ΠΟΛΥΤΕΧΝΕΙΟ
ΚΡΗΤΗΣ

Αναδιατασσόμενα Ψηφιακά Συστήματα

Αναφορά εργασίας

Σωτήριος Μιχαήλ

Χανιά, 2019

Αλγόριθμος

Ο αλγόριθμος για τον οποίο σχεδιάστηκε ο επιταχυντής είναι ένας accumulator, ο οποίος περιγράφεται από τη παρακάτω συνάρτηση:

$$data2[l][k] = \sum_{i=0}^{dim} data0[k][i] \cdot data1[l][i]$$

Το dim μπορεί να είναι 4 ή 16 (περιλαμβάνονται παραλλαγές του IP για κάθε διαφορετική τιμή του dim). Ο πίνακας data0 είναι διαστάσεων dim x dim και των πινάκων data1 και data2 είναι dim x size, όπου το size είναι σταθερό.

Ο αλγόριθμος επίσης ελέγχει κάθε στοιχείο του πίνακα data2 έτσι ώστε να μη περνά ένα αριθμιτικό όριο. Εάν όλα τα k στοιχεία μίας γραμμής l του πίνακα αυτού ξεπερνούν αυτό το όριο, τότε όλα τα στοιχεία της γραμμής αυτής επιστρέφουν στη τιμή 0.0.

Για μεγάλες διαστάσεις πινάκων, οι οποίοι θα ήταν χρήσιμοι σε πραγματικές εφαρμογές, μία single-thread υλοποίηση αυτού του αλγορίθμου μπορεί να χρειαστεί αρκετά δευτερόλεπτα έως λεπτά εκτέλεσης. Για αυτό το λόγο θα σχεδιάσουμε έναν επιταχυντή υλικού για τον αλγόριθμο αυτό πάνω για μία πλατφόρμα FPGA, η οποία θα παρουσιαστεί μετέπειτα σε αυτή την αναφορά. Κατά τη σχεδίαση, χρησιμοποιήθηκαν τα εργαλεία Vivado, Vivado HLS και SDx της Xilinx, σε περιβάλλον Linux.

Αρχιτεκτονική επιταχυντή

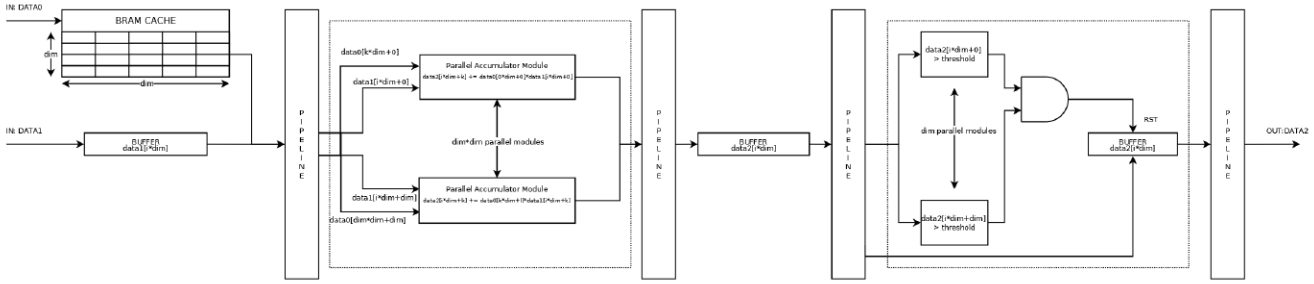
Για την καλύτερη δυνατή απόδοση του επιταχυντή, θα προσπαθήσουμε να εκμεταλλευτούμε όσο το δυνατόν περισσότερο τις δυνατότητες παραλληλισμού που μας προσφέρει μία υλοποίηση σε αναδιατασσόμενη λογική. Με τροποποίηση του αλγορίθμου, αυτός ο παραλληλισμός μπορεί να επιτευχθεί σε μεγάλο βαθμό. Ένα άλλο σημείο επιβράδυνσης (bottleneck) κατά την εκτέλεση του αλγορίθμου αυτού όσο αναφορά την υλοποίηση του σε υλικό, είναι η ανάγνωση των δεδομένων εισόδου.

