ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΜΟΥ ΥΨΗΛΩΝ ΕΠΙΔΟΣΕΩΝ (ECE 415)

Ακαδημαϊκό έτος 2021-2022

4η Εργαστηριακή Άσκηση

Βελτιστοποίηση υλοποιημένου προγράμματος CUDA, επικάλυψη υπολογισμών με streams.

Φοιτητές:

Ηλιάδης Ηλίας, ΑΕΜ: 2523

Μακρής Δημήτριος-Κων/νος – ΑΕΜ: 2787

0 Υπόβαθρο και στόχοι εργαστηριακής άσκησης

Στην συγκεκριμένη εργαστηριακή άσκηση κληθήκαμε να βελτιστοποιήσουμε τον ήδη υπάρχον κώδικα που υλοποιήθηκε στα πλαίσια της προηγούμενης άσκησης ο οποίος εφαρμόζει, με την χρήση της συνέλιξης, ένα διαχωρίσιμο δισδιάστατο φίλτρο πάνω σε ένα δισδιάστατο πίνακα (εικόνα) . Εκτός απο την αξιοποίηση του μοντέλου μνήμης στην CUDA, πειραματιστήκαμε με μεγέθη πινάκων μεγαλύτερα απο τα επιτρεπτέα και δημιουργήσαμε επικάλυψη μεταξύ των υπολογισμό των kernel και μεταφοράς δεδομένων με την χρήση των CUDA streams. Οι εν λόγω υλοποιήσεις περιγράφονται λεπτομερώς παρακάτω.

1 Πληροφορίες απο το 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

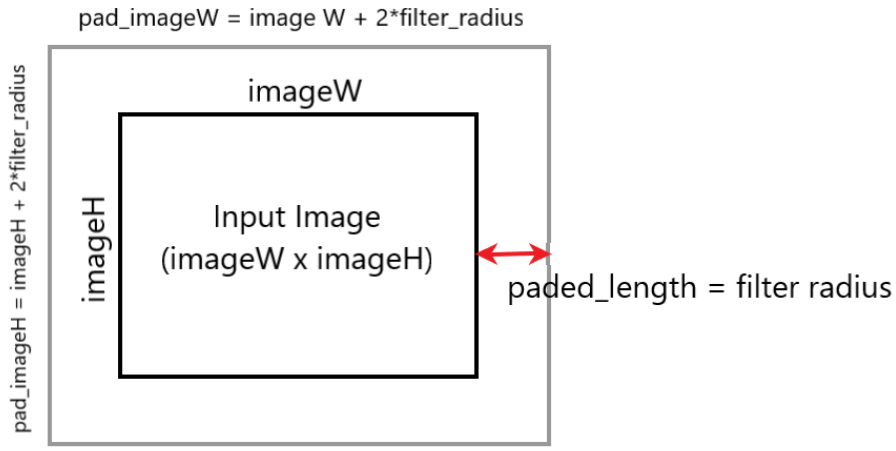
Supports MultiDevice Co-op Kernel Launch: No
Device PCI Domain ID / Bus ID / location ID: 0 / 6 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

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

2 Βελτιστοποίηση κώδικα προηγούμενης εργασίας

Με αφορμή τον υπολογισμό του τελικού πίνακα στην GPU δίχως control flow divergence, στον κώδικα της προηγούμενης εργασίας εφαρμόστηκε περιμετρικά padding στην εικόνα με μέγεθος pad = filter_radius. Ο padded πίνακας απεικονίζεται παρακάτω.



