Παράλληλα και Διανεμημένα Συστήματα

Υλοποίηση του αλγορίθμου Non Local Means για την αποθορυβοποίηση εικόνας και βελτίωσής του με τη χρήση CUDA.

Τμήμα Ηλεκτρολόγων Μηχανικών και ΜηχανικώνΥπολογιστών Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Όνομα: Παναγιώτης

Επώνυμο: Κασπαρίδης

ΑΕΜ: 8157

Περιεχόμενα

1.Εισαγωγή

Η παρούσα εργασία εξετάζει την υλοποίηση του αλγορίθμου με την χρήση της CUDA. Η CUDA είναι μια πλατφόρμα που μας επιτρέπει να εκτελούμε κώδικα στην GPU, σχεδιασμένη από την NVIDIA. Προυπόθεση για να μπορεί κανείς να εκτελέσει το αρχείο του, θα πρέπει να έχει κάρτα γραφικών της NVIDIA συμβατή με κάποια έκδοση της CUDA.

Ο κώδικας που είχαμε να κατασκευάσουμε και να βελτιώσουμε πετυχαίνει την αποθοριβοποίηση μιας εικόνας μέσω του υπολογισμού του μέσου όρου όλων των pixels της εικόνα, σταθμισμένο με το βαθμό ομοιότητας με το pixel αναφοράς. Μαθηματικά εκφρασμένος ο αλγόριθμος υπάρχει στο pdf της εκφώνησης της εργασίας.

Χρησιμοποιώ το “f^” εννοώντας την τιμή του pixel ως αποτέλεσμα της εφαρμογής του αλγορίθμου.

1.1 Αρχεία της CUDA και συναρτήσεις διαχείρησής τους από την Matlab

Υλοποίηση με GLOBAL Memory → nlmGlobal.cu & Matlab: → NlmeansGlb.m

1η Υλοποίηση με Shared Memory → Shared.cu & Matlab: → SharedKernel.m

2η Υλοποίηση με Shared Memory → SharedWithMatlab.cu & Matlab: → SharedWithMatlab.m

2.Τεχνική ανάλυση

2.1 Αποδόμηση αλγορίθμου

Για να είναι δυνατή η παραλληλοποίηση του αλγορίθμου θα πρέπει να δούμε ποια κομμάτια μπορούν να ανεξαρτητοποιηθούν.

- Υπολογισμός της παραμέτρου Z : Η γνώση της παραμέτρου είναι αναγκαία για τον υπολογισμό των βαρών. Υπολογίζεται από τον τύπο

$$Z(i) = \sum_j e^{-\frac{\|f(\mathcal{N}_i) - f(\mathcal{N}_j)\|^2_{G(a)}}{\sigma^2}}$$

  Να διευκρινήσω ότι i,j είναι pixels και στους κώδικές μου παρουσιάζονται με συντεταγμένες.

- Υπολογισμός της f^ : Εφόσον έχουμε υπολογίσει όλα τα Z, η f^(x) είναι ο υπολογισμός του

$$\hat{f}(\mathbf{x}) = \sum_{\mathbf{y}\in\Omega} w(\mathbf{x},\mathbf{y}) f(\mathbf{y}), \qquad \forall \mathbf{x} \in \Omega, \qquad w(i,j) = \frac{1}{Z(i)} e^{-\frac{\|f(\mathcal{N}_i) - f(\mathcal{N}_j)\|^2_{G(a)}}{\sigma^2}},$$

  οπότε για κάθε pixel αναφοράς(x) το Z(x) είναι γνωστό .
- Τα x,y,i,j είναι pixel και στους κώδικες τα θεωρώ σαν διανύσματα με συντεταγμένες.

2.2 Επιλογη των kernels

Χρησημοποίησα δυο kernel για την υλοποίηση του αλγορίθμου.

Το πρώτο είναι για των υπολογισμό του πίνακα Z και το δεύτερο για τον τελικό υπολογισμό του πίνακα f^,δηλαδή τις τιμές της αποθορυβοποιημένης εικόνας.

Για την καλύτερη παρουσίαση του kernel θα παραθέσω και επεξηγήσω τον κώδικα από την υλοποίησή μου χωρίς την χρήση shared memory και έπειτα θα περάσω στις αλλαγές που χρειάστηκαν τα kernel για να χρησιμοποιήσω την shared memory.

2.3 Υλοποίηση με χρήση της Global memory

Η υλοποίηση αυτή έχει όνομα nlmGlobal.cu

Υλοποίηση του kernel Z:

```
__global__ void Zcalc(float const * const A, float *Z,float const * const H,int patchSize,float patchSigma,float fltSigma, int m, int n) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  if(x<m-(2*patchSize-1)/2 && y<n-(2*patchSize-1)/2){

    int i,j,k,l,counter=0;
    float FNij=0.0;
    float temp=0.0;
    patchSize=(patchSize-1)/2;
    for(i=patchSize;i<m-patchSize;i++){
        for(j=patchSize;j<n-patchSize;j++){
            for(k=-patchSize;k<=patchSize;k++){
                        for(l=-patchSize;l<=patchSize;l++){
                                temp=(f(x+patchSize+k,y+patchSize+l)-f(i+k,j+l))*H[counter];
                                temp=temp*temp;
          FNij=FNij+(temp);
          counter++;
                         }
                       }
                Z(x+patchSize,y+patchSize)=Z(x+patchSize,y+patchSize)+expf(-(FNij/(fltSigma)));
                FNij=0.0;
                counter=0;
              }
    }
  }
}
```

Τα threads είναι ίσα με τον αριθμό των pixels της εικόνας. Κάθε thread υπολογίζει το Z για το συγκεκριμένο pixel. Οι διαστάσεις m,n είναι οι επεκταμένες διαστάσεις του πίνακα της εικόνας στην οποία έχει προστεθεί το απαραίτητο padding, ώστε όταν ένα pixel είναι στα περιθώρια των πλευρών του πίνακα η γειτονιά του να έχει τιμή. Το γέμισμα των extra κομματιών του πίνακα γίνεται στην matlab και δεν χρειάζεται περαιτέρω επεξήγηση.Οι δυο for που οι επαναλήψεις τους

είναι ίσες με όλα τα pixels της εικόνας, έχουν τον ρόλο του αθροιστή και κάθε φορά προσδιορίζουν ένα pixel της εικόνας και υπολογιζουν την συνεισφορά της γειτονιάς του στον υπολογισμό της Z του pixel αναφοράς.

Μέσα υπολογίζεται η ευκλείδια απόσταση μεταξύ της γειτονιάς του pixel αναφοράς και μιας άλλης γειτονιάς pixel, οι τιμές των οποίων είναι σταθμησμένες με τις τιμές από ένα gaussian patch.Και κάθε φορά αθροίζεται στην τιμή της Z.

Η συνάρτηση expf είναι συνάρτηση ειδικά υλοποιήσιμη για να εκμεταλεύεται την αρχιτεκτονική της gpu.

