

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΗΥ421 Συστήματα Υπολογισμού Υψηλών Επιδόσεων

LAB4

Ματζώρος Χρήστος Κωνσταντίνος

Οι μετρήσεις για την εργασία πραγματοποιήθηκαν στο μηχάνημα asrtemis το οποίο έχει τα παρακάτω χαρακτηριστικά(device query):

Device 1: "Tesla K80"

CUDA Driver Version / Runtime Version	10,0 / 10,0
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) >

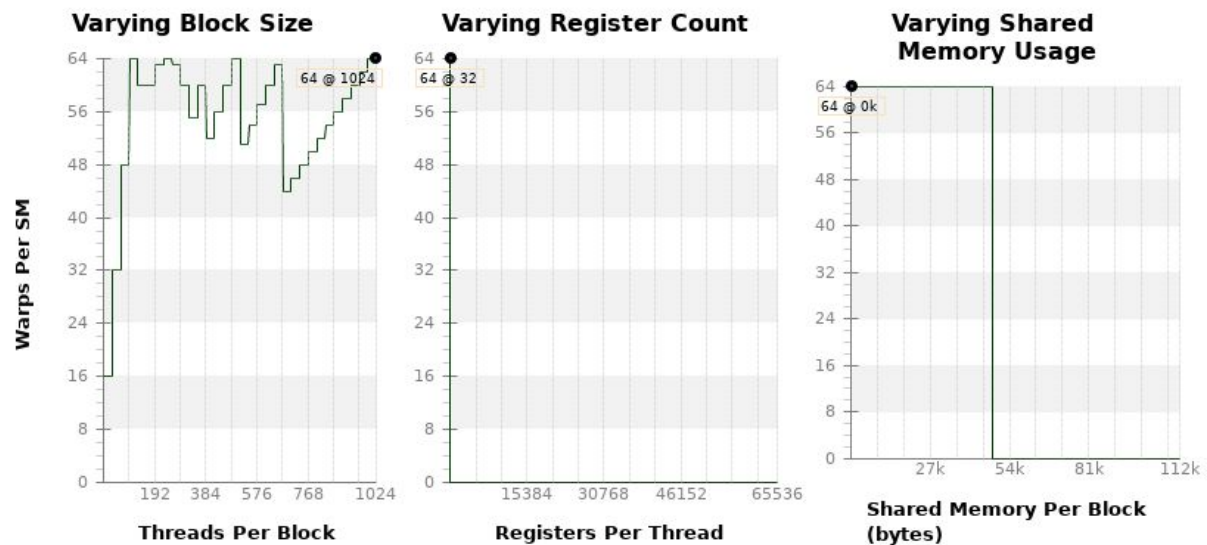
> Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU1) : Yes
 > Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU0) : Yes
 deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10,0, CUDA Runtime Version = 10,0,
 NumDevs = 2

Σε όλα τα αρχεία το μέγεθος της εικόνας, το μέγεθος του φίτρου και το μέγεθος του tile δίνονται παραμετρικά στην αρχή του προγράμματος μέσω του #define.

1) Για τα διαφορετικά στάδια των βελτιστοποιήσεων χρησιμοποιούμε εικόνα μεγέθους 8192x8192 και ακτίνα φίλτρου 32. Στο profiling που έγινε στον αρχικό κώδικα για τα δύο kernel invocations καθώς και για τις μεταφορές μνήμης από τον Host στο Device αλλά και από το Device στο Host παίρνουμε τα εξής αποτελέσματα στους χρόνους εκτέλεσης:

convolutionGPURow()	200.94688ms
convolutionGPUColumn()	193.60343ms
MemCpy(Host to Device)	92.48453ms
MemCpy(Device to Host)	213.98633ms

Επιπλέον παρατηρούμε ότι και στους 2 kernels δεν έχουμε κάποιο πρόβλημα με το occupancy, δηλαδή κάνουμε καλή χρήση των resources της GPU όπως είναι οι registers και η shared memory και δεν περιορίζουμε τον αριθμό των warps σε κάθε Streaming Multiprocessor. Αυτό φαίνεται στο παρακάτω διάγραμμα στο nvnv:



Η πρώτη αλλαγή που χρειάστηκε να γίνει στον κώδικά μας ήταν να τοποθετήσουμε την μνήμη του φίλτρου σε κάποιο είδος μνήμης που να είναι cached. Επιλέχθηκε να τοποθετηθεί στην constant memory γνωρίζοντας ότι ο κώδικας του kernel χρησιμοποιεί την μνήμη του φίλτρου μόνο για αναγνώσεις και μάλιστα από τα ίδια σημεία. Επιπλέον η shared memory είναι πολύτιμη από την άποψη πως καλό είναι να γίνονται και εγγραφές σε αυτή, οπότε δεν χρησιμοποιείται σε αυτή την περίπτωση. Η δήλωση του φίλτρου (d_Filter) στην constant memory μας δίνει τους παρακάτω χρόνους εκτέλεσης του προγράμματος κατά το profiling:

convolutionGPURow()	169.54805ms
convolutionGPUColumn()	159.90432ms
MemCpy(Host to Device)	188.97343ms
MemCpy(Device to Host)	212.73131ms

Παρατηρούμε μία αύξηση στον χρόνο μεταφοράς της μνήμης από τον Host στο Device παρόλα αυτά μειώνεται ο χρόνος εκτέλεσης των δύο kernels στην GPU συνολικά κατά 16.49% και μειώνεται ο συνολικός χρόνος εκτέλεσης, οπότε αυτή η βελτίωση θεωρείται αποδεκτή. Επιπλέον το occupancy περέμεινε το ίδιο καλό με προηγούμενως.

Σαν επόμενα βήματα ήταν να αλλάξουμε την μεταβλητή που δείχνει το μέγεθος της ακτίνας του φίλτρου(filterR) από την global memory στην static memory κάτι που δεν μας ωφέλησε, οπότε απορρίφθηκε. Στην συνέχεια βάλαμε να υπολογίζεται μία μόνο φορά και να αποθηκεύεται σε ένα register ένα common expression: `"idy*(padded_image_width)"` που υπολογίζεται άσκοπα πολλές φορές μέσα στο loop του ενός kernel. Επίσης αυτή η αλλαγή δεν μας βοήθησε κάπου. Έχοντας παρατηρήσει ότι γίνεται αρκετή χρήση των registers κάποιες μεταβλητές που ήταν δηλωμένες ως registers δηλώθηκαν σε κατάλληλο πίνακα στη shared memory για κάθε block όπως για παράδειγμα τα indexes idx, idy των νημάτων μέσα στο grid, αλλά ό,τι αλλαγή επιδιώχθηκε δεν βοήθησε στην μείωση του χρόνου εκτέλεσης. Έχοντας βεβαιωθεί ότι δεν μπορούν να γίνουν άλλες αλλαγές που να αφορούν τους registers και τις μνήμες προχωρήσαμε σε κάτι που αφορά την υλοποίηση της εφαρμογής. Το επόμενο βήμα βελτιστοποίησης ήταν να μειώσουμε το μέγεθος του πίνακα εισόδου για κάθε kernel στο ελάχιστο, αφαιρώντας ένα κομμάτι από το padding στον πίνακα εισόδου για κάθε kernel (σχήμα a). Αυτό σημαίνει ότι όταν κάνω convolution κατά γραμμές δεν χρειάζεται να περνάω τον πίνακα εισόδου με padding σε όλες τις πλευρές αλλά μόνο αριστερά και δεξιά της εικόνας(σχήμα b) και όταν κάνω convolution κατά στήλες δεν χρειάζεται ο πίνακας που περνάω ως είσοδο στον kernel να έχει padding περιμετρικά αλλά μόνο στο πάνω και κάτω μέρος της εικόνας(σχήμα c). Επιπλέον ο πίνακας εξόδου περιείχε άσκοπα και το padding κάτι που δεν μας χρησίμευε, οπότε η τελική εικόνα απο τον δεύτερο kernel δεν περιέχει πλέον padding(σχήμα d). Όλα αυτά τα κομμάτια padding που αφαιρέθηκαν είχαν ως αποτέλεσμα να γίνεται πλέον λιγότερη δέσμευση μνήμης στην global memory καθώς και μεταφορές λιγότερης μνήμης από και προς την GPU.



