



ΔΗΜΟΚΡΙΤΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΡΑΚΗΣ

ΤΜΗΜΑ
ΗΜ&ΜΥ

Υλοποίηση μετασχηματισμού απόστασης στην επεξεργασία εικόνας

Εξαμηνιαία εργασία του μαθήματος
Σχεδιασμός Ενσωματωμένων Συστημάτων

Σιδέρης Πέτρος ΑΜ: 58341
Ταχτατζής Αναστάσιος ΑΜ: 58439

Ξάνθη, Νοέμβριος 2024

Περιεχόμενα

Περιγραφή προβλήματος.....	3
Μέρος 1ο - Μέθοδοι Βελτιστοποίησης.....	5
1. Αρχική υλοποίηση.....	5
2. Χρήση memcpy για την αντιγραφή των πινάκων.....	10
3. Loop Collapsing.....	11
4. Loop Unrolling.....	13
5. Loop Tiling.....	15
6. Specifying a global variable to a specific CPU register.....	17
7. Event driven execution using labels and array Lookup Table.....	19
Σύγκριση διαφορετικών τεχνικών βελτιστοποίησης.....	21
Μέγεθος πίνακα δεδομένων και αριθμός προσπελάσεων.....	22
Μέρος 2ο - Μνήμη.....	23
Αρχιτεκτονική 1 - ROM/RAM BUS = 4 (4x8 bits = 32 bits bus width).....	25
Αρχιτεκτονική 2 - ROM/RAM BUS = 2 (2x8 bits = 16 bits bus width).....	25
Αρχιτεκτονική 3 - ROM/RAM HIGH SPEED (same bus width, faster ram).....	26
Σύγκριση αρχιτεκτονικών μνήμης.....	26
Υλοποιήσεις με χρήση buffer.....	27
Πρώτη υλοποίηση - Χρήση προσωρινού buffer για διάκριση αντικειμένων.....	28
Αρχιτεκτονική 1 - ROM/RAM BUS = 4 (4x8 bits = 32 bits bus width).....	29
Αρχιτεκτονική 2 - ROM/RAM BUS = 2 (2x8 bits = 16 bits bus width).....	29
Αρχιτεκτονική 3 - ROM/RAM HIGH SPEED (same bus width, faster ram).....	30
Δεύτερη υλοποίηση - Χρήση προσωρινού buffer σε όλο τον αλγόριθμο.....	31
Σύγκριση αρχιτεκτονικών στην υλοποίηση με χρήση buffer.....	32
Μέρος 3ο - Εφαρμογή επαναχρησιμοποίησης δεδομένων.....	33
Ιεραρχία μνήμης δύο επιπέδων.....	33
L1 cache.....	34
L2 cache.....	34
Μετασχηματισμοί επαναχρησιμοποίησης δεδομένων.....	35
Εικόνα 33: Η δεύτερη ακολουθία αντιγραφής δεδομένων (current_line).....	36
Προτεινόμενη ιεραρχία.....	37
Υλοποίηση μετασχηματισμών επαναχρησιμοποίησης δεδομένων.....	38
Υλοποίηση με χρήση του πίνακα current_line.....	38
Υλοποίηση με χρήση του πίνακα block.....	40
Σύγκριση των αποτελεσμάτων.....	42
Βιβλιογραφικές αναφορές.....	43

Περιγραφή προβλήματος

Ο κώδικας που έπρεπε να υλοποιήσουμε σχετιζόταν με την εφαρμογή ενός μετασχηματισμού απόστασης στην επεξεργασία δυαδικής (binary) εικόνας. Γνωρίζοντας ότι δεν υπάρχει ένας και μοναδικός τρόπος για τον προσδιορισμό της απόστασης στις ψηφιακές εικόνες, καθώς και παρατηρώντας το αποτέλεσμα του προτεινόμενου αλγορίθμου καταλήξαμε στο συμπέρασμα ότι καλούμαστε να χρησιμοποιήσουμε την μέθοδο της Οικοδομικής Απόστασης (City-block ή Manhattan distance ή Minkowski first order metric).

Σύμφωνα με την γεωμετρία στην οποία υπάγεται, οι κινήσεις περιορίζονται σε οριζόντιες και κατακόρυφες διαδρομές, όπως ακριβώς συμβαίνει σε ένα ορθογώνιο πλέγμα. Η απόσταση εκφράζεται από το άθροισμα των απόλυτων διαφορών μεταξύ των δύο διαστάσεων. Αξιοσημείωτο είναι το γεγονός ότι παρόλο που παρουσιάζει μικρότερη υπολογιστική πολυπλοκότητα έναντι της δημοφιλέστερης και πιο ακριβής Ευκλείδειας απόστασης, στις περισσότερες των περιπτώσεων δίνει παρόμοια αποτελέσματα.



Εικόνα 1: Σύγκριση των μεθόδων μετασχηματισμού απόστασης.

Περιγραφή του αλγορίθμου:

Ο αλγόριθμος μετασχηματισμού απόστασης, όπως ορίζεται στη βιβλιογραφία που μας δόθηκε, λειτουργεί επαναληπτικά και μπορεί να περιγραφεί ως εξής:

Βήμα 1: Αρχικοποίηση

- Ορίζουμε αρχικά μια μεταβλητή $k = 1$.
- Εντοπίζουμε όλα τα εικονοστοιχεία του αντικειμένου που βρίσκονται σε απόσταση μικρότερη ή ίση με k από τα εικονοστοιχεία του φόντου.
- Θέτουμε την τιμή αυτών των εικονοστοιχείων ίση με k .
- Αν όλα τα εικονοστοιχεία έχουν υποστεί μετασχηματισμό (δηλαδή, καλύπτονται όλες οι περιοχές της εικόνας), προχωράμε στο Βήμα 3.

Βήμα 2: Αύξηση επανάληψης

- Αυξάνουμε την τιμή του k κατά 1: $k = k + 1$
- Επαναλαμβάνουμε τη διαδικασία από το Βήμα 1 για τα υπόλοιπα εικονοστοιχεία.

Βήμα 3: Τερματισμός

- Ο αλγόριθμος τερματίζεται όταν όλα τα εικονοστοιχεία έχουν μετασχηματιστεί, δηλαδή όταν όλες οι αποστάσεις έχουν υπολογιστεί.

```

#include "opts.h" /* Περιλαμβάνει τις απαιτούμενες δηλώσεις (N, M, current_y) */

void distance_transformer() {
    int input[N][M];          /* Πίνακας για αντιγραφή των δεδομένων εισόδου */
    int distance[N][M];        /* Πίνακας για τον υπολογισμό αποστάσεων */
    int transformed = 0;        /* Έγινε μετασχηματισμός κατά τη διάρκεια της επανάληψης; */
    int k = 1;                 /* Μεταβλητή για την τρέχουσα απόσταση */
    int i = 0, j = 0;          /* Μεταβλητές για τους δείκτες επανάληψης */

    /* Αντιγραφή του πίνακα `current_y` στον προσωρινό πίνακα `input` */
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            input[i][j] = current_y[i][j];
        }
    }

    /* Αρχικοποίηση του πίνακα `distance` για διαχωρισμό αντικειμένων και φόντου */
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            if (input[i][j] == 0) {
                distance[i][j] = 0; /* Τα αντικείμενα έχουν αρχική απόσταση 0 */
            } else {
                distance[i][j] = -1; /* Τα υπόλοιπα pixels ορίζονται ως μη επισκεπτόμενα */
            }
        }
    }

    /* Υπολογισμός αποστάσεων με χρήση γειτονικών pixels. */
    for (;;) {
        transformed = 0; /* Επαναφορά του flag αλλαγής */

        /* Έλεγχος κάθε pixel για ενημέρωση των γειτόνων του */
        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                if (distance[i][j] == k - 1) { /* Εξετάζεται η τρέχουσα απόσταση */
                    if (i > 0 && distance[i - 1][j] == -1) { /* Πάνω */
                        distance[i - 1][j] = k;
                        transformed = 1;
                    }
                    if (i < N - 1 && distance[i + 1][j] == -1) { /* Κάτω */
                        distance[i + 1][j] = k;
                        transformed = 1;
                    }
                    if (j > 0 && distance[i][j - 1] == -1) { /* Αριστερά */
                        distance[i][j - 1] = k;
                        transformed = 1;
                    }
                    if (j < M - 1 && distance[i][j + 1] == -1) { /* Δεξιά */
                        distance[i][j + 1] = k;
                        transformed = 1;
                    }
                }
            }
        }

        /* Αν δεν έγινε αλλαγή, τερματίζουμε τη διαδικασία */
        if (!transformed) {
            break;
        }

        /* Αύξηση της απόστασης κατά 1 (ανώτατο όριο το 255) */
        if (k < 255) {
            k++;
        }
    }

    /* Ενημέρωση του πίνακα `current_y` με τα υπολογισμένα αποτελέσματα */
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            current_y[i][j] = distance[i][j];
        }
    }
}

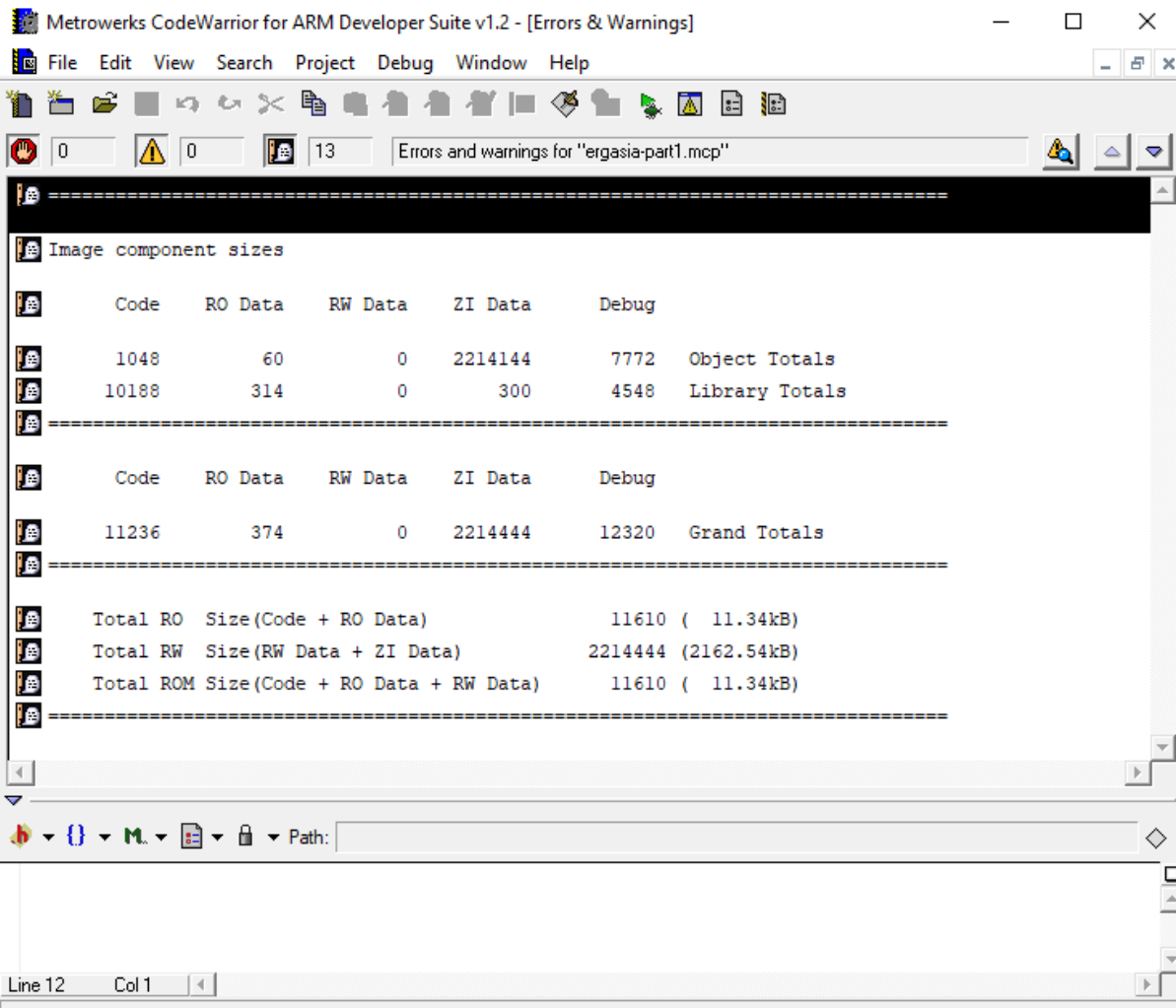
```

Μέρος 1ο - Μέθοδοι Βελτιστοποίησης

Για την υλοποίηση των βελτιστοποιήσεων, εφαρμόσαμε μία αλλαγή ανά optimization και, στο τέλος, συνδυάσαμε συγκεντρωτικά τις καλύτερες από αυτές. Οι μετρήσεις έλαβαν χώρα υπό τις εξής συνθήκες: χρησιμοποιήθηκε το προεπιλεγμένο επίπεδο βελτιστοποίησης -O1 του CodeWarrior C Compiler, δουλεύοντας στην ίδια δυαδική εικόνα διαστάσεων 496x372 (Images/binary/cherry_496x372_BIN.yuv).

1. Αρχική υλοποίηση

Αρχικά, αντιγράφουμε τα bytes της εικόνας εισόδου στον προσωρινό buffer input. Στη συνέχεια, αρχικοποιούμε τον πίνακα distance ώστε να διαχωρίζει τα pixels του αντικειμένου από του background. Με διαδοχικές επαναλήψεις, συγκρίνουμε κάθε pixel με τους γείτονες του, και το ενημερώνουμε σύμφωνα με την απόσταση από το πλησιέστερο pixel του αντικειμένου (βάθος k). Η διαδικασία επαναλαμβάνεται μέχρι να σταματήσουν οι ενημερώσεις (όλα τα pixels έχουν μετασχηματιστεί).



	Code	RO Data	RW Data	ZI Data	Debug	
	1048	60	0	2214144	7772	Object Totals
	10188	314	0	300	4548	Library Totals
	Code	RO Data	RW Data	ZI Data	Debug	
	11236	374	0	2214444	12320	Grand Totals
	Total RO	Size(Code + RO Data)			11610	(11.34kB)
	Total RW	Size(RW Data + ZI Data)			2214444	(2162.54kB)
	Total ROM	Size(Code + RO Data + RW Data)			11610	(11.34kB)

Εικόνα 2: Μεγέθη στοιχείων του generated firmware image.

Debugger Internals							
Debugger Internals							
Reference P...	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
\$statistics	320481178	635583431	418381257	164600466	52601708	0	635583431

Εικόνα 3: Αποτελέσματα αρχικής υλοποίησης.

Για την αρχική μας υλοποίηση του αλγορίθμου προτιμήσαμε να χρησιμοποιήσουμε μία naïve προσέγγιση επιλέγοντας να αγνοήσουμε γνωστά Optimization Good Practises όπως η χρήση της δημοφιλούς και thread-safe συνάρτησης memcpy για την αντιγραφή ενός συνεχόμενου chunk μνήμης από το ένα μέρος στο άλλο, αντ' αυτού επιλέγοντας την χρήση 2 nested for loops για την αντιγραφή των input image bytes σε κάποιον προσωρινό buffer:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        input[i][j] = current_y[i][j];
    }
}
```

Στην συνέχεια, αρχικοποιούμε τον πίνακα distance, όπου σκοπός του για αρχή είναι να διαχωρίζει τα pixels του αντικειμένου από του εκείνα του background:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        if (input[i][j] == 0) {
            distance[i][j] = 0; /* Object pixel has 0 distance */
        } else {
            distance[i][j] = -1; /* -1 for background pixels */
        }
    }
}
```

Προχωρώντας τώρα στην κύρια επανάληψη ενημέρωσης του αλγορίθμου μας, η λογική επικεντρώνεται στη διάδοση της πληροφορίας από τα pixels του αντικειμένου προς το υπόλοιπο πεδίο. Ο στόχος είναι να επεκτείνουμε τις τιμές από τα αντικείμενα (τιμές 0) στα γειτονικά background pixels (-1), αυξάνοντας σταδιακά την τιμή της απόστασης, k . Σε κάθε βήμα του βρόχου, εξετάζουμε όλα τα pixels του πίνακα distance. Αν κάποιο pixel έχει τιμή $k - 1$, τότε θεωρείται ενεργό για το τρέχον βήμα, καθώς περιβάλλει το "μέτωπο" της επέκτασης. Στη συνέχεια, ελέγχουμε όλους τους γείτονές του (πάνω, κάτω, αριστερά, δεξιά). Αν κάποιος γείτονας έχει τιμή -1 (δηλαδή δεν έχει ακόμη ενημερωθεί), τον ενημερώνουμε με την τρέχουσα τιμή k , που αντιπροσωπεύει την απόσταση από το αντικείμενο.

