ΤΜΉΜΑ ΗΛΕΚΤΡΟΛΌΓΩΝ ΜΗΧΑΝΙΚΏΝ ΚΑΙ ΜΗΧΑΝΙΚΏΝ ΥΠΟΛΟΓΙΣΤΏΝ

ΠΑΝΕΠΙΣΤΉΜΙΟ ΘΕΣΣΑΛΊΑΣ

CE421 ΣΥΣΤΉΜΑΤΑ ΥΠΟΛΟΓΙΣΜΟΎ ΥΨΗΛΏΝ ΕΠΙΔΌΣΕΩΝ

Πουρναρόπουλος Φοίβος, ΑΕΜ: 2614

Ηλιάδης Γρηγόρης, ΑΕΜ: 2522

Στη συγκεκριμένη εργασία καλούμαστε να γράψουμε κώδικα σε περιβάλλον CUDA με σκοπό την εφαρμογή ενός δισδιάστατου διαχωρίσιμου φίλτρου συνέλιξης σε έναν δισδιάστατο πίνακα εισόδου.

Καταγραφή αποτελεσμάτων του deviceQuery στο σύστημα csl-artemis

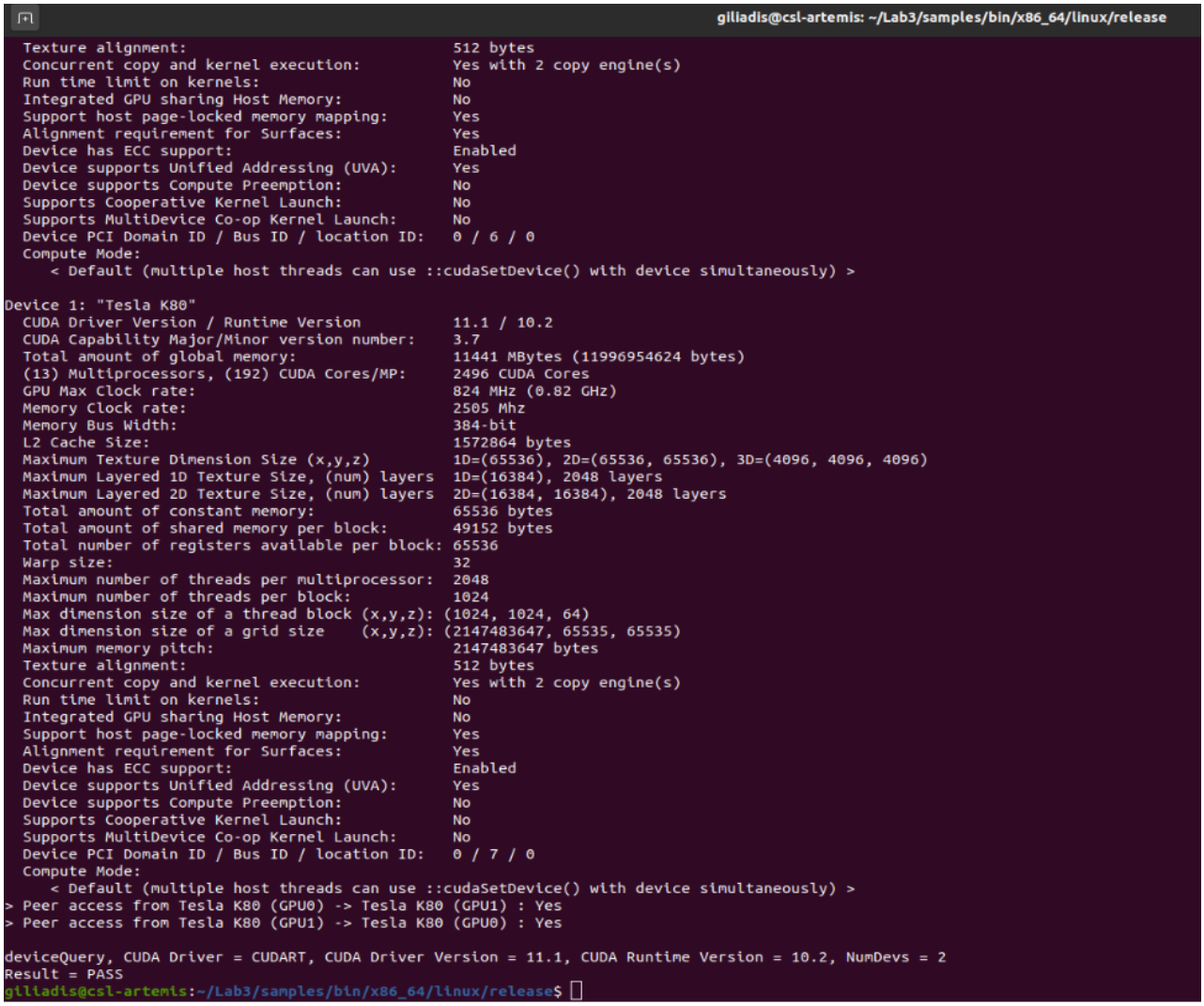
```
giliadis@csl-artemis: ~/Lab3/samples/bin/x86_64/linux/release
concurrentKernels      fastWalshTransform          matrixMul_nvrtc         quasirandomGenerator        simpleDrvRuntime
conjugateGradient      FDTD3d                      MC_EstimatePiInlineP    quasirandomGenerator_nvrtc  simpleGL
giliadis@csl-artemis:~/Lab3/samples/bin/x86_64/linux/release$ ./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
```



Βήμα 1

Το flag που χρησιμοποιήθηκε για τη μεταγλώττιση των προγραμμάτων είναι το -O4, έτσι ώστε ο κώδικας της CPU να μεταγλωττίζεται με το μέγιστο αριθμό βελτιστοποιήσεων. Στη μεταγλώττιση που γίνεται με σκοπό τις μετρήσεις, δεν βάζουμε το flag -G διότι δεν επιτρέπει τις βελτιστοποιήσεις του compiler. Χρησιμοποιείται μόνο για το στάδιο του debug, εφόσον δεν μας απασχολεί η επίδοση του κώδικα στο συγκεκριμένο στάδιο.

Βήμα 2

Στο βήμα αυτό, γράψαμε κώδικα για τη GPU ώστε να χρησιμοποιείται ένα μοναδικό block. Πιο συγκεκριμένα, δεσμεύουμε την κατάλληλη μνήμη στο device και μεταφέρουμε σε αυτό τα απαραίτητα δεδομένα. Εκτελούμε 2 kernels, ο πρώτος υλοποιεί συνέλιξη κατά γραμμές και ο δεύτερος κατά στήλες. Μόλις τελειώσουν οι kernels επιστρέφουμε το αποτέλεσμα πίσω στον host και συγκρίνουμε τα αποτελέσματα της GPU σε σχέση με αυτά της CPU. Εάν η ακρίβεια που ορίσαμε δεν ικανοποιείται, τότε τερματίζουμε το πρόγραμμα. Σε όλες τις πιθανές περιπτώσεις που τερματίζεται το πρόγραμμα αποδεσμεύεται όλη η δυναμικά δεσμευμένη μνήμη και γίνεται reset στο device.

