז ככה מבטיחים שכל בלוק יעבוד רק על פילטר תקף, בלי לגרום לשגיאות או בזבוז משאבים סתם.

```
int outPixelIdx = blockIdx.x * blockDim.x + threadIdx.x;
if (outPixelIdx >= outH * outW) return;
```

הפקודה int outPixelIdx = blockIdx.x * blockDim.x + threadIdx.x מחשבת לכל ת׳רד את מספר הפיקסל שהוא אחראי עליו בפלט, כאילו כל מפת הפלט (בגודל Ho * Wo) נפרסה לשורה אחת ארוכה.
כדי לוודא שהאינדקס לא חורג ממספר הפיקסלים האפשריים.
בודקים if (outPixelIdx >= outH * outW) return .
אם כן, הת׳רד פשוט לא עושה כלום.
למשל, אם blockIdx.x = 2, blockDim.x = 256, threadIdx.x = 10
אז outPixelIdx = 2×256 + 10 = 522 .
כלומר, זה הפיקסל ה522 במטריצת הפלט.
אם יש פחות מ523 פיקסלים, הפקודה return תעצור את הת׳רד.

```
int outY = outPixelIdx / outW;
int outX = outPixelIdx % outW;
```

השורות קוד האלה מחשבות את שורת ועמודת הפלט מתוך אינדקס חד ממדי של פיקסל outPixelIdx במטריצת הפלט בגודל Ho×Wo.

```
int inY = outY - pad_h + fy;
int inX = outX - pad_w + fx;
```

השורות קוד האלה מחשבות את המיקום המתאים בקלט שממנו נלקח ערך עבור מיקום מסוים בפלט, תוך התחשבות בפדינג ובמיקום הפילטר.
לדוגמה, אם מחשבים את ערך הפלט במיקום (2,3) עם padding של 1 ופילטר במיקום fy=1 , הפקודה תחזיר inY = 3 - 1 + 1 = 3.
כלומר נקרא ערך מהשורה השלישית בקלט.

```
float sum = 0.0f;
if ((unsigned)inY < (unsigned)H && (unsigned)inX <
(unsigned)W)
{
for (int d = 0; d < depth; d++) {
int inIdx = d * H * W + inY * W + inX;

int fOff  =(((filterIdx * filterH + fy)*filter + fx) *
depth) + d;

float xv = input[inIdx];
float wv = filters[fOff];
sum += xv * wv;
                                        }
    } else {
        sum = 0.0f;
}
```

הקטע הזה הוא אחראי על חישוב סכום הdot product בין פאץ קטן (האזור בinput שהפילטר נמצא עליו) מתוך הinput לבין השכבה שמתאימה בפילטר (עבור פיקסל פלט מסוים, מיקום פילטר מסוים וערוץ עומק מסוים).

תחילה נוצר משתנה מקומי בשם sum שאליו מצטבר תוצאת הסכום של כל המכפלות בין הפיקסלים לערכי הפילטר לעומק.

לאחר מכן מתבצעת בדיקה האם הקואורדינטות (inY, inX) של הפיקסל הנוכחי בתמונה (לאחר לקיחת padding בחשבון) נמצאות בתוך גבולות הקלט.
וההמרה לunsigned מאפשרת לבדוק גם ערכים שליליים כי שלילי נהיה מספר חיובי גדול מאוד, ולכן אולי הוא לא יעבור את התנאי.
לדוגמה, אם inY = -1, אז 1- (unsigned) הוא 4294967295, שהוא גדול מH, ולכן התנאי נכשל.

ואם התנאי נכשל אז sum מקבל ערך 0.

ובמידה ויש המצאות בגבולות הקלט אז מתחילים לולאה על כל ערוצי העומק ,בשביל לבצע מכפלה נפרדת לכל אחד.
בתוך הלולאה מחשבים את המיקום של הפיקסל מתוך מערך הקלט החד ממדי, לפי הסדר NCHW.
לדוגמה, אם d=1, H=4, W=4, inY=2, inX=3, אז inIdx = 1*16 + 2*4 + 3 = 16 + 8 + 3 =27
ולאחר מכן מחשבים את המיקום של ערך המשקל המתאים (filter offset -fOff) מתוך מערך הפילטרים החד ממדי, תוך שילוב של אינדקס הפילטר, מיקום הפילטר בתוך המסכה (fy,fx) והערוץ d .

לאחר מכן המערכת לוקחת ערכים מהזיכרון באמצעות המצביעים שהוגדרו בתחילת הקרנל input ו filters ,
ומכניסה אותם למשתנים xv וwvו בהתאמה.
כאשר xv אוגר את הערך מהקלט וwv אוגר את המשקל של הפילטר במיקום המסוים.

משתנה sum מקבל את ערך מכפלתם (לאורך כל ערוץ העומק) ומצטבר לתוצאה כוללת עבור אותו הפיקסל.

```
int partialIdx = ( (filterIdx * (filterH * filterW) +
fyfx) * outH * outW ) + outPixelIdx;

partial_outputs[partialIdx] = sum;
```

בקטע קוד הזה מתבצעת כתיבה של תוצאת החישוב של החישוב החלקי למערך התוצאות הביניים partial_outputs.
קודם כל, מחושב האינדקס partialIdx שמייצג מיקום חד ממדי בתוך המערך.
האינדקס הזה נקבע לפי שילוב של שלוש פרמטרים :

- אינדקס הפילטר filterIdx .
- מיקום הפילטר בתוך חלון הקונבולוציה fyfx .
- ואינדקס פיקסל הפלט outPixelIdx .

החישוב מתבצע ככה : מספר הפילטר מוכפל בגודל הפילטר (filterH * filterW) לזה מתווסף fyfx בשביל למקם את האופסט הנוכחי בתוך הפילטר.
ואז יש הכפלה בגודל מפת הפלט (outH * outW) בשביל לחשב את תחילת הבלוק הרלוונטי בזיכרון.
ובסוף, מתווסף outPixelIdx בשביל שיהיה אפשר לגשת לתוצאה הספציפית עבור הפיקסל הנוכחי.
לאחר חישוב האינדקס, נשמר בו תוצאת הסכום sum באמצעות הצבה בpartial_outputs[partialIdx].
לדוגמה, עבור פילטר מספר 1 : (filterIdx = 1), מיקום פילטר שלישי (fyfx = 2), ופיקסל פלט מספר 5 (outPixelIdx = 5), ייחשב האינדקס כ (5 + Ho*Wo * (2 + (R*S) * 1)) והתוצאה תישמר במיקום הזה.
וככה מתבצעת הכתיבה לזיכרון של כל המכפלות הסקלריות שנדרשות לשלב הסיכום הסופי.

4.3.3. הקרנל sum_kernel

הקרנל הזה שמייצג את החלק השני באלגורתם, אחראי לעל השלב הסופי של החישוב.
הוא מקבל את כל החישובים החלקיים שנוצרו בשלב הראשון (כל אותן מכפלות סקלריות חלקיות של הפילטרים מול הקלט מהקרנל scalar_prods_kernel), ומחבר אותם לערך אחד סופי עבור כל פיקסל ביציאה של כל פילטר.
כלומר, אם הקרנל הראשון פיזר את העבודה למטריצות זמניות ,אז הקרנל השני אוסף את כל החלקים האלו ומבצע את פעולת הסיכום וככה שבסוף מתקבלת מפה סופית ונקייה של הפילטר על הקלט.

```
__global__ void sum_kernel(
    const float* __restrict__ partial_outputs,
    float* __restrict__ output,
    int outH, int outW,
    int numFilters, int kernelSize)
```

הקרנל sum_kernel הוא קרנל CUDA שמוגדר לריצה על הGPU, כאשר כל ת׳רד מבצע חישוב עצמאי על חלק מהנתונים.
הוא מקבל מצביע אל מערך partial_outputs שמכיל את כל החישובים החלקיים מכל מיקומי הפילטר (fx, fy), ומצביע לoutput שאליו תיכתב התוצאה הסופית לאחר סיכום.
שני המשתנים outH ו outW מייצגים את ממדי הפלט (גובה ורוחב).
numFilters הוא מספר הפילטרים הכולל (M).
וkernelSize מייצג את מספר האיברים בפילטר (למשל 9 בפילטר 3×3).
השימוש ב__restrict על שני המצביעים עוזר לקומפיילר להבין שאין חפיפות בזיכרון, כדי לאפשר אופטימיזציה של הקריאות והכתיבות.

```
int filterIdx   = blockIdx.y;
if (filterIdx >= numFilters) return;
```

