3ο μέρος εργασίας του μαθήματος "Σχεδιασμός Ενσωματωμένων Συστημάτων:

"Ανιχνευτής ακμών κατά Prewitt και Robert cross στην επεξεργασία εικόνας"

ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Όνομα Καθηγητή: Ιωάννης Βούρκας

Ονόματα φοιτητών: Γεώργιος Τοκατλίδης(58352)

Μιχάλης Τσακίρογλου(58486)

Η/Μ Υποβολής: 10/01/2025

Ομάδα: 27

Πίνακας Περιεχομένων

Εισαγωγή

Για αυτή την εργαστηριακή άσκηση τίθεται το ζήτημα ιεραρχίας μνήμης και τεχνικών επαναχρησιμοποίησης δεδομένων, με σκοπό τη μελέτη της διακύμανσης της απόδοσης του προγράμματος.

Η λογική εξέλιξη του κώδικα βασίστηκε στον σταδιακό εμπλουτισμό της ιεραρχίας, ξεκινώντας από μια πρωτόλεια μορφή του όπου γίνεται χρήση μιας μόνο μεγάλης μνήμης RAM, καταλήγοντας στο ζητούμενο της παρούσας άσκησης που χρησιμοποιούνται δύο ενδιάμεσες μνήμες cache εγγύτερα στον επεξεργαστή, για την αξιοποίηση συχνά χρησιμοποιούμενων δεδομένων με γρηγορότερες ταχύτητες προσπέλασης.

Οι 3 ιεραρχίες που εφαρμόστηκαν είναι οι ακόλουθες:

- Εκδοχή 1: Χρήση RAM, όπως στο πρώτο σκέλος της 2^{ης} εργαστηριακής άσκησης από όπου και πάρθηκε ο αντίστοιχος κώδικας.
- Εκδοχή 2: Χρήση RAM και cache, όπως στο δεύτερο σκέλος της 2^{ης} εργαστηριακής άσκησης από όπου και πάρθηκε ο αντίστοιχος κώδικας.
- Εκδοχή 3: Χρήση RAM και L1, L2 Caches, όπως ορίζεται από την εκφώνηση αυτής της άσκησης. Για αυτή την ιεραρχία υπήρξαν αρκετές εκδοχές κώδικα, ξεκινώντας από πιο απλοϊκές υλοποιήσεις μέχρι και πιο σύνθετες. Περαιτέρω ανάλυσή τους γίνεται στη συνέχεια της αναφοράς.

Ανάλυση Αρχιτεκτονικών

Σύμφωνα με τα παραπάνω λοιπόν, για τις εκδοχές 2 και 3 εισάγονται κάποιοι buffer για την επαναχρησιμοποίηση δεδομένων.

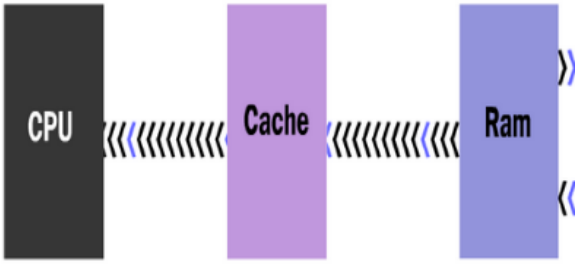
Στην εκδοχή 2, γίνεται χρήση μόνο ενός μικρού block του πίνακα που χρησιμοποιείται για τους gradient υπολογισμούς (και είναι διαφορετικό σε κάθε επανάληψη, με κάποια κοινά στοιχεία), το οποίο και φορτώνεται στη μια και μοναδική cache του συστήματος, η οποία είναι η L1 Cache της επόμενης εφαρμογής.

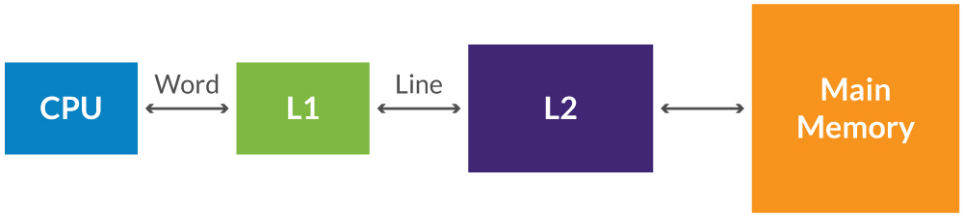
Στην εκδοχή 3, φορτώνονται οι 3 γραμμές που εμπεριέχουν μέσα όλα τα απαραίτητα στοιχεία του πίνακα εισόδου για τον υπολογισμό μιας γραμμής του πίνακα εξόδου, οι οποίες μεταφέρονται από τη RAM στην L2 Cache, όντας και μεγάλες αθροιστικά σε μέγεθος. Στη συνέχεια φορτώνεται το αντίστοιχο block, όπως και στη προηγούμενη εκδοχή, με τη διαφορά ότι η προφόρτωση γίνεται από την L2 αντί για την RAM, με

σκοπό το κέρδος σε ταχύτητα προσπέλασης λόγω της συχνότητας εμφάνισης αυτής της διεργασίας.

