ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΜΟΥ ΥΨΗΛΩΝ ΕΠΙΔΟΣΕΩΝ (ECE 415)

Ακαδημαϊκό έτος 2021-2022

5η Εργαστηριακή Άσκηση

Παραλληλοποίηση και βελτιστοποίηση ολοκληρωμένης εφαρμογής στην GPU.

Φοιτητές:

Ηλιάδης Ηλίας, ΑΕΜ: 2523

Μακρής Δημήτριος-Κων/νος – ΑΕΜ: 2787

Σκοπός της 5ης εργαστηριακής άσκησης ήταν παραλληλοποίηση και βελτιστοποίηση μιας εφαρμογής η οποία πραγματοποιεί εξίσωση ιστογράμματος εικόνας, με σκοπό την βελτίωση της αντίθεσής της, κι έτσι, και της ποιότητάς της.

CUDA Device Query

```
 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 simulta-
neously) >
```

```
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 simulta-
neously) >
> 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
```

1. Παραλληλοποίηση histogram equalization στην GPU

Αρχικά λάβαμε μετρήσεις απο όλα τα στάδια της διαδικασίας υπολογισμού του ιστογράμματος στην CPU σε συνδυασμό με τον profiler της nvidia και εντοπίσαμε ότι το πιο χρονοβόρο κομμάτι του υπολογισμού είναι η συνάρτηση **histogram equalization.** Με βάση τις μετρήσεις αποφασίσαμε να βελτιστοποιήσουμε τον υπολογισμό μεταφέροντας το histogram equalization στην GPU, αφήνοντας προς το παρόν τον υπολογισμό του ιστογράμματος στην CPU. Συγκεκριμένα υπολογίζεται το ιστόγραμμα στην CPU και έπειτα μεταφέρεται το αποτέλεσμα στην GPU ούτως ώστε να υπολογιστεί το equalization. Για να ξεκινήσουμε την υλοποίηση μας στην GPU μεταφέραμε σε δυο kernel τους εικονιζόμενους υπολογισμούς του histogram equalization:

```
for(i = 0; i < nbr_bin; i ++){
    cdf += hist_in[i];
    //lut[i] = (cdf - min)*(nbr_bin - 1)/d;
    lut[i] = (int)(((float)cdf - min)*255/d + 0.5);
    if(lut[i] < 0){
        lut[i] = 0;
    }

    |
}
```

Υπολογισμός cdf και lut στον kernel **cdf_lut**

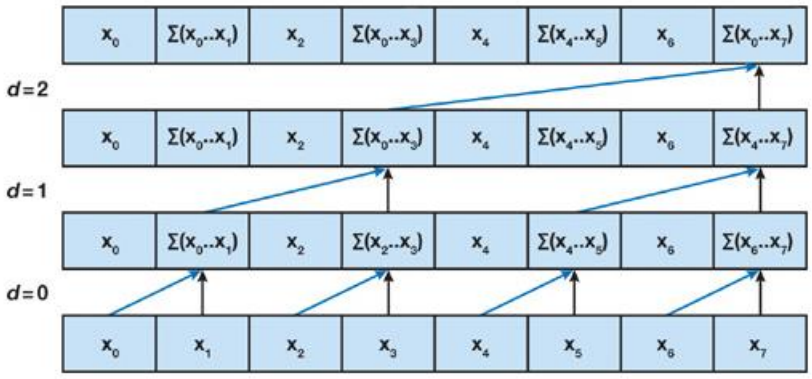
```
/* Get the result image */
for(i = 0; i < img_size; i ++){
    if(lut[img_in[i]] > 255){
        img_out[i] = 255;
    }
    else{
        img_out[i] = (unsigned char)lut[img_in[i]];
    }

}
```

Κατασκευή τελικής εικόνας στον kernel **resultImage**

Για τον υπολογισμό του cdf στην GPU, υλοποιήσαμε την παράλληλη εκδοχή του αλγόριθμου **prefix sum,** χρησιμοποιώντας ισοζυγισμένο δυαδικό δέντρο (balanced binary tree) ούτως ώστε να προσεγγίσουμε το efficiency του ακολουθιακού αλγορίθμου και να εκμεταλλευτούμε τον παραλλιλισμό της GPU στον μέγιστο δυνατό βαθμό. Σύμφωνα με τον τρόπο με τον οποίο εκτελείται ο αλγόριθμος, η γεωμετρία του Grid αποτελείται απο ένα μονοδιάστατο block με 128 threads.