בקטע הקוד הזה הקרנל מתחיל בלזהות על איזה פילטר מתוך כל הפילטרים הוא בעצם עובד.
השורה הראשונה אומרת שכל בלוק בציר y של הגריד מוקדש לפילטר אחר מתוך M פילטרים.
ולכן המשתנה filterIdx מחזיק את האינדקס של הפילטר הנוכחי.
ומיד אחרי זה יש בדיקה של האם הקרנל עבר על כל הפילטרים.

```
int outPixelIdx = blockIdx.x * blockDim.x + threadIdx.x;
if (outPixelIdx >= outH * outW) return;
```

כאן הקוד מחשב לכל ת׳רד מה הוא הפיקסל בפלט שעליו הוא אחראי.
השורה הראשונה לוקחת את מספר הבלוק בציר X (blockIdx.x) כפול מספר הת׳רדים בכל בלוק (blockDim.x).
ומוסיפה את האינדקס המקומי של הת׳רד (threadIdx.x).
השורה השנייה בודקת האם אותו אינדקס גדול או שווה למספר הפיקסלים הכולל בפלט (outH*outW).
אם כן, אז זה אומר שהת׳רד יצא מגבולות הפלט, ולכן הוא פשוט חוזר בלי להמשיך.

```
float sum = 0.0f;
for (int k = 0; k < kernelSize; k++) {
int pIdx = ((filterIdx * kernelSize) + k) *outH   * outW
+ outPixelIdx;

sum += partial_outputs[pIdx];
}
int outIdx = filterIdx * outH * outW + outPixelIdx;
output[outIdx] = sum;
```

בקטע קוד הזה מתבצע בעצם החיבור של כל התוצאות החלקיות אל הפלט הסופי.
השורה הראשונה מאתחלת משתנה בשם sum שישמש לאגירת הסכום.
אחרי זה יש לולאה שרצה על כל התוצאות החלקיות של אותו פיקסל .
ואז היא מחשבת לכל איטרציה את האינדקס המתאים בתוך המערך של התוצאות החלקיות (pIdx) ואז מוסיפה את הערך שמתאים לsum.
בסיום של הלולאה, יש חישוב של אינדקס הפלט הסופי (outIdx) עבור אותו פילטר ואותו פיקסל בפלט, ובסוף נכתבת לשם התוצאה המצטברת בsum, שהיא בעצם הערך המלא של הפיקסל בערוץ המסוים לאחר שהפילטר השלם הוחל עליו.

4.4. הרצת טורית על הCPU

בהרצה על הCPU מבוצע בדיוק את אותו התהליך חישוב כמו בגרסת הCUDA, אבל כולו מתבצע על המעבד הראשי בלבד.
במקום להריץ קרנלים לגריד של אלפי ליבות במקביל, הקוד מריץ את אותן פעולות בלולאות רגילות אחת אחרי השנייה ישירות בזיכרון של המחשב.
ואין צורך בהעתקות נתונים בין הHost לDevice או בניהול ת׳רדים של הGPU.
כל שלבי הכפל והסכימה מתבצעים מקומית, ככב שההבדל היחיד הוא **באופן הביצוע** ולא **באלגוריתם עצמו**.

הגדרת התצוגה:

הוגדר מבנה בשם Config שמטרתו לרכז בצורה מסודרת את כל הפרמטרים הנדרשים להרצת ניסוי יחיד.
השימוש במבנה זה מאפשר לנהל בקלות סדרה של תצורות שונות מבלי לפזר משתנים רבים מידי בקוד, וככה בעצם לשמור על בהירות וקריאות של הקוד .

```
struct Config {
    const char* table;
    const char* label;
    int N, H, R, M, C;
};
```

באמצעות המבנה הזה ניתן להגדיר בצורה ברורה וקומפקטית את כל מאפייני הניסוי, ולאפשר לקוד הראשי לעבור על רשימת תצורות מוכנה מראש.

פונקציית עזר למדידת זמן:

```
static double now_us(){
    timespec ts; clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
    return (double)ts.tv_sec*1e6 +
(double)ts.tv_nsec/1e3;}
```

הפונקציה הזאת מודדת זמן ריצה בלינוקס עם דיוק גבוה, באמצעות clock_gettime עם CLOCK_MONOTONIC_RAW שהוא לא מושפע מעדכונים של השעון.
היא מחזירה את הזמן במיקרו שניות באמצעות חיבור השניות והננו שניות לאחר המרה.
בפונקציית run שתוסבר בהמשך, מודדים את הזמן לפני ואחרי לולאת ההרצה, מחשבים ממוצע, ומציגים את התוצאה במילי שניות.

```
size_t inIdx(int c, int y, int x, int H, int W) {
    return ((size_t)c * (size_t)H + (size_t)y) *
(size_t)W + (size_t)x;
}
size_t filtIdx_cuda(int m, int c, int fy, int fx, int C,
int R, int S) {
    return (((size_t)m * (size_t)R + (size_t)fy) *
(size_t)S + (size_t)fx) * (size_t)C + (size_t)c);
}
size_t partIdx_cuda(int k, int m, int y, int x, int M,
int Ho, int Wo, int K) {
    return (((size_t)m * (size_t)K + (size_t)k) *
(size_t)Ho + (size_t)y) * (size_t)Wo + (size_t)x;
}
size_t outIdx(int m, int y, int x, int Ho, int Wo) {
    return ((size_t)m * (size_t)Ho + (size_t)y) *
(size_t)Wo + (size_t)x;
}
```

כאן מחושב האינדקסים החד ממדיים בשביל הגישות לזיכרון במערכים מרובי ממדים, כדי לעבוד בCPU בצורה זהה למה שקורה בGPU.
כל אחת מתרגמת את המיקום הלוגי (לדוגמא ערוץ, שורה, עמודה) למיקום פיזי במערך חד ממדי, לפי הסדר של האחסון שמקובל בכל סוג נתון.
ההמרה נעשית עם size_t בשביל להימנע מגלישות בזיכרון.

```
static void fill_random(vector<float>& v){
    mt19937 rng(42);
    uniform_real_distribution<float> U(-1.f, 1.f);
    for (auto& x : v) x = U(rng);
}
```

נוצר גם כאן שימוש בseed קבוע (42) שיבטיח שכל ריצה תחזור על אותם ערכים רנדומליים, וככה בעצם תוצאות הניסוי יהיו ניתנות להשוואה ולשחזור.

תוכנית הMAIN:

```
int INNER = X;
```

הגדרת משתנה מסוג INT בגדול X.
גודל זה מתאר את כמות הפעמים שהאלגורתם ירוץ על מעבד, כלומר הCPU .

```
vector<Config> cases = {
        {"T3-1x1","A", 1,  7,1,  256, 832},
        {"T3-1x1","B", 1, 14,1, 1024, 256},
        {"T3-1x1","C", 1, 27,1,  256,  64},
        {"T4-3x3","A", 1,  4,3,  384, 192},
        {"T4-3x3","B", 1, 13,3,  384, 384},
        {"T5-5x5","A", 1,  7,5,  128,  348},
};
```

במערך cases מוגדרות שישה תצורות ניסוי, שכל אחת מהן משלבת שם פילטר (1×1, 3×3, 5×5), תווית זיהוי (A,B,C), כמה תמונות נכנסות בכל הרצה (N) גודל תמונה (H), גודל פילטר (R), מספר פילטרים (M) ומספר ערוצים בקלט (C).
הערכים נבחרו ככה שהם ישקפו תרחישים אופייניים בקונבולוציות של רשתות CNN כמו שמתואר במאמר[1] .

ביצוע סדרת התצורות:

בקטע קוד הזה מתבצעת הקריאה לריצה על הCPU.
כל התצורות נשלחות בצורה מסודרת לפונקציית run לפי הגדרת Config עד שנגמר התצורות לשליחה.

```
for (const auto& cfg : cases) {
        run(cfg, INNER);
}
return 0;
```

הפונקציה run היא ההרצה של ניסוי בודד.

```
static float run(const Config& cfg, int inner_runs)
```

היא מקבלת תצורה אחת ואת מספר האיטרציות שיש לבצע, ומכינה את כל מה שנדרש כדי להריץ את הקרנל: חישוב גדלים, הקצאת זיכרון, יצירת נתוני קלט אקראיים.