Ξεκινήσαμε τις βελτιστοποιήσεις αλλάζοντας τον κώδικα του kernel που εφαρμόζει συνέλιξη κατά γραμμές. Ο κώδικας που συντάχθηκε στην προηγούμενη εργασία απεικονίζεται παρακάτω:

```
////////////////////////////////////////////////////////////////////////////////
// Reference row convolution filter
////////////////////////////////////////////////////////////////////////////////

__global__ void convolutionRowGPU(double *d_Dst, double *d_Src, double *d_Filter,
                      int imageW, int imageH, int pad_imageW, int filterR) {

    int k;

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

    double sum = 0;
    for (k = -filterR; k <= filterR; k++) {
        int d = x + k;

        sum += d_Src[y * pad_imageW + d] * d_Filter[filterR - k];
    }

    d_Dst[(y+filterR)*pad_imageW + x + filterR] = sum;
}
```

Αρχικά σκεφτήκαμε να αλλάξουμε τον τρόπο ανάγνωσης των πινάκων d_Src και d_Filter καθώς μετά την αρχικοποίηση τους αποθηκεύονται στην global memory της GPU, η οποία είναι uncached off-chip μνήμη με αποτέλεσμα κάθε ανάγνωση στοιχείου απο τους πίνακες να είναι πολύ ακριβή.

Επειδή το διαχωρίσιμο φίλτρο (d_Filter) έχει σταθερές τιμές μετά την αρχικοποίηση του και τα στοιχεία του χρησιμοποιούνται και απο τους δυο kernel σε μεγάλο βαθμό, αποθηκεύτηκε στην constant memory της GPU η οποία είναι **cached** off-chip memory, μειώνοντας το penalty κάθε ανάγνωσης. Όσον αφορά τον πίνακα d_Src (ο οποίος ορίζεται στην συνάρτηση με διεύθυνση &d_Input[filter_radius * pad_imageW + filter_radius], δείχνοντας με αυτον τον τρόπο στο πρώτο στοιχείο του πίνακα με δεδομένα της εικόνας) παρατηρούμε απο τον παραπάνω κώδικα οτι κάθε block προσπελαύνει συνολικά **(2*filter_radius + blockDim.x) * blockDim.y** του πίνακα d_Src τα οποία του ανατίθονται απο τον διαμερσμό του πίνακα στο grid της GPU. Για να μειώσουμε τις ακριβές αναγνώσεις απο την global memory διαχωρίσαμε τα δεδομένα του πίνακα σε υποσύνολα τα οποία να χωράνε στην shared memory του εκάστοτε block, η οποία είναι on-chip memory και πολύ πιο γρήγορη απο την global memory. Ο πίνακας της shared memory (s_data) αρχικοποιείται δυναμικά πριν την εκτέλεση του kernel και έχει μέγεθος **(2*filter_radius + blockDim.x) * blockDim.y** . Γνωρίζουμε η εκτέλεση του block στον streaming multiprocessor γίνεται σε wraps, καθένα απο τα οποία αποτελείται απο 32 threads του block. Το hardware επιλέγει κάθε φορά ένα wrap προς εκτέλεση. Επειδή οι αναγνώσεις του πίνακα d_Src γίνονται κατα μήκος ενός wrap, αρχικοποιήσαμε τον πίνακα s_data με τέτοιο τρόπο ώστε τα threads του κάθε wrap να κάνουν prefetch στοιχεία προς ανάγνωση του πίνακα d_Src για τα threads με το ίδιο threadIdx.x που ανήκουν στο γειτονικό wrap. Με αυτον τον τρόπο πραγματοποιούμε περισσότερες αναγνώσεις στην shared memory, οι οποίες είναι πολύ πιο «φθηνές» συγκριτικά με τις αναγνώσεις απο την global memory. Ο βελτιστοποιημένος κώδικας είναι ο εξής:

```
////////////////////////////////////////////////////////////////////////////////
// Reference row convolution filter
////////////////////////////////////////////////////////////////////////////////

__global__ void convolutionRowGPU(double *d_Dst, double *d_Src, int pad_imageW, int filterR) {

    int k;

      int x =  blockIdx.x*blockDim.x + threadIdx.x +filterR;
      int y =  blockIdx.y*blockDim.y +threadIdx.y + filterR;
      int idx = y *(pad_imageW) + x;
      int x_Dim = 2*filterR + blockDim.x;

      extern __shared__ double s_data[];

      for(int k = 0; k + threadIdx.x < x_Dim ; k+= blockDim.x){
              s_data[threadIdx.y*x_Dim + threadIdx.x + k] = d_Src[idx - filterR + k];
      }

      __syncthreads();

    double sum = 0;

    for (k = -filterR; k <= filterR; k++) {
        sum += s_data[threadIdx.y*x_Dim + threadIdx.x + filterR + k] * d_Filter[filterR - k];
    }

    d_Dst[idx] = sum;
}
```