Σε αυτήν την περίπτωση, θέτουμε *transformed* = 1, υποδεικνύοντας ότι έγιναν αλλαγές κατά το τρέχον βήμα. Μετά την επεξεργασία όλων των pixels, ελέγχουμε αν έγιναν αλλαγές (δηλαδή εάν *transformed* = 1). Αν δεν υπήρξε καμία αλλαγή, σημαίνει ότι όλες οι δυνατές αποστάσεις έχουν υπολογιστεί, και ο βρόχος τερματίζεται. Διαφορετικά, αυξάνουμε την τιμή της απόστασης k κατά 1 και επαναλαμβάνουμε τη διαδικασία.

```

for (;;) {
    transformed = 0;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            if (distance[i][j] == k - 1) {
                if (i > 0 && distance[i - 1][j] == -1) {
                    distance[i - 1][j] = k;
                    transformed = 1;
                }

                if (i < N - 1 && distance[i + 1][j] == -1) {
                    distance[i + 1][j] = k;
                    transformed = 1;
                }

                if (j > 0 && distance[i][j - 1] == -1) {
                    distance[i][j - 1] = k;
                    transformed = 1;
                }

                if (j < M - 1 && distance[i][j + 1] == -1) {
                    distance[i][j + 1] = k;
                    transformed = 1;
                }
            }
        }
    }

    if (!transformed)
        break;

    if (k < 255)
        k++;
}

```

Τέλος, αντιγράφουμε τις υπολογισμένες αποστάσεις από τον πίνακα distance πίσω στον αρχικό πίνακα current_y:

```
for (i = 0; i < N; i++) {  
    for (j = 0; j < M; j++) {  
        current_y[i][j] = distance[i][j];  
    }  
}
```

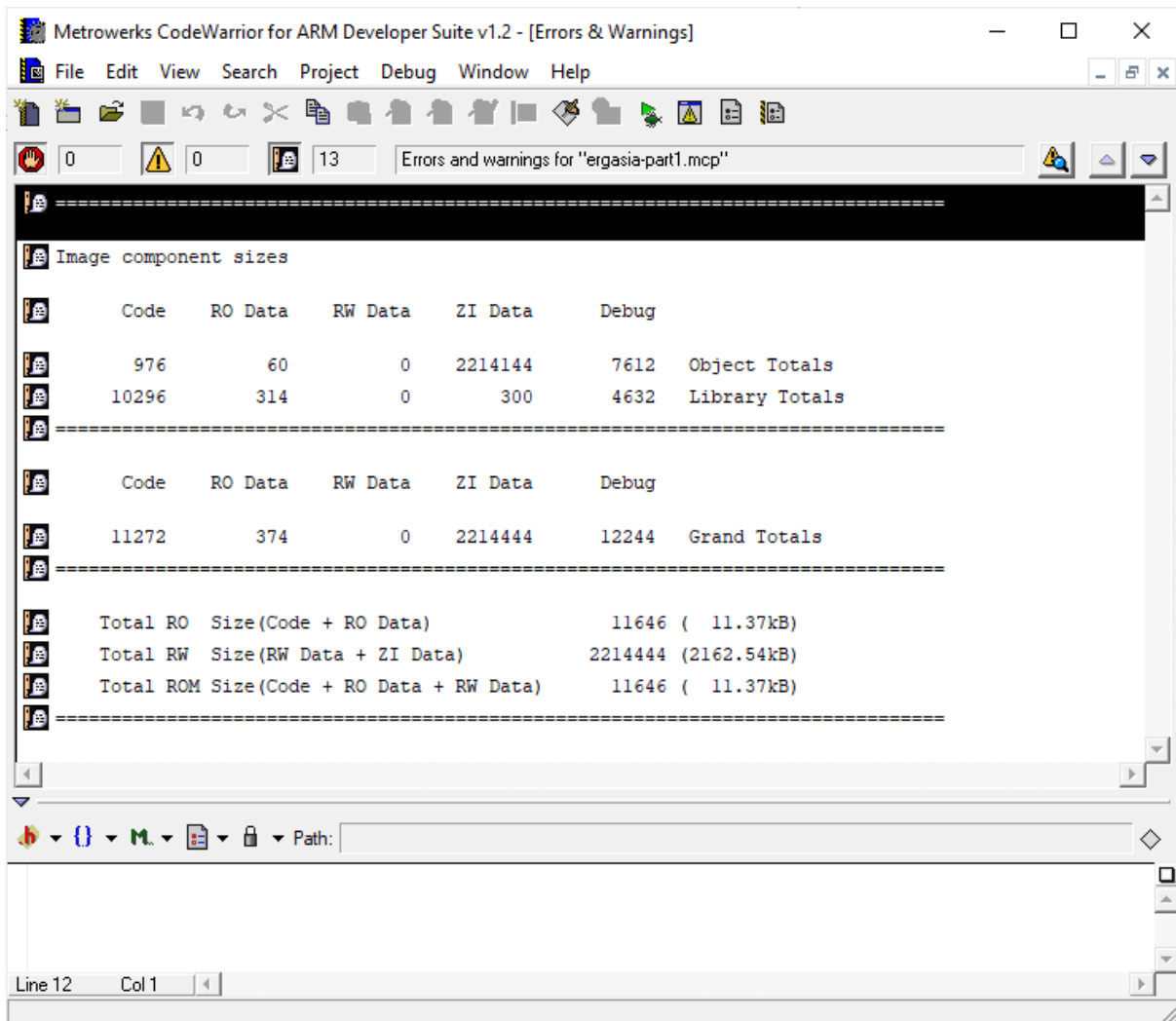
Στην προσπάθεια μας να διαμορφώσουμε ένα σωστό baseline για τα μετέπειτα benchmark που θέλαμε να τρέξουμε, αποφασίσαμε να μην παραμετροποιήσουμε τον C Compiler του CodeWarrior για παραπάνω Optimization από μεριάς του (π.χ. χρησιμοποιώντας το -O2 Flag, το ντε φάκτο επίπεδο βελτιστοποίησης καθώς προσφέρει ισορροπία μεταξύ ταχύτητας, μεγέθους κώδικα και σταθερότητας).

Σε αυτό το σημείο, θα θέλαμε να εξηγήσουμε τη σημασία των πεδίων στη δεύτερη φωτογραφία, τα οποία παρουσιάζουν τα αποτελέσματα της αρχικής υλοποίησης:

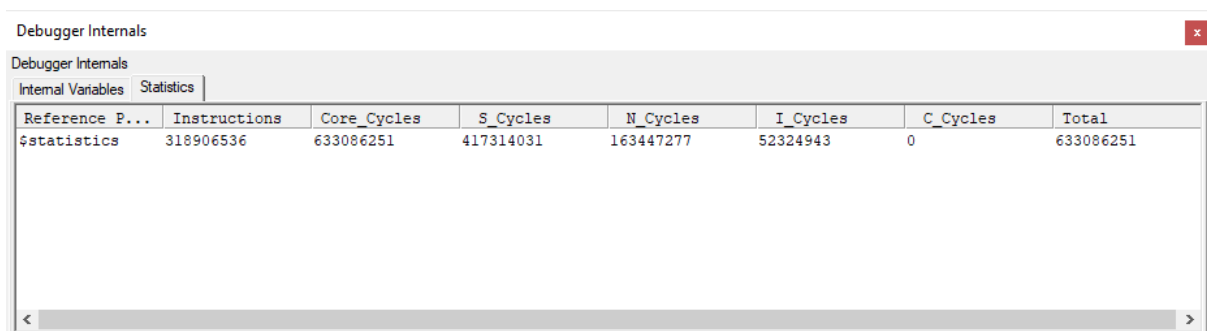
1. **Instructions:** Συνολικός αριθμός εντολών που εκτελέστηκαν κατά τη διάρκεια της προσομοίωσης. Οι ARM επεξεργαστές, ως αρχιτεκτονικές RISC, εκτελούν πολλές εντολές γρήγορα λόγω της απλοποιημένης σχεδίασής τους.
2. **Core Cycles:** Αριθμός κύκλων του επεξεργαστή για την εκτέλεση όλων των εντολών. Περιλαμβάνει όλους τους τύπους κύκλων (stall, normal, idle).
3. **Stall (S) Cycles:** Κύκλοι καθυστερήσεων, όπως αναμονές για μνήμη ή εξαρτήσεις δεδομένων.
4. **Normal (N) Cycles:** Κύκλοι κανονικής λειτουργίας χωρίς καθυστερήσεις.
5. **Idle (I) Cycles:** Κύκλοι όπου ο επεξεργαστής είναι αδρανής, π.χ., λόγω απουσίας εντολών ή αναμονής εξωτερικών γεγονότων.
6. **Coprocessor Cycles (C):** Κύκλοι εκτέλεσης σε συνεπεξεργαστή, που στα πλαίσια του πρώτου μέρους της εργασίας απουσιάζουν.

2. Χρήση memcry για την αντιγραφή των πινάκων

Η παρακάτω υλοποίηση λειτουργεί στο ίδιο πλαίσιο με την αρχική, ωστόσο η ειδοποίηση διαφορά είναι η χρήση της συνάρτησης memcry (απ' το string.h header file) για την αντιγραφή και την μετακίνηση συνεχόμενου block της μνήμης απ' το ένα μέρος σε άλλο. Ωστόσο, η συγκεκριμένη προσέγγιση έρχεται με σημαντικά μειονεκτήματα όπως η αύξηση μεγέθους του παραγόμενου image και η μείωση της φορητότητας του προγράμματος μας.



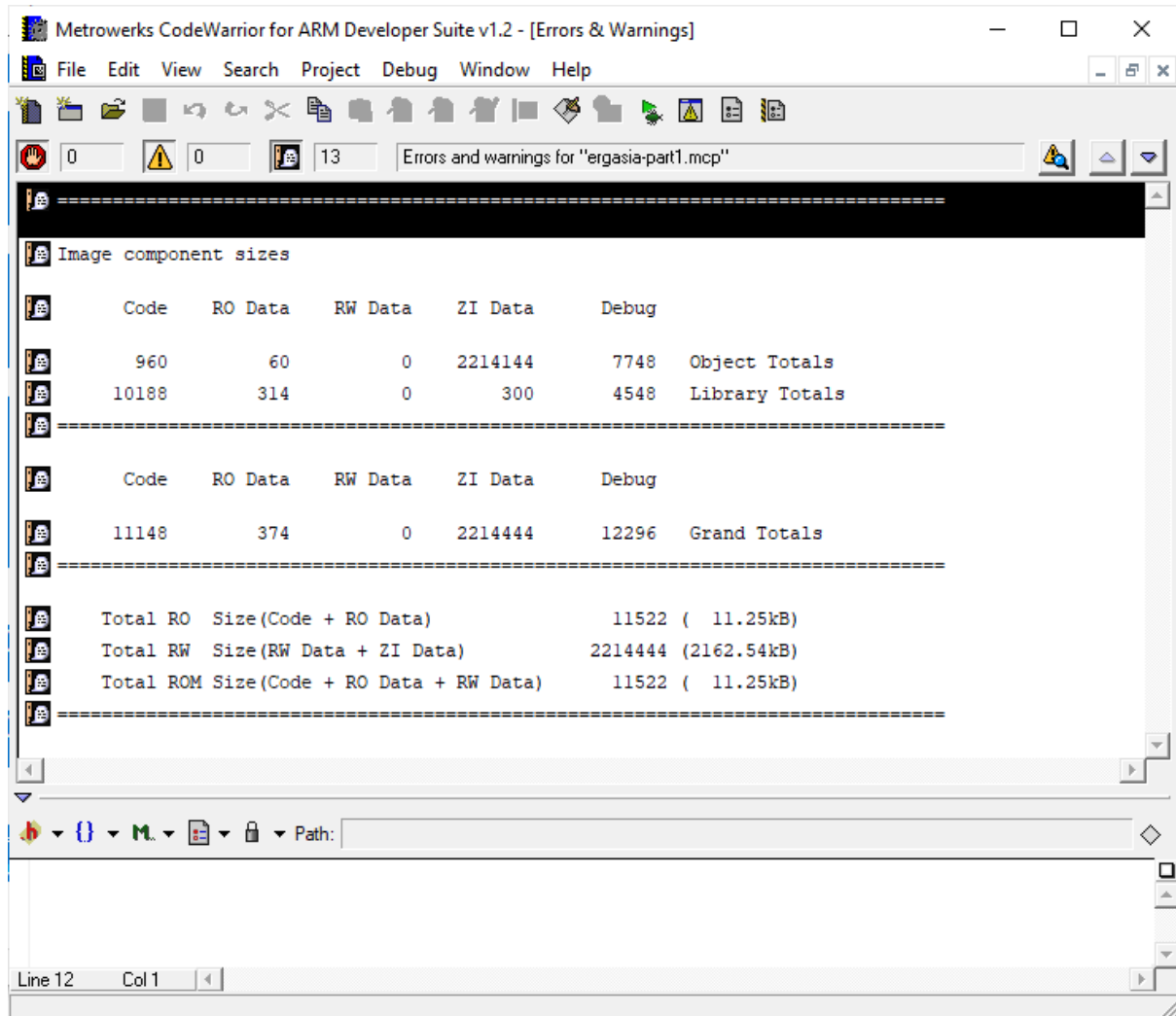
Εικόνα 4: Μεγέθη στοιχείων του generated firmware image.



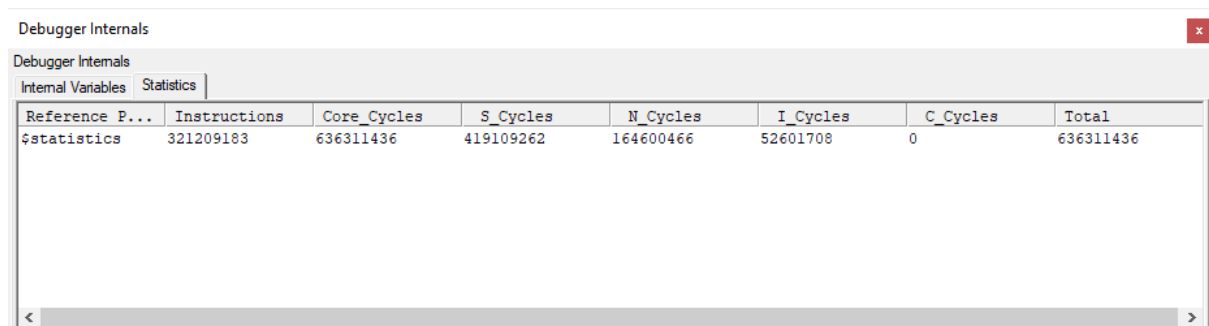
Εικόνα 5: Αποτελέσματα δεύτερης υλοποίησης με memcry.

3. Loop Collapsing

Αντίθετα με την προηγούμενη υλοποίηση, η αντιγραφή και η μετακίνηση συνεχόμενων block της μνήμης γίνεται με την χρήση pointer arithmetics.



Εικόνα 6: Μεγέθη στοιχείων του generated firmware image.



Εικόνα 7: Αποτελέσματα τρίτης υλοποίησης με loop collapsing

Πιο συγκεκριμένα, για την αντιγραφή των input image bytes στον προσωρινό buffer input έχουμε:

```
int *input_ptr = &input[0][0]; /* Get first element's address */
int *current_y_ptr = &current_y[0][0];
int *distance_ptr = &distance[0][0];

for (i = 0; i < N * M; i++) {
    *(input_ptr + i) = *(current_y_ptr + i); /* Dereference it */
}

for (i = 0; i < N * M; i++) {
    *(distance_ptr + i) = (*(input_ptr + i) == 0) ? 0 : -1;
}
```

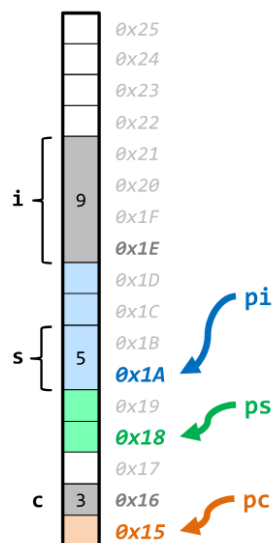
Με αυτόν τον τρόπο εξαλείφουμε την ανάγκη για nested loops, αύξοντας ωστόσο την επιφάνεια λάθους (π.χ. κάνοντας Out of Bounds access, οδηγώντας σε Undefined Behaviour, χωρίς να έχουμε τρόπο εντοπισμού στο compile time παρα μόνο στο runtime) καθώς και την μείωση της αναγνωσιμότητας/συντηρησιμότητας του προγράμματος μας.

