ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

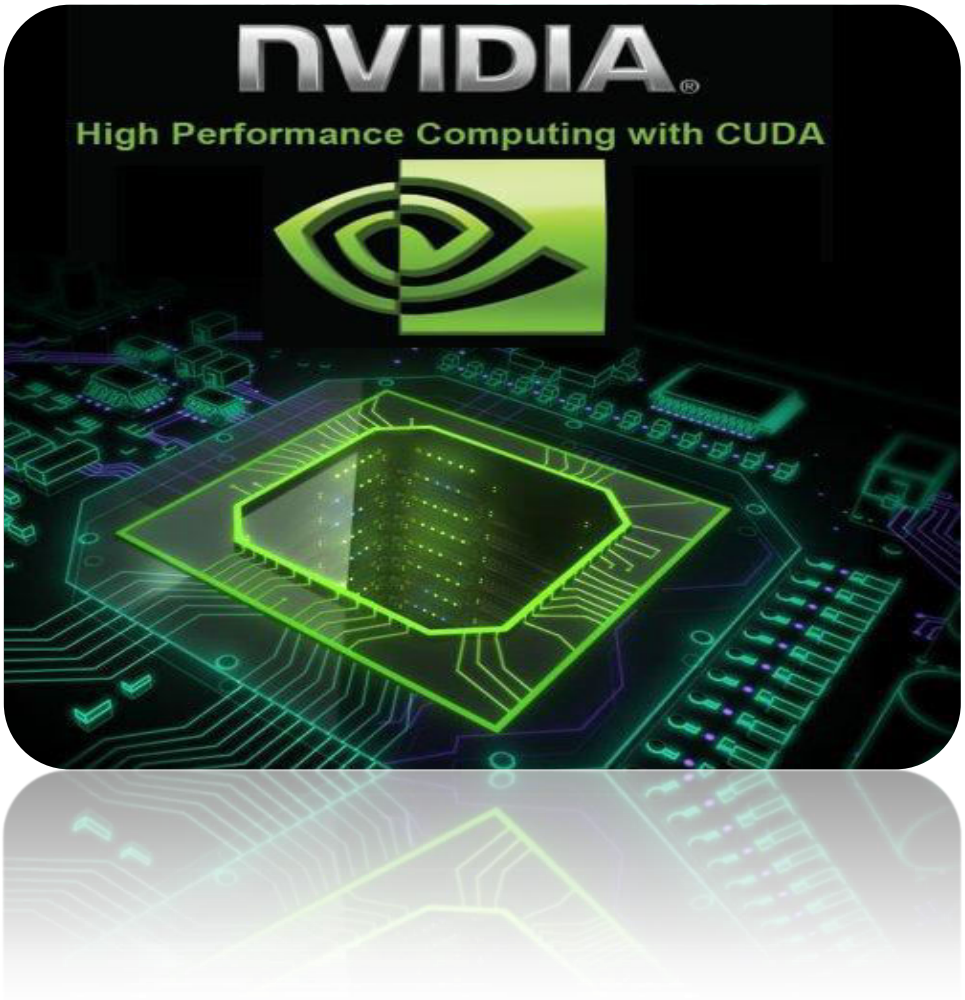
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΙΑ Γ

ΠΑΡΑΛΛΗΛΑ ΚΑΙ ΔΙΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

CUDA PROGRAMMING



ΦΟΙΤΗΤΗΣ: ΚΑΜΠΑΣ ΠΡΟΔΡΟΜΟΣ

ΑΕΜ: 8151

ΕΙΣΑΓΩΓΗ

Η εργασία αυτή αποτελεί την τρίτη κατά σειρά εργασία που δόθηκε στους φοιτητές προς υλοποίηση στα πλαίσια του μαθήματος «Παράλληλα και Διανεμημένα Συστήματα». Η εργασία ασχολείται με τον παράλληλο προγραμματισμό σε κάρτες γραφικών NVIDIA με χρήση του περιβάλλοντος CUDA. Οι τεράστιες δυνατότητες των GPU στην παράλληλη επεξεργασία δεδομένων αποτελεί έναν από τους καλύτερους και ταχύτερους τρόπους επεξεργασίας εικόνων. Αυτό είναι και το θέμα της εργασίας, η επεξεργασία δηλαδή εικόνων στις οποίες έχει εισαχθεί κάποιος Gaussian θόρυβος με τη χρήση του αλγορίθμου "**non local means**", ώστε η εικόνα που θα προκύψει στο πέρας της επεξεργασίας να είναι ίδια, ή σχεδόν ίδια με την αρχική.

Για το σκοπό αυτό χρησιμοποιήθηκαν 3 εικόνες διαφορετικής ανάλυσης, 64x64, 128x128 και 256x256 με περιοχές ανάλυσης αντίστοιχα 3x3, 5x5, 7x7. Πέρα από την ανάλυση του κώδικα που συντάχθηκε για τις ανάγκες της εργασίας παρουσιάζονται επίσης όλες οι εικόνες που υπέστησαν επεξεργασία, τα σφάλματα σε σχέση με την αρχική εικόνα καθώς και οι χρόνοι που λήφθηκαν από το «ΔΙΑΔΗΣ».

ΖΗΤΟΥΜΕΝΑ ΤΗΣ ΕΡΓΑΣΙΑΣ

Local Means για την αποθορυβοποίηση εικόνας. Στόχος είναι η βελτίωση της απόδοσής του με τη χρήση CUDA. Σε αντίθεση με φίλτρα *local mean*, που υπολογίζουν την μέση τιμή σε μία γειτονιά κάθε pixel για να εξομαλύνουν την εικόνα, **ο *Non Local Means*** υπολογίζει τον μέσο όρο όλων των pixels στην εικόνα, σταθμισμένο με το βαθμό ομοιότητας με το pixel αναφοράς. Το αποτέλεσμα είναι καλύτερη ευκρίνεια και διατήρηση των λεπτομερειών της αρχικής εικόνας.

