ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδίαση Συστημάτων Υλικού - Λογισμικού

Εργαστήριο 1: Εξοικείωση με το εργαλείο Vitis HLS

Από τους φοιτητές:
Σωτήριος Καραντζούλης 10952
Κωνσταντίνος Γκαρίπης 10924

Διδάσκων: Ιωάννης Παπαευσταθίου
Υπεύθυνοι Εργαστηρίου: Α. Αθανασιάδης - Δ. Καρανάσσος

December 5, 2025

Περιεχόμενα

1 Ερώτημα 1: Κώδικας Επιταχυντή και Testbench

1.1 Περιγραφή Λειτουργικότητας

Σε αυτό το ερώτημα σχεδιάστηκε ο hardware accelerator `IMAGE_DIFF_POSTERIZE`. Ο επιταχυντής δέχεται δύο εικόνες A και B (πίνακες μεγέθους `HEIGHT` × `WIDTH`), υπολογίζει την απόλυτη διαφορά των pixel τους και εφαρμόζει κατωφλίωση (thresholding) με βάση τις τιμές $T1 = 32$ και $T2 = 96$.

1.2 Source Code (Accelerator)

Παρακάτω παρατίθεται ο κώδικας της συνάρτησης `IMAGE_DIFF_POSTERIZE`:

```
#define WIDTH 512
#define HEIGHT 512
#define T1 32
#define T2 96
#include <stdint.h>

void IMAGE_DIFF_POSTERIZE(uint8_t A [HEIGHT][WIDTH], uint8_t B[HEIGHT][
    WIDTH], uint8_t C[HEIGHT][WIDTH]){

    #pragma HLS INTERFACE mode=bram port=A
    #pragma HLS INTERFACE mode=bram port=B
    #pragma HLS INTERFACE mode=bram port=C
    #pragma HLS INTERFACE s_axilite port=return
  uint8_t D; // difference of pixel values
  int16_t temp_d;

// Partition the arrays so more there can be more access in one
    iteration with loop unrolling and pipelining
// #pragma HLS ARRAY_PARTITION variable=A type=cyclic factor=64 dim=2
// #pragma HLS ARRAY_PARTITION variable=B type=cyclic factor=64 dim=2
// #pragma HLS ARRAY_PARTITION variable=C type=cyclic factor=64 dim=2


  // iterate through A and B
  for(int i = 0 ; i < HEIGHT ; i++){

        // #pragma HLS pipeline

    for(int j =0 ; j < WIDTH ; j++){


      temp_d = A[i][j] - B[i][j];
      D = (temp_d < 0)? -temp_d : temp_d; // get the difference of each
    corresponding pixels
      if(D < T1){
        C[i][j] = 0;
      }
      else if(D >= T1 && D <T2){
        C[i][j] = 128;
      }
      else{
        C[i][j] = 255;
      }
    }
```

```
  }

}
```

Listing 1: Κώδικας του Accelerator (Source.cpp)

