ΤΜΉΜΑ ΗΛΕΚΤΡΟΛΌΓΩΝ ΜΗΧΑΝΙΚΏΝ ΚΑΙ ΜΗΧΑΝΙΚΏΝ ΥΠΟΛΟΓΙΣΤΏΝ

ΠΑΝΕΠΙΣΤΉΜΙΟ ΘΕΣΣΑΛΊΑΣ

CE421 ΣΥΣΤΉΜΑΤΑ ΥΠΟΛΟΓΙΣΜΟΎ ΥΨΗΛΏΝ ΕΠΙΔΌΣΕΩΝ

Πουρναρόπουλος Φοίβος, ΑΕΜ: 2614

Ηλιάδης Γρηγόρης, ΑΕΜ: 2522

Χρησιμοποιώντας την προηγούμενη εργασία ως υπόβαθρο, πλέον καλούμαστε να βελτιστοποιήσουμε το πρόγραμμα που υλοποιήθηκε σε CUDA με σκοπό την εφαρμογή ενός διαχωρίσιμου δισδιάστατου φίλτρου πάνω σε ένα δισδιάστατο πίνακα που θεωρούμε ότι αντιστοιχεί σε μία εικόνα. Πιο συγκεκριμένα, βασιζόμαστε στην έκδοση με padding η οποία εξαλείφει το φαινόμενο του divergence με αριθμούς διπλής ακρίβειας για να εκτελέσουμε τον κώδικα σε πίνακες πολύ μεγαλύτερου μεγέθους και να επικαλύψουμε διαφορετικούς υπολογισμούς μεταξύ τους και υπολογισμούς με μεταφορές δεδομένων.

Βήμα 0

Σε αυτό το βήμα τρέχουμε το deviceQuery στο σύστημα csl-artemis, όπου θα γίνουν όλες οι μετρήσεις και καταγράφουμε τα αποτελέσματα.

```
spournar@csl-artemis:~/lab3/samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.1 / 10.2
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 6 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

```
Device 1: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.1 / 10.2
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 7 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU1) : Yes
> Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU0) : Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.1, CUDA Runtime Version = 10.2, NumDevs = 2
Result = PASS
spournar@csl-artemis:~/lab3/samples/1_Utilities/deviceQuery$
```

Σε όλα τα παρακάτω βήματα οι χρόνοι εκτέλεσης υπολογίζονται ως εξής: Το κάθε πείραμα γίνεται 12 φορές και εξαιρείται ο μέγιστος και ο ελάχιστος χρόνος εκτέλεσης. Έπειτα βρίσκουμε το μέσο όρο και την τυπική απόκλιση.

Βήμα 1

Στο συγκεκριμένο βήμα κάνουμε βελτιστοποιήσεις στον κώδικα με σκοπό να μειώσουμε τον συνολικό χρόνο εκτέλεσης. Οι βελτιστοποιήσεις με τη σειρά που έγιναν είναι οι εξής:

1)Χρήση της shared memory για κάθε thread block της γεωμετρίας - Memory coalescing

Πιο αναλυτικά, στην υλοποίηση μας εξετάζουμε εάν η εικόνα εισόδου είναι μικρότερη από το μέγιστο αριθμό threads σε ένα block(1024 στο συγκεκριμένο σύστημα). Εάν το παραπάνω ισχύει, τότε βάζουμε ολόκληρη την εικόνα εισόδου μέσα σε ένα μοναδικό block, το οποίο θα έχει διαστάσεις ImageWidth, ImageHeight στους άξονες x,y αντίστοιχα, διαφορετικά χρησιμοποιούμε 1024 threads σε κάθε block 32 στον άξονα x, 32 στον άξονα y. Το κάθε νήμα υπολογίζει ένα στοιχείο του τελικού πίνακα-αποτελέσματος. Επιπλέον, η shared memory χρησιμοποιείται με σκοπό να μειώσουμε όσο το δυνατό περισσότερο τις προσπελάσεις στη global memory και να αξιοποιήσουμε την cache, στην οποία οι προσπελάσεις είναι σημαντικά πιο γρήγορες. Αυτό θα έχει ως αποτέλεσμα να προσπελαύνουμε την global memory μόνο μία φορά για κάθε στοιχείο του πίνακα εισόδου για κάθε block νημάτων και όταν χρειαστούμε ξανά τα ίδια στοιχεία προσπελαύνουμε την shared memory. Χρησιμοποιείται συγχρονισμός στο επίπεδο των blocks, ώστε να είμαστε σίγουροι ότι όλα

τα νήματα ενός block έχουν φορτώσει τις θέσεις που τους αντιστοιχούν, για να μπορέσουν να συνεχίσουν στον υπολογισμό του αποτελέσματος. Για να μπορέσουμε να κάνουμε σωστά τους υπολογισμούς, χρειαζόμαστε σε κάθε block να έχουμε και τα γειτονικά στοιχεία της εικόνας, δηλαδη filterR στοιχεία αριστερά και δεξιά για κάθε γραμμή νημάτων του kernelRow, ενώ χρειαζόμαστε τα filterR στοιχεία πάνω και κάτω από το block για κάθε στήλη του kernelColumn για να εφαρμόσουμε το φίλτρο. Αυτό συνεπάγεται ότι κάποια από τα νήματα θα φορτώσουν παραπάνω στοιχεία στη shared memory σε σχέση με κάποια άλλα, στις περιπτώσεις όπου ο αριθμός BlockDim.x + 2*filterR ή BlockDim.y+2*filterR δεν διαιρείται ακριβώς με το BlockDim.x ή BlockDim.y αντίστοιχα. Στην παραπάνω περίπτωση θα υπάρξει divergence όταν κάποια threads θα φορτώσουν στοιχεία ενώ κάποια άλλα σταματούν καθώς έχουν τελειώσει με τα στοιχεία που μετέφεραν. Για να μοιράσουμε τη "δουλειά" όσο περισσότερο γίνεται και να πετύχουμε memory coalescing για τις προσπελάσεις στην global memory, το κάθε νήμα φορτώνει με ένα stride που ισούται με BlockDim.x για kernelRow και BlockDim.y για kernelColumn, ώστε να φορτώσουμε μία γραμμή/στήλη στη shared memory για κάθε γραμμή/στήλη των thread blocks, μόνο εφόσον το άλμα του κάθε νήματος αντιστοιχεί σε νόμιμη θέση στη global memory. Ταυτόχρονα, επιτυγχάνουμε memory coalescing στις προσπελάσεις στη global memory και στους 2 kernels, καθώς τα νήματα ενός warp προσπελαύνουν διαδοχικές θέσεις στη global memory (ίδιας γραμμής και επόμενης στήλης σε σχέση με το προηγούμενο νήμα), ώστε να ομαδοποιήσουμε όσο το δυνατό περισσότερο τα memory transactions και να μειώσουμε το latency των προσπελάσεων στην "ακριβή" global memory. Για να μπορέσουμε να υλοποιήσουμε τη χρήση της shared memory χρησιμοποιούμε την **παραδοχή ότι το μέγεθος φίλτρου περιορίζεται στο διάστημα [1,80]** διότι πρέπει για το μέγιστο αριθμό από νήματα σε ένα block να "χωράνε" στη shared memory και επιπλέον 2*filterR*BlockDim.y για kernelRow και 2*filterR*BlockDim.x για kernelColumn αντίστοιχα.

