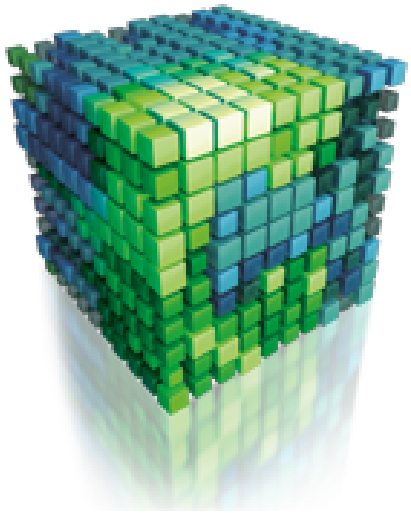
ΠΑΡΑΛΛΗΛΑ ΚΑΙ ΔΙΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

ΕΡΓΑΣΙΑ ΙΙΙ

Παραλληλοποίηση του αλγορίθμου Pipeline for non local means με χρήση Cuda μέσω Matlab.



ΜΑΣΤΟΡΑΣ ΡΑΦΑΗΛ ΕΥΑΓΓΕΛΟΣ

ΑΕΜ 7918

ΙΑΝΟΥΑΡΙΟΣ 2017

Περιγραφή του προγράμματος

Το πρόγραμμα που μας δόθηκε υλοποιεί τον αλγόριθμο Non Local Means ο οποίος βοηθάει στην αποθορυβοποίηση μιας εικόνας, με πολύ καλά αποτελέσματα. Σε αντίθεση με τα Local Means φίλτρα που παίρνουν την μέση τιμή μιας γειτονιάς pixel , ο αλγόριθμος Non local means χρησιμοποιεί την μέση τιμή όλων των pixel της εικόνας, σταθμισμένη με την ομοιότητα των εκάστοτε pixel. Τα παραπάνω έχουν σαν αποτέλεσμα την πολύ καλή αποθορυβοποίηση μιας εικόνας χωρίς να χάνονται τα ιδιαίτερα χαρακτηριστικά της ή έστω σε πολύ μικρό βαθμό.

Υλοποίηση χωρίς shared memory:

Το πρόγραμμα που υλοποίησα χρησιμοποιεί Matlab για κάποιες βασικές λειτουργίες και για την αποθορυβοποίηση της εικόνας καλεί Cuda kernel γραμμένο σε C. Αρχικά ζητείται από τον χρήστη να επιλέξει την ανάλυση της εικόνας και τον αριθμό των γειτονικών σημείων με βάση τον οποίο θα γίνει η επεξεργασία της εικόνας. Έπειτα αφότου διαβάσει το πρόγραμμα την εικόνα και της εισάγει θόρυβο, ορίζω τις τιμές των k.ThreadBlockSize=[32 32] και k.GridSize=ceil([m n] ./ [32 32]) όπου m*n το μέγεθος της εικόνας. Δημιουργώ τους απαραίτητους float πίνακες στη GPU με την εντολή single(zeros([m n], 'gpuArray')) ενώ επίσης μεταβιβάζω τον πίνακα J (θορηβοποιημένη εικόνα) και H (gaussian patch) στην gpu με την εντολή J=single(zeros([m n], 'gpuArray')) , όμοια για τον H. Έπειτα γίνεται η κλίση του cuda kernel με την εντολή If=gather(feval(k,Jgpu,If,Zgpu,filtSigma*filtSigma,patchSize,m,n,H)) που θα επιστρέψει στον πίνακα If την αποθορηβοποιημένη εικόνα.

Μέσα στο kernel , πρώτα βρίσκω την γειτονιά κάθε σημείου. Αν κάποιο σημείο του patch δεν υπάρχει (ακριανά σημεία) τότε παίρνω το διαμετρικό του. Έπειτα βρίσκω το τετράγωνο της διαφοράς των εκάστοτε σημείων επί τον Gaussian συντελεστή των σημείων , ο οποίος λειτουργεί ως συντελεστής ομοιότητας , δηλαδή όσο πιο μακρυά είναι το εκάστοτε σημείο από το κεντρικό της γειτονιάς , τόσο λιγότερο θα συμβάλει στην παραπάνω πράξη. Το αποτέλεσμα αυτής της πράξης υψώνεται στο e με αρνητικό πρόσημο, αφού ξανα υψωθεί στο τετράγωνο και διαιρεθεί από τον συντελεστή φιλτραρίσματος. *Γνωρίζω ότι ο τύπος που περιέγραψα δεν είναι ακριβώς η ευκλείδια απόσταση μεταξύ των σημείων καθώς θα έπρεπε να χρησιμοποιήσω την ρίζα αυτού. Ωστόσο με αυτόν τον τύπο πέτυχα τα καλύτερα αποτελέσματα από όλους όσους δοκίμασα.*

Στην συνέχεια το παραπάνω αποτέλεσμα προστίθεται στους πίνακες Z και If. Επίσης ο πίνακας If το πολλαπλασιάζει με το αντίστοιχο θορυβοποιημένο pixel. Τέλος μετά το πέρας της παραπάνω συνάρτησης, ο πίνακας If[i] διαιρείται από τον πίνακα Z[i] και όταν τελειώσουν όλα τα threads τα παραπάνω, η εικόνα μας έχει επεξεργαστεί και έχει απομακρυνθεί ο θόρυβος.

Υλοποίηση με shared memory:

Η υλοποίηση με shared memory διαφέρει ελάχιστα από την παραπάνω υλοποίηση. Η σημαντική διαφορά είναι πως αντί να καλέσω μία φορά τον cuda kernel για ολόκληρη την εικόνα, την σπάω σε μικρά κομμάτια μεγέθους [32 32] και καλώ τον kernel μια φορά για το καθένα κομμάτι. Το σπάσιμο της εικόνας σε κομμάτια το επιτυγχάνω με την εντολή J = mat2cell(J,repmat(bs(1),1,nb(1)), repmat(bs(2),1,nb(2))) όπου δημιουργεί έναν πίνακα που τα κελιά του αποτελούνται από μικρότερους πίνακες 32x32 , οι οποίοι όλοι μαζί δημιουργούν τον αρχικό πίνακα. Έπειτα μέσα σε μια διπλή for παίρνω κάθε ένα από αυτά τα κομμάτια με την εντολή Jsplit = cell2mat(J(i,j)) και καλώ τον kernel με την εντολή

splitIf{i,j}=gather(feval(k,Jsplit,If,Zgpu,filtSigma*filtSigma,patchSize,bs(1),bs(2),H));
η οποία θα αποθηκεύσει στον πίνακα splitIf το κάθε ένα από τα κερματισμένα αποτελέσματα.