שליפת פרמטרי התצורה והגדרת פרמטרי הקונבולוציה:

```
int N = cfg.N, C = cfg.C, H = cfg.H, W = cfg.H;
```

כאן נשלפים מתוך התצורה הערכים הבסיסיים: מספר הדוגמאות (N).
מספר הערוצים בקלט (C).
גובה התמונה (H) והרוחב (W) כאשר במקרה שלנו מוגדר שהרוחב שווה לגובה.

```
int M = cfg.M, R = cfg.R, S = cfg.R;
```

כאן נשלפים מספר הפילטרים (M) וגודל הפילטר (R) הערך S מוגדר שווה לR.
כלומר הפילטר ריבועי (גובה ורוחב שווים).

```
int pad = (R - 1) / 2, stride = 1, dil = 1, groups = 1;
```

כאן מוגדרים פרמטרים נוספים להפעלת הקונבולוציה:

- pad- חישוב הpadding ככה שהקונבולוציה תהיה בעצם "סימטרית" ותשמור על ממדי הקלט כמעט זהים.
- stride = 1 - ההזזה של הפילטר אל מול הפיקסל קלט מתבצעת צעד אחד בכל פעם.
- dil = 1 - ללא דילול (כלומר צפיפות פילטר אחידה).
- groups = 1 - כל הערוצים מחוברים יחד ללא חלוקה לקבוצות

חישוב גדלי המערכים בקונבולוציה:

```
size_t xin = (size_t)N * C * H * W;
```

חישוב מספר הערכים בקלט: מספר דוגמאות (N) כפול מספר ערוצים (C) כפול גובה (H) כפול רוחב (W).

```
size_t win = (size_t)M * C * R * S;
```

מחשב את מספר הערכים במשקולות: מספר פילטרים (M) כפול מספר ערוצים (C) כפול גובה פילטר (R) כפול רוחב פילטר (S).

```
int Ho = H + 2 * pad - R + 1;
```

מחשב את גובה הפלט לאחר הוספת padding ויישום הפילטרים.

```
int Wo = W + 2 * pad - S + 1;
```

מחשב את רוחב הפלט באותו צורה.

```
size_t yout = (size_t)N * M * Ho * Wo;
```

מחשב את מספר הערכים בפלט: מספר דוגמאות כפול מספר פילטרים כפול ממדי הפלט החדשים

בקטע הבא מתבצעת ההכנה של נתוני הקלט והפילטרים לפני ההרצה על הCPU.
נוצרות ארבעה מערכות נתונים בזיכרון :

```
vector<float> hx(xin), hw(win), hy(yout, 0.0f);
vector<float> partial((size_t)(R*S) * M * Ho * Wo);
fill_random(hx);
fill_random(hw);
```

hx (xin) – זהו המערך שמייצג את נתוני הקלט, בגודל התלוי במספר הדוגמאות, הערוצים והמידות המרחביות.

hw (win) – מערך שמייצג את משקולות הפילטרים, בגודל התלוי במספר הפילטרים, ערוצי הקלט וגודל הפילטר.

hy (yout, 0.0f) – מערך לפלט של ההרצה, מאתחל את כולו לערכים אפסיים כדי לשמור על עקביות בניסויים.

partial((R*S) * M * Ho * Wo) – זה מאגר ביניים לשלב 1, בגודל $(R \cdot S)\times M\times Ho\times Wo$.

לאחר יצירת המערכים של נתוני הקלט ומשקולות הפילטרים, שתי המערכים הראשונים (hx וhw) מתמלאים בערכים אקראיים אחידים באמצעות הפונקציה fill_random.

```
static void fill_random(vector<float>& v) {
    mt19937 rng(42);
    uniform_real_distribution<float> U(-1.f, 1.f);
    for (auto& x : v) x = U(rng);
}
```

נוצר גם שימוש בseed קבוע (42) שיבטיח שכל ריצה תחזור על אותם ערכים רנדומליים.

```
double t0 = now_us();
for (int i = 0; i < inner_runs; i++) {
computeScalarProducts(hx.data(), hw.data(),
partial.data(),H, W, Ho, Wo, pad, pad, C, M, R, S);
sumPartialResults(partial.data(), hy.data(), Ho, Wo, M,
R*S);
    }
double t1 = now_us();

double ms = double((t1 - t0) / inner_runs / 1000.0f);
printf("%s %s: N=%d, HxW=%dx%d, R=S=%d, M=%d, C=%d ->
Ho×Wo=%dx%d | Avg=%.3f ms\n",
cfg.table, cfg.label, N, H, W, R, M, C, Ho, Wo, ms);
return ms;
```

בקטע הזה נמדד הזמן הממוצע לביצוע ההרצה על הCPU.
בהתחלה נלקחה כמו חותמת זמן התחלתית במיקרו שניות באמצעות פונקציית now_us.
לאחר מכן בוצעה לולאה במספר חזרות לפי הערך inner_runs.
כאשר בכל איטרציה בוצעו שני שלבים:

- חישוב מכפלות סקלריות בין הקלט לפילטרים (computeScalarProducts),
- ולאחר מכן סכימת התוצאות החלקיות לפלט הסופי (sumPartialResults).

בסיום הלולאה נלקחה חותמת זמן נוספת, ומחושב הזמן הכולל שחלף.
הזמן חולק במספר החזרות בשביל לקבל זמן ממוצע, והומר ממיקרו שניות למילי שניות.
ולבסוף, הזמן הממוצע הוצג יחד עם פרטי ההרצה והוחזר להמשך שימוש.

computeScalarProducts:

הפונקציה מממשת על הCPU את השלב הראשון של פעולת הקונבולוציה, שבו בעצם מחושב הdot products בין האזורי קלט לבין הפילטרים השונים לאורך עומק הערוצים שלהם.
הפונקציה עצמה פועלת בצורה טורית על כל ממדי התמונה והפילטר, ואז מאחסנת את התוצאות במערך partial.
כל החישובים האלו מבוצעים בזיכרון של הHost ובלי הרצה מקבילית, בניגוד למימוש על הGPU שבה הפעולה הזאת מפוצלת בין אלפי ליבות.

```
static void computeScalarProducts(
const float* __restrict__ input,
const float* __restrict__ filters,
float* __restrict__ partial,
int H, int W, int Ho, int Wo, int pad_h, int pad_w,
int C, int M, int R, int S)
```

הפרמטרים של הפונקציה מגדירים את כל הנתונים שנדרשים לחישוב של הקונבולוציה על הCPU:

- input - זה מצביע למערך של הקלט (התמונה או המפת מאפיינים הקודמת), בגודל N×C×H×W, כאשר כאן N=1.
- filters - מצביע למערך המשקולות של הפילטרים, בגודל M×C×R×S (מספר פילטרים, ערוצים, גובה ורוחב הפילטר).
- partial - מצביע למערך שבו מאוחסנות הdot products של כל מכפלה סקלרית, בגודל (R×S)×M×Ho×Wo.
- H, W - גובה ורוחב הקלט המקורי.
- Ho, Wo - גובה ורוחב הפלט לאחר הקונבולוציה.
- pad_h, pad_w - גודל הpadding (המסגרת של האפסים שסביב התמונה).
- C - מספר ערוצי הקלט.
- M - מספר הפילטרים.
- R, S - ממדי הפילטר (גובה ורוחב).

ביחד, כל הפרמטרים האלה מאפשרים לפונקציה לעבור על כל הפיקסלים והפילטרים, ואז לחשב את כל המכפלות שמתאימות, ואז לשמור את התוצאה עבור כל מיקום במערך הביניים partial.

```
{
const int K = R*S;       //1
for (int fy=0; fy<R; ++fy){  //2
for (int fx=0; fx<S; ++fx){  //2
const int k = fy*S + fx;     //2
for (int m=0; m<M; ++m){     //3
for (int y=0; y<Ho; ++y){    //3
for (int x=0; x<Wo; ++x){    //3
float acc = 0.0f;            //3
const int inY = y - pad_h + fy;   /3
const int inX = x - pad_w + fx;   /3
if ((unsigned)inY < (unsigned)H && (unsigned)inX < //4
(unsigned)W)                                       //4
{                                                  //4
for (int c=0; c<C; ++c){                           //4
float xv = input[  inIdx(c,inY,inX,H,W) ];         //4
float wv = filters[ filtIdx_cuda(m,c,fy,fx,C,R,S) ]; //4
acc += xv * wv;                                      //4
}                                                    //4
}                                                    //4
partial[ partIdx_cuda(k,m,y,x,M,Ho,Wo,K) ] = acc;    //5
}
}
}
}
}
}
```

בתחילה, חושב מספר ההיסטים האפשריים בקרנל על ידי הכפלה בין הגובה לרוחב שלו (1).