2)Χρήση constant memory για αποθήκευση του φίλτρου της συνέλιξης

Σε αυτή τη βελτιστοποίηση χρησιμοποιούμε την constant memory για την αποθήκευση του φίλτρου, καθώς είναι read-only και έχει πολύ χαμηλότερο latency προσπέλασης σε σχέση με τη global memory διότι χρησιμοποιεί διαφορετικό read path που περιλαμβάνει ειδικές caches. Επίσης, το γεγονός ότι τα threads του ίδιου warp προσπελαύνουν την ίδια διεύθυνση του φίλτρου σε κάθε επανάληψη, πραγματοποιεί αυτές τις προσπελάσεις παράλληλα για το κάθε νήμα του warp, κάτι που δε θα γινόταν εάν τα νήματα διάβαζαν διαφορετικές διευθύνσεις, όπου θα υπήρχε σειριοποίηση. Το μήκος του φίλτρου στην constant memory ορίζεται να είναι ίσο με 2*80+1 καθώς όπως προαναφέρθηκε το 80 είναι η μέγιστη τιμή φίλτρου που μπορούμε να υποστηρίξουμε για να μπορούμε να έχουμε τον απαραίτητο χώρο στη shared memory.

3) Χρήση registers για εξοικονόμηση χρόνου σε επαναλαμβανόμενες εκφράσεις

Η συγκεκριμένη βελτιστοποίηση είναι η χρήση των registers, έτσι ώστε το κάθε νήμα να έχει αποθηκευμένες τις τιμές των συχνά επαναλαμβανόμενων πράξεων ώστε να τις υπολογίζει μόνο μία φορά κατά την εκτέλεση των kernels.

4)Χρήση pinned host memory για πιο γρήγορες μεταφορές μνήμης μεταξύ host/device

Εφόσον στη συγκεκριμένη εργασία δεν μετράμε το χρόνο για δέσμευση μνήμης στον host, θεωρούμε ότι μία επιπλέον βελτιστοποίηση που επιφέρει σημαντική μείωση του χρόνου εκτέλεσης

και μεταφορών μνήμης μεταξύ host και device είναι η χρήση της pinned host memory. Με αυτό τον τρόπο, μειώνουμε σημαντικά τους χρόνους για τις μεταφορές δεδομένων από το host στο device και αντίστροφα, καθώς πλέον παρακάμπτουμε το στάδιο του extra copy για μεταφορά από pageable memory σε pinned memory, κάτι που θα έπρεπε να γίνει στην περίπτωση που η μνήμη του host δεσμευόταν σε pageable memory διότι τα memory copies χρειάζονται pinned host memory. Επιπρόσθετα, χρησιμοποιεί DMA, γεγονός που κάνει τις μεταφορές μεταξύ host και device ακόμα πιο γρήγορες διότι πλέον δεν υπάρχει το overhead του fetching και decoding των εντολών, κάτι που η CPU θα έπρεπε να κάνει σε άλλη περίπτωση κατά τη διάρκεια του copy. Επιπλέον, ενισχύει την χρήση ασύγχρονων μεταφορών μνήμης και τη χρήση streams, τα οποία θα χρησιμοποιήσουμε σε επόμενο βήμα της εργασίας. Έτσι, τα memcpy μπορούν να γίνουν έως και δύο φορές πιο γρήγορα σε σχέση με πριν, κάτι που βελτιώνει την επίδοση σημαντικά.

Έχοντας κάνει αυτές τις βελτιστοποιήσεις και τις παραδοχές καλούμαστε να χρησιμοποιήσουμε τον profiler της nvidia ώστε να συγκρίνουμε την αρχική έκδοση κώδικα σε σχέση με την βελτιστοποιημένη έκδοση.

Αποτελέσματα profiler για αρχική-βελτιστοποιημένη έκδοση του κώδικα

Και για τους δύο κώδικες έγινε εκτέλεση με ακτίνα φίλτρου 32 και μέγεθος εικόνας 8192*8192 και η εκτέλεση έγινε με την εξής εντολή:

nvprof –o profile.metrics --analysismetrics ./exe

Το παραγόμενο .metrics αρχείο γίνεται import στον nvdia visual profiler ώστε να μπορεί να διαβαστεί και στις παρακάτω εικόνες παραθέτουμε μερικά από τα αποτελέσματα του τα οποία θεωρούμε ότι είναι πρόσφορα για σχολιασμό. Σε κάθε εικόνα εμφανίζονται στον κάτω πίνακα οι μετρήσεις για τον αρχικό κώδικα και στον πάνω για τον τελικό-βελτιστοποιημένο κώδικα. Οι μετρήσεις αφορούν τους kernels και συγκεκριμένα η πρώτη γραμμή τον kernelRow και η δεύτερη τον kernelColumn για τον εκάστοτε πίνακα.

Invocations	Avg. Duration
1	42,81735 ms
1	54,7238 ms

Invocations	Avg. Duration
1	197,789 ms
1	203,042 ms

Εικόνα 1: Χρόνοι εκτέλεσης των kernels

Παρατηρούμε ότι ο χρόνος εκτέλεσης των kernels στον βελτιστοποιημένο κώδικα είναι πολύ μικρότερος σε σχέση με τον αρχικό(περίπου υποτετραπλασιάζεται).