Ακολουθήσαμε την ίδια λογικη και στην συνέλιξη κατα στήλες, αλλάζοντας όμως την αρχικοποίηση των δεδομένων στην shared memory με τρόπο ώστε να είναι αποδοτικό το caching των δεδομένων της εικόνας καθώς προσπέλαση των δεδομένων της εικόνας στον συγκεκριμένο κώδικα πραγματοποιείται κατά στήλες.

Ο πίνακας της shared memory έχει μέγεθος **(2*filter_radius + blockDim.y) * blockDim.x** και αρχικοποιείται δυναμικά πρίν την εκτέλεση του kernel. Παρακάτω απεικονίζονται οι κώδικες της προηγούμενης και της βελτιστοποιημένης έκδοσης του κώδικα.

```
////////////////////////////////////////////////////////////////////////////////
// Reference column convolution filter
////////////////////////////////////////////////////////////////////////////////

__global__ void convolutionColumnGPU(double *d_Dst, double *d_Src, double *d_Filter,
                                     int imageW, int imageH, int pad_imageH, int filterR) {

    int k;

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

    double sum = 0;

    for (k = -filterR; k <= filterR; k++) {
        int d = y + k;

        sum += d_Src[d * pad_imageH + x + filterR] * d_Filter[filterR - k];
    }

    d_Dst[y * imageW + x] = sum;
}
```

Αρχική έκδοση convolutionColumnGPU

```
////////////////////////////////////////////////////////////////////////////////
// Reference column convolution filter
////////////////////////////////////////////////////////////////////////////////

__global__ void convolutionColumnGPU(double *d_Dst, double *d_Src, int pad_imageH, int filterR) {

    int k;

    int x =  blockIdx.x*blockDim.x + threadIdx.x + filterR;
    int y =  blockIdx.y*blockDim.y +threadIdx.y + filterR;
    int idx = y*pad_imageH + x;
    int y_Dim = 2*filterR + blockDim.y;
    int dst_idx = (y-filterR)*(pad_imageH - 2*filterR) + (x-filterR);

    extern __shared__ double s_data[];

    for(int k = 0; k + threadIdx.y < y_Dim; k+= blockDim.y ){
        s_data[threadIdx.y * blockDim.x + k*blockDim.x + threadIdx.x] = d_Src[idx + (k - filterR)*pad_imageH];
    }

    __syncthreads();

    double sum = 0;

    for (k = -filterR; k <= filterR; k++) {
        sum += s_data[(threadIdx.y * blockDim.x) + (filterR + k)*blockDim.x + threadIdx.x] * d_Filter[filterR - k];
    }

    d_Dst[dst_idx] = sum;
}
```