Βήμα 3

Α) Στο συγκεκριμένο ερώτημα χρησιμοποιήθηκε ακτίνα φίλτρου ίση με 4. Μέσα από πειράματα διαφόρων τιμών για τη διάσταση του πίνακα εισόδου προκύπτει ότι το μέγιστο μέγεθος εικόνας που μπορούμε να υποστηρίξουμε χωρίς να συμβεί κάποιο σφάλμα εκτέλεσης είναι 32x32. Αυτό συμβαίνει διότι ο κώδικας χρησιμοποιεί μόνο ένα block για τον υπολογισμό της εφαρμογής του φίλτρου στον πίνακα, γεγονός που περιορίζει το μέγεθος πίνακα να είναι παντα μικρότερο ή ίσο από τα συνολικά threads που χωράνε σε ένα block της GPU. Πιο συγκεκριμένα, κάθε νήμα υπολογίζει ένα στοιχείο του τελικού πίνακα-αποτελέσματος. Όταν δώσουμε μέγεθος μεγαλύτερο από τα threads που μπορεί να υποστηρίξει ένα block, το πρόγραμμα τερματίζει με σφάλμα χρόνου εκτέλεσης, που ελέγχεται σε κάθε κλήση kernel σε συνδυασμό με την

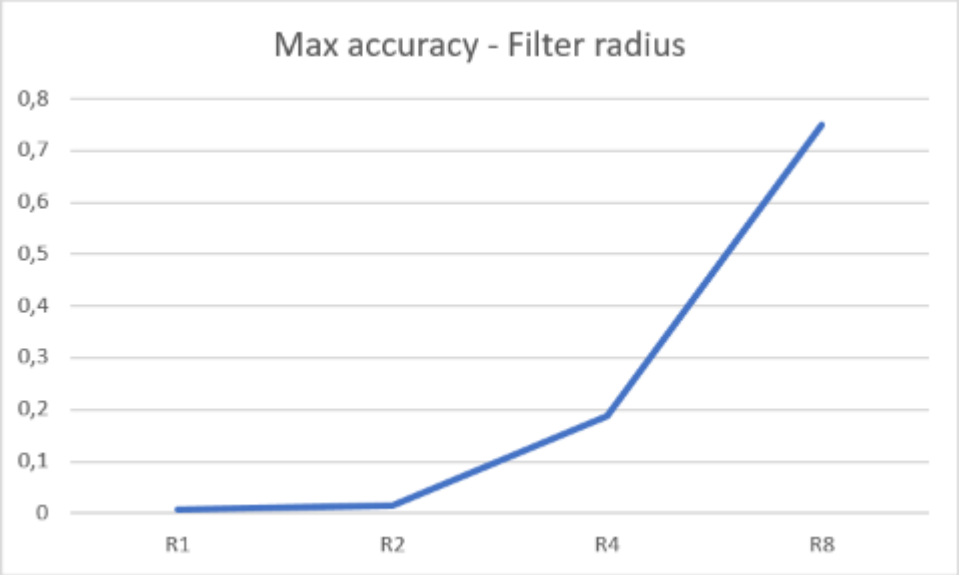
κλήση cudaDeviceSynchronize() . Αυτό γίνεται ώστε ο host να περιμένει να τελειώσει ο kernel πριν ελέγξει εάν έχει συμβεί κάποιο σφάλμα, εφόσον τα kernel launches είναι ασύγχρονα σε σχέση με τον host. Συνεπώς, η GPU μας περιορίζει σε μέγιστο μέγεθος 32x32.

Β) Για το μέγιστο μέγεθος που μπορούμε να υποστηρίξουμε χωρίς σφάλμα χρόνου εκτέλεσης(32x32), η μέγιστη ακρίβεια που μπορεί να υποστηριχθεί χωρίς να παρουσιάζονται σφάλματα σύγκρισης σε σχέση με το μέγεθος του φίλτρου παρουσιάζεται στον παρακάτω πίνακα. Στο συγκεκριμένο ερώτημα, χρησιμοποιούμε ακτίνες φίλτρου που είναι δυνάμεις του 2. Επίσης, κατ' εξαίρεση, εξετάζουμε τι συμβαίνει στην περίπτωση που η ακτίνα ισούται με 15, ώστε να έχουμε περισσότερα δείγματα και το μέγεθος φίλτρου να προσεγγίζει όσο το δυνατό περισσότερο το μέγεθος του πίνακα. Σε όλα τα πειράματα που ακολουθούν, ο τρόπος με τον οποίο βρίσκουμε τη μέγιστη ακρίβεια που μπορούμε να υποστηρίξουμε χωρίς σφάλματα σύγκρισης είναι παίρνοντας τη διαφορά μεταξύ του κάθε στοιχείου του πίνακα αποτελέσματος της GPU και της αντίστοιχης θέσης της CPU. Η μέγιστη διαφορά που προκύπτει μετά από αυτό τον έλεγχο είναι και η μέγιστη ακρίβεια που μπορούμε να υποστηρίξουμε, καθώς δεν εξαρτάται από το πλήθος των εκτελέσεων της εφαρμογής στη GPU διότι ο πίνακας input θα έχει πάντα τις ίδιες τιμές λόγω του σταθερού seed στη συνάρτηση srand και μπορούμε να βγάλουμε συμπεράσματα με μία μόνο εκτέλεση.

Πίνακας

Ακτίνα φίλτρου	1	2	4	8	15
Μέγιστη ακρίβεια	0,007812	0,015625	0,1875	0,75	1

Διάγραμμα



Σχολιασμός αποτελεσμάτων: Παρατηρούμε ότι όσο αυξάνεται το μέγεθος του φίλτρου, η ακρίβεια που υποστηρίζεται αυξάνεται σημαντικά μεταξύ δύο διαδοχικών τιμών ακτίνας φίλτρου που είναι δυνάμεις του 2. Το ίδιο ισχύει και για ακτίνα ίση με 15 σε σχέση με ακτίνα φίλτρου 8. Αυτό οφείλεται στις βελτιστοποιήσεις που κάνει η GPU στις πράξεις κινητής υποδιαστολής. Πιο συγκεκριμένα, οι GPUs παρέχουν λειτουργίες FMAD και FMA(Fused Multiply-Add), οι οποίες συγχωνεύουν μία πράξη πολλαπλασιασμού και μία πράξη πρόσθεσης που είναι εξαρτημένες μεταξύ τους, για αριθμούς διπλής και απλής ακρίβειας. Αυτό βοηθάει στην επίδοση, διότι αυτή η πράξη διαρκεί όσο θα διαρκούσε μία από τις 2 πράξεις αν εκτελούνταν διαδοχικά. Όμως, ο τρόπος στρογγυλοποίησης είναι διαφορετικός από το αν τις εφαρμόζαμε ξεχωριστά. Αυτό έχει ως αποτέλεσμα να βλέπουμε μεγαλύτερη διαφορά μεταξύ των αποτελεσμάτων της GPU και της CPU, γεγονός που μειώνει την ακρίβεια που μπορούμε να υποστηρίξουμε. Επίσης, όσο αυξάνεται η ακτίνα φίλτρου, τόσο περισσότερες πράξεις (πολλαπλασιασμοί και προσθέσεις) γίνονται για τον υπολογισμό του κάθε pixel διότι τα γειτονικά pixels που συμμετέχουν στη συνέλιξη είναι περισσότερα.