Pointer Arithmetic: Decrement by 1

```
char c = 3;
short s = 5;
int i = 9;

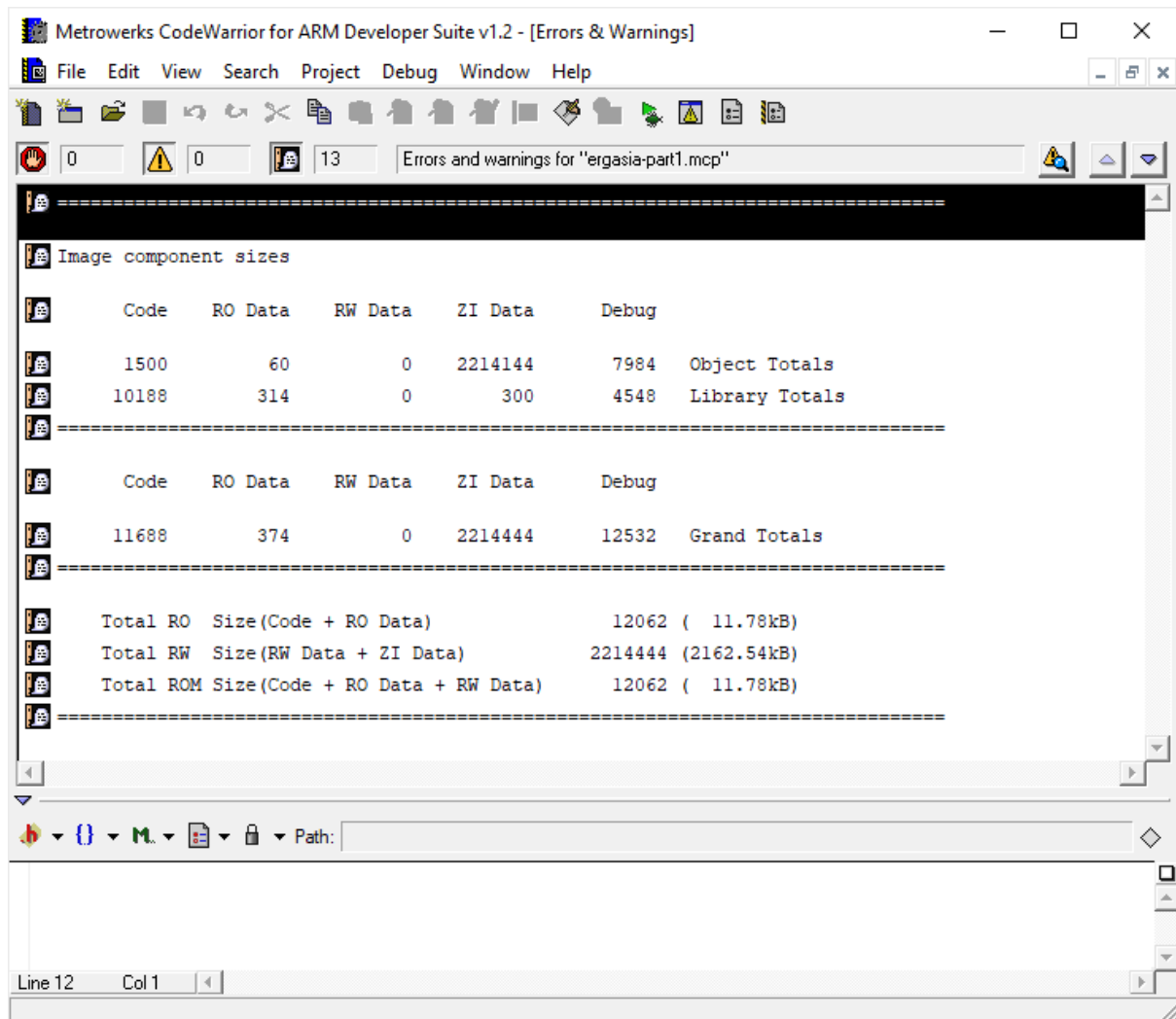
char* pc = &c;
short* ps = &s;
int* pi = &i;

pc--;
ps--;
pi--;
```

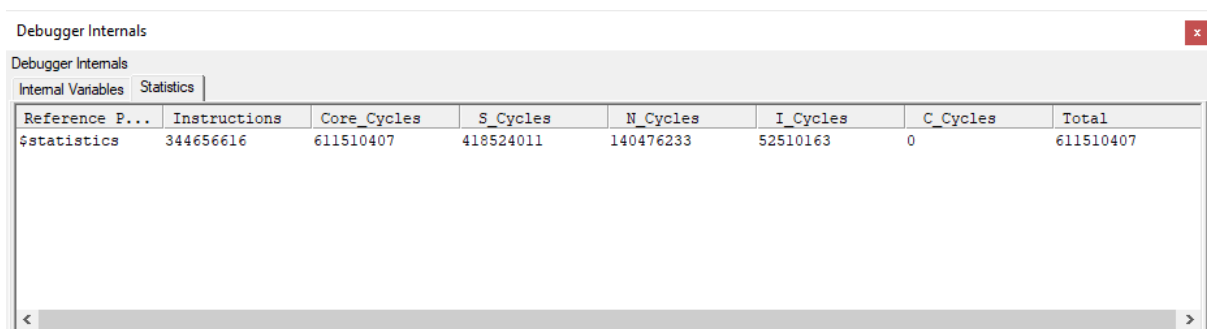


Εικόνα 8: Pointer arithmetics στην γλώσσα C.

4. Loop Unrolling



Εικόνα 9: Μεγέθη στοιχείων του generated firmware image.



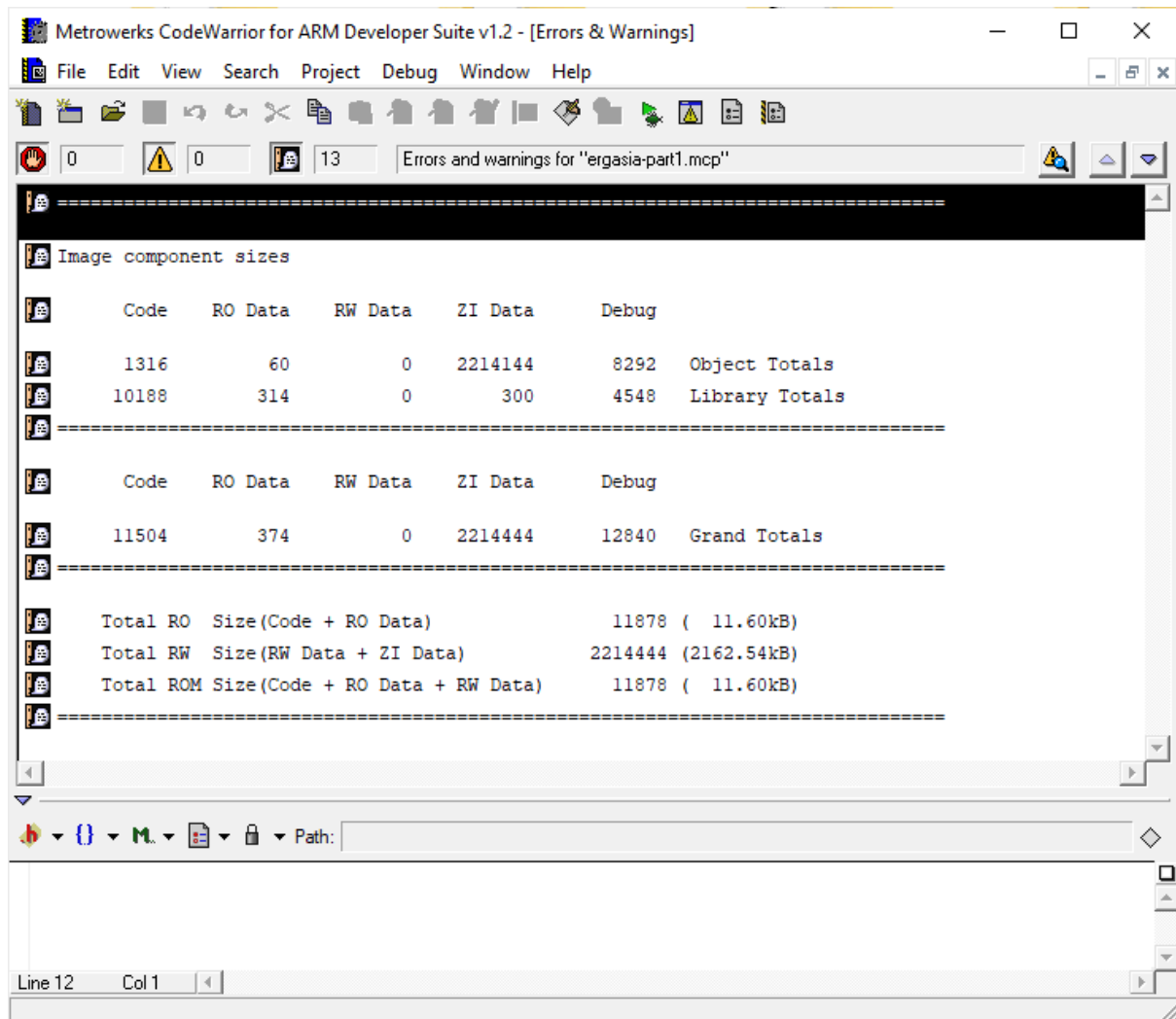
Εικόνα 10: Αποτελέσματα τέταρτης υλοποίησης με handrolled partial loop unrolling.

Στην παραπάνω υλοποίηση, κάναμε χρήση της τεχνικής του partial loop unrolling, η οποία στοχεύει στη μείωση του overhead από τον έλεγχο των συνθηκών και την ενημέρωση του δείκτη επανάληψης σε κάθε βήμα του loop. Αυτή η μέθοδος επιτρέπει την εκτέλεση πολλαπλών επαναλήψεων σε μία μόνο βηματική διαδικασία, ενισχύοντας έτσι την απόδοση. Για παράδειγμα, στην αντιγραφή των byte της εικόνας εισόδου (input image bytes) στον προσωρινό buffer input, τροποποιήσαμε το loop ως εξής:

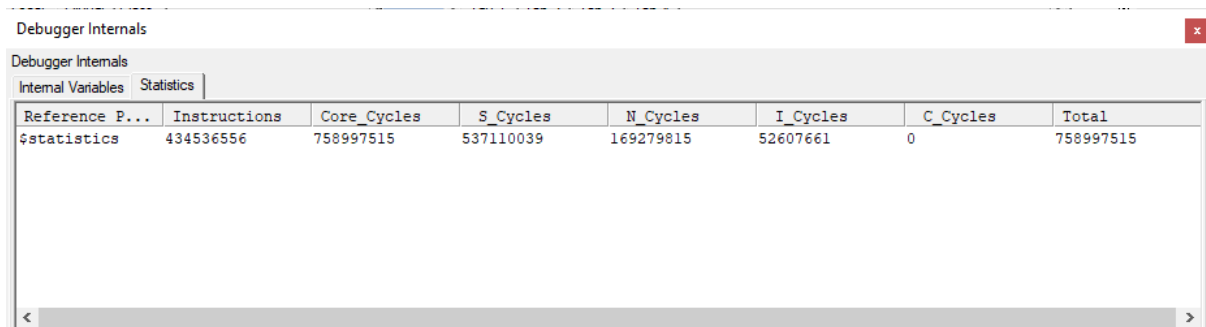
```
for (i = 0; i < N; i++) {  
    for (j = 0; j < M; j += 4) {  
        input[i][j] = current_y[i][j];  
        if (j + 1 < M) input[i][j + 1] = current_y[i][j + 1];  
        if (j + 2 < M) input[i][j + 2] = current_y[i][j + 2];  
        if (j + 3 < M) input[i][j + 3] = current_y[i][j + 3];  
    }  
}
```

Με αυτόν τον τρόπο μειώνουμε το overhead της διαχείρισης του loop (όπως τον έλεγχο συνθηκών και την ενημέρωση των μετρητών που αναγκάζονται να εκτελούνται σε κάθε επανάληψη της for loop), επιταχύνοντας την εκτέλεση, αυξάνοντας ωστόσο δραματικά τις απαιτήσεις μνήμης του προγράμματος.

5. Loop Tiling



Εικόνα 12: Μεγέθη στοιχείων του generated firmware image.



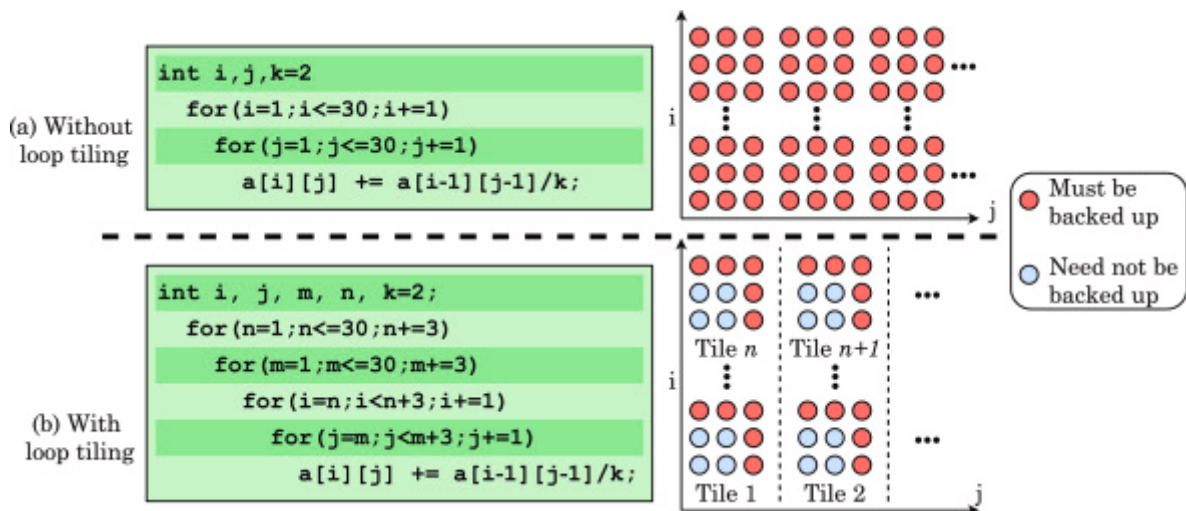
Εικόνα 13: Αποτελέσματα πέμπτης υλοποίησης με loop tiling (tile_size = 32).

Στην παραπάνω υλοποίηση, εφαρμόσαμε τη μέθοδο του tile-based processing, δηλαδή χωρίσαμε τα μεγάλα loops σε μικρότερα "πλακάκια" (tiles), προκειμένου να βελτιστοποιήσουμε την εκμετάλλευση του Cache Locality. Αυτή η προσέγγιση επιτρέπει στον αλγόριθμο να εκτελείται πιο αποδοτικά, καθώς μικρότερα τμήματα δεδομένων παραμένουν στη μνήμη cache κατά την επεξεργασία. Για παράδειγμα, για την αντιγραφή των byte της εικόνας εισόδου (input image bytes) στον προσωρινό buffer input, μετασχηματίσαμε τα for loops ως εξής:

```
int tileSize = 32;

for (i = 0; i < N; i += tileSize) {
    for (j = 0; j < M; j += tileSize) {
        for (ii = i; ii < i + tileSize && ii < N; ++ii) {
            for (jj = j; jj < j + tileSize && jj < M; ++jj) {
                input[ii][jj] = current_y[ii][jj];
            }
        }
    }
}
```

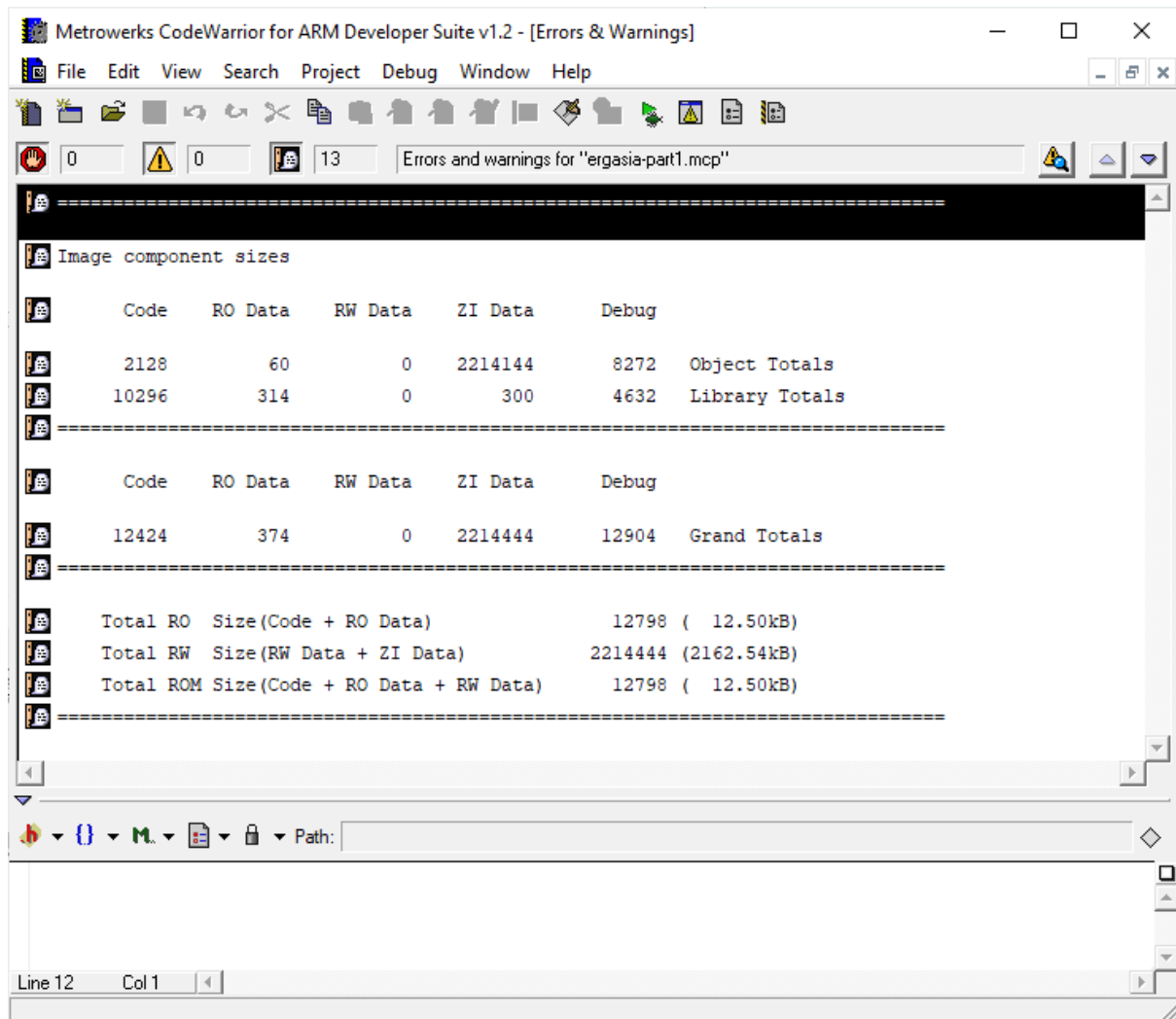
Αντί να επεξεργαζόμαστε ολόκληρο τον πίνακα σειριακά, ο υπολογισμός περιορίζεται σε μικρότερα blocks, που χωρούν πιο αποδοτικά στις cache μνήμες, μειώνοντας έτσι την ανάγκη για πρόσβαση στην κύρια μνήμη. Με τον τρόπο αυτό γλιτώνουμε τις συχνές προσκομίσεις του CPU στην RAM.