Device Memory Read Throughput	Device Memory Write Throughput
47.023 GB/s	15.672 GB/s
12.36 GB/s	12.262 GB/s

Device Memory Read Throughput	Device Memory Write Throughput
10.18 GB/s	3.392 GB/s
3.829 GB/s	3.305 GB/s

Εικόνα 2: Device memory read/write throughput

Στο βελτιστοποιημένο κώδικα βλέπουμε τετραπλασιασμό περίπου τόσο του read όσο και του wtite throughput όσον αφορά το device memory. Στην περίπτωση του read,αυτό οφείλεται κυρίως στην αντικατάσταση ενός μεγάλου αριθμού αναγνώσεων από την global memory για το φίλτρο με αναγνώσεις από την constant memory, οι οποίες είναι σημαντικά πιο γρήγορες. Επιπλέον, οι προσπελάσεις γίνονται παράλληλα όταν τα νήματα σε ένα warp προσπελαύνουν την ίδια θέση της constant memory, κάτι που ισχύει στην περίπτωσή μας και γι'αυτό αυξάνεται σημαντικά το throughput. Επίσης η αποσυμφόρηση της global memory καθώς και το βελτιστοποιημένο πρότυπο προσπέλασης, το οποίο εξηγείται στην βελτιστοποιήση 1, επιτρέπει το memory coalescing κατά την προσπέλαση threads στη global memory. Έτσι, επιτυγχάνεται συνολικά η μεταφορά περισσότερων δεδομένων στον ίδιο χρόνο.

Global Store Throughput	Global Load Throughput
12.538 GB/s	37.615 GB/s
9.81 GB/s	29.431 GB/s

Global Store Throughput	Global Load Throughput
2.714 GB/s	198.487 GB/s
2.644 GB/s	209.217 GB/s

Εικόνα 3:Global load/store Throughput.

Avg. Dynamic SMem
24576
24576

Avg. Dynamic SMem
0
0

Εικόνα 4: Αξιοποίηση της κοινόχρηστης μνήμης.

Σχετικά με την εικόνα 3, το μικρότερο global load throughput δεν συνεπάγεται χειρότερη επίδοση. Αντίθετα, αυτό συμβαίνει διότι πλέον δεν ζητάμε τον ίδιο όγκο δεδομένων από την global memory, καθώς χρησιμοποιείται η shared και η constant για μεγάλο πλήθος προσπελάσεων. Όσον αφορά την αύξηση του global store throughput, αυτό οφείλεται στην αποσυμφόρηση της global memory λόγω των προσπελάσεων που προαναφέρθηκαν, με αποτέλεσμα να έχουμε λιγότερα αιτήματα προς τη global memory και να μπορούμε να τα εξυπηρετήσουμε γρηγορότερα.

Στην εικόνα 4 φαίνεται η χρήση της shared memory στον βελτιστοποιημένο κώδικα, κάτι που δε συνέβαινε και στην προηγούμενη έκδοση.

Shared Memory Efficiency	Global Memory Load Efficiency	Global Memory Store Efficiency	L2 Throughput (L1 Reads)
99,9%	100%	100%	37.615 GB/s
52,3%	100%	100%	29.431 GB/s

Shared Memory Efficiency	Global Memory Load Efficiency	Global Memory Store Efficiency	L2 Throughput (L1 Reads)
0%	91,7%	100%	198.487 GB/s
0%	84,7%	100%	209.217 GB/s

Εικόνα 5

Η αντικατάσταση ενός μέρους των αναγνώσεων στην global memory με αναγνώσεις από την shared memory έχει ως αποτέλεσμα την αξιοποίηση της shared memory. Επίσης, αυξάνεται το Global Memory Load Efficiency και για τους δύο kernels. Αυτό συμβαίνει διότι πλέον έχουμε καλύτερο πρότυπο προσπέλασης στη global memory λόγω του ότι πλέον το φίλτρο προσπελαύνεται από την constant memory και δεν το διαβάζουμε από τη global. Συνεπώς, από τη global διαβάζουμε μόνο τον πίνακα εισόδου. Όσον αφορά την αύξηση του efficiency για τον kernelColumn, αυτό οφείλεται και στο γεγονός ότι στον τελικό κώδικα τα threads ενός warp προσπελαύνουν κατά γραμμές τον

πίνακα εισόδου για να φορτώσουν τα στοιχεία στη shared memory, όπως εξηγείται πιο αναλυτικά στη βελτιστοποίηση 1. Το Global Memory Store Efficiency παραμένει το ίδιο καθώς οι εγγραφές στην global memory παραμένουν ίδιες. Επίσης η μείωση στο throughput της L2 cache που αφορά τις αναγνώσεις από την L1 cache δεν συνεπάγεται μη αποδοτική προσπέλαση της μνήμης, διότι οφείλεται στην μείωση του αριθμού των συνολικών αναγνώσεων από την global memory και κατά συνέπεια από τις caches στον βελτιστοποιημένο κώδικα. Επιπλέον, αυτό μπορεί να οφείλεται και στο γεγονός ότι βελτιώνεται το πρότυπο προσπέλασης στην L1-cache, συνεπώς πηγαίνουμε λιγότερες φορές να διαβάσουμε από την L2. Τέλος παρατηρούμε αποδοτικότερη χρήση της shared memory στον kernelRow από ότι στον kernelColumn επειδή στον πρώτο το κάθε νήμα κάνει προσπέλαση του πίνακα στην shared memory κατά γραμμές ενώ στον δεύτερο κατά στήλες. Ο πίνακας είναι αποθηκευμένος κατά γραμμές τόσο στην global όσο και στην shared memory, με αποτέλεσμα στον kernelRow να έχουμε καλύτερο πρότυπο προσπέλασης.

Shared Load Transactions	Shared Store Transactions	Global Load Transactions	Global Store Transactions
136314880	6427830	12582912	4194304
266338304	6445845	12582912	4194304

Shared Load Transactions	Shared Store Transactions	Global Load Transactions	Global Store Transactions
0	0	408944640	4194304
0	0	534773760	4194304

Εικόνα 6

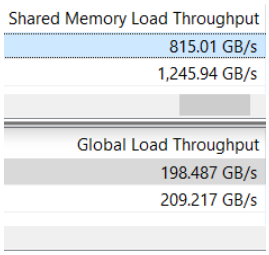
Αντικατάσταση μεγάλου αριθμού αναγνώσεων από την global memory με αναγνώσεις από την ταχύτερη shared memory, που έχει ως αποτέλεσμα τη μείωση των load transactions. Αυτό συνεπάγεται μείωση των Global Load Transactions στον βελτιστοποιημένο κώδικα και αποσυμφόρηση της global memory. Ο αριθμός των Global Store Transactions παραμένει ο ίδιος καθώς μόλις γίνουν οι υπολογισμοί από το κάθε Thread για το σημείο που του αντιστοιχεί το αποτέλεσμα αποθηκεύεται πίσω στην global memory τόσο στην αρχική όσο και στην βελτιστοποιημένη έκδοση του κώδικα.

