

ΗΜΜΥ ΠΟΛΥΤΕΧΝΕΙΟΥ ΚΡΗΤΗΣ

ΗΡΥ591-ΑΝΑΔΙΑΤΑΣΣΟΜΕΝΑ ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ 2019-2020

Υλοποίηση Αλγόριθμου σε Hardware με τη
χρήση τεχνικών high-level synthesis και του
περιβάλλοντος SDSoC

Φοιτητές

ΚΑΛΟΓΕΡΑΚΗΣ ΣΤΕΦΑΝΟΣ

ΖΑΧΑΡΙΟΥΔΑΚΗΣ ΝΙΚΟΛΑΣ

Διδάσκων

Ν. ΑΛΑΧΙΩΤΗΣ



**ΠΟΛΥΤΕΧΝΕΙΟ
ΚΡΗΤΗΣ**

Περιεχόμενα

1	Εισαγωγή	2
2	Συντομη περιγραφή εργαλείων	2
2.1	Xilinx Vivado HLS	2
2.2	Xilinx SDSoc Development Environment	2
3	Περιγραφή της αρχιτεκτονικής του Zynq-7000 AP-SoC	2
4	Περιγραφή της λειτουργίας της συνάρτησης myFunc	4
5	Μηχανισμός Ελέγχου ορθότητας τροποποιήσεων	7
6	Περιγραφή της αρχιτεκτονικής του επιταχυντή	7
7	Δικαιολόγηση σχεδιαστικών αποφάσεων και τροποποιήσεων του κώδικα	9
8	Αναφορά των HLS ντιρεκτίβων που χρησιμοποιήθηκαν, περιγραφή της λειτουργίας τους, και δικαιολόγηση της χρήσης τους	10
9	Σχεδιαστικά στάδια	12
10	Αναφορά των SDS ντιρεκτίβων που χρησιμοποιήθηκαν, του sds_lib API που χρησιμοποιήθηκαν, περιγραφή της λειτουργίας τους και δικαιολόγηση χρήσης τους	13
11	Αξιολόγηση απόδοσης	15
12	Συμπεράσματα	16
13	Παράρτημα	17
14	Βιβλιογραφία	18

1 Εισαγωγή

Στα πλαίσια της εργασίας εξαμήνου στο μάθημα Αναδιατασσόμενα Ψηφιακά Συστήματα κληθήκαμε να μελετήσουμε και να εμβαθύνουμε στην αναδιατασσόμενη λογική χρησιμοποιώντας τεχνικές όπως High-Level Synthesis αλλά και του περιβάλλοντος SDSoC . Πιο συγκεκριμένα, στόχος ήταν η δημιουργία αρχιτεκτονικής για την επιτάχυνση ενός αλγορίθμου(περιγράφεται στην συνέχεια), όπως και η βέλτιστη δυνατή χρήση πόρων του Zedboard development board επιτυγχάνοντας την καλύτερη δυνατή απόδοση για την συνάρτηση όταν αυτή καλείται από το processing system του Zynq-7000 AP-SoC . Τα εργαλεία που χρησιμοποιήθηκαν για αυτόν τον σκοπό είναι:

- Xilinx Vivado HLS 2018.3
- Xilinx SDSoC Development Environment 2018.3

2 Συντομη περιγραφή εργαλείων

2.1 Xilinx Vivado HLS

Το εργαλείο Vivado High-Level Synthesis(HLS) μετατρέπει μια υλοποίηση προδιαγραφών C σε υλοποίηση register transfer level(RTL) που δύναται να πραγματοποιήσει synthesize σε μια FPGA(field programmable gate array). Στοχεύοντας στην εκτέλεση σε μία FPGA, το εργαλείο επιτρέπει στον χρήστη να βελτιστοποιήσει τον κώδικα του σε επίπεδα throughput, latency και power. Οι γλώσσες προδιαγραφών C που υποστηρίζονται είναι η C , $C++$ και $SystemC$.

2.2 Xilinx SDSoC Development Environment

Το περιβάλλον SDSoC παρέχει ένα framework για ανάπτυξη hardware accelerated embedded processor application χρησιμοποιώντας κοινές γλώσσες προγραμματισμού. Περιλαμβάνει επίσης ένα περιβάλλον με flow παρόμοιο του Eclipse IDE ενώ με την χρήση του sdsc/sds++ compiler αναλύει το εκάστοτε πρόγραμμα για να αποφασίσει το dataflow μεταξύ software και hardware συναρτήσεων. Ο ίδιος compiler επιπρόσθετα παράγει hardware IP και software control κώδικα ικανά να συγχρονίσουν τον hardware accelerator και το software της εφαρμογής.

3 Περιγραφή της αρχιτεκτονικής του Zynq-7000 AP-SoC

Πριν ξεκινήσουμε με την ανάλυση της σχεδίασης του accelerator προβαίνουμε σε συνοπτική περιγραφή του board που χρησιμοποιούμε για τελική εκτέλεση.Η αρχιτεκτονική του Zynq-7000 SoC αποτελείται από δύο μεγάλα sections με τα εξής συστατικά:

1. PS: processing system που υποστηρίζει software routines, operating systems κλπ
 - (α') δύο ARM Cortex-A9 επεξεργαστές, συχνότητας $866MHz$ έως $1GHz$ πολλαπλά περιφερειακά

(β') hard silicon core

2. PL: programmable logic που είναι ιδανικό για την υλοποίηση high speed hardware συναρτήσεων χρησιμοποιώντας κάποια γλώσσα περιγραφής υλικού όπως VHDL, Verilog

(α') logic cells: 28k - 444k(430k έως 6.6M πύλες)