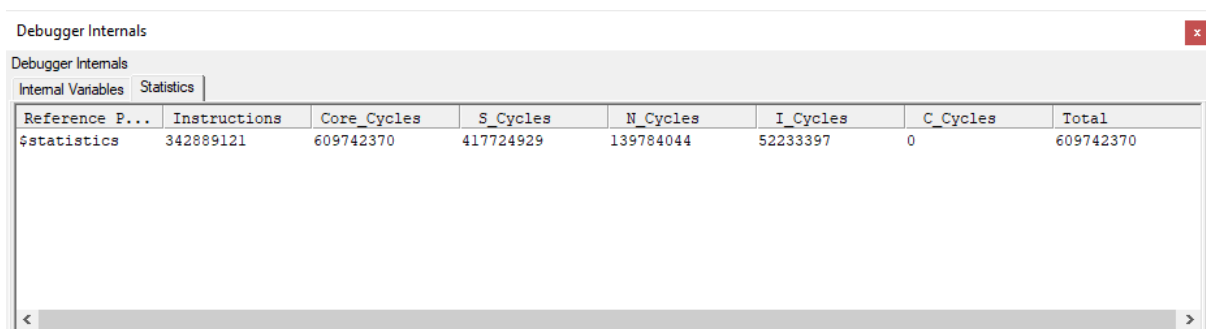
Εικόνα 14: Loop tiling. A demonstration on how Cache Locality works.

Τέλος, στις δύο τελευταίες υλοποιήσεις του αλγορίθμου μας επιλέξαμε να ενεργοποιήσουμε όλα τα compiler flags που μας παρέχει το CodeWarrior (-O2 και -Otime). Στην συνέχεια κάναμε manually aggressively unroll σε όλα for loops της υλοποίησης μας μαζί με χρήση της συνάρτησης memcpy (ομοίως δείξαμε στην δεύτερη υλοποίηση).

6. Specifying a global variable to a specific CPU register



Εικόνα 15: Μεγέθη στοιχείων του generated firmware image.



Εικόνα 16: Αποτελέσματα υλοποίησης με __global_reg.

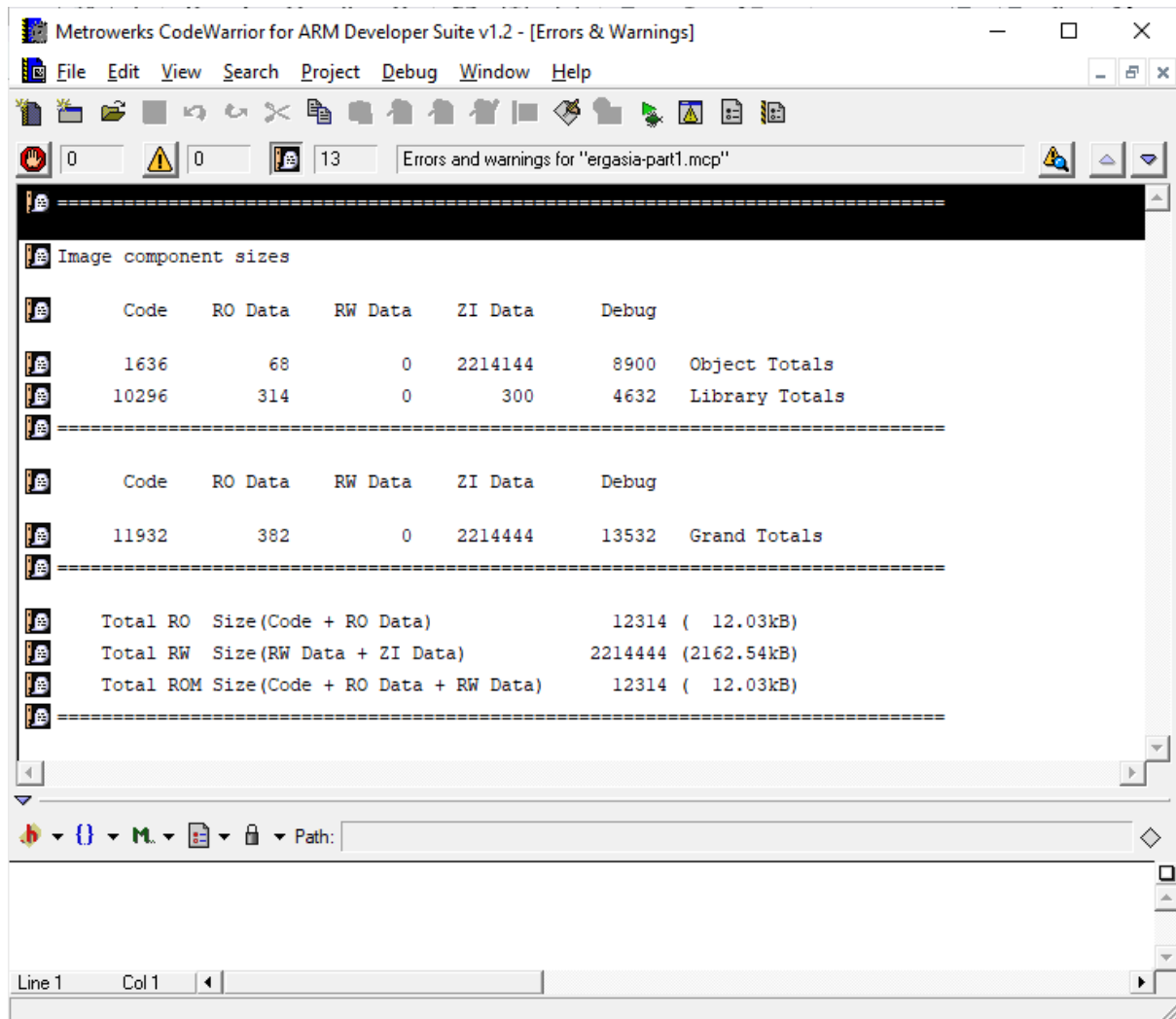
Στην πρώτη από τις δύο τελευταίες υλοποιήσεις μας, χρησιμοποιήσαμε το keyword `__global_reg` [6] ένα ARM Compiler specific feature που εξαναγκάζει τον Compiler να δεσμεύσει έναν CPU Register (που καθορίζουμε εμείς με την χρήση ενός αριθμού μεταξύ του 1 και του 11). Αυτό αντιστοιχεί σε έναν καταχωρητή στην περιοχή r4 έως r11) για μία συγκεκριμένη global τιμή. Με αυτόν τον τρόπο εξασφαλίζουμε ότι ο Compiler δεν θα επαναχρησιμοποιήσει τον καθορισμένο καταχωρητή για άλλους σκοπούς, εμποδίζοντας το σύστημα να κάνει Generate Assembly κώδικα που θα μπορούσε να χρησιμοποιήσει τον ίδιο καταχωρητή για άλλες λειτουργίες, μειώνοντας έτσι πιθανά σφάλματα ή παρεμβολές.

Έπειτα από πολλές δοκιμές αποφασίσαμε να αποθηκεύσουμε με αυτόν τον τρόπο τα iterators `i` και `j` που χρησιμοποιούμε στα for loops του κώδικα μας, προσφέροντας έτσι στον CPU όσο πιο άμεση πρόσβαση γίνεται σε μνήμη την οποία προσκομίζει συχνά. Η αποθήκευση αυτών των τιμών στους προκαθορισμένους καταχωρητές, μειώνει την ανάγκη για πρόσβαση στη μνήμη και ενισχύει την ταχύτητα εκτέλεσης, αφού οι καταχωρητές έχουν ταχύτερη πρόσβαση σε σχέση με τη μνήμη.

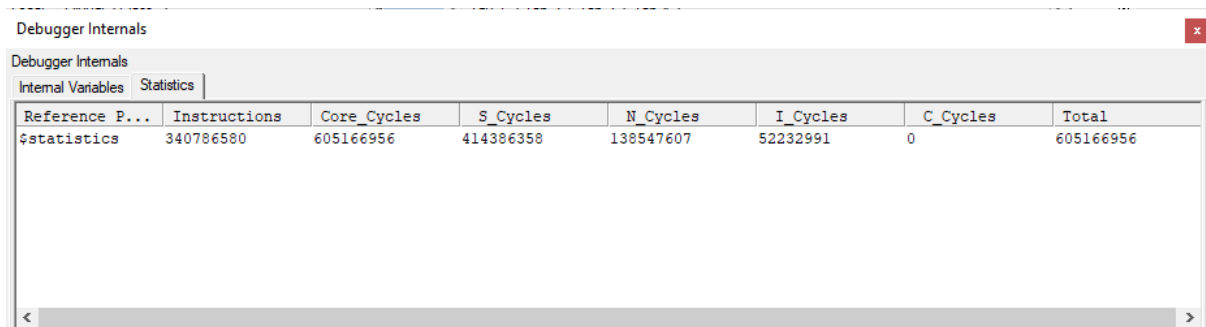
Επιπρόσθετα, προσπαθήσαμε να κάνουμε optimize τον τρόπο που ο compiler κάνει dump δεδομένα στην μνήμη με σκοπό να εξαλείψουμε wasted spaces λόγω padding misalignments κάνοντας specify με το `__align` ARM Compiler specific feature, το οποίο δίνει εντολή στον compiler να ευθυγραμμίσει μια μεταβλητή σε ένα όριο n-byte οδηγώντας πολλές φορές σε καλύτερη ευθυγράμμιση των μεταβλητών. Ωστόσο δεν παρατηρήσαμε κάποιο performance gain μιας που όλα τα datatypes στον κώδικα μας είναι σταθερού μεγέθους. Σε περίπτωση διαφορετικής υλοποίησης όπου θα χρησιμοποιούσαμε κάποιο datatype όπως τα structs ίσως επέφερε καλύτερα αποτελέσματα.

Τέλος, στην προσπάθεια μας περαιτέρω βελτιστοποίησης του κώδικα, εφαρμόσαμε μια τεχνική αναδιάρθρωσης των συνθηκών, που περιλαμβάνει τον μετασχηματισμό σύνθετων συνθηκών σε πιο διακριτά και αναγνώσιμα βήματα. Συγκεκριμένα, αντικαταστήσαμε απευθείας τους τελεστές ternary με μια ενδιάμεση μεταβλητή, η οποία απλοποιεί τη λογική και βελτιώνει την αναγνωσιμότητα του κώδικα. Η προσέγγιση αυτή όχι μόνο καθιστά τον κώδικα πιο ευανάγνωστο, αλλά μπορεί επίσης να προσφέρει οριακά βελτιωμένη υπολογιστική απόδοση σε περιβάλλοντα με υψηλές απαιτήσεις επεξεργασίας, λόγω της απλούστευσης της λογικής ελέγχου.

7. Event driven execution using labels and array Lookup Table



Εικόνα 17: Μεγέθη στοιχείων του generated firmware image.



Εικόνα 18: Αποτελέσματα τελικής υλοποίησης. (0.75% speedup)

Ομοίως με την προηγούμενη υλοποίηση μας, χρησιμοποιούμε `__global_reg`, `__align` καθώς και μία τεχνική αναδιάρθρωσης των συνθηκών. Στη νέα υλοποίηση, αξιοποιήσαμε τον πίνακα `f`, ο οποίος περιέχει σταθερές τιμές (-1 και 0) για την αρχικοποίηση του πίνακα `distance` βάσει της κατάστασης των στοιχείων του πίνακα `input`. Αυτή η τεχνική αποτελεί μια μορφή πίνακα αναζήτησης (lookup table) που επιτρέπει τη γρήγορη (η πρόσβαση σε ένα στοιχείο είναι $O(1)$) και αποδοτική ανάθεση τιμών μέσω του λογικού χειριστή ! αντί για πολλαπλές συνθήκες και εκφράσεις. Με τον τρόπο αυτόν, μειώνεται η πολυπλοκότητα της λογικής αρχικοποίησης και βελτιώνεται η αναγνωσιμότητα του κώδικα, ενώ ταυτόχρονα περιορίζεται το κόστος εκτέλεσης, καθώς αποφεύγεται η εκτέλεση περιττών εντολών. Για παράδειγμα, για την αντιγραφή των byte της εικόνας εισόδου (input image bytes) στον προσωρινό buffer `input`, μετασχηματίσαμε τα for loops ως εξής:

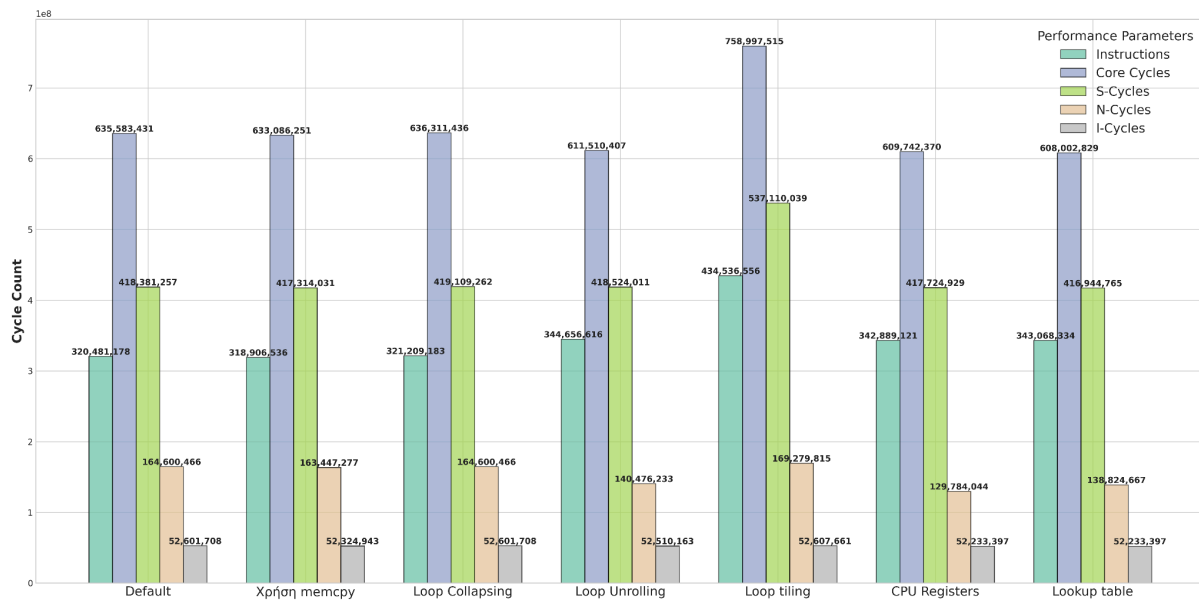
```
__align(8) int f[2] = {-1, 0};

for (i = 0; i < N; i += 4) {
    for (j = 0; j < M; j += 4) {
        temp = !(input[i][j]);
        distance[i][j] = f[temp];

        ...
    }
    if (i + 1 < N) { ... }
    if (i + 2 < N) { ... }
    if (i + 3 < N) { ... }
}
```

Επιπλέον, αναθεωρήσαμε τη ροή του αλγορίθμου εισάγοντας την εντολή `goto LOOP` για την υλοποίηση της επανάληψης, αντικαθιστώντας το άπειρο `for (;;)`. Αυτή η αλλαγή στοχεύει στη μείωση της πολυπλοκότητας της ροής και καθιστά πιο προφανή την έξοδο από τον βρόχο όταν πληρούνται οι συνθήκες. Επιπρόσθετα, αξιοποιήσαμε τον λογικό χειριστή ! για να ανατρέψουμε την τιμή των στοιχείων του πίνακα `input` κατά την αρχικοποίηση. Αυτό επιτρέπει την αντικατάσταση της σύνθετης λογικής `input[i][j] == 0` με έναν πιο άμεσο τρόπο υπολογισμού που μεταφράζεται σε πιο αποτελεσματικό assembly κώδικα. Ο συνδυασμός αυτής της τεχνικής με τον πίνακα `f` διασφαλίζει την αποφυγή επαναλαμβανόμενων υπολογισμών και οδηγεί σε έναν πιο αποδοτικό κώδικα.