Υλοποίηση του kernel f^:

```
__global__ void fCalc(float const * const A,float const * const Z,float const * const H, float *f_,int patchSize,float patchSigma,float
fltSigma, int m, int n){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  if(x<m-(2*patchSize-1)/2 && y<n-(2*patchSize-1)/2){
              int i,j,k,l,counter=0;
    patchSize=(patchSize-1)/2;
    float FNij=0.0;
    float temp=0.0;
    float Z_local=Z(x+patchSize,y+patchSize);
    for(i=patchSize;i<m-patchSize;i++){
        for(j=patchSize;j<n-patchSize;j++){
                        for(k=-patchSize;k<=patchSize;k++){
                          for(l=-patchSize;l<=patchSize;l++){
                                  temp=(f(x+patchSize+k,y+patchSize+l)-f(i+k,j+l))*H[counter];
                                  temp=temp*temp;
          FNij=FNij+(temp);
          counter++;
                           }
                         }
                f_(x+patchSize,y+patchSize)=f_(x+patchSize,y+patchSize)+((1/Z_local)*expf(-(FNij/(fltSigma))))*f(i,j);
                FNij=0.0;
                counter=0;
              }
    }
  }
}
```

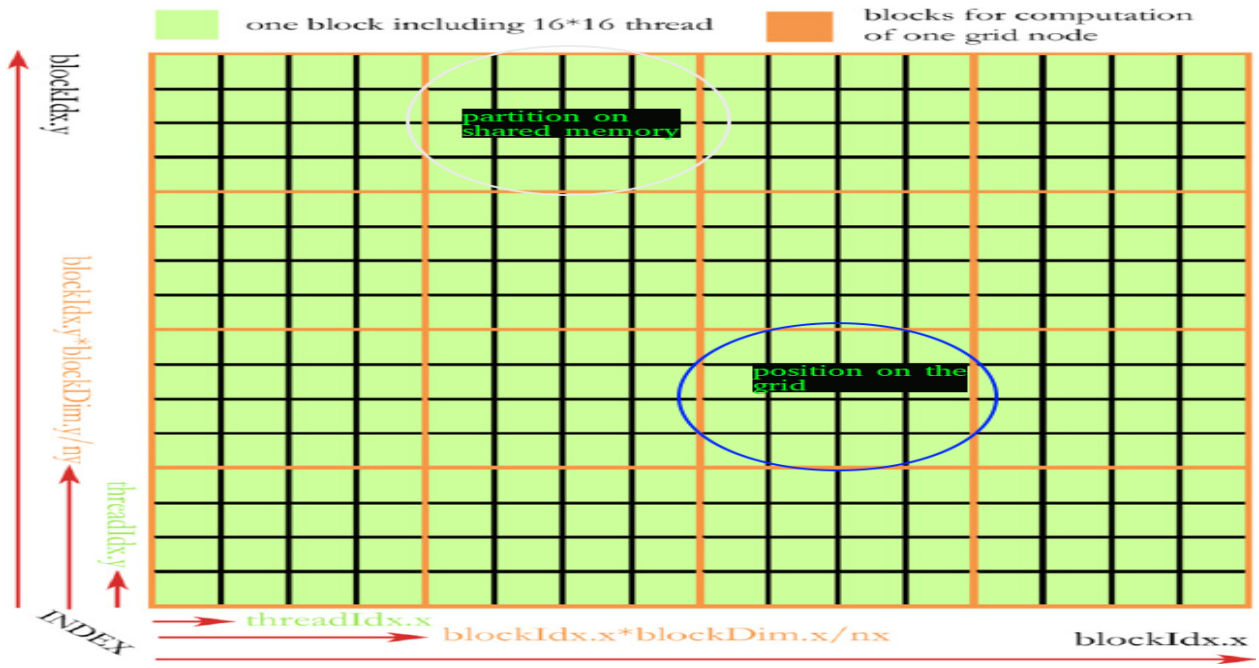
Η υλοποίηση αυτού του kernel είναι όμοια με του Z. Φαίνεται εδώ ότι για τον υπολογισμό του αποθορυβοποιημένου pixel χρειάζεται μόνο μια τιμή του Z για το pixel αυτό γι’ αυτό και την περνάω σε μια local μεταβλητή για να μήν κάνω πολλές προσπελάσεις στην global memory που είναι αργή. Η υλοποίηση είναι η μετάφραση του μαθηματικού τύπου υπολογισμού του w(x,y) και τελικά της f^. Παραπάνω έχει εξηγηθεί ο ρόλος της διπλής for για επαναλήψεις m*n (όπου m,n οι κανονικές διαστάσεις του πίνακα).

2.4 Βασική υλοποίηση με χρήση Shared Memory

Η υλοποίηση αυτή έχει όνομα Shared.cu

Τα προβλήματα στην υλοποίηση αυτή οφείλονται στο γεγονός ότι η shared memory είναι πολύ μικρή συγκριτικά με τον μέγεθος του πίνακα της εικόνας και θα πρέπει να φορτώνουμε σε αυτήν κομμάτια του πίνακα.Βέβαια, λόγο της ευκλείδιας αποστασης που υπολογίζεται στην γειτονιά των pixels, ενώ εμείς έχουμε κάποιο αριθμό threads, για παράδειγμα ας υποθέσουμε 32x32 threads και έστω η γειτονιά μας είναι ένας πίνακας 5x5 τότε θα χρειαστούμε ένα κομμάτι του πίνακα 36x36 για να κάνουμε χρήση όλων των thread στον υπολογισμό.Γενικά θα χρειαστεί να φορτώσουμε στην shared memory ένα κομμάτι του πίνακα μεγέθους threadsPerBlock+patchSize-1,όπου patchSize είναι το πχ 5x5.

2.4.1 Τρόπος μεταφοράς των δεδομένων:



Η παραπάνω εικόνα δείχνει τις θέσεις των shared πινάκων, ωστόσο οι πινακες δεν περιέχουν ακριβώς τον ίδιο αριθμό στοιχειών με threads, αλλα περισσότερα στοιχεία ανάλογα με το padding και χρειάζεται η αποθήκευση τους.

Έστω ότι έχουμε ένα μπλόκ με αριθμό threadsPerBlock, για να υπολογίσω τον Z και την f^ τελικά χρειάζομαι να μεταφέρω από την global memory 2 πίνακες. Ο ένας θα περιέχει τις τιμές των pixel στην κανονική εικόνα, δηλαδή θα δίχνει στην global τιμή του pixel. Ο άλλος θα παίρνει κάθε φορά ένα partition του πίνακα και με αυτό θα κάνουμε τους υπολογισμούς μας, έπειτα θα αλλάζει partition μέχρι να πάρει όλες τις τιμές του πίνακα. Και οι δυο πίνακες να έχουν περισσότερα απο threadsPerBlock*threadsPerBlock στοιχεία, όπως είπαμε.Το γέμισμα των πινάκων γίνεται με tovds ίδιο τρόπο αλλάζουν, μόνο οι δείκτες στον πίνακα της εικόνας.

```
float *s_A=&Memory[0];//local block position in s_A
		float *g_A=&Memory[dimension*dimension];//block's
global position in g_A
		if(x<m-2*pad && y<n-2*pad){
		  z_local=Z(x+pad,y+pad);
		  g_A[x_local +y_local*dimension]=A[x+y*m];
		  if(x_local>blockDim.x-patchSize){
				g_A[(x_local+patchSize-1) +
y_local*dimension]=A[x+patchSize-1 +y*m];
		  }
		  if(y_local>blockDim.y-patchSize){
		  g_A[x_local + (y_local+patchSize-1)*dimension]=A[x +
(y+patchSize-1)*m];
		  }
		  if(x_local>blockDim.x-patchSize &&
y_local>blockDim.y-patchSize ){
				g_A[x_local+patchSize-1 +
(y_local+patchSize-1)*dimension]=A[x+patchSize-1 + (y+patchSize-
1)*m];
	}
		}
```

Με g_A συμβολίζω τον πίνακα που περιέχει τις σωστες τιμές των pixel(τα pixel αναφοράς). pad=patchSize-1/2 δηλαδή όσο η επέκταση του πίνακα από την μια μόνο πλευρά. Οι x_local και y_local περιέχουν το threadIdx.x και threadIdx.y αντίστοιχα, ενώ οι x,y την θέση του thread μέσα στο grid. Ο πίνακας A είναι ο πίνακας της εικόνας και είναι επεκταμένος. Δηλαδή αν έχουμε εικόνα 64x64 και patchSize=[5 5] τότε ο πίνακας αυτός θα είναι 68x68 και οι πίνακες στην shared memory θα είναι επεκταμένοι κατά patchSize-1.