4) Το πρόβλημα που αντιμετωπίσαμε στο προηγούμενο ερώτημα οφείλεται στον περιορισμό των Threads που μπορεί να υποστηρίξει ένα block. Για τον λόγο αυτό χρησιμοποιήσαμε πολλαπλά blocks οργανωμένα σε ένα grid. Αυτό λύνει το παραπάνω πρόβλημα καθώς μας δίνει τη δυνατότητα να χρησιμοποιήσουμε πολύ περισσότερα threads συνολικά εκμεταλλευόμενοι επιπλέον blocks στην γεωμετρία μας. Επειδή οι εικόνες είναι τετραγωνικές και έχουν μέγεθος που είναι δύναμη του 2, δημιουργούμε (image_size/size_of_block) blocks μεγέθους ίσου με τον μέγιστο αριθμό threads per block, που είναι επίσης δύναμη του 2. Με βάση αυτούς τους περιορισμούς επιλέγουμε το grid να έχει πάντα τετραγωνική μορφή. Σε αυτή την έκδοση κώδικα, το κάθε thread χρησιμοποιεί και το blockIdx εκτός από το threadIdx, εφόσον πλέον έχουμε πολλαπλά blocks που διαχειρίζονται τους πίνακες εισόδου και αποτελέσματος. Σύμφωνα με το deviceQuery του συστήματος στο οποίο εκτελέστηκαν τα πειράματα το μέγιστο μέγεθος των διατάσεων του grid

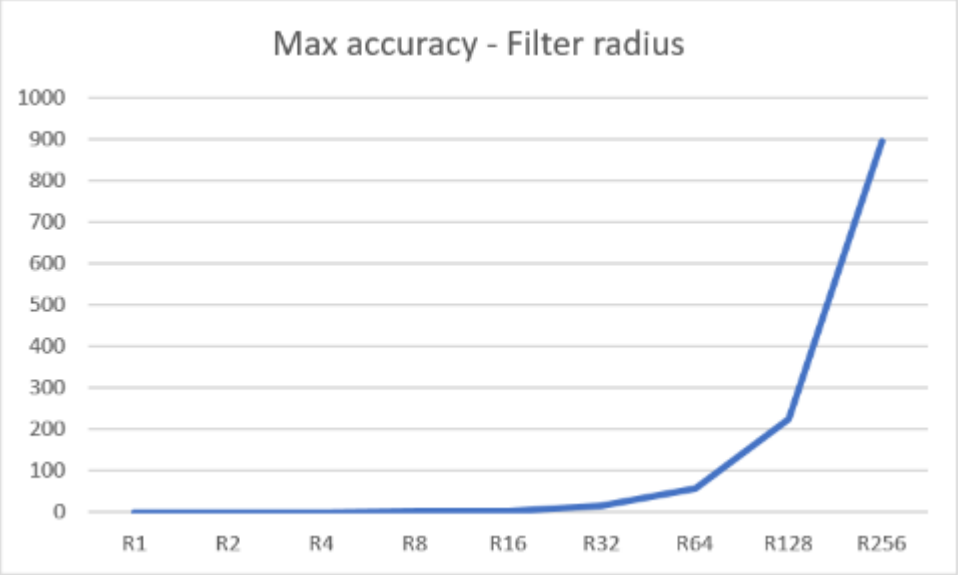
είναι (x,y,z)=(2147483647,65535,65535). Παρ' όλα αυτά διαπιστώσαμε ότι μπορούμε να υποστηρίξουμε μεγέθη εικόνας έως 16384x16384 γεγονός που οφείλεται στο συνολικό μέγεθος της global memory του συστήματος το οποίο είναι σύμφωνα με το deviceQuery 11441 MB. Για αυτό και για τετραγωνικούς πίνακες με διαστάσεις που είναι δυνάμεις του 2 και μέγεθος μεγαλύτερο από 16384x16384 προκύπτει σφάλμα χρόνου εκτέλεσης στην cudaMalloc. Στη συγκεκριμένη έκδοση κώδικα χρησιμοποιήσαμε τη συνάρτηση cudaGetDevice ώστε να πάρουμε το id της συσκευής που χρησιμοποιείται, και στη συνέχεια μέσω της cudaGetDeviceProperties παίρνουμε τον μέγιστο αριθμό threads που υποστηρίζονται ανά block.

5) α) Η μέγιστη ακρίβεια σε σχέση με το μέγεθος του φίλτρου για το μέγιστο μέγεθος εικόνας(16384x16384) υπολογίστηκε με τον ίδιο τρόπο όπως στο ερώτημα 3. Τα μεγέθη του φίλτρου είναι δυνάμεις του δύο και το μέγιστο μέγεθός του περιορίζεται από το μέγιστο μέγεθος εικόνας. Παρ' όλα αυτά στις μετρήσεις συμπεριλαμβάνουμε μέχρι και φίλτρο μεγέθους 256 γιατί για μεγαλύτερο μέγεθος ήταν υπερβολικά χρονοβόρες οι μετρήσεις. Παρακάτω παρατίθενται ο πίνακας και το διάγραμμα που προέκυψε από τις μετρήσεις.

Πίνακας

Ακτίνα φίλτρου	1	2	4	8	16	32	64	128	256
Μέγιστη ακρίβεια	0.007812	0.046875	0.375	1.5	4.0	14.0	56.0	224.0	896.0

Διάγραμμα



Σχολιασμός αποτελεσμάτων:

Παρατηρούμε εκθετική αύξηση της μέγιστης ακρίβειας που οδηγεί σε επιτυχείς συγκρίσεις μεταξύ αποτελεσμάτων CPU και GPU. Αυτό όπως εξηγείται και στο ερώτημα 3 οφείλεται στις βελτιστοποιήσεις FMA/FMAD που πραγματοποιεί η GPU. Το αποτέλεσμα της συνδυαστικής πράξης πρόσθεσης και πολλαπλασιασμού διαφέρει από το συνολικό αποτέλεσμα των πράξεων αν αυτές εκτελούνταν ξεχωριστά. Συνεπώς, όσο αυξάνουμε το μέγεθος του φίλτρου αυξάνεται και ο αριθμός των συνδυαστικών πράξεων λόγω της συνεισφοράς περισσότερων γειτονικών pixels για τον υπολογισμό κάθε στοιχείου. Αυτό έχει ως αποτέλεσμα να αυξάνεται και η απόκλιση των αποτελεσμάτων εξαιτίας του “αθροίσματος” των επιμέρους αποκλίσεων.

