

# **Crimmins Speckle Removal Filter**

Τεχνολογία Παράλληλης Επεξεργασίας

**Αναστάσιος Φραγκόπουλος 58633**

Πρώτη εξαμηνιαία εργασία  
Ακαδ. Έτος 2024-2025

Εργαστήριο Αρχιτεκτονικής Υπολογιστών και Συστημάτων Υψηλών Επιδόσεων  
Δημοκρίτειο Πανεπιστήμιο Θράκης  
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

# 1 Σειριακός Αλγόριθμος

Ο Crimmins Speckle removal είναι ένας αλγόριθμος που μειώνει από ασπρόμαυρες εικόνες τον θόρυβο salt-and-pepper, που είναι ένα είδος θορύβου που δίνει την εμφάνιση μίας εικόνας πασπαλισμένης με αλάτι και πιπέρι. Οι επανειλημμένες χρήσεις του αλγόριθμου σε μια εικόνα δίνουν καλύτερα αποτελέσματα μείωσης του θορύβου αλλά προκαλούν θόλωση της εικόνας.

Ο αλγόριθμος αρχικά αυξάνει την φωτεινότητα των pixel που είναι σκοτεινότερα από τους γείτονες του και μετά μειώνοντας την φωτεινότητα των pixel που είναι φωτεινότερα από τους γείτονες του. Η σύγκριση αυτή γίνεται μεταξύ του κάθε pixel και ζευγαριού pixel των 8 γειτόνων για μια κατεύθυνση σε κάθε πέρασμα (πάνω-κάτω, δεξιά-αριστερά, διαγώνια αριστερά-δεξιά, διαγώνια δεξιά-αριστερά).

Για κάθε επανάληψη του αλγορίθμου, γίνονται πρώτα οι παρακάτω συγκρίσεις για την διόρθωσή των σκοτεινών pixel για τις τέσσερις κατευθύνσεις

```
uint8_t dark_pass_logic(uint8_t a, uint8_t b, uint8_t c)
{
    if(a >= b + 2) b++;
    if(a > b && b <= c) b++;
    if(c > b && b <= a) b++;
    if(c >= b + 2) b++;
    return b;
}
```

και έπειτα οι παρακάτω συγκρίσεις για την διόρθωση των φωτεινών pixel για τις τέσσερις κατευθύνσεις

```
uint8_t light_pass_logic(uint8_t a, uint8_t b, uint8_t c)
{
    if(a <= b - 2) b--;
    if(a < b && b >= c) b--;
    if(c < b && b >= a) b--;
    if(c <= b - 2) b--;
    return b;
}
```

Όπου b το κεντρικό pixel κάθε γειτονιάς και a, c το ζευγάρι pixel για μια κατεύθυνσή (π.χ. για την επανάληψη πάνω-κάτω το a θα είναι το pixel πάνω από το b και το c το pixel κάτω από το b).

Οπότε για μία επανάληψη του, ο αλγόριθμος περνάει όλα τα pixel μίας εικόνας τέσσερις φορές για την διόρθωση των σκοτεινών και άλλες τέσσερις για την διόρθωση των φωτεινών pixel. Δηλαδή κάνει  $8 \cdot m \cdot n$  επαναλήψεις όπου m το πλάτος και n το μήκος της εικόνας σε pixel. Άρα η πολυπλοκότητα του είναι  $O(k \cdot m \cdot n)$ , όπου k είναι ο αριθμός των επαναλήψεων του αλγόριθμου για μια εικόνα.