Κάθε φορά που υπολογίζεται μια γραμμή του πίνακα εξόδου, επαναλαμβάνεται η διαδικασία φόρτωσης των απαραίτητων δεδομένων από τον πίνακα εισόδου που βρίσκεται στη RAM, στον μεγάλο buffer της L2 που διατηρεί 3 γραμμές του. Σε αυτό το σημείο είναι που διαφέρουν και οι διαφορετικοί αλγόριθμοι που αναπτύχθηκαν.

Παρακάτω φαίνονται τα σχήματα που περιγράφουν τους συνδυασμούς αντιγραφής δεδομένων και τις τοπολογίες των ιεραρχιών που τις συνοδεύουν για τις εκδοχές 2 και 3 αντίστοιχα.





Αλλαγές στον κώδικα

Πριν γίνει η εξήγηση των αλγορίθμων που συγκρίθηκαν υπάρχει μια **σημαντική σημείωση**. Στους αλγορίθμους του 1ου και 2ου εργαστηρίου έγινε ηθελημένη αγνόηση των ελέγχων αλλαγής γραμμών στους εκάστοτε πίνακες που διαβάζονταν - γράφονταν.

Αρχικά, αυτό που συμβαίνει **χωρίς τους ελέγχους αλλαγής γραμμών**, είναι ότι στη τελευταία στήλη της εικόνας εξόδου (όπως και για τη πρώτη και προτελευταία για τον Prewitt), γίνονται υπολογισμοί οι οποίοι θεωρητικά είναι λανθασμένοι. Αυτό οφείλεται στο γεγονός ότι κάποια από τα γειτονικά στοιχεία τους θα βρίσκονταν στην επόμενη στήλη τα οποία και δεν υπάρχουν και αφού είναι σε flat μορφή ο πίνακας. Αυτά τα στοιχεία μεταφράζονται σαν τα πρώτα στοιχεία από επόμενες γραμμές.

Αυτό το καθιστούσαν εφικτό, θεωρητικά και πρακτικά, αρκετά γεγονότα. Η επιλογή απεικόνισης των πινάκων σε **flat μορφή επιτρέπει** πρακτικά να παραληφθεί αυτός ο έλεγχος, χωρίς να υπάρχει η ανησυχία για προσπέλαση δεδομένων εκτός ορίων που θα οδηγούσε σε σφάλματα ή απρόβλεπτη συμπεριφορά του προγράμματος. Επίσης, η εφαρμογή που αναπτύσσεται αφορά edge detection, οπότε και τα όρια μιας εικόνας δεν παρέχουν χρήσιμη πληροφορία για αυτή την εφαρμογή. Έτσι λοιπόν ο συνδυασμός αυτών των δύο κύριων στοιχείων, προτρέπει τον προγραμματιστή να κάνει αυτή την εσκεμμένη παράλειψη, σώζοντας αρκετό overhead από διαδοχικούς ελέγχους εντός του επαναληπτικού βρόχου.

Αυτός ο έλεγχος στη περίπτωση που θέλουμε να φορτώνουμε σωστά δεδομένα στους buffers όμως, είναι πλέον αναπόφευκτος, καθώς πρέπει να είναι γνωστό πότε το πρόγραμμα βρίσκεται στο τέλος της μιας γραμμής του πίνακα εξόδου, ώστε να φορτώσει τα νέα απαραίτητα δεδομένα του πίνακα εισόδου στον μεγάλο buffer και να γίνουν σωστά οι υπολογισμοί. Βέβαια αυτός ο έλεγχος γίνεται αμιγώς για τον καθορισμό της χρονικής στιγμής που θα φορτωθούν τα νέα δεδομένα και δεν εμποδίζεται ο υπολογισμός των στοιχείων της τελευταίας στήλης (και προτελευταίας για τον Prewitt) για τους λόγους που αναλύθηκαν παραπάνω και για να μην περιπλέξει παραπάνω την ήδη υπάρχουσα λογική αλλαγής γραμμής. Αυτό καθίσταται δυνατό και λόγω του μεγέθους unrolling που γίνεται, σε συνδυασμό με κατάλληλο (για το συγκεκριμένο unrolling) padding στον μεγάλο buffer, δηλαδή βάζοντάς του δύο επιπρόσθετα στοιχεία στο τέλος, έτσι ώστε κατά την προσπέλασή του να μην ξεπερνούνται τα όριά του και οδηγείται το πρόγραμμα σε απρόβλεπτη συμπεριφορά. Διαφορετικά θα έπρεπε κάθε φορά να ελέγχεται αν βρισκόμαστε στο τέλος της γραμμής και με βάση το μέγεθος του unrolling να εμποδίζουμε ορισμένους υπολογισμούς και προσπελάσεις να γίνουν σε εκείνη τη φάση του βρόχου. Αυτό είναι ένα ζήτημα που ανακύπτει από τη κλήση του Prewitt και μόνο, καθώς τα στοιχεία του γεμίζουν τον μεγάλο buffer, σε αντίθεση με το Robert Cross που αφήνει τη τελευταία του γραμμή κενή.

```
#pragma arm section zidata = "L2Cache"
unsigned char big_buff[3*width+2];
#pragma arm section
```