Το γέμισμα του πίνακα g_A είναι απλό, περνάω τις πρώτες threadsPerBlock * threadsPerblock τιμές. Όμως επειδή δεν έχω γεμίσει τον πίνακα βάζω κάποια threads , αυτά φαίνονται στις if , να πάρουν κάποια extra τιμή η οποία βρίσκεται απλά από μια μετατόπιση μιας συντεταγμένης ή και των δύο κατα patchSize-1. Έτσι έχω πάρει ένα πίνακα επεκταμένο ανάλογα το patchSize.Όπως φαίνεται παραπάνω κώδικα ο g_A όπως και ο s_A δείχνουν στις διευθύνσεις του πίνακα Memory, ο οποίος είναι τύπου extern __shared__ float Memory[], και έχω ορίσει σαν μέγεθος της shared memory 16kb στην matlab.Οι pointers είναι απαραίτητοι, γιατί με την δήλωση του extern __shared__ οι πίνακες θα πρέπει να δείχνουν στα σωστά σημεία της shared memory.

Έπειτα χρειάζομαι ένα κομμάτι του πίνακα για να κάνω τις πράξεις του Z και μετά να φορτόσω άλλο υπολειπόμενο τμήμα του πίνακα. Αυτά τα τμήματα τα φορτώνω s_A με βήμα μετατόπισης να είναι threadsPerBlock στον x_local ή/και στο y_local. Για να περάσει μία φορά όλος ο πίνακας στην s_A χρειάζομαι μια διπλή for για την κατάλληλη μετακήνιση των συντεταγμένων των στοιχείων.

```
for(int xpos=0;xpos<(m-patchSize+1);xpos=xpos+blockDim.x){
for(int ypos=0;ypos<(n-patchSize+1);ypos=ypos+blockDim.y){
   __syncthreads();
		if(x<m-2*pad && y<n-2*pad){
			s_A[x_local +y_local*dimension]=A[x_local+xpos+(y_local+ypos)*m];
			if(x_local>blockDim.x-patchSize){
				s_A[(x_local+patchSize-1) + y_local*dimension]=A[x_local+xpos+patchSize-1 +(y_local+ypos)*m];
			}
			if(y_local>blockDim.y-patchSize){
				s_A[x_local + (y_local+patchSize-1)*dimension]=A[x_local +xpos + (y_local+ypos+patchSize-1)*m];
			}
			if(x_local>blockDim.x-patchSize && y_local>blockDim.y-patchSize ){
				s_A[x_local+patchSize-1 + (y_local+patchSize-1)*dimension]=A[x_local+xpos+patchSize-1 + (y_local+ypos+patchSize-1)*m];
			}
		}
		__syncthreads();
```

Οι 2 for τελειώνουν στο τέλος του προγράμματος αφού ο κύριος υπολογισμός του Z και f βασίζεται στα partitions, που φορτώνονται στην μεταβλητή s_A. Αξίζει να σημειωθεί ότι η μετακίνηση του block γίνεται για threadsPerBlock(πχ 32), ενώ οι διαστάσεις είναι μεγαλύτερες(πχ αν το patchSize=5 τότε έχουμε 36x36 πίνακα). Στην ουσία θεωρώ, ότι μετατοπίζονται τα threads στον μη επεκταμένο πίνακα, ενώ διαβάζουν τιμές από τον επεκταμένο. Για να έχω τις σωστές τιμές στους μετέπειτα υπολογισμούς μου, προσθέτω σε όλες τις συντεταγμένες το (patchSize-1)/2, πχ αν είναι 5 προσθέτω 2 και στο x_local και στο y_local ώστε να δείχνουν τις σωστές τιμές των pixels. Τα

“γύρω γύρω” pixels είναι υπεύθυνα για την επέκταση του πίνακα,που είναι απαραίτητη για τα pixels στα όρια των πλευρών. Μια αριθμητική επαλήθευση για έναν 128x128 πίνακα εικόνας και patchSize = [5 5] είναι: για τον άξονα x: 1η φορά παίρνω [0-36) 2η φορά παίρνω από το [32-68) 3η φορά παίρνω από το [64-100) 3η φορά παίρνω από [96-132). Όμοια είναι και στο y άξονα. Οι σωστές τιμές των pixel στην shared memory Θα βρίσκονταν στο διάστημα 2-34 με 2-34. Οι συγχρονισμοί είναι απαραίτητοι ώστε να μην διαβάσει κάποιο thread λάθος τιμή τόσο στην αρχή κάθε επανάληψης όσο και στο τέλος της if που φαίνεται στον κώδικα παραπάνω.

Με τον ίδιο ακριβώς τρόπο φορτώνονται οι τιμές στο kernel της f.

Το gaussian patch φορτώνεται σε έναν constant πίνακα. Τον έχω ορίσει μεγέθους 121 ώστε να δέχεται μελλοντικά μέχρι 11x11 patchSize. Οι τιμές μεταφέρονται μέσα από την matlab με την δήλωση `setConstantMemory(p,'s_H',H);` όπου s_H είναι το όνομα της μεταβλητής στο kernel της CUDA και H η μεταβλητή στην matlab που μεταβιβάζω.

2.4.2 Εκτέλεση του κορμού των kernels

Οι πυρήνες των kernel δεν διαφέρουν σημαντικά απ’ ότι στην υλοποίηση με global memory το μόνο που αλλάζει είναι οι μεταβλητές, που πλέον είναι η g_A στην θέση της f(x+patchSize+k,y+patchSize+l) με ακριβώς το ίδιο σκεπτικό, απλά τα x,y είναι οι συντεταγμένες του thread στο block που ανήκει. Στην θέση της f(i+k,j+l)) είναι η s_A που περιέχει κάποιο τμήμα του πίνακα. Γίνεται ένας μερικώς υπολογισμός της μεταβλητής Z ή f^ και έπειτα ο πίνακας s_A αλλάζει περιεχόμενο και προστίθεται ο υπολογισμός από το νέο partition στον υπάρχοντα υπολογισμό. Τμήματα του κώδικα φαίνονται παρακάτω:

```
for(int i=pad;i<dimension-pad;i++){
  for(int j=pad;j<dimension-pad;j++){
     for(int p=-pad;p<=pad;p++){
       for(int l=-pad;l<=pad;l++){
temp=(g_A[(x_local+pad +l)+(y_local+pad + p)*dimension]-s_A[(i+l) +
(j+p)*dimension])*s_H[counter];
FNij=FNij+temp*temp
counter++;
   }
}
z_local=z_local+expf(-(FNij/filtsigma));
FNij=0;
counter=0;
}
}
```

```
for(i=pad;i<dimension-pad;i++){
  for(j=pad;j<dimension-pad;j++){
    for(k=-pad;k<=pad;k++){
       for(l=-pad;l<=pad;l++){
temp=(g_A[(x_local+pad +l)+(y_local+pad + k)*dimension]-s_A[(i+l) +
(j+k)*dimension])*s_H[counter];
FNij=FNij+temp*temp;
counter++;
            }
         }
f_local=f_local+(1/Z_local)*(expf(-(FNij/filtsigma)))*s_A[i+j*dimension];
FNij=0;
counter=0;
}
}
```

Όπως φαίνεται, αποθηκεύω τις τιμές των Z και f^ σε local μεταβλητές για να εξοικονομήσω ταχύτητα και να μη κάνω προσπελάσεις στην μνήμη, όπως επίσης να μη δεσμεύω αχρείαστο χώρο στην shared memory. Δίνω τις τιμές στο τέλος του kernel έξω από τις 2 for. Επίσης τις αρχικοποιώ στο 0.