(a)



(b)

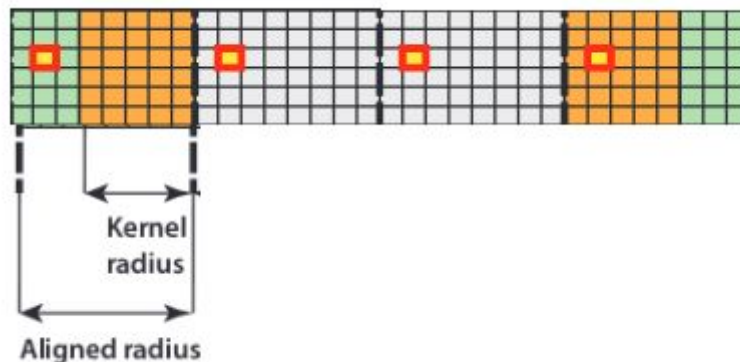






(c)



(d)

Το επόμενο βήμα ήταν να κάνουμε χρήση της shared memory σε κάθε block. Έτσι υλοποιήθηκε ένα εσωτερικό tiling στο επίπεδο του block όπου κάθε νήμα μπορεί να υπολογίζει περισσότερα από ένα στοιχεία της εικόνας. Αρχικά όλα τα νήματα του block συνεργάζονται και φέρνουν στην shared memory κάποια στοιχεία της εικόνας εισόδου όπως φαίνεται παρακάτω. Για να το πετύχουμε αυτό χωρίς να κάνουμε ελέγχους μέσα στον κώδικα του kernel προσθέτουμε ένα επιπλέον κομμάτι στο padding ώστε να γίνει aligned με τον αριθμό των νημάτων.



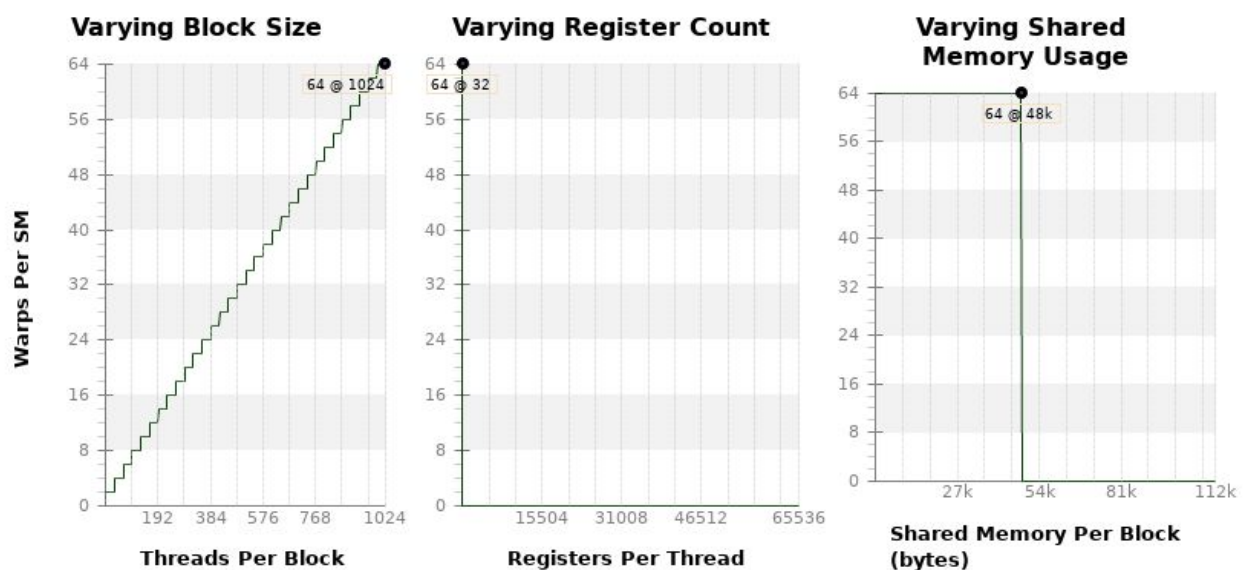
-  Αυτά τα στοιχεία γράφονται στην shared memory από το ίδιο νήμα.
-  Στοιχεία που περιέχουν χρήσιμη πληροφορία για τον υπολογισμό.
-  Στοιχεία του αρχικού padding(όσο η ακτίνα).
-  Στοιχεία που γίνονται aligned για να αποφύγουμε ελέγχους στον kernel(divergence).