Τέλος όταν η παραπάνω διαδικασία τελειώσει συγκεντρώνω όλα τα παραπάνω αποτελέσματα σε έναν ενιαίο πίνακα με την εντολή If = cell2mat(splitIf), το ίδιο κάνω και για τον πίνακα J που είχα κερματίσει. Για τα παραπάνω χρειάζεται βέβαια και διαφορετικός ορισμός των παραμέτρων του kernel. Αυτή την φορά το k.ThreadBlockSize=[32 32] , ενώ το k.GridSize=ceil([32 32] ./ [32 32]) δηλαδή [1 1]. Σκέφτηκα ότι δημιουργώντας μόνο ένα block ίσως μπορούσα να καλέσω σύγχρονος τον kernel για τα επιμέρους κομμάτια της εικόνας με μια parfor άλλα το Matlab δεν μου το επέτρεπε.

Μέσα στο cuda kernel τώρα η αλλαγές που χρειάστηκε να κάνω είναι η δήλωση τον μεταβλητών __shared__ float sharedJ[1024] , sharedZ[1024] ; καθώς και η αρχικοποίηση τους με την εντολή sharedJ[p*rows+q]=J[p*rows+q] και sharedZ[p*rows+q]=Z[p*rows+q] . Έπειτα αντί για τους πίνακες J Και Z χρησιμοποιώ τους sharedJ και sharedZ.

Όλα τα παραπάνω τα κάνω προκειμένου να εκμεταλλευτώ την shared memory της GPU, καθώς είναι μικρού μεγέθους και δεν θα χωρούσε ολόκληρη την εικόνα . Έτσι κατάφερα να πετύχω πολύ μεγάλη επιτάχυνση.

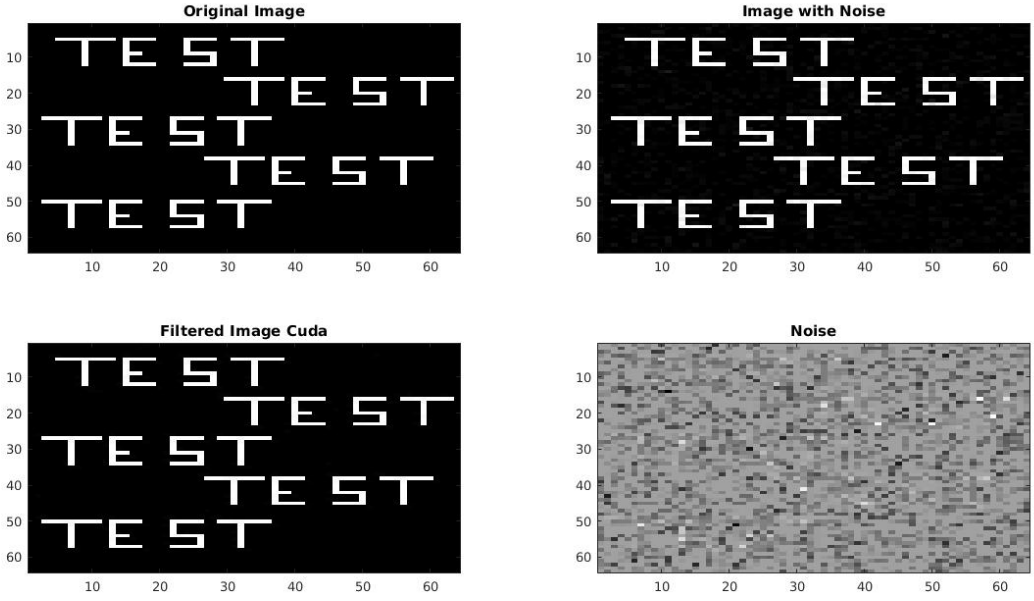
Πίσω στο Matlab τώρα ,συγκεντρώνονται τα επιμέρους κομμάτια της αποθορυβοποιημένης εικόνας και εμφανίζονται τα αποτελέσματα στην οθόνη.

Έλεγχος ορθότητας αποτελεσμάτων:

Για τον έλεγχο ορθότητας των αποτελεσμάτων χρησιμοποίησα δύο δείκτες σε αναλογία με τα αποτελέσματα της υλοποίησης που μας δόθηκε. Ο πρώτος δείκτης στον οποίο βασίστηκα περισσότερο είναι ο Peak signal-to-noise ratio (PSNR) ο οποίος μας δίνει αρκετά καλή εικόνα του πόσο όμοια είναι η αποθοροιβοποιημένη εικόνα με την αρχική. Θεώρησα πως τιμές μεγαλύτερες του 30 είναι αποδεκτές καθώς εκεί κυμαίνονται και οι τιμές που μας δίνει και το αποτέλεσμα του δοθέν σειριακού προγράμματος. Το εύρος του δείκτη PSNR είναι γύρω στο 30 καθώς η μέγιστη τιμή Pixel που μπορούμε να έχουμε σε grayscale θεωρήσα πως είναι το 1 (φαίνεται από τα matfile των εικόνων). Ο δεύτερος δείκτης είναι ο Structural Similarity Index (SSIM) με ελάχιστη τιμή το 0 και μέγιστη το 1, θεώρησα πως αποδεκτά αποτελέσματα είναι κοντά στο 0.9 .

Όλα τα αποτελέσματα ,που δίνει ο αλγόριθμος αυτός, είναι μεγαλύτερα της ελάχιστης τιμής 30 και πολύ κοντά στα αντίστοιχα του αρχικού αλγορίθμου. Σε μερικές περιπτώσεις ίσως και καλύτερα.