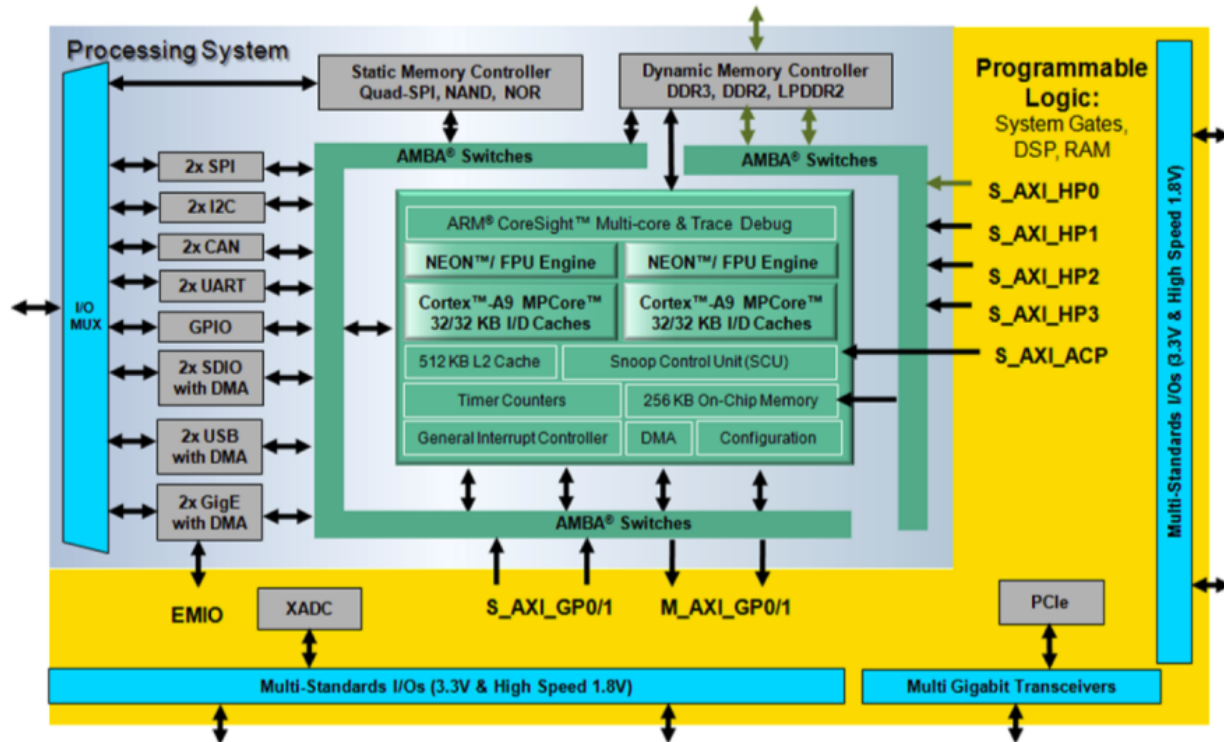
(β') flip-flops: 35k-554k

(γ') DSP/MAC: 80-2050

(δ') μέγιστη απόδοση DSP: 100 - 2622 GMACs

(ε') AD μετατροπέας: δύο 12bits

Τέλος, αξίζει να επισημάνουμε ότι υπάρχει ένα καλά ορισμένο interface μεταξύ αυτών των section, που χρησιμοποιείται ευρέως γνωστό ως AXI(Advanced eXtensible Interface)



Σχήμα 1: Αρχιτεκτονική του Zynq-7000 AP-SoC

4 Περιγραφή της λειτουργίας της συνάρτησης myFunc

Η συνάρτηση που δόθηκε για την υλοποίηση του accelerator είναι η παρακάτω

```
1 void myFunc (unsigned int size, unsigned int dim, dataType_t threshold,
2             dataType_t * data0, dataType_t * data1, dataType_t * data2)
3 {
4     unsigned int i, k, l;
5
6     for ( i = 0 ; i < size ; i ++ )
7     {
8         for ( k = 0 ; k < dim ; k ++ )
9         {
10             data2 [ i*dim + k ] = 0.0 ;
11         }
12
13         for ( k = 0 ; k < dim ; k ++ )
14         {
15             for ( l = 0 ; l < dim ; l ++ )
16             {
17                 data2 [ i*dim + k ] += data0 [ k * dim + l ] * data1 [ i*dim+ l ];
18             }
19         }
20
21         int r = 1 ;
22
23         for ( l = 0 ; r && ( l < dim ) ; l ++ )
24         {
25             r = ( data2 [ i*dim + l ] > threshold ) ;
26         }
27
28         if ( r )
29         {
30             for ( l = 0 ; l < dim ; l ++ )
31             {
32                 data2 [ i*dim + l ] = 0.0;
33             }
34         }
35     }
36 }
37
38
```

Listing 1: myFunc

Όπως μπορούμε να παρατηρήσουμε η myFunc() αξιοποιεί δύο arrays τα data0, data1 και μετά από επεξεργασία γεμίζει με δεδομένα τον πίνακα data2. Αξίζει να αναφερθεί ότι οι παράμετροι size και dim χρησιμοποιούνται για να διατρέξουμε όλα τα στοιχεία των πινάκων με τον data0 να είναι διαστάσεων dim*dim (data0[dim*dim]), ενώ το data1, data2 διαστάσεων dim*size(data1[dim*size], data2[dim*size]) όπως ορίστηκαν απο την main συνάρτηση. Η συ-

νάρτηση threshold χρησιμοποιείται σαν όριο ελέγχου και αντίστοιχα ενημέρωσης ανά περίπτωση για τον πίνακα data2.