- Παρακάτω δείχνεται ο τρόπος που πραγματώνονται τα παραπάνω για τις συναρτήσεις Robert Cross και Prewitt στην αρχική υλοποίηση.

```
// func to apply Prewitt operator
void applyPrewitt() {
    f_id = im_size - idx - 1;
    bigbuffsize = 3 * width;
    idx = width + 1;
    cache_load_rep = 1;
    bbi = width + 1;
    t = 0;
    max_value = 1;
    iic = bbc = oic = sbc = 0;
    //int P1,P2,P3,P4;

    do{

        if(idx % width == 1) {
            t = cache_load_rep * width;
            for (i = 0; i < bigbuffsize; i++) {
                big_buff[i] = input_image[t + i];
                iic += 1;
            }
            cache_load_rep += 1;
            bbi = width + 1;
        }

        small_buff[0] = big_buff[bbi - width - 1];  // I1
        small_buff[1] = big_buff[bbi - width];      // I2
        small_buff[2] = big_buff[bbi - width + 1];  // I3
        small_buff[3] = big_buff[bbi - 1];          // I4
        small_buff[4] = big_buff[bbi];              // I5
        small_buff[5] = big_buff[bbi + 1];          // I6
        small_buff[6] = big_buff[bbi + width - 1];  // I7
        small_buff[7] = big_buff[bbi + width];      // I8
        small_buff[8] = big_buff[bbi + width + 1];  // I9
        small_buff[9] = big_buff[bbi - width + 2];  // I10
        small_buff[10] = big_buff[bbi + 2];         // I11
        small_buff[11] = big_buff[bbi + width + 2]; // I12
        bbi += 2;

        P1 = small_buff[0] + small_buff[1] + small_buff[2] - small_buff[6] - small_buff[7] - small_buff[8];
        P2 = small_buff[0] + small_buff[3] + small_buff[6] - small_buff[2] - small_buff[5] - small_buff[8];
        P3 = small_buff[1] + small_buff[2] + small_buff[9] - small_buff[7] - small_buff[8] - small_buff[11];
        P4 = small_buff[1] + small_buff[4] + small_buff[7] - small_buff[9] - small_buff[10] - small_buff[11];
```

Prewitt first part of the code

```
        outim[idx] = abs(P1) + abs(P2);
        outim[idx + 1] = abs(P3) + abs(P4);

        oic += 2; // because of the if statements(reads)
        sbc += 24; // because of the calcs(reads)
        bbc += 12; // we copy them to small buff

        if (outim[idx] > max_value)
        {
            max_value = outim[idx];
            oic += 1;
        }

        if (outim[idx + 1] > max_value)
        {
            max_value = outim[idx + 1];
            oic += 1;
        }

        idx += 2;

    } while(idx < f_id );

    // now we normalize

    // each pixel to the range [0, 255]
    for (i = 0; i < im_size; i++) {
        output_image[i] = (unsigned char)((outim[i] * 255) / max_value);
        oic += 1;
    }

    printf("In Prewitt Function we access the Input Image matrix: %d times\n", iic);
    printf("In Prewitt Function we access the Output Image matrix: %d times\n", oic);
    printf("In Prewitt Function we access the Big Buff matrix: %d times\n", bbc);
    printf("In Prewitt Function we access the Small Buff matrix: %d times\n", sbc);


}
```

Prewitt second part of the code

```
// func to  apply Roberts Cross operator
void applyRobertsCross() {
    f_id = im_size - width - 4;
    bigbuffsize = 2 * width;
    idx = 0;
    cache_load_rep = 0;
    bbi = 0;
    t = 0;
    max_value = 1;
    iic = bbc = oic = sbc = 0;

    do{

        if(idx % width == 0) {
            t = cache_load_rep * width;
            for (i = 0; i < bigbuffsize; i++) {
                big_buff[i] = input_image[t + i];
                iic += 1;
            }
            cache_load_rep += 1;
            bbi = 0;
        }

        small_buff[0] = big_buff[bbi];          // I1
        small_buff[1] = big_buff[bbi + 1];          // I2
        small_buff[2] = big_buff[bbi + width];      // I3
        small_buff[3] = big_buff[bbi + width + 1];  // I4
        small_buff[4] = big_buff[bbi + 2];          // I5
        small_buff[5] = big_buff[bbi + width + 2];  // I6
        small_buff[6] = big_buff[bbi + 3];          // I7
        small_buff[7] = big_buff[bbi + width + 3];  // I8
        small_buff[8] = big_buff[bbi + 4];          // I9
        small_buff[9] = big_buff[bbi + width + 4];  // I10

        bbi += 4;

        outim[idx] = (abs(small_buff[0]-small_buff[3]) + abs(small_buff[1]-small_buff[2]));
        outim[idx + 1] = (abs(small_buff[1]-small_buff[5]) + abs(small_buff[4]-small_buff[3]));
        outim[idx + 2] = (abs(small_buff[4]-small_buff[7]) + abs(small_buff[6]-small_buff[5]));
        outim[idx + 3] = (abs(small_buff[6]-small_buff[9]) + abs(small_buff[8]-small_buff[7]));
```