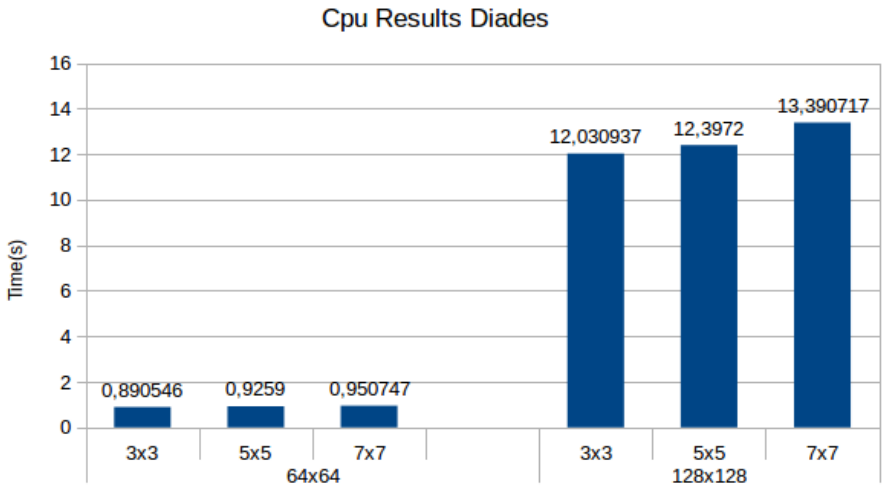
Επίσης όπως ζητήθηκε δημιούργησα μια τεχνητή είσοδο για δοκιμή της ορθότητας του αποτελέσματος. Παραμένει αναλλοίωτη οπτικά ωστόσο αν κοιτάξουμε το matfile έχει μικρές αποκλίσεις στις τιμές, πράγμα που συμβαίνει ακόμα και στον αρχικό κώδικα που μας δόθηκε.



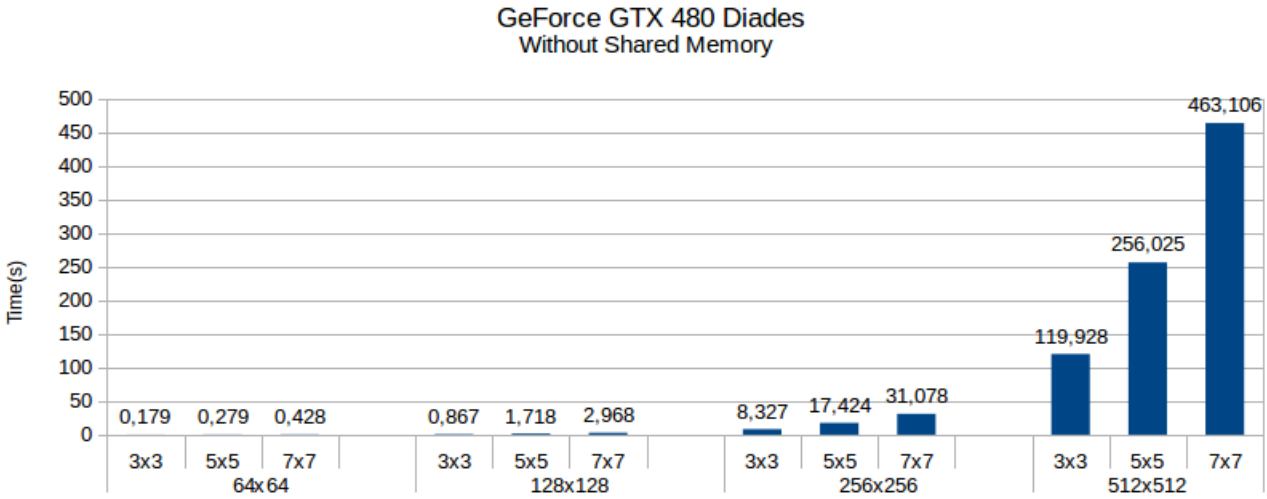
Αποτελέσματα και σχολιασμός

Δυστυχώς ο σειριακός αλγόριθμος δεν έτρεχε για εικόνες μεγαλύτερες του 128x128 καθώς απαιτούσε 16GB διαθέσιμης μνήμης! Ωστόσο τέτοιο πρόβλημα δεν αντιμετωπίζει και η δική μου υλοποίηση. Όπως θα δούμε με την χρήση cuda και με την τεχνική διαμοιρασμού της εικόνας σε μικρότερα κομμάτια , δίνεται η δυνατότητα να επεξεργαστούμε πολύ μεγάλες εικόνες σε μικρό χρονικό διάστημα, με πολύ καλά οπτικά αποτελέσματα.

Ας δούμε τους χρόνους του σειριακού προγράμματος.

