Συστήματα Υπολογισμού Υψηλών Επιδόσεων (ECE 415)
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας

4η Εργαστηριακή Άσκηση

Στοιχεία φοιτητών:

Μπαλτάς Νικόλαος, 2757
Ξωχέλλη Ελένη, 2761

0.Παράθεση Device Query

```
CUDA Device Query (Runtime API) version (CUDART static linking)
  Detected 1 CUDA Capable device(s)
  Device 0: "NVIDIA GeForce GTX 1650 SUPER"
  CUDA Driver Version / Runtime Version          11.5 / 11.5
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 3912 MBytes (4101505024 bytes)
  (020) Multiprocessors, (064) CUDA Cores/MP:    1280 CUDA Cores
  GPU Max Clock rate:                            1755 MHz (1.75 GHz)
  Memory Clock rate:                             6001 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),
                                                 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  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 3 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 38 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.5, CUDA Runtime Version = 11.5, NumDevs = 1
Result = PASS
```

1.Βελτιστοποίησεις

Μετά την προηγούμενη εργασία κώδικας που αξιοποιήθηκε χρησιμοποιεί τύπο double και έχει padding με στόχο τον περιορισμό του φαινομένου του divergence.

Στα πλαίσια των βελτιστοποιήσεων κρίναμε ότι για το φίλτρο που χρησιμοποιείται συνεχώς για υπολογισμούς και είναι read only θα ήταν ωφέλιμο να εκμεταλλευτούμε την constant memory.

Έτσι ο πίνακας `__constant__ double filterConst[]` ορίζεται στην αρχή του κώδικα και προκύπτει η παραδοχή πως από εδώ και πέρα το μέγιστο μέγεθος φίλτρου θα περιορίζεται από τον συγκεκριμένο ορισμό. (Στην έκδοση που παραδίδεται χρησιμοποιείται φίλτρο με radius μέχρι 32)

Επιπλέον οι μεταβλητές που προσπελαύνουμε περισσότερο ανάλογα με την κάθε περίπτωση ορίζονται ως `register`.

Οι μικρές αυτές αλλαγές παραμένουν στην συνέχεια τον κώδικα γιατί μέσω profiling παρατηρούμε βελτίωση της επίδοσης. Ο συνολικός χρόνος εκτέλεσης μειώνεται και οι δραστηριότητες της GPU αποτελούν μεγαλύτερο μέρος του συνολικού χρόνου.

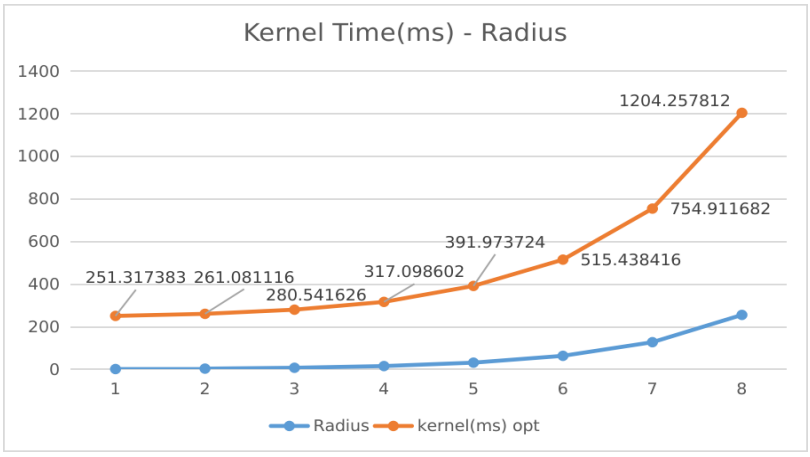
2.Παρατηρήσεις

Οι μετρήσεις χρόνου για τον χρόνο εκτέλεσης στην GPU έγιναν με την δημιουργία δύο event, το `cudaEvent_t start` που δηλώνει την αρχή μέτρησης του χρόνου και το `cudaEvent_t stop` που δηλώνει το τέλος της μέτρησης του χρόνου. Παρακάτω φαίνεται ο τρόπος με τον οποίο υπολογίζεται ο χρόνος μεταφοράς δεδομένων από το host στο device, ο χρόνος εκτέλεσης των δύο kernel και τελικά οι μεταφορά του αποτελέσματος από το device στο host.