Roberts Cross first part of the code

```
        bbc += 10;
        oic += 4; // because of the initializations and the if statements
        sbc += 16;

        if (outim[idx] > max_value)
        {
            max_value = outim[idx];
            oic += 1;
        }
        if (outim[idx + 1] > max_value)
        {
            max_value = outim[idx + 1];
            oic += 1;
        }
        if (outim[idx + 2] > max_value)
        {
            max_value = outim[idx + 2];
            oic += 1;
        }
        if (outim[idx + 3] > max_value)
        {
            max_value = outim[idx + 3];
            oic += 1;
        }

        idx+=4;
    } while(idx < f_id);

    // each pixel to the range [0, 255]
    for (i = 0; i < im_size; i++) {
        output_image[i] = (unsigned char)((outim[i] * 255) / max_value);
        oic += 1;
    }

    printf("In Roberts Function we access the Input Image matrix: %d times\n", iic);
    printf("In Roberts Function we access the Output Image matrix: %d times\n", oic);
    printf("In Roberts Function we access the Big Buff matrix: %d times\n", bbc);
    printf("In Roberts Function we access the Small Buff matrix: %d times\n", sbc);


}
```

Roberts Cross second part of the code

- Όπως φαίνεται από τον κώδικα η συνθήκη ελέγχου ορίζεται από τον αριθμό των στοιχείων που περιέχονται σε μια γραμμή, το μέγεθος του unrolling και τον δείκτη από τον οποίο εκκινείται ο υπολογισμός της νέας γραμμής. Το μέγεθος της γραμμής είναι 640.

- Για τον Robert Cross είναι πιο βολικά τα πράγματα, καθώς η εκκίνηση υπολογισμών σε μια γραμμή γίνεται από το πρώτο στοιχείο (με δείκτη 0). Το unrolling με βαθμό 4 θα οδηγήσει τον δείκτη bbi, που είναι απαραίτητος για να δεικτοδοτείται κατάλληλα ο μεγάλος buffer, μετά από διαδοχικές προσθήκες, στη τιμή 640 που είναι και το μέγεθος μιας γραμμής. Εκείνη τη στιγμή, αφού έχουν προηγηθεί οι υπολογισμοί για τα 4 τελευταία στοιχεία του πίνακα εξόδου, σηματοδοτείται η αλλαγή γραμμής και φορτώνονται τα νέα δεδομένα, ενώ αρχικοποιείται εκ νέου ο bbi. Αυτό γίνεται παίρνοντας το υπόλοιπο της διαίρεσης του idx με το width (το οποίο είναι ακριβώς το ίδιο με το να χρησιμοποιούσαμε τον bbi, καθώς ο idx θα έχει πάντα τιμή idx = bbi + k * width).

- Για τον Prewitt αντίστοιχα, το σημείο εκκίνησης σε κάθε γραμμή είναι το $2^ο$ στοιχείο της $2^ης$ γραμμής. Οπότε ξεκινάει ο καθολικός δείκτης idx από το k * width + 1 και ο bbi αντίστοιχα από το width + 1. Με unrolling βαθμού 2, μετά από αρκετές επαναλήψεις θα φτάσει στη τιμή width + 641. Με αντίστοιχο τρόπο, αφού έχουν προηγηθεί οι υπολογισμοί των 2 τελευταίων στοιχείων ορισμένης γραμμής του πίνακα εξόδου, σηματοδοτείται η αλλαγή γραμμής.

Αυτός ο έλεγχος χρησιμοποιήθηκε σε όλους τους αλγορίθμους που αναπτύχθηκαν στη συνέχεια.

Ανάλυση Αλγορίθμων

Αρχική Υλοποίηση

Τώρα ακολουθεί η εξήγηση των αλγορίθμων που αναπτύχθηκαν για την $3^η$ εκδοχή της ιεραρχίας μνήμης. Σαν κοινός παράγοντας όλων, είναι πως συχνά χρησιμοποιούμενοι δείκτες, μετρητές και μεταβλητές που εμφανίζονται σε κάθε επανάληψη του βρόχου, έγιναν global ZI data και ενσωματώθηκαν στην L1 Cache έτσι ώστε να μην προκαλείται περιττό bottleneck από τόσο μικρά-σε-μέγεθος στοιχεία. Οπότε με κόστος την ανεπαίσθητα μεγαλύτερη L1, κάθε επανάληψη τρέχει γρηγορότερα. Έχουν προστεθεί κάποιες μεταβλητές για τη κατάλληλη δεικτοδότηση των buffer και counters για την αρίθμηση των προσπελάσεων ανάγνωσης πινάκων(εισόδου και buffers), και άλλες για να αποθηκεύουν συχνά επαναχρησιμοποιούμενες πράξεις.

Για αρχή, έγινε μια πιο απλοϊκή προσέγγιση, όπου κάθε φορά που αλλάζει η γραμμή υπολογισμού για τον πίνακα εξόδου, φορτώνονται 3 νέες γραμμές από την RAM. Αυτή είναι η πιο απλή και άμεση μέθοδος, διότι ο προγραμματιστής δεν αξιοποιεί το γεγονός ότι οι 2 τελευταίες κατά σειρά γραμμές από τους προηγούμενους υπολογισμούς είναι ίδιες με τις 2 πρώτες κατά σειρά γραμμές, για τον υπολογισμό της επόμενης γραμμής του πίνακα εξόδου.