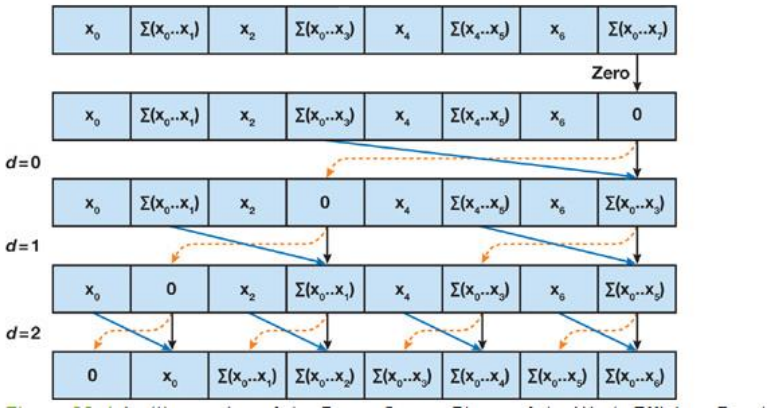
Ο αλγόριθμος αποτελείται απο δύο στάδια, **up-sweep phase** και **down-sweep phase**.

Up-sweep phase: Στο πρώτο στάδιο διατρέχεται το δυαδικό δέντρο ξεκινώντας απο τα φύλλα προς την ρίζα, υπολογίζοντας σε κάθε στάδιο τοπικά αθροίσματα απο κόμβους του δέντρου που έχουν απόσταση d = 0, 1, ... ,log(n) μεταξύ τους, όπου **n** το μέγεθος του πίνακα cdf. Τα αποτελέσματα του αθροίσματος αποθηκεύονται κάθε φορά στο δεξιότερο στοιχείο μέχρι να φτάσουμε στην ρίζα, όπου εκεί θα περιέχεται το συνολικό άθροισμα απο όλα τα στοιχεία του πίνακα του ιστογράμματος. Τα αθροίσματα υπολογίζονται σε κάθε στάδιο απο τα μισά threads, καταλήγοντας στο τελευταίο στάδιο με 1 thread. Η εν λόγω διαδικασία απεικονίζεται παρακάτω:



Up-sweep phase of work-efficient sum scan algorithm

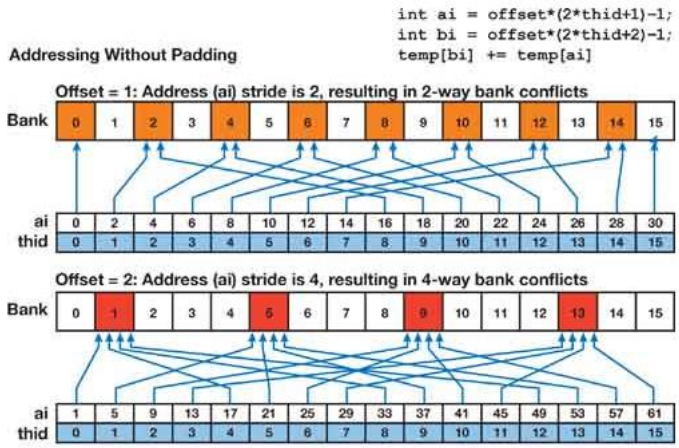
Down-sweep phase: Σε δεύτερη φάση, δατρέχεται απο πολλαπλά threads ξανά το δέντρο, ξεκινώντας απο την ρίζα προς τα φύλλα ξεκινώντας με ένα thread μέχρις ότου να εκτελούνται καί τα 128 threads του block. Αρχικά ενθέτουμε την τιμή 0 στην ρίζα του δέντρου και σε κάθε στάδιο, κάθε κόμβος του εν λόγω επιπέδου ενθέτει την τιμή του στο αριστερό παιδί και το άθροισμα απο την τρέχουσα τιμή του κόμβου και της προηγούμενης τιμής του αριστερού παιδιού (πριν την ένθεση) στο δεξί παιδί. Η διαδικασία ολοκληρώνεται έχοντας ενθέσει στον πίνακα cdf όλα τα αθροίσματα. Η εν λόγω διαδικασία απεικονίζεται παρακάτω:



Down-sweep phase of work-efficient sum scan algorithm

Σημειώνεται ότι επειδή υπάρχει εξάρτηση δεδομένων μεταξύ των στοιχείων, με την ολοκήρωση κάθε επιπέδου συγχρονίζονται όλα τα threads που συμμετείχαν στον υπολογισμό.