L2 Read Transactions	L2 Write Transactions
50334789	16777231
50334908	16777229

L2 Read Transactions	L2 Write Transactions
1226843468	16777230
1327507133	16777228

Εικόνα 7:

Μειωμένες προσπελάσεις για ανάγνωση στην L2 cache στον τελικό κώδικα. Οι προσπελάσεις για εγγραφή παραμένουν οι ίδιες όπως εξηγείται και στην παραπάνω εικόνα.



Εικόνα 8

Επάνω φαίνεται το Shared Memory Load Throughput για τον βελτιστοποιημένο κώδικα και κάτω το Global Load Troughput για τον αρχικό κώδικα. Αν και δεν μπορούμε να κάνουμε άμεση σύγκριση του Load Throughput μεταξύ global και shared memory διότι διαφέρει ο αριθμός των προσπελάσεων για ανάγνωση στην κάθε μία, μπορούμε όμως να παρατηρήσουμε ότι το Load throughput στην shared memory είναι πολύ μεγαλύτερο, διότι οι προσπελάσεις σε αυτή γίνονται σημαντικά πιο γρήγορα από ότι στην global memory.

Μετρήσεις χρόνου εκτέλεσης για αρχική και βελτιστοποιημένη έκδοση κώδικα για είσοδο 8192x8192 και ακτίνα φίλτρου 32

Έκδοση Κώδικα	Αρχική	Βελτιστοποιημένη
Μέσος χρόνος εκτέλεσης (sec)	0.581626	0.255566
Τυπική απόκλιση (sec)	0.057425	0.013164

Σχολιασμός αποτελεσμάτων

Όσον αφορά τους χρόνους εκτέλεσης παρατηρούμε σημαντική μείωση που κάνει το χρόνο του νέου κώδικα x2.27 φορές πιο γρήγορο σε σχέση με τον αρχικό. Σε αυτό συνέβαλαν όλα όσα προαναφέρθηκαν σχετικά με τις βελτιστοποιήσεις που έγιναν.

Αποτελέσματα profiler για μεταφορές δεδομένων (HtoD & DtoH) για αρχική έκδοση κώδικα

για εικόνα 8192x8192 και ακτίνα φίλτρου 32

```
giliadis@csl-artemis:~/lab4$ nvprof ./old_conv
Enter filter radius : 32
Enter image size. Should be a power of two and greater than 65 : 8192
Image Width x Height = 8192 x 8192

Allocating and initializing host arrays...
==63754== NVPROF is profiling process 63754, command: ./old_conv
==63754== Warning: Auto boost enabled on device 0. Profiling results may be inconsistent.
CPU computation...
CPU_time is: 9.109779 seconds
GPU computation...
GPU_time is: 0.663646 seconds
Max difference in pos:3937604-> 0.000000 ->GPU: 28261394.104234, CPU: 28261394.104234
Check completed.
==63754== Profiling application: ./old_conv
==63754== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   28.04%  188.54ms         1  188.54ms  188.54ms  188.54ms  kernelRow(double*, double*, double*, int, int, int)
                   24.35%  163.73ms         2  81.866ms  1.8560us  163.73ms  [CUDA memcpy HtoD]
                   23.90%  160.70ms         1  160.70ms  160.70ms  160.70ms  kernelColumn(double*, double*, double*, int, int, int)
                   22.17%  149.06ms         1  149.06ms  149.06ms  149.06ms  [CUDA memcpy DtoH]
```

Αποτελέσματα profiler για μεταφορές δεδομένων (HtoD & DtoH) για βελτιστοποιημένη έκδοση κώδικα για εικόνα 8192x8192 και ακτίνα φίλτρου 32

```
giliadis@csl-artemis:~/lab4$ nvprof ./newconvHost
Enter filter radius : 32
Enter image size. Should be a power of two and greater than 65 : 8192
Image Width x Height = 8192 x 8192

Allocating and initializing host arrays...
==63771== NVPROF is profiling process 63771, command: ./newconvHost
==63771== Warning: Auto boost enabled on device 0. Profiling results may be inconsistent.
CPU computation...
CPU_time is: 8.885420 seconds
GPU computation...
==63771== Warning: Profiling results might be incorrect with current version of nvcc compiler used to compile
ing results. Ignore this warning if code is already compiled with the recommended nvcc version
GPU_time is: 0.276873 seconds
Max difference in pos:3937604-> 0.000000 ->GPU: 28261394.104234, CPU: 28261394.104234
Check completed.
==63771== Profiling application: ./newconvHost
==63771== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   33.25%  95.362ms         2  47.681ms  2.0160us  95.360ms  [CUDA memcpy HtoD]
                   28.88%  82.840ms         1  82.840ms  82.840ms  82.840ms  [CUDA memcpy DtoH]
```

Σχολιασμός αποτελεσμάτων

Η χρήση cudaHostAlloc αντί για malloc για την δέσμευση μνήμης δυναμικά στον host επιταχύνει σημαντικά τις μεταφορές μνήμης μεταξύ host και device όπως φαίνεται από τις παραπάνω εικόνες με τα αποτελέσματα του profiler. Στην πράξη, οι μεταφορές μνήμης που χρησιμοποιούν απευθείας pinned host memory γίνονται περίπου x1.7 και x1.8 φορές πιο γρήγορα σε σχέση με τον αρχικό κώδικα για HtoD και DtoH αντίστοιχα, κάτι που είναι λογικό να συμβαίνει όπως επεξηγήθηκε παραπάνω.