Βελτιστοποιημένη Υλοποίηση

Έπειτα, μια υλοποίηση που αξιοποιούσε το προαναφερθέν στοιχείο, ήταν η εξομοίωση μιας FIFO δομής όπου επί της ουσίας, οι 2 τελευταίες γραμμές από τους προηγούμενους υπολογισμούς μεταφέρονται μια γραμμή πάνω – εκτοπίζοντας την πρώτη γραμμή που υπήρχε αρχικά στον buffer, και εισάγοντας στο τέλος της δομής τη νέα γραμμή που θα χρειαστεί. Με αυτήν την τεχνική, διατηρούνταν η σωστή ταξινόμηση των γραμμών – βάση της σειράς τους -, πράγμα απαραίτητο για να περαστούν με τη σωστή σειρά και τα στοιχεία του αντίστοιχου block στον μικρότερο buffer σε κάθε επανάληψη, ενώ το μεγαλύτερο μέρος της μεταφοράς δεδομένων γινόταν εντός της ίδιας μνήμης, δηλαδή της L2.

```
//push last two lines in the first two lines and bring last line from ram
if(idx % width == 1) {
    t = cache_load_rep * width;
    for (i = 0; i < bigbuffsize-width; i++) {
        big_buff[i] = big_buff[i + width];
        bbc += 1;
    }
    for (i = bigbuffsize-width; i < bigbuffsize; i++) {
        big_buff[i] = input_image[t + i];
        iic += 1;
    }
    cache_load_rep += 1;
    bbi = width + 1;
}
```

Υλοποίηση Prewitt

```
//push last two lines in the first two lines and bring last line from ram
if(idx % width == 0) {
    t = cache_load_rep * width;
    for (i = 0; i < bigbuffsize-width; i++) {
        big_buff[i] = big_buff[i + width];
        bbc += 1;
    }
    for (i = bigbuffsize-width; i < bigbuffsize; i++) {
        big_buff[i] = input_image[t + i];
        iic += 1;
    }

    cache_load_rep += 1;
    bbi = 0;
}
```

Υλοποίηση Prewitt

Ενδιάμεσες Υλοποιήσεις

Δημιουργήθηκαν άλλες δύο εκδοχές κώδικα για αυτήν την ιεραρχία, παρόλαυτα οι μετρήσεις που λήφθηκαν δεν παρουσίαζαν ικανοποιητικές βελτιώσεις, όπως οι παραπάνω υλοποιήσεις και δεν συμπεριλήφθηκαν στις τελικές μετρήσεις της αναφοράς, αλλά θα βρίσκονται στο αρχείο zip του παραδοτέου. Μια αλλαγή που δοκιμάστηκε ήταν η προσθήκη ενός buffer που διατηρεί τις τιμές που επρόκειτο να γραφτούν σε ορισμένη γραμμή του πίνακα εξόδου με τη λογική πως κατά το διάστημα που γίνεται μεταφορά δεδομένων από την RAM προς την L2 Cache, θα μπορούσε να γίνεται και μεταφορά δεδομένων κατά την αντίθετη φορά, έτσι ώστε να αποφεύγεται η επανειλημμένη επικοινωνία επεξεργαστή και RAM σε διαδοχικές επαναλήψεις. Από τις μετρήσεις που έγιναν για τους κώδικες που αξιοποιούσαν αυτή τη τεχνική δεν φάνηκαν βελτιώσεις λόγω του αυξημένου overhead, οπότε εν τέλει δεν διατηρήθηκε για τις τελικές μετρήσεις η συγκεκριμένη προσθήκη. Σε άλλη εφαρμογή, ή με άλλες παραμέτρους του συστήματος θα μπορούσε να αποφέρει θετική επίδραση στην ταχύτητα εκτέλεσης, αλλά όπως διαπιστώνεται στη συνέχεια, είναι πολυπαραγοντική η

συνάρτηση της απόδοσης, οπότε αν και εκ πρώτης όψεως φαινόταν συνετή επιλογή, δεν ήταν τελικά χρήσιμη για το συγκεκριμένο πρόβλημα. Η τελευταία αλλαγή, ήταν η υλοποίηση ενός κυκλικού μεγάλου buffer κατά τον οποίο, σε κάθε αλλαγή γραμμής του πίνακα εξόδου, αντικαθίσταται η μικρότερη-κατά-σειρά γραμμή από τη νέα γραμμή. Ένας δείκτης χρησιμοποιούνταν για να αλλάζει κάθε φορά το head της δομής (start_idx), έτσι ώστε να είναι γνωστό ποια είναι η επόμενη προς αντικατάσταση γραμμή, αλλά και η μεταφορά δεδομένων από την L2 στην L1 να γίνεται με τη σωστή σειρά. Για την Prewitt, οπότε ήταν απαραίτητο να γίνει και ο αντίστοιχος υπολογισμός των δεικτών του small_buff με χρήση αυτού του δείκτη (start_idx), ώστε τα στοιχεία του small_buff να πάρουν με τη σωστή σειρά τις τιμές και οι υπολογισμοί να μην αλλοιωθούν. Στον Robert cross αυτό δεν ήταν απαραίτητο, διότι το kernel είναι 2x2 και οι απόλυτες διαφορές που παίρνονται είναι μεταξύ διαγώνιων στοιχείων, οπότε δεν είχε σημασία ποια γραμμή βρισκόταν στον μεγάλο buffer πρώτη.