1.3 Testbench Code

Για να γεμίσουμε τους πίνακες A και B επιλέξαμε μία απλή συνάρτηση της γραμμής και στήλης κάθε στοιχείου. Παρακάτω παρατίθεται ο κώδικας του Testbench που χρησιμοποιήθηκε για την επαλήθευση:

```
#include <stdio.h>
#include <stdint.h>

#define WIDTH 512
#define HEIGHT 512


void IMAGE_DIFF_POSTERIZE(uint8_t A [HEIGHT][WIDTH], uint8_t B[HEIGHT][
    WIDTH], uint8_t C[HEIGHT][WIDTH]);

int main(){
    uint8_t A[HEIGHT][WIDTH] ;
  uint8_t B[HEIGHT][WIDTH] ;

  uint8_t ref_data[HEIGHT][WIDTH] = {};
    // filling matrices with dummy values
    for(int i=0; i<HEIGHT; i++){

        for(int j=0; j<WIDTH; j++){
            A[i][j] = (uint8_t) (i*j) % 256;
            B[i][j] = (uint8_t) (i/(j+1)) % 256;
        }
    }

  uint8_t C[HEIGHT][WIDTH] = {};

  IMAGE_DIFF_POSTERIZE(A, B, C);

    FILE *ref= fopen("C:/Users/user/Desktop/ref_output.dat", "r");

    if(ref==NULL){

    FILE *out = fopen("C:/Users/user/Desktop/ref_output.dat", "w");
    if (!out) {
        printf("File open error");
        return 1;
    }

      for(int i = 0 ; i < HEIGHT ; i++){

        for(int j =0 ; j < WIDTH ; j++){
          fprintf(out, "%3d ", C[i][j]);
        }
        fprintf(out, "\n");
      }
    fclose(out);
```

```
46
47      printf("Reference output from C simulation written.\n");
48      return 0;
49      }
50
51      for (int i = 0; i < HEIGHT; i++) {
52        for(int j = 0; j< WIDTH; j++ ){
53          if (fscanf(ref, "%3d", &ref_data[i][j]) != 1) {
54              printf("Error: reference file corrupted at index (%d,%d).\n"
      , i,j);
55              fclose(ref);
56              return 1;
57          }
58        }
59      }
60      fclose(ref);
61
62      bool match = 1;
63      for (int i = 0; i < HEIGHT; i++) {
64        for(int j = 0; j< WIDTH; j++ ){
65          if(C[i][j]!= ref_data[i][j]){
66            printf("Mismatch at index (%d,%d)", i,j);
67            match = 0;
68          }
69        }
70        }
71      if(match){
72        printf("Test PASSED!\n");
73      }
74      else{
75        printf("Test FAILED!\n");
76      }
77    return 0;
78
79  }
```

Listing 2: Κώδικας του Testbench (tb.cpp)

2 Ερώτημα 2: Αποτελέσματα Αρχικής Σύνθεσης (C Synthesis)

Η σύνθεση πραγματοποιήθηκε με τις default ρυθμίσεις του εργαλείου, χωρίς τη χρήση directives βελτιστοποίησης.

Διαστάσεις Πινάκων: HEIGHT = 256, WIDTH = 256.

Τα αποτελέσματα που προέκυψαν από το *Synthesis Report* παρουσιάζονται στον παρακάτω πίνακα:

Metric	**Value**
Estimated Clock Period	5.093 ns
Worst Case Latency	65538 cycles
Number of DSP48E	0
Number of BRAMs	0
Number of FFs	57
Number of LUTs	272

Σχολιασμός: Παρατηρούμε μικρή αξιοποίηση πόρων κατά την σύνθεση και μεγάλες τιμές στο latency, αποτελέσματα λογικά όταν δεν εφαρμόζουμε κάποια βελτιστοποίηση. Η μηδενική χρήση BRAM ήταν αναμενόμενη παρόλο που δηλώνεται σαν Interface στην υλοποίηση, διότι θα αποδοθεί σαν πόρος εξωτερικά του IP core το οποίο θα συνθέσει το Vitis. Στην συνέχεια, θα παρατηρήσουμε την αύξηση των πόρων που χρησιμοποιούμε όσο μειώνουμε το latency.

3 Ερώτημα 3: Αποτελέσματα C/RTL Cosimulation

Πραγματοποιήθηκε C/RTL Cosimulation για να επιβεβαιωθεί η ορθότητα της σχεδίασης σε επίπεδο RTL και να μετρηθεί ο ακριβής χρόνος εκτέλεσης.

Διαστάσεις Πινάκων: HEIGHT = 256, WIDTH = 256.

Πίνακας 1: Αποτελέσματα C/RTL Cosimulation

Metric	**Value**
Total Execution Time	655595 ns
Min Latency	65536 cycles
Avg Latency	65536 cycles
Max Latency	65536 cycles

4 Ερώτημα 4: Βελτιστοποίηση με Vitis HLS Directives

4.1 (i) Επίδραση Μεγέθους Εικόνας (Scaling)

Κρατώντας σταθερό το `HEIGHT = 256` και μεταβάλλοντας το `WIDTH`, παρατηρήθηκαν τα εξής σχετικά με το Latency και τον χρόνο εκτέλεσης:

Πίνακας 2: Επίδραση μεταβολής πλάτους (WIDTH) με σταθερό HEIGHT

HEIGHT	WIDTH	Total Latency (cycles)	Execution Time
256	64	256	2975
256	256	512	5355
256	512	1024	10655

Παρατηρήσεις: Παρατηρούμε ότι μεταβάλλοντας το `WIDTH` για σταθερό `HEIGHT = 256` το execution time και το latency παραμένουν αυξάνονται.