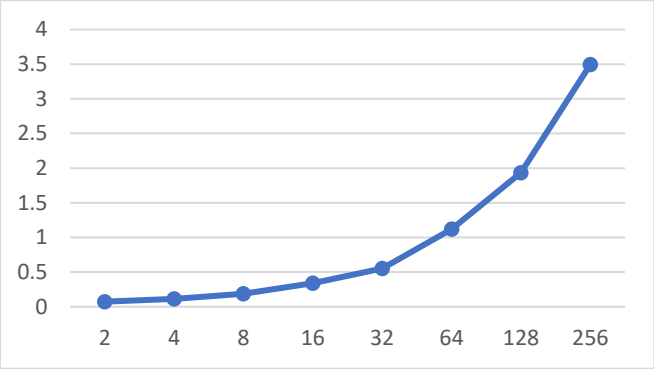
```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
//Μεταφορές μνήμης από host στο device
RowGPU
ColGPU
//Μεταφορές μνήμης από device στο host
cudaEventRecord(stop);
cudaEventSynchronize(stop);
```

Στο παρακάτω διάγραμμα παρατηρούμε ότι όσο μεγαλώνει το μέγεθος του φίλτρου αυξάνεται ο χρόνος εκτέλεσης του kernel. Αυτό συμβαίνει διότι μέσα στον κώδικα των δύο

kernel κάθε thread θα εκτελεί περισσότερους υπολογισμούς λόγω του for loop που κινείται σε συνάρτηση με το μέγεθος του φίλτρου.



Στο επόμενο διάγραμμα βλέπουμε την σχέση του χρόνου εκτέλεσης kernels με τον χρόνο για μεταφορές μνήμης. Όσο αυξάνεται το μέγεθος του φίλτρου αυξάνεται αναλογικά και ο χρόνος εκτέλεσης συνολικά. .



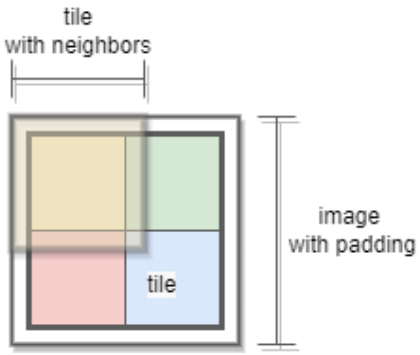
Στο διάγραμμα αριστερά είναι εμφανές ότι με σταθερό το μέγεθος της εικόνας και με την αύξηση του φίλτρου ο χρόνος που χρειάζεται για την μεταφορά δεδομένων παραμένει ίδιος. Αυξάνεται όμως το kernel time γιατί απαιτούνται περισσότεροι υπολογισμοί για το κάθε pixel. Έτσι αυξάνεται συνεχώς ο λόγος:

$$\alpha = \frac{\chi\rho\acute{o}\nu o\varsigma\ \epsilon\kappa\tau\acute{\epsilon}\lambda\epsilon\sigma\eta\varsigma}{\chi\rho\acute{o}\nu o\varsigma\ \mu\epsilon\tau\alpha\varphi o\rho\acute{\omega}\nu}$$

Τελικά η αύξηση μεγέθους του φίλτρου συνεπάγεται περισσότερο χρόνο για υπολογισμούς και επομένως κάθε load δεδομένων αξιοποιείται περισσότερο από τον κώδικα.