ΜΕΤΡΗΣΕΙΣ – ΑΠΟΤΕΛΕΣΜΑΤΑ

Τα scatter και memory maps δεν αλλάζουν δραματικά και φυσικά η μεθοδολογία δημιουργίας είναι ίδια με αυτής του 2^{ου} εργαστηρίου. Αυτό που επαναυπολογίζεται είναι το επιπρόσθετο μέγεθος που χρειάζεται για τις μικρές μεταβλητές που χρησιμοποιούνται εντός των συναρτήσεων Robert Cross & Prewitt για να επαναπροσδιοριστεί το μέγεθος της L1 Cache, καθώς και το μέγεθος της L2 που θα φιλοξενεί 3 γραμμές τύπου unsigned char, δηλαδή 3*640 = 1920 bytes και άλλα 2 byte για το padding. Μετατρέποντας αυτά τα μεγέθη σε δεκαεξαδικό σύστημα λαμβάνεται το απαραίτητο μέγεθος για τις μνήμες L2, L1.

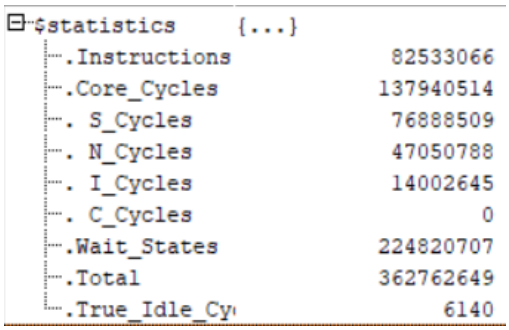
Η ROM προσαρμόζεται ανάλογα με το πόσες εντολές δημιουργούνται από τις νέες τεχνικές ελέγχου, μεταφοράς δεδομένων και υπολογισμών, πράγμα που φαίνεται από το Make εντός του Codewarrior. Φυσικά, οι τιμές που τέθηκαν στο τέλος ήταν προσεγγιστικές για τυχόν βελτιώσεις και προσθήκες που μπορεί να γίνονταν κατά την εκπόνηση της εργασίας.

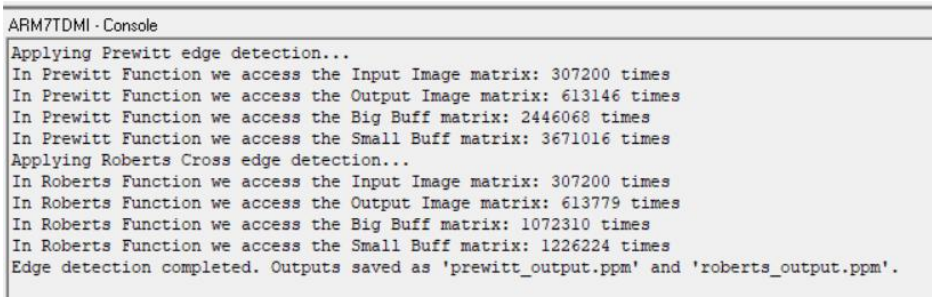
Επίσης, για να μην υπάρξει σύγχυση μεταξύ των scatter και memory maps διαφορετικών εκδοχών κώδικα της ίδιας αρχιτεκτονικής, χρησιμοποιείται το scatter και memory map του πιο απαιτητικού σε μνήμη προγράμματος.

Για τις παρακάτω ταχύτητες βγάλαμε τα εξής αποτελέσματα:

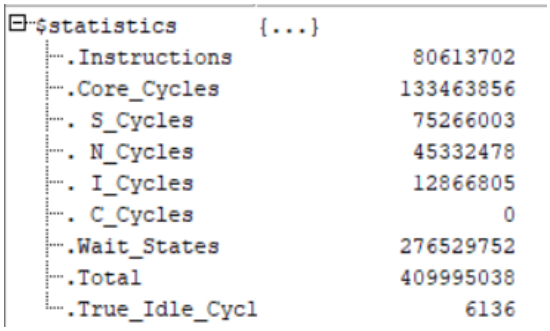
```
ROM 4 R 1/1 1/1
L1Cash 4 RW 20/20 20/20
L2Cash 4 RW 40/40 40/40
RAM 4 RW 1000/500 1000/500
```