Ο αλγόριθμος βασίζεται στην εύρεση παρόμοιων γειτονιών σε όλη την εικόνα και στον υπολογισμό της αποθορυβοποιημένης τιμής ως εξής:

$$\hat{f}(\mathbf{x}) = \sum_{y \in \Omega} w(x, y) f(y) \qquad \forall\ \mathbf{x} \in \Omega$$

όπου Ω _ R2 το πεδίο ορισμού της εικόνας, $f : \Omega \rightarrow R$ η θορυβώδης εικόνα και $\hat{f}(x): \Omega \rightarrow R$ η προσέγγιση της αποθορυβοποιημένης εικόνας.

Ο πίνακας βαρών $w(i; j)$ ορίζεται από την σχέση:

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{||f(Ni)-f(Nj)||^2 G(a)}{\sigma^2}}$$

$$Z(i) = \sum_j e^{-\frac{||f(Ni)-f(Nj)||^2 G(a)}{\sigma^2}}$$

όπου ως Nk ορίζεται μία τετράγωνη γειτονιά σταθερού μεγέθους με κέντρο το pixel k.

Χρησιμοποιώντας τον κώδικα που δίνεται σε MATLAB για επαλήθευση, το πρόγραμμά σας θα πρέπει να:
– Υλοποιεί τον υπολογισμό του $\hat{f}$ με χρήση δικού σας CUDA kernel, για τύπο δεδομένων float (η εικόνα
παίρνει τιμές στο διάστημα [0 1]).

– Αξιοποιεί την shared memory για μείωση των αναγνώσεων από την global memory, ώστε να επιταχυνθείπεραιτέρω η υλοποίηση.

Παραδώστε:
– Αναφορά 3 – 4 σελίδων που να περιέχει:

a) Περιγραφή της μεθόδου παραλληλισμού που χρησιμοποιήσατε.

b) Σχεδίαση και περιγραφή τεχνητής εισόδου, για έλεγχο ορθότητας.

c) Σχόλια για την ταχύτητα υπολογισμών στο diades, για μεγέθη εικόνων 64x64, 128x128 , 256x256 και μεγέθη γειτονιών 3x3, 5x5, 7x7.

d) Σχόλια για τα αποτελέσματα της αποθορυβοποίησης σε εικόνες που επιλέξατε.
– Τον κώδικα του προγράμματός σας.

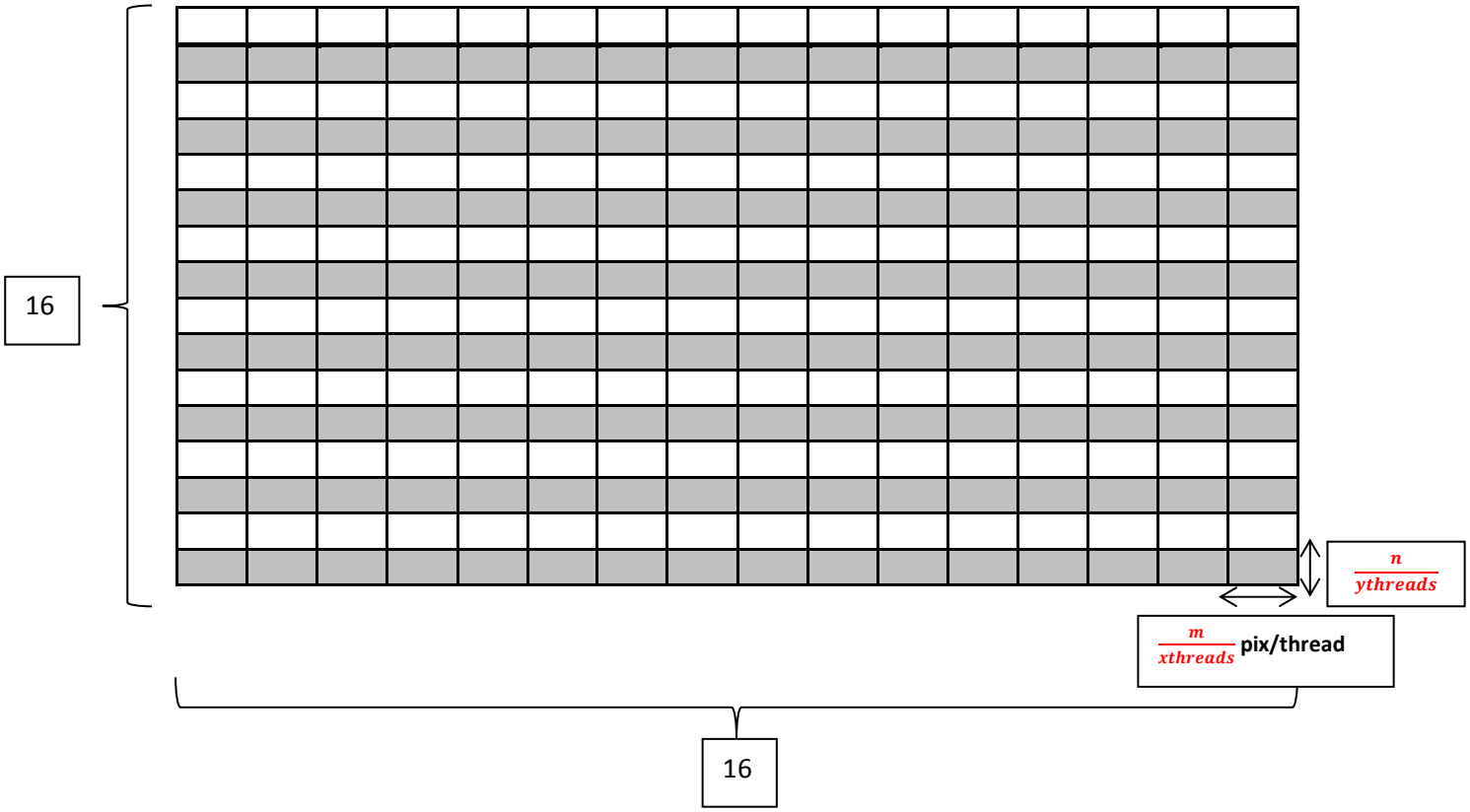
Δεοντολογία: Εάν χρησιμοποιήσετε κώδικες από το διαδίκτυο ή αλλού, να αναφέρετε την πηγή και τις αλλαγές που κάνατε.