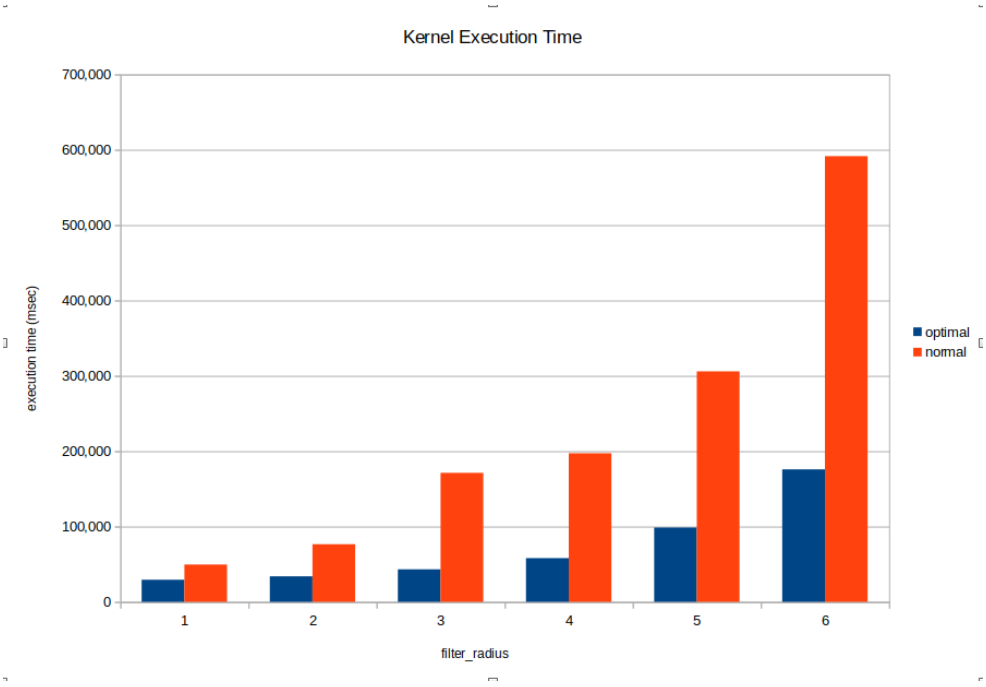
Βελτιστοποιημένη έκδοση convolutionColumnGPU

Τέλος έγινε αλλαγή του τρόπου δέσμευσης του πίνακα h_OutputGPU στην μεριά του Host απο malloc σε cudaHostAlloc. Με αυτον τον τρόπο η εν λόγω μνήμη στον host γίνεται pinned με αποτέλεσμα το device να πραγματοποιεί εγγραφές/αναγνώσεις πολυ πιο γρήγορα σε σχέση με την έκδοση οπου γίνεται χρήση malloc.

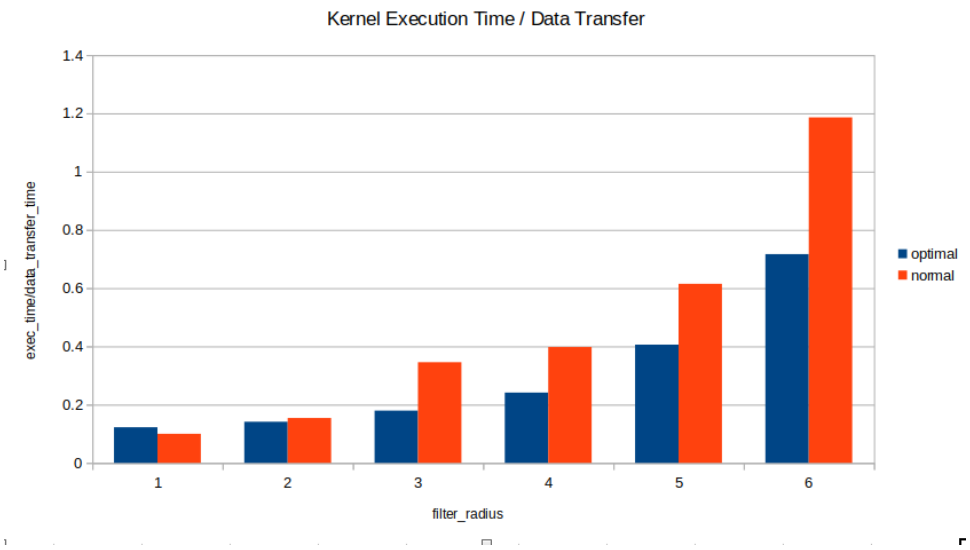
3 Πειραματισμός με διαφορετικά μεγέθη φίλτρων

Οι μετρήσεις για τα επιθυμητά μεγέθη πραγματοποιήθηκαν με την εντολή **nvprof ./<executable filename>** .

οι αναφερόμενες τιμές προέκυψαν απο διαδιχοκή εκτέλεση της παραπάνω εντολής (12 φορές) και εξάγοντας την μέγιστη και την ελάχιστη τιμή υπολογίζεται ο μέσος όρος τους.