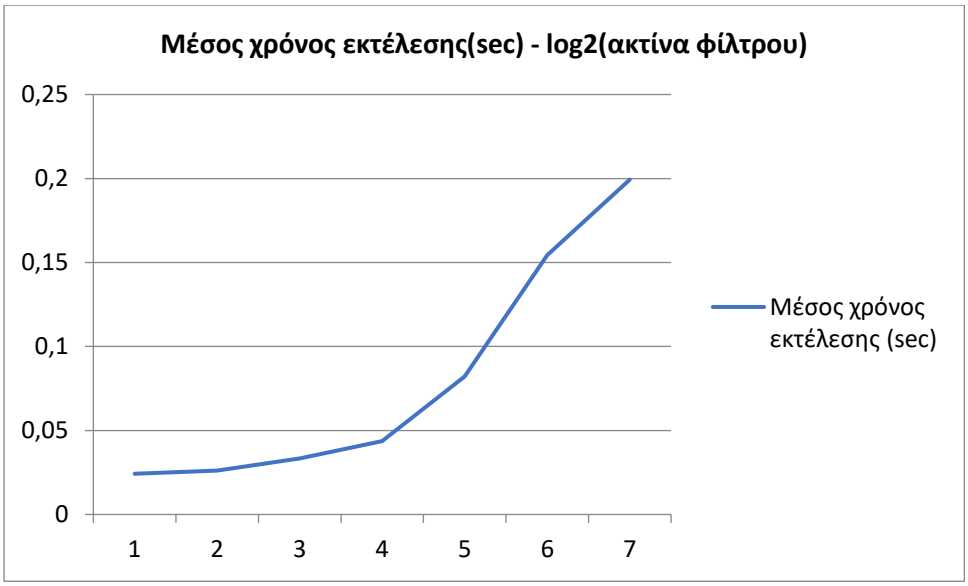
Βήμα 2

Σε αυτό το βήμα χρησιμοποιούμε σταθερή εικόνα μεγέθους **8192x8192**, δοκιμάζοντας διαφορετικές τιμές ακτίνας φίλτρου με σκοπό να παρατηρήσουμε το πώς επηρεάζει η ακτίνα φίλτρου το χρόνο εκτέλεσης των kernels καθώς και το λόγο χρόνου εκτέλεσης των kernels προς χρόνου μεταφορών μνήμης. Παρακάτω απεικονίζονται τα αντίστοιχα διαγράμματα και έπειτα σχολιάζονται τα αποτελέσματα των μετρήσεων για τις διαφορετικές τιμές του φίλτρου.

Χρόνοι εκτέλεσης των kernels για διαφορετικές ακτίνες φίλτρου

Ακτίνα φίλτρου	**2**	**4**	**8**	**16**	**32**	**64**	**80**
Μέσος χρόνος εκτέλεσης (sec)	0.024236	0.026215	0.033414	0.043743	0.08215	0.154472	0.199368
Τυπική απόκλιση (sec)	0.002816	0.003739	0.00415	0.006556	0.01312	0.017223	0.016191

Διάγραμμα χρόνου εκτέλεσης kernels - ακτίνας φίλτρου



Χρόνοι μεταφορών μνήμης(DtoH & HtoD) για διαφορετικές ακτίνες φίλτρου

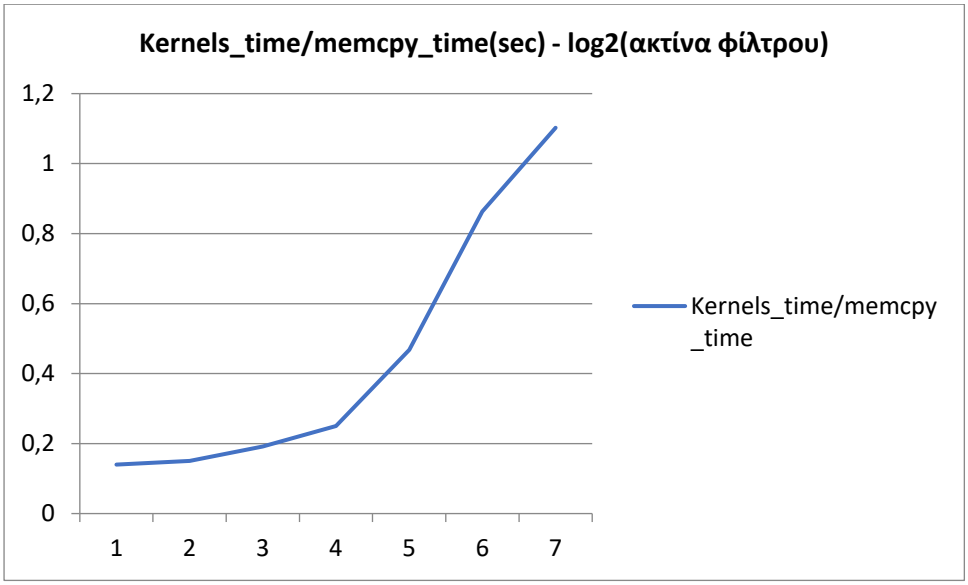
Ακτίνα φίλτρου	**2**	**4**	**8**	**16**	**32**	**64**	**80**
Μέσος χρόνος (sec)	0.173168	0.173946	0.174020	0.174644	0.175547	0.179006	0.180836
Τυπική απόκλιση (sec)	0.000804	0.001178	0.001021	0.001049	0.000417	0.000955	0.001677

Διάγραμμα χρόνου μεταφορών μνήμης - ακτίνας φίλτρου

Λόγος (kernels_time/memcpy_time) για διαφορετικές ακτίνες φίλτρου

Ακτίνα φίλτρου	**2**	**4**	**8**	**16**	**32**	**64**	**80**
Kernels_time/memcpy_time	0.139956574	0.1507076909	0.1920124124	0.2504695266	0.4679658439	0.8629431416	1.102479595

Διάγραμμα λόγου (kernels_time/memcpy_time) - ακτίνας φίλτρου



Συνολικός χρόνος εκτέλεσης για διαφορετικές ακτίνες φίλτρου

Ακτίνα φίλτρου	**2**	**4**	**8**	**16**	**32**	**64**	**80**
Μέσος χρόνος εκτέλεσης (sec)	0.197404	0.200201	0.207641	0.218387	0.257884	0.333475	0.380040
Τυπική απόκλιση (sec)	0.002836	0.003626	0.004245	0.006175	0.012793	0.017252	0.016219

Διάγραμμα Μέσου χρόνου εκτέλεσης(sec) – ακτίνα φίλτρου

Σχολιασμός αποτελεσμάτων

Από τις παραπάνω μετρήσεις μπορούμε να συμπεράνουμε ότι όσο μεγαλώνει η ακτίνα φίλτρου ο χρόνος εκτέλεσης των kernels αυξάνεται σημαντικά. Αυτό συμβαίνει διότι αν διπλασιάσουμε την ακτίνα φίλτρου το κάθε thread θα πρέπει να εκτελέσει διπλάσιες πράξεις σε σχέση με πριν καθώς διπλασιάζονται τα γειτονικά στοιχεία του πίνακα που πρέπει να συμπεριλάβει στους υπολογισμούς της συνέλιξης. Επίσης, λόγω της αύξησης του φίλτρου οι προσπελάσεις τόσο στη shared όσο και στην global memory αυξάνονται, κάτι που επιβαρύνει ακόμη περισσότερο το χρόνο εκτέλεσης των kernels. Όσον αφορά τις μεταφορές δεδομένων, ο διπλασιασμός του φίλτρου σε κάθε βήμα δεν μεγαλώνει σημαντικά το συνολικό πίνακα καθώς το extra padding που προστίθεται είναι αμελητέο σε σχέση με το πραγματικό μέγεθος πίνακα, ειδικά για μεγάλους πίνακες εισόδου. Αυτό έχει ως αποτέλεσμα να μην υπάρχει μεγάλη αύξηση του χρόνου των memory copies, καθώς αυξάνεται με πολύ μικρότερο ρυθμό σε σχέση με το ρυθμό αύξησης του χρόνου εκτέλεσης των kernels. Είναι φανερό και από τον πίνακα του λόγου (kernels_time/memcpy_time), στον οποίο σε κάθε βήμα όπου αυξάνουμε την ακτίνα φίλτρου πλησιάζει όλο και περισσότερο στη μονάδα, ενώ για τη μέγιστη ακτίνα που υποστηρίζουμε (80) παρατηρούμε ότι ο λόγος ξεπερνά τη μονάδα. Αυτό σημαίνει ότι σε αυτή την περίπτωση η εκτέλεση των kernels είναι πιο αργή σε σχέση με το χρόνο μεταφορών δεδομένων.