2.5 Παραλλαγή της προηγούμενης υλοποιήσης με Shared memory (χειρότερη από την 2.4)

Η υλοποίηση αυτή έχει όνομα SharedWithMatlab.cu

Σε αυτήν την υλοποίηση κάνω ακριβώς το ίδιο όπως και πριν με μια μόνο διαφορά. Τα kernels μου παίρνουν έναν extra πίνακα που περιέχει το partition που θα φορτωθεί στον πίνακα s_A. Το partiotion αυτό είναι δημιουργημένο στην matlab και περνιέται σαν όρισμα στα kernels. Ο λόγος που προσθέτω κι αυτήν την υλοποίηση είναι επειδή είχα διαφορές σε χρόνους και ποιότητα εικόνας και θεωρησα ότι αξίζει να υπάρχει. Ιδιαίτερη ανάλυση δεν χρειάζεται είναι πανομοιότυπος με την προηγούμενη υλοποίηση απλά πιο απλουστευμένος. Αντιγράφει 2 πίνακες στην shared memory, η μεταφορά τους είναι ίδια ακριβώς, απλά σε αυτήν την υλοποίηση δεν υπάρχουν οι δυο μεταβλητές xpos & ypos για την αλλαγή του partition. Την δουλειά αυτή την κάνω από την matlab.

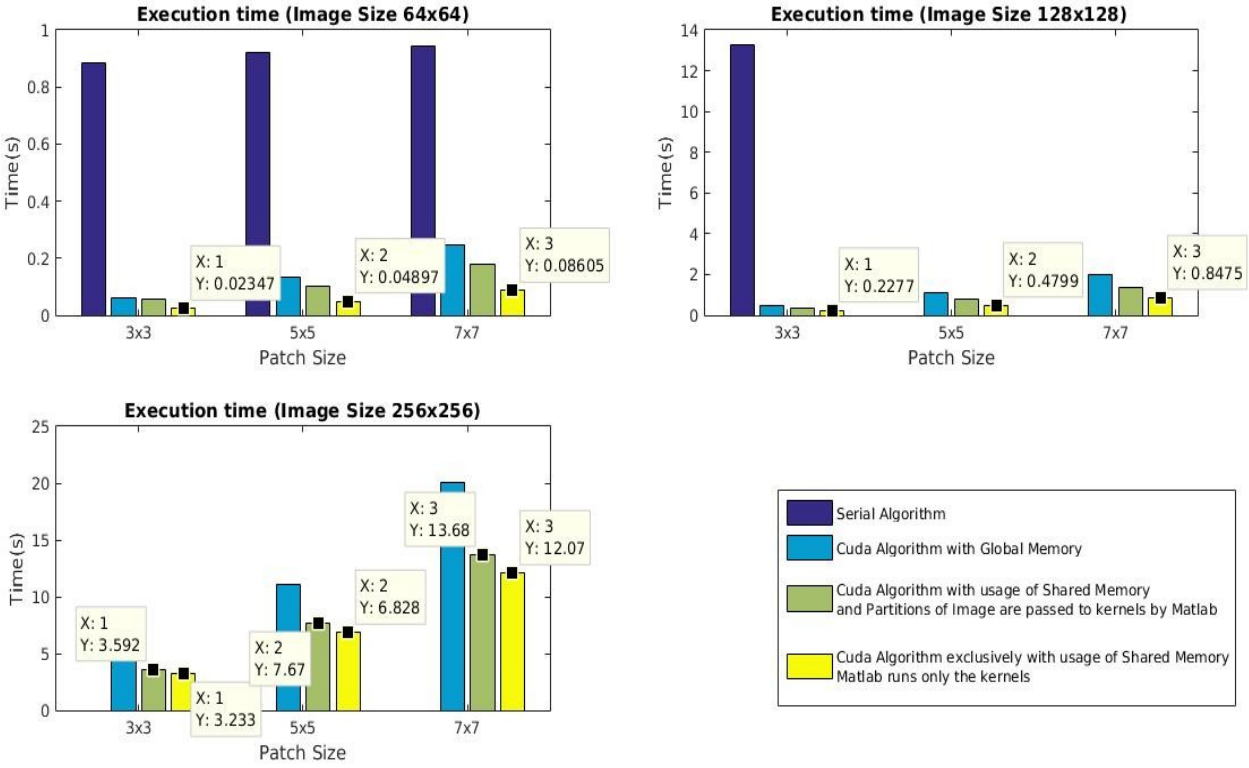
2.6 Matlab αρχεία

Για να τρέξω τα kernels χρησιμοποίησα την συναρτηση parallel.gpu.CUDAKernel . Στο αντικείμενο που δημιουργείται, ορίζω το μέγεθος της shared memory που θα χρησιμοποιήσω και μεταφέρω όπως είπα την constant μεταβλητή μου. Έπειτα κάνω επέκταση του πίνακα της εικόνας με την χρήση της εντολής `J=padarray(J,(patchSize-1)./2,'symmetric','both');` όπου J είναι ο πίνακας της εικόνας και patchSize η γειτονιά των pixel. Με την συναρτηση feval καλώ τα kernels και με την gather μαζεύω πίσω από την gpu τα δεδομένα που θέλω . Όλοι οι πίνακες που στέλνω είναι τύπου gpuArray και όπως ζητήται είναι float. Υπολογίζω το PSNR σαν δείκτη ποιότητας της παραγόμενης εικόνας συγκρινόμενη με την αρχική.

