ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ Η/Υ

ECE415 - ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΜΟΥ ΥΨΗΛΩΝ ΕΠΙΔΟΣΕΩΝ

Lab 5 - Παραλληλοποίηση και βελτιστοποίηση ολοκληρωμένης εφαρμογής βελτίωσης αντίθεσης εικόνας στη GPU

Φοιτητές:

Καραγεώργος Νικόλαος 02528, nkarageorgos@uth.gr

Λαμπρινός Ισίδωρος 02551, ilamprinos@uth.gr

Ιανουάριος 2022

Περιεχόμενα

Εισαγωγή

Στην παρούσα εργασία μας ζητήθηκε η παραλληλοποίηση και βελτιστοποίηση ολοκληρωμένης εφαρμογής στη GPU η οποία πραγματοποιεί βελτίωση της αντίθεσης εικονών με τόνους του γκρι (ασπρόμαυρες) με χρήση της διαδικασίας εξίσωσης ιστογράμματος.

Όλη η ανάπτυξη έγινε στο σύστημα inf-mars1 (10.64.82.31) το οποιο διαθέτει μία κάρτα GTX690, με 2 chips, ενώ οι τελικές μετρήσεις εγίναν στο πιο ισχυρό csl-artemis (10.64.82.65) το οποίο διαθέτει μία κάρτα Tesla K80 η οποία έχει 2 GK210 GPU chips

Device query του csl artemis

```
./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          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11997020160 bytes)
  (013) 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 shared memory per multiprocessor:        114688 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 Managed Memory:                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          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11997020160 bytes)
  (013) 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 shared memory per multiprocessor:        114688 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 Managed Memory:                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 = 11.4, CUDA Runtime Version
     = 11.5, NumDevs = 2
Result = PASS
```

Στρατηγική παραλληλοποίησης

Η εφαρμογή του μετασχηματισμού εξίσωσης ιστογράμματος συνοψίζεται στα παρακάτω βήματα:

1. Υπολογισμός του ιστογράμματος της αρχικής εικόνας.
2. Υπολογισμός της σωρευτικής πυκνότητας πιθανότητας κάθε απόχρωσης του γκρι
3. Κατασκευή ενός πίνακα αναφοράς ο οποίος θα χρησιμοποιηθεί για την

απεικόνιση της κάθε απόχρωσης του γκρι στην αρχική εικόνα στην αντίστοιχη απόχρωση στην εξισωμένη εικόνα

4. Κατασκευή της τελικής εικόνας.

Στην εργασία το βήμα 1 υλοποιείται από την συνάρτηση histogram ενώ τα βήματα 2, 3 και 4 υλοποιούνται μαζί από την συνάρτηση histogram_equalization. Κάνοντας profiling στον κώδικα που μας δίνεται διαπιστώνουμε ότι τα κύρια "hotspots" είναι οι δύο παραπάνω συναρτήσεις. Για είσοδο την δεύτερη μεγαλύτερη εικόνα (ship.pgm) οι δύο συναρτήσεις καταλαμβάνουν σχετικά ίδιο χρόνο εκτέλεσης. Για είσοδο την πρώτη μεγαλύτερη εικόνα (planet_surface.pgm) η histogram_equalization να καταλαμβάνει παράπανω χρόνο συγκριτικά με την histogram. Οπότε αποφασίσαμε να ξεκινήσουμε την παραλληλοποίηση της εφαρμογής από την histogram_equalization.

Histogram equalization

Η συνάρτηση αυτή αποτελείται από δύο ευδιάκριτα κομμάτια. Το πρώτο που υπολογίζει τη σωρευτική πυκνότητα πιθανότητας (cdf) κάθε απόχρωσης του γκρι και κατασκευάζει τον πίνακα αναφοράς(lut-lookup table) ο οποίος θα χρησιμοποιείται για την απεικόνιση της κάθε απόχρωσης του γκρι στην αρχική εικόνα στην αντίστοιχη απόχρωση στην εξισωμένη εικόνα.Ενώ το δεύτερο κομμάτι κατασκεύαζει την τελική εικόνα. Οπότε για αυτή την συνάρτηση υλοποιήσαμε δύο ξεχωριστούς kernels, έναν για κάθε κομμάτι.