לאחר מכן נסרקו כל ההיסטים האפשריים בקרנל, כלומר כל זוג (fy, fx) ולכל אחד מהם שויך אינדקס יחיד בשם k (2).

עבור כל היסט ועבור כל הפילטרים (m), מתבצעת סריקה של כל הפיקסלים האפשריים בתמונת הפלט (y, x).
עבור כל מיקום כזה הוגדר משתנה ביניים בשם acc שישמש לצבירת התוצאה החלקית,
בשביל זה חושבו הקואורדינטות של הקלט שמתאימות כשיש התחשבות בפדינג (3).

לפני ההמשך של החישוב, נבדק האם הקואורדינטות האלו תקפות (כלומר נמצאות בתוך הגבולות של התמונה המקורית).
אם כן, אז בוצעה לולאה על כל ערוצי הקלט c שבמהלכה נשלפו ערך הקלט וערך המשקל מהפילטר, והוכפלו ונוספו לצבירה (4).

בסיום של הלולאות הפנימיות, נשמרה התוצאה המצטברת במערך התוצאות החלקיות partial לפי אינדקס שמשקף את המיקום k ואת הפילטר m ואת מיקום הפלט (y, x) (5).

:sumPartialResults

הפונקציה הזאת ממשת על הCPU את השלב השני של האלגוריתם, שבו נלקחות התוצאות החלקיות שחושבו בשלב הקודם ומסוכמות לכל מיקום פלט.
המטרה של הפונקציה היא בעצם ליצור את הפלט הסופי של פעולת הקונבולוציה באמצעות סכימה של כל התרומות מההיסטים (R×S) עבור כל פילטר, בכל נקודת פלט במטריצה.

```
const float* __restrict__ partial,
float* __restrict__ output,
int Ho, int Wo,
int M, int K)
```

תיאור הפרמטרים :

- const float* __restrict__ partial- מצביע לקריאה בלבד אל מערך תוצאות הביניים מהשלב הראשון. גודל המערך K × M × Ho × Wo .
- float* __restrict__ output- מצביע לכתיבה אל מערך הפלט הסופי, לאחר סכימת התרומות מכל ההיסטים. גודל המערך M × Ho × Wo .
- int Ho - גובה הפלט לאחר ביצוע הקונבולוציה.
- int Wo - רוחב הפלט לאחר ביצוע הקונבולוציה.
- int M- מספר הפילטרים (כלומר, מספר הערוצים ביציאה).
- int K - מספר ההיסטים הכולל (R × S) כלומר כמות התרומות שצריך לסכם לכל פיקסל פלט.

```
{
for (int m=0; m<M; ++m){              //1
    for (int y=0; y<Ho; ++y){         //1
      for (int x=0; x<Wo; ++x){       //1
        float s = 0.0f;              //2
        for (int k=0; k<K; ++k)      //3
        s += partial[partIdx_cuda(k,m,y,x,M,Ho,Wo,K)];//3
        output[ outIdx(m,y,x,Ho,Wo) ] = s;     //4
        }
      }
    }
}
```

הלולאות האלה עוברות על כל הפלט, בהתחלה את כל הפילטרים m, לאחר מכן את כל שורות הפלט y, ובסוף את עמודות הפלט x.
ככה בכל שלב מוגדר תא פלט יחיד שעליו מתבצעת הסכימה (1).

משתנה צובר s מאותחל ל0.0 בשביל כל תא פלט (2).

ואז עבור כל תא פלט, נשלפות כל התרומות של ההיסטים k של הקרנל ובכל איטרציה נשלפת התרומה שמתאימה ממערך הביניים partial לפי אינדקס מחושב, ואז מתווספת למצבר (3).

בסיום הסכימה, הערך שהצטבר s נכתב למיקום שמתאים במערך הפלט output.
וככה נבנה הערך הסופי עבור אותו פילטר (m) ותא פלט (y, x).

4.5. ניתוח קרנלים באמצעות NSIGHT COMPUTE

לאחר בניית הקובץ EXE של המערכת ניתן לייצר פרופיל ביצועים באמצעות התוכנה NSIGHT COMPUTE .

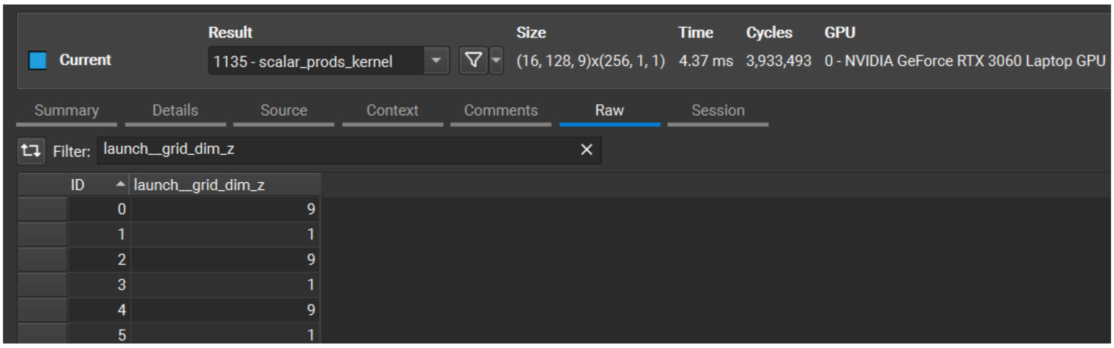
ניתן להבין ולהסיק מהנתונים שמתקבלים (מטריקות), הרבה על התנהגות ויעילות הקרנל בהתאם לחומרה.
כמו זמן הריצה של הקרנל, שימוש בזיכרון משותף, עומסים על DRAM וצווארי בקבוק אפשריים.

לאחר ביצוע פרופיל למערכת נמצא כי ישנם מטריקות (פרמטרים בתוכנה) שמציגות אפשרות לשיפור הכללי של המערכת.
(ID זוגיים הם scalar_prods_kernel וID אי זוגיים הם sum_kernel).

הפרופיל נמדד עבור קונפיגורציה הרצה: {3x3,"E1", 1, 32,3, 64, 64}

launch__grid_dim_z:

זה בעצם מדד שמציין את עומק הגריד בציר Z כלומר, בכמה שכבות נפרדות הקרנל מופעל במקביל.
כמו שהוגדר הקרנל רץ 9 פעמים נפרדות, אחת לכל (fy, fx) של הפילטר.
וזה מצב שיוצר שכפול מיותר של משאבים והקצאות בלוקים.
וכנראה שסתם יוצר עומס על בקר הגריד.

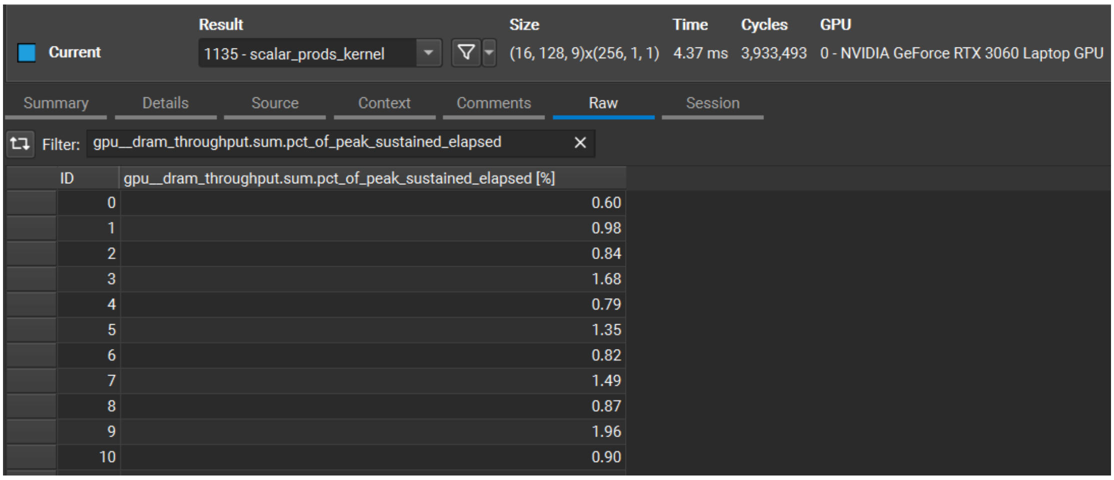


איור 10: תוצאה עבור מטריקה launch__grid_dim_z

ניתן לראות שכל פעם שהקרנל scalar_prods_kernel מיועד לריצה אז הוא רץ 9 פעמים נפרדות.
ולכן ניתן לבדוק אפשרות של הרצה על ציר Z יחיד בלבד.

gpu__dram_throughput.sum.pct_of_peak_sustained_elapsed:

המטריקה הזאת מציגה את האחוז ניצול רוחב הפס של זיכרון הDRAM ביחס לערך המקסימלי האפשרי שיכול להיות.
כשהערך גבוה, ניתן להסיק שמתבצעות הרבה קריאות וכתיבות לזיכרון הגלובלי, וזה יוצר עומס ועיכובים בחישוב.
לאחר ניתוח של הנתונים נמצא שהניצול היה גבוה .