gr\_flower384x256.raw



out\_par.raw



Σχήμα 1: Παράδειγμα αλγορίθμου για 2 επαναλήψεις

gr\_flower384x256.raw



out\_par.raw



Σχήμα 2: Παράδειγμα αλγορίθμου για 8 επαναλήψεις

## 2 Υλοποίηση του αλγόριθμου

Κύριο στοιχείο στην υλοποίηση του αλγόριθμου είναι η συνάρτηση που είναι υπεύθυνη για το πέρασμα του κάθε pixel της εικόνας και είναι φτιαγμένη έτσι ώστε να μπορεί να ξαναχρησιμοποιηθεί για όλες τις κατευθύνσεις ανάλογα με την είσοδο που της δίνεται. Επιπλέον, οι συναρτήσεις που κάνουν την σύγκρισή των γειτονικών pixel, που αναφέρθηκαν παραπάνω δίνονται ως είσοδος στην συνάρτηση για τον ίδιο λόγο. Η κάθε επανάληψη της εξωτερικής for loop μεταβάλλει την γραμμή της εικόνας στην οποία βρισκόμαστε και η εσωτερική μεταβάλλει την στήλη της εικόνας στην οποία βρισκόμαστε. Στην αλλαγή γραμμής υπολογίζουμε έναν pointer που δείχνει στην αρχή κάθε γραμμής, για το a, b, c, ώστε να γλιτώσουμε ένα μέρος των πράξεων που χρειάζεται στις επαναλήψεις του άξονα x. Τέλος, γράφει στο pixel b την τιμή διασφαλίζοντας ότι θα είναι μεταξύ 0 και 255.

```

void pass_func(uint8_t *image, uint8_t *tmp_image, uint32_t width,
               uint32_t height, int dx, int dy,
               uint8_t (*pass_logic_func)(uint8_t, uint8_t, uint8_t)) {

    for(int y = 1; y < height - 1; y++) {
        uint8_t *row_a = tmp_image + (y-dy) * width;
        uint8_t *row_b = tmp_image + y * width;
        uint8_t *row_c = tmp_image + (y+dy) * width;
        uint8_t *row_out = image + y * width;

        for(int x = 1; x < width - 1; x++) {
            uint8_t a = row_a[x-dx];
            uint8_t b = row_b[x];
            uint8_t c = row_c[x+dx];

            b = pass_logic_func(a, b, c);
            row_out[x] = (b < 0) ? 0 : (b > 255) ? 255 : b;
        }
    }
}

```

Οι πράξεις για την επιλογή των a, b, c φαίνεται από το παρακάτω διάγραμμα που δείχνει τις συντεταγμένες (x, y) του κάθε pixel μίας γειτονιάς.

+-----+	+-----+	+-----+
y-1, x-1	y-1, x	y-1, x+1
+-----+	+-----+	+-----+
y, x-1	y, x	y, x+1
+-----+	+-----+	+-----+
y+1, x-1	y+1, x	y+1, x+1
+-----+	+-----+	+-----+

Όπως φαίνεται από την συνάρτηση για κάθε μία από τις 8 επανάληψης της πρέπει να της δίνουμε δύο εικόνες, μία με την προηγούμενη κατάσταση της εικόνας και μία στην οποία θα γράψουμε το αποτέλεσμα. Για να αποφύγουμε την συνεχείς δημιουργία και διαγραφή των buffer μεταξύ των επαναλήψεων, επέλεξα να ανταλλάσω τους pointer των εικόνων, έτσι ώστε η εικόνα στην οποία γράφεται το αποτέλεσμα στην επόμενη επανάληψη να γίνεται η προηγούμενη κατάσταση και η προηγούμενη κατάσταση να γίνεται το καινούργιο αποτέλεσμα.

### 3 Παράλληλη υλοποίηση του Αλγόριθμου

Στον Crimmins Speckle removal αλγόριθμο το μόνο που πρέπει να παραμένει σειριακό είναι η σειρά με την οποία κάνουμε τα περάσματα για την διόρθωση των pixel για κάθε κατεύθυνση. Για τον λόγο αυτό επέλεξα να παραλληλοποιήσω την συνάρτηση που είναι υπεύθυνη για αυτά τα περάσματα. Έκανα την εξωτερική for loop παράλληλη έτσι ώστε να χωριστεί η εικόνα σε  $n/p$  κομμάτια όπου  $n$  οι γραμμές της εικόνας και  $p$  οι πυρήνες του συστήματος.

```
void pass_func_par(uint8_t *image, uint8_t *tmp_image, uint32_t width,
uint32_t height, int dx, int dy,
uint8_t (*pass_logic_func)(uint8_t, uint8_t, uint8_t), int chunk)
{
    #pragma omp parallel for schedule(static, chunk)
    for(int y = 1; y < height - 1; y++) {
        uint8_t *row_a = tmp_image + (y-dy) * width;
        uint8_t *row_b = tmp_image + y * width;
        uint8_t *row_c = tmp_image + (y+dy) * width;
        uint8_t *row_out = image + y * width;

        for(int x = 1; x < width - 1; x++) {
            uint8_t a = row_a[x-dx];
            uint8_t b = row_b[x];
            uint8_t c = row_c[x+dx];

            b = pass_logic_func(a, b, c);

            row_out[x] = (b < 0) ? 0 :
                (b > 255) ? 255 :
                b;
        }
    }
}
```