Ο πρώτος kernel (lut_gpu) κατασκευάζει τον πίνακα cdf με την τεχνική prefix sum χρησιμοποιώντας parallel reduction. Για την αποθήκευση του πίνακα cdf χρησιμοποιούμε την shared memory καθώς η συγκεκριμένη τεχνική δεν δημιουργεί bank conflicts. Ωστόσο παρόλο που η μέθοδος που χρησιμοποίησαμε δεν είναι η πιο αποδοτική (naive prefix sum) κάθως εκτελεί τους περισσοτέρους υπολογισμούς με σειριακό τρόπο, δοκιμάσαμε και την πιο αποδοτική μέθοδο η οποία είναι αρκέτα πολυπλοκότερη αλλά δεν είχε κάποια διαφόρα στον χρόνο καθώς ο πίνακας με τον οποίο δουλεύουμε είναι αρκετά μικρός (256 ints). Φτιάχνουμε τον συγκεκριμένο kernel με την γεωμετρία ενός block με 256 threads όσες και οι πιθανές τιμές των πινάκων hist και lut. Να σημειωθεί επίσης πως για το πρώτο κομμάτι της histogram_equalization ο κώδικας που βρίσκει το min (δηλαδή την πρώτη θέση στον πίνακα hist για την οποία υπάρχουν pixels που έχουν

την τιμή της) παραμένει στην CPU και δίνουμε ως όρισμα το min στον kernel lut_gpu. Ο χρόνος εκτέλεσης αυτού του κώδικα είναι μηδαμινός.

```
__global__ void lut_gpu(int *lut, int *hist, int img_size, int min){


    int idx  = threadIdx.x, d;
    __shared__ int scdf[nbr_bin];

    scdf[idx] = hist[idx];

    for( d = 1; d < nbr_bin; d = d << 1){
        __syncthreads();

        if( idx < (nbr_bin-d) ){
            scdf[idx + d] += scdf[idx];
        }
    }

    __syncthreads();

    lut[idx] = (int)(((float)scdf[idx] - min)*255/(img_size - min) + 0.5);

}
```

Figure 1: lut_gpu kernel

Τον δεύτερο kernel (result_image) που κατασκευάζει την τελική εικόνα τον φτιάχνουμε με γεωμέτρια image_size/1024 blocks (+1 block αν το υπόλοιπο της διαίρεσης δεν είναι 0) με 1024 threads. Επειδή στην περιπτωση που δεν διαιρείται ακριβώς το image_size με το 1024 θα υπάρχουν παραπάνω threads από ότι pixels ελέγχουμε έαν το μοναδικό index του thread στο grid είναι εντός εικόνας και μέτα αυτό προχωράει σε υπολογισμό της νέας του τιμής στην τελική εικόνα.

```
__global__ void image_result(unsigned char * img_out, int *lut, unsigned char * img_in, int img_size){

   int idx = threadIdx.x + blockIdx.x*blockDim.x;

   int reg = lut[img_in[idx]];

   if (idx < img_size){
       if(reg > 255){
           img_out[idx] = 255;
       }
       else if(reg < 0){
           img_out[idx] = 0;
       }
       else{
           img_out[idx] = (unsigned char)reg;
       }
   }

}
```

Figure 2: result_image kernel

Για την δέσμευση μνήμης της τελικής εικόνας μετά απο δοκιμές (cudaMalloc, cudaMallocHost) είδαμε ότι μας συμφέρει περισσότερο η cudaMallocManaged. Με αυτόν τον τρόπο ο kernel γράφει το αποτέλεσμα της τελικής εικόνας στον πίνακα result.img και τον ίδιο πίνακα επιστρέφουμε στον host χωρίς να χρειαστεί να τον κάνουμε cudaMemcpy όλη την εικόνα ξανά το οποίο θα ήταν αρκέτα πιο δαπανηρό. Για την δέσμευση μνήμης της αρχικής εικόνας είδαμε ότι δεν υπάρχει διαφορά στον χρόνο εκτέλεσης και την αφήσαμε ως έχειν.