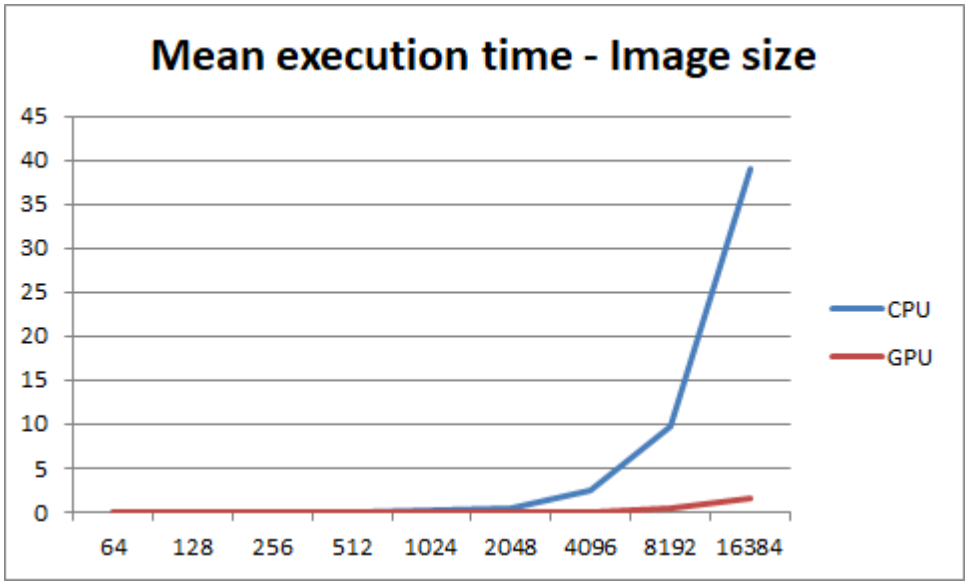
Στη συγκεκριμένη εργασία υπολογίζουμε το απόλυτο σφάλμα, δηλαδή την διαφορά μεταξύ των αποτελεσμάτων της CPU σε σχέση με τα αποτελέσματα της GPU στις αντίστοιχες θέσεις. Στην πραγματικότητα, τόσο η CPU όσο και η GPU επηρεάζονται από σφάλματα ακρίβειας λόγω των στρογγυλοποιήσεων. Γι’ αυτό, θα ήταν προτιμότερο να χρησιμοποιήσουμε συστήματα μεγαλύτερης ακρίβειας εάν θέλουμε να δούμε το σχετικό σφάλμα μεταξύ των αποτελεσμάτων CPU, GPU και του πραγματικού αποτελέσματος των πράξεων, ώστε να έχουμε καλύτερη εικόνα αναφορικά με τα σφάλματα. Επιπλέον, όσο περισσότερες πράξεις εφαρμόζουμε για να υπολογίσουμε μία θέση του πίνακα αποτελέσματος, τόσο καταλήγουμε να χρησιμοποιούμε αριθμούς οι οποίοι απέχουν πιο πολύ από το μηδέν. Το παραπάνω σε συνδυασμό με το γεγονός ότι η αναπαράσταση γίνεται πιο “αραιή” όσο απομακρυνόμαστε από το μηδέν αυξάνουν την πιθανότητα να καταλήξουμε σε αριθμούς που δεν αναπαρίστανται ακριβώς από το σύστημα. Επίσης, οι διαφορετικές στρογγυλοποιήσεις που μπορεί να εφαρμόζονται μεταξύ CPU και GPU, συμβάλλουν στο να υπάρχει μεγαλύτερη πιθανότητα οι τελικοί αριθμοί CPU και GPU να αναπαρασταθούν με βάση διαφορετικούς πλησιέστερους αριθμούς που αναπαρίστανται ακριβώς. Αυτό το φαινόμενο συμβάλλει δραστικά στο να αυξάνονται συνεχώς τα σφάλματα όσο αυξάνουμε την ακτίνα φίλτρου που χρησιμοποιούμε. Τα ίδια συμπεράσματα ισχύουν και για ερώτημα 3β που προηγήθηκε, όμως στο συγκεκριμένο χρησιμοποιήσαμε το νέο μέγιστο μέγεθος πίνακα που μπορεί να διαχειριστεί η GPU, δηλαδή 16384x16384, γεγονός που αυξάνει σε μεγάλο βαθμό το πλήθος των πράξεων και κάνει πιο ορατά τα αποτελέσματα.

5) β) Στο ακόλουθο ερώτημα μας ζητείται να κατασκευάσουμε το διάγραμμα χρόνου εκτέλεσης CPU και GPU ως συνάρτηση του μεγέθους του πίνακα εισόδου. Εκτελέσαμε το κάθε πρόγραμμα 12 φορές και πήραμε μέσο χρόνο και τυπική απόκλιση, αφαιρώντας το μέγιστο και ελάχιστο χρόνο εκτέλεσης από τα δείγματα. Θεωρήσαμε ότι για την εργασία είναι καλό να χρησιμοποιήσουμε ακρίβεια μseconds για τον υπολογισμό του χρόνου εκτέλεσης. Για το συγκεκριμένο ερώτημα χρησιμοποιήσαμε ακτίνα φίλτρου ίση με 16. Στον χρόνο εκτέλεσης της GPU έχει συμπεριληφθεί και ο χρόνος για τη μεταφορά μνήμης προς/από τον host.

Πίνακας

Ακτίνα φίλτρου	64	128	256	512	1024	2048	4096	8192	16384
CPU Μέσος χρόνος (sec)	0.000375	0.001519	0.006415	0.028091	0.137460	0.564021	2.424804	9.759788	38.994835
Τυπική απόκλιση	0.000005	0.000013	0.000067	0.000222	0.001406	0.008137	0.013210	0.065369	0.304757
GPU Μέσος χρόνος (sec)	0.000126	0.000217	0.000535	0.001894	0.006402	0.024528	0.115516	0.395977	1.657203
Τυπική απόκλιση	0.000004	0.000004	0.000008	0.000018	0.000059	0.000170	0.000187	0.035600	0.061844

Διάγραμμα



Σχολιασμός αποτελεσμάτων

Από τις παραπάνω μετρήσεις συμπεραίνουμε ότι σε όλες τις δοκιμές ο χρόνος της GPU είναι μικρότερος από το χρόνο της CPU για το ίδιο workload. Πιο συγκεκριμένα, στο πρώτο βήμα παρατηρούμε speedup κατά ένα παράγοντα 2.9x. Όσο αυξάνουμε το workload, είναι φανερό ότι το speedup που πετυχαίνουμε χρησιμοποιώντας την GPU ολοένα και αυξάνεται. Στο τελευταίο στάδιο με μέγεθος πίνακα 16384x16384 φτάνει να έχει speedup ίσο με 23.5x .Αυτό είναι λογικό καθώς η GPU παρέχει παραλληλισμό σε αντίθεση με τη CPU, όπου ο κώδικας εκτελείται σειριακά.

Στην εκτέλεση της GPU το κάθε νήμα αναλαμβάνει να υπολογίσει τη συνέλιξη για ένα συγκεκριμένο σημείο του πίνακα-αποτελέσματος με βάση τη θέση του στη γεωμετρία τόσο του block όσο και του grid. Συνεπώς, επιτυγχάνουμε πολλές περισσότερες πράξεις ταυτόχρονα σε σχέση με τη CPU, δεδομένου ότι τα διάφορα blocks του grid δρομολογούνται σε διαφορετικούς streaming multiprocessors και επιπλέον μέσα σε κάθε streaming multiprocessor τα threads ενός warp

εκτελούν τις ίδιες εντολές ταυτόχρονα. Τέλος, οι βελτιστοποιήσεις FMA/FMAD συμβάλουν στην μείωση του χρόνου της GPU, καθώς στον χρόνο που η CPU εκτελεί μία πράξη πολλαπλασιασμού ή πρόσθεσης, η GPU θα έχει υπολογίσει και τους 2 στον ίδιο χρόνο. Συμπερασματικά, το hardware της GPU μας επιτρέπει να κάνουμε πολύ περισσότερες πράξεις στον ίδιο χρόνο σε σχέση με την CPU, χάρη στον πολύ μεγαλύτερο αριθμό από ALUs που διαθέτει.