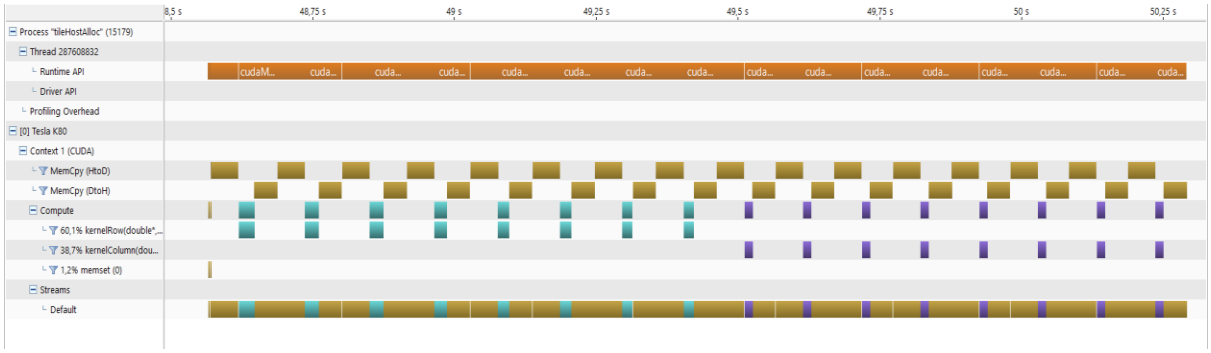
<u>Βήμα 3)</u>

Σε αυτό το βήμα πρέπει να υποστηρίξουμε πολύ μεγαλύτερες εικόνες στη GPU. Έτσι, η εκτέλεση των kernels “χωρίζεται” σε blocks του αρχικού πίνακα εισόδου, ώστε το κάθε block να χωράει στη GPU. Στην υλοποίησή μας το μέγεθος του block είναι παραμετρικό και δίνεται από το χρήστη κατά την εκτέλεση του προγράμματος. **Στο βήμα αυτό έχουμε κάνει την παραδοχή ότι το μέγεθος του block θα είναι πάντα ακέραιο πολλαπλάσιο του ImageWidth και θα είναι δύναμη του 2, ώστε να μεταφέρουμε σε κάθε βήμα μία “ταινία” γραμμών στον kernel της GPU**. Υπολογίζουμε τον συνολικό αριθμό από tiles που θα έχουμε για το δεδομένο μέγεθος του πίνακα εισόδου και έπειτα σε for-loop υλοποιείται η blocked συνέλιξη τόσο για το kernelRow όσο και για το kernelColumn. Επίσης, σε κάθε βήμα εκτός από τη μνήμη για ένα tile φορτώνεται και επιπλέον μνήμη που αφορά

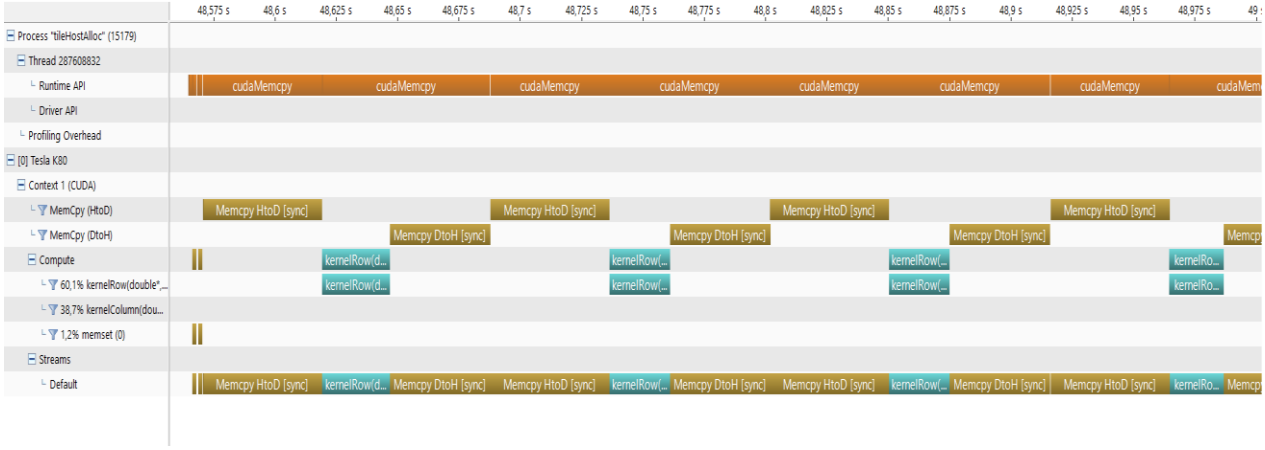
τα δεξιά και αριστερά filterR στοιχεία καθώς και επιπλέον μνήμη για τα filterR στοιχεία που βρίσκονται πάνω και κάτω για κάθε tile, διότι είναι απαραίτητα για να γίνουν σωστά οι υπολογισμοί του τελικού tile-αποτελέσματος της εικόνας εξόδου. **Η γεωμετρία που ορίζουμε στο συγκεκριμένο πρόβλημα είναι η εξής:** Εάν η εικόνα χωράει σε ένα thread block , τότε χρησιμοποιούμε ένα μοναδικό τετράγωνο block(imageW*imageH), συνεπώς και 1 tile. Σε άλλη περίπτωση, το grid που χρησιμοποιούμε θα έχει συγκεκριμένο αριθμό από blocks στον άξονα x, με sqrt(maxThreadsPerBlock) νήματα στον άξονα x του block ώστε να καλύπτει ολόκληρες τις γραμμές της εικόνας εισόδου, ενώ στον άξονα των y του grid θα έχουμε κατάλληλο αριθμό από blocks ώστε να συμπληρωθεί σωστά η επιθυμητή "ταινία" γραμμών. Πιο συγκεκριμένα, στον άξονα y, εάν το tile έχει περισσότερες γραμμές από sqrt(maxThreadsPerBlock), τότε βάζουμε sqrt(maxThreadsPerBlock) νήματα στον άξονα των y του κάθε block και ορίζουμε grid.y ίσο με ((#γραμμών ενός tile) / block.y), διαφορετικά βάζουμε σε κάθε block στον άξονα y τόσα νήματα όσο και το πλήθος γραμμών που αντιστοιχεί σε ένα tile. Σε κάθε περίπτωση καταλήγουμε να έχουμε ένα ορθογώνιο grid για την «ταινία» γραμμών.