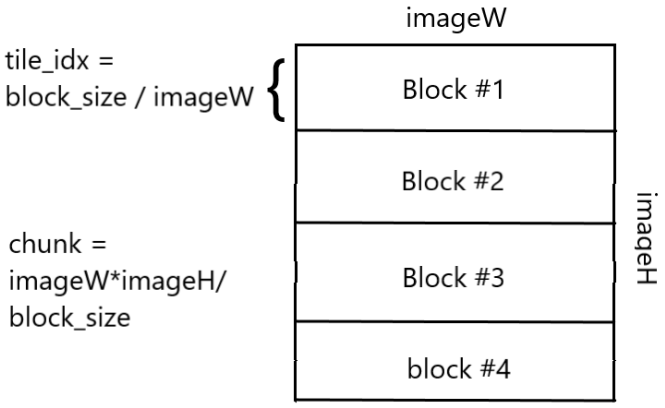
Χρόνος εκτέλεσης των kernels



Σχέση χρόνου εκτέλεσης kernels / μεταφοράς δεδομένων

4 Υποστήριξη μεγάλων εικόνων πάνω στην GPU

Για να καταφέρουμε να υποστηρίξουμε πολύ μεγαλύτερες εικόνες πάνω στην GPU, χωρίζουμε την εικόνα κατα γραμμές παράγωντας blocks τα οποία εκτελούνται διαδοχικά στους kernel ώστε όλα τα δεδομένα να χωρέσουν στην GPU. Για την δημιουργία των block ζητείται απο το πρόγραμμα το επιθυμητό block size (se bytes) το οποίο θα πρέπει να είναι δύναμη του 2 και μεγαλύτερο η ίσο της μίας διάστασης της εικόνας (δηλαδή block_size/imagew >= 1, το οποίο καθορίζει το πλήθος γραμμών της εικόνας που καταλαμβάνει το κάθε block). Το πλήθος των διαδοχικών κλήσεων των kernel καθορίζεται απο την μεταβλητή **chunk = imageW*imageH / block_size**. Οι παραπάνω μεταβλητές απεικονίζονται στο παρακάτω παράδειγμα



Όσον αφορά την γεωμετρία των block, για την μέγιστη αξιοποίηση τους υλοποιήσαμε την παρακάτω γεωμετρία:

```
if(imageW > MAX_X_DIM){
  block.x = MAX_X_DIM;
  grid.x = imageW/block.x;

  if(block_size/imageW > MAX_X_DIM){
    block.y = MAX_Y_DIM;
    grid.y = (block_size/imageW)/MAX_Y_DIM;
  }

  else{
    block.y = block_size/imageW;
    grid.y = 1;
  }

  tile_idx = block_size/imageW;
}

else{
  grid =1;
  block.x = imageW;
  block.y = imageH;
  chunk = 1;
  tile_idx = imageH;
}
```

Αρχικά χωρίζουμε την εικόνα σε imageW/32 blocks κατα την οριζόντια διάσταση του grid. Αν το πλήθος των γραμμών που καταλαμβάνει το κάθε block (tile_idx = block_size/imageW) ξεπερνά την μέγιστη κατακόρυφη διάσταση του block (32), τότε ορίζουμε μέγιστη κατακόρυφη διάσταση στο κάθε block και κατακόρυφα στο grid ορίζουμε tile_idx/32 blocks. Αν το πλήθος των γραμμών που καταλαμβάνει το κάθε block είναι μικρότερο του 32, τότε το κάθε block θα έχει διάσταση ίση με το πλήθος των γραμμών του πίνακα που καταάμβάνει το κάθε block όπως και το grid θα είναι μονοδιάστατο (grid.y = 1). Τέλος, σε περίπτωση που το μέγεθος της εικόνας είναι μικρότερο ή ίσο με το max πλήθος των threads ανα block, ο υπολογισμός πραγματοποιείται σε μία επανάληψη χωρίς tiling.