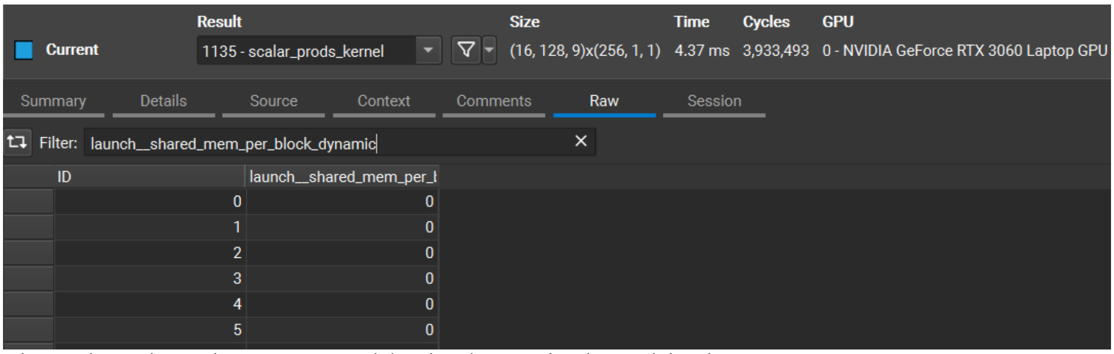


איור 11 : תוצאה עבור מטריקה
gpu__dram_throughput.sum.pct_of_peak_sustained_elapsed

הנתונים הם באחוזים.
ניתן לראות שהערכים די גבוהים וזה בעייתי, כי נראה שיש פה שימוש מסיבי בDRAM .
הערה : ישנם ערכים הגדולים מ1, איך זה יכול להיות?
ובכן, הערך יכול להיות גדול מ1 בגלל שNsight Compute מודד את השימוש הממוצע ביחס לרוחב הפס התאורטי, ובמקרים של עומס רגעי או חישוב ממוצע לא אחיד יכול להיות שהקצב שנמדד חורג זמנית מהערך המוגדר כ100%.

launch__shared_mem_per_block_dynamic [byte/block]:

המטריקה הזאת מציגה את כמות הזיכרון המשותף שהוקצתה לכל בלוק בזמן ריצת הקרנל. במערכת נראה שהערך הזה היה אפס, כלומר הקרנל לא השתמש בכלל בזיכרון המשותף וכל גישה לנתונים בוצעה רק מהDRAM וזה מה שגרם לעומס גבוה יותר על הזיכרון הגלובלי.

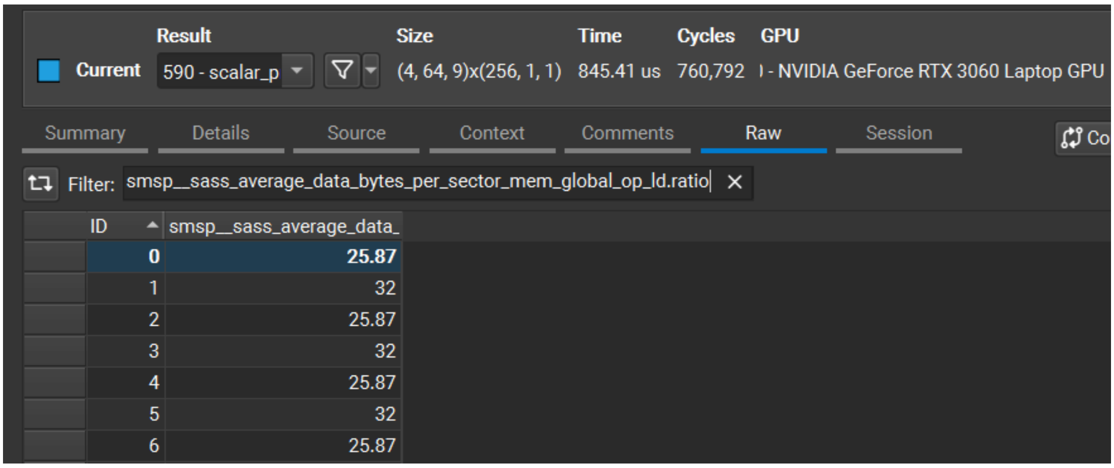


איור 12 : תוצאה עבור מטריקה launch__shared_mem_per_block_dynamic [byte/block]

ניתן לראות כי אין שימוש בזיכרון המשותף, וניתן להסיק שהGPU לא מגדיר את הזיכרון המשותף אוטומטית.

smsp__sass_average_data_bytes_per_sector_mem_global_op_ld.ratio:

המטריקה הזו מודדת עד כמה הקריאות של הGPU מהזיכרון הגלובלי יעילות.
כל קריאה לזיכרון מתבצעת ביחידות שנקראות sectors (למשל 128 בתים), והמטריקה הזאת בודקת כמה מתוך כל sector באמת נוצלו.
כשהערך גבוה זה סימן שהת׳רדים ניגשים לכתובות סמוכות בזיכרון, ככה שהGPU מצליח למזג את הקריאות ולנצל כמעט את כל הנתונים בכל גישה.
כשהערך נמוך זה אומר שהקריאות מפוזרות, וכל גישה משתמשת רק בחלק קטן מהsector, וזה מה שגורם לבזבוז ברוחב הפס ולביצועים נמוכים יותר.



איור 13 : תוצאה עבור מטריקה
smsp__sass_average_data_bytes_per_sector_mem_global_op_ld.ratio

ניתן לראות בIDים הזוגיים שהם קרנל scalar_prods_kernel שזהו ערך יחסית נמוך.
אבל בקרנל הסכימה יש שימוש טוב בגישה המאוחדת.

4.6. גרסה מאוחדת של האלגורתם

לאחר סקירת הנתונים מהתוכנה Nsight Compute בוצע עדכון למימוש האלגורתם באמצעות CUDA על גבי הGPU לקובץ cuconv_lib.cu.
לאור הממצאים הוחלט לאחד את שני הקרנלים לקרנל יחיד, למניעת כתיבות וקריאות מיותרות לזיכרון (סוף הקרנל הראשון כתיבה לpartial outputs ואז קריאה בקרנל השני).

ולהוסיף שימוש בזיכרון משותף באופן ידני.
ושימוש בLDG (טעינה מהזיכרון לקריאה בלבד דרך cache מהיר ואז זה יקטין גישות לDRAM ויאיץ ביצועים).
לכן כאן יצוין כל מה ששונה מהמערכת המקורית.

השינויים בקוד המאוחד:

קודם הגדרת הנתונים להשקת הקרנל.

```
int threads = 256;
int blocks_x = (Ho * Wo + threads - 1) / threads;
dim3 grid(blocks_x, M, 1);
```

כאן מוגדרת תצורת השקה של הקרנל.
בכל בלוק יהיו 256 ת׳רדים, ומספר הבלוקים בציר X יחושב ככה שכל ת׳רד יטפל בפיקסל אחד בפלט (Ho × Wo).
הגריד עצמו יוגדר ככה שכל בלוק בציר Y מתאים לפילטר שונה M, כדי שכל הגריד יכסה את כל הפילטרים והפיקסלים בפלט.
וציר Z בגריד הוגדר ל1 לבדיקת הביצועים לאור הממצאים הקודמים.

```
size_t shmem = (size_t)C * sizeof(float);
fused_conv_kernel<<<grid, threads, shmem>>>(
    x, w, y,
    H, W,
    pad_h, pad_w,
    R, S,
    C, M,
    Ho, Wo
);
```

בשלב הזה מוקצה מראש לכל בלוק של הקרנל המאוחד זיכרון משותף בגודל של וקטור אחד באורך C, שהוא בעצם עומק הפילטר.
ההקצאה נעשית על ידי הכפלה של מספר הערוצים C בגודל של מספר מסוג float, ונשמרת במשתנה shmem.
הזיכרון הזה משמש כל בלוק לאחסון מקומי ויעיל של פילטר אחד, ככה שכל הת׳רדים בבלוק יכולים לשתף ביניהם את המידע הזה מבלי לפנות לזיכרון הגלובלי האיטי יותר.
לאחר מכן הקרנל המאוחד מושק בעזרת ההוראה **fused_conv_kernel<<<grid, threads**.