Ανάλυση του κώδικα

Για την υλοποίηση της συγκεκριμένης εργασίας συντάχθηκαν 2 διαφορετικές εκδόσεις του CUDA kernel που ζητήθηκε. Για τον πρώτο αλγόριθμο, ο οποίος παρατίθεται μαζί με τον τελικό, θα δοθεί μια σύντομη ανάλυση καθώς δεν αποτελεί το βασικό αλγόριθμο που δημιουργήθηκε σε αυτή την εργασία. Αντιθέτως, για τον δεύτερο θα δοθεί μια πιο αναλυτική παρουσίαση.

Ο πρώτος αλγόριθμος, δεν κάνει καθόλου χρήση της shared memory, ούτε χρησιμοποιεί πολλά blocks. Δημιουργήθηκε μόνο ως ένα μέτρο σύγκρισης μεταξύ του αλγορίθμου που δόθηκε από τους διδάσκοντες σε MATLAB και του τελικού αλγορίθμου. Επειδή στο «ΔΙΑΔΗΣ» οι εικόνες 256x256 δεν μπορούν να υποστούν επεξεργασία εξαιτίας του μεγάλου όγκου μνήμης που απαιτείται, ο αλγόριθμος αυτός θα δώσει μια αίσθηση για την επιτάχυνση που προκύπτει από την εκμετάλλευση όσο το δυνατόν περισσότερο των δυνατοτήτων της κάρτας γραφικών GTX 480 που χρησιμοποιεί ο «ΔΙΑΔΗΣ».

Η λογική του αλγορίθμου είναι η εξής: γίνεται χρήση ενός block και 16x16 νημάτων στα οποία γίνεται η επεξεργασία όλων των pixels. Εάν **m** ο αριθμός των pixels στις γραμμές και **n** στις στήλες τότε κάθε νήμα είναι υπεύθυνο για την επεξεργασία $\frac{m*n}{xthreads*ythreads}$ pixels. Σε κάθε νήμα υπάρχουν 2 βρόχοι επανάληψης που υλοποιούν τον καταμερισμό των pixels στα επιμέρους νήματα και μέσα σε αυτούς 2 ακόμα βρόχοι οι οποίοι κάνουν την επεξεργασία καθενός pixel με όλα τα υπόλοιπα, δηλαδή επαναλήψεις 0→m, 0→n . Οι 4 αυτοί βρόχοι είναι που καθιστούν το πρόγραμμα αυτό πολύ αργό καθώς σε κάθε νήμα ανατίθεται τεράστιος επεξεργαστικός φόρτος. Μέσα σε αυτούς τους 4 βρόχους υπάρχουν άλλοι 2 για τον υπολογισμό των γειτόνων του καθενός (οι οποίοι δεν μπορούν να αποφευχθούν) και με αριθμό επαναλήψεων 0→patchSizeX , 0→patchSizeY.



Ο **βασικός** αλγόριθμος που τίθεται στους διδάσκοντες προς αξιολόγηση κάνει χρήση 16x16 blocks και 16x16 threads για την επιτάχυνση της επεξεργασίας της θορυβοποιημένης εικόνας.

Ο λόγος που χρησιμοποιήθηκαν οι αριθμοί αυτοί είναι διότι εξετάστηκαν οι περιπτώσεις όπου τα blocks=32 και threads=32 χωρίς καμία βελτίωση το ήδη υπάρχων μοντέλο. Κάθε block είναι υπεύθυνο για την επεξεργασία $\alpha = \frac{m*n}{gridDim.x*gridDim.y}$ pixel**.** Έτσι, κάθε pixel θα επεξεργαστεί από $\frac{xthreads*ythreads}{a}$, που σημαίνει ότι στις αναλύσεις που ζητήθηκαν:

- Για την εικόνα 64x64 κάθε block επεξεργάζεται 4x4 = 16 pixels που με τη σειρά του συνεπάγεται ότι μέσα στο block κάθε pixel θα υποστεί επεξεργασία από 4x4 = 16 νήματα, και επομένως κάθε νήμα θα επεξεργαστεί τα επιμέρους $[\frac{m}{4}, \frac{n}{4}]$ pixels της εικόνας.

- Για την εικόνα 128x128 κάθε block επεξεργάζεται 8x8 = 64 pixels που με τη σειρά του συνεπάγεται ότι μέσα στο block κάθε pixel θα υποστεί επεξεργασία από 2x2 = 4 νήματα, και επομένως κάθε νήμα θα επεξεργαστεί τα επιμέρους $[\frac{m}{2}, \frac{n}{2}]$ pixels της εικόνας.

- Για την εικόνα 256x256 κάθε block επεξεργάζεται 16x16 = 256 pixels που με τη σειρά του συνεπάγεται ότι μέσα στο block κάθε pixel θα υποστεί επεξεργασία από 1x1 = 1 νήμα, και επομένως κάθε νήμα θα επεξεργαστεί τα επιμέρους $[\frac{m}{1}, \frac{n}{1}]$ pixels της εικόνας. Αυτό συνεπάγεται και μία αρκετά μεγάλη καθυστέρηση καθώς ένα νήμα θα έχει 2 βρόχους επανάληψης μέχρι 256, άρα 256x256=65536 επαναλήψεις, αριθμός αρκετά μεγάλος. Ωστόσο, συγκριτικά με τον πρώτο αλγόριθμο καθώς ένα νήμα επεξεργάζεται ένα στοιχείο αντί για 16x16=256 είναι, όπως θα παρουσιαστεί και παρακάτω, περίπου 30 φορές πιο γρήγορος.