Σε αυτή την υλοποίηση δοκιμάσαμε να αποθηκεύσουμε τον πίνακα lut (look-up table) στην constant και στην shared memory άλλα είδαμε ότι καμία απο τις δύο δεν μας συνέφερε αφού ο χρόνος εκτέλεσης αυξάνοταν και έτσι τον αφήσαμε στην global. Αυτό ίσως εξηγείται καθώς το μοτίβο προσπέλασης του look-up table είναι τυχαίο οπότε στην constant memory οι προσπελάσεις των threads του ίδιου warp σε διαφορετικές διευθύνσεις θα γινόντουσαν σειριακά και στην shared memory θα ήταν πολύ πιθανό να δημιουργηθούν bank conflicts.

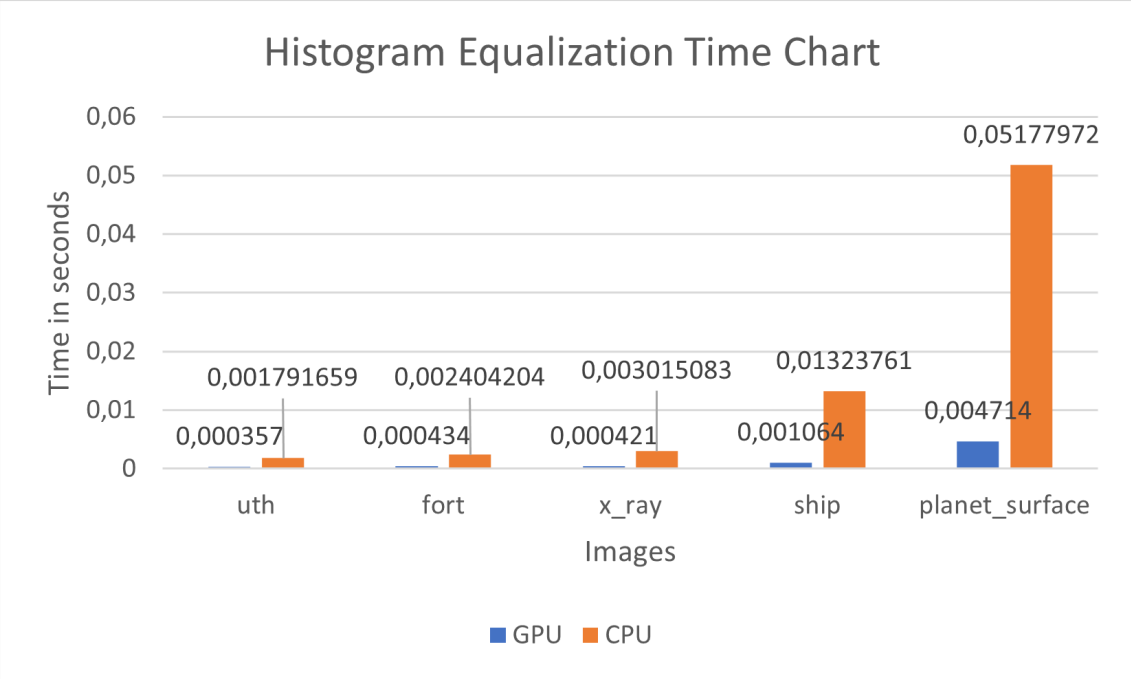


Figure 3: Histogram equalization time on CPU and GPU

Στο χρόνο εκτέλεσης της GPU συμπεριλαμβάνονται οι εκτελέσεις των kernel αλλά και οι μεταφορές δεδομένων από τον host στο device και το αντίστροφο. Παρατηρούμε ότι ο χρόνος της histogram equalization μείωνεται αρκέτα στη GPU και η διαφορά αυτή γίνεται πιο έντονη όσο μεγαλώνει το μέγεθος της εικόνας.