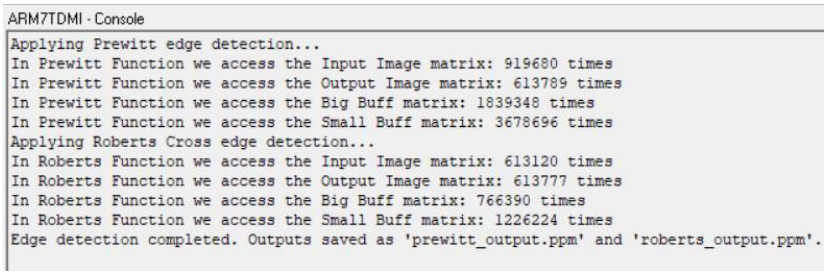
Από το **lab3 codes/final-fifo without out buff/final.c**





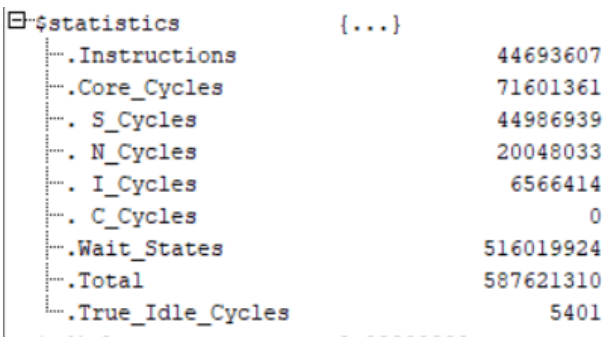
Από το **lab3 codes/initial/lab3.c**

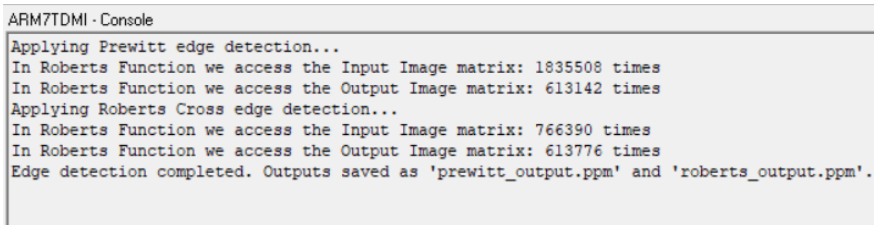




Από το **lab2 codes/best.c**

Ο παρακάτω κώδικα είναι από την δεύτερη εργασία με τις αναφερόμενες αλλαγές στις επαναχρησιμοποιούμενες μεταβλητές. Το memory map είναι διαφορετικό για το καθένα αλλά οι ταχύτητες που χρησιμοποιούμε για την RAM είναι η ίδια.





Από το αρχείο **lab2 codes/best_buff.c**

Σε αυτόν τον κώδικα χρησιμοποιούμε μόνο μια cache η οποία έχει ταχύτητα ίση με την L1Cache

```
$statistics    {...}
  .Instruction          48736131
  .Core_Cycles          80117831
  . S_Cycles            45964667
  . N_Cycles            26888617
  . I_Cycles             7264572
  . C_Cycles                   0
  .Wait_States         291024190
  .Total               371142046
  .True_Idle_C              5553
```

```
ARM7TDMI - Console
Applying Prewitt edge detection...
In Prewitt Function we access the Input Image matrix: 1839348 times
In Prewitt Function we access the Output Image matrix: 613782 times
In Prewitt Function we access the Output Image matrix: 3678696 times
Applying Roberts Cross edge detection...
In Prewitt Function we access the Input Image matrix: 766390 times
In Prewitt Function we access the Output Image matrix: 613776 times
In Prewitt Function we access the Cache buffer matrix: 1226224 times
Edge detection completed. Outputs saved as 'prewitt_output.ppm' and 'roberts_output.ppm'.
```

Στη συνέχεια πραγματοποιήθηκαν μετρήσεις με βελτιωμένη μνήμη RAM ώστε να διαπιστωθεί η επίδραση των δύο buffer σε μια γρήγορη αρχιτεκτονική.

Τα πειράματα γίνανε για τις ακόλουθες ταχύτητες:

```
ROM 4 R 1/1 1/1
L1Cash 4 RW 20/20 20/20
L2Cash 4 RW 40/40 40/40
RAM 4 RW 200/150 200/150
```

Από το **lab3 codes/final-fifo without out buff/final.c**

```
$statistics    {...}
  .Instruction          82533066
  .Core_Cycles         137940514
  . S_Cycles            76888509
  . N_Cycles            47050788
  . I_Cycles            14002645
  . C_Cycles                   0
  .Wait_States          45509867
  .Total               183451809
  .True_Idle_C              6140
```

Από το **lab3 codes/initial/lab3.c**

```
$statistics        {...}
  .Instructions          80613702
  .Core_Cycles          133463856
  . S_Cycles             75266003
  . N_Cycles             45332478
  . I_Cycles             12866805
  . C_Cycles                    0
  .Wait_States           54263937
  .Total                187729223
  .True_Idle_Cycles          6136
```

Από το **lab2 codes/best.c**

$statistics	{...}	
.Instructions		44693607
.Core_Cycles		71601361
. S_Cycles		44986939
. N_Cycles		20048033
. I_Cycles		6566414
. C_Cycles		0
.Wait_States		94868487
.Total		166469873
.True_Idle_Cycles		5401