Η shared memory χρησιμοποιήθηκε για την αποθήκευση 3 πινάκων ·

- Του δισδιάστατου πίνακα **output** με διαστάσεις [**blockDim.x** , **blockDim.y]** στον οποίο αποθηκεύεται κάθε στιγμή το επιμέρους γινόμενο $e^{-\frac{||f(Ni)-f(Nj)||^2 G(a)}{\sigma^2}}$*** f(j).** Τα επιμέρους αυτά αθροίσματα, λαμβάνοντας υπόψη τον αριθμό των νημάτων που χρησιμοποιούνται για την επεξεργασία καθενός pixel (όπως αναλύθηκε παραπάνω), αθροίζονται κατάλληλα και προκύπτουν για κάθε pixel μέσα στο block τα συνολικά γινόμενα $e^{-\frac{||f(Ni)-f(Nj)||^2 G(a)}{\sigma^2}}$ * f(j). Πρέπει να προσέξουμε ότι δεν έχει γίνει η διαίρεση με το Z(i) ακόμα. Θα γίνει μετά την ολοκλήρωση των συνολικών παραπάνω γινομένων.

- Του δισδιάστατου πίνακα z_per_block με διαστάσεις [**blockDim.x , blockDim.y]** σε κάθε θέση του οποίου αποθηκεύεται το αποτέλεσμα της μεταβλητής z_per_thread, που περιέχει το μερικό άθροισμα των $e^{-\frac{||f(Ni)-f(Nj)||^2 G(a)}{\sigma^2}}$ για τα pixel που έχουν ήδη εξεταστεί. Έχοντας περάσει αυτές τις τιμές στη shared memory μπορούμε πλέον να πραγματοποιήσουμε επιμέρους αθροίσεις στον πίνακα, με την ίδια ακριβώς λογική με τον πίνακα **output[ ][ ]** ώστε να προκύψουν τα συνολικά Z(i) που χαρακτηρίζουν το pixel i.

- Του δισδιάστατου πίνακα Z_total με διαστάσεις $[\frac{m}{blockDim.x}, \frac{n}{blockDim.y}]$ στον οποίο αποθηκεύονται τα συνολικά Z(i) για κάθε pixel μέσα στο block.

Μέσα στον κώδικα υπάρχουν κάποια σημεία που δεν είναι τόσο προφανής ο τρόπος χρήσης τους γι' αυτό κρίθηκε σκόπιμο να αναλυθούν περαιτέρω. Τα σημεία αυτά είναι οι ορισμοί των μεταβλητών x,y, xx, yy, x_k, y_k .

- **x = blockIdx.x*(m/thrds) + threadIdx.x/w3 ,** w_3 = thrds/w2

  y = blockIdx.y*(n/thrds) + threadIdx.y/w3 w_2 = m/gridDim.x

Οι μεταβλητές x, y αποτελούν τα κέντρα των pixel τα οποία εξετάζει κάθε νήμα. Στην περίπτωση όπου m=n=64 τότε ένα pixel επεξεργάζεται από 4x4=16 νήματα. Πράγματι, με τα δεδομένα που δόθηκαν w_2=4, w_3=4 και στο κάθε pixel αντιστοιχούν 4 threadIdx.x και 4 threadIdx.y. Επομένως, οι μεταβλητές αυτές ορίστηκαν έτσι ώστε το αντίστοιχο νήμα να επεξεργάζεται το κατάλληλο pixel.

- **xx = (threadIdx.x % w3)*w1 + e** w1 = w2*m/thrds

  yy = (threadIdx.y % w3)*w1 + g

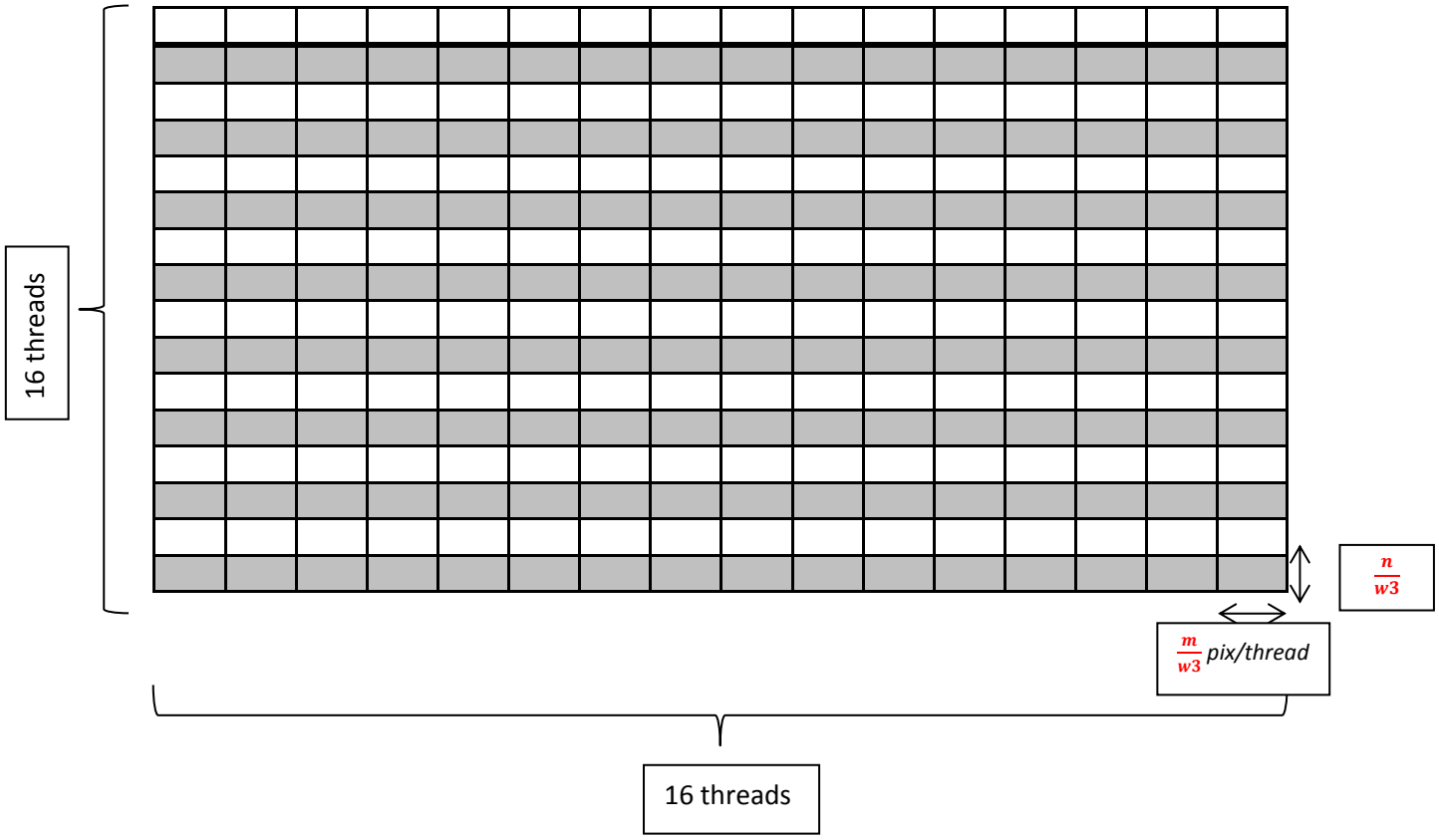
Με τη χρήση αυτών των μεταβλητών μέσα στους βρόχους επανάληψης βρίσκουμε το κέντρο κάθε στοιχείου j που συγκρίνεται με το τρέχων pixel i σε κάθε νήμα. Στην περίπτωση όπου m=n=64 τότε w_1=16 , e: $0\rightarrow \frac{m}{w3}$, g: $0 \rightarrow \frac{n}{w3}$, άρα ανάλογα με το κάθε νήμα θα επεξεργαστεί και διαφορετικό κομμάτι του θορυβοποιημένου πίνακα.