Σημαντικό εδώ είναι ότι για τον rxeon2, πριν το τρέξουμε τον παράλληλο αλγόριθμο, πρέπει να θέσουμε τις μεταβλητές συστήματος OMP\_PLACE=cores, OMP\_PROC\_BIND=close. Ο rxeon2 έχει τέσσερις φυσικούς Intel Xeon Gold 5218 επεξεργαστές ο κάθε ένας με την δικιά του μνήμη και δικιά του cache. Η μεταβλητές αυτές απλά λένε στο openmp να αρχίζει να γεμίζει πρώτα τα threads του ίδιου επεξεργαστή και να μην τα διανέμει ομοιόμορφα σε όλους. Αυτό μας διασφαλίζει ότι τα δεδομένα που χρειάζεται κάθε thread είναι κοντά τους οπότε δεν έχουμε μείωση του χρόνου γιατί πρέπει να επικοινωνήσουν διαφορετικοί επεξεργαστές μεταξύ τους όλη την ώρα. [2], [3]

Για να μπορώ να ελέγξω ότι τα αποτελέσματα του σειριακού και του παράλληλου αλγόριθμου είναι ίδια, έφτιαξα μια συνάρτηση (image\_validator) που περνάει τις δύο εικόνες και τις ελέγχει pixel προς pixel αν είναι ίδιες.

## 4 Πειραματικά αποτελέσματα

CPU <sub>s</sub>	T <sub>s</sub>	T <sub>p</sub>	Sp
1	0.0069	0.007	0.98
2	0.0068	0.004	1.71
4	0.0069	0.0025	2.8
8	0.0069	0.0043	1.63
16	0.0068	0.0175	0.39
32	0.007	0.014	0.5
64	0.0077	0.0168	0.46

(α') lena256.raw with 1 iteration

CPU <sub>s</sub>	T <sub>s</sub>	T <sub>p</sub>	Sp
1	0.1039	0.1037	1
2	0.1041	0.0657	1.58
4	0.1031	0.0287	3.59
8	0.1029	0.0159	6.49
16	0.1019	0.0163	6.27
32	0.1023	0.0378	2.71
64	0.1044	0.0725	1.44