Η διαδικασία υπολογισμού των kernel σε block εικόνας απεικονίζεται παρακάτω:

```
pos = 0;

for(i=0; i < chunk; i++){
  checkCudaErrors(cudaMemcpy(d_Input, h_temp + pos, (pad_imageW)*(tile_idx+2*filter_radius)*sizeof(double), cudaMemcpyHostToDevice));

  convolutionRowGPU<<<grid,block,(block.x + 2*filter_radius)*block.y*sizeof(double)>>>(d_OutputGPU,d_Input,pad_imageW, filter_radius);

  checkCudaErrors(cudaMemcpy(h_temp2 + pad_imageW*filter_radius + pos, d_OutputGPU + pad_imageW*filter_radius, pad_imageW * tile_idx * sizeof(double), cudaMemcpyDeviceToHost));

  pos += pad_imageW * tile_idx;
}

cudaDeviceSynchronize();

pos = 0;
int final_pos = 0;

for(i=0; i < chunk; i++){
  checkCudaErrors(cudaMemcpy(d_Input, h_temp2 + pos, (pad_imageW)*(tile_idx+2*filter_radius)*sizeof(double), cudaMemcpyHostToDevice));

  convolutionColumnGPU<<<grid,block,(block.y + 2*filter_radius)*block.x*sizeof(double)>>>(d_OutputGPU,d_Input,pad_imageH,filter_radius);

  checkCudaErrors(cudaMemcpy(h_OutputGPU + final_pos, d_OutputGPU, imageW * tile_idx *sizeof(double), cudaMemcpyDeviceToHost));

  pos += pad_imageW * tile_idx;
  final_pos += imageW * tile_idx;
}
```

Αρχικά επειδή παρατηρήσαμε εξαρτήσεις δεδομένων μεταξυ των block της εικόνας ανάμεσα στις δυο κλήσεις των kernel , υπολογίζουμε διαδοχικά τα blocks για την συνέλιξη κατά γραμμές, συνενώνουμε τα κομμάτια παράγωντας μια ενιαία εικόνα και επαναλαμβάνουμε την διαδικασία για την συνέλιξη κατα στήλες (και στις δυο περιπτώσεις chunk φορές). Ο πίνακας εισόδου d_Input , έχει διαστάσεις ίσες με τις διαστάσεις του κάθε block. Ο δείκτης pos δείχνει ποιό τμήμα της εικόνας πρέπει να δεσμευτεί απο τον πίνακα h_temp (αρχική εικόνα) και σε ποιό σημείο να γίνει η εγγραφή των αποτελεσμάτων (h_temp2). Στην δεύτερη διαμέριση, παίρνουμε τα δεδομένα απο τον h_temp2 απο την θέση που δείχνει ο δείκτης pos, υπολογίζουμε την συνέλιξη κατα στήλες και τέλος γίνεται η εγγραφή των αποτελεσμάτων στο h_OutputGPU στην θέση που δείχνει ο δείκτης final_pos ο οποίος ανανεώνεται διαφορετικά απο τον δείκτη pos καθώς ο τελικός πίνακας δεν είναι padded.

Τέλος όπως παρατηρήσαμε και απο τον profiler οτι δέν υπάρχει κάποια επικάλυψη ανάμεσα στον κώδικα που εκτελείται απο διαφορετικούς kernel ή σε εκτέλεση κώδικα και μεταφορά δεδομένων. Αυτο είναι λογικό καθώς σε αυτο το σημείο της υλοποίησης δεν χρησιμοποιούμε κάποια ασύγχρονη δομή (streams).

5 Επικάλυψη με χρήση streams

Για να υπάρξει επικάλυψη στον μεγαλύτερο δυνατό βαθμό μεταξύ της εκτέλεσης kernel και μεταφοράς δεδομένων ανάμεσα σε Host και Device , δημιουργήθηκαν δυο streams (stream0, stream1) , το καθένα απο τα οποία εκτελεί διαφορετικό σύνολο δεδομένων, ασύγχρονα σε κάθε kernel και η μεταφορά δεδομένων μετατράπηκε σε ασύγχρονη (cudaMemcpy -> cudaMemcpyAsync).