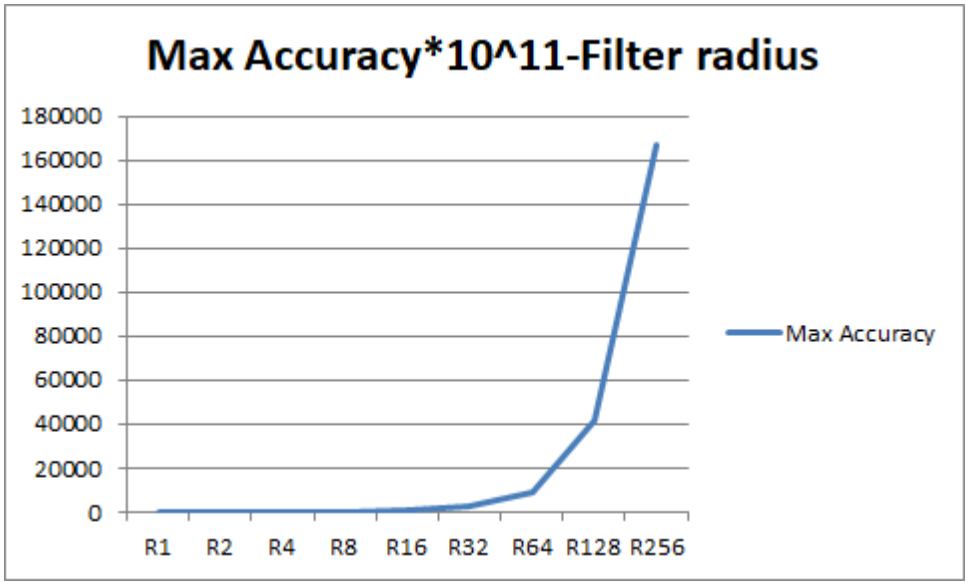
Ερώτημα 6) α)

Σε αυτό το βήμα επαναλαμβάνουμε τα ερωτήματα του βήματος 5, με τη διαφορά ότι πλέον έχουμε πράξεις που αφορούν doubles. Έχει προστεθεί το κατάλληλο typdef στον κώδικα ώστε να γίνει η αλλαγη από floats σε doubles.

Πίνακας

Ακτίνα φίλτρου	1	2	4	8	16	32	64	128	256
Μέγιστη ακρίβεια	1.455191522836685e-11	8.731149137020111e-11	6.984919309616089e-10	2.793967723846436e-09	7.450580596923828e-09	2.607703208923 34e-08	8.940696716308 594e-08	4.172325134277344e-07	1.668930053710938e-06

Διάγραμμα



Σχολιασμός αποτελεσμάτων

Όσον αφορά την ακρίβεια που μπορούμε να υποστηρίξουμε, παρατηρείται ότι στους doubles υπάρχει πολύ καλύτερη ακρίβεια, όπως είναι αναμενόμενο λόγω των επιπλέον bits που χρησιμοποιούνται για την αναπαράσταση των αριθμών. Ένα σημαντικό σχόλιο για την περίπτωση των doubles είναι ότι διαχειρίζονται καλύτερα τόσο τους αριθμούς κοντά στο μηδέν όσο και τους

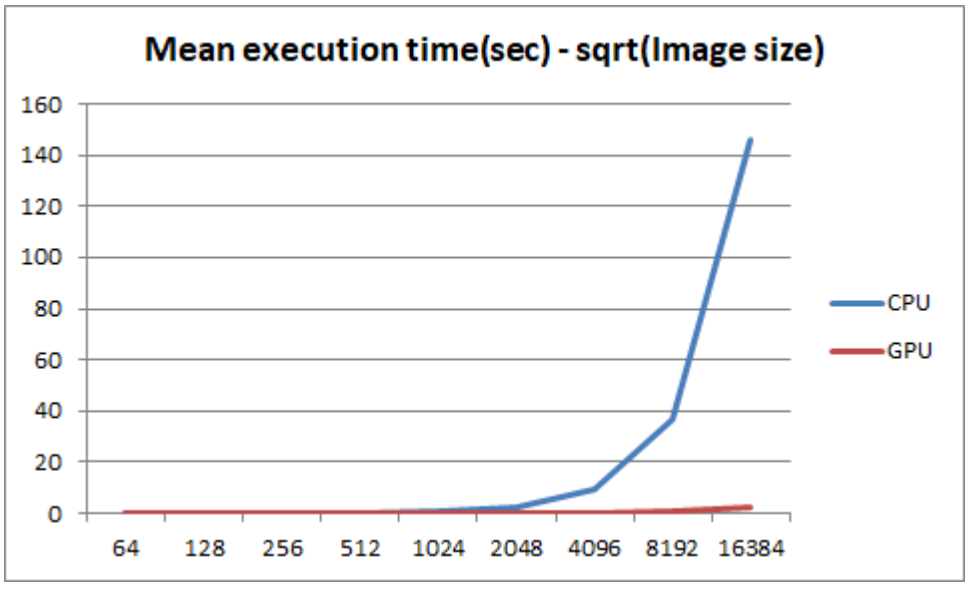
αριθμούς που απέχουν αρκετά από αυτό(δηλαδή για μεγάλες τιμές ακτίνας φίλτρου), διότι οι αναπαραστάσεις των αριθμών είναι πιο πυκνές σε όλο το εύρος τους. Επομένως, μπορούμε για το ίδιο workload να επιτύχουμε πολύ καλύτερη ακρίβεια. Στο παραπάνω διάγραμμα οι τιμές της ακρίβειας έχουν πολλαπλασιαστεί με έναν παράγοντα 10^11 για να είναι πιο εμφανής η κλιμάκωση ανάλογα με την αύξηση της ακτίνας φίλτρου. Επίσης, παρατηρούμε ότι για ακτίνα φίλτρου ίση με 256 όπου η μέγιστη ακρίβεια είναι η πιο “κακή”, εξακολουθεί να είναι πολύ καλύτερη σε σχέση με την ακρίβεια που βλέπαμε σε όλα τα πειράματα που αφορούσαν floats.