Histogram

Η συνάρτηση histogram σαρώνει όλη την εικόνα για να καταγράψει την τιμή κάθε pixel και έτσι αυξάνει την τιμή διαφόρων θέσεων του πίνακα hist με απρόβλεπτο τρόπο. Οπότε αυτή η αύξηση θα πρέπει να γίνεται ατομικά με την χρήση της atomicAdd η οποία βεβαία μετατρέπει αρκέτα τον κώδικα σε σειριάκο. Χρησιμοποιούμε την shared memory για την αποθήκευση του hist οπότε οι ατομικές αυξήσεις γίνονται εκεί και στο τέλος τα partial sum που έχουν υπολογίστει από κάθε block αθροίζονται με μία ακόμη atomicAdd και το τελικό αποτέλεσμα αποθηκεύεται στην global. Χωρίς την χρήση της shared memory σε αυτήν την υλοποίηση ο χρόνος εκτέλεσης ηταν αρκετά χειρότερος.

Τα blocks αυτού του kernel είναι διδιάστατα 16x16 , δηλαδή κάθε block έχει 256 threads όσα και του μέγεθος του hist.

```
__global__ void histogram_gpu( int *hist_out, unsigned char *img_in, int img_size){

    __shared__ int shist[256];
    int x = threadIdx.x + blockIdx.x*blockDim.x;
    int y = threadIdx.y + blockIdx.y*blockDim.y;
    int offset = blockDim.x*gridDim.x;
    int idx = threadIdx.x + threadIdx.y*blockDim.x;
    int image_idx = x +y*offset;

    shist[idx] = 0;

    if(image_idx < img_size){
        __syncthreads();
        atomicAdd(&shist[img_in[image_idx]], 1);
    }

    __syncthreads();

    atomicAdd(&hist_out[idx], shist[idx]);

}
```