Από το **lab2 codes/best_buff.c**

$statistics	{...}	
.Instructions		48736131
.Core_Cycles		80117831
. S_Cycles		45964667
. N_Cycles		26888617
. I_Cycles		7264572
. C_Cycles		0
.Wait_States		53545197
.Total		133663053
.True_Idle_Cycles		5553

Συμπεράσματα

Το γενικό συμπέρασμα είναι πως η απόδοση της εφαρμογής εξαρτάται από πολλές αλληλεξαρτώμενες παραμέτρους. Η ταχύτητα του επεξεργαστή, σε συνδυασμό με τις ταχύτητες των μνημών μεμονωμένα, αλλά και οι ταχύτητες των μνημών συγκριτικά μεταξύ τους επηρεάζουν άμεσα το ποια ιεραρχία και ποιες τεχνικές επιφέρουν καλύτερο αποτέλεσμα.

Μια πιο περίπλοκη ιεραρχία σαν αυτή της τρίτης εκδοχής απαιτεί 2 πράγματα, κατά κύριο λόγο, για να φανούν πραγματικές διαφορές και να αξίζει το κόστος και η πολυπλοκότητα. Το overhead εντολών που προκύπτει μεταξύ επικοινωνίας των μνημών και η μεταφορά δεδομένων μεταξύ τους, πρέπει να υπερκαλύπτεται από τη ταχύτητα που λαμβάνεται πίσω. Για να είναι αυτό εφικτό απαιτούνται αρκετά γρήγορες μνήμες cache μεμονωμένα αλλά και συγκριτικά με τη κύρια μνήμη (π.χ. μια RAM) όπως και ένας αρκετά γρήγορος μικροελεγκτής, ώστε να μπορεί να αξιοποιήσει τις παροχές των μικρών και γρήγορων μνημών, δηλαδή να μην υπάρχει bottleneck από μεριάς CPU. Όταν κάτι από αυτά δεν πληρείται, το να κλιμακώνεται η ιεραρχία και να γεμίζει ο χώρος με στάδια διαφορετικών Cache επιφέρει το αντίθετο του επιθυμητού αποτελέσματος.

Πρακτικά, σε αυτήν την εφαρμογή, ο μικροελεγκτής είναι περιορισμένος σε ταχύτητα ρολογιού όπως είναι αναμενόμενο. Αυτό σημαίνει πως, πολύ γρήγορες Cache δε πρόκειται να κάνουν διαφορά για χρόνους προσπέλασης 20ns και κάτω, οπότε ακόμα και να υπήρχαν τέτοιες μνήμες τη διάθεσή μας, το αποτέλεσμα θα ήταν ίδιο και το κόστος θα ήταν αυξημένο. Στη συνέχεια παίζει μεγάλο ρόλο πόσο αργή είναι η RAM, ώστε να καταφύγει κανείς σε λύσεις πολυπλοκότερης ιεραρχίας. Όπως φαίνεται και από τις μετρήσεις, για αρκετά μεγάλους χρόνους προσπέλασης, η Τρίτη εκδοχή της ιεραρχίας επιφέρει θετικά αποτελέσματα. Περαιτέρω, ενώ παρατηρούμε πως αυξάνονται αρκετά οι εντολές(instructions) που εκτελούνται λόγω επιπρόσθετων αντιγραφών τα wait_states μειώνονται αρκετά και ξεπερνάμε αυτό το overhead.

Ενώ για μικρότερους χρόνους προσπέλασης, καθυστερεί την εκτέλεση του προγράμματος. Παρόλα, αυτά στην δεύτερη περίπτωση που η ταχύτητα της μνήμη RAM πήρε τις τιμές 200/150 200/150 παρατηρούμε από τις μετρήσεις πως η πιο γρήγορη εκδοχή ήταν αυτή με τον ένα buffer(lab2 codes/best_buff.c) γεγονός που υποδηλώνει ότι ο κώδικας μας επωφελείται από μία cache αλλά δεν είναι αρκετά πολύπλοκος για να προχωρήσουμε σε παραπάνω ενδιάμεσες μνήμες.

Στη πρωτόλεια μορφή του κώδικα αυτού του εργαστηρίου(lab3 codes/initial/lab3.c) μεταφέρονται χωρίς περαιτέρω αξιοποίηση χαρακτηριστικών του προβλήματος, δεδομένα από τη RAM στην L2 πράγμα που εκ πρώτης όψεως είναι απόλυτα εσφαλμένο. Όμως ένας αργός επεξεργαστής ο οποίος θέλει να γράψει και να διαβάσει στην ίδια μνήμη, όπως γίνεται στην εκδοχή του κώδικα που υλοποιείται μια ψεύδο-FIFO, λόγω dependencies και stalls που παρουσιάζει, καθυστερεί πολύ παραπάνω, από το να μετέφερε «τυφλά» τα δεδομένα (ακόμα και ίδια να είναι τα περισσότερα), από τη μια μνήμη στην άλλη.

Άρα ακόμα και εντός της ίδιας ιεραρχίας, πιο σύνθετος κώδικας δεν σημαίνει απαραίτητα πιο γρήγορος.