Ερώτημα 6) β)

Πίνακας

Ακτίνα φίλτρου	64	128	256	512	1024	2048	4096	8192	16384
CPU Μέσος χρόνος (sec)	0.001486	0.006243	0.026068	0.106995	0.514357	2.281968	9.085020	36.515182	145.657285
Τυπική απόκλιση	0.000009	0.000029	0.000042	0.000432	0.005577	0.027431	0.076487	0.520002	2.422043
GPU Μέσος χρόνος (sec)	0.000146	0.000253	0.000668	0.002068	0.006850	0.037577	0.147373	0.588310	2.268728
Τυπική απόκλιση	0.000003	0.000005	0.000011	0.000035	0.000086	0.000385	0.002619	0.008329	0.065110

Διάγραμμα



Σχολιασμός αποτελεσμάτων

Στο συγκεκριμένο ερώτημα είναι φανερό το tradeoff μεταξύ επίδοσης και ακρίβειας. Πιο συγκεκριμένα, οι μέσοι χρόνοι εκτέλεσης στην περίπτωση των doubles είναι σαφώς μεγαλύτεροι από εκείνους των floats, τόσο για τη CPU όσο και για τη GPU. Όμοια με το ερώτημα 5β) η GPU η οποία εξακολουθεί να μας προσφέρει καλύτερη επίδοση σε σχέση με τη CPU. Όμως, εάν συγκρίνουμε την επίδοση της GPU και στις 2 περιπτώσεις, θα παρατηρήσουμε ότι στην περίπτωση των doubles η επίδοση είναι "χειρότερη" σε σχέση με τους floats. Αυτό είναι λογικό, εάν σκεφτούμε τον περιορισμό που θέτει το hardware της GPU, το οποίο περιέχει πολύ λιγότερα double precision units σε σχέση με τα single precision units, κάτι το οποίο μειώνει τον μέγιστο δυνατό παραλληλισμό που πετυχαίνουμε για τους doubles. Επομένως, με την υλοποίηση των doubles έχουμε πολύ καλύτερη ακρίβεια που μπορούμε να υποστηρίξουμε χωρίς σφάλματα σύγκρισης διότι προσεγγίζουμε καλύτερα τα αποτελέσματα της CPU καθώς έχουμε περισσότερα bits για την αναπαράσταση των αριθμών και τυχόν στρογγυλοποιήσεις δε χάνουν τόση πληροφορία σε σχέση με τους floats, όμως υστερεί πολύ από την άποψη της επίδοσης, έχοντας τον περιορισμό του hardware λόγω του κόστους της απόκτησης περισσότερων μονάδων διπλής ακρίβειας. Ένας ακόμη σημαντικός παράγοντας που επηρεάζει αρνητικά την επίδοση της GPU είναι ότι πλέον έχουμε μεγαλύτερους πίνακες σε bytes να μεταφέρουμε στην GPU μέσω του host, αλλά και αντίστροφα για να επιστρέψουμε τα αποτελέσματα στη CPU. Αυτό συμβαίνει διότι οι floats είναι 4 bytes, ενώ οι doubles είναι 8 bytes. Επομένως, μεταφέρουμε τον διπλάσιο αριθμό από bytes μεταξύ host και GPU, κάτι που κάνει τον κώδικα των doubles ακόμα πιο αργό, εφόσον το cudaMemCpy είναι αρκετά "ακριβό" και σίγουρα μη αμελητέο στον συνολικό χρόνο εκτέλεσης της GPU. Συμπερασματικά, οι doubles προσφέρουν μία καλύτερη προσέγγιση σε σχέση με τα αποτελέσματα CPU, όμως υστερούν στο θέμα της επίδοσης σε σχέση με τους floats.

Θεωρητικό ερώτημα 7)

Α) Στο συγκεκριμένο ερώτημα πρέπει να υπολογίσουμε τις φορές που διαβάζεται κάθε στοιχείο της εικόνας εισόδου και του φίλτρου κατά την εκτέλεση των kernels. Όσον αφορά τον πίνακα εισόδου, είναι φανερό ότι θα έχει διαφορετικές συνολικές προσπελάσεις ανάλογα με τη θέση που βρίσκεται το κάθε στοιχείο καθώς και ανάλογα με την ακτίνα φίλτρου που χρησιμοποιείται. Στην παρακάτω εξήγηση τα i και j παίρνουν τιμές από 0 εως και N - 1, όπου N η διάσταση για γραμμές και στήλες του πίνακα.

Στον **kernelRow** διακρίνουμε 3 περιπτώσεις με βάση τη στήλη που βρίσκεται το κάθε στοιχείο, οι οποίες είναι οι εξής:

1) j >= filterR && j <= N – filterR – 1

Τα σημεία με αυτό το χαρακτηριστικό θα προσπελαστούν 1 φορά ως το κεντρικό pixel, ενώ στη συνέχεια θα προσπελαστούν filterR φορές από τα pixels στα αριστερά τους, και ακόμη filterR φορές από τα pixels στα δεξιά τους. Επομένως, θα έχουμε σε αυτά 2*filterR + 1 προσπελάσεις στο kernelRow.

2) j < filterR

Τα στοιχεία αυτά θα προσπελαστούν 1 φορά ως κέντρα, filterR φορές από τα δεξιά τους στοιχεία και επιπλέον j φορές από τα αριστερά τους στοιχεία για τη συνέλιξη. Επομένως έχουμε συνολικά

1 + filterR + j προσπελάσεις.

3) j > N – filterR - 1

Για τη συγκεκριμένη περίπτωση τα στοιχεία θα προσπελαστούν 1 φορά ως κέντρα, filterR φορές από τα αριστερά τους pixels και επιπλέον N – 1 – j φορές από τα δεξιά pixels. Επομένως έχουμε συνολικά filterR + N – j προσπελάσεις.

Με τον ίδιο ακριβώς τρόπο προσέγγισης μπορούμε να δείξουμε και τις συνολικές προσπελάσεις για τα στοιχεία του πίνακα εισόδου από τον **kernelColumn**. Διακρίνουμε τις εξής 3 περιοχές:

1) i >= filterR && i <= N – filterR – 1

Τα σημεία με αυτό το χαρακτηριστικό θα προσπελαστούν 1 φορά ως το κεντρικό pixel, ενώ στη συνέχεια θα προσπελαστούν filterR φορές από τα pixels της ίδιας στήλης και μικρότερης γραμμής, και ακόμη filterR φορές από τα pixels ίδιας στήλης και μεγαλύτερης γραμμής. Επομένως, θα έχουμε σε αυτά 2*filterR + 1 προσπελάσεις στο kernelColumn.

2) i < filterR

Τα στοιχεία αυτά θα προσπελαστούν 1 φορά ως κέντρα, filterR φορές από τα στοιχεία ίδιας στήλης και μεγαλύτερης γραμμής και επιπλέον i φορές από τα στοιχεία ίδιας στήλης και μικρότερης γραμμής για τη συνέλιξη. Επομένως έχουμε συνολικά 1 + filterR + i προσπελάσεις.

3) i > N – filterR - 1

Για τη συγκεκριμένη περίπτωση τα στοιχεία θα προσπελαστούν 1 φορά ως κέντρα, filterR φορές από τα pixels ίδιας στήλης και μικρότερης γραμμής και επιπλέον N – 1 – i φορές από τα pixels ίδιας στήλης και μεγαλύτερης γραμμής. Επομένως έχουμε συνολικά filterR + N – i προσπελάσεις.

Επομένως, για να βρούμε το συνολικό αριθμό προσπελάσεων για οποιαδήποτε θέση του πίνακα εισόδου αρκεί να προσθέσουμε τις προσπελάσεις από το kernelRow και το kernelColumn για το δεδομένο σημείο (i, j).

Με βάση τον παραπάνω τρόπο προσέγγισης του προβλήματος, προκύπτουν όλες οι πιθανές περιοχές στον πίνακα εισόδου και οι συνολικές προσπελάσεις από kernelRow και kernelColumn που συνοψίζονται στον παρακάτω πίνακα.