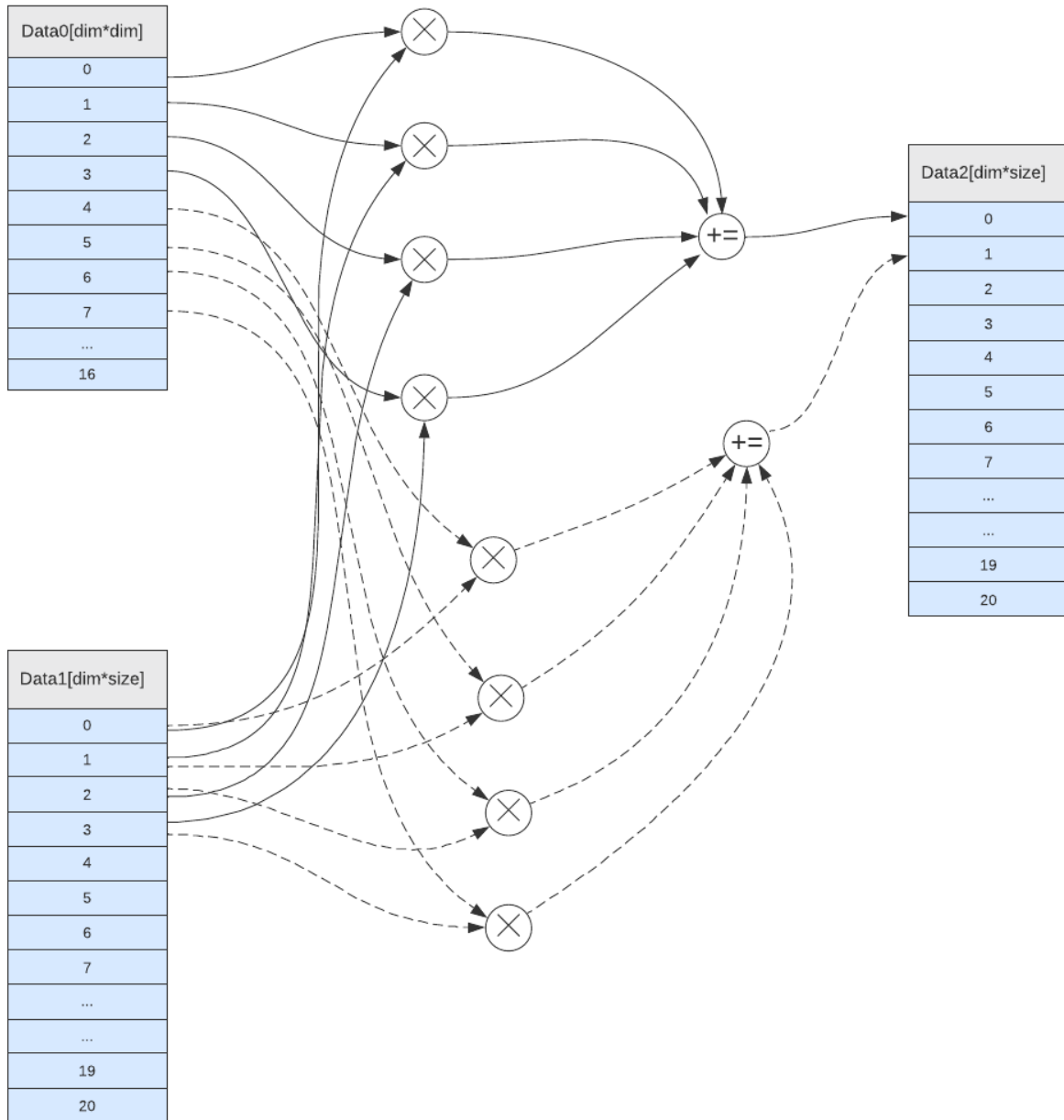
Προκειμένου να προχωρήσουμε στην υλοποίηση του επιταχυντή και την βέλτιστη απεικόνιση της συνάρτησης σε hardware, πρώτο βήμα είναι η εις βάθος κατανόηση του κώδικα και των εξαρτήσεων που υπάρχουν μεταξύ των στοιχείων και είναι ικανές να δημιουργήσουν bottle-necks στην παραλληλοποίηση του κώδικα. Μετά από μελέτη του κώδικα παρατηρήσαμε ότι το μεγαλύτερο πρόβλημα συναντάται σε αυτό το σημείο κώδικα

```
1 data2 [ i*dim + k ] += data0 [ k * dim + 1 ] * data1 [ i*dim+ 1 ];
```

Από το παρακάτω διάγραμμα ροής σε ένα στιγμιότυπο της εκτέλεσης μπορούμε να αντιληφθούμε τις επιμέρους εξαρτήσεις κατά την διάρκεια εισαγωγής τιμών στον πίνακα data2, ενώ οι πίνακες data0, data1 να πραγματοποιούν συνεχώς ανάγνωση των στοιχείων τους

Algorithm Data flow example

Dim=4, Size = 5



Σχήμα 2: Function `myFunc()` Dataflow

Αντίστοιχα προβλήματα στην παραλληλοποίηση δημιουργούν οι έλεγχοι της μεταβλητής r η οποία εισάγεται κατά τον έλεγχο του threshold. Συγκεκριμένα, τα σημεία

```
1 for ( l = 0 ; r && ( l < dim ) ; l ++ )  
2 {  
3 ...  
4 }  
5  
6 if ( r )  
7 {  
8 ...  
9 }
```

αποτρέπουν την βέλτιστη δυνατή παραλληλοποίηση, αφού με την τρέχουσα μορφή υλοποίησης υποχρεώνουν το κομμάτι των ελέγχων να πραγματοποιείται σειριακά δημιουργώντας ακόμα ένα bottleneck.

5 Μηχανισμός Ελέγχου ορθότητας τροποποιήσεων

Προκειμένου να επαληθεύσουμε την ορθότητα των σχεδιαστικών μας τροποποιήσεων, είναι σημαντικό να υπάρχει ένας μηχανισμός ελέγχου σε σχέση με την αρχική μας συνάρτηση `myFunc()`. Συγκεκριμένα, ο μηχανισμός αυτός ελέγχου παρείχε στις συναρτήσεις ίδια δεδομένα εισόδου και σύγκρινε τα δεδομένα εξόδου ανά στοιχείο της συνάρτησης `myFunc` και του επιταχυντή μας. Σε περίπτωση που τα στοιχεία και στις δύο περιπτώσεις είναι ίδια εμφανίζεται μήνυμα επιτυχούς εκτέλεσης ενώ σε αντίθετη περίπτωση μήνυμα σφάλματος.