Λόγω αυτής της τεχνικής περιορίζεται η shared memory σε κάθε block και πειραματικά βλέπουμε ότι το φίλτρο μπορεί να φτάσει μέχρι μέχρι radius = 63. Αν βάλουμε παραπάνω δεν μας είναι αρκετή η shared memory. Αφήνουμε το εσωτερικό tiling να υπολογίζει 2 σημεία της εικόνας και όχι παραπάνω καθώς όπως φάνηκε κατα το profiling, αν προσπαθήσουμε για παραπάνω στοιχεία είτε χαλάμε το occupancy είτε δεν αρκεί η shared memory. Με το `__syncthreads` στον κώδικα του kernel βεβαιωνόμαστε πως έχουν ολοκληρωθεί οι εγγραφές στην shared memory πριν ξεκινήσουν οι αναγνώσεις από αυτή. Στην συνέχεια οι αναγνώσεις γίνονται μεταξύ θέσεων της μνήμης που υπάρχουν στην shared memory(Input) και στην constant memory(filter). Εξασφαλίζοντας ότι αυτές οι

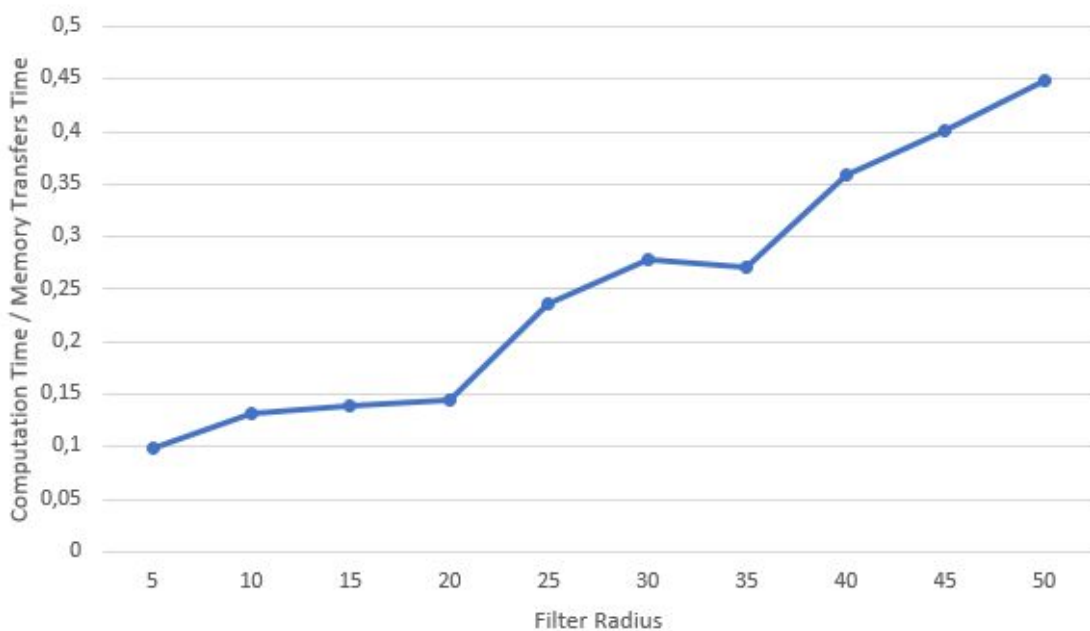
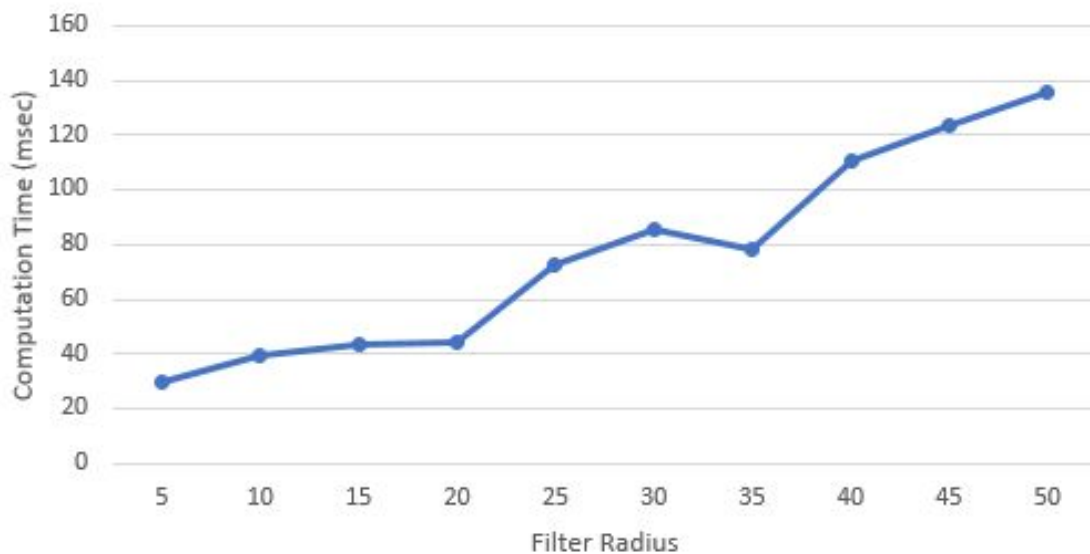
αναγνώσεις θα είναι cached η εκτέλεση του kernel γίνεται με πολύ μεγαλύτερη ταχύτητα. Παρακάτω παρουσιάζονται κάποια στοιχεία της τελικής έκδοσης του κώδικα σε σχέση με τον αρχικό. Φαίνεται ότι έχουμε σημαντική διαφορά στον χρόνο εκτέλεσης των kernels, ενώ οι χρόνοι μεταφοράς της μνήμης δεν μεταβάλλονται σημαντικά. Ο κώδικας βρίσκεται στο αρχείο `pad_improved.cu`.