ובעזרת ההוראה shmem>>>(x, w, y, H, W, pad_h, pad_w, R, S, C, M, Ho, Wo) , לקרנל מועברים מצביעים לקלט, לפילטרים ולפלט, וביחד עם כל פרמטרי הממדים שדרושים: גובה ורוחב הקלט, גודל הפדינג, מידות הפילטרים, מספר הערוצים ומספר הפילטרים, וגם גובה ורוחב הפלט.
כל המשתנים האלו מאפשרים לקרנל לבצע את חישובי הקונבולוציה.

הקרנל fused_conv_kernel:

לאחר השקת הקרנל המערכת פונה לקרנל המאוחד עצמו.

```
__global__ void fused_conv_kernel(
    const float* __restrict__ input,
    const float* __restrict__ filters,
    float* __restrict__ output,
    int H, int W,
    int pad_h, int pad_w,
    int R, int S,
    int C, int M,
    int Ho, int Wo
)
```

כאן מוכרז הקרנל המאוחד עם הנתונים ש"נשלחו" אליו, הרלוונטיים לביצוע הקונבולוציה.

```
int m = blockIdx.y;
int outPixelIdx = blockIdx.x * blockDim.x + threadIdx.x;
if (m >= M || outPixelIdx >= Ho * Wo) return;
```

בקטע הזה הקרנל "ממקם" כל ת׳רד על העבודה שלו בפלט.
קודם כל מזוהה איזה פילטר מטופל כרגע על ידי הבלוק בציר m = blockIdx.y.
אחר כך מחושב אינדקס הפיקסל פלט החד ממדי עבור אותו ת׳רד בתוך הבלוק ובתוך הגריד, ככה שכל ת׳רד מטפל בפיקסל אחר בHo*Wo.
ולבסוף נעשית בדיקת גבולות, אם הפילטר חורג ממספר הפילטרים m או שהפיקסל חורג מגודל מפת הפלט, הת׳רד יוצא return .

```
int outY = outPixelIdx / Wo;
int outX = outPixelIdx % Wo;
```

בקטע הזה מתבצעת המרה מאינדקס רציף לקואורדינטות, outY היא השורה (חלוקה שלמה בWo), וoutX היא בעצם העמודה (השארית % Wo).
נגיד למשל: Wo=10, אינדקס 23 הוא (outY,outX)=(2,3).

```
extern __shared__ float sFilter[];
```

המשתנה sFilter שמוגדר כאן הוא אזור משותף של זיכרון שכל ת׳רד בתוך אותו בלוק יכול לגשת אליו ביחד.
במקרה הזה, המערכת משתמשת בו כדי להעתיק לתוכו את הפילטר .
כלומר את הווקטור באורך C, רק פעם אחת מהזיכרון הגלובלי .
וזה חוסך המון זמן ריצה כי במקום שכל ת׳רד יקרא מחדש את אותם הערכים מהזיכרון הגדול והאיטי, הם משתפים ביניהם עותק אחד מהיר.

```
float acc = 0.0f;
```

אתחול של משתנה שיצבור את התוצאה של מכפלת הנקודה (כלומר סכום המכפלות בין הפילטר לבין הקלט בתת אזור מסוים).

```
for (int fy = 0; fy < R; ++fy){    //1
   for (int fx = 0; fx < S; ++fx){      //2
    for (int d = threadIdx.x; d < C; d += blockDim.x){//3
       int fOff = (((m * R + fy) * S + fx) * C) + d;  //4
       sFilter[d] = LDG(&filters[fOff]);              //5
    }
      __syncthreads();                                //6

      int inY = outY - pad_h + fy;
      int inX = outX - pad_w + fx;                 //7

if((unsigned)inY<(unsigned)H&&(unsigned)inX<(unsigned)W)
        {                                      //8
           for (int d = 0; d < C; ++d){        //9
           int inIdx = d * H * W + inY * W + inX; //10
           acc += LDG(&input[inIdx]) * sFilter[d]; //11
           }
        }

        __syncthreads();  //6
    }
}
```

בהתחלה נסרק ציר הY של הפילטר.
כל סיבוב של הלולאה מייצג מעבר על שורה אחת בפילטר, ככה שכל ערך של fy מצביע על היסט אנכי אחר מתוך R השורות של הפילטר (1).

לאחר מכן נסרק ציר הX של הפילטר.
לכל ערך fy, עוברים כעת על כל העמודות של הפילטר בעזרת המשתנה fx, עד שמכוסים כל היסטי הרוחב האפשריים בגודל S (2).

בשלב הבא נטענים ערכי הפילטר לזיכרון המשותף.
כל ת׳רד בבלוק אחראי לטעון חלק שונה מערוצי העומק C, ככה שכל הערוצים נטענים במקביל. והפעולה הזאת מאפשרת ניצול יעיל יותר של רוחב הפס של הזיכרון (3).

לכל ערוץ שנטען מחושב ההיסט שמתאים במערך המשקולות.
המשתנה fOff מגדיר את המיקום של אותו ערך פילטר בזיכרון הגלובלי, בהתאם לפילטר הנוכחי m, ולמיקום שלו בתוך הקרנל (fy, fx), ולעומק d (4).

ערך הפילטר הנמצא בכתובת fOff נטען מהזיכרון הגלובלי באמצעות הפונקציה LDG, ונשמר לתוך המערך sFilter[d] שבזיכרון המשותף.
בצורה הזאת נוצר עותק מהיר של הנתונים שכל הת׳רדים בבלוק יכולים להשתמש בו (5).

לאחר הטעינה מתבצע סנכרון __syncthreads().
הפעולה הזאת מבטיחה שכל הת׳רדים בבלוק סיימו את הטעינה לפני שמתחילים להשתמש במידע.
עוד סנכרון מתבצע מאוחר יותר, כדי לוודא שכל השרשורים סיימו את שלב השימוש לפני המעבר להיסט הבא (6).

בהמשך מחושבות קואורדינטות הקלט שמתאימות לפיקסל הפלט הנוכחי.
ובשביל זה מתבצעת התאמה של ההיסטים (fy, fx) עם מיקומי הפלט (outY, outX), עם החסרת הpadding בציר הגובה והרוחב (7).

לפני ביצוע של החישוב נבדק גם האם הקואורדינטות שנוצרו נמצאות בתוך הגבולות של התמונה המקורית.
אם לא, אז הפיקסל נמצא מחוץ לטווח ואין צורך לעבד אותו (8).

אם המיקום תקין, אז מתבצעת לולאה על כל ערוצי העומק C.
בכל מעבר נשלפים ערך הקלט וערך המשקל שתואם מהזיכרון המשותף, ונעשית מכפלה בין השניים (9).

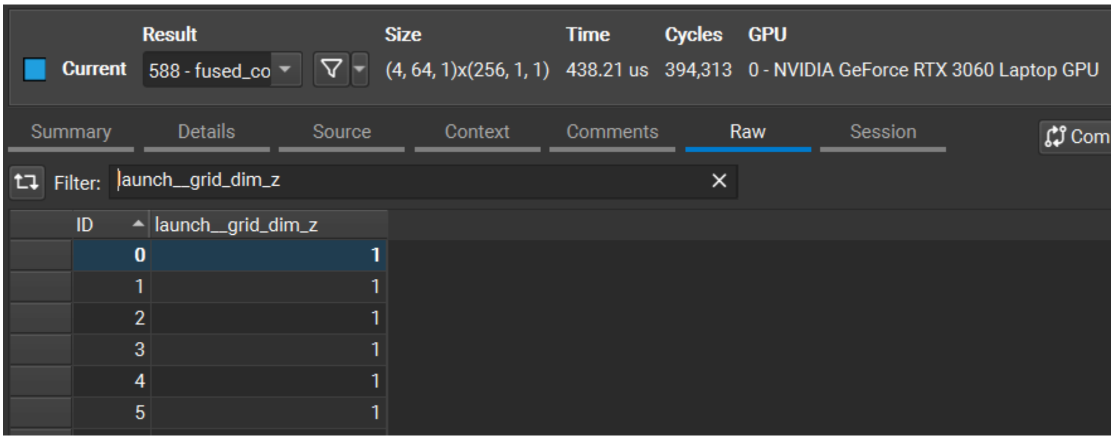
לכל ערוץ שמחושב יש אינדקס שטוח שמתאים במערך הקלט.
בהתאם לעומק d ולמיקום (inY, inX).
הפעולה הזאת נדרשת כדי להגיע למקום הנכון בזיכרון הפיזי (10).