Σύγκριση διαφορετικών τεχνικών βελτιστοποίησης



Εικόνα 19: Σύγκριση των μεθόδων βελτιστοποίησης διαγραμματικά.

Στο παραπάνω διάγραμμα φαίνονται συγκεντρωτικά τα αποτελέσματα όλων των τεχνικών βελτιστοποίησης που υλοποιήσαμε. Πιο συγκεκριμένα, η πιο αποτελεσματική μέθοδος βελτιστοποίησης του κώδικα μας είναι μέσω της τεχνικής του Loop Unrolling, η οποία, με την αποφυγή του overhead των υπολογισμών ορίων στις βρόχους, επιτρέπει ταχύτερη εκτέλεση των εντολών. Εντούτοις, παρατηρείται σημαντική αύξηση στον αριθμό των κύκλων (cycles) για την εκτέλεση, ιδιαίτερα στις τεχνικές που σχετίζονται με την τμηματοποίηση των δεδομένων στη μνήμη (Loop Tiling), καθώς η καλύτερη διαχείριση της cache συνοδεύεται από αυξημένο υπολογιστικό κόστος για τη διαχείριση των τμημάτων (tiles).

Η χρήση του CPU Registers βελτιώνει ακόμα περισσότερο την απόδοση, επιταχύνοντας την εκτέλεση των εντολών με την τοποθέτηση των δεδομένων σε συγκεκριμένους καταχωρητές, μειώνοντας έτσι την ανάγκη για πρόσβαση στη μνήμη. Αυτό συμβάλλει στη μείωση του αριθμού των κύκλων εκτέλεσης, διατηρώντας παράλληλα υψηλή απόδοση στην εκτέλεση των εντολών.

Παράλληλα, η ενσωμάτωση της τεχνικής Lookup Table προσφέρει σημαντική βελτίωση στην ταχύτητα του προγράμματος, καθώς μειώνεται η ανάγκη για επαναλαμβανόμενους υπολογισμούς κατά τη διάρκεια της εκτέλεσης. Αυτή η τεχνική είναι ιδιαίτερως χρήσιμη σε περιπτώσεις όπου απαιτείται η εκτέλεση πολύπλοκων ή επαναλαμβανόμενων μαθηματικών πράξεων, καθώς η προσπέλαση των δεδομένων στους πίνακες είναι σαφώς πιο γρήγορη από την πραγματοποίηση αυτών των πράξεων σε πραγματικό χρόνο.

Μέγεθος πίνακα δεδομένων και αριθμός προσπελάσεων

Στον παρακάτω πίνακα μπορούμε να δούμε αναλυτικά τον αριθμό sequential (s-cycles) και non-sequential (n-cycles) reads στη μνήμη, για τις διάφορες υλοποιήσεις μας:

Μέθοδος	S - Cycles	N - Cycles	Συνολικά	Ποσοστό Βελτίωσης
Default	418381257	164600466	582981723	–
Χρήση memcpy	417314031	163447277	580761308	0.38%
Loop Collapsing	419109262	164600466	583709728	-0.12%
Loop Unrolling	418524011	140476233	559000244	4.11%
Loop Tiling	537110039	169279815	706389854	-21.168%
CPU Registers	416944765	138824667	555769432	4.67%
Lookup table	414386358	138547607	552933965	5.15%

Πίνακας 1: Σύγκριση βελτιστοποιήσεων

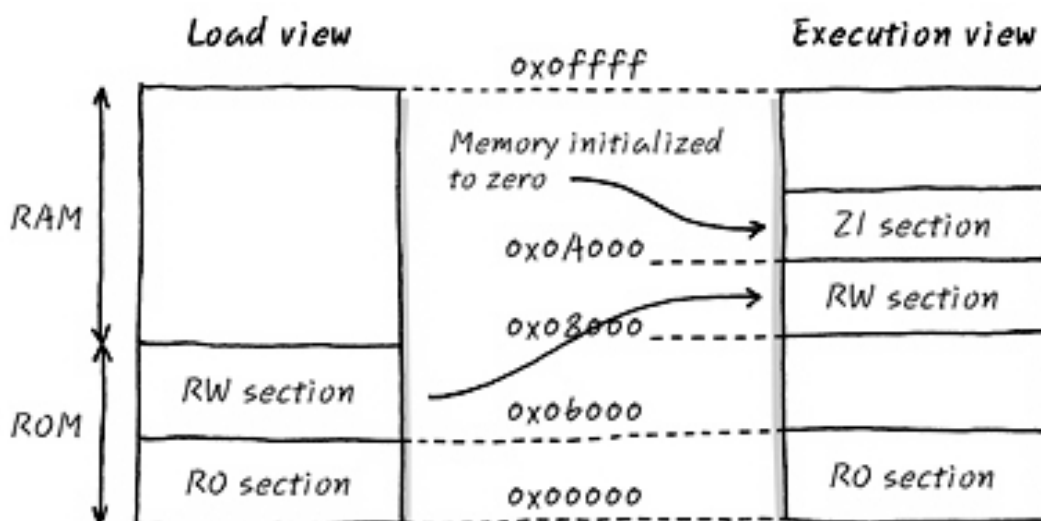
Όπως παρατηρούμε, η χρήση των CPU registers έχει το μεγαλύτερο ποσοστό βελτίωσης, σε αντίθεση με το Loop Tiling και το Loop Collapsing, τα οποία έχουν αρνητική επίδραση στην επίδοση του προγράμματος μας. Τέλος, παρατηρούμε ότι το συνολικό μέγεθος του πίνακα δεδομένων είναι 2162.54 kB.

Μέρος 2ο - Μνήμη

Στο δεύτερο μέρος της εξαμηνιαίας εργασίας καλούμαστε να υλοποιήσουμε μία ιεραρχία μνήμης δεδομένων δύο επιπέδων (ROM για τον κώδικα του προγράμματος σας και RAM για τα δεδομένα) χρησιμοποιώντας τις τεχνικές που καλύφθηκαν στο δεύτερο εργαστήριο. Για τον σκοπό αυτό χρειάζεται να χρησιμοποιήσουμε την τελευταία υλοποίηση του προγράμματος μας, που πραγματοποιήσαμε στο πλαίσιο του πρώτου μέρους.

Σε κάθε υπολογιστικό σύστημα, η μνήμη διαχωρίζεται σε δύο βασικές περιοχές: την περιοχή Load και την περιοχή Execution. Ο διαχωρισμός αυτός εξυπηρετεί τη διαφορετική χρήση της μνήμης ανάλογα με την κατάσταση του συστήματος, είτε πρόκειται για την κατάσταση αδρανείας (Load) είτε για την κατάσταση εκτέλεσης (Execution). Στην περιοχή Load, όπου το σύστημα βρίσκεται σε αδρανή κατάσταση, η μνήμη ROM αποθηκεύει στατικά δεδομένα και τον απαραίτητο κώδικα που θα εκτελεστεί κατά την ενεργοποίηση του συστήματος.

Αυτή η περιοχή είναι σταθερή και μη τροποποιήσιμη, διασφαλίζοντας την ακεραιότητα του αρχικού κώδικα και των δεδομένων. Όταν το σύστημα εισέρχεται στην περιοχή Execution, εκκινεί η δυναμική λειτουργία του, και μέρος των δεδομένων μεταφέρεται στη μνήμη RAM. Αυτή η μεταφορά είναι απαραίτητη, καθώς η RAM είναι ταχύτερη και επιτρέπει την τροποποίηση των δεδομένων κατά την εκτέλεση. Στην περιοχή Execution, η RAM χρησιμοποιείται για την αποθήκευση δεδομένων εργασίας (όπως οι στοίβες και οι μεταβλητές) και την εκτέλεση των απαραίτητων πράξεων του συστήματος.

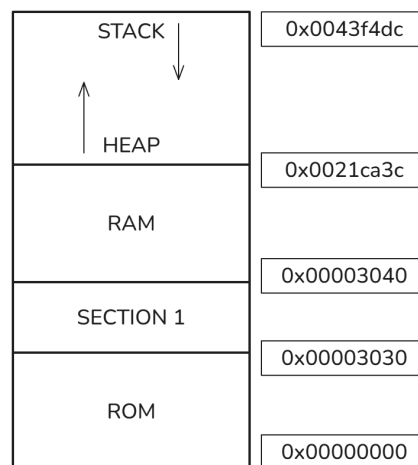


Εικόνα 20: Οπτικοποίηση σύγκρισης Load view - Execution view ARM

Αρχικά, για τη δημιουργία του αρχείου scatter.txt (linker script), υπεύθυνο για τη χαρτογράφηση (mapping) των Load View Regions στις Execution View Regions, υπολογίσαμε όλες τις διευθύνσεις μνήμης που απαιτούνται: το μέγεθος της ROM, του custom section SECTION1, και της (S)RAM. Πιο συγκεκριμένα το μέγεθος της ROM προκύπτει από το άθροισμα του RO Data και του RW Data, το μέγεθος αυτού του custom section ορίστηκε στα 16 bytes (0x10), καθώς γνωρίζουμε ότι θα αποθηκευτούν εκεί 4 ακέραιοι τύπου int (4 x 4 bytes) ενώ στο τέλος το μέγεθος της (S)RAM υπολογίζεται ως το άθροισμα των RW Data και ZI Data.

```
ROM 0x00000000 0x00003030
{
    ROM 0x00000000 0x00003030
    {
        *.o ( +RO )
    }
    SECTION1 0x00003030 0x00000010
    {
        * ( .rwdata )
        *.o ( +RW )
    }
    RAM 0x00003040 0x0021ca3c
    {
        *.o ( +ZI )
    }
}
```

Για τον υπολογισμό του μεγέθους της (D)RAM, η συνολική μνήμη πρέπει να καλύπτει όλες τις περιοχές που απαιτούνται κατά την εκτέλεση του προγράμματος, περιλαμβάνοντας τον εκτελέσιμο κώδικα, τα δεδομένα ανάγνωσης και εγγραφής, καθώς και τις μηδενικές αρχικοποιήσεις. Συγκεκριμένα, το μέγεθος της μνήμης προσδιορίζεται ως το άθροισμα του κώδικα (Code ή Instructions), των δεδομένων μόνο για ανάγνωση (RO Data), των δεδομένων που μπορούν να διαβαστούν και να γραφούν (RW Data) και των δεδομένων που αρχικοποιούνται με μηδενικές τιμές κατά την εκκίνηση (ZI Data).



Εικόνα 21: Οπτικοποίηση των ορίων της μνήμης

To base address του stack, δηλαδή το σημείο έναρξης της στοίβας στη μνήμη, υπολογίζεται ως το άθροισμα αυτών των περιοχών, έτσι ώστε να διασφαλίζεται ότι κάθε τμήμα διαθέτει επαρκή χώρο και ότι δεν υπάρχει σύγκρουση μεταξύ διαφορετικών περιοχών μνήμης. Αυτή η προσεκτική διαχείριση είναι απαραίτητη για τη σωστή λειτουργία του προγράμματος, καθώς κάθε περιοχή έχει συγκεκριμένη χρήση και απαιτήσεις.

```
#include <rt_misc.h>
```

```
__value_in_regs struct __initial_stackheap __user_initial_stackheap(unsigned R0, unsigned SP,
unsigned R2, unsigned SL) {
    struct __initial_stackheap config;
    config.heap_base = 0x0021fa7c;
    config.stack_base = 0x0043f4dc;
    return config;
}
```

Αρχιτεκτονική 1 - ROM/RAM BUS = 4 (4x8 bits = 32 bits bus width)

```
00000000 00002fd4 ROM 4 R 1/1 1/1
00002fd4 00000010 SECTION1 4 RW 250/50 250/50
00002fe4 0021ca2c RAM 4 RW 250/50 250/50
```

Debugger Internals										
Debugger Internals										
Internal Variables Statistics										
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idle_Cycles	
Statistics	340971621	605537278	414386633	138917399	52233246	0	26024019	631561297	373056	

Εικόνα 22: Αποτελέσματα Αρχιτεκτονικής 1

Αρχιτεκτονική 2 - ROM/RAM BUS = 2 (2x8 bits = 16 bits bus width)

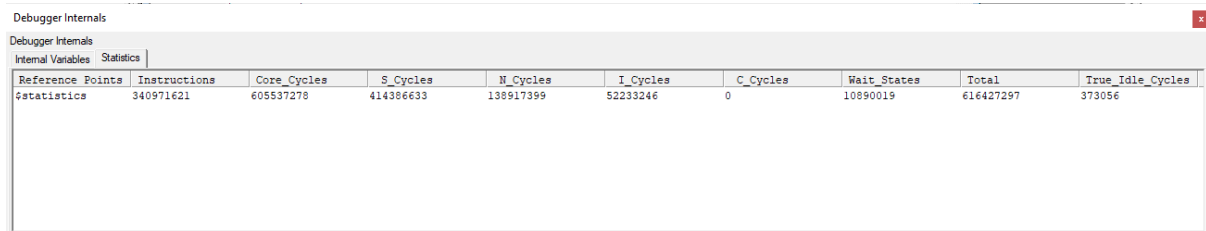
```
00000000 00002fd4 ROM 2 R 1/1 1/1
00002fd4 00000010 SECTION1 2 RW 250/50 250/50
00002fe4 0021ca2c RAM 2 RW 250/50 250/50
```

Debugger Internals										
Debugger Internals										
Internal Variables Statistics										
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idle_Cycles	
Statistics	340971621	605537278	414386633	138917399	52233246	0	525161123	1130698401	373056	

Εικόνα 23: Αποτελέσματα Αρχιτεκτονικής 2

Αρχιτεκτονική 3 - ROM/RAM HIGH SPEED (same bus width, faster ram)

```
00000000 00002fd4 ROM 4 R 1/1 1/1
00002fd4 00000010 SECTION1 4 RW 250/50 250/50
00002fe4 0021ca2c RAM 4 RW 250/50 250/50
```



Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idle_Cycles
Statistics	340971621	605537278	414386633	138917399	52233246	0	10890019	616427297	373056

Εικόνα 24: Αποτελέσματα Αρχιτεκτονικής 3

Σύγκριση αρχιτεκτονικών μνήμης

Στον πίνακα 2 φαίνονται αναλυτικά τα wait states και τα total cycles των τριών αρχιτεκτονικών που υλοποιήσαμε, καθώς και το ποσοστό βελτίωσης σε σχέση με την αρχιτεκτονική 1.