- **xk = x - Reg_x/2 + c**

  yk = y - Reg_y/2 + d

Οι μεταβλητές αυτές χρησιμοποιούνται για να καθορίσουν τους γείτονες του κάθε pixel. Γνωρίζουμε ότι Reg_x = Reg_y = 3 ή 5 ή 7. Επομένως, η διαίρεση με το δύο δίνει 1 , 2 ,3 αντίστοιχα. Χρησιμοποιώντας c: 0→Reg_x , d: 0→Reg_y παίρνουμε όλους τους γείτονες που πρέπει να επεξεργαστούμε. Σε περίπτωση που κάποιο από τα x_k, $y_κ$ προκύψει αρνητικό τότε χρησιμοποιώντας mirror padding επιλέγουμε εκείνο το pixel.

Με το συγκεκριμένο αλγόριθμο πετύχαμε πολύ μικρότερο φόρτο επεξεργασίας ανά νήμα με αποτέλεσμα τη μεγάλη επιτάχυνση των υπολογισμών. Πράγματι, για ένα block στον παρακάτω πίνακα για 16x16 threads παρατηρούμε τη μείωση των υπολογισμών ανά νήμα. Κάθε νήμα εκτελεί 2 βρόχους επανάληψης από $0\rightarrow \frac{m}{w3}$.



Pipeline_non_local_means.m

Για να τρέξει ο κώδικας σωστά απαιτείται και η τροποποίηση του αρχείου **pipeline_locan_means.m .** Στο ίδιο αρχείο που δόθηκε προστέθηκε η εισαγωγή του CUDA kernel, οι απαραίτητες εντολές ώστε οι μεταβλητές που επιθυμούμε να περάσουν σαν ορίσματα στην GPU και να εκτελεστεί ο kernel εντολές για λήψη του χρόνου επεξεργασίας (όπου χρειάζεται η εντολή **wait(gpuDevice))** και η εντολή **saveas(gcf,'image.png')** ώστε η εικόνα που δημιουργεί το MATLAB να αποθηκευτεί και να μπορέσουμε να τη δούμε από τον υπολογιστή μας. Για τη μείωση του χρόνου επεξεργασίας μέσα στην κάρτα γραφικών το mirror padding για τα pixels που βρίσκονται στις άκρες της εικόνα έγινε από το MATLAB. Όλα τα στοιχεία του πίνακα που περιέχει την εικόνα με θόρυβο αντιγράφεται σε έναν άλλο στον οποίο στις άκρες τοποθετούνται τα αντίστοιχα στοιχεία που προέρχονται από το padding της εικόνας.

Σχεδίαση τεχνητής εισόδου

Για τον έλεγχο της ορθότητας των υπολογισμών χρησιμοποιήθηκαν 3 εικόνες διαστάσεων 64x64, 128x128, 256x256 που υπάρχουν στο φάκελο **Matlab_images.** Αρχικά, απαιτείται επεξεργασία της εικόνας ώστε να κάθε pixel να παίρνει τιμές στο διάστημα [0,1] και να μετατραπεί σε αρχείο με κατάληξη **.mat** . Για τη μετατροπή αυτή χρησιμοποιήθηκε ένα μικρό script υλοποιημένο σε MATLAB το οποίο παρατίθεται στο φάκελο **mat_script.** Το μόνο που χρειάζεται είναι κάθε στοιχείο του πίνακα της εικόνας να διαιρεθεί με το 256 ώστε η εικόνα να βρίσκεται στην περιοχή τιμών [0,1] και στη συνέχεια καθένα από αυτά τα στοιχεία να αποθηκευτούν σε έναν πίνακα με κατάληξη .mat .

Ταχύτητα υπολογισμών στο «ΔΙΑΔΗΣ»

Υπολογισμοί στο «ΔΙΑΔΗΣ» έγιναν και για τους 3 κώδικες που συζητήθηκαν, δηλαδή τον κώδικα που δόθηκε από τους διδάσκοντες υλοποιημένο σε MATLAB ,τον πρώτο αλγόριθμο που αναφέρθηκε παραπάνω και τον βασικό αλγόριθμο που χρησιμοποιήθηκε, έχοντας όπως φαίνεται και στον παρακάτω πίνακα πολύ μεγάλη επιτάχυνση σε σχέση με τους προαναφερθέντες. Ο χρόνος μετράται σε δευτερόλεπτα. Ο πίνακας με διαφορετικό χρώμα παρουσιάζει τους χρόνους από τον «κακό» αλγόριθμο.