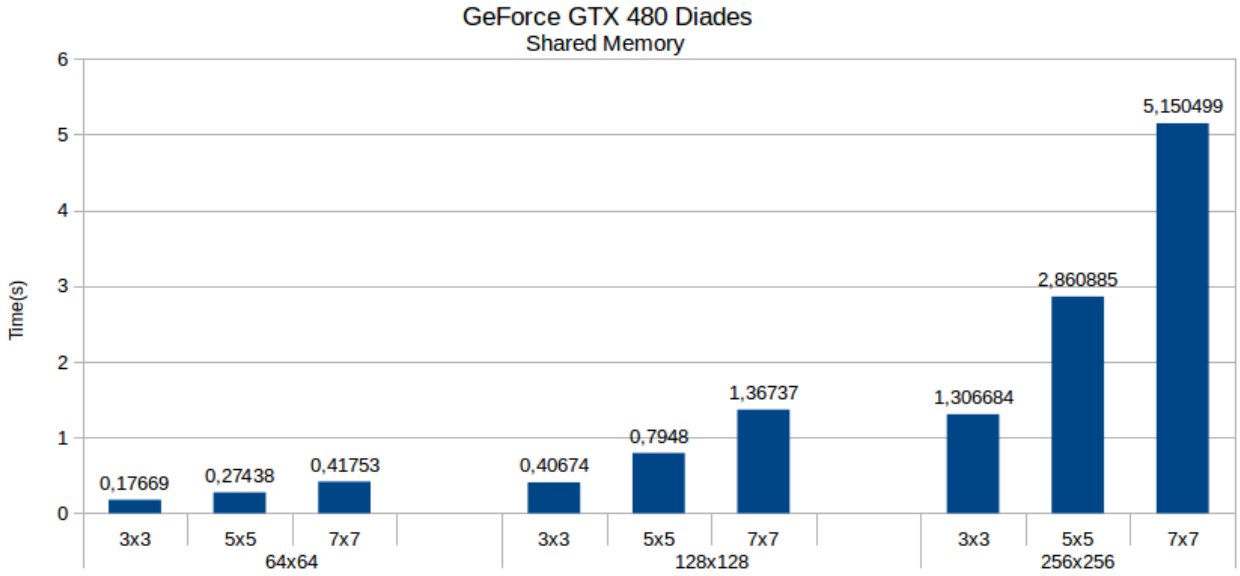


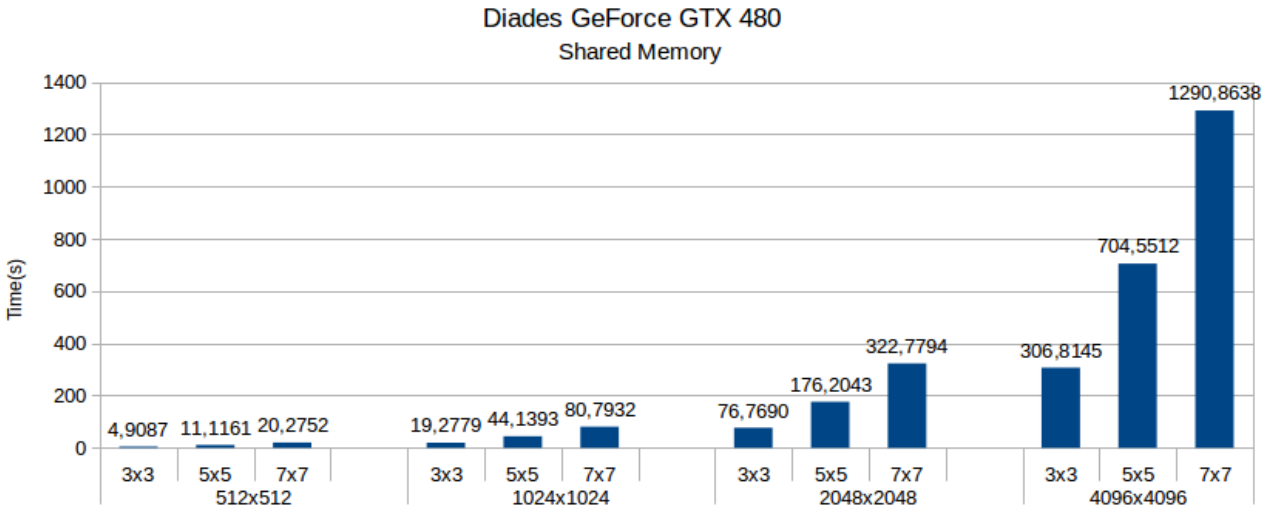
Βλέπουμε ότι ήδη σε μία αύξηση του μεγέθους της εικόνας *4 ο χρόνος 13πλασιάστηκε. Μπορούμε να υποθέσουμε ότι θα μεγάλωνε ακόμα περισσότερο σε επόμενες μεγαλύτερες τιμές εικόνων καθώς θα χρειαζόταν η χρήση σελιδοποιημένης περιοχής μνήμης, καθώς αν χρειάζονται 16GB μνήμης για 256*256 εικόνα, το τετραπλάσιο μνήμης θα χρειαζόταν για μια εικόνα 512*512. (Τα παραπάνω οφείλονται στην χρήση της εντολής squareform(pdist(B)) που δημιουργεί πολύ μεγάλο πίνακα). Τώρα ας δούμε τα ζητούμενα αποτελέσματα **χωρίς** την χρήση Shared memory στο server diades.



Ήδη παρατηρούμε πολύ μεγάλη βελτίωση της ταχύτητας ενώ μας επιτρέπεται η επεξεργασία μεγαλύτερων εικόνων.

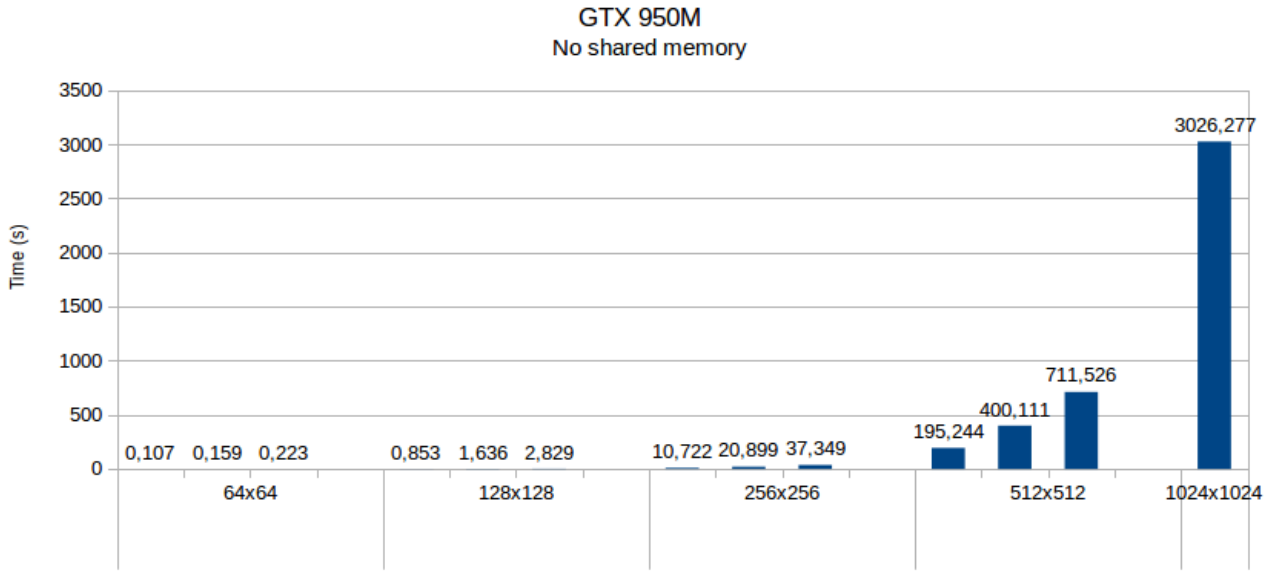
Και τα αποτελέσματα της χρήσης Shared Memory και κατακερματισμό της εικόνας σε μικρότερα κομμάτια.