3.Αποτελέσματα

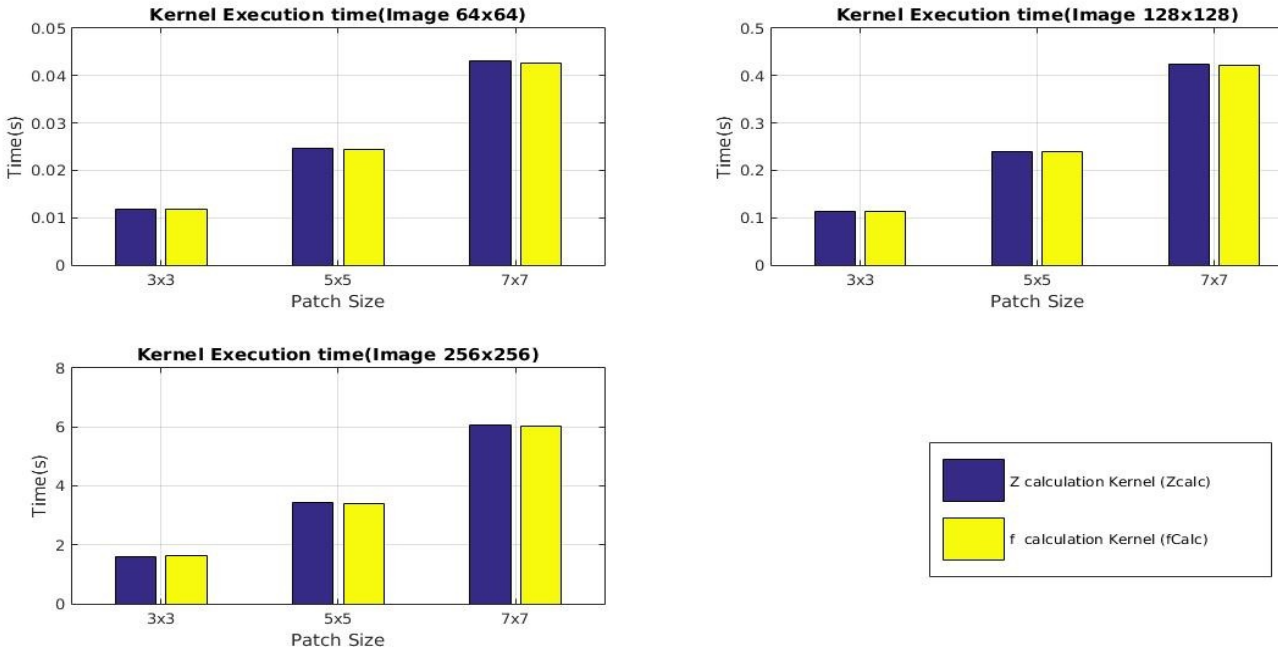
Τα αποτελέσματα που θα παρουσιάσω είναι από τον διάδη. Τεχνικά χαρακτηριστικά της GPU παρουσιάζονται στο τέλος. Τα νούμερα που έχω είναι για διαφορετικά patch Size. Κρατάω σταθερές τις παραμέτρους filtSigma=0.02 και patchSigma=5/3, οι οποίες επιρρεάζουν την ποιότητα της αποθορυβοποιημένης εικόνας και τίποτα άλλο.

3.1 Χρόνοι Εκτέλεσης



Οι συνολικοί χρόνοι εκτέλεσης παρουσιάζονται στο παραπάνω διάγραμμα. Ο σειριακός αλγόριθμος εκτελέστηκε για μέγεθος μέχρι 128x128 και για patch Size 3x3, όπου ο αλγόριθμος ,που είναι γραμμένος σε CUDA(κίτρινη ράβδος στα bar chart) σε σχέση με τον σειριακό, είναι 58 φορες ταχύτερος.

Παρακάτω παρουσιάζεται διαγράμμα με τους χρόνους εκτέλεσης των kernels της υλοποίησής μου. Παρουσιάζω διαγράμματα μόνο για την πιο γρήγορη υλοποίηση με shared memory (υλοποίηση με κίτρινο χρώμα στο πιο πάνω bar chart)



Παρατήρηση: Πρέπει να αναφέρω ότι η υλοποίηση που θεωρώ καλύτερη, επειδή είναι ταχύτερη και όπως θα δείξω παρακάτω έχει καλύτερη ποιότητα εικόνας (ομοιότητα με την αρχική), δεν έτρεχε στον διάδη για μπλοκ με threads 32x32. Η υλοποίηση αυτή και οι χρόνοι που παίρνω είναι με 16x16 threads per block ενώ οι άλλοι κώδικες της CUDA εκτελούνται με 32x32 threads per block, δηλαδή με το ¼ των threads ο κώδικαςς αυτός είναι πάλι ταχύτερος. Το error που μου εμφάνιζε είναι :

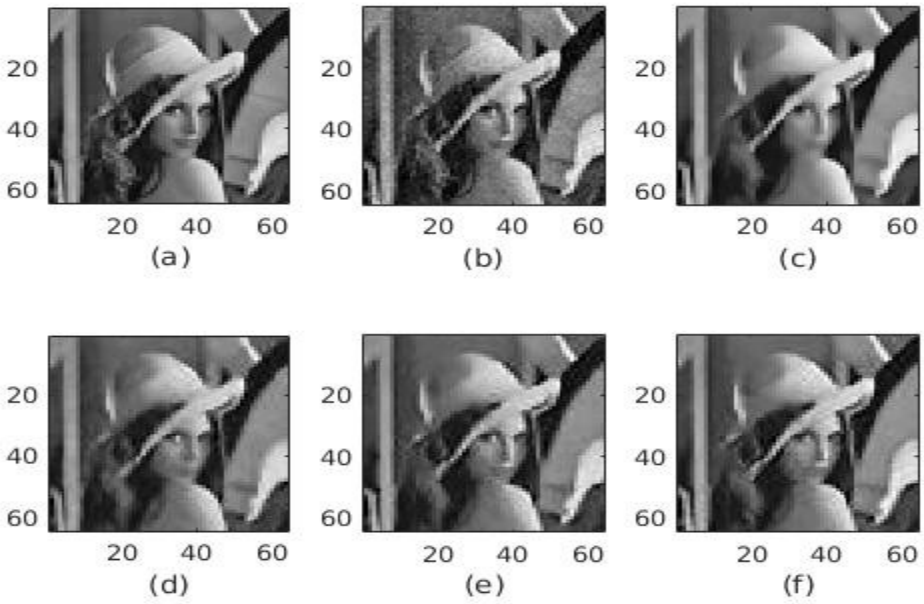
> nvalid size input to kernel ThreadBlockSize. You must provide a
>
> vector of up to 3 positive integers whose product is <= 896. The
>
> maximum value in each dimension is: [1024,1024,64].

η παράμετρος ThreadBlockSize δέχεται σαν είσοδο ένα διάνυσμα threadsPerBlock ίσο με [32 32]. Στην GPU του υπολογιστή μου δεν είχα αυτό το πρόβλημα. Τεχνικά χαρακτηρηστικά των 2 GPU θα παρουσιάσω στο τέλος.

3.2 Παρουσίαση αποτελεσμάτων & Αξιολόγηση των εικόνων

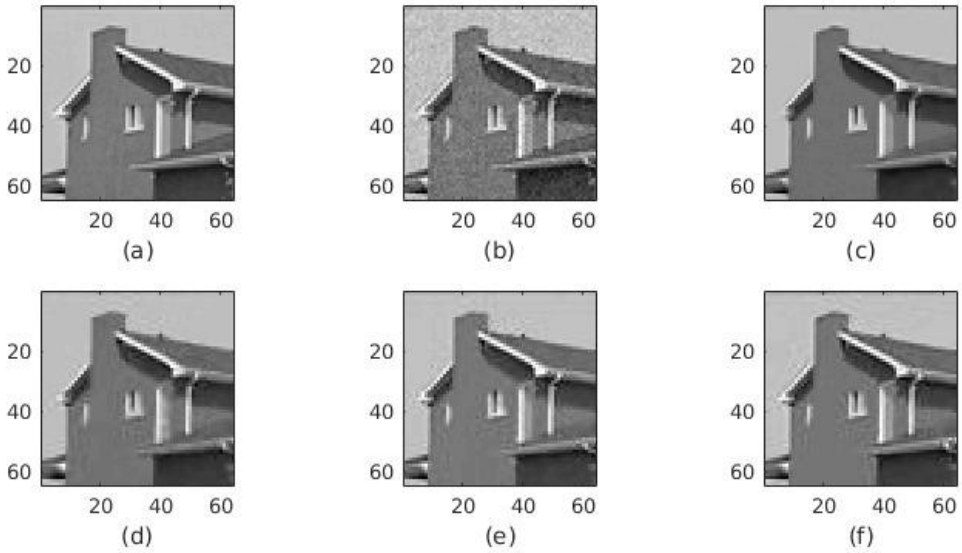
Για την αξιολόγηση της ποιότητας της εικόνας βασίστηκα στον δείκτη PSNR. Στην περίπτωση που 2 εικόνες είναι ίδιες το PSNR έχει άπειρη τιμή. Θέλω όσο το δυνατόν μεγαλύτερο PSNR.