6 Περιγραφή της αρχιτεκτονικής του επιταχυντή

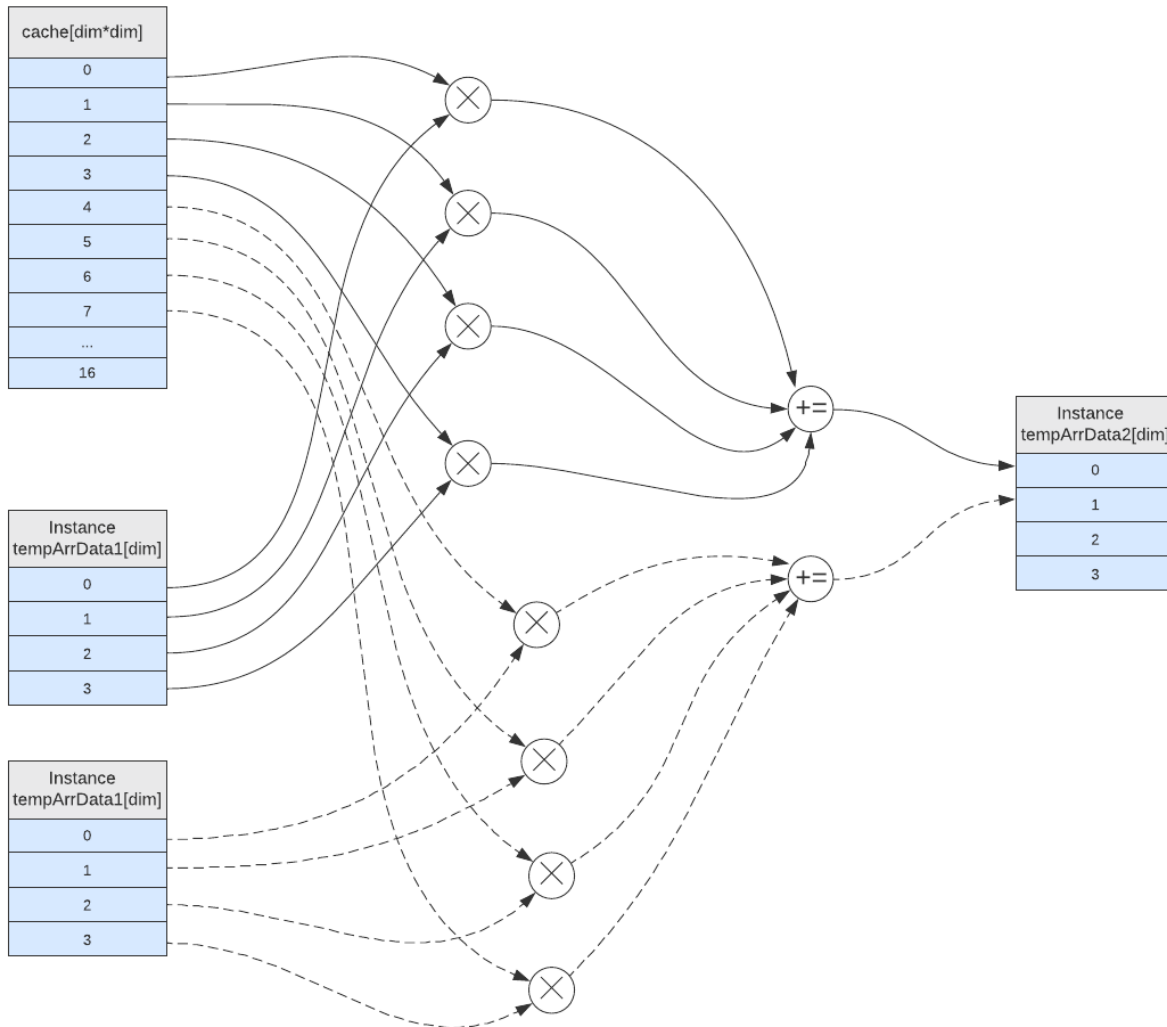
Όπως μπορεί να γίνει εύκολα κατανοητό από τα προηγούμενα η συνάρτηση στην τρέχουσα μορφή της δημιουργεί αρκετά προβλήματα στην παραλληλοποίηση και απαιτείται να αλλάξουμε προσέγγιση αναφορικά με την αρχιτεκτονική του επιταχυντή, επιτυγχάνοντας τις λιγότερες δυνατές εξαρτήσεις μεταξύ των πράξεων δίχως να επηρεάζεται η ορθότητα υλοποίησης του αλγόριθμου (αξιοποιούμε τον μηχανισμό ελέγχου ορθότητας).

Προς αυτή την κατεύθυνση χρησιμοποιήσαμε κάποια στατικά arrays, τα οποία λειτουργούν σαν cache αποφεύγοντας παράλληλα τις πολλαπλές αναγνώσεις των πινάκων. Πιο συγκεκριμένα, χρησιμοποιήσαμε μια cache για κάθε μια εκ των `data0`, `data1`, `data2` (με ονόματα `cache`, `tempArrData1` και `tempArrData2` αντίστοιχα). Λεπτομέρειες αναφορικά με την υλοποίηση ακολουθούν σε επόμενη ενότητα.

Στην παρακάτω εικόνα μπορούμε να διακρίνουμε τον τρόπο που οι συγκεκριμένες caches συντέλεσαν στην βελτίωση της παραλληλοποίησης. Το συγκεκριμένο στιγμιότυπο είναι το ίδιο με το στιγμιότυπο της προηγούμενης ενότητας, που παρουσιάστηκε παράδειγμα ροής της συνάρτησης `myFunc`.

Optimized Algorithm Data flow example

Dim=4, Size = 5



Σχήμα 3: Optimized myFunc() Dataflow

Παρατηρούμε ότι για την περίπτωση των data1 και data2 δημιουργούνται instances από arrays διαστάσεων dim, ενώ τα δεδομένα που δεν χρειάζονται πλέον πραγματοποιούν overwrite από δεδομένα που χρησιμοποιούνται σε επόμενο iteration. Θα μπορούσαμε να πούμε ότι έχουμε μια δενδρική δομή, με τα φύλλα να είναι ανεξάρτητα μεταξύ τους (ανεξαρτησία ανά dim) μεγιστοποιώντας τον παραλληλισμό.

7 Δικαιολόγηση σχεδιαστικών αποφάσεων και τροποποιήσεων του κώδικα

Με γνώμονα τις παραπάνω ενότητες και έχοντας στο μυαλό μας την αρχιτεκτονική που αναλύθηκε προχωρήσαμε σε τροποποιήσεις σε σχέση με τον αρχικό κώδικα.

Σαν αρχικό βήμα, δημιουργήσαμε τις απαραίτητες caches όπως φαίνεται παρακάτω

```
1  dataType_t  cache[dim*dim];
2  dataType_t  tempArrData1[dim];
3  dataType_t  tempArrData2[dim];
```

Η cache[dim*dim] δέχεται όλα τα δεδομένα του πίνακα data0, η tempArrData1[dim] δέχεται δεδομένα από τον πίνακα data1 ανά dim ενώ η tempArrData2[dim] ενημερώνεται προσωρινά με τιμές που δέχεται τελικά ο πίνακας data2. Τα δεδομένα του data0 διαβάζονται μία φορά στην αρχή εκτέλεσης του κώδικα

```
1  copyLoop: for ( i = 0 ; i < dim ; i++){
2      cache[i*dim] = data0[i*dim];
3      cache[i*dim+1] = data0[i*dim+1];
4      cache[i*dim+2] = data0[i*dim+2];
5      cache[i*dim+3] = data0[i*dim+3];
6  }
```

Το loop υπεύθυνο για την ενημέρωση της data2, διαφοροποιήθηκε με βάση την καινούργια αρχιτεκτονική (βλ. ενότητα αρχιτεκτονικής επιταχυντή) με τις παρακάτω αλλαγές

```
1  for ( k = 0 ; k < dim ; k ++ )
2  {
3      if(k == 0){
4          for(l = 0 ; l < dim ; l ++){
5              tempArrData1[l] = data1[ i * dim + l];
6          }
7      }
8
9      for(l = 0 ; l < dim ; l ++){
10         tempVal =(cache[k*dim+l]*tempArrData1[l]);
11         tempArrData2[k] +=tempVal;
12     }
13
14     if(tempArrData2[k] <= threshold){
15         r = 1;
16     }
17
18 }
```

Σε κάποια σημεία επίσης τοποθετήθηκαν έλεγχοι για να αποφύγουμε πολλαπλές αναγνώσεις/γράψιμο ίδιων δεδομένων σπαταλώντας πόρους.