Figure 4: histogram kernel

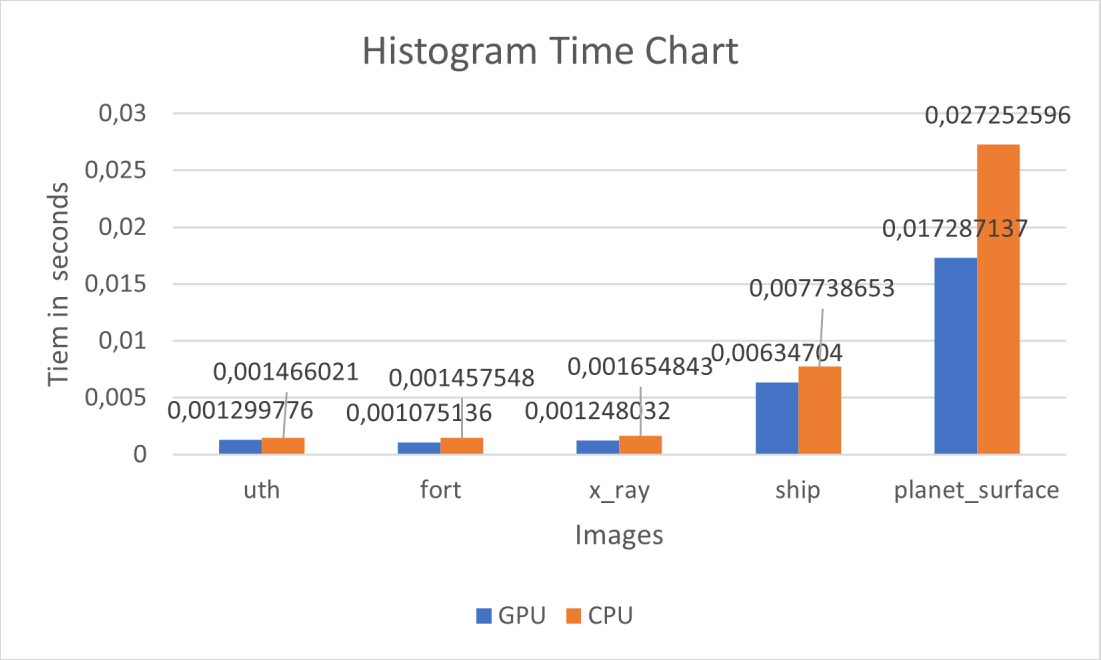
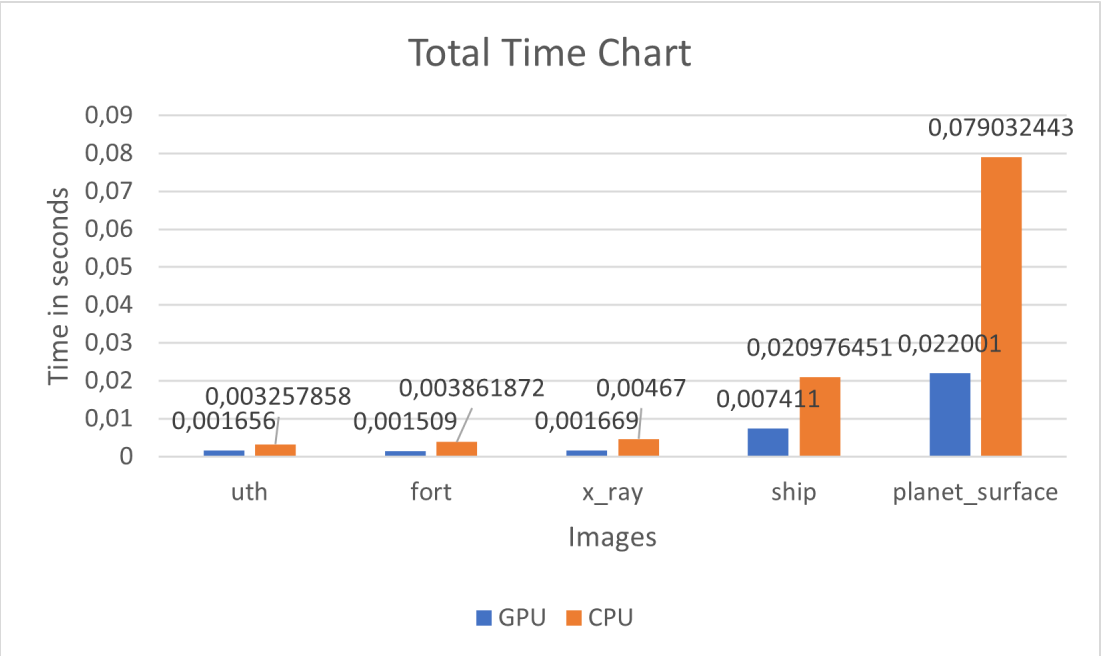


Figure 5: Histogram time on CPU and GPU

Παρατηρούμε ότι ο χρόνος της histogram μείωνεται αρκέτα στη GPU και η διαφορά αυτή γίνεται πιο έντονη όσο μεγαλώνει το μέγεθος της εικόνας. Ωστόσο δεν πετυχένεται το ίδιο μεγάλο ποσοστό μείωσης του χρόνου εκτέλεσης όπως στην histogram equalization. Αυτό οφείλεται στην χρήση των atomicAdd που μειώνουν την παραλληλοποίηση.

Σύγκριση συνολικού χρόνου εκτέλεσης σε CPU και GPU



Στο χρόνο εκτέλεσης της GPU συμπεριλαμβάνονται όλες οι εκτελέσεις των kernel αλλά και όλες οι μεταφορές δεδομένων από τον host στο device και το αντίστροφο. Παρατηρούμε πως στην GPU πετυχαίνονται καλύτεροι χρόνοι για όλες τις εικόνες που μας δίνονται και αυτή η διαφορά αυτή γίνεται πιο έντονη όσο μεγαλώνει το μέγεθος της εικόνας.

Μεταγλώττιση κώδικα

Ο κώδικας βρίσκεται μέσα στο φάκελο Code και μεταγλωττίζεται με την εντολή:

nvcc -O4 -g main.cu histogram-equalization.cu contrast-enhancement.cu -o executable

Η υλοποίηση των kernels γίνεται στο αρχείο contrast-enhancement.cu οπότε το αρχείο histogram-equalization.cu δεν χρησιμοποιείται κάπου και μπορεί να παραληφθεί.