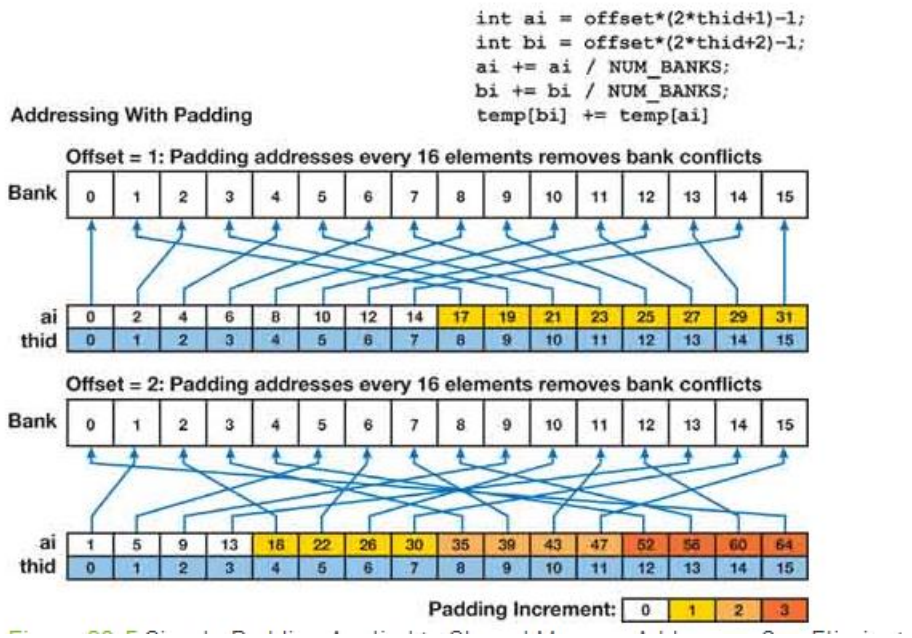
Bank Conflict Avoidance: Ο εν λόγω αλγόριθμος επιδέχεται περεταίρω βελτιστοποίηση , λόγω του κακού μοτίβου προσπέλασεων που παρουσιάζει μέχρι στιγμής στην μνήμη της GPU. Συγκεκριμένα επειδή δεσμεύουμε στην shared memory για τον υπολογισμό του cdf έναν πίνακα ακεραίων 256 θέσεων, με το τρέχων memory access pattern υπάρχει περίπτωση πολλαπλά thread του ίδιου wrap να πραγματοποιήσουν εγγραφές στο ίδιο bank, προκαλώντας έτσι **bank conflicts**. Η παρουσία bank conflict προκαλεί σειριακή προσπέλαση των δεδομένων. Συγκεκριμένα σε ένα ισοζυγισμένο δυαδικό δέντρο, διπλασιάζεται το stride μεταξύ προσπελάσεων στην μνήμη σε κάθε επίπεδο του υπολογισμού, διπλασιάζοντας παράλληλα τον αριθμό των thread που συμπέφτουν στο ίδιο bank. Ο τρόπος με τον οποίο δημιουργούνται τα bank conflicts κατα την διάρκεια του υπολογισμού παρουσιάζεται παρακάτω:



Για να αποφύγουμε τα bank conficts καθώς πραγματοποιούμε προσπελάσεις στην shared memory, προσθέτουμε κατάλληλο padding στον πίνακα της shared memory ανά **#banks** στοιχεία. Αυτο πραγματοποιείται χρησιμοποιώντας το macro:

```
#define NUM_BANKS 32

#define LOG_NUM_BANKS 5

#define CONFLICT_FREE_OFFSET(n) \

((n) >> NUM_BANKS + (n) >> (2 * LOG_NUM_BANKS))
```

Θέσαμε NUM_BANKS = 32 σύμφωνα με την αρχιτεκτονική του device στο csl-artemis. Η διαδικασία εισαγωγής offset για την εξάλειψη των conflict απεικονίζεται απρακάτω (παράδειγμα αρχιτεκτονικής με 16 banks):