	Wait states	Βελτίωση σε σχέση με 1	Total Cycles	Βελτίωση σε σχέση με 1
Αρχιτεκτονική 1	26025304	-	631562582	-
Αρχιτεκτονική 2	52516248	-101.83%	1130699686	-78.98%
Αρχιτεκτονική 3	10890019	58.15%	616427297	2.41%

Πίνακας 2: Σύγκριση αρχιτεκτονικών

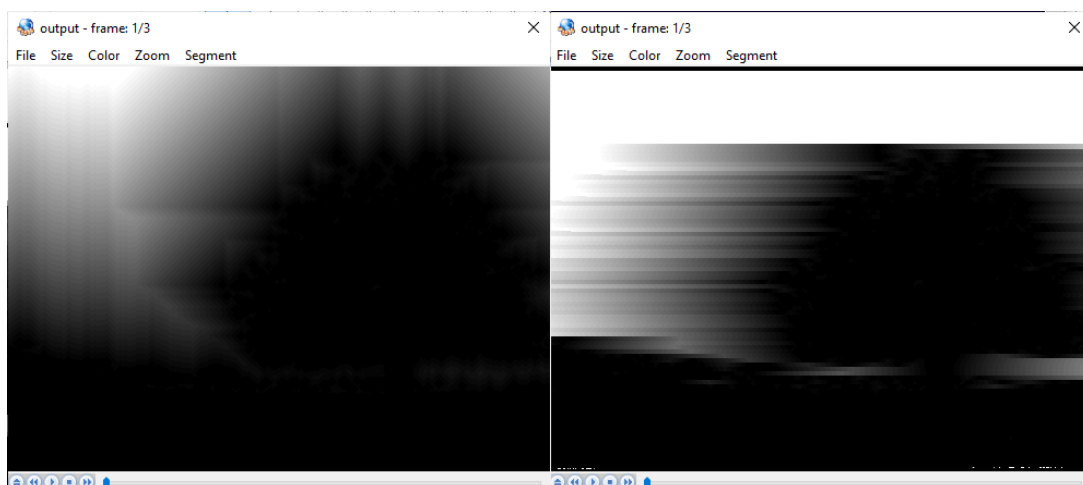
Παρατηρούμε ότι η μείωση του bus width στην Αρχιτεκτονική 2, έχει αρνητική επίδραση στην απόδοση του κώδικα. Αντιθέτως, η Αρχιτεκτονική 3 παρουσιάζει βελτίωση σε σχέση με την 1, λόγω της χρήσης γρηγορότερης μνήμης. Ωστόσο αυτό συμβαίνει μόνο μόνο στα wait states, καθώς στα total cycles δεν παρατηρείται μεγάλη βελτίωση.

Υλοποιήσεις με χρήση buffer

Η ιδέα πίσω από αυτή τη βελτιστοποίηση βασίζεται στη χρήση ενός buffer, δηλαδή ενός μικρότερου πίνακα που αποθηκεύει προσωρινά τμήματα δεδομένων από τον κύριο πίνακα. Αυτή η προσέγγιση μειώνει τις προσπελάσεις στη μνήμη και αξιοποιεί καλύτερα την cache του συστήματος, αφού η επεξεργασία γίνεται σε τοπικό επίπεδο με τα δεδομένα που βρίσκονται ήδη στη μνήμη cache. Όταν εργαζόμαστε με μεγάλους πίνακες, οι επαναλαμβανόμενες προσπελάσεις σε μεγάλες περιοχές της κύριας μνήμης μπορεί να είναι αργές και να προκαλούν καθυστερήσεις. Χρησιμοποιώντας έναν buffer, περιορίζεται η ανάγκη πρόσβασης στη μνήμη και βελτιώνεται η απόδοση, μειώνοντας τη διάρκεια των καθυστερήσεων που σχετίζονται με τη μεταφορά δεδομένων από τη μνήμη.

Στην πρώτη υλοποίηση, ο buffer χρησιμοποιείται για την ανάγνωση δεδομένων από τον κύριο πίνακα `current_y` σε μπλοκ των 3 γραμμών, αποτρέποντας την πολλαπλή προσπέλαση στη μνήμη και βελτιώνοντας τη χωρική τοπικότητα των δεδομένων. Ωστόσο, η χρήση του buffer δεν επεκτείνεται σε όλο το μήκος του αλγορίθμου, καθώς περιορίζεται σε μια αρχική φάση και δεν επαναχρησιμοποιείται για τις συνεχείς προσπελάσεις του πίνακα `distance`. Αυτό συμβαδίζει με την περιγραφή του ερωτήματος, που ζητούσε τη χρήση ενός μικρότερου πίνακα ως buffer για την αποδοτικότερη εκτέλεση του προγράμματος στην ιεραρχία μνήμης. Παρά την εφαρμογή του buffer, οι επαναλαμβανόμενες προσπελάσεις στο στάδιο εξέτασης των γειτόνων δεν επωφελούνται πλήρως από αυτόν, καθώς οι γειτονικές τιμές απαιτούν προσβάσεις στον κύριο πίνακα και δεν αποθηκεύονται τοπικά στον buffer, μειώνοντας την αναμενόμενη απόδοση.

Στη δεύτερη υλοποίηση, έγινε προσπάθεια να επεκταθεί η χρήση του buffer σε όλο το μήκος του αλγορίθμου, καλύπτοντας τις ανάγκες του αλγορίθμου για καλύτερη τοπικότητα των δεδομένων. Ωστόσο, παρά την καλύτερη αξιοποίηση του buffer, το τελικό αποτέλεσμα δεν ταιριάζει πλήρως με το αναμενόμενο, αφού οι συνεχείς ενημερώσεις του πίνακα `distance` δημιουργούν την ανάγκη για επαναφορά δεδομένων από τη μνήμη.



Αριστερά: Αποτέλεσμα πρώτης υλοποίησης. Δεξιά: Αποτέλεσμα δεύτερης υλοποίησης

Πρώτη υλοποίηση - Χρήση προσωρινού buffer για διάκριση αντικειμένων

```
#include <string.h>
#include "opts.h"

__global_reg(1) int i; // r4
__global_reg(2) int j; // r5
__global_reg(3) int k; // r6
__global_reg(4) int b; // r7

#pragma arm section rwdata=".dram"
__align(8) int f[2] = {-1, 0};
#pragma arm section

#pragma arm section zidata=".cache"
__align(8) int distance[N][M];
__align(8) int buffer[ROWS][M];
#pragma arm section

#pragma Otime
void distance_transformer() {
    __asm("MOV r6, #1"); // k + 1 (since k is stored in r6 it's perhaps easier)

    for (i = 0; i < N; i += ROWS) { // ROWS = 3
        memcpy(buffer, &current_y[i][0], ROWS * M * sizeof(int)); // Αντιγραφή ROWS * M στοιχείων
        for (b = 0; b < ROWS; ++b) {
            for (j = 0; j < M; j += 4) {
                distance[i + b][j] = f[!buffer[b][j]];
                if (j + 1 < M) distance[i + b][j + 1] = f[!buffer[b][j + 1]];
                if (j + 2 < M) distance[i + b][j + 2] = f[!buffer[b][j + 2]];
                if (j + 3 < M) distance[i + b][j + 3] = f[!buffer[b][j + 3]];
            }
        }
    }

    LOOP:
    for (i = 0; i < N; ++i) {
        for (j = 0; j < M; j += 2) {
            if (distance[i][j] == k - 1) {
                if (i > 0 && distance[i - 1][j] == -1) distance[i - 1][j] = k;
                if (i < N - 1 && distance[i + 1][j] == -1) distance[i + 1][j] = k;
                if (j > 0 && distance[i][j - 1] == -1) distance[i][j - 1] = k;
                if (j < M - 1 && distance[i][j + 1] == -1) distance[i][j + 1] = k;
            }
            if (j + 1 < M) {
                if (distance[i][j + 1] == k - 1) {
                    if (i > 0 && distance[i - 1][j + 1] == -1) distance[i - 1][j + 1] = k;
                    if (i < N - 1 && distance[i + 1][j + 1] == -1) distance[i + 1][j + 1] = k;
                    if (j > -1 && distance[i][j] == -1) distance[i][j] = k;
                    if (j < M - 2 && distance[i][j + 2] == -1) distance[i][j + 2] = k;
                }
            }
        }
    }

    if (k < 255) {
        k++;
        goto LOOP;
    }

    memcpy(current_y, distance, N * M * sizeof(int));
}
```

Ομοίως με πριν υπολογίζουμε τις νέες διεύθυνσης μνήμης όπου προκύπτει το νέο scatter.txt

```
ROM 0x00000000 0x00002eff          ; Load mode ROM size
{
    ROM 0x00000000 0x00002ef0      ; Execution mode ROM size,
    {
        *.o ( +RO )                ; including all object files.
    }
    DRAM 0x00002ef0 0x002d8934      ; Simple dynamic ram.
    {
        * ( .dram )                 ; Match sections tagged as ".dram"
        *.o ( +RW, +ZI )            ; and include all object files
    }
    CACHE 0x002db7f0 0x000b5a48     ; Cache region for our buffer.
    {
        * ( .cache )                ; match all sections with ".cache"
    }
}
```

Αρχιτεκτονική 1 - ROM/RAM BUS = 4 (4x8 bits = 32 bits bus width)

```
00000000 00002ef0 ROM 4 R 1/1 1/1
00002ef0 002d8934 DRAM 4 RW 250/50 250/50
002db7f0 000bfff8 CACHE 4 RW 1/1 1/1
```

Debugger Internals									
Debugger Internals									
Internal Variables Statistics									
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idle_Cycles
\$statistics	341464317	606404103	415064616	139106374	52233113	0	23845414	630249517	373055

Εικόνα 26: Αποτελέσματα Αρχιτεκτονικής 1 με χρήση buffer

Αρχιτεκτονική 2 - ROM/RAM BUS = 2 (2x8 bits = 16 bits bus width)

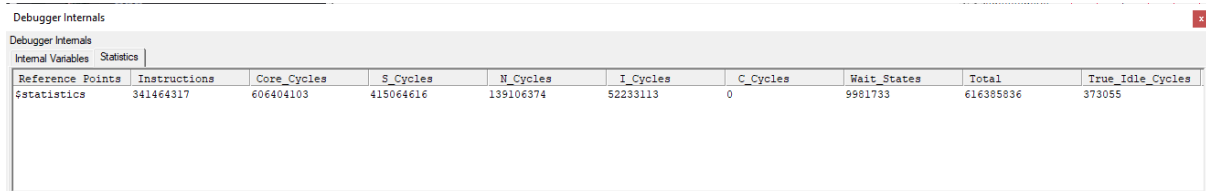
```
00000000 00002ef0 ROM 2 R 1/1 1/1
00002ef0 002d8934 DRAM 2 RW 250/50 250/50
002db7f0 000bfff8 CACHE 2 RW 1/1 1/1
```

Debugger Internals									
Debugger Internals									
Internal Variables Statistics									
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idle_Cycles
\$statistics	341464317	606404103	415064616	139106374	52233113	0	572053434	1178457537	373055

Εικόνα 27: Αποτελέσματα Αρχιτεκτονικής 2 με χρήση buffer

Αρχιτεκτονική 3 - ROM/RAM HIGH SPEED (same bus width, faster ram)

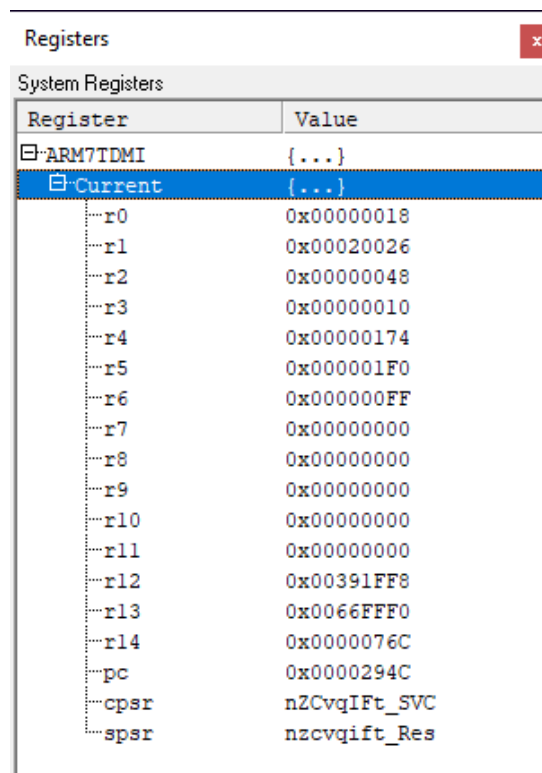
```
00000000 00002ef0 ROM 4 R 1/1 1/1
00002ef0 002d8934 DRAM 4 RW 110/30 110/30
002db7f0 000bfff8 CACHE 4 RW 1/1 1/1
```



Debugger Internals									
Internal Variables Statistics									
Reference Points	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total	True_Idle_Cycles
\$statistics	341464317	606404103	415064616	139106374	52233113	0	9981733	616385836	373055

Εικόνα 28: Αποτελέσματα Αρχιτεκτονικής 3 με χρήση buffer

Σε αυτό το σημείο αξίζει να αναφερθούμε σε ένα από τα σημαντικότερα optimizations που έχουμε χρησιμοποιήσει στην υλοποίησή μας (assigning a variable to a register). Μέσω του AXD debugger μπορούμε να παρακολουθήσουμε ανα πάσα στιγμή (μέσω breakpoints) ή στο τέλος του προγράμματος μας την τιμή που έχουν λάβει όλοι οι registers του CPU:



Registers	
System Registers	
Register	Value
ARM7TDMI	{...}
Current	{...}
r0	0x00000018
r1	0x00020026
r2	0x00000048
r3	0x00000010
r4	0x00000174
r5	0x000001F0
r6	0x000000FF
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00391FF8
r13	0x0066FFF0
r14	0x0000076C
pc	0x0000294C
cpsr	nZCvqIFt_SVC
spsr	nzcvcift_Res

Εικόνα 29: Registers values as shown inside of AXD Debugger.

Δεύτερη υλοποίηση - Χρήση προσωρινού buffer σε όλο τον αλγόριθμο

```
#include <string.h>
#include "opts.h"

__global_reg(1) int i; // r4
__global_reg(2) int j; // r5
__global_reg(3) int k; // r6
__global_reg(4) int b; // r7

#pragma arm section rwdata=".dram"
__align(4) int f[2] = {-1, 0};
#pragma arm section

#pragma arm section zidata=".cache"
__align(4) int buffer1[4][M];
__align(4) int buffer2[4][M];
__align(4) int distance[N][M];
#pragma arm section

__inline void calculate_distance(int b) {
    for (j = 0; j < M; j += 4) {
        buffer2[i + b][j] = f[!buffer1[b][j]];
        if (j + 1 < M) buffer2[i + b][j + 1] = f[!buffer1[b][j + 1]];
        if (j + 2 < M) buffer2[i + b][j + 2] = f[!buffer1[b][j + 2]];
        if (j + 3 < M) buffer2[i + b][j + 3] = f[!buffer1[b][j + 3]];
    }
}

__inline void check_neighbors(int b) {
    for (j = 0; j < M; j += 2) {
        if (buffer2[i + b][j] == k - 1) {
            if (i + b > 0 && buffer2[i + b - 1][j] == -1) buffer2[i + b - 1][j] = k;
            if (i + b < N - 1 && buffer2[i + b + 1][j] == -1) buffer2[i + b + 1][j] = k;
            if (j > 0 && buffer2[i + b][j - 1] == -1) buffer2[i + b][j - 1] = k;
            if (j < (M - 1) && buffer2[i + b][j + 1] == -1) buffer2[i + b][j + 1] = k;
        }
        if (j + 1 < M) {
            if (buffer2[i + b][j + 1] == k - 1) {
                if (i + b > 0 && buffer2[i + b - 1][j + 1] == -1) buffer2[i + b - 1][j + 1] = k;
                if (i + b < N - 1 && buffer2[i + b + 1][j + 1] == -1) buffer2[i + b + 1][j + 1] = k;
                if (j > -1 && buffer2[i + b][j] == -1) buffer2[i + b][j] = k;
                if (j < (M - 2) && buffer2[i + b][j + 2] == -1) buffer2[i + b][j + 2] = k;
            }
        }
    }
}

#pragma Otime
void distance_transformer() {
    for (i = 0; i < N; i += 4) {
        memcpy(buffer1, &current_y[i][0], 4 * M * sizeof(int));

        calculate_distance(0);
        calculate_distance(1);
        calculate_distance(2);
        calculate_distance(3);

        for (k = 1; k < 255; ++k) {
            check_neighbors(0);
            check_neighbors(1);
            check_neighbors(2);
            check_neighbors(3);
        }

        memcpy(distance, &buffer2[i][0], 4 * M * sizeof(int));
    }

    memcpy(current_y, distance, N * M * sizeof(int));
}
```

Σύγκριση αρχιτεκτονικών στην υλοποίηση με χρήση buffer

Στον πίνακα 3 φαίνονται αναλυτικά τα wait states και τα total cycles των τριών αρχιτεκτονικών που υλοποιήσαμε στην πρώτη υλοποίηση με την χρήση buffer, καθώς και το ποσοστό βελτίωσης σε σχέση με την αρχιτεκτονική 1.