Η υλοποίηση με streams παρουσιάζεται παρακάτω:

```
pos = 0;

for(i=0; (i < (chunk/2) ) || (chunk == 1); i++){
  checkCudaErrors(cudaMemcpyAsync(d_Input0, h_temp + pos,
                 (pad_imageW)*(tile_idx+2*filter_radius)*sizeof(double),
                 cudaMemcpyHostToDevice, stream0));
  if(chunk != 1)
    checkCudaErrors(cudaMemcpyAsync(d_Input1, h_temp + pos + (pad_imageW * tile_idx) ,
                   (pad_imageW)*(tile_idx+2*filter_radius)*sizeof(double),
                   cudaMemcpyHostToDevice, stream1));

  convolutionRowGPU<<<grid,block,(block.x + 2*filter_radius)*block.y*sizeof(double),stream0>>>(d_OutputGPU0,
                                                                    d_Input0,pad_imageW, filter_radius);

  if(chunk != 1)
    convolutionRowGPU<<<grid,block,(block.x + 2*filter_radius)*block.y*sizeof(double),stream1>>>(d_OutputGPU1,
                                                                    d_Input1,pad_imageW, filter_radius);

  checkCudaErrors(cudaMemcpyAsync(h_temp2 + pad_imageW*filter_radius + pos,
                                d_OutputGPU0 + pad_imageW*filter_radius,
                                pad_imageW * tile_idx * sizeof(double),
                                cudaMemcpyDeviceToHost,stream0));
  if(chunk == 1)
    break;

  checkCudaErrors(cudaMemcpyAsync(h_temp2 + pad_imageW*filter_radius + pos + (pad_imageW * tile_idx) ,
                 d_OutputGPU1 + pad_imageW*filter_radius, pad_imageW * tile_idx * sizeof(double),
                 cudaMemcpyDeviceToHost,stream1));

  pos += 2*(pad_imageW * tile_idx);
}
```

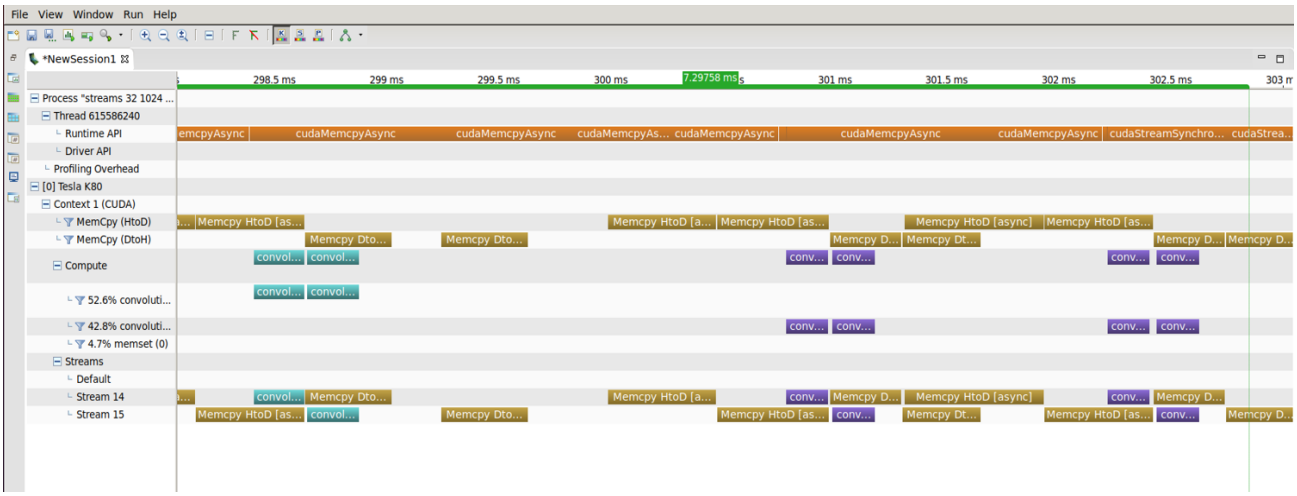
convolutionRowGPU kernel execution with streams

```
cudaDeviceSynchronize();

pos = 0;
int final_pos = 0;

for(i=0; (i < (chunk/2) ) || (chunk == 1); i++){
  checkCudaErrors(cudaMemcpyAsync(d_Input0, h_temp2 + pos,
                 (pad_imageW)*(tile_idx+2*filter_radius)*sizeof(double),
                 cudaMemcpyHostToDevice, stream0));

  if(chunk != 1)
    checkCudaErrors(cudaMemcpyAsync(d_Input1, h_temp2 + pos + (pad_imageW * tile_idx),
                   (pad_imageW)*(tile_idx+2*filter_radius)*sizeof(double),
                   cudaMemcpyHostToDevice, stream1));

  convolutionColumnGPU<<<grid,block,(block.y + 2*filter_radius)*block.x*sizeof(double),stream0>>>(d_OutputGPU0,
                                                                    d_Input0,pad_imageH,filter_radius);

  if(chunk != 1)
    convolutionColumnGPU<<<grid,block,(block.y + 2*filter_radius)*block.x*sizeof(double),stream1>>>(d_OutputGPU1,
                                                                      d_Input1,pad_imageH,filter_radius);


  checkCudaErrors(cudaMemcpyAsync(h_OutputGPU + final_pos,
                 d_OutputGPU0, imageW * tile_idx *sizeof(double),
                 cudaMemcpyDeviceToHost,stream0));
  if(chunk == 1)
    break;

  checkCudaErrors(cudaMemcpyAsync(h_OutputGPU + final_pos + (imageW * tile_idx) ,
                  d_OutputGPU1, imageW * tile_idx *sizeof(double),
                  cudaMemcpyDeviceToHost,stream1));

  pos += 2*(pad_imageW * tile_idx);
  final_pos += 2*(imageW * tile_idx);
}

cudaStreamSynchronize(stream0);
cudaStreamSynchronize(stream1);
```