Ο ολοκληρωμένος κώδικας του υπολογισμού του cdf και του lut παρουσιάζεται παρακάτω:

```
__global__ void cdf_lut(int d_in, int min, int *d_hist, int nbr_bin, int img_size){
    extern __shared__ int cdf[];
    int thread_id = threadIdx.x;
    int offset = 1;

    int ai = thread_id;
    int bi = thread_id + (nbr_bin / 2);
    int ai_hist = d_hist[ai];
    int bi_hist = d_hist[bi];
    int temp;

    cdf[ai + CONFLICT_FREE_OFFSET(ai)] = ai_hist;
    cdf[bi + CONFLICT_FREE_OFFSET(bi)] = bi_hist;


    for(int d = nbr_bin >> 1; d > 0; d >>= 1)
    {
        __syncthreads();
        if(thread_id < d)
        {

            ai = offset * ((2*thread_id) + 1) - 1;
            bi = offset * ((2*thread_id) + 2) - 1;

            ai += CONFLICT_FREE_OFFSET(ai);
            bi += CONFLICT_FREE_OFFSET(bi);

            cdf[bi] += cdf[ai];
        }

        offset *= 2;
    }

    if(thread_id == 0) {
        cdf[nbr_bin - 1 + CONFLICT_FREE_OFFSET(nbr_bin - 1)] = 0;
    }

    for(int d = 1; d < nbr_bin; d *= 2)
    {
        offset >>= 1;
        __syncthreads();

         if(thread_id < d)
         {
             ai = offset * ((2*thread_id) + 1) - 1;
             bi = offset * ((2*thread_id) + 2) - 1;

             ai += CONFLICT_FREE_OFFSET(ai);
             bi += CONFLICT_FREE_OFFSET(bi);

             temp = cdf[ai];
             cdf[ai] = cdf[bi];
             cdf[bi] += temp;
         }
     }

     __syncthreads();
     ai = thread_id;
     bi = thread_id + nbr_bin / 2;

     cdf[ai + CONFLICT_FREE_OFFSET(ai)] += ai_hist;
     cdf[bi + CONFLICT_FREE_OFFSET(bi)] += bi_hist;



     d_hist[ai] = (int)(((float)cdf[ai + CONFLICT_FREE_OFFSET(ai)]-min)*255/d_in+0.5);
     if(d_hist[ai] < 0) { d_hist[ai] = 0; }
     d_hist[bi] = (int)(((float)cdf[bi+CONFLICT_FREE_OFFSET(bi)]-min)*255/d_in+0.5);
     if(d_hist[bi] < 0) { d_hist[bi] = 0; }
}
```

Για την διαδικασία παραγωγής της τελικής εικόνας, δημιουργήσαμε τον kernel resultImage, στον οποίο φορτώνουμε τον πίνακα που περιέχει τις τιμές του lut απο την global στην shared memory του κάθε block καθώς πραγματοποιούνται αρκετά loads κατα την διαδικασία παραγωγής της τελικής εικόνας. Η γεωμετρία του grid αποτελείται απο (img_size/1024) + 1 blocks των 1024 threads. Το extra block δημιουργείται σε περίπτωση που το μέγεθος της εικόνας δεν είναι ακέραιο πολλαπλάσιο των thread per block, ώστε να πραγματοποιηθούν οι υπολογισμοί που προκύπτουν απο το υπόλοιπο της διάιρεσης. Επίσης για να αποκλεισουμε την δημιουργία thread εκτός των ορίων της εικόνας χρησιμοποιούμε την if που φαίνεται στον παρακάτω κώδικα του kernel:

```
__global__ void resultImage(int *lut_GPU, unsigned char *img_in, unsigned char * img_out,size_t img_size)
{
    int tx = blockIdx.x*blockDim.x+threadIdx.x;
    int val = img_in[tx];
    extern __shared__ int lut[];

    if( threadIdx.x < 256) {
        lut[threadIdx.x] = lut_GPU[threadIdx.x];
    }
    __syncthreads();
    if( (size_t)tx  < img_size) {
        if(lut[val] > 255 )
            img_out[tx] = 255;
        else
            img_out[tx] = (unsigned char)lut[val];
    }
}
```

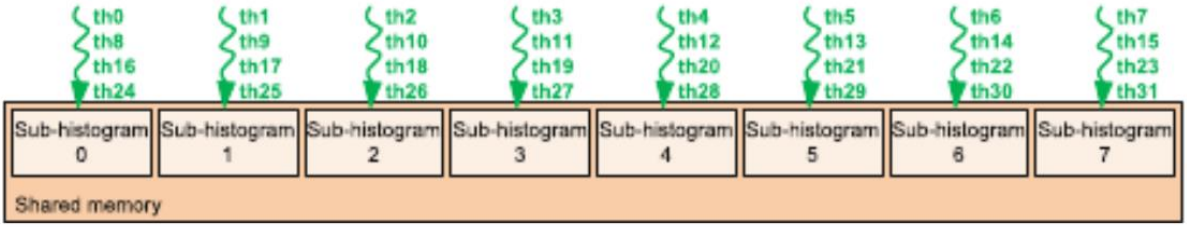
2. Υπολογισμός εξίσωσης ιστογράμματος αποκλειστικά στην GPU

Έχοντας μεταφέρει τον υπολογισμό του histogram-equalization στην GPU, αποφασίσαμε να πραγματοποιήσουμε και την παραγωγή του ιστογρλαμματος στην GPU, εκμεταλλεύομενοι τον παραλληλισμό που προσφέρει η GPU. Ο αρχικός κώδικας της CPU είναι ο ακόλουθος:

```
void histogram(int * hist_out, unsigned char * img_in, int img_size, int nbr_bin){
    int i;
    for ( i = 0; i < nbr_bin; i ++){
        hist_out[i] = 0;
    }

    for ( i = 0; i < img_size; i ++){
        hist_out[img_in[i]] ++;
    }
}
```