Περιοχή i	Περιοχή j	Συνολικές προσπελάσεις
i >= filterR && i <= N –filterR –1	j >= filterR && j <= N–filterR–1	2 + 4*filterR
i >= filterR && i <= N –filterR –1	j < filterR	2 + 3* filterR + j
i >= filterR && i <= N –filterR –1	j > N –filterR – 1	1 + 3 * filterR + N – j
i < filterR	j >= filterR && j <= N –filterR – 1	2+ 3*filterR + i
i > N –filterR – 1	j >= filterR && j <= N –filterR-1	1+ 3*filterR + N – i
i < filterR	j < filterR	2 + 2*filterR + i + j
i< filterR	j > N-1 – filterR	1+ 2*filterR + i + N – j
i> N- filterR - 1	j < filterR	1+ 2*filterR + N-i + j
i> N- filterR - 1	j >N - filterR – 1	2 * filterR + 2*N – i – j

Αναφορικά με τις συνολικές προσπελάσεις σε κάθε θέση του φίλτρου, εφόσον χρησιμοποιούμε το ίδιο φίλτρο τόσο στον kernelRow όσο και στον kernelColumn και σε συνδυασμό με τη συμμετρία

του πίνακα εισόδου, αρκεί να βρούμε τις προσπελάσεις για έναν από τους δύο και να τις πολλαπλασιάσουμε επί δύο. Επίσης, λόγω της φύσης του προβλήματος όπου ασχολούμαστε με φίλτρο μήκους 2*filterR + 1 και με τετραγωνικούς πίνακες εισόδου, συνεπάγεται ότι οι θέσεις filterR + i και filterR – i θα έχουν ίσο αριθμό συνολικών προσπελάσεων (1<= i <= filterR), ενώ η θέση στο μέσον του φίλτρου, δηλαδή η θέση filterR, θα προσπελαστεί μία φορά για κάθε στοιχείο του πίνακα εισόδου κατά την εκτέλεση ενός kernel. Επιπλέον, τα στοιχεία filterR + i θα προσπελαστούν N-i φορές για κάθε γραμμή του πίνακα εισόδου διότι για τα i τελευταία στοιχεία θα βγαίνουν εκτός ορίων πίνακα. Πολλαπλασιάζοντας αυτό τον αριθμό με το πλήθος των γραμμών προκύπτει ότι κατά την εκτέλεση **ενός** kernel θα έχουμε προσπελάσεις ίσες με N*(N - i), 1<=i<=filterR.

Συνεπώς, το στοιχείο στη θέση filterR θα προσπελαστεί **2*N*N φορές** συνολικά. Επίσης, τα στοιχεία στις θέσεις filterR + i και filterR – i θα προσπελαστούν το καθένα **2*N*(N-i) φορές** αθροιστικά για τους 2 kernels, με 1 <= i <= filterR.

7) β) Στο ερώτημα αυτό μετράμε ως προσπελάσεις μνήμης μόνο τις αναγνώσεις από την Global Memory της GPU. Επίσης, αγνοούμε την αποθήκευση του αποτελέσματος και θεωρούμε τους πολλαπλασιασμούς και τις προσθέσεις ως ξεχωριστές πράξεις.

Για να βρούμε το λόγο προσπελάσεων στην global memory προς πράξεις κινητής υποδιαστολής, αρκεί να σκεφτούμε ότι ο μόνος τρόπος για να έχουμε προσπελάσεις στη global memory και πράξεις κινητής υποδιαστολής είναι να ισχύει η συνθήκη if σε κάθε kernel. Όταν το σώμα της συνθήκης if εκτελείται, τότε έχουμε 2 προσπελάσεις στη μνήμη και 2 πράξεις κινητής υποδιαστολής. Πιο συγκεκριμένα, η πρώτη προσπέλαση στη μνήμη αφορά κάποια θέση του πίνακα εισόδου, ενώ η δεύτερη προσπέλαση αφορά κάποια θέση του φίλτρου. Επιπλέον, έχουμε έναν πολλαπλασιασμό μεταξύ ενός στοιχείου του πίνακα εισόδου με την κατάλληλη τιμή του φίλτρου και έπειτα το αποτέλεσμα του πολλαπλασιασμού προστίθεται στην παλιά τιμή του sum. Έστω ότι η συνθήκη if εκτελείται x φορές συνολικά για τους 2 kernels, όπου x ένας θετικός ακέραιος αριθμός. Τότε, ο λόγος προσπελάσεων στην global memory προς πράξεις κινητής υποδιαστολής θα ισούται με (2*x)/(2*x). Επομένως, ανεξάρτητα από το ποιος είναι ο αριθμός x, ο λόγος ισούται με 1/1 διότι έχουμε ίσο αριθμό προσπελάσεων στην global memory της GPU και πράξεων κινητής υποδιαστολής.

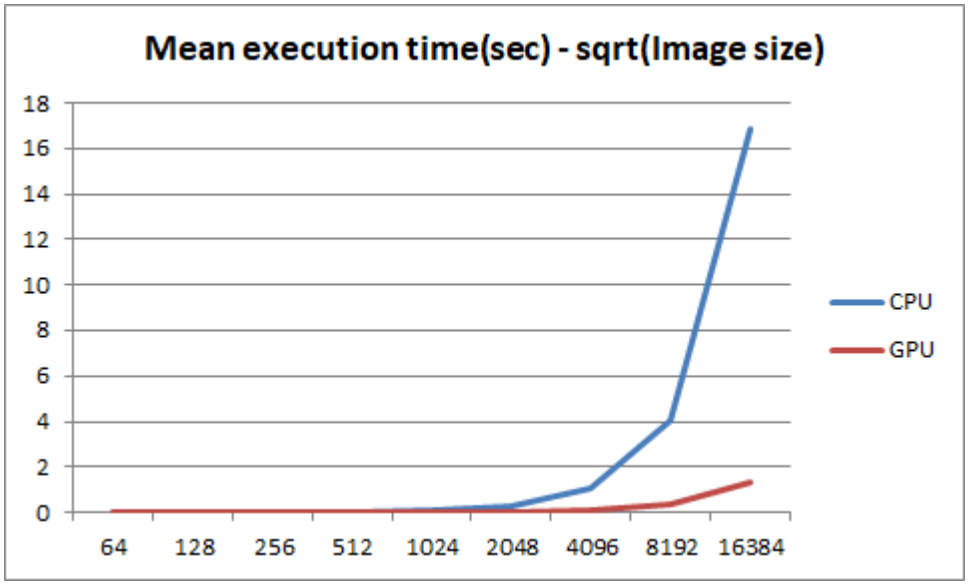
8) Στο ακόλουθο ερώτημα, προσπαθούμε να εξαλείψουμε το φαινόμενο του divergence για τα threads εντός του ίδιου warp, δηλαδή να εφαρμόσουμε padding στον αρχικό πίνακα εισόδου ώστε να μην υπάρχει πλέον η συνθήκη if που εξετάζει εάν το φίλτρο “βγαίνει” εκτός ορίων πίνακα. Αυτό θα έχει ως αποτέλεσμα τα threads του ίδιου warp να μην μένουν αδρανή εφόσον πλέον δεν υπάρχει περίπτωση να εκτελέσουν διαφορετικό κώδικα μεταξύ τους. Με αυτό τον τρόπο κερδίζουμε σε επίδοση, καθώς στο divergence έχουμε αδρανή threads που θα τρέξουν μόλις τελειώσουν τα threads που εκτελούν διαφορετικό κώδικα από αυτά. Άρα στον ίδιο χρόνο θα εκτελούμε μεγαλύτερο αριθμό από νήματα ταυτόχρονα, επεκτείνοντας το βαθμό της παραλληλοποίησης όσον αφορά τη χρήση των πόρων της GPU. Έτσι, διαχειριζόμαστε καλύτερα τους πόρους του συστήματος, γεγονός που φαίνεται και από τη σύγκριση του μέσου χρόνου εκτέλεσης μεταξύ του κώδικα για το βήμα 8 με τον κώδικα του βήματος 4. Πλέον ο πίνακας εισόδου είναι διαστάσεων (N+2*FILTER_RADIUS)*(N+2*FILTER_RADIUS), γεγονός που “βυθίζει” τον αρχικό πίνακα σε έναν άλλο μεγαλύτερο, ώστε το φίλτρο να μη βγαίνει εκτός ορίων. Για το λόγο αυτό, στη GPU το κάθε thread αρκεί να υπολογίσει την ίδια θέση που είχε αναλάβει και στην προηγούμενη