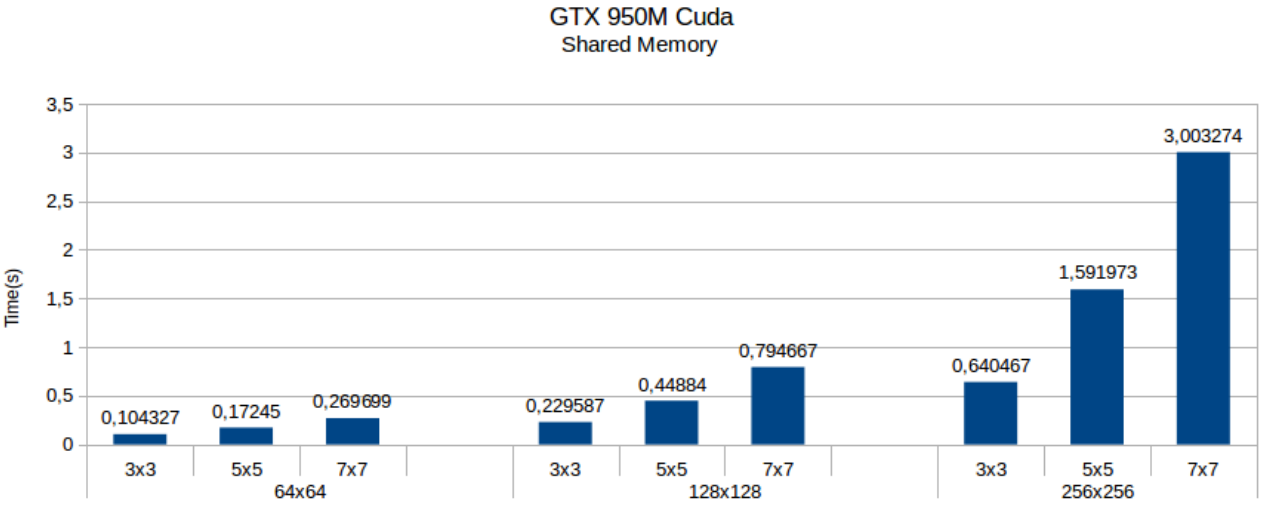


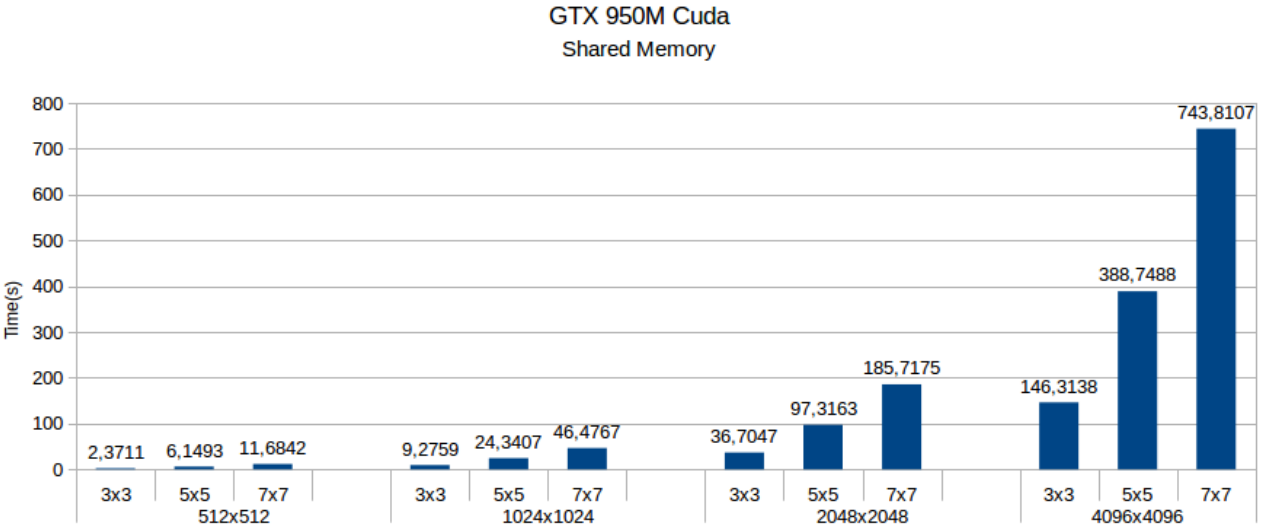
Εδώ τώρα φαίνεται πια η απίστευτη επιτάχυνση που μπορεί να επιτύχει κανείς χρησιμοποιώντας την shared memory της κάρτας γραφικών. Ενδεικτικά για 128x128 πίνακα ,χωρίς shared memory έχουμε επιτάχυνση σε σχέση με το σειριακό πρόγραμμα της τάξης 1387.65% ενώ με χρήση shared memory έχουμε 2957.8% επιτάχυνση. Ασφαλώς όμως αυτές οι τιμές είναι πραγματικά μικρές καθώς και το μέγεθος της εικόνας είναι μικρό. Αν μπορούσαμε να συγκρίνουμε την επιτάχυνση για μεγαλύτερες εικόνες σίγουρα θα ήταν κατά πολύ μεγαλύτερη.

Ας δούμε τώρα αντίστοιχα αποτελέσματα με την κάρτα γραφικών GTX 950M (laptop).

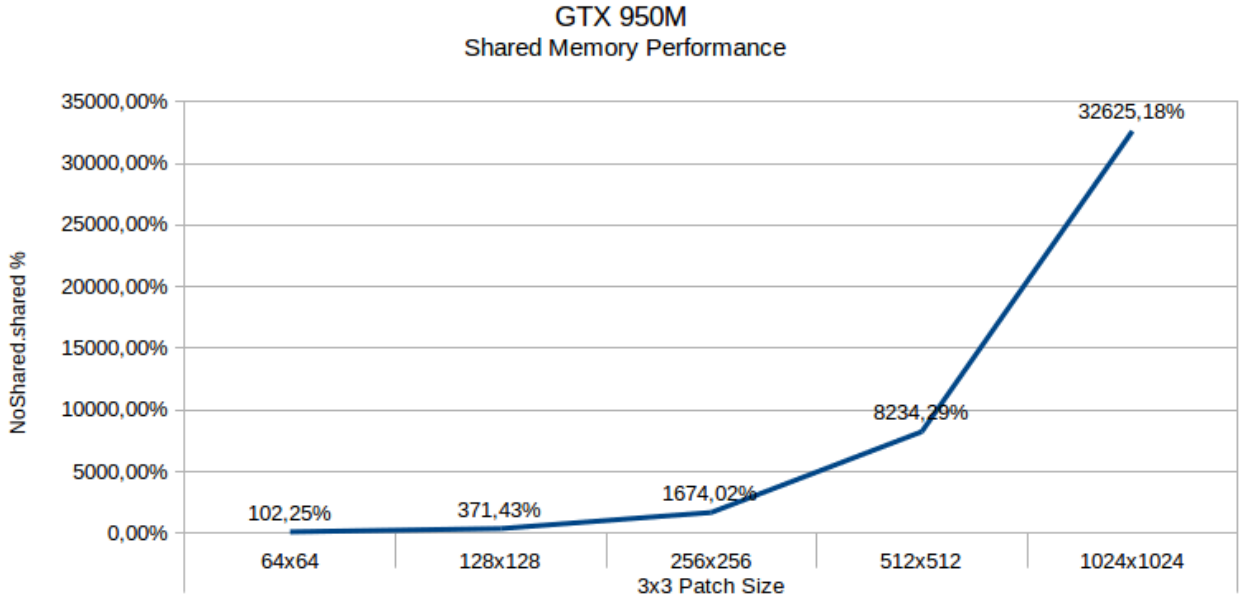


(Το αποτέλεσμα 1024x1024 αφορά 3x3 patchSize)





Και τέλος σύγκρισης μεταξύ υλοποίησης με shared memory και χωρίς για την GTX 950M.