Σε αυτό το σημείο καταγράφουμε το timeline και κάποια βασικά στοιχεία του profiling για να τα συγκρίνουμε με το επόμενο ερώτημα, στο οποίο γίνεται χρήση streams για καλύτερη εκμετάλλευση των πόρων της GPU.

Timeline για την έκδοση με tiled convolution χωρίς streams για ακτίνα φίλτρου 32, tile 33554432 και μέγεθος πίνακα 16384x16384



Μεγέθυνση στιγμιοτύπου



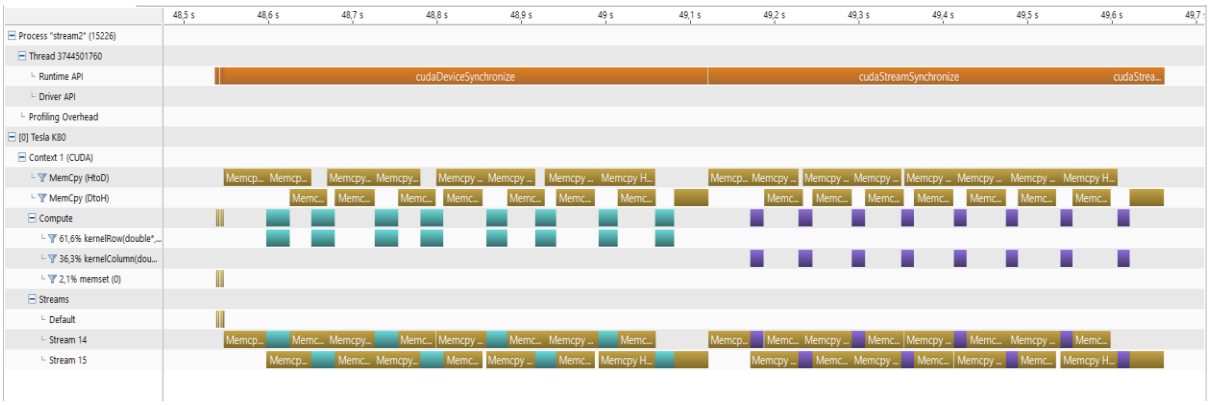
Πίνακας για χρόνο εκτέλεσης GPU για εικόνα 16384x16384, ακτίνα φίλτρου 32 και μέγεθος tile 67108864

Μέγεθος εικόνας	16384x 16384
Ακτίνα φίλτρου	32
Μέγεθος tile	67108864
Μέσος χρόνος εκτέλεσης(sec)	1.706560
Τυπική απόκλιση (sec)	0.007742

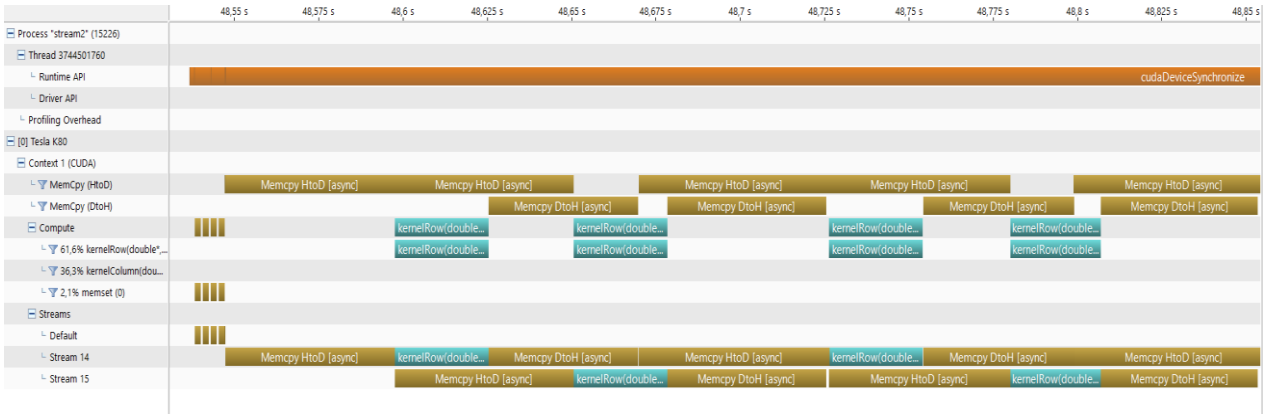
Βήμα 4)

Στο βήμα αυτό προσπαθούμε να επικαλύψουμε διαφορετικές εκτελέσεις kernels μεταξύ τους, καθώς και υπολογισμούς με μεταφορές δεδομένων. Λόγω της φύσης του προβλήματος, ο αριθμός των tiles θα είναι πάντα άρτιος, επομένως θα διαιρείται ακριβώς με το 2 εκτός από την περίπτωση που ο πίνακας χωράει σε ένα μόνο tile. Το γεγονός αυτό μας επιτρέπει να εφαρμόσουμε στα 2 for-loops του προηγούμενου βήματος loop unrolling κατά παράγοντα 2, ενώ ταυτόχρονα δημιουργούμε 2 διαφορετικά streams τα οποία μας βοηθούν να επικαλύψουμε τις μεταφορές δεδομένων κατά τη διάρκεια των οποίων η GPU μένει αδρανής καθώς περιμένει τα δεδομένα ενός tile, με σκοπό να κρύψουμε το latency των memory copies (τα οποία είναι αρκετά “ακριβά” σχετικά με την επίδοση) μέσω υπολογισμών που αφορούν διαφορετικά tiles. Αυτό έχει ως αποτέλεσμα την καλύτερη αξιοποίηση των πόρων της GPU σε σχέση με το προηγούμενο βήμα όπου δεν υπήρχε καμία επικάλυψη. Στην περίπτωση που το πλήθος των tiles είναι 1, τότε απλώς εκτελείται μόνο ο κώδικας για το πρώτο tile και έπειτα βγαίνουμε από την επανάληψη. Έπειτα από το τέλος των kernelRow, περιμένουμε μέχρι όλα τα kernel invocations να έχουν ολοκληρωθεί πριν αρχίσουμε τους υπολογισμούς των kernelColumn. Όταν ολοκληρωθεί και το 2ο for-loop χρησιμοποιούμε συγχρονισμό για να βεβαιωθούμε ότι ο host περιμένει έως ότου το κάθε stream τελειώσει όλους τους υπολογισμούς και τις μεταφορές δεδομένων καθώς αυτά γίνονται ασύγχρονα ως προς τον host. Επίσης, εφόσον το unroll γίνεται κατά 2, είναι αναγκαίο να δεσμεύσουμε 2 φορές τον κάθε πίνακα που χρειαζόμασταν στο προηγούμενο βήμα ώστε οι ενέργειες που επικαλύπτονται χρονικά να λειτουργούν σε διαφορετικά δεδομένα και να μην υπάρχει καμία εξάρτηση μεταξύ τους.

Timeline για έκδοση κώδικα με tiled convolution και χρήση streams για ακτίνα φίλτρου 32, tile 33554432 και μέγεθος πίνακα 16384x16384



Μεγέθυνση στιγμιοτύπου