3.Μεγαλύτερο input

Μέχρι τώρα το πρόγραμμα μπορούσε να υποστηρίξει εικόνες διάστασης έως και 8192 pixel, αφού σε κάθε kernel launch ο αριθμός των threads είναι περιορισμένος και φυσικά η αναλογία με τα pixel είναι 1 προς 1.



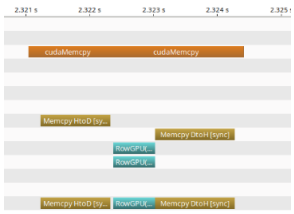
Για να μην περιοριζόμαστε από το grid και να διευκολυνθεί η εκτέλεση την σπάμε σε βήματα με διαδοχικά kernel launches τόσα όσα απαιτεί το input του χρήστη.

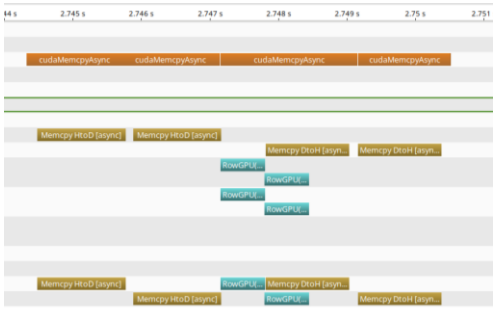
Βρίσκουμε το απαιτούμενο runTimes ανάλογα με το μέγεθος του block/tile που ορίζεται στα define. Για την ορθότητα της εφαρμογής στο κάθε tile δίνεται ένα κομμάτι επεξεργασία μαζί και με τα γειτονικά pixel αυτού είτε πρόκειται για μηδενικό padding είτε για γειτονικά εσωτερικά στοιχεία.

Σημειώνουμε ότι μετά το τέλος της πρώτης φάσης RowGPU ο πίνακας ανασυντίθεται και διαιρείται ξανά για ανανέωση των γειτονικών τιμών πριν την δεύτερη φάση ColGPU.

4.Χρήση streams

Το «σπάσιμο» της εκτέλεσης σε βήματα με διαδοχικά kernel launches μπορεί να φέρει καλύτερη επίδοση με προσπάθεια για παράλληλη εκτέλεση.
Δημιουργήσαμε 2 streams, δεσμεύσαμε τον αναγκαίο χώρο για το κάθε ένα και σε κάθε επανάληψη γίνεται ασύγχρονη μεταφορά input, τρέξιμο των streams και ασύγχρονη μεταφορά του output με `cudaMemcpyAsync()`.





Στα δύο στιγμιότυπα του profiler φαίνεται πώς η υλοποίηση με χρήση 2 streams πετυχαίνουμε παράλληλη εκτέλεση και επομένως καλύτερη επίδοση σε σχέση με την προηγούμενη μέθοδο εκτέλεσης όπου δεν υπήρχε καμία επικάλυψη. Βέβαια, μεγάλο μέρος του χρόνου εκτέλεσης καταλαμβάνεται από τις μεταφορές δεδομένων για τα δύο streams.

Μέσω profiling βλέπουμε επίσης ότι η χρήση των streams φέρνει συνολικά πιο καλά αποτελέσματα παράλληλα με μείωση του χρόνου εκτέλεσης. Με nvprof ο χρόνος kernel time φαίνεται να έχει μειωθεί 8% (κατά μέσο όρο).

5.Δυνατότητες πόρων

Το μέγιστο μέγεθος εικόνας που μπορούμε να υποστηρίξουμε μετά την υλοποίηση της blocked έκδοσης του κώδικα μας είναι 16384X16384.
Ο πόρος που μας περιορίζει είναι το μέγεθος του heap size στον host μας που δεν αρκεί κατά την δυναμική δέσμευση μνήμης με την calloc, έτσι όταν δοκιμάζουμε να τρέξουμε το πρόγραμμα για μέγεθος εικόνας 32768X32768 συμβαίνει buffer overflow και το πρόγραμμα μας τερματίζει.