ובסוף, ערך הקלט נטען בעזרת LDG, שמוכפל בערך הפילטר שמתאים מsFilter[d], והתוצאה מצטברת במשתנה acc.
בצורה הזאת נבנה סכום מכפלות שמייצג את הערך הסופי של הפיקסל פלט (11).

```
int outIdx = m * Ho * Wo + outPixelIdx;
output[outIdx] = acc;
```

השורה קוד הזאת מחשבת את המיקום המדויק של הפיקסל במפת הפלט של הפילטר הנוכחי, ואז שומרת לשם את הערך שחושב (acc) .

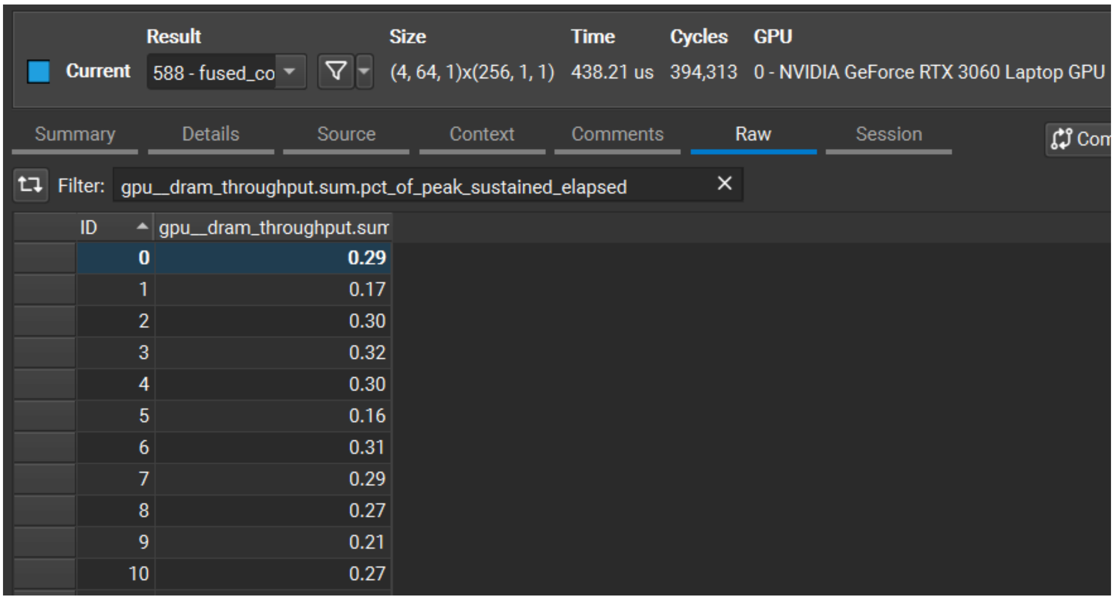
launch__grid_dim_z:



איור 14 : תוצאה עבור מטריקה launch__grid_dim_z

ניתן לראות שמספר הריצות של הבלוק בציר Z ירד ל1 וכל ההיסטים של הפילטר רצים על גריד על x וy וגם 1 על z כמתוכנן.

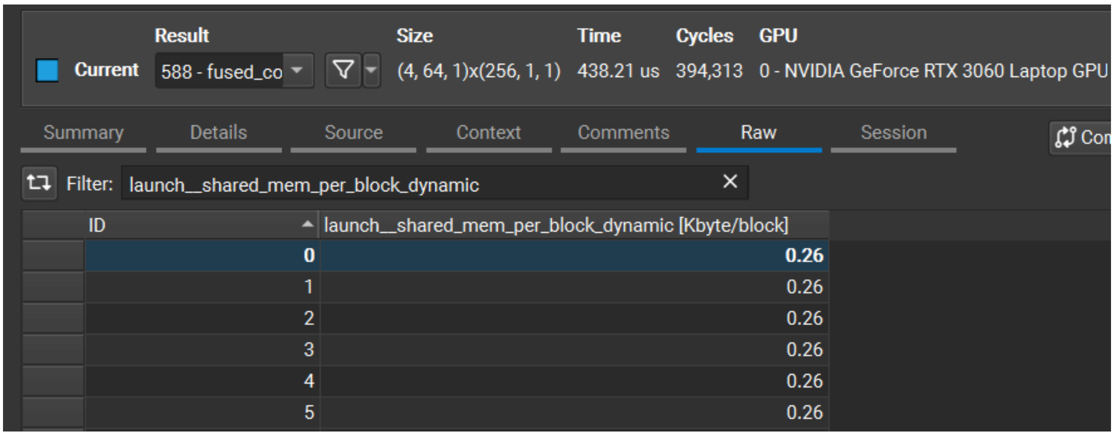
gpu__dram_throughput.sum.pct_of_peak_sustained_elapsed:



איור 15 : תוצאה עבור מטריקה
gpu__dram_throughput.sum.pct_of_peak_sustained_elapsed

ניתן לראות כי לאחר איחוד של הקרנלים, ניצול של רוחב הפס לDRAM ירד בפי 4 בממוצע. המשמעות היא בעצם ירידה די דרמטית במספר הגישות לזיכרון הגלובלי, בגלל שימוש יעיל יותר בזיכרון המשותף ובcache הפנימי של הGPU.

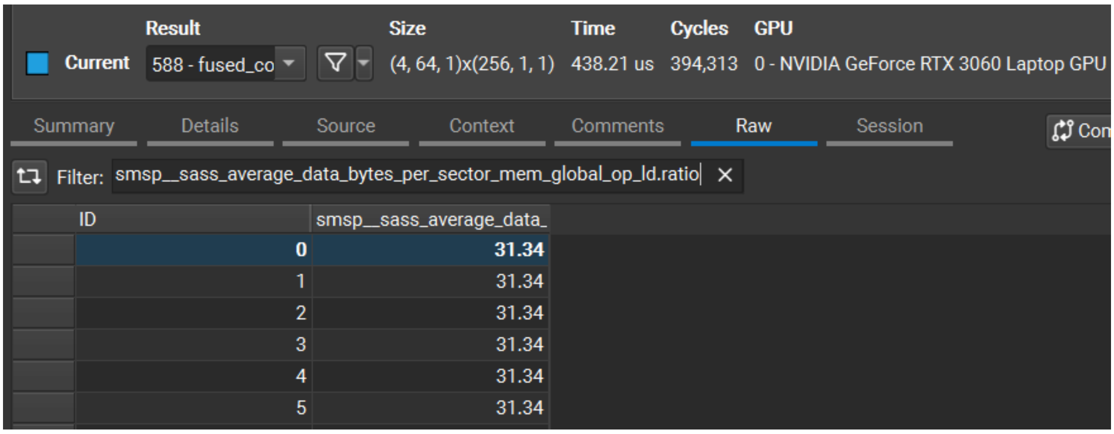
launch__shared_mem_per_block_dynamic [byte/block]:



איור 16 : תוצאה עבור מטריקה launch__shared_mem_per_block_dynamic [byte/block]

ניתן לראות שלאחר איחוד של הקרנלים נראה שיש הקצאה דינמית של זיכרון משותף בגודל של 0.26 KB לכל בלוק, לעומת 0 בקרנל המקורי.
(עבור קונפיגורציית הרצה: {3x3,״E1״, 1, 32,3, 64, 64}).
המשמעות היא בעצם שבקרנל המאוחד הפילטר נטען פעם אחת בלבד לזיכרון מהיר ונגיש לכל הת׳רדים בבלוק, במקום שכל ת׳רד יקרא את אותם ערכים כל פעם מהזיכרון הגלובלי.
השימוש בזה הפחית משמעותית את מספר הגישות לDRAM .

smsp__sass_average_data_bytes_per_sector_mem_global_op_ld.ratio:



איור 17 : תוצאה עבור מטריקה
smsp__sass_average_data_bytes_per_sector_mem_global_op_ld.ratio

ניתן לראות כי בקרנל המאוחד הערך עלה מ 26 bytes/sector ל 31 bytes/sector.
והמשמעות היא שקריאות הזיכרון הפכו ליותר רציפות ויעילות, ככה שכל קריאה לDRAM מנצלת טוב יותר את רוחב הפס.
זה סימן לשיפור בcoalescing וביעילות הגישה לזיכרון בגלובלי.