4.2 (ii) Βέλτιστη Υλοποίηση

Μετά από πειραματισμό με διάφορα directives (`PIPELINE`, `UNROLL`, `ARRAY_PARTITION`), η βέλτιστη υλοποίηση για διαστάσεις 256×256 δίνει τα παρακάτω αποτελέσματα:

Directives που χρησιμοποιήθηκαν:

- `#pragma HLS ARRAY_PARTITION variable=A type=cyclic factor=64 dim=2` για όλους τους πίνακες
- `#pragma HLS PIPELINE II=1` στο εσωτερικό loop

Πίνακας 3: Σύγκριση Πόρων και Επίδοσης Βέλτιστης Υλοποίησης

Metric	Value (Optimized)
Estimated Clock Period	4.939 ns
DSP48E Used	0
BRAMs Used	0
FFs Used	58
LUTs Used	26497
Timing (Cosimulation)	
Total Execution Time	5355 ns
Min Latency	512
Avg Latency	512
Max Latency	512

Αιτιολόγηση: Το ARRAY_PARTITION χρησιμοποιήθηκε για να χωριστούν οι πίνακες σε περισσότερα memory banks ώστε να επιτρέπεται η ταυτόχρονη ανάγνωση πολλών δεδομένων. Ο τύπος cyclic επιλέχθηκε ως πλέον κατάλληλος για column traversal όπως εδώ

(εξού και dim=2). Στο factor δόθηκε η τιμή 64 έπειτα από μία διαδικασία trial and error ώστε να βρεθεί το μέγιστο factor που δεν παράγει error στο cosimulation για 256x256 Πίνακες.
Το Pipeline χρησιμοποιήθηκε ως το κύριο directive που επιτυγχάνει το speedup. Με αυτόν τον τρόπο, στο εσωτερικό loop οι τιμές των πινάκων διαβάζονται με διοχέτευση χωρίς δηλαδή να περιμένει το επόμενο iteration το προηγούμενο. Άλλωστε, κάθε iteration είναι εντελώς ανεξάρτητο από όλα τα υπόλοιπα.

Σημείωση: Επιχειρήθηκε, επίσης, να χρησιμοποιηθεί το directive #pragma HLS unroll το οποίο υλοποιεί loop unrolling στο εσωτερικό loop, χωρίς κάποια ιδιαίτερη διαφορά. Η ιδέα ήταν να εκτελούνται κάποιες επαναλήψεις ταυτόχρονα με loop unrolling και να αρχίζουν και οι επόμενες με pipeling. Δυστυχώς, ο συνδιασμός αυτός απορρίφθηκε χάρην απλότητος στον κώδικα, εφόσον δεν παρουσιαζόταν διαφορά στο execution time.

4.3 (iii) Υπολογισμός Επιτάχυνσης (Speed-up)

Η επιτάχυνση (Speed-up) που επιτεύχθηκε σε σχέση με την αρχική (non-optimized) υλοποίηση είναι:

$$Speedup = \frac{T_{initial}}{T_{optimized}} = \frac{655595}{5355} = \mathbf{122.43} \tag{1}$$

5 AXI - Stream Interface & Caching

5.1 Staged Caching

Το πρωτόκολλο $AXI-Stream$ είναι σχεδιασμένο για ταχεία ανταλλαγή δεδομένων μεταξύ των IP cores της $FPGA$. Κύριο χαρακτηρηστικό του είναι η έλλειψη overhead που θα είχε ένα πρωτόκολλο με διευθύνσεις ενώ μπορέι να επιτύχει μέγιστη μετάδοση 512 bits/cycle. Για την παρούσα εφαρμογή αυτό σημαίνει οτι μπορεί να μεταφέρει συνολικά 64 στοιχεία του πίνακα μας ανα κύκλο. Χωρίζουμε την εκτέλεση του προγράμματος σε τρία στάδια: Input Caching, Processing, Output Caching. Πρώτα θα μελετήσουμε την περίπτωση όπου περιμένουμε την ολοκλήρωση κάθε σταδίου πριν προχωρήσουμε στο επόμενο, χρησιμοποιώντας ένα for loop για το καθένα.