Πίνακας για χρόνο εκτέλεσης GPU για εικόνα 16384x16384, ακτίνα φίλτρου 32 και μέγεθος tile 67108864 θέσεων του πίνακα εισόδου.

Μέγεθος εικόνας	16384x16384
Ακτίνα φίλτρου	32
Μέγεθος tile	67108864
Μέσος χρόνος εκτέλεσης(sec)	1.194021
Τυπική απόκλιση (sec)	0.008168

<u>Σχολιασμός αποτελεσμάτων και σύγκριση με το προηγούμενο ερώτημα</u>

Σχετικά με τους χρόνους εκτέλεσης, παρατηρούμε ότι με τη χρήση streams επιτυγχάνουμε για δεδομένη είσοδο x1.43 περίπου φορές καλύτερο χρόνο εκτέλεσης. Όπως φαίνεται από τα timelines καθώς και τους πίνακες χρόνου εκτέλεσης, ο κώδικας με την χρήση streams είναι πολύ πιο αποδοτικός σε σχέση με την προηγούμενη έκδοση. Αυτό είναι αναμενόμενο, καθώς στον ίδιο χρόνο αξιοποιούμε καλύτερα όλους τους πόρους της GPU και καταφέρνουμε να “κρύψουμε” το latency των memory copies, διότι ταυτόχρονα μπορούμε να εκτελούμε και υπολογισμούς, έχοντας για όσο το δυνατό περισσότερο χρόνο την GPU ενεργή. Στο γεγονός αυτό συμβάλλει και το ότι το σύστημα csl-artemis διαθέτει 2 copy engines, όπως φαίνεται και από το deviceQuery, κάτι που μπορεί να μας δώσει τη δυνατότητα για ταυτόχρονη μεταφορά μνήμης HtoD και DtoH, ενισχύοντας την παραλληλοποίηση που μπορούμε να πετύχουμε. Επίσης, το γεγονός ότι χρησιμοποιούμε pinned host memory, δίνει την ευκολία για μεγαλύτερο ταυτοχρονισμό, καθώς σε κάθε βήμα η μνήμη που χρειάζεται για την εκτέλεση του tile μεταφέρεται άμεσα. Αυτό οφείλεται στο γεγονός ότι χρησιμοποιούνται DMA Engines και επιπλέον παρακάμπτουμε το extra copy από pageable memory σε pinned memory για τη μνήμη του host. Με αυτό τον τρόπο, επιτυγχάνουμε επικάλυψη διότι επικαλύπτουμε μεταφορές δεδομένων HtoD και DtoH, και επιπλέον υπολογισμούς kernels με μεταφορές δεδομένων και από τη στιγμή που ξεκινάει η εκτέλεση 2 ή περισσότερων tiles το pipeline γεμίζει έως ότου να φτάσει να υπολογίζεται το τελευταίο tile όπου εκεί “αδειάζει” το pipeline. Τα παραπάνω ισχύουν τόσο για τον kernelRow όσο και για τον kernelColumn.

Βήμα 5)

Το μέγιστο μέγεθος εικόνας που μπορούμε να υποστηρίξουμε πλέον είναι **65536x65536**. Ο πόρος που μας περιορίζει πλέον είναι η μνήμη της CPU, καθώς αν προσπαθήσουμε να δεσμεύσουμε τον αμέσως μεγαλύτερο πίνακα που είναι δύναμη του 2, διαστάσεων 131072x131072 η cudaHostAlloc αποτυγχάνει διότι η μνήμη που ζητήσαμε δεν μπορεί να δεσμευθεί.

Συμπερασματικά, στην συγκεκριμένη εργασία βλέπουμε ότι μέσω των tiles μπορούμε να υποστηρίξουμε πολύ μεγαλύτερες εικόνες σε σχέση με την προηγούμενη έκδοση κώδικα, ενώ ταυτόχρονα με τη χρήση των CUDA streams αξιοποιούμε όσο το δυνατό καλύτερα τον ταυτοχρονισμό, κάτι που έχει μεγάλη επίδραση στην συνολική επίδοση του προγράμματος.