	Αρχική Έκδοση Κώδικα	Τελική Έκδοση Κώδικα
<code>convolutionGPURow()</code>	200.94688ms	52.56707ms
<code>convolutionGPUColumn()</code>	193.60343ms	38.66099ms
<code>MemCpy(Host to Device)</code>	92.48453ms	93.89146ms
<code>MemCpy(Device to Host)</code>	213.98633ms	207.85248ms

Επιπλέον παρακάτω φαίνεται πώς επηρεάζεται το occupancy στην τελική έκδοση του κώδικά μας:

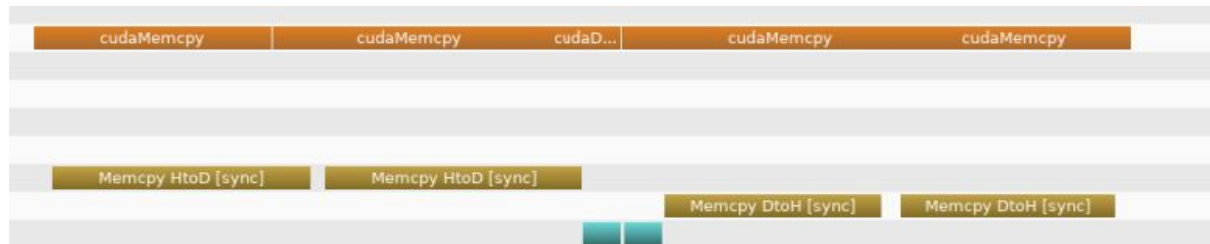


2) Στο πρώτο διάγραμμα παρατηρούμε ότι ο χρόνος εκτέλεσης των kernels αυξάνεται καθώς αυξάνεται το μέγεθος του φίλτρου, κάτι που δικαιολογείται αφού γίνονται περισσότερες επαναλήψεις στον κώδικα του kernel για τον υπολογισμό του αποτελέσματος ενός σημείου της εικόνας. Στο δεύτερο διάγραμμα παρατηρούμε ότι λόγος του χρόνου εκτέλεσης kernels προς τον χρόνο μεταφορών μνήμης αυξάνεται σε σχέση με το φίλτρο. Αυτό μας δείχνει πως κατά την αύξηση του φίλτρου επηρεάζεται περισσότερο ο χρόνος εκτέλεσης παρά ο χρόνος που γίνονται οι μεταφορές.