Βλέπουμε αρχικά το utilization του κώδικα μετά την υλοποίηση του $AXIS$ και την ενσωμάτωση Caching. Έχουμε, πλέον, χρήση BRAM χωρίς ρητή ανάθεση των δεδομένων μας με Binding. Λαμβάνουμε τελικά εκτέλεση σε 67.591 κύκλους εκ των οποίων 1024 για κάθε cache στάδιο (64 bytes/cycle x 1024 cycles = 65.536 = 256^2 για cache όλου του πίνακα) και 65.537 cycles για το στάδιο των υπολογισμών (όσα τα στοιχεία + 1 μάλλον καθυστέρηση).

Name	**BRAM_18K**	**DSP**	**FF**	**LUT**	**URAM**
Expression	-	-	0	2	-
Instance	-	-	107	886	-
Memory	96	-	0	0	0
Multiplexer	-	-	0	5068	-
Register	-	-	523	-	-
Total	**96**	**0**	**630**	**5956**	**0**
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	6	0	~ 0	1	0

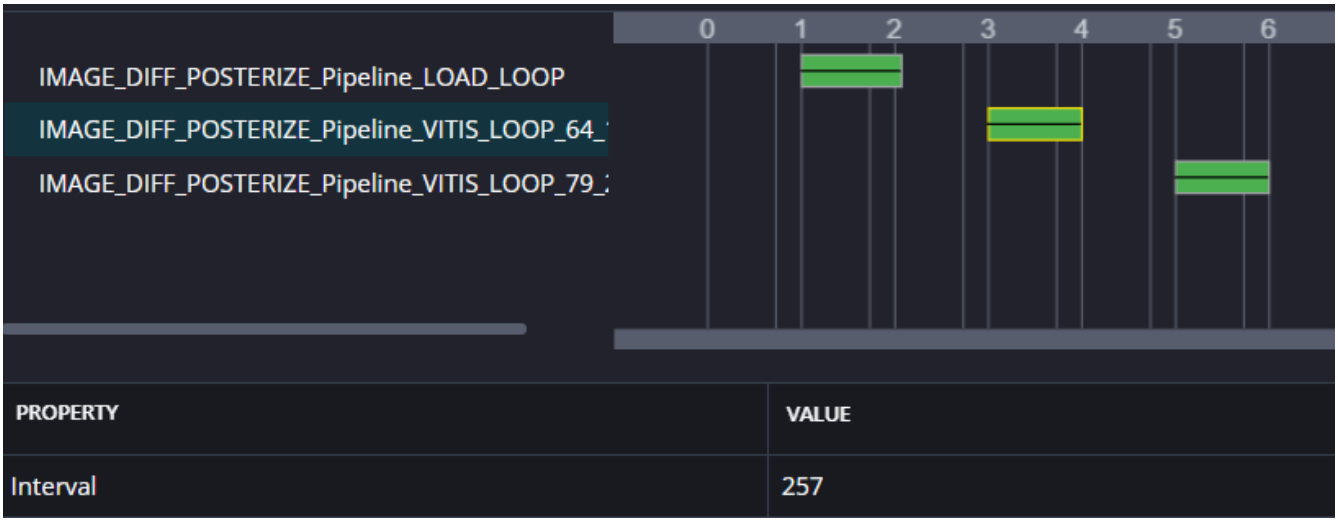
Πίνακας 4: HLS Synthesis Resource Utilization

Η βελτιστοποίηση στην περίπτωση του $AXIS$ αφορά τρία loops. Τα δύο όπως στις προηγούμενες παραγράφους για να διατρέξουμε γραμμές και στήλες. Το τρίτο υλοποιεί την διαχείρηση του ενός πακέτου των 64 *byte* που προωθεί το πρωτόκολλο. Εφόσον αυτό θα συμβαίνει σε κάθε κύκλο χρησιμοποιούμε αυτήν την φορά το UNROLL directive για την επεξεργασία του πακέτου στον απαραίτητο χρόνο. Στους υπόλοιπους βρόγχους εφαρμόζουμε το PIPELINE directive όπως και πριν. Παρατηρώντας τα στοιχεία για το utilization γίνεται άμεσα αντιληπτή η αυξημένη κατανάλωση των πόρων κυρίως για την υλοποίηση της μνήμης. Αυτή τη φορά δεν χρησιμοποιείται BRAM, γεγονός που αποδίδουμε στην δράση του ARRAY_PARTITIONING ($type = cyclic$, $factor = 64$). Πριν προχωρήσουμε στα δεδομένα να σημειωθεί ότι δοκιμάστηκε και το "$type = complete$" το οποίο έδωσε την ίδια κατανάλωση και τους ίδιους χρόνους. 'Ισως αυτό συνδέεται με το γεγονός ότι το πλάτος του $AXIS$ είναι όσο το $factor$.