Lena 64x64



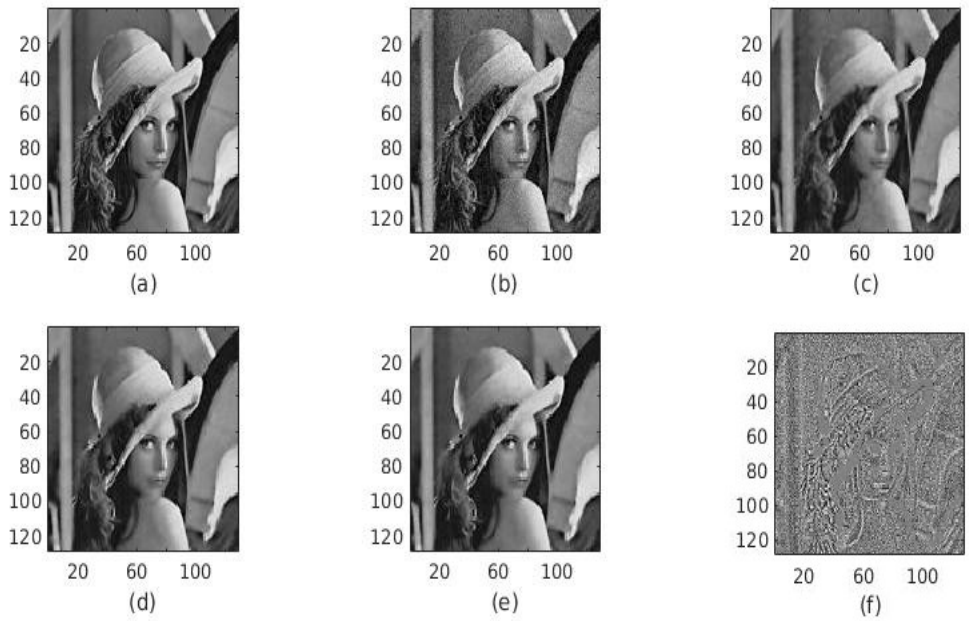
(a) Input Image, (b) Noisy Image, (c)Serial NLM algorithm(PatchSize=5x5) PSNR=30.1661, (d)CUDA NLM algorithm(PatchSize=3x3) PSNR=30.421829, (e)CUDA NLM Algorithm(PatchSize=5x5) PSNR=31.804831,(f) CUDA NLM Algorithm(PatchSize=7x7) PSNR=31.847513

House 64x64



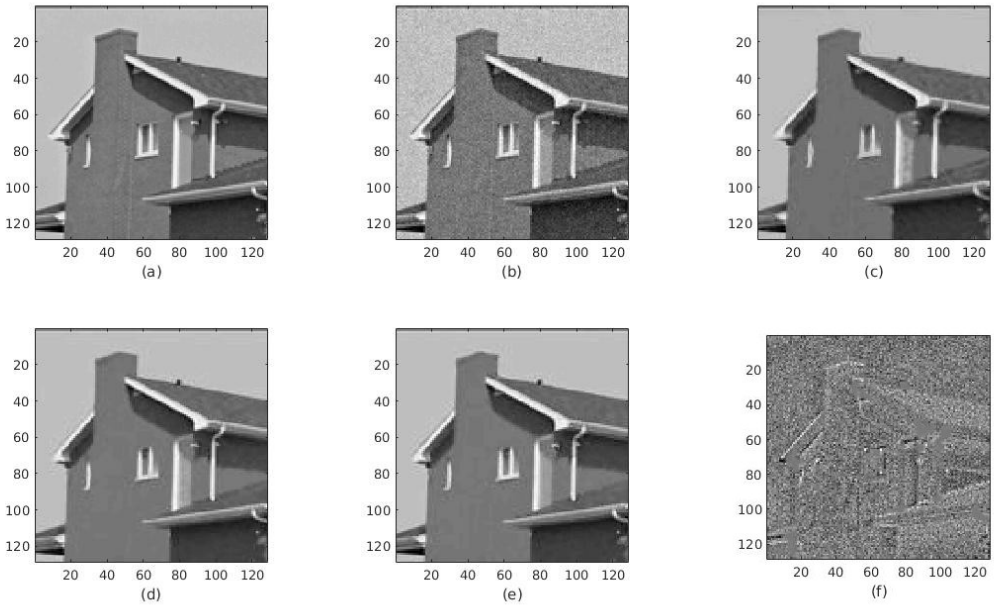
(a) Input Image, (b) Noisy Image, (c)Serial NLM algorithm(PatchSize=5x5) PSNR=30.8764, (d)CUDA NLM algorithm(PatchSize=3x3) PSNR=30.652372,(e)CUDA NLM Algorithm(PatchSize=5x5) PSNR=32.952957,(f) CUDA NLM Algorithm(PatchSize=7x7) PSNR=33.194031

Lena 128x128



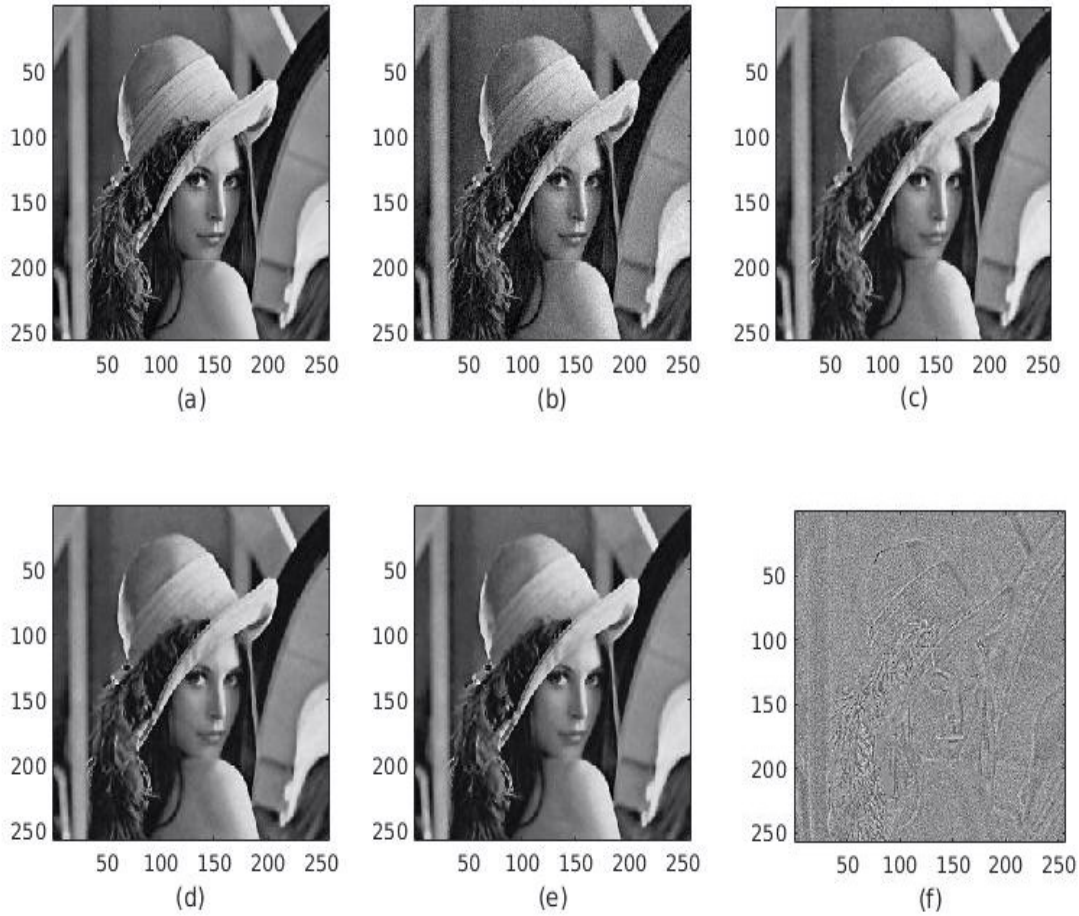
(a) Input Image, (b) Noisy Image, (c) NLM CUDA Image(PatchSize=3x3) PSNR=30.711241, (d) NLM CUDA Image(PatchSize=5x5) PSNR=31.587526, (e) NLM CUDA Image(PatchSize=7x7) PSNR=31.817949, (f) Residual Image (on PatchSize=7x7)