Ακόμα, διαφοροποιήθηκαν σημεία ελέγχου που δημιουργούσαν bottlenecks στην συνάρτηση myFunc, αναφορικά με την μεταβλητή threshold. Οι έλεγχοι αυτοί δεν μπορούν να εξαλειφθούν χωρίς να παρουσιαστεί σφάλμα στην εγκυρότητα των αποτελεσμάτων, αλλά είναι δυνατό να διαφοροποιηθούν σε συνδυασμό με μια μεταβλητή flag, χωρίς να επηρεαστεί η ροή των παράλληλων tasks

```
1   zeroAsn:  for ( l = 0 ; l < dim ; l ++ )
2       {
3   //Clever way to get rid of existing if statement. With a flag and a product
4       tempArrData2[l] *= r;
5       if(l == dim - 1){
6           for ( k = 0 ; k < dim ; k ++ )
7               {
8                   data2 [ i*dim + k ] = tempArrData2[k];
9               }
10      }
11  }
```

8 Αναφορά των HLS ντιρεκτίβων που χρησιμοποιήθηκαν, περιγραφή της λειτουργίας τους, και δικαιολόγηση της χρήσης τους

Το σύνολο των directives που χρησιμοποιήθηκαν κατά την υλοποίησή μας αναλύονται παρακάτω

#pragma HLS INTERFACE

Κατά την σύνθεση σε C, όλες οι διεργασίες που αφορούν input και output λειτουργίες υλοποιούνται μέσω ορισμάτων των συναρτήσεων. Κατά την RTL σχεδίαση τα ίδια input και output πρέπει να υλοποιηθούν από συγκεκριμένα ports με την λειτουργία τους να βασίζεται σε κάποιο συγκεκριμένο πρωτόκολλο.

Η χρησιμότητα του συγκεκριμένου directive είναι προφανής για τα ορίσματα data0, data1(inputs) και data2(output). Όπως μπορούμε να διακρίνουμε και από την δήλωση του directive στον κώδικα

```
1  #pragma HLS INTERFACE ap_bus depth=16 port=data0 //For simulation only
2  #pragma HLS INTERFACE ap_bus depth=4000 port=data1
3  #pragma HLS INTERFACE ap_bus depth=4000 port=data2
```

χρησιμοποιήθηκαν και τα attributes ap_bus, depth και port. Το port πραγματοποιεί ανάθεση της μεταβλητής που εφαρμόζεται το directive. Το attribute ap_bus χρησιμοποιείται στις περιπτώσεις με pointers και pass by reference port ως bus interface που είναι αυτό που χρειαζόμαστε βάσει ορισμού της συνάρτησης από την εκφώνηση. Τέλος, το depth συγκεκριμενοποιεί τον μέγιστο αριθμό δειγμάτων για το test bench μας (Σύμφωνα με το manual η παράμετρος δείχνει το μέγιστο μέγεθος FIFO που απαιτείται για δημιουργία κατά το RTL co-simulation). Στα παραδείγματα μας, τρέχαμε το test bench για την περίπτωση που το $dim = 4$ και $size = 1000$. Μπορεί

να γίνει εύκολα αντιληπτό ότι οι τιμές που θέτουμε στην κάθε περίπτωση εξαρτάται από το βάθος του εκάστοτε array (Υπενθυμίζουμε ότι `data0[dim*dim]`, `data1[dim*size]`, `data2[dim*size]`).

Παρατήρηση: Εκτός από την περίπτωση του `ap_bus` πειραματιστήκαμε και με διαφορετικά αντίστοιχα attributes τα οποία δεν βελτίωναν στην πράξη το τελικό αποτέλεσμα με κάποιο τρόπο. Δεν προχωρήσαμε όμως σε περισσότερη ανάλυση και εμβάθυνση καθώς το συγκεκριμένο directive αξιοποιείται μόνο για την περίπτωση του simulation ενώ στο περιβάλλον του SDSoc δεν έχει εφαρμογή

#pragma HLS PIPELINE

Το συγκεκριμένο directive ήταν από τα προαπαιτούμενα του project, αλλά εκτός αυτού αποτελεί ίσως το πιο χρησιμοποιημένο HLS directive αφού συντελεί στην μείωση του initiation interval μιας συνάρτησης ή ενός loop επιτρέποντας την ταυτόχρονη εκτέλεση διαδικασιών.

Μια pipelined συνάρτηση ή loop δύναται να επεξεργαστεί δεδομένα εισόδου κάθε N κύκλους ρολογιού, όπου N είναι το initiation interval της εκάστοτε συνάρτησης ή loop. Από το συγκεκριμένο προκύπτει και όρισμα που χρησιμοποιούμε με το $II = < int_value >$ να είναι τα αρχικά του initiation interval με default τιμή το 1.

Στην περίπτωση μας, το βέλτιστο initiation interval έχει την τιμή του `dim` (στην περίπτωση του *αεζελεατορ* με $dim = 4$ περιμένουμε $II = 4$, ενώ για *accelerator* $dim = 16$ $II = 16$). Αυτό προέκυψε από το σημείο του κώδικα που πραγματοποιούμε το pipeline, δηλαδή εσωτερικά του `size loop`, αφού όλοι οι εσωτερικοί υπολογισμοί πραγματοποιούνται σε `loop dim` επαναλήψεων.

Αξίζει να αναφέρουμε ότι η επιλογή του συγκεκριμένου σημείου σημείου βασίστηκε τόσο στην γνώση της μεταβλητής `dim` ανά περίπτωση όσο και στην τυχαιότητα της μεταβλητής `size` (θα μπορούσε να είχε ανατεθεί μια πολύ μεγάλη τιμή και να δημιουργούσε πρόβλημα στα resources αλλά και εξαιρετικά μεγάλο II). Ακόμα, η χρήση του pipeline σε πιο εσωτερικά loop δεν οδηγούσε σε τόσο μεγάλο παραλληλισμό.

#pragma HLS DATAFLOW

Το directive `dataflow` επιτρέπει task level pipelining, επιτρέποντας συναρτήσεις και loop να πραγματοποιούν overlap στην λειτουργία τους, αυξάνοντας την παραλληλοποίηση της RTL υλοποίησης και το συνολικό throughput της σχεδίασης.

Στην υλοποίηση μας, παρατηρούμε βελτίωση του interval με την χρήση του συγκεκριμένου directive που εκμεταλλεύεται το γέμισμα της προσωρινής cache του `data0` και της υλοποίησης σε pipeline παραλληλοποιώντας τα ανεξάρτητα μεταξύ τους tasks.