MATLAB

	64x64	128x128	256x256
3x3	**2,1328**	**37,43**	-
5x5	**1,124**	**28,544**	-
7x7	**0,96**	**20.05**	-

CUDA bad algorithm

	64x64	128x128	256x256
3x3	**0.75**	**14.15**	**263.57**
5x5	**1.96**	**37,17**	**682,086**
7x7	**3,75**	**71,56**	**1313,05**

CUDA proper algorithm

	64x64	128x128	256x256
3x3	**0.031**	**0.41**	**8,15**
5x5	**0.07**	**1.2**	**21,29**
7x7	**0.14**	**2,6**	**41,1**

Παρατηρούμε ότι για τις συνθήκες που επιλέχθηκαν ο αλγόριθμος είναι πολύ γρήγορος τόσο σε σχέση με τον «κακό» αλγόριθμο σε CUDA όσο και με τον αλγόριθμο σε MATLAB. Οι επιταχύνσεις οι οποίες λαμβάνονται είναι πολύ μεγάλες. Στον παρακάτω πίνακα παρουσιάζονται οι επιταχύνσεις του αλγορίθμου σε σχέση με το MATLAB για τις περιπτώσεις των 64x64 και 128x128 και με τον «κακό» αλγόριθμο σε CUDA για 256x256 pixels.

Acceleration(*Matlab/Proper & bad/Proper*)

	64x64	128x128	256x256
3x3	**68,8**	**91,29**	**32,34**
5x5	**16,05**	**23,78**	**32,037**
7x7	**6,857**	**7,71**	**31,94**

Ο αλγόριθμος που υλοποιήθηκε για τα πλαίσια αυτής της εργασίας παρατηρούμε ότι είναι έως και 91 φορές ταχύτερος από τον αλγόριθμο σε MATLAB για patchSize 3x3 και 128x128 pixels ενώ η ελάχιστη επιτάχυνση βρίσκεται στις περιπτώσεις όπου το patchSize είναι 7x7 και 64x64 pixels. Σε σχέση με τον «κακό» αλγόριθμο η επιτάχυνση για 256x256 είναι 32 περίπου. Από τα αποτελέσματα αυτά είναι προφανής ο λόγος που χρησιμοποιούνται οι κάρτες γραφικών για παράλληλο προγραμματισμό. Οι ταχύτητες επεξεργασίας είναι πολύ μεγαλύτερες από τις αντίστοιχες που τρέχουν σε CPU.

Σφάλματα

Για να παρουσιαστούν τα σφάλματα, λόγω του μεγάλου όγκου δεδομένων, κρίθηκε σκόπιμο να βρεθούν μόνο το μέγιστο σφάλμα που προκύπτει σε ένα pixel της φιλτραρισμένης εικόνας σε σχέση με την αρχική καθώς και η μέση τιμή του σφάλματος όλων των στοιχείων. Στη δεύτερη περίπτωση, για την εξαγωγή της μέσης τιμής υπολογίστηκε στον πίνακα που επιστρέφει στο MATLAB Χ η μέση τιμή κάθε στήλης και έπειτα από τον πίνακα που προκύπτει η μέση τιμή του. Με αυτό το σκεπτικό στους παρακάτω πίνακες απεικονίζονται το μέγιστο σφάλμα και η μέση τιμή του σφάλματος. Όλοι οι υπολογισμοί σε σχέση με τον αλγόριθμο που υλοποιήθηκε σε CUDA.

Max Error

	64x64	128x128	256x256
3x3	**0.12**	**0.17**	**0.16**
5x5	**0.116**	**0.114**	**0.148**
7x7	**0.131**	**0.113**	**0.1578**

Mean Value

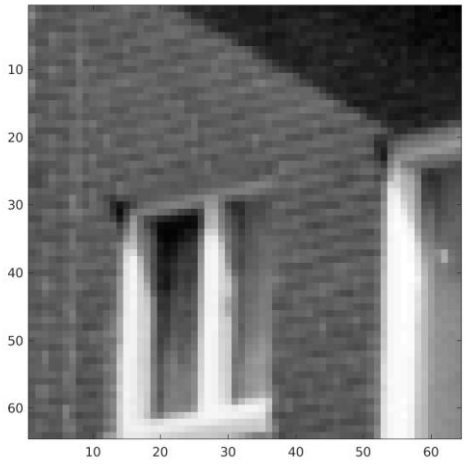
	64x64	128x128	256x256
3x3	**$5*10^{-4}$**	**0.0012**	**0.0012**
5x5	**0.0003**	**0.0015**	**0.0008**
7x7	**$5.4*10^{-5}$**	**0.0014**	**0.0008**

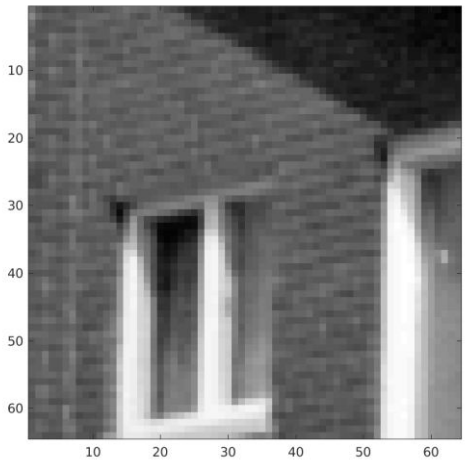
Παρατηρούμε ότι η μέση τιμή του σφάλματος είναι ένας αριθμός πολύ μικρός ενώ το μέγιστο σφάλμα σε ένα pixel επιδρά λίγο με απόρροια η φιλτραρισμένη εικόνα να φαίνεται ίδια ακριβώς με την αρχική. Παρακάτω παρατίθενται οι φιλτραρισμένες εικόνες σε σχέση με τις αρχικές ενώ στο φάκελο **Matlab_Images** υπάρχουν και οι υπόλοιπες εικόνες (με θόρυβο και τα υπολείμματα).