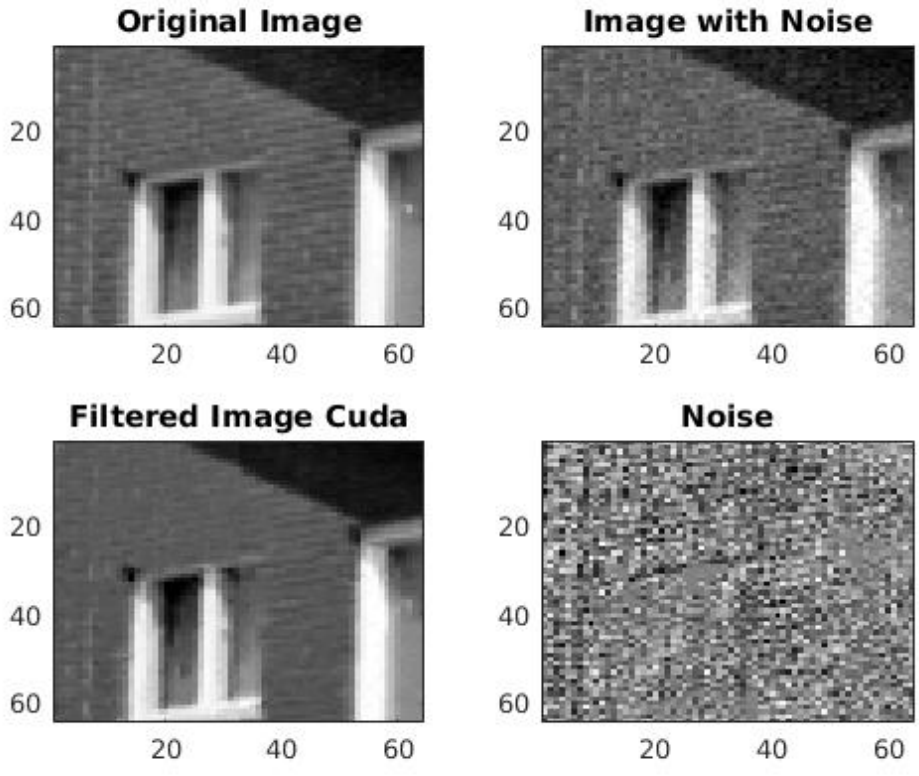


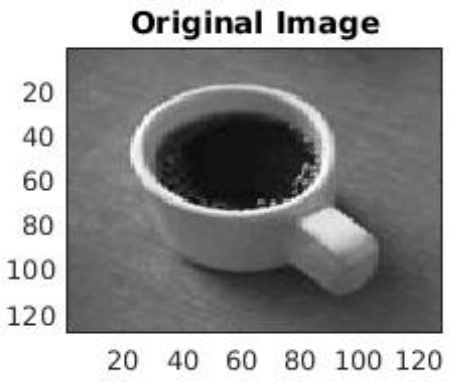
Όπως βλέπουμε η διαφορά είναι ιδιαίτερα μεγάλη και όσο μεγαλώνει το μέγεθος της εικόνας η βελτίωση της απόδοσης ακολουθεί εκθετική μορφή!

Επίσης μεγάλο πλεονέκτημα της υλοποίησης με με διαμοιρασμό της εικόνας είναι η ελάχιστη χρήση μνήμης της GPU. Για οποιαδήποτε μέγεθος εικόνας η χρήση της μνήμης κάρτας γραφικών δεν ξεπερνάει τα 100 MB.

Επίσης απλά αναφέρω ότι παρατηρείται 200% επιτάχυνση υπέρ της GTX950M έναντι της GTX 480 στην υλοποίηση με shared memory. Ενώ η GTX480 υπερτερεί της GTX950 περίπου κατά 165% στην υλοποίηση χωρίς shared memory. Ίσως τα παραπάνω οφείλονται σε ταχύτερη shared memory της GTX950M και μεγαλύτερου δίαυλου στην GTX480 δίνοντας το προβάδισμα όταν συγκρίνουμε την υλοποίηση χωρίς shared memory.

Οπτικά αποτελέσματα




Original Image
20
40
60
80
100
120
20 40 60 80 100 120

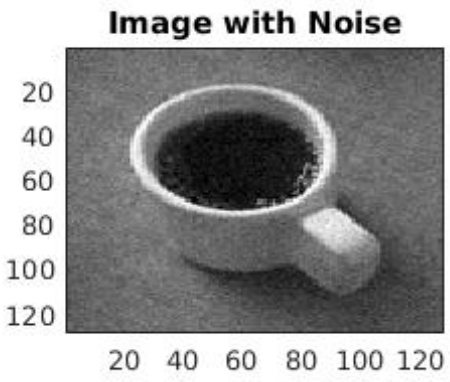
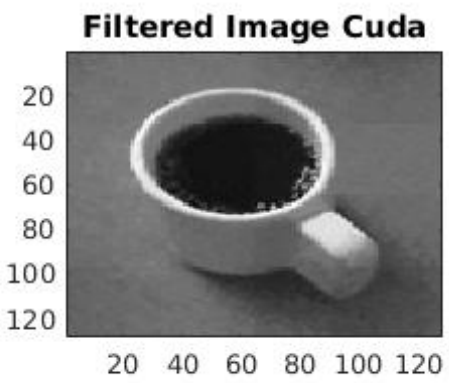
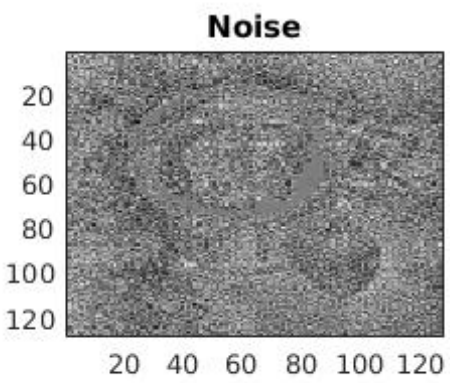
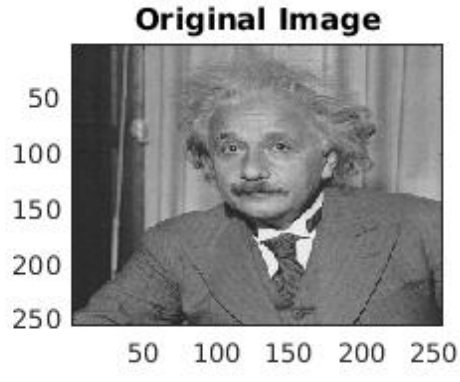