#pragma HLS UNROLL

Ακόμα ένα συνηθισμένο directive, που πραγματοποιεί unroll τα loops δημιουργώντας πολλαπλές ανεξάρτητες διεργασίες αντί για μία. Κατά το RTL design τα πολλά αντίγραφα του σώματος του loop που έχουν δημιουργηθεί επιτρέπουν κάποια iteration να πραγματοποιούνται παράλληλα.

Το συγκεκριμένο directive το χρησιμοποιούμε στην αρχή της εκτέλεσης πριν την εφαρμογή του pipeline όταν τοποθετούμε τα δεδομένα του πίνακα `data0` σε μια προσωρινή cache.

```
1 copyLoop: for ( i = 0 ; i < dim ; i++){
```

```

2 #pragma HLS unroll skip_exit_check factor=4
3     cache[i*dim] = data0[i*dim];
4     cache[i*dim+1] = data0[i*dim+1];
5     cache[i*dim+2] = data0[i*dim+2];
6     cache[i*dim+3] = data0[i*dim+3];
7
8 }
```

Τα *attributes* *factor* και *skip_exit_check* τα χρησιμοποιούμε εκμεταλλευόμενοι και πάλι την γνώση του *dim* στην περίπτωση του κάθε accelerator. Το attribute *factor = dim* καθορίζει ακριβώς το unroll που θα πραγματοποιηθεί, ενώ ο έλεγχος εξόδου γίνεται πάντα σε συγκεκριμένο αριθμό επαναλήψεων(ανά *dim*) και το attribute *skip_exit_check* μας γλιτώνει τα resources ελέγχου στην έξοδο.

Παρατήρηση: Το Vivado HLS, πραγματοποιεί από μόνο του βελτιστοποιήσεις ειδικά στις πιο καινούργιες εκδόσεις, και στην περίπτωση εφαρμογής του directive pipeline, πραγματοποιεί αυτόματα loop unrolling. Αυτό το παρατηρήσαμε και από παραδείγματα εκτέλεσης του κώδικα, αφού δεν παρατηρήσαμε καμία αλλαγή στην απόδοση όταν εφαρμόσαμε unrolling εσωτερικά του pipelined loop.

9 Σχεδιαστικά στάδια

Στην περίπτωση του High-Level Synthesis που πραγματοποιήσαμε δεν υπάρχει αυστηρά ορισμένη μεθοδολογία για την βελτιστοποίηση αλγορίθμων, αλλά πολλά διαφορετικά directives και προσεγγίσεις ικανές να βελτιώσουν το interval και το latency του εκάστοτε αλγορίθμου.

Προκειμένου να φτάσουμε στο τελικό αποτέλεσμα και στις σχεδιαστικές επιλογές που αναλύθηκαν παραπάνω, περάσαμε από αρκετά στάδια πραγματοποιώντας πολλές δοκιμές στο εργαλείο Vivado HLS ειδικά στην περίπτωση του accelerator για $dim = 4$ που υλοποιήθηκε πρώτα.

Αναφέρονται συνοπτικά οι αλλαγές που πραγματοποιήθηκαν σε κάθε στάδιο(accelerator ,dim= 4)

- Init: Ο αρχικός κώδικας χωρίς αλλαγές
- Optm_1: Πρώτα στάδια αλλαγών στον κώδικα. Προσθήκη pipeline pragma στο size loop. Initiation Interval = 16
- Optm_2: Προσθήκη πολλών caches για την αποφυγή bottlenecks στο data0. Initiation Interval =4 (Στόχος)
- Optm_3: Μετατροπή πολλαπλών caches σε μια στην περίπτωση του data0(hardcoded). Προσθήκη caches για data1, data2
- Optm_4: Μετατροπή πολλαπλών caches σε μια στην περίπτωση του data0, με for loop(γενίκευση του Optm_3)
- Optm_5: Προσθήκη dataflow pragma. Δοκιμές με arbitrary δεδομένα χωρίς βελτίωση(σχόλια κώδικα)

Παρατηρούμε τα παρακάτω metrics για το κάθε στάδιο

	Init	Optim_1	Optim_2	Optim_3	Optim_4	Optim_5
Target	10.00	10.00	10.00	10.00	10.00	10.00
Estimated	8.750	8.750	8.750	8.750	8.750	8.750

Table 1: Timing(ns)

		Init	Optim_1	Optim_2	Optim_3	Optim_4	Optim_5
Latency	min	205001	16030	4066	4050	4054	4051
	max	205001	16030	4066	4050	4054	4051
Interval	min	205001	16030	4066	4050	4054	4034
	max	205001	16030	4066	4050	4054	4034

Table 2: Latency (clock cycles)

	Init	Optim_1	Optim_2	Optim_3	Optim_4	Optim_5
BRAM_18K	0	0	0	0	0	0
DSP48E	5	5	21	21	23	23
FF	805	1348	4172	3880	4502	5087
LUT	1386	1523	5098	4196	4800	5966

Table 3: Utilization Estimates

Όπως μπορούμε να παρατηρήσουμε, στα περισσότερα στάδια είδαμε βελτιώσεις στην υλοποίηση, ενώ στο τελικό Optim_5 το latency και το interval είναι αισθητά καλύτερο από την αρχική υλοποίηση. Παράλληλα, είναι εμφανές από τον τελευταίο πίνακα ότι τα resources που χρησιμοποιούνται αυξάνονται από στάδιο σε στάδιο. Το συγκεκριμένο γεγονός δεν μας δημιουργεί πρόβλημα καθώς δεν έχουμε περιορισμούς στα resources που μπορούμε να αξιοποιήσουμε.

10 Αναφορά των SDS ντιρεκτίβων που χρησιμοποιήθηκαν, του sds_lib API που χρησιμοποιήθηκαν, περιγραφή της λειτουργίας τους και δικαιολόγηση χρήσης τους

Αρχικά για να περάσουμε στο SDSoC περιβάλλον κάναμε κάποιες βασικές αλλαγές που θα πρόσθεταν καλύτερο performance στην υλοποίησή μας. Πιο συγκεκριμένα, ασχοληθήκαμε με το memory allocation, όπου στο προηγούμενο παραδοτέο γινόταν όπως γνωρίζουμε από την C με malloc() και free(). Στην περίπτωση του SDSoC όμως αλλάξαμε τον τρόπο και αντί για malloc() και free() χρησιμοποιήσαμε sds_alloc() και sds_free() για το allocation των array που θα δοθούν σαν ορίσματα στην hardware function. Ο λόγος που επιλέξαμε αυτές τις συναρτήσεις

της `sds.lib.h` βιβλιοθήκης είναι καθώς έτσι εξασφαλίζουμε ότι τα δεδομένα μας θα τοποθετηθούν σε φυσική και συνεχή μνήμη με αποτέλεσμα να έχουμε πιο γρήγορο `read` και `write` και συνεπώς καλύτερο performance.