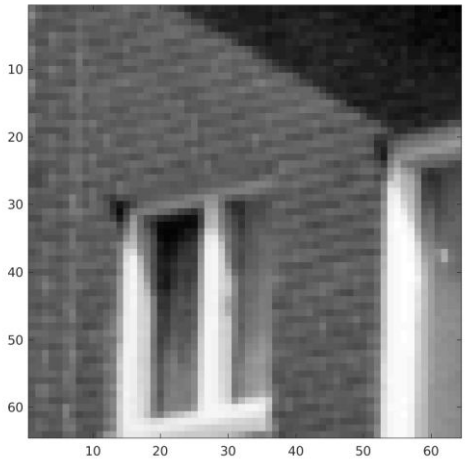
64x64

Με τη σειρά βρίσκονται οι εικόνες για περιοχές (3x3 , 5x5 , 7x7)

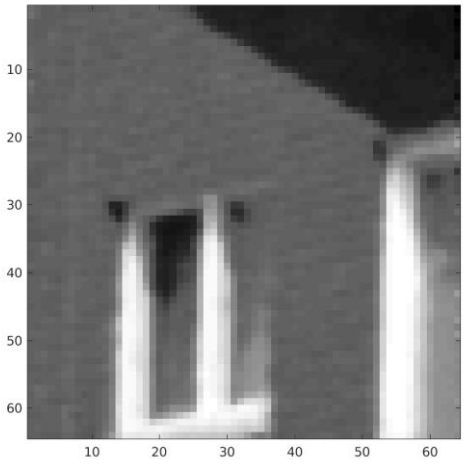
Original

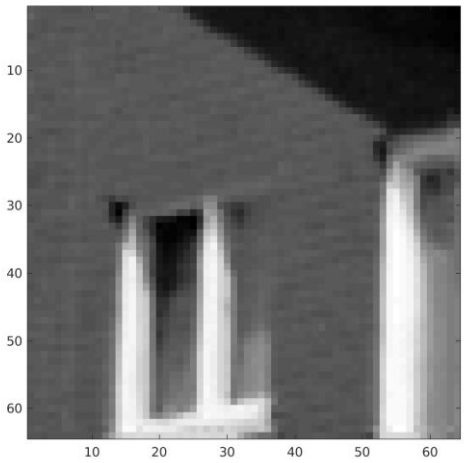


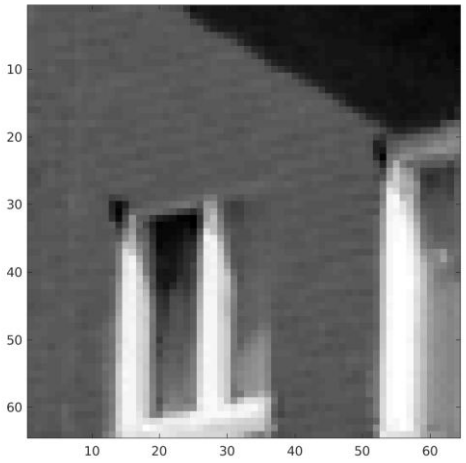




Filtered



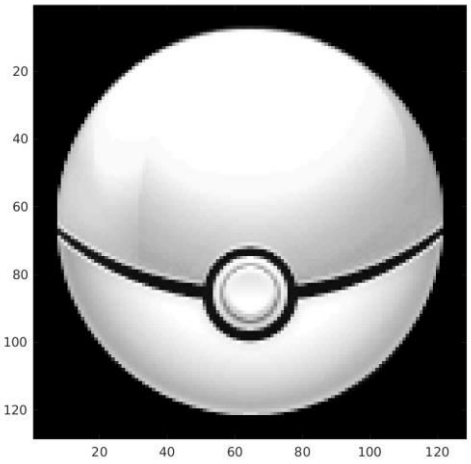


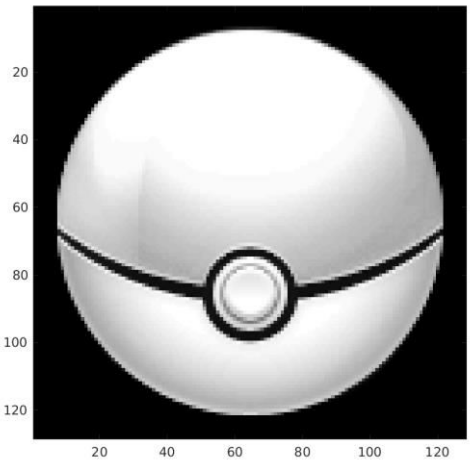


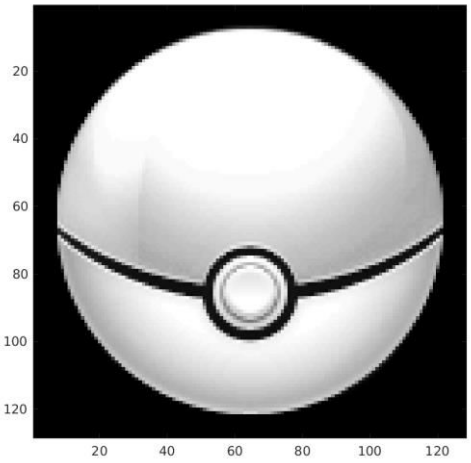
128x128

Με τη σειρά βρίσκονται οι εικόνες για περιοχές (3x3 , 5x5 , 7x7)