Image with Noise
20
40
60
80
100
120
20 40 60 80 100 120


Filtered Image Cuda
20
40
60
80
100
120
20 40 60 80 100 120


Noise
20
40
60
80
100
120
20 40 60 80 100 120


Original Image
50
100
150
200
250
50 100 150 200 250

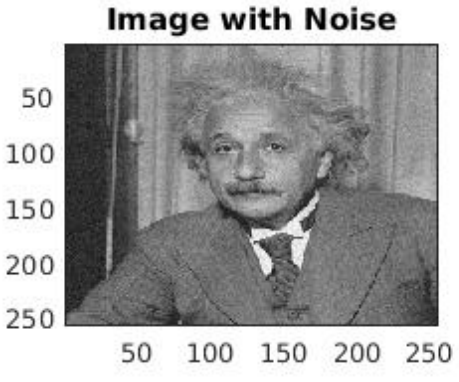
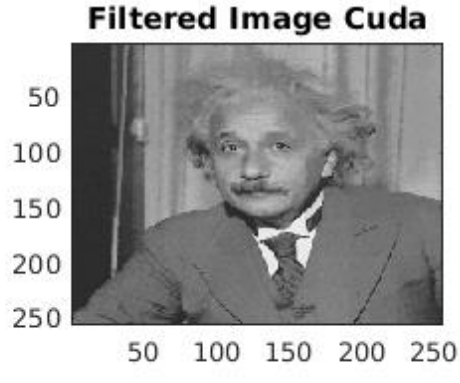
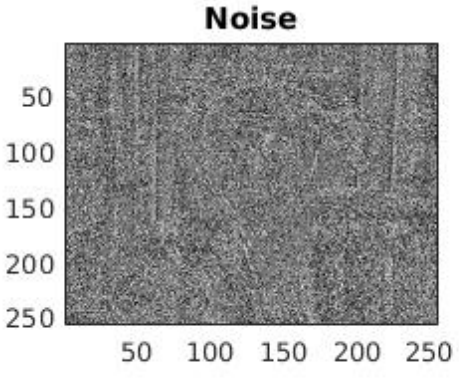
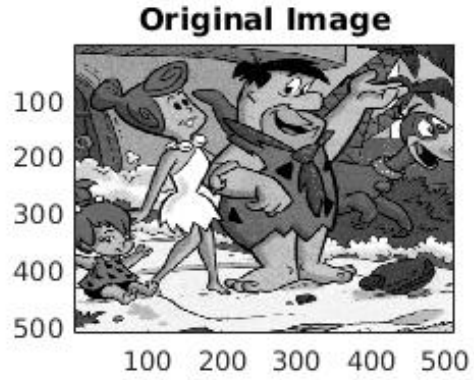


Image with Noise
50
100
150
200
250
50 100 150 200 250


Filtered Image Cuda
50
100
150
200
250
50 100 150 200 250


Noise
50
100
150
200
250
50 100 150 200 250


Original Image
100
200
300
400
500
100 200 300 400 500

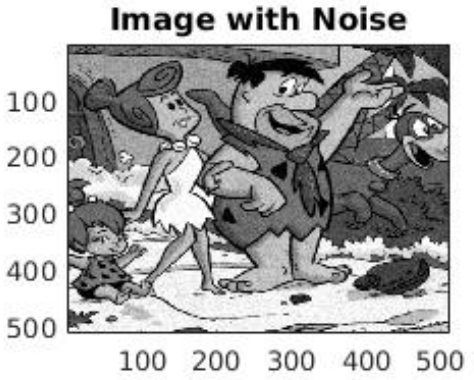
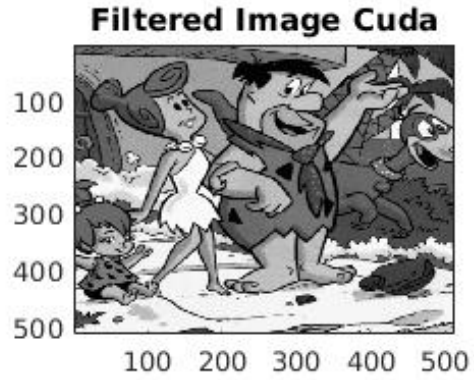
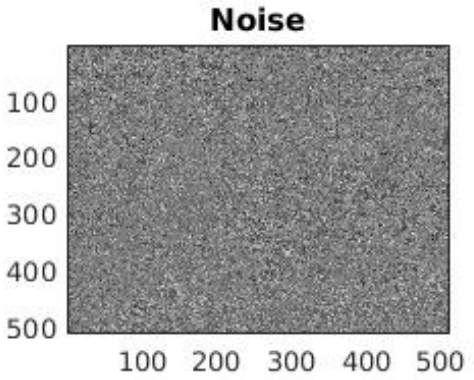
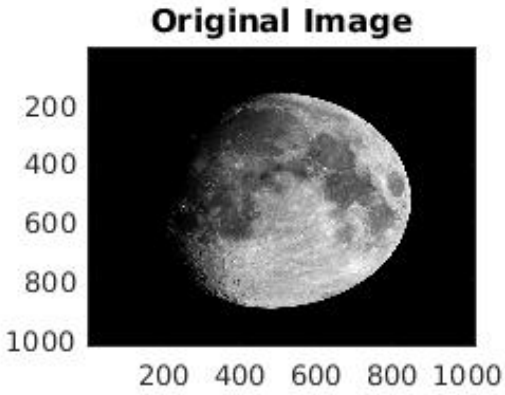