Συνεχίζοντας με την `myAccel` function για να εξασφαλίσουμε ότι οι προσβάσεις μνήμης θα γίνουν με συνεχή τρόπο, όπως προαναφέραμε χρησιμοποιούμε το εξής directive:

```
1 #pragma SDS data mem_attribute(data0:PHYSICAL_CONTIGUOUS, data1:
    PHYSICAL_CONTIGUOUS, data2:PHYSICAL_CONTIGUOUS)
```

Με αυτόν τον τρόπο, μέσω του `PHYSICAL_CONTIGUOUS`, ο χρήστης ενημερώνει τον SD-SoC compiler ότι τα arrays `data0`, `data1`, `data2`, έχουν κατανομηθεί σε block μνήμης που είναι φυσικά συνεχή, έτσι ώστε να γνωρίζει πως θα επιτύχει τις προσβάσεις.

Άλλο ένα `#pragma` που χρησιμοποιήθηκε ήταν το `#pragma SDS data access_pattern()` το οποίο βοήθησε στο να δείξουμε στον compiler ότι τα ορίσματα που δίνονται στην συνάρτησή μας θα λειτουργήσουν σαν streaming port μέσω της οποίας θα επικοινωνούν η `main` με την hardware function μας. Ο τρόπος που χρησιμοποιήθηκε είναι ο παρακάτω και έγινε με αυτόν το τρόπο, διότι πρώτον αναφερόμαστε σε pointers και δεύτερον αν η εφαρμογή του `#pragma SDS data access_pattern()` γινόταν μετά από το `#pragma SDS data copy` τότε ο compiler θα το αγνοούσε. Επίσης χρησιμοποιήθηκε με `SEQUENTIAL` αντί για `RANDOM` γιατί θέλουμε να διαβάσει μία φορά με τη σειρά την κάθε θέση μνήμης.

```
1 #pragma SDS data access_pattern(data0:SEQUENTIAL, data1:SEQUENTIAL, data2:
    SEQUENTIAL)
```

Σημαντικότερη ίσως σχεδιαστική επιλογή για να πετύχουμε μια βέλτιστη υλοποίηση είναι η επιλογή των κατάλληλων Data Movers. Ένας Data Mover, παίζει καθοριστικό ρόλο για το interconnect, μεταφέροντας δεδομένα μεταξύ του PS(processing system) και των accelerators καθώς και μεταξύ των accelerators. Οι Data Movers είναι βασισμένοι στις ιδιότητες και το μέγεθος των δεδομένων που μεταφέρονται. Στην περίπτωση μας, επιλέξαμε τον `AXIDMA_SIMPLE` data mover, ο οποίος είναι η πιο αποδοτική bulk transfer μηχανή αλλά υποστηρίζει έως `8MB`. Ο συγκεκριμένος data mover, όπως φαίνεται και από το όνομα του, χρησιμοποιεί τον DMA Controller για την μεταφορά δεδομένων. Προαπαιτήση του `AXIDMA_SIMPLE` είναι τα δεδομένα να είναι contiguous, το οποίο έχουμε ήδη διασφαλίσει με την χρήση του `SDS data mem_attribute` όπως αναφέραμε και παραπάνω.

```
1 #pragma SDS data data_mover(data0:AXIDMA_SIMPLE, data1:AXIDMA_SIMPLE,
    data2:AXIDMA_SIMPLE)
```

Τέλος, χρησιμοποιούμε το directive `#pragma SDS data copy()` που αντιγράφει τα δεδομένα μεταξύ του host processor memory και του hardware function χρησιμοποιώντας κάθε τον επιλεγμένο data mover για την μεταφορά δεδομένων.

```

1 #pragma SDS data copy(data0[0:dim*dim])
2 #pragma SDS data copy(data1[0:dim*size])
3 #pragma SDS data copy(data2[0:dim*size])

```

11 Αξιολόγηση απόδοσης

Όπως και την ανάλυση στο High-Level Synthesis έτσι και στο τελικό στάδιο του SDSoC η βέλτιστη υλοποίηση προέκυψε μετά από αρκετές δοκιμές εκτέλεσης του αλγόριθμου πάνω στο ZedBoard. Καθοριστικός παράγοντας αναφορικά με την απόδοση, όπως προαναφέρθηκε έπαιξε το interconnect μεταξύ του PS(processing system) και του accelerator.

Συνοπτικά αναφέρουμε τις τροποποιήσεις κάθε υλοποίησης στα directives(άρα το interconnect) που θα χρησιμοποιήσουμε στην συνέχεια:

- Υλοποίηση 1: Υλοποίηση με directives mem_attribute, access_pattern, copy
- Υλοποίηση 2: Ίδια με υλοποίηση 1 με προσθήκη του directive data_mover
- Υλοποίηση 3: Ίδια με υλοποίηση 2 με προσθήκη του directive sys_port(ACP)
- Υλοποίηση 4: Ίδια με υλοποίηση 2 με προσθήκη του directive sys_port(AFI)

Αναλυτικά για όλες τις διαφορετικές υλοποιήσεις φαίνεται το Hardware execution time(σε sec) για διαφορετικές τιμές του size(η παράμετρος SDSoCσιζε ρυθμίζει τοexecution time με το dim να συμπεριφέρεται σαν σταθερά).

	Υλοποίηση 1	Υλοποίηση 2	Υλοποίηση 3	Υλοποίηση 4
<i>Size</i> = 10000	0,000554	0,000470	0,000470	0,001580
<i>Size</i> = 100000	0,004555	0,004472	0,004444	0,013298
<i>Size</i> = 1000000	0,043963	0,043857	0,043872	0,132049

Πίνακας 4: Χρόνοι εκτέλεσης για διαφορετικό size(s)

Παρατηρούμε λοιπόν, ότι η υλοποίηση μας προσεγγίζει σε αρκετά μεγάλο βαθμό το θεωρητικό άνω όριο γεγονός που δείχνει ότι η υλοποίηση μας είναι βέλτιστη.