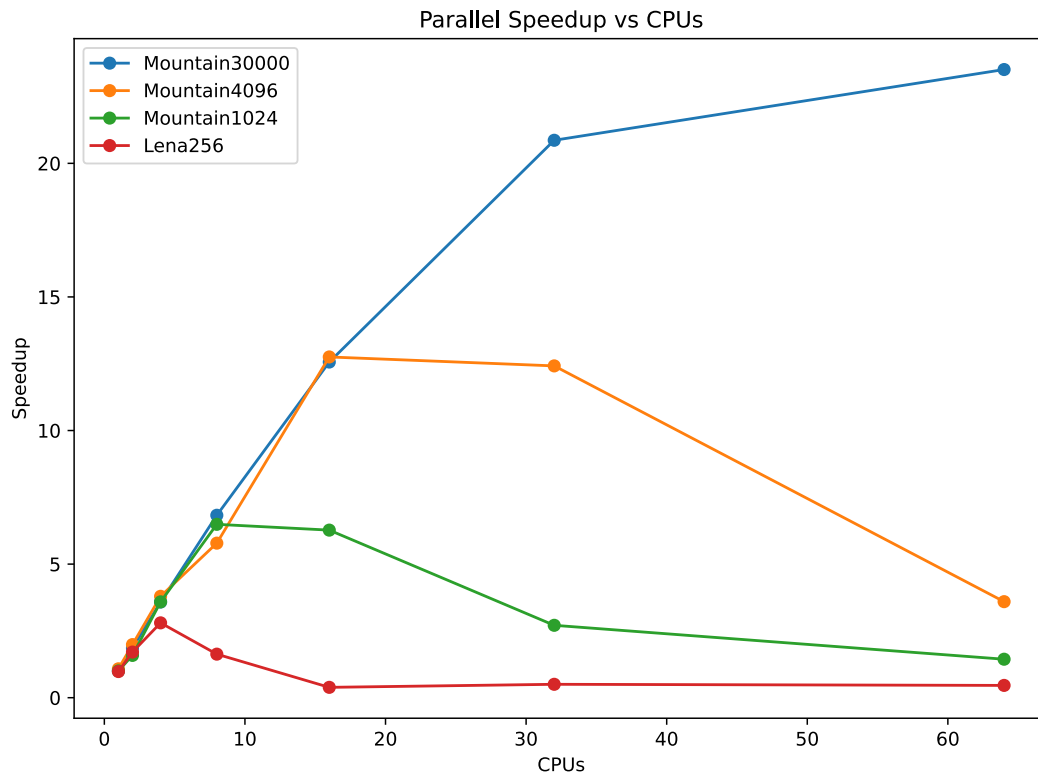
(β') mountain1024.raw with 1 iteration

CPU <sub>s</sub>	T <sub>s</sub>	T <sub>p</sub>	Sp
1	1.3593	1.2575	1.08
2	1.3305	0.6695	1.99
4	1.3311	0.3508	3.79
8	1.3325	0.2304	5.78
16	1.3538	0.1062	12.75
32	1.3457	0.1084	12.42
64	1.3625	0.379	3.59

(γ') mountain4096.raw with 1 iteration

CPU <sub>s</sub>	T <sub>s</sub>	T <sub>p</sub>	Sp
1	60.7031	60.569	1
2	60.99	32.7999	1.86
4	60.825	16.9826	3.58
8	60.7495	8.897	6.83
16	61.1811	4.87	12.56
32	60.8926	2.9187	20.86
64	60.8866	2.5897	23.51

(δ') mountain30000.raw with 1 iteration



Σχήμα 3: Speedup ανα Cores για διαφορετικά μεγέθους προβλήματα

Μπορούμε να παρατηρήσουμε ότι για μικρές εικόνες (μικρό μέγεθος πρόβλημα) το speedup αρχίζει να μειώνεται όσο αυξάνονται οι πυρήνες. Αυτό συμβαίνει γιατί οι γραμμές που παίρνει κάθε πυρήνας μειώνονται πολύ, σε βαθμό που ο έξτρα κώδικας που χρειάζεται ο παράλληλος αλγόριθμος για να κληθούν οι διαφορετικοί πυρήνες και να συγχρονιστούν προκαλούν την μείωση της απόδοσης του. Όσο μεγαλώνουν οι εικόνες (μεγαλώνει το μέγεθος του προβλήματος) ο κάθε πυρήνας έχει περισσότερες γραμμές να επεξεργαστεί οπότε η απόδοση αρχίζει να μειώνεται για μεγαλύτερο πλήθος πυρήνων. Για την εικόνα mountain30000, που έχει πάρα πολλές γραμμές, δεν μπορούμε να δούμε με τους 64 πυρήνες μείωση του speedup.

Σημείωση, ότι τα πειραματικά αποτελέσματα πάρθηκαν ενώ έτρεχε στον rxeon2 ένα άλλο πρόγραμμα που χρησιμοποιούσε σχεδόν 90% του επεξεργαστή, οπότε τα αποτελέσματα θα μπορούσαν να είναι και καλύτερα.

## Αναφορές

- [1] R. Fisher κ.α. Crimmins Speckle Removal. 2003. URL: <https://homepages.inf.ed.ac.uk/rbf/HIPR2/crimmins.htm> (επίσκεψη 14/04/2025).
- [2] Michael Klemm κ.α. Advanced OpenMP Tutorial: NUMA, Vectorization & Tasking. Tutorial presented at IWOMP 2023. 12 Σεπτ. 2023. URL: <https://www.iwomp.org/wp-content/uploads/iwomp-2023-advanced-openmp-tutorial.pdf>.
- [3] Ruud van der Pas. NUMA in OpenMP --- Home Sweet Home. Part I: What is NUMA? Oracle Linux Engineering. 15 Νοέ. 2021. URL: [https://www.openmp.org/wp-content/uploads/OpenMPBoothTalks-SC21-Ruud-NUMA.part\\_.1.pdf](https://www.openmp.org/wp-content/uploads/OpenMPBoothTalks-SC21-Ruud-NUMA.part_.1.pdf).