House 128x128



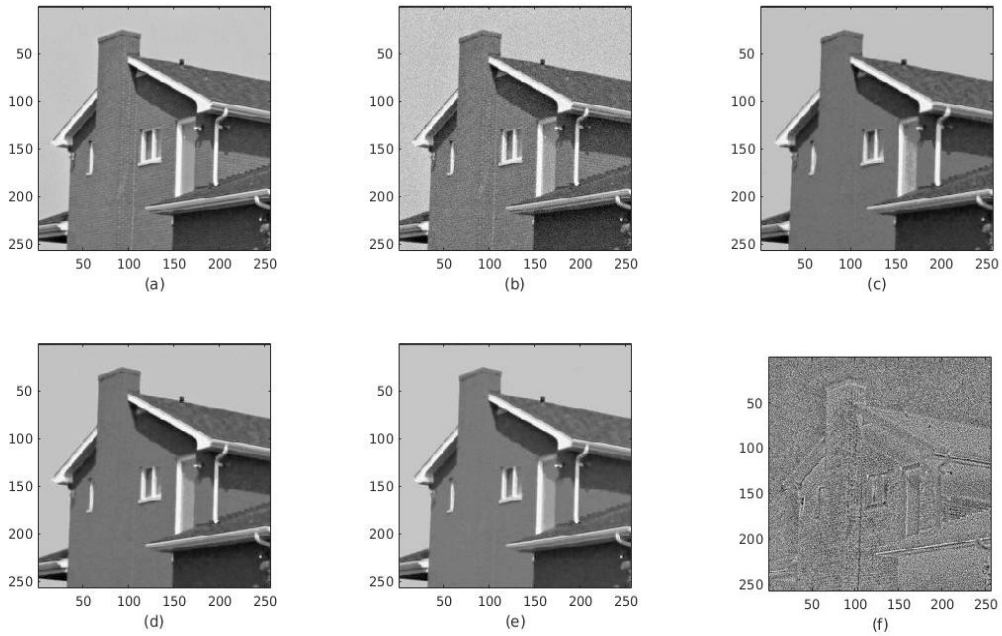
(a) Input Image, (b) Noisy Image, (c) NLM CUDA Image(PatchSize=3x3) PSNR=31.701256, (d) NLM CUDA Image(PatchSize=5x5) PSNR=33.813412, (f) NLM CUDA Image(PatchSize=7x7) PSNR=34.172672, (f) Residual Image (on PatchSize=7x7)

Lena 256x256



(a) Input Image, (b) Noisy Image, (c) CUDA NLM Image(PatchSIze=3x3) PSNR=31.285667, (d) CUDA NLM Image(PatchSIze=5x5) PSNR=31.578163 , (e) CUDA NLM Image(PatchSIze=7x7) PSNR=31.713551, (f) Residual Image (on PatchSize=7x7)

House 256x256



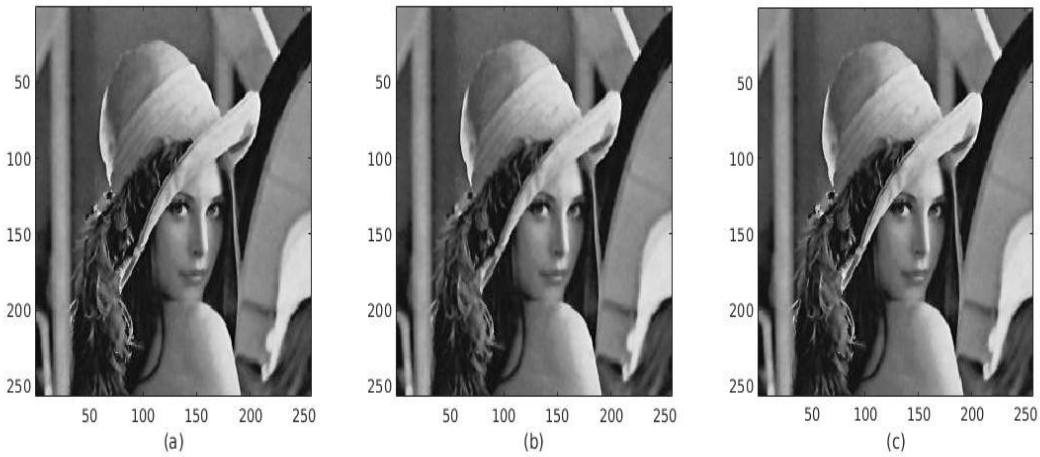
(a) Input Image, (b) Noisy Image, (c) CUDA NLM Image(PatchSIze=3x3) PSNR=32.481476, (d) CUDA NLM Image(PatchSIze=5x5) PSNR=33.339966 , (e) CUDA NLM Image(PatchSIze=7x7) PSNR=33.391075, (f) Residual Image (on PatchSize=7x7)

3.2.1 Σύγκριση υλοποίησεων

Βάσει του PSNR η παράλληλη υλοποιήση σε CUDA έχει υψηλότερο PSNR και σαν ποιότητα την θεωρώ πιο καλή. Ωστόσο εικόνες του σειριακού αλγορίθμου για 64x64 οπτικά μου φαίνονται πιο ωραία στο μάτι.

Ο λόγος που δεν επέλεξα τις άλλες υλοποιήσεις της CUDA, εκτός του μεγαλύτερου χρόνου εκτέλεσης, είναι και ο δείκτης PSNR, οι άλλες δύο υλοποιήσεις είχαν μικρότερη τιμή.