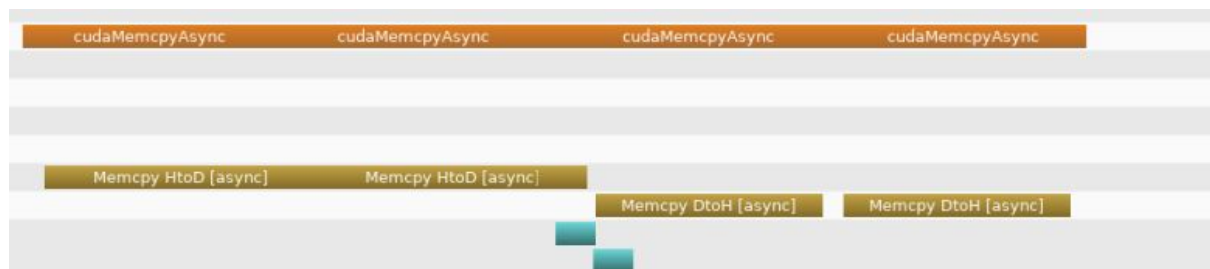


3) Για να μπορέσουμε να υποστηρίξουμε μεγαλύτερες εικόνες πάνω στη GPU χρειάστηκε η εκτέλεση των kernels να είναι blocked, με τέτοιο τρόπο ώστε όλα τα δεδομένα που απαιτούνται να χωράνε στη GPU. Αυτό μπορούμε να το πετύχουμε κάνοντας πολλαπλά kernel invocations χωρίζοντας ολόκληρη την εικόνα σε μικρότερα κομμάτια(tiles). Το μέγεθος του tile πρέπει να είναι δύναμη του 2 και αντιπροσωπεύει τον αριθμό των νημάτων στο grid. Το μέγεθος του tile καθώς και το μέγεθος της εικόνας και της ακτίνας του φίλτρου ορίζονται παραμετρικά στην αρχή του προγράμματος με `#define`. Επιπλέον με το `#define ENABLE_PROFILER_STATS` καθορίζουμε για το αν θέλουμε πλέον να εκτελεστεί ο κώδικας και στην CPU και να γίνει η σύγκριση του αποτελέσματος της με αυτό της GPU ή να γίνει μόνο η εκτέλεση στην GPU. Αν θέλουμε να εκτελεστεί ο κώδικας μόνο στην GPU αρκεί να

βάλουμε αυτό το `#define` σε σχόλια. Ο κώδικας βρίσκεται στο αρχείο `tiled.cu`. Στο παρακάτω στιγμιότυπο της εκτέλεσης του προγράμματος φαίνεται η μεταφορά δεδομένων στον πρώτο `kernel(convolutionGPURow())` από τον `host`, η εκτέλεση του κώδικα του `kernel` και η μεταφορά του αποτελέσματος πίσω στο `host` για 2 tiles(2 `kernel` invocations):



4) Παρατηρούμε ότι στον κώδικα του tiling δεν υπάρχει καμία επικάλυψη των μεταφορών μνήμης και του υπολογισμού στους `kernels`. Το επόμενο βήμα είναι να κάνουμε χρήση των `streams` ώστε να πετύχουμε επικάλυψη υπολογισμών και μεταφορών μνήμης (αρχείο `tiled_streams.cu`). Όπως φαίνεται παρακάτω πετυχαίνουμε επικάλυψη σε σχέση με το προηγούμενο πρόβλημα:



5) Το μέγιστο μέγεθος εικόνας που μπορεί να υποστηριχθεί είναι 32768×32768 . Ο πόρος που μας περιορίζει πλέον είναι η δέσμευση μνήμης για τους πίνακες στην RAM. Αν βάλουμε σαν μέγεθος εισόδου την επόμενη μεγαλύτερη δύναμη του 2 δηλαδή 65536×65536 έχουμε `buffer overflow` κατά την εκτέλεση σε κάποιες από τις `malloc()` που απαιτούνται.