Ο τρόπος υλοποίησης του υπολογισμού του ιστογράμματος βασίστηκε κυρίως στην συσχέτιση των τιμών των γειτωνικών pixel της εικόνας. Συγκεκριμένα, οι τιμές των περισσότερων τιμών των γειτονικών pixel (δηλαδή το "χρώμα" του) είτε είναι ίδιες είτε βρίσκονται στο ίδιο εύρος τιμών. Με αυτον τον τρόπο, τα thread του ίδιου wrap είναι πολύ πιθανό να προσπελάσουν μικρό εύρος θέσεων του πίνακα του ιστογράμματος λόγω του distribution. Με αυτόν τον τρόπο δημιουργούνται αρκετά συχνά position conflicts κατα την εκτέλεση. Γι αυτο τον λόγο δημιουργούμε R αντίγραφα του πίνακα του ιστογράμματος ανα block (μερικά ιστογράμματα), όπου

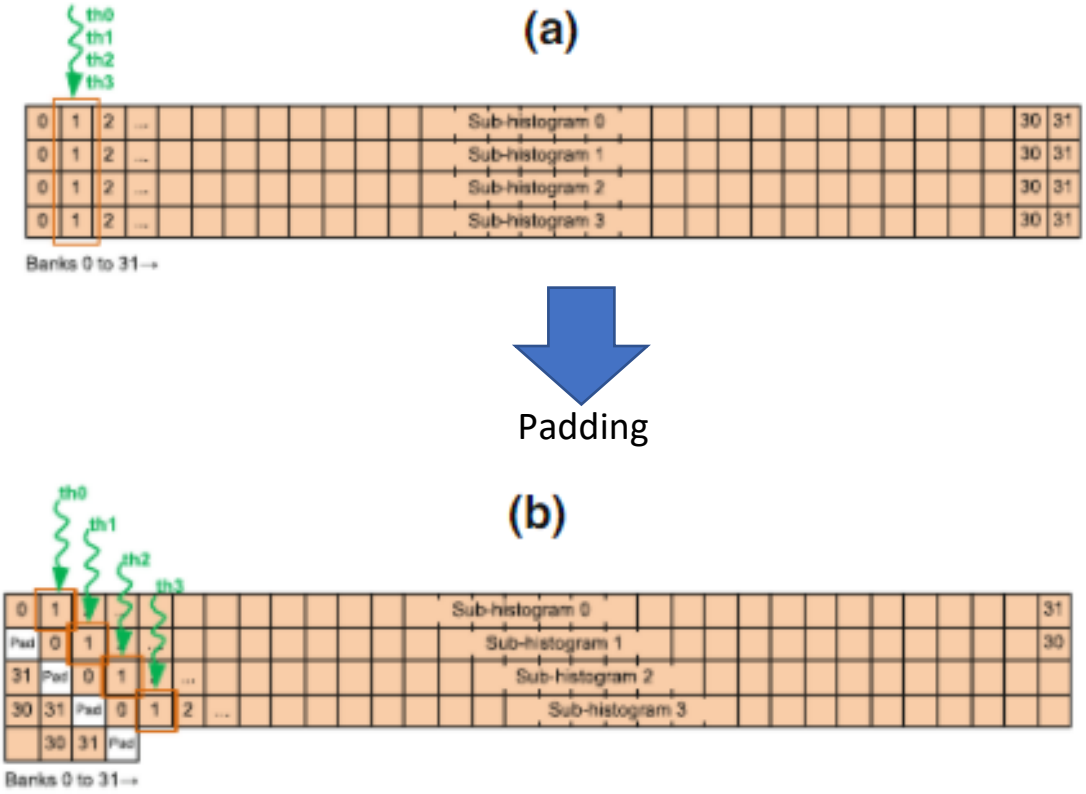
R = (histogram_array_size) / (threads per wrap) = 256/32 = 8 . Με αυτη την υλοποίηση, τα position conflicts που προαναφέρθηκαν μετατρέπονται σε bank conflicts, τα οποία εξαλείφονται με τρόπο που θα περιγράψουμε παρακάτω. Με την δημιουργία μερικών ιστογραμμάτων, διαδοχικά thread θα προσπελάυνουν διαδοχικά μερικά ιστογράμματα, μειώνοντας έτσι τις σειριακές προσπελάσεις στην μνήμη που προκαλούνται απο την ανάγνωση ενός στοιχείου του πίνακα απο πολλαπλά thread του ίδιου wrap. Η εν λόγω υλοποίηση παρουσιάζεται παρακάτω ως στιγμιότυπο εκτέλεσης ενός block:



Για να εξαλειφθούν τα bank conflicts που προκύπτουν, αρχικοποιούμε τον πίνακα στην shared memory εισάγοντας padding με τον ακόλουθο τρόπο:

```
__shared__ int sh_hist[(256+1)*8];
```

Η εξάλειψη των bank conflicts με το παραπάνω padding παρουσιάζεται παρακάτω:



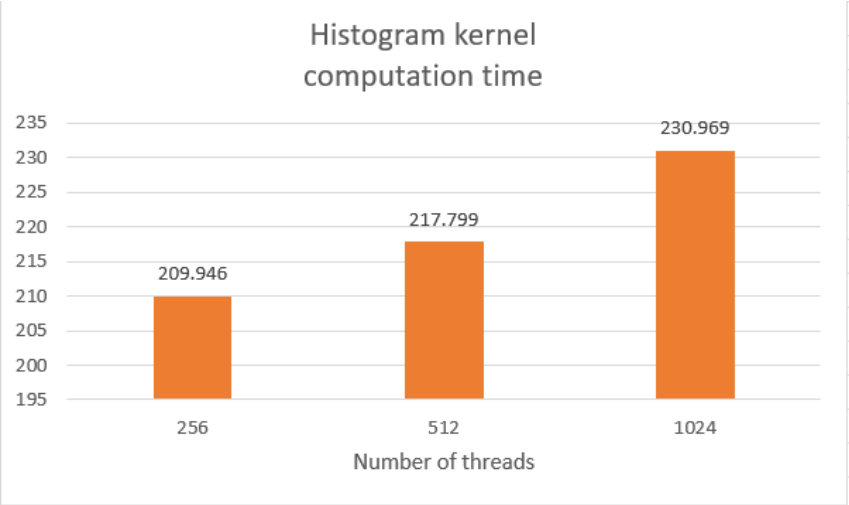
Τέλος, επειδή όπως αναφέραμε και πρίν οι περισσότερες εικόνες χωρίζονται σε περιοχές χρωμάτων, με αποτέλεσμα διαδοχικά wraps να προσπελαύνουν pixel με παραπλήσιες ή ίσες τιμές με αποτέλεσμα να δημιουργούνται πολλά inter-wrap conflicts. Συνεπώς, ο χρόνος εκτέλεσης του κώδικα του ιστογράμματος εξαρτάται άμεσα απο το distribution των pixel. Γι αυτό τον λόγο, τα δεδομένα προσπελαύνονται με τέτοιο τρόπο, ώστε wraps του ίδιου block να διαχωρίζονται όσο περισσότερο γίνεται. Το παραπάνω γίνεται με το ακόλουθο indexing:

```
//Indexes
const int warpid = (int)(threadIdx.x / 32);
const int lane = threadIdx.x % 32;
const int warpsperblock = blockDim.x / 32;

//offset to per-block sub-histogram
const int offset = (256+1) * (threadIdx.x % 8);

//Interleaved read access
const int begin = (size/warpsperblock)*warpid + 32 * blockIdx.x + lane;
const int end = (size/warpsperblock) * (warpid + 1);
const int step = 32 * gridDim.x;
```

Ο ολοκληρωμένος κώδικας στην GPU είναι ο εξής:

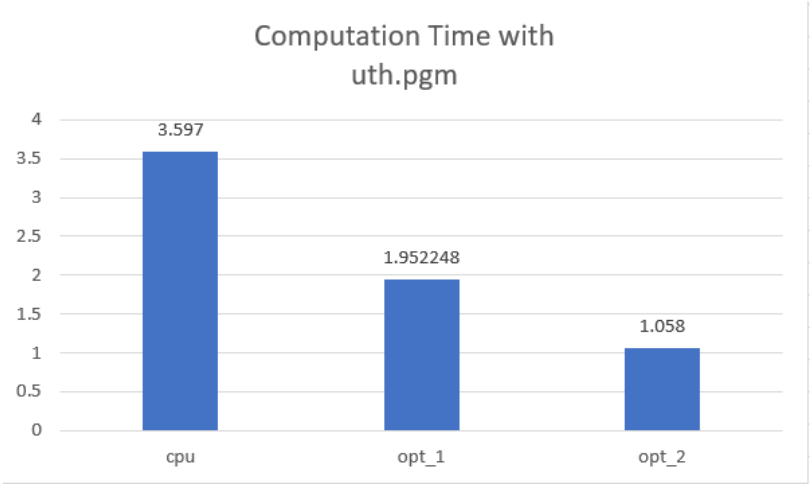
```
__global__ void histogram_GPU(int *hist_GPU, unsigned char *img,int size)
{
    //Padding 1 to nbr_bins (256)
    __shared__ int sh_hist[(256+1)*8];
    //Indexes
    const int warpid = (int)(threadIdx.x / 32);
    const int lane = threadIdx.x % 32;
    const int warpsperblock = blockDim.x / 32;

    //offset to per-block sub-histogram
    const int offset = (256+1) * (threadIdx.x % 8);

    //Interleaved read access
    const int begin = (size/warpsperblock)*warpid + 32 * blockIdx.x + lane;
    const int end = (size/warpsperblock) * (warpid + 1);
    const int step = 32 * gridDim.x;

    //Initialization
    for(int pos = threadIdx.x; pos < ((256+1)*8); pos+=blockDim.x){
        sh_hist[pos] = 0;
    }
    __syncthreads();

    for(int i = begin; i < end; i+=step){
        int d = img[i];
        atomicAdd(&sh_hist[offset + d], 1);
    }

    __syncthreads();

    //Merge sub_histograms and write in global memory
    for(int pos = threadIdx.x; pos < 256; pos+=blockDim.x){
        int sum = 0;
        for(int base = 0; base < ((256+1) * 8); base+= 256+1){
                sum+= sh_hist[base+pos];
        }
        atomicAdd(hist_GPU + pos, sum);
    }

}
```

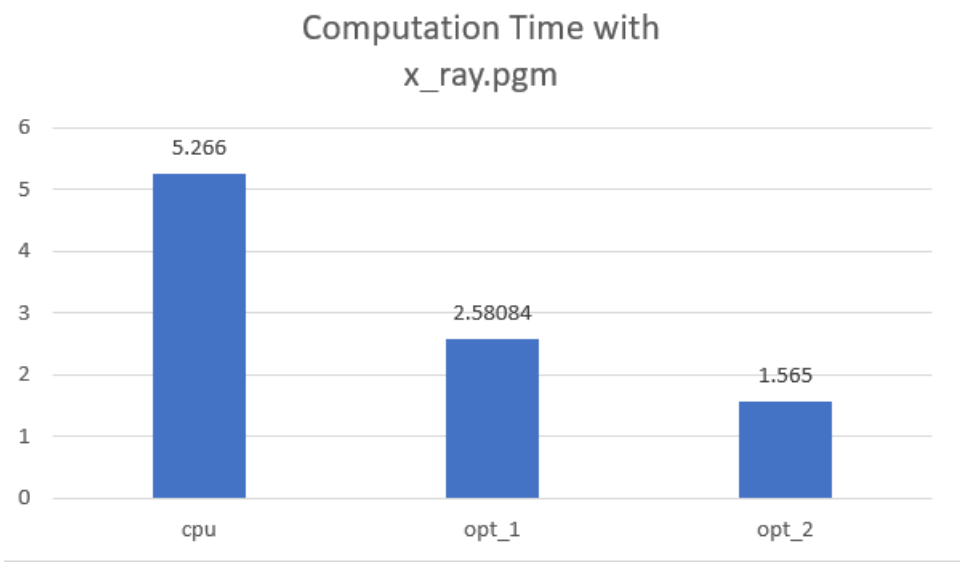
Παρακάτω παραθέτουμε τον χρόνο υπολογισμού του ιστογράμματος(σε msec) για διαφορετικό αριθμό thread per block, επιλέγοντας εικόνα εισόδου μεγέθους 375 MB:

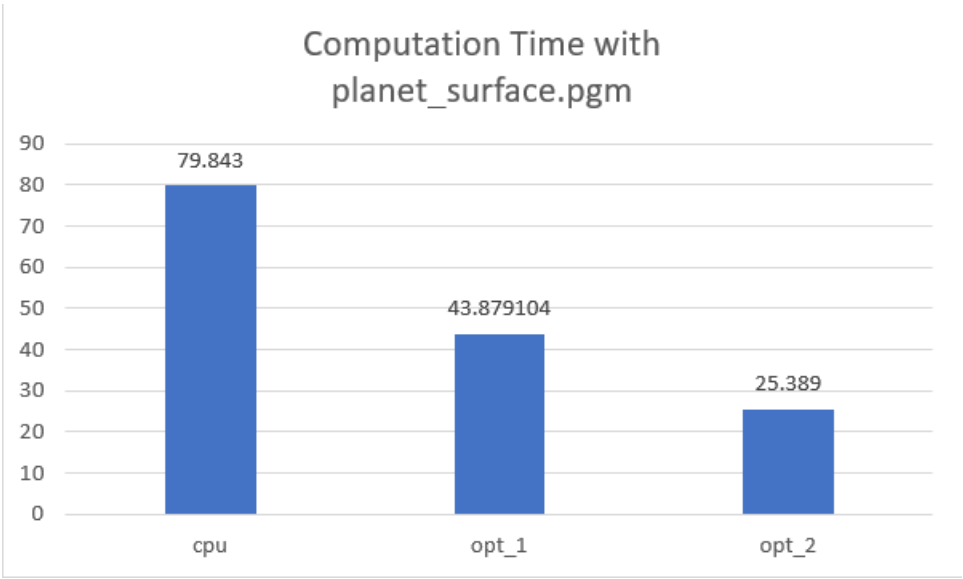


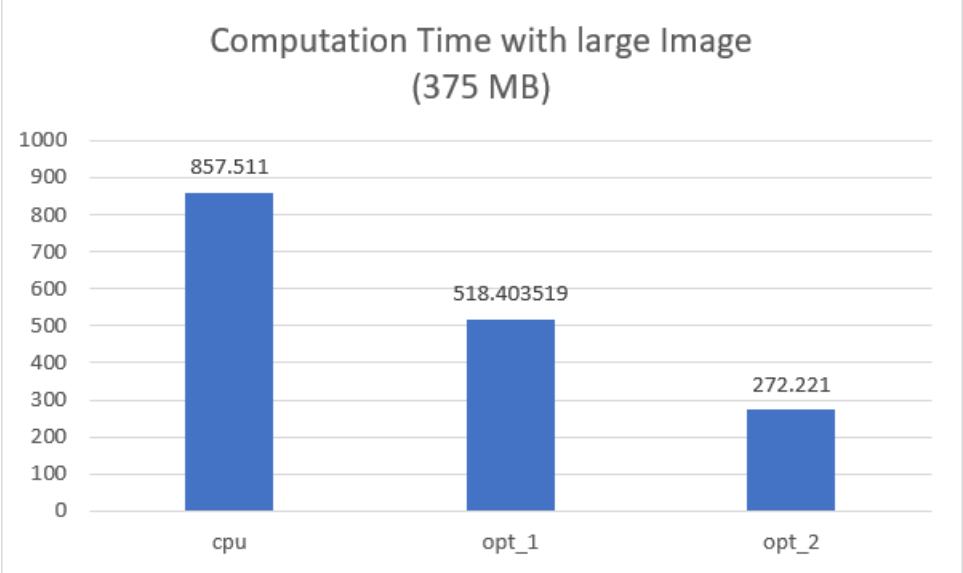
3. Μετρήσεις χρόνου υπολογισμού όλων των εκδόσεων

Παρακάτω παρουσιάζονται τα διαγράμματα των χρόνων υπολογισμού της εξίσωσης του ιστογράμματος **σε milliseconds** για διάφορες εικόνες εισόδου, μεταξύ αλλων και μια εικόνα που δημιουργήσαμε εμείς με μέγεθος 375MB ώστε να ελέχουμε καλύτερα την επίδοση της κάθε έκδοσης. Οι χρόνοι που παρουσιάζονται προκύπτουν απο τον μέσο όρο 12 διαδοχικών εκτελέσεων απο τις οποίες αφαιρείται η μέγιστη και η ελάχιστη τιμή. Οι μεταβλητές στον οριζόντιο άξονα αντιστοιχούν στις εκδόσεις με την σειρά που τις παρουσιάσαμε.




Computation Time with
x_ray.pgm
6
5
4
3
2
1
0
5.266
2.58084
1.565
cpu
opt_1
opt_2


Computation Time with
planet_surface.pgm
90
80
70
60
50
40
30
20
10
0
79.843
43.879104
25.389
cpu
opt_1
opt_2



4. Οδηγίες μεταγλώττισης / εκτέλεσης

Περιεχόμενα παραδοτέου:

1. **Code:** περιέχει τον αρχικό κώδικα στην CPU.
2. **gpu_code_v1:** περιέχει τον κώδικα που παρουσιάζεται στην πρώτη βελτιστοποίηση.
3. **gpu_code_v2:** περιέχει τον κλωδικα που παρουσιάζεται στην τελευταία βελτιστοποίηση.
4. **metrics.xlsx:** μετρήσεις χρόνου υπολογισμού για όλες τις εκδόσεις.

Σκάθε έκδοση βελτιστοποίησης (Code/ , gpu_code_v1/ , gpu_code_v2/) περιέχεται ένα Makefile για την μεταγλώττιση και την παραγωγή εκτελέσιμων αρχείων. Για την εκτέλεση του κώδικα εκτελούμε την παρακάτω εντολή στο τερματικό:

./main <input_image_file_path> <output_image_file_path>