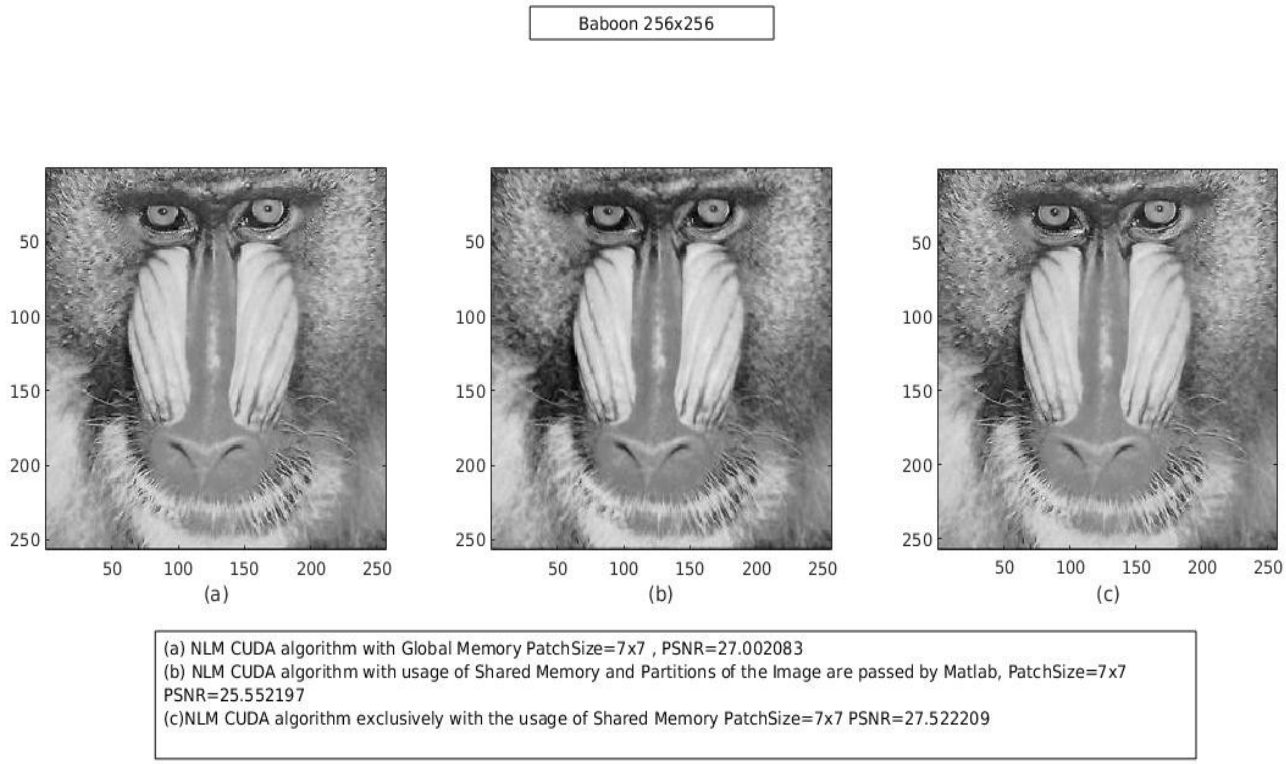
Παρακάτω έχω μια εικόνα από κάθε μέθοδο υλοποίησης και την τιμή PSNR. Δείχνω μόνο για εικόνα 256x256, τα αποτελέσματα των μετρήσεων μου υπάρχουν στο φάκελο (matlab/resultsTimePsnr).



(a) NLM CUDA algorithm with Global Memory PatchSize=7x7 , PSNR=29.331062

(b) NLM CUDA algorithm with usage of Shared Memory and Partitions of the Image are passed by Matlab, PatchSize=7x7 PSNR=30.325363

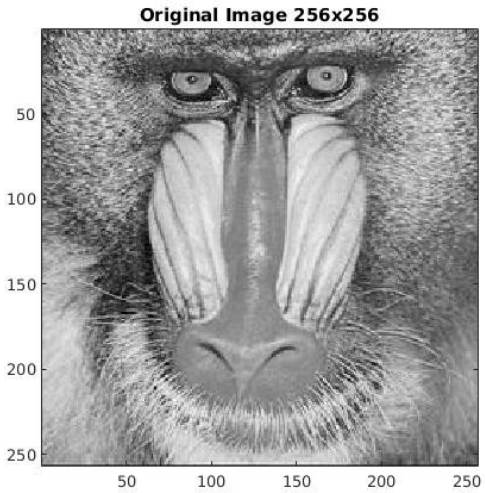
(c)NLM CUDA algorithm exclusively with the usage of Shared Memory PatchSize=7x7 PSNR=31.713551



Όπως φαίνεται, την καλύτερη ποιότητα εικόνας έχει ο κώδικας που είναι αποκλειστικά με την χρήση της Shared memory και περιέχει τις 2 for για την αλλαγή του partition στον πίνακα s_A.

Στην τελευταία εικόνα επειδή υπάρχουν πολλές λεπτομέρειες, ο δείκτης PSNR είναι χαμηλότερος.

Η κανονική εικόνα είναι



Παρατηρήσεις : Είναι αξιόλογο ότι μία μικρή αλλάγη στον κώδικα δίνει αλλαγή στους χρόνους εκτέλεσης και την ποιότητα της εικόνας. Οι δύο κώδικες που χρησιμοποιούν την shared memory είναι ίδιοι στο σκεπτικό και στην υλοποίηση. Στον έναν επέλεξα να δίνω τμήματα της εικόνας από την matlab και να τα περνάω στα kernels και στην άλλη υλοποίηση έκανα αυτήν την δουλεία μέσα στον κώδικα της CUDA.Ο δεύτερος τρόπος φαίνεται να έχει καλύτερους χρόνους ακόμη και με λιγότερα threads,(έτρεχε σε μπλοκ των 16x16) και καλύτερη ποιότητα εικόνας. Φαίνεται πως πρέπει να ελαχιστοποιούνται οι μετακινήσεις δεδομένων μεταξύ shared και global memory όπως

επίσης οι μετακινήσεις δεδομένων μεταξύ global με την μνήμη του υπολογιστή. Εκτός από καθυστέρηση στο χρόνο που έχει, φαίνεται να επιρρεάζει και την ακρίβεια των αριθμών(μεγαλώνει το σφάλμα).

Στην κάρτα γραφικών στον προσωπικό μου υπολογιστή, πιο γρήγορη ήταν η υλοποίηση που τα partitions φορτόνονταν από την matlab (χωρίς τις for στα kernels δηλαδή). Συγκριτικά θα αναφέρω ότι για 128x128 εικόνα με patchSize=[7 7], η υλοποίηση με την αλλαγή των partitions μέσα στα kernels είχε χρόνο εκτέλεσης 2,615421sec ενώ η άλλη που τα partitions τα περνάει η matlab είχε 2,043284 sec. Τα νούμερα αυτά τα θεωρώ αμφισβητήσιμα γιατί το μηχάνημα είναι παλιό και προβληματικό. Ίσως να μην έχω τόσο καλή gpu όσο ο διάδης. Ωστόσο ο δείκτης PSNR εξακολουθεί να είναι μεγαλύτερος στην υλοποίηση με τα partitions να εναλλάσσονται μέσα στα kernels.

4. Τεχνικά χαρακτηριστικά GPU

Διάδης :

NVIDIA GeForce GTX 480

GPU ENGINE SPECS:	
CUDA Cores	480
Graphics Clock (MHz)	700
Processor Clock (MHz)	1401
Texture Fill Rate (billion/sec)	42
MEMORY SPECS:	
Memory Clock (MHz)	1848
Standard Memory Config	1536 MB GDDR5
Memory Interface Width	384-bit
Memory Bandwidth (GB/sec)	177.4

Του Προσωπικού μου υπολογιστή: (Είναι χειρότερη από του Διάδη.)

Nvidia GeForce GT 640M

GPU Specifications

Chipset	NVIDIA
GPU	GK107 N13P-LP
Fabrication Process	28nm
CUDA Cores	384 Cores
Texture Fill Rate	16 GT/s

Memory Specifications

Memory Clock	783.0 MHz
Memory Type	GDDR 5
Memory Bandwidth	64 GB/sec
Memory Interface Width	128-bit

Βιβλιογραφία

1. https://en.wikipedia.org/wiki/Non-local_means
2. Antoni Buades, Bartomeu Coll, and J-M Morel. A non-local algorithm for image denoising. In 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), volume 2, pages 60–65. IEEE, 2005.
3. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio
4. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4XCCz9Awy