	<i>Size</i> = 10000	<i>Size</i> = 100000	<i>Size</i> = 1000000
Υλοποίηση 2	21.276.595	22.361.359	22.801.377

Πίνακας 5: Λόγος $\frac{size}{Exec.time}$ για υλοποίηση που χρησιμοποιήθηκε

Από τα παραπάνω αποτελέσματα παρατηρούμε ότι οι υλοποιήσεις 2,3 είναι πολύ κοντά σε απόδοση, τελικά όμως επιλέξαμε να προχωρήσουμε με την υλοποίηση 2. Στην ενότητα Παράρτημα υπάρχει screenshot εκτέλεσης της υλοποίησης 2 μετά από εκτέλεση κώδικα στο ZedBoard Για ρολόι $clk = 100MHz$ και $II(InitiationInterval) = 4$ του accelerator προκύπτει με απλή μέθοδο των τριών ότι το άνω θεωρητικό όριο που μπορεί η συνάρτηση να εκτελεστεί είναι

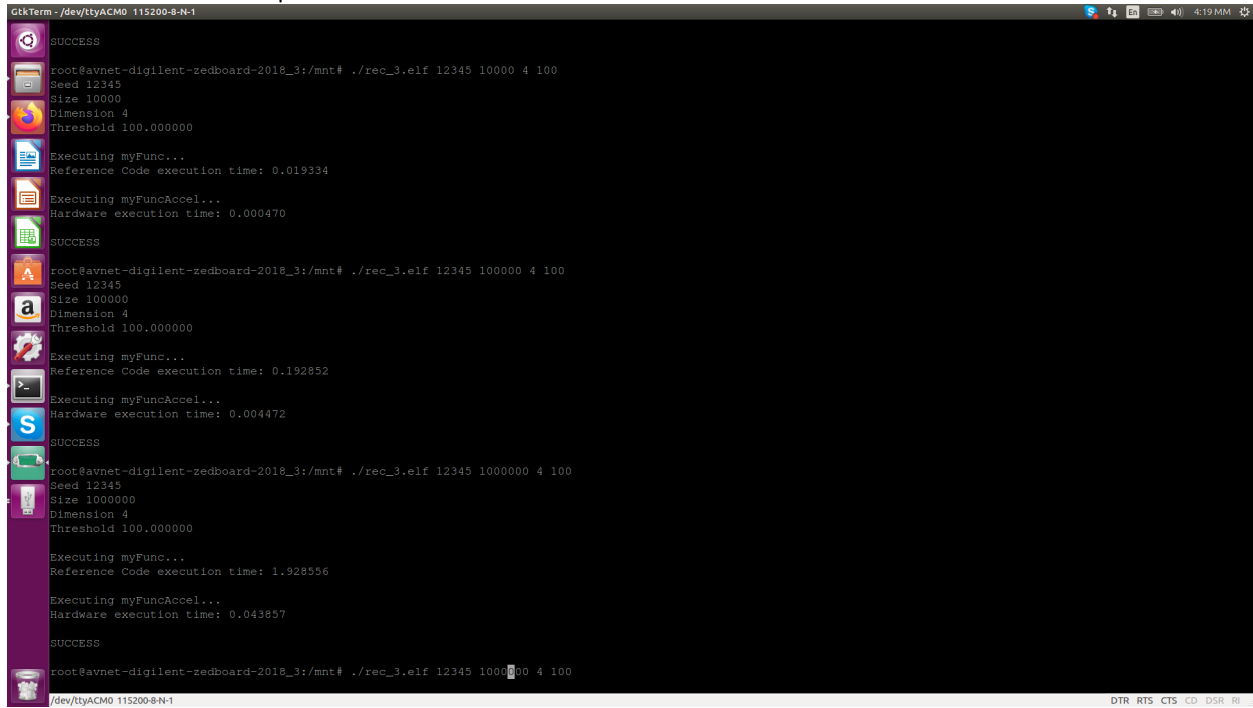
$\frac{100 \times 10^6}{4} = 25 \times 10^6$ φορές. Η απόδοση της συνάρτησης προκύπτει από την διαίρεση του size με το execution time και φαίνεται στον παρακάτω πίνακα

12 Συμπεράσματα

Στα πλαίσια της συγκεκριμένης εργασίας ασχοληθήκαμε με την δημιουργία ενός accelerator με τεχνικές High-Level Synthesis και με το εργαλείο SDSoC. Σαν πρώτο βήμα τροποποιήσαμε τον αλγόριθμό μας προκειμένου να επιδέχεται τον βέλτιστο δυνατό παραλληλισμό. Έπειτα πειραματιστήκαμε με τις διάφορες υλοποιήσεις σε επίπεδο High-Level Synthesis με στόχο να πετύχουμε το βέλτιστο Interval και Latency για τον αλγόριθμό μας και τέλος τροποποιήσαμε κατάλληλα τον κώδικα μας προκειμένου να μπορεί να τρέξει στο ZedBoard. Αντιληφθήκαμε και στην πράξη, ότι η επιτάχυνση αλγορίθμων είναι διαδικασία τοπικά βέλτιστη, αφού δεν υπάρχει αυστηρά καθορισμένη μεθοδολογία που να βελτιώνει τον εκάστοτε αλγόριθμο. Είδαμε τέλος την σημασία βελτιστοποίησης του interconnect μεταξύ processing system και accelerator προκειμένου να προκύψει το καλύτερο αποτέλεσμα.

13 Παράρτημα

Screenshot εκτέλεσης accelerator στο ZedBoard



```
GtkTerm - /dev/ttyACM0 115200-8-N-1
SUCCESS
root@avnet-diligent-zedboard-2018_3:/mnt# ./rec_3.elf 12345 10000 4 100
Seed 12345
Size 10000
Dimension 4
Threshold 100.000000
Executing myFunc...
Reference Code execution time: 0.019334
Executing myFuncAccel...
Hardware execution time: 0.000470
SUCCESS
root@avnet-diligent-zedboard-2018_3:/mnt# ./rec_3.elf 12345 100000 4 100
Seed 12345
Size 100000
Dimension 4
Threshold 100.000000
Executing myFunc...
Reference Code execution time: 0.192852
Executing myFuncAccel...
Hardware execution time: 0.004472
SUCCESS
root@avnet-diligent-zedboard-2018_3:/mnt# ./rec_3.elf 12345 1000000 4 100
Seed 12345
Size 1000000
Dimension 4
Threshold 100.000000
Executing myFunc...
Reference Code execution time: 1.928556
Executing myFuncAccel...
Hardware execution time: 0.043857
SUCCESS
root@avnet-diligent-zedboard-2018_3:/mnt# ./rec_3.elf 12345 1000000 4 100
DTR RTS CTS CD DSR RI
```

Σχήμα 4: Optimized Execution Screenshot in Zedboard

14 Βιβλιογραφία

Lecture slides

https://www.xilinx.com/html_docs/xilinx2019_1/sdsoc_doc/ysq1526001978941.html#ysq1526001978941

https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

http://www.ioe.nchu.edu.tw/Pic/CourseItem/4468_20_Zynq_Architecture.pdf

https://courses.e-ce.uth.gr/CE435/labs/lab2_doc.pdf

https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623.html

<https://isca.teicrete.gr/wp-content/uploads/2016/06/TrouliGeorgiaEirini2017.pdf>