	Wait states	Βελτίωση σε σχέση με Ερ. 1	Total Cycles	Βελτίωση σε σχέση με Ερ. 1
Αρχιτεκτονική 1	23845414	8.37%	630249517	0.21%
Αρχιτεκτονική 2	532053434	5.41%	1178457537	-0.042%
Αρχιτεκτονική 3	9981733	8.34%	616385836	0.007%

Πίνακας 3: Σύγκριση αρχιτεκτονικών στην υλοποίηση με χρήση buffer

Παρατηρείται ότι, παρόλο που οι wait states μειώνονται σημαντικά στις Αρχιτεκτονικές 1 και 3, η συνολική μείωση στους κύκλους εκτέλεσης είναι ελάχιστη, ενώ στην Αρχιτεκτονική 2 υπάρχει ακόμη και αρνητική βελτίωση. Συγκεκριμένα, η Αρχιτεκτονική 1 παρουσιάζει μείωση των wait states σε 23,845,414 με βελτίωση 8.37%, ενώ η Αρχιτεκτονική 3 φτάνει στα 9,981,733 με παρόμοια βελτίωση 8.34%. Ωστόσο, η μείωση στους συνολικούς κύκλους εκτέλεσης είναι μόλις 0.21% και 0.007% αντίστοιχα, γεγονός που υποδεικνύει πως η χρήση του buffer δεν έχει αποδώσει τα αναμενόμενα οφέλη. Η Αρχιτεκτονική 2, από την άλλη, εμφανίζει περισσότερους wait states και αρνητική βελτίωση στους κύκλους εκτέλεσης, γεγονός που καταδεικνύει αναποτελεσματικότητα στη χρήση του buffer.

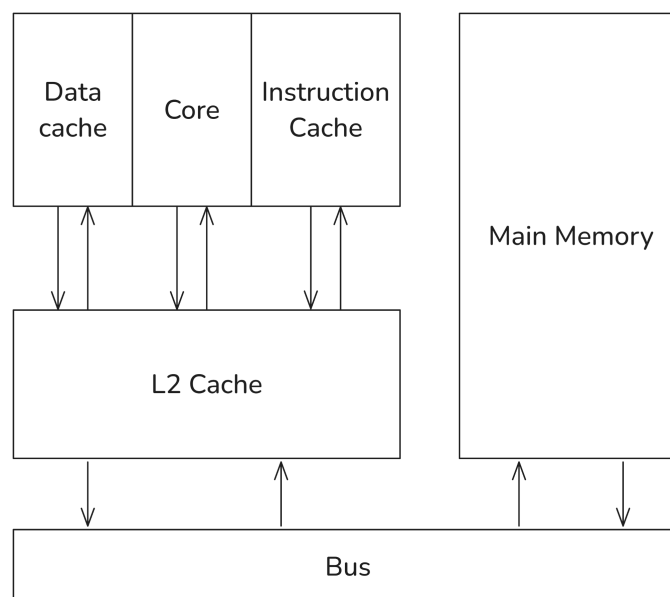
Αυτά τα αποτελέσματα πιθανώς οφείλονται σε μη ιδανική υλοποίηση του buffer. Η διαχείριση των δεδομένων (πχ memcpy) ενδέχεται να προσθέτει επιπλέον overhead, το οποίο εξουδετερώνει το όφελος από τη μείωση των wait states. Συνολικά, η υλοποίηση με τη χρήση buffer δεν φαίνεται να βελτιστοποιεί την απόδοση του αλγορίθμου, γεγονός που απαιτεί περαιτέρω ανάλυση και βελτίωση.

Μέρος 3ο - Εφαρμογή επαναχρησιμοποίησης δεδομένων

Στο τρίτο και τελευταίο μέρος της εξαμηνιαίας εργασίας καλούμαστε να τροποποιήσουμε τον κώδικα μας, έτσι ώστε να εφαρμόσουμε την λογική της επαναχρησιμοποίησης των δεδομένων. Για αρχή θα υλοποιήσουμε δύο διαφορετικές ακολουθίες αντιγραφής δεδομένων και στη συνέχεια θα δούμε μια προτεινόμενη ιεραρχία μνήμης τριών επιπέδων (RAM και 2 μνήμες cache).

Ιεραρχία μνήμης δύο επιπέδων

Η ιεραρχία μνήμης αναφέρεται σε μια ιεραρχία τύπων μνήμης, με ταχύτερες και μικρότερες μνήμες πιο κοντά στον πυρήνα και πιο αργή και μεγαλύτερη μνήμη πιο μακριά. Αυτή η ιεραρχία αποτελείται συνήθως από δευτερεύοντα χώρο αποθήκευσης, όπως μονάδες δίσκου, και κύρια αποθήκευση, συμπεριλαμβανομένων των flash, SRAM και DRAM. Στα ενσωματωμένα συστήματα, αυτή η ιεραρχία συχνά υποδιαιρείται σε μνήμη εντός και εκτός τσιπ, με τη μνήμη στο τσιπ να είναι σημαντικά ταχύτερη λόγω της εγγύτητάς της στον πυρήνα. Ένα βασικό στοιχείο αυτής της ιεραρχίας είναι η κρυφή μνήμη του επεξεργαστή, ένα μικρό, γρήγορο μπλοκ μνήμης που βρίσκεται μεταξύ του πυρήνα και της κύριας μνήμης, κρατώντας αντίγραφα των στοιχείων που προσπελάστηκαν πρόσφατα στην κύρια μνήμη. Η κρυφή μνήμη παίζει καθοριστικό ρόλο στην επιτάχυνση της εκτέλεσης, καθώς επιτρέπει σημαντικά ταχύτερες προσβάσεις σε σύγκριση με την κύρια μνήμη. Μετακινώντας κώδικα ή δεδομένα σε ταχύτερη μνήμη κατά την πρώτη τους πρόσβαση, οι επόμενες προσβάσεις σε αυτόν τον κώδικα ή δεδομένα γίνονται πολύ πιο γρήγορες, με αποτέλεσμα αυξημένη απόδοση.



Εικόνα 30: Τυπική διάταξη μνήμης cache σε αρχιτεκτονική von Neumann.

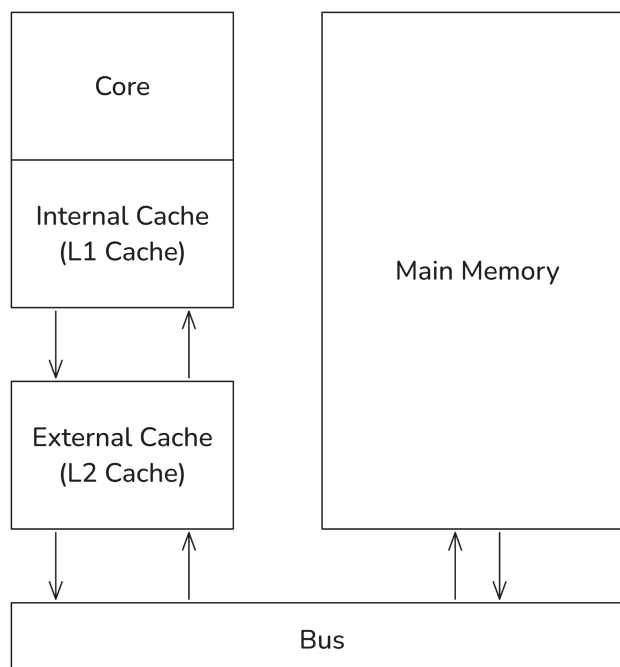
Η ιεραρχία μνήμης η οποία θα υλοποιήσουμε περιλαμβάνει δύο επίπεδα κρυφής μνήμης, τα L1 και L2. Τα δύο επίπεδα διαφέρουν μεταξύ τους και εξυπηρετούν διαφορετικές χρήσεις της μνήμης.

L1 cache

Η κρυφή μνήμη L1, γνωστή και ως προσωρινή μνήμη επιπέδου 1, είναι ο μικρότερος και ταχύτερος τύπος κρυφής μνήμης σε ένα σύστημα υπολογιστή. Είναι μια μικρή, on-chip μνήμη που αποθηκεύει τα δεδομένα και τις οδηγίες που έχει πρόσβαση ο επεξεργαστής με τη μεγαλύτερη συχνότητα. Είναι συνήθως το πρώτο μέρος όπου ο επεξεργαστής αναζητά δεδομένα ή οδηγίες πριν αποκτήσει πρόσβαση στην κύρια μνήμη, καθώς έχει σχεδιαστεί για να παρέχει τους ταχύτερους δυνατούς χρόνους πρόσβασης. Η κρυφή μνήμη L1 είναι συνήθως μικρή, με μέγεθος από μερικά kilobyte έως μερικές εκατοντάδες kilobyte, αλλά είναι εξαιρετικά γρήγορη και χρησιμοποιείται για την αποθήκευση των πιο κρίσιμων δεδομένων και οδηγιών στα οποία χρειάζεται ο επεξεργαστής να έχει γρήγορη πρόσβαση.

L2 cache

Η κρυφή μνήμη L2, είναι ένας τύπος κρυφής μνήμης που είναι μεγαλύτερη και πιο αργή από την κρυφή μνήμη L1, αλλά μικρότερη και πιο γρήγορη από την κύρια μνήμη του υπολογιστή. Συνήθως βρίσκεται στο ίδιο τσιπ με τον επεξεργαστή, αλλά μπορεί επίσης να βρίσκεται σε ξεχωριστό τσιπ ή μονάδα. Η L2 λειτουργεί ως προσωρινή μνήμη μεταξύ τη L1 και της κύριας μνήμης. Η κρυφή μνήμη L2 είναι συνήθως μεγαλύτερη από την κρυφή μνήμη L1, με μέγεθος από μερικές εκατοντάδες kilobyte έως αρκετά megabyte.



Εικόνα 30: Μια τυπική διάταξη μνημών cache

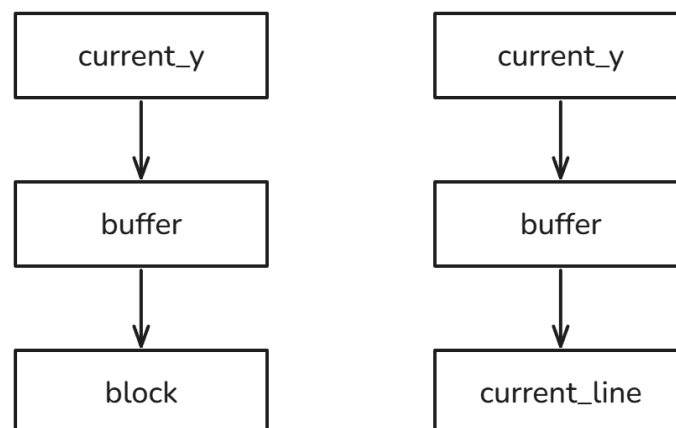
Μετασχηματισμοί επαναχρησιμοποίησης δεδομένων

Οι μετασχηματισμοί επαναχρησιμοποίησης δεδομένων αποτελούν θεμελιώδεις τεχνικές βελτιστοποίησης στη διαχείριση μνήμης. Στο πλαίσιο αυτό, θα εξετάσουμε δύο διαφορετικές προσεγγίσεις: τη χρήση ενός πίνακα `current_line` και τη χρήση ενός πίνακα `block`.

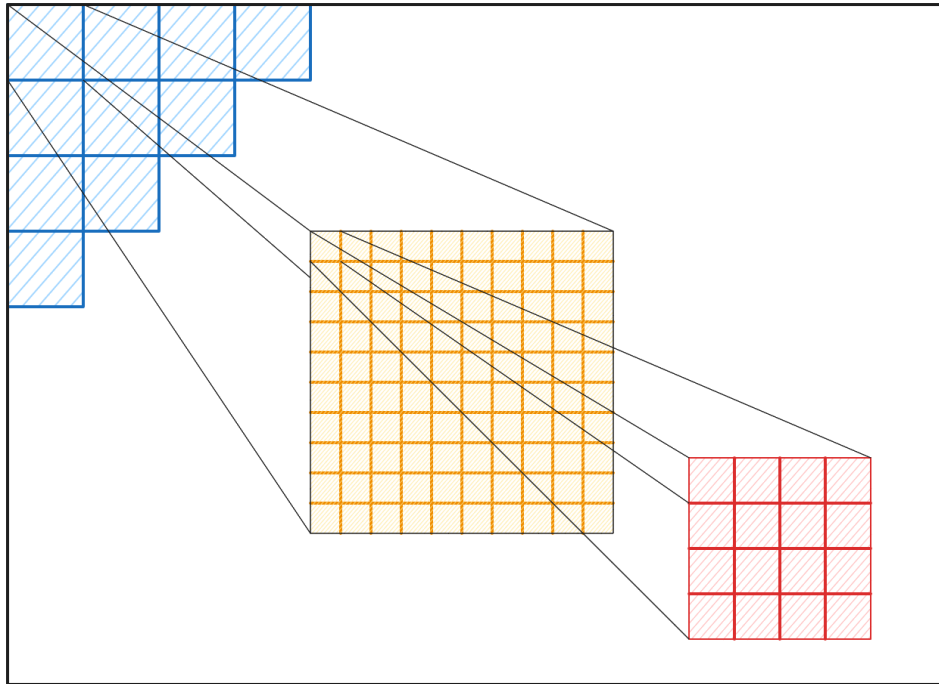
Η πρώτη προσέγγιση, που χρησιμοποιεί τον πίνακα `current_line`, επικεντρώνεται στην προσωρινή αποθήκευση μιας "γραμμής" δεδομένων από τον αρχικό πίνακα. Αυτή η τεχνική είναι ιδιαίτερα αποτελεσματική όταν τα δεδομένα προσπελούνται με γραμμική σειρά ή όταν υπάρχει ανάγκη για επαναλαμβανόμενη πρόσβαση στα ίδια δεδομένα κατά μήκος μιας διάστασης του πίνακα. Η προσέγγιση αυτή μειώνει σημαντικά τις προσπελάσεις στην κύρια μνήμη, καθώς τα δεδομένα που χρειάζονται συχνά διατηρούνται σε έναν μικρότερο, πιο προσβάσιμο `buffer`.

Η δεύτερη προσέγγιση, που χρησιμοποιεί τον πίνακα `block`, εστιάζει στην αποθήκευση ενός δισδιάστατου τμήματος δεδομένων. Αυτή η τεχνική είναι ιδιαίτερα χρήσιμη όταν οι υπολογισμοί απαιτούν πρόσβαση σε γειτονικά στοιχεία και στις δύο διαστάσεις του πίνακα. Ο πίνακας `block` λειτουργεί ως ένας τοπικός καταχωρητής που περιέχει όλα τα απαραίτητα δεδομένα για έναν συγκεκριμένο υπολογισμό, ελαχιστοποιώντας έτσι την ανάγκη για συνεχείς προσπελάσεις στον αρχικό πίνακα.

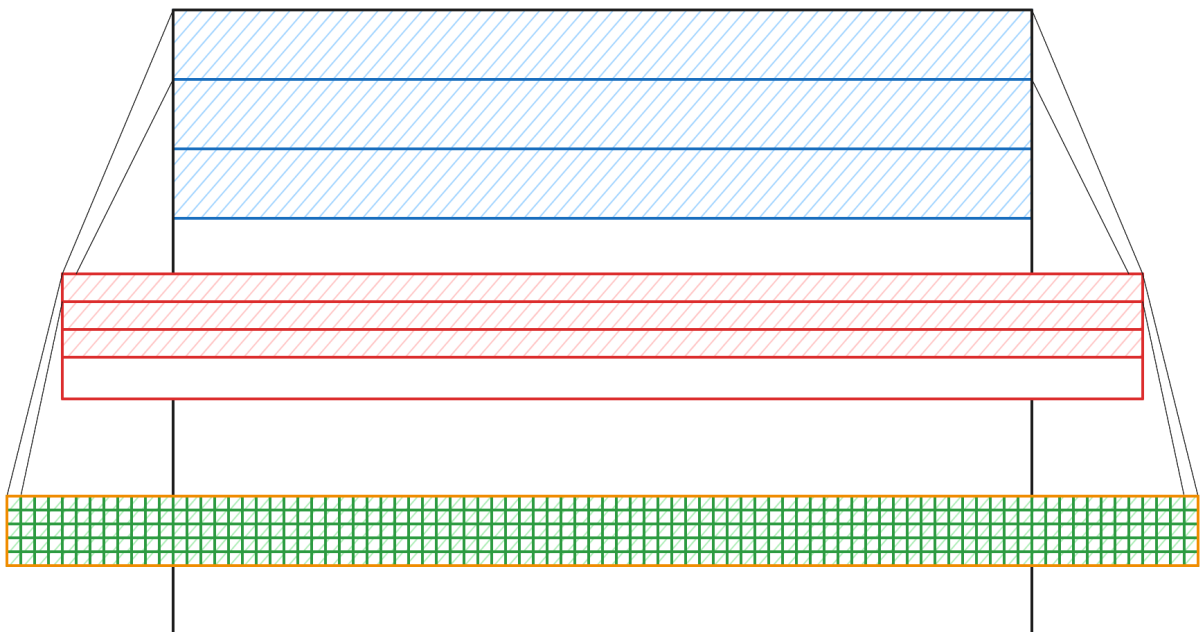
Η επιλογή μεταξύ των δύο αυτών τεχνικών εξαρτάται από διάφορους παράγοντες, όπως το μοτίβο πρόσβασης στα δεδομένα, τους περιορισμούς μνήμης του συστήματος, και τη φύση των υπολογισμών που πρέπει να εκτελεστούν. Η τεχνική του `current_line` είναι συχνά προτιμότερη σε περιπτώσεις όπου υπάρχει έντονη χωρική τοπικότητα κατά μήκος μιας διάστασης, ενώ η τεχνική του `block` μπορεί να είναι καταλληλότερη όταν απαιτείται συχνή πρόσβαση σε ένα συμπαγές τμήμα του πίνακα. Και οι δύο τεχνικές στοχεύουν στη βελτιστοποίηση της χρήσης της ιεραρχίας μνήμης, αλλά προσεγγίζουν το πρόβλημα από διαφορετικές οπτικές γωνίες.