5. ניסויים

בפרק הזה נבחנת היעילות של מימושי הקונבולוציה שפותחו בפרויקט, בהשוואה לביצוע זהה בסביבת CPU.
מטרת הניסויים האלו היא לאמוד את זמני הריצה.
לכן, בוצעו שתי קבוצות ניסוי:

1. קונפיגורציות המאמר "T3-1×1, T4-3×3, T5-5×5" שבהן הושוו שלושה מימושים: GPU מפוצלת-מקורית, GPU מאוחדת-מחודשת, CPU מקורית.

2. סדרת E1-E4 3×3 שמדמה עלייה מונוטונית בנפח החישוב ומציבה את הCPU מול שתי הגרסאות של הGPU.

בכל אחת מהריצות האלו נשמרו תנאים אחידים: גודל תמונה N=1, צעד (stride) של 1, padding סימטרי, ותצורת זיכרון בפורמט NCHW.

המדדים שנאספו כוללים את זמן הריצה הממוצע, התוצאות מוצגות באמצעות טבלאות.

5.1. קונפיגורציות המאמר

```
{"T3-1x1","A", 1,  7,1,  256, 832},
{"T3-1x1","B", 1, 14,1, 1024, 256},
{"T3-1x1","C", 1, 27,1,  256,  64},
{"T4-3x3","A", 1,  4,3,  384, 192},
{"T4-3x3","B", 1, 13,3,  384, 384},
{"T5-5x5","A", 1,  7,5,  128,  48},
```

תצורה	GPU מאוחדת-מחודשת	CPU מקורית	GPU מפוצלת-מקורית
T3-1x1 A	0.686	237.219	0.599
T3-1x1 B	0.689	1167.742	0.6775
T3-1x1 C	0.632	274.234	0.5795
T4-3x3 A	0.715	167.386	1.751
T4-3x3 B	1.355	4570.385	2.634
T5-5x5 A	0.700	121.043	1.700

טבלה 1 : תוצאות קונפיגורציות מאמר

כל הזמנים במילישניות (ms) והם ממוצע של 20 הרצות.

5.2. קונפיגורציות להמחשת יתרון הGPU מול הCPU

```
{"3x3","E1", 1,  32,3,  64,  64},
{"3x3","E2", 1,  32,3, 128, 128},
{"3x3","E3", 1,  64,3, 128, 128},
{"3x3","E4", 1,  64,3, 256, 256},
```

תצורה	GPU מאוחדת-מחודשת	GPU מפוצלת-מקורית	CPU מקורית
E1	0.5415	1.057	870.594
E2	1.0615	1.653	3850.151
E3	4.3710	4.856	20179.631
E4	16.2835	16.892	154838.484

טבלה 2 : תוצאות קונפיגורציות המחשה

כל הזמנים במילישניות (ms) והם ממוצע של 20 הרצות.

6. מסקנות

ניתן לראות שהGPU בכל הגרסאות והקונפיגורציות מהיר משמעותית מהCPU.

כאשר ההבדל הכי גדול הוא בקונפיגורצית E4 עבור הקרנל המאוחד שהוא מהיר פי 9508 מאשר הCPU.
זה ככה בגלל שהGPU בעל רוחב פס חישובי גדול ומנצל זיכרון משותף והCPU מוגבל בעומס החישובי שלו.

ניתן גם לראות כי בעבור קונפיגורציות שבהם יש פילטר של 1X1 זמן הריצה של הגרסה המפוצלת קטן יותר מאשר הגרסה המאוחדת על הGPU, זאת משום שהגרסה המאוחדת לא מבצעת טיפול במקרה הפרטי של פילטר 1X1 .
לעומת הגרסה המפוצלת שכן מבצעת טיפול אישי במקרה, ולא קוראת לביצוע סכימה מיותרת .
ולא מוסיפה שלבים כמו טעינה לזיכרון המשותף ומחסומי סנכרון שלא תורמים במצב הזה.
מה שמראה את החשיבות של התניה במקרים פרטיים.
ובפילטר 1×1 אין תועלת לחלוקה להיסטים, כי כל פיקסל פלט מתקבל בכפל פשוט אחד.
בגלל זה כל תוספת לוגיקה וניהול זיכרון פנימי רק מעכבים.

אבל בשאר הקונפיגורציות ללא יוצא מן הכלל יש יתרון ברור לגרסה המאוחדת.

7. רשימות

7.1. רשימת איורים

איור 1 : מבנה כללי של CNN 5
איור 2 : המחשה של פעולת הקונבולוציה בתמונת קלט 6
איור 3 :מבנה מעבד GPU לעומתCPU 7
איור 4 : השוואה בין גישות זיכרון Coalesced לעומתUncoalesced 9
איור 5 : שילוב מטריצות חלקיות לקבלת פלט סופי בתהליך הקונבולוציה 11
איור 6 :הרכיב GPU שרץ על Windows הוא Dedicated GPU 14
איור 7 :תהליך זרימת התוכנה 16
איור 8 : תהליך פעולת קובץmain.cpp 17
איור 9 : פעולת הקונבולוציה מתבצעת בשני שלבים, בשלב הראשון מחושבות מכפלות סקלריות ליצירת תוצאות חלקיות, ובשלב השני מתבצעת סכימתן לקבלת הפלט הסופי. 26
איור 10 : תוצאה עבור מטריקה launch__grid_dim_z 47
איור 11 : תוצאה עבור מטריקה gpu__dram_throughput.sum.pct_of_peak_sustained_elapsed .48
איור 12 : תוצאה עבור מטריקה launch__shared_mem_per_block_dynamic [byte/block] 49
איור 13 : תוצאה עבור מטריקה smsp__sass_average_data_bytes_per_sector_mem_global_op_ld.ratio 49
איור 14 : תוצאה עבור מטריקה launch__grid_dim_z 54
איור 15 : תוצאה עבור מטריקה gpu__dram_throughput.sum.pct_of_peak_sustained_elapsed .54
איור 16 : תוצאה עבור מטריקה launch__shared_mem_per_block_dynamic [byte/block] 55
איור 17 : תוצאה עבור מטריקה smsp__sass_average_data_bytes_per_sector_mem_global_op_ld.ratio 55

7.2. רשימת טבלאות

טבלה 1 : תוצאות קונפיגורציות מאמר 57
טבלה 2 : תוצאות קונפיגורציות המחשה 58

8. מקורות

8.1. מקורות ספרותיים

[1]**Title:** cuConv: A CUDA Implementation of Convolution for CNN Inference
Author(s): Marc Jorda, Pedro Valero Lara, Antonio J Pe˜na
Source: https://arxiv.org/pdf/2103.16234
Published: 2021

[2]עיבוד מקבילי, מתוך ויקיפדיה:
https://he.wikipedia.org/wiki/%D7%A2%D7%99%D7%91%D7%95%D7%93_%D7%9E%D7%A7%D7%91%D7%99%D7%9C%D7%99

[3]**CUDA C basics:** https://www.nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf

[4]**GPU Optimization Fundamentals:** https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf

[5]**Kernel Profiling Guide:**
https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

[6]**Nsight Compute Guide :**
https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

8.2. מקורות עזר חזותיים

[1]**what is a convolution? by: 3Blue1Brown**
https://youtu.be/KuXjwB4LzSA?si=-DAl89EAp_6tZv1d

[2]**Convolutional Neural Networks from Scratch | In Depth : Erai**
https://www.youtube.com/watch?v=jDe5BAsT2-Y

9. נספחים

9.1 קוד מקור לפרוייקט

An updated version of the project is available on GitHub:
https://github.com/NERIYAE/Final-Project-GPU-Accelerated-CNN-Inference