Image with Noise
100
200
300
400
500
100 200 300 400 500


Filtered Image Cuda
100
200
300
400
500
100 200 300 400 500


Noise
100
200
300
400
500
100 200 300 400 500


Original Image
200
400
600
800
1000
200 400 600 800 1000

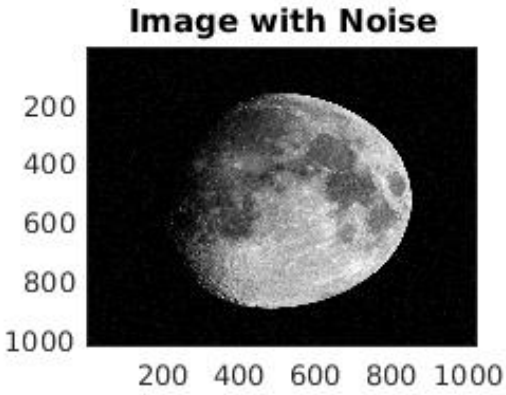
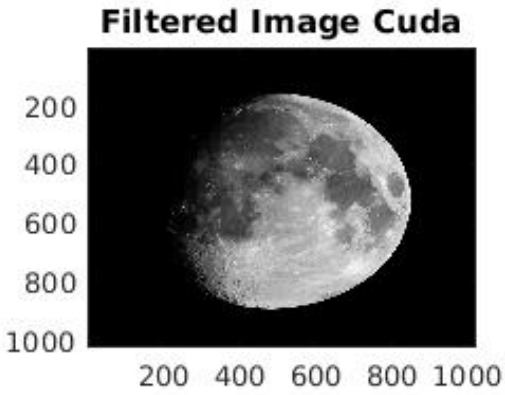
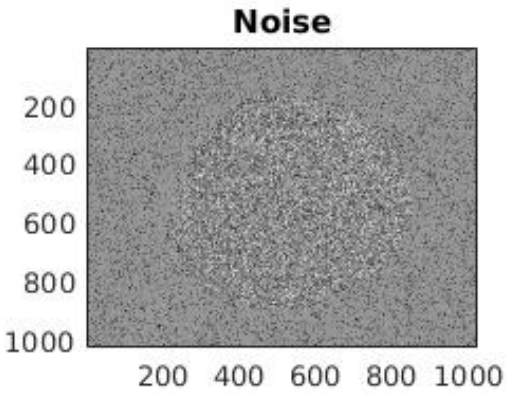
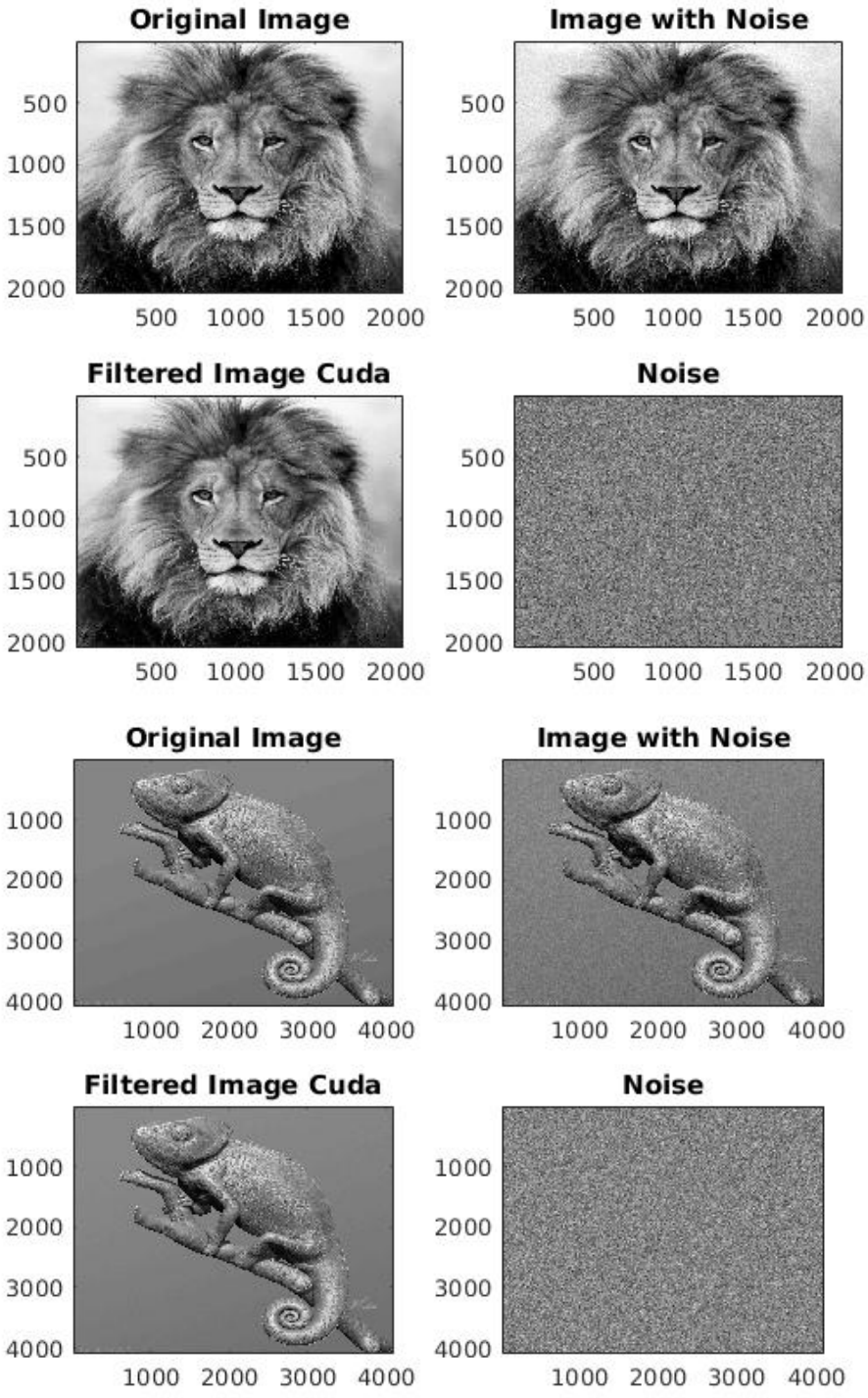


Image with Noise
200
400
600
800
1000
200 400 600 800 1000


Filtered Image Cuda
200
400
600
800
1000
200 400 600 800 1000


Noise
200
400
600
800
1000
200 400 600 800 1000



Όπως βλέπουμε τα οπτικά αποτελέσματα είναι πολύ καλά, χωρίς να χάνονται οι λεπτομέρειες της εκάστοτε εικόνας σε μεγάλο βαθμό. Όλα τα παραπάνω αποτελέσματα είναι έπειτα από την υλοποίηση με shared memory, αλλα δεν διαφέρουν σχεδόν καθόλου με την αντίστοιχη υλοποίηση χωρίς shared memory. Η μικρή διαφορά τους είναι στις μικρού μεγέθους εικόνες (εώς 128x128) όπου καλύτερο οπτικό αποτέλεσμα θεωρώ πως έχει η nosharedMemory υλοποίηση.