Εικόνα 31: Αριστερά: Το Δέντρο της πρώτης ακολουθίας αντιγραφής με την χρήση `block`. Δεξιά: Η Δέντρο της δεύτερης ακολουθίας με την χρήση `current_line`.



Εικόνα 32: Η πρώτη ακολουθία αντιγραφής δεδομένων (*block*)



Εικόνα 33: Η δεύτερη ακολουθία αντιγραφής δεδομένων (*current_line*)

Προτεινόμενη ιεραρχία

Η προτεινόμενη ιεραρχία μνήμης βασίζεται στη δομή και τις απαιτήσεις του προβλήματος, καθώς και στα χαρακτηριστικά κάθε επιπέδου. Η μνήμη οργανώνεται σε περιοχές (ROM, DRAM, L1, και L2), κάθε μία με συγκεκριμένο ρόλο για μέγιστη αποδοτικότητα. Η ROM αποθηκεύει μόνο δεδομένα ανάγνωσης, η DRAM εξυπηρετεί δυναμική αποθήκευση, ενώ οι L1 και L2 δεσμεύονται για προσωρινά δεδομένα (buffers) κρίσιμων υπολογισμών, προσφέροντας ταχύτερη πρόσβαση και βελτιωμένη αξιοποίηση της μνήμης.

scatter.txt

```
ROM 0x00000000 0x00002ecc           ; Load mode ROM size
{
    ROM 0x00000000 0x00002ecc       ; Execution mode ROM size,
    {
        *.o ( +RO )                 ; including all object files.
    }
    DRAM 0x00002ecc 0x002d8934       ; Simple dynamic ram.
    {
        * ( .dram )                  ; Match sections tagged as ".dram"
        *.o ( +RW, +ZI )             ; and include all object files
    }
    L1 0x002db800 0x000b5a48         ; L1 region for our buffer.
    {
        * ( .L1 )                    ; match all sections with ".L1"
    }
    L2 0x002db7f0 0x000b5a48         ; L2 region for our buffer.
    {
        * ( .L2 )                    ; match all sections with ".L2"
    }
}
```

memory.map

```
00000000 00002ecc ROM 4 R 1/1 1/1
0000dffc 002d8934 DRAM 4 RW 250/50 250/50
002e6930 000b5a48 L1 4 RW 1/1 1/1
003a6928 000b5a48 L2 4 RW 10/10 10/10
```

stack.c

```
#include <rt_misc.h>

__value_in_regs struct __initial_stackheap __user_initial_stackheap(unsigned R0,
unsigned SP, unsigned R2, unsigned SL) {
    struct __initial_stackheap config;
    config.heap_base = 0x00f00ffc;
    config.stack_base = 0xff3f4dc;
    return config;
}
```

Υλοποίηση μετασχηματισμών επαναχρησιμοποίησης δεδομένων

Υλοποίηση με χρήση του πίνακα `current_line`

```
#include <string.h>
#include <stdio.h>
#include "opts.h"

__global_reg(1) int i; // r4
__global_reg(2) int j; // r5
__global_reg(3) int k; // r6
__global_reg(4) int b; // r7
__global_reg(5) int a; // r8 (buffer counter)
__global_reg(6) int c; // r9 (distance counter)
__global_reg(7) int d; // r10 (current_line counter)

#pragma arm section rwdata=".dram"
__align(8) int f[2] = {-1, 0};
#pragma arm section

#pragma arm section zidata=".L1"
__align(8) int current_line[ROWS][M];
#pragma arm section

#pragma arm section zidata=".L2"
__align(8) int distance[N][M];
__align(8) int buffer[TILE_ROWS][M];
#pragma arm section

__inline int strength_reduction(int x, int y) {
    d++;
    k = !current_line[x][y];
    return f[k];
}

#pragma O2
#pragma Otime
void distance_transformer() {
    __asm("MOV r6, #1");

    memset(distance, -1, N * M * sizeof(int));
    memset(buffer, 0, TILE_ROWS * M * sizeof(int));
    memset(current_line, 0, ROWS * M * sizeof(int));

    for (i = 0; i < N; i += TILE_ROWS) {
        a += TILE_ROWS * M;
        memcpy(buffer, &current_y[i][0], TILE_ROWS * M * sizeof(int));
        for (b = 0; b < TILE_ROWS; b += ROWS) {
            d += ROWS * M;
            memcpy(current_line, &buffer[b][0], ROWS * M * sizeof(int));
            for (j = 0; j < M; j += 4) {
                c++;
                d++;
                distance[i + b][j] = strength_reduction(b % ROWS, j);
                if (j + 1 < M) {
                    c++;
                    d++;
                    distance[i + b][j + 1] = strength_reduction(b % ROWS, j + 1);
                }
                if (j + 2 < M) {
                    c++;
                    d++;
                    distance[i + b][j + 2] = strength_reduction(b % ROWS, j + 2);
                }
                if (j + 3 < M) {
                    c++;
                    d++;
                    distance[i + b][j + 3] = strength_reduction(b % ROWS, j + 3);
                }
            }
        }
    }
}
```

```

    }
  }
}

LOOP:
for (i = 0; i < N; ++i) {
  for (j = 0; j < M; j += 2) {
    c++;
    if (distance[i][j] == k - 1) {
      if (i > 0) c++;
      if (i > 0 && distance[i - 1][j] == -1) distance[i - 1][j] = k;

      if (i < N - 1) c++;
      if (i < N - 1 && distance[i + 1][j] == -1) distance[i + 1][j] = k;

      if (j > 0) c++;
      if (j > 0 && distance[i][j - 1] == -1) distance[i][j - 1] = k;

      if (j < M - 1) c++;
      if (j < M - 1 && distance[i][j + 1] == -1) distance[i][j + 1] = k;
    }

    if (j + 1 < M) {
      c++;
      if (distance[i][j + 1] == k - 1) {
        if (i > 0) c++;
        if (i > 0 && distance[i - 1][j + 1] == -1) distance[i - 1][j + 1] = k;

        if (i < N - 1) c++;
        if (i < N - 1 && distance[i + 1][j + 1] == -1) distance[i + 1][j + 1] = k;

        if (j > -1) c++;
        if (j > -1 && distance[i][j] == -1) distance[i][j] = k;

        if (j < M - 2) c++;
        if (j < M - 2 && distance[i][j + 2] == -1) distance[i][j + 2] = k;
      }
    }
  }
}

if (k < 255) {
  k++;
  goto LOOP;
}

memcpy(current_y, distance, N * M * sizeof(int));

printf("Accesses to current_line: %d\n", d);
printf("Accesses to distance: %d\n", c);
printf("Accesses to buffer: %d\n", a);
}

```

Αξιοσημείωτη αποτελεί η χρήση της `strength_reduction` συνάρτησης, όπου υλοποιείται μια απλοποιημένη και αποδοτική λογική για την επεξεργασία δεδομένων ενός πίνακα. Η συνάρτηση, δηλωμένη ως `inline` για να μειώσει το κόστος κλήσης, υπολογίζει το λογικό συμπλήρωμα μιας τιμής από τον πίνακα `current_line` στις συντεταγμένες (x, y) και επιστρέφει την αντίστοιχη τιμή από τον πίνακα `f`. Η χρήση της εντός του υπολογισμού του πίνακα `distance` συνδυάζει αποδοτικότητα και ευκολία επαναχρησιμοποίησης, διασφαλίζοντας την ταχύτερη εκτέλεση κρίσιμων λειτουργιών και την απλότητα στον σχεδιασμό του κώδικα.

Υλοποίηση με χρήση του πίνακα block

```
#include <string.h>
#include <stdio.h>
#include "opts.h"

__global_reg(1) int i; // r4
__global_reg(2) int j; // r5
__global_reg(3) int k; // r6
__global_reg(4) int b; // r7
__global_reg(5) int a; // r8 (buffer counter)
__global_reg(6) int c; // r9 (distance counter)
__global_reg(7) int d; // r10 (block counter)

#pragma arm section rwdata=".dram"
__align(8) int x = 0;
__align(8) int y = 0;
__align(8) int bi = 0;
__align(8) int bj = 0;
__align(8) int global_x = 0;
__align(8) int global_y = 0;
__align(8) int f[2] = {-1, 0};
#pragma arm section

#pragma arm section zidata=".L1"
__align(8) int block[ROWS][ROWS];
#pragma arm section

#pragma arm section zidata=".L2"
__align(8) int distance[N][M];
__align(8) int buffer[TILE_ROWS][M];
#pragma arm section

__inline int strength_reduction(int x, int y) {
    d++;
    k = !block[x][y];
    return f[k];
}

#pragma O2
#pragma Otime
void distance_transformer() {
    __asm("MOV r6, #1");

    memset(distance, -1, N * M * sizeof(int));
    memset(buffer, 0, TILE_ROWS * M * sizeof(int));

    for (i = 0; i < N; i += TILE_ROWS) {
        a += TILE_ROWS * M;
        memcpy(buffer, &current_y[i][0], TILE_ROWS * M * sizeof(int));
        for (b = 0; b < TILE_ROWS; b += ROWS) {
            for (bi = 0; bi < TILE_ROWS; bi += ROWS) {
                for (bj = 0; bj < M; bj += ROWS) {
                    for (x = 0; x < ROWS; x++) {
                        for (y = 0; y < ROWS; y++) {
                            block[x][y] = buffer[b + x][bj + y];
                        }
                    }

                    for (x = 0; x < ROWS; x++) {
                        for (y = 0; y < ROWS; y++) {
                            global_x = i + b + x;
                            global_y = bj + y;
                            if (global_x < N && global_y < M) {
                                c++;
                                d++;
                                distance[global_x][global_y] = strength_reduction(x, y);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

    }
}

LOOP:
for (i = 0; i < N; ++i) {
    for (j = 0; j < M; j += 2) {
        c++;
        if (distance[i][j] == k - 1) {
            if (i > 0) c++;
            if (i > 0 && distance[i - 1][j] == -1) distance[i - 1][j] = k;

            if (i < N - 1) c++;
            if (i < N - 1 && distance[i + 1][j] == -1) distance[i + 1][j] = k;

            if (j > 0) c++;
            if (j > 0 && distance[i][j - 1] == -1) distance[i][j - 1] = k;

            if (j < M - 1) c++;
            if (j < M - 1 && distance[i][j + 1] == -1) distance[i][j + 1] = k;
        }

        if (j + 1 < M) {
            c++;
            if (distance[i][j + 1] == k - 1) {
                if (i > 0) c++;
                if (i > 0 && distance[i - 1][j + 1] == -1) distance[i - 1][j + 1] = k;

                if (i < N - 1) c++;
                if (i < N - 1 && distance[i + 1][j + 1] == -1) distance[i + 1][j + 1] = k;

                if (j > -1) c++;
                if (j > -1 && distance[i][j] == -1) distance[i][j] = k;

                if (j < M - 2) c++;
                if (j < M - 2 && distance[i][j + 2] == -1) distance[i][j + 2] = k;
            }
        }
    }
}

if (k < 255) {
    k++;
    goto LOOP;
}

memcpy(current_y, distance, N * M * sizeof(int));

printf("Accesses to block: %d\n", d);
printf("Accesses to distance: %d\n", c);
printf("Accesses to buffer: %d\n", a);
}

```

Ο παραπάνω κώδικας μετρά τις προσπελάσεις στους buffers μέσω των μεταβλητών μετρητών a, c, και d, οι οποίες έχουν οριστεί σε συγκεκριμένα global registers για αποδοτικότητα. Η μεταβλητή a αυξάνεται κάθε φορά που εκτελείται η λειτουργία memcpy ή άλλες πράξεις που αφορούν το buffer buffer, καταγράφοντας το συνολικό πλήθος στοιχείων που μεταφέρονται. Ο μετρητής d αυξάνεται σε κάθε προσπέλαση του πίνακα block, όπως μέσα στη συνάρτηση strength_reduction, όπου κάθε προσπέλαση του πίνακα προσθέτει μία αύξηση. Ο μετρητής c παρακολουθεί όλες τις προσπελάσεις του πίνακα distance κατά τη διάρκεια των υπολογισμών, περιλαμβάνοντας τόσο τις αναγνώσεις όσο και τις εγγραφές, ειδικά στους βρόχους επεξεργασίας. Αυτή η στρατηγική επιτρέπει την παρακολούθηση και τη βελτιστοποίηση της χρήσης των buffers στο πρόγραμμα.

Σύγκριση των αποτελεσμάτων

Στο παρακάτω μέρος του report, στόχος μας είναι η σύγκριση τριών διαφορετικών εκδόσεων του αλγορίθμου που έχουμε υλοποιήσει. Η πρώτη έκδοση είναι αυτή που δεν χρησιμοποιεί buffer και όλα τα δεδομένα διαχειρίζονται στη μνήμη RAM. Στη δεύτερη και τρίτη έκδοση, έχουμε ενσωματώσει τη χρήση buffer, εφαρμόζοντας τις τεχνικές `current_line` και `block array` για την αντιγραφή δεδομένων από τη μνήμη RAM στις μνήμες cache (L1 και L2). Για να εκτελέσουμε τη σύγκριση, χρησιμοποιήσαμε τον ήδη υλοποιημένο αλγόριθμο του πρώτου και δεύτερου μέρους του project, εισάγοντας κατάλληλες `#pragma directives` ώστε να καθοδηγήσουμε τον compiler στην τοποθέτηση των δεδομένων και των buffers στη μνήμη RAM και στις διάφορες επίπεδα της cache. Στη συνέχεια, πραγματοποιήσαμε μετρήσεις για την ταχύτητα εκτέλεσης και για τον αριθμό των προσπελάσεων σε κάθε πίνακα, με τη βοήθεια του προσομοιωτή ARMulator.

Στο επόμενο τμήμα παρατίθενται τα αποτελέσματα που προέκυψαν από την εκτέλεση των τριών εκδόσεων του αλγορίθμου.

# προσπελάσεων	Χωρίς buffer	Υλοποίηση block (L1 & L2 cache)	Υλοποίηση current_line (L1 & L2 cache)
current_line/block array	–	1476096	307520
distance array	47403692	48518864	47839311
buffer	–	190464	184512
Σύνολο	47403692	50185424	48331343

	Wait states	Total Cycles
Αρχική υλοποίηση	–	552933965
Υλοποίηση current_line	22344730	676637195
Υλοποίηση block	101356762	804641546

Βιβλιογραφικές αναφορές

- [1] distance-transform JavaScript library. Distance Field transformation.
Retrieved from <https://www.npmjs.com/package/@thi.ng/distance-transform>
- [2] The Distance Transform and its Computation – An Introduction.
Retrieved from <https://arxiv.org/pdf/2106.03503>
- [3] Strength reduction.
Retrieved from https://en.wikipedia.org/wiki/Strength_reduction
- [4] Options That Control Optimization in GCC. Retrieved from
<https://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/Optimize-Options.html>
- [5] Specifying Attributes of Variables in GCC. Retrieved from
<https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html>
- [6] Compiler Specific Features. Retrieved from
<https://developer.arm.com/documentation/dui0472/m/Compiler-specific-Features>
- [7] Ν. Η. Παπαμάρκος, Ψηφιακή Επεξεργασία και Ανάλυση Εικόνας, σσ. 2.42-2.47
- [8] <https://developer.arm.com/documentation/dui0378/f/the-c-and-c--library-functions-reference/legacy-function--user-initial-stackheap--?lang=en>
- [9] <https://developer.arm.com/documentation/dui0041/c/Linker/The-scatter-load-description-file/Structure-of-the-description-file>
- [10] <https://developer.arm.com/documentation/dui0375/c/Using-the-Inline-and-Embedded-Assemblers-of-the-ARM-Compiler/Inline-assembler-rules-for-compiler-keywords--asm-and-asm?lang=en>
- [11] <https://developer.arm.com/documentation/dui0041/c/ARM-Compiler-Reference/Compiler-specific-features/Pragmas?lang=en>
- [12] <https://developer.arm.com/documentation/den0042/a/Caches/Memory-hierarchy>
- [13] Διαφάνειες εργαστηριακών διαλέξεων του μαθήματος “Σχεδιασμός Ενσωματωμένων Συστημάτων”