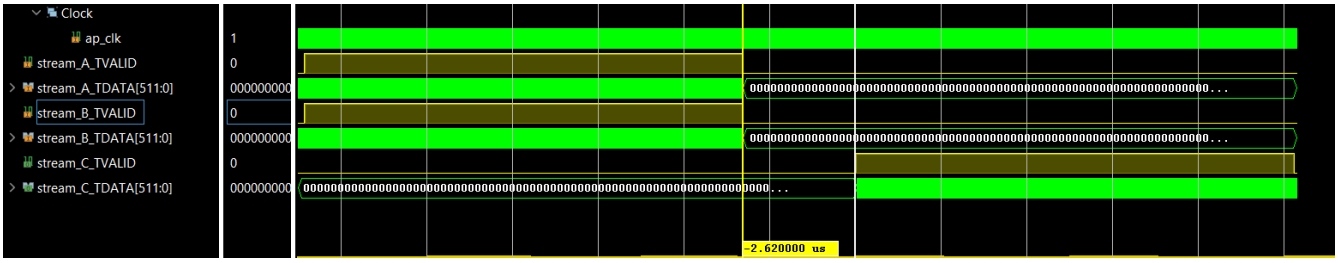
Name	BRAM_18K	DSP	FF	LUT	URAM
Expression	-	-	0	2	-
Instance	-	-	1616	21126	-
Memory	0	-	6144	25344	0
Multiplexer	-	-	0	28492	-
Register	-	-	523	-	-
Total	**0**	**0**	**8283**	**74964**	**0**
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	0	1	19	0

Πίνακας 5: HLS Synthesis Resource Utilization

Ολοκληρώνοντας για την συγκεκριμένη προσέγγιση ενδιαφέρον παρουσιάζουν τα αποτελέσματα σε Schedule & Wave Viewers απο τα οποία εξάγουμε συμπεράσματα για την απόδοση καθώς και για την σημασία που έχει ο σειριακός τρόπος διαχείρησης των δεδομένων όταν διερευνούμε την απόδοση μόνον του υπολογιστικού σταδίου. Παρατηρούμε την ολοκλήρωση του τελευταίου σε μόλις 257 κύκλους ρολογιού, οπότε έχουμε πετύχει τέλειο pipeline για κάθε γραμμή του πίνακα.