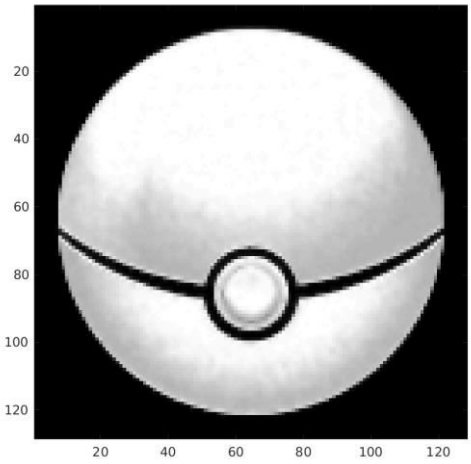
Original

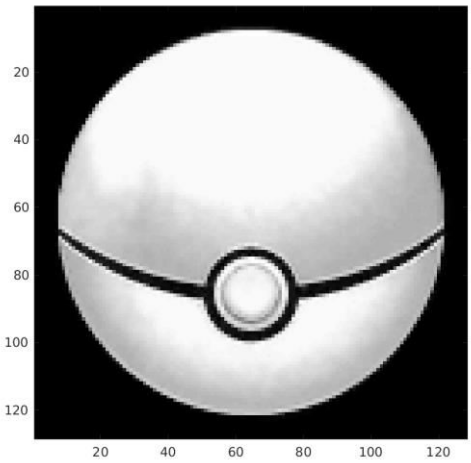


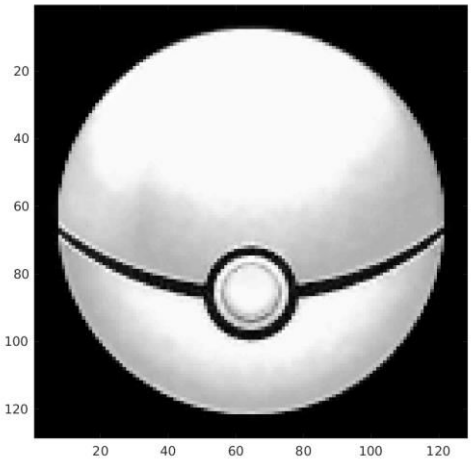




Filtered



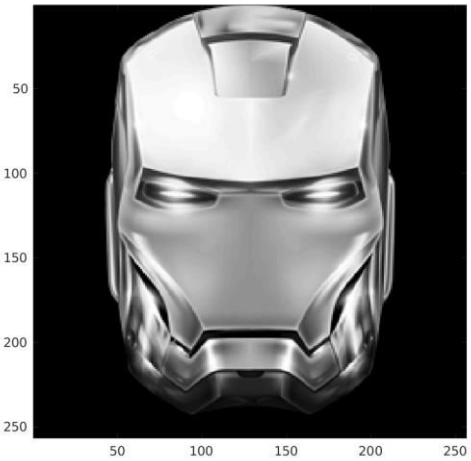


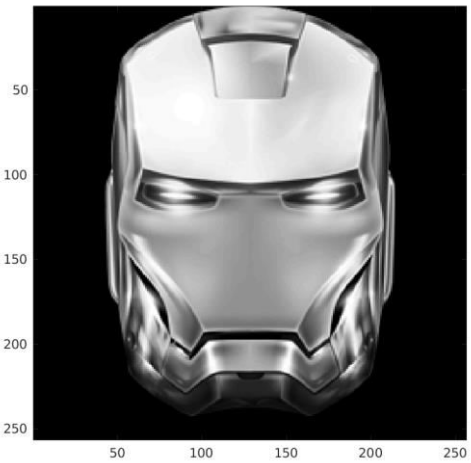


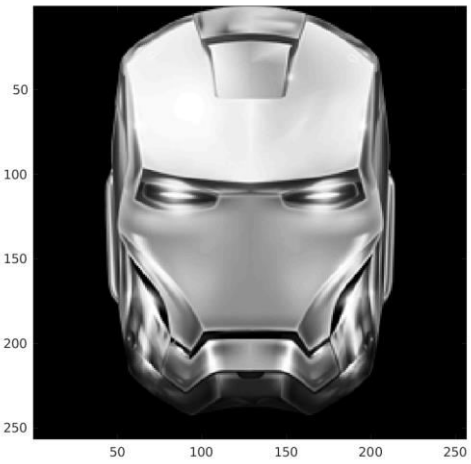
256x256

Με τη σειρά βρίσκονται οι εικόνες για περιοχές (3x3 , 5x5 , 7x7)

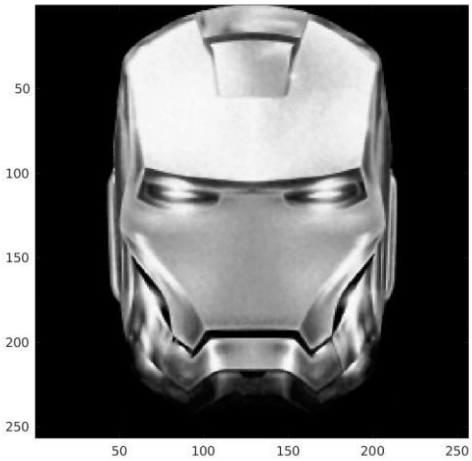
Original

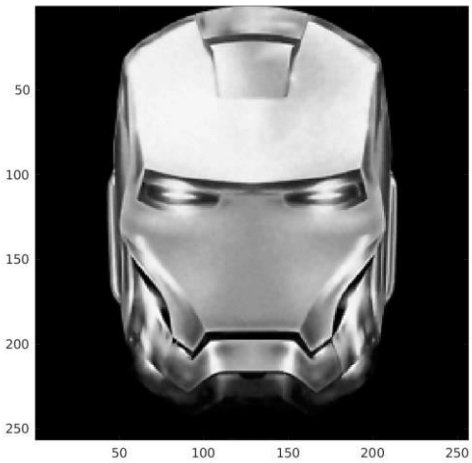


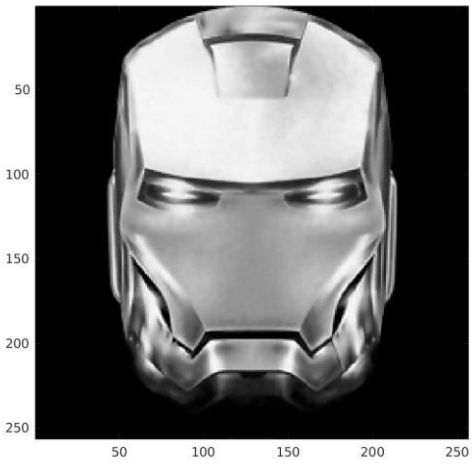




Filtered







Όπως φαίνεται ξεκάθαρα, οι εικόνες παρουσιάζουν πολύ μεγάλη ομοιότητα μεταξύ τους. Είναι προφανές ότι δεν έχει εξαλειφθεί όλος ο θόρυβος, ωστόσο, ο αλγόριθμος non-local-means δούλεψε με αρκετά μεγάλη ακρίβεια. Πολύ πιθανόν, ο θόρυβος να μειωθεί με την επιλογή ενός άλλου **filtSigma** και **patchSigma.** Με τα δοθέντα, η ευκρίνεια της εικόνας δεν μπορεί να βελτιωθεί περισσότερο.