convolutionColumnGPU kernel execution with streams

Όπως φαίνεται κάθε stream δεσμεύει και εκτελεί διαφορετικά block εικόνας , προσαρμόζοντας τον δείκτη pos κατάλληλα για το κάθε stream, μειώνοντας με αυτον τον τρόπο τα for loop iterations στο μισό. Για την περίπτωση στην οποία χρειάζεται μία μόνο εκτέλεση του κάθε kernel (chunk = 1, tile_idx = imageH), γίνεται χρήση ενός μόνο stream.

Παρακάτω παρουσιάζουμε το timeline απο το report του nvvp profiler, για filter_radius = 32, imageW = 1024 και block_size = 4096. Παρατηρούμε ότι υπάρχει επικάλυψη ανάμεσα στις κλήσεις των kernel και στις μεταφορές δεδομένων μεταξύ Host και Device.



6 Περιορισμός μεγέθους εικόνας

Οι εικόνες που μπορούμε να υποστηρίξουμε είναι μέχει 32768x32768. Στον αρχείο /proc/memInfo εντοπίσαμε τον διαθέσιμο χώρο που διαθέτει ο host στο csl-artemis ο οποίος είναι 129366220 kB. Παρατηρήσαμε ότι ο κώδικας χρησιμοποιεί τον μέγιστο χώρο , χωρίς να έχουμε ενεργοποιήσει το κομμάτι εκτέλεσης της CPU , με μέγεθος εικόνας 65536x65536 (χωρις να λαμβάνουμε υπόψην το padding_size και οτι το κάθε στοιχείο είναι sizeof(double)). Προσπαθήσαμε να πραγματοποιήσουμε το πείραμα με τον εν λόγω μέγεθος εξοικονομώντας κάποιες δεσμεύσεις πινάκων αλλα δεν καταφέραμε να εξοικονομήσουμε αρκετή μνήμη στην μερία του Host (αποτυγχάνει η δέσμευση του πίνακα h_OutputGPU). Έπειτα απο αυτη την μελέτη είναι ξεκάθαρο ότι ο περιορισμός προέρχεται απο την διαθέσιμη μνήμη στην μεριά του Host.