Σχήμα 1: *Schedule Viewer*



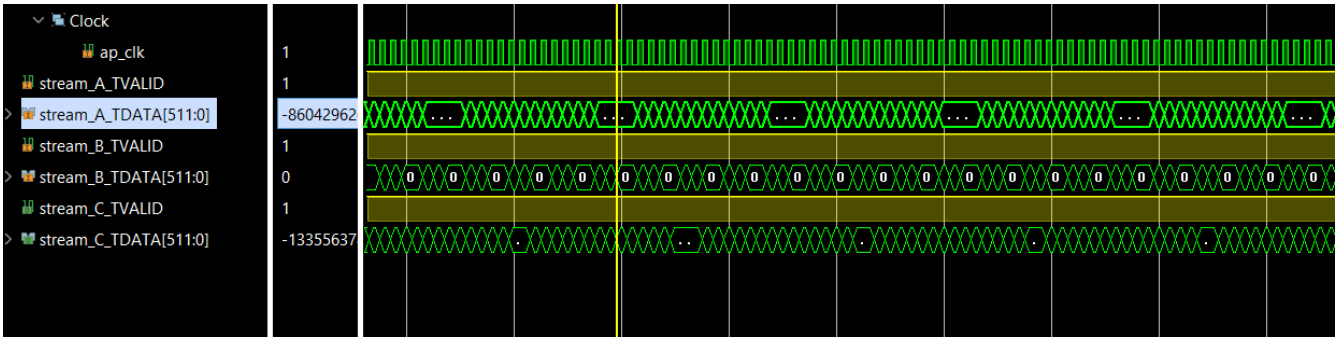
Σχήμα 2: *Wave Viewer*

5.2 Pipelined Caching

Φέρνοντας τα τρία στάδια που περιγράψαμε κάτω από κοινό βρόγχο για την διάσχιση των γραμμών ακολουθούμε μία pipelined λογική με χρόνο εκτέλεσης $t = 10625ns$. Εφαρμόζουμε και πάλι συνδυασμό των τριών directives όπως και πριν. Παρακάτω βλέπουμε την διαφορά στο utilization με την απλή υλοποίηση. Δεν χρησιμοποιείται BRAM αλλά σημαντικός αριθμός Flip-Flop και LUT's, ενώ το utilization φτάνει το 10%. Βλέπουμε μείωση στο μισό λόγω της χρήσης Pipelined λογικής διότι, δεν χρειάζεται κάθε φορά να διαχειριζόμαστε το σύνολο της πληροφορίας αλλά μόνο το μέρος αυτής που έχει εισέλθει στην σωληνογραμμή, όσο δηλαδή επιτρέπει το AXIS bandwidth. Διαπιστώνουμε, ακόμη, ότι το αποτέλεσμα αυτό είναι κοντά στο βέλτιστο. Δείξαμε ότι το $AXIS$ χρειάζεται 1024 κύκλους για να ολοκληρώσει το caching και για 1 $cycle = 10ns$ θα έχουμε ένα φράγμα για $t = 10240ns$. Υπο την σκέψη αυτή επιχειρήσαμε να υλοποιήσουμε τον υπολογισμό αμέσως στην λήψη κάθε πακέτου και την άμεση επιστροφή του, επιτυγχάνοντας 5% utilization και ελάχιστα καλύτερο χρόνο στα $t = 10390ns$ χωρίς χρήση πινάκων και Array Partition.

Name	**BRAM_18K**	**DSP**	**FF**	**LUT**	**URAM**
Expression	-	-	0	20794	-
Instance	0	-	36	40	0
Memory	0	-	3608	14883	0
Multiplexer	-	-	0	6456	-
Register	-	-	3978	-	-
Total	**0**	**0**	**7622**	**42173**	**0**
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	0	~ 0	10	0

Πίνακας 6: HLS Synthesis Resource Utilization



Σχήμα 3: *Pipelined Caching*

5.3 Σύγκριση & Συμπεράσματα

Θα δούμε τελικά ότι σε κάθε περίπτωση πετυχαίνουμε κάτι διαφορετικό. Χωρίς pipelining θα γίνει αισθητό το πλεονέκτημα που μας δίνεται στην διαχείρηση των δεδομένων μας όταν έχει προηγηθεί caching, το οποίο θα αυξήσει την απόδοση του καθαρού υπολογιστικού σταδίου. Από την άλλη πλευρά μπορέσαμε να λάβουμε και ένα δεύτερο συμπέρασμα: Η χρήση μίας pipelined δομής για μία τέτοια υλοποίηση, μπορεί να βελτιώσει την συνολική ταχύτητα της εκτέλεσης αλλά και να μειώσει σημαντικά τους πόρους που καταναλώνουμε.

Μέθοδος	**Total Cycles**	**Execution Time** (ns)	**Speedup (Mixed)**
No Optimization	67591	676115 ns	-
Serial Caching	2317	23315	29.1
Pipelined Caching	1029	10625	65.7

Πίνακας 7: Σύγκριση Επιδόσεων

Την καλύτερη απόδοση λάβαμε τελικά στην χρήση απλού caching όπου απομονώνουμε το υπολογιστικό στάδιο. Η τελική μέγιστη επιτάχυνση (Speed-up) που επιτεύχθηκε σε σχέση με την αρχική (non-optimized) υλοποίηση είναι:

$$Speedup = \frac{T_{initial}}{T_{optimized}} = \frac{67591}{257} = \mathbf{263} \tag{2}$$