έκδοση κώδικα, απλώς μετατοπισμένη κατά FILTER_RADIUS τόσο στον άξονα x όσο και στον y, ώστε να βρεθούν στην κατάλληλη θέση του νέου πίνακα. Η αλλαγή αυτή αφορά καθαρά τον κώδικα της GPU, όμως χρησιμοποιήσαμε και τον ίδιο πίνακα για τις συναρτήσεις που υλοποιούν τη συνέλιξη στη CPU, καθώς μπορεί να εξαλειφθεί το if statement, και κατά συνέπεια να κερδίσουμε και σε επίδοση για τη CPU. Τέλος, στον κώδικα του padding αρχικοποιήσαμε σε μηδέν τα στοιχεία που είναι εκτός ορίων του αρχικού μας πίνακα, ώστε να μη συνεισφέρουν στις πράξεις της συνέλιξης.

Πίνακας για padding

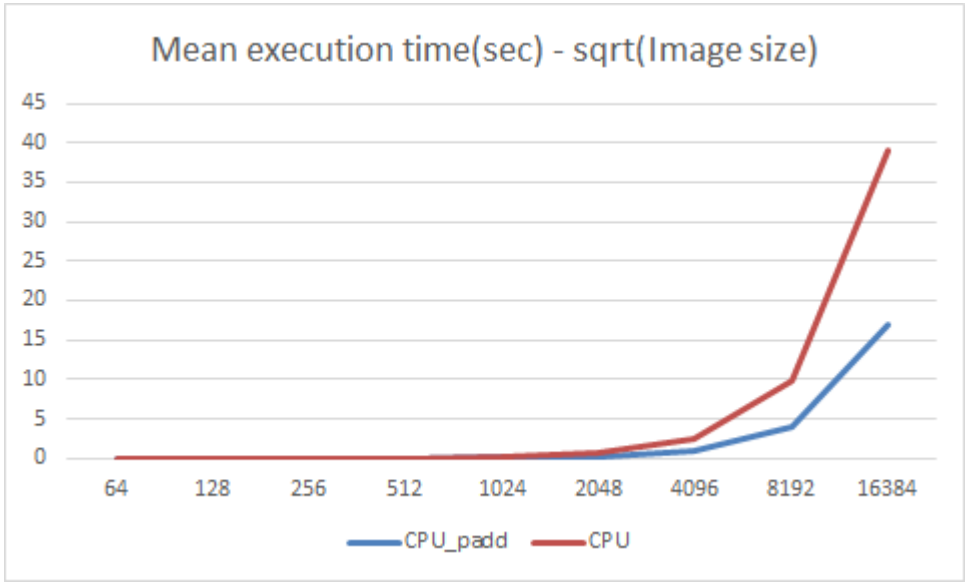
Ακτίνα φίλτρου	64	128	256	512	1024	2048	4096	8192	16384
CPU Μέσος χρόνος (sec)	0.000201	0.000845	0.003563	0.014125	0.056178	0.231410	1.029130	4.033123	16.827325
Τυπική απόκλιση	0.000000	0.000005	0.000015	0.000051	0.000231	0.000195	0.000857	0.008757	0.056423
GPU Μέσος χρόνος (sec)	0.000132	0.000232	0.000605	0.002014	0.006625	0.024955	0.092542	0.353942	1.323058
Τυπική απόκλιση	0.000004	0.000004	0.000008	0.000021	0.000077	0.000237	0.001692	0.030031	0.052203

Διάγραμμα για padding



Διάγραμμα για σύγκριση χρόνου εκτέλεσης κώδικα βήματος 4 σε σχέση με padding για την GPU:

Διάγραμμα για σύγκριση χρόνου εκτέλεσης κώδικα βήματος 4 σε σχέση με padding για την CPU:



Σχολιασμός αποτελεσμάτων

Όπως φαίνεται και από τις παραπάνω μετρήσεις, η αλλαγή αυτή βοηθάει τόσο την απόδοση της GPU όσο και της CPU. Αναφορικά με τη GPU, για μικρά workloads παρατηρούμε παρόμοιο χρόνο εκτέλεσης. Επειδή το πρόβλημα είναι παραμετρικό, δηλαδή δουλεύει για διαφορετικές τιμές ακτίνας φίλτρου και μεγέθους πίνακα, το φαινόμενο του divergence μπορεί κάποιες φορές να μην

εμφανίζεται και κατά συνέπεια σε αυτές τις περιπτώσεις δεν παρατηρείται βελτίωση στο padding. Όμως, το padding έχει καλύτερη επίδοση από την προηγούμενη έκδοση κώδικα, όπως φαίνεται και από το διάγραμμα που συγκρίνει την επίδοση των 2 εκδόσεων στην GPU, κυρίως όταν το workload μεγαλώνει σημαντικά, όπου σε αυτή την περίπτωση φαίνεται η επιρροή του divergence στην επίδοση του προγράμματος. Δηλαδή, όσο αυξάνεται μέγεθος του πίνακα, τόσο περισσότερο αρνητική επίδραση επιφέρει το divergence, εφόσον αυτό συμβαίνει σε περισσότερα warps που αφορούν threads που βρίσκονται σε “ακριανές” θέσεις λόγω του αυξημένου μεγέθους πίνακα. Αυτό μας δείχνει ότι κάποιες “μικρές” αλλαγές στη δομή δεδομένων του προβλήματος μας οδηγούν σε καλύτερη εκμετάλλευση του hardware και κατά συνέπεια καλύτερη επίδοση. Όσον αφορά την βελτίωση του χρόνου εκτέλεσης της CPU, παρατηρούμε ότι αυτό οφείλεται στο ότι πλέον δεν έχουμε branch prediction στο hardware λόγω της εξάλειψης της συνθήκης if, επομένως δεν υπάρχουν branch misses λόγω λανθασμένης πρόβλεψης που μπορεί να υπήρχαν στην προηγούμενη περίπτωση τα οποία είναι αρκετά “ακριβά” από άποψη χαμένων κύκλων μηχανής.