Μελετώντας την λειτουργία του αλγορίθμου, παρατηρούμε πως για κάθε γραμμή, δηλαδή τα 4 ή 16 στοιχεία, του πίνακα data2, χρησιμοποιούμε όλο τον πίνακα data0. Ο πίνακας αυτός έχει επίσης σημαντικά μικρότερο μέγεθος από τους άλλους δύο. Στη χειρότερη περίπτωση, όπου dim = 16, ο πίνακας data0 έχει μέγεθος 16 x 16 = 256 στοιχεία, και δεδομένου πως κάθε στοιχείο είναι ένας αριθμός float, έχουμε ένα εγγυημένο μέγιστο μέγεθος 1 kbyte. Έτσι, μπορούμε να δημιουργήσουμε μία cache με τα δεδομένα του data0, μέσω BRAM, έτσι ώστε να ελαχιστοποιήσουμε το χρόνο ανάγνωσης δεδομένων. Για τα δεδομένα προς ανάγνωση από το πίνακα data1 και προς εγγραφή προς τον πίνακα data2, χρησιμοποιούμε buffers , και αυτοί υλοποιημένοι σε BRAM, οι οποίοι έχουν μέγεθος μία γραμμή κάθε πίνακα, δηλαδή ένα εγγυημένο μέγιστο των 64 bytes (για dim = 16).

Ο υπολογισμός κάθε γινομένου και κάθε αθροίσματος θέλουμε να γίνεται παράλληλα, έτσι ώστε να μεγιστοποιήσουμε τον παραλληλισμό. Επομένως, έχουμε dim x dim παράλληλα modules, τα οποία εκτελούν ένα πολλαπλασιασμό, και στη συνέχεια εκτελούνται dim παράλληλα αθροίσματα, των αντίστοιχων γινομένων, και έχουμε ως αποτέλεσμα μία γραμμή του πίνακα data2, η οποία αποθηκεύεται σε buffer, έτσι ώστε να ελεγχθεί εάν τα στοιχεία της ξεπερνούν το όριο που έχει θέσει ο χρήστης.

Για τον έλεγχο του ορίου, ελέγχουμε με παράλληλους συγκριτές, εάν το κάθε στοιχείο του buffer αυτού ξεπερνά το όριο. Εάν ξεπερνά το όριο, ο κάθε συγκριτής θα έχει έξοδο 1 και με μία απλή πύλη AND, έχουμε το αποτέλεσμα εάν όλα τα στοιχεία της γραμμής ξεπερνούν το όριο. Εάν ναι, τότε θέτουμε όλα τα στοιχεία του buffer ως 0.0, έτσι ώστε να εγγραφούν στο πίνακα εξόδου data2.

Η αρχιτεκτονική αυτή επεξεργάζεται μία γραμμή του πίνακα τη φορά. Για να αυξήσουμε και άλλο την απόδοση του επιταχυντή, χρησιμοποιούμε pipelines για την αρχιτεκτονική μας, έτσι ώστε σε κάθε pipeline stage της αρχιτεκτονικής να επεξεργάζεται μία γραμμή του πίνακα. Τα επίπεδα του αλγορίθμου είναι προφανή, με αρχή την ανάγνωση δεδομένων, έπειτα τις αριθμητικές πράξεις, τον έλεγχο ορίου και τέλος την εγγραφή στο πίνακα εξόδου.



Εικόνα 1: *Ένα block diagram της pipelined αρχιτεκτονικής*

HLS κώδικας

Για τη σχεδίαση της αρχιτεκτονικής, όπως αναφέρθηκε και προηγουμένως, χρησιμοποιήθηκε το εργαλείο Xilinx Vivado και Vivado HLS. Το εργαλείο Vivado HLS, χρησιμοποιώντας κώδικα C, τον αποτυπώνει σε αναδιατασσόμενη λογική με χρήση συγκεκριμένων directives, τα οποία καθοδηγούν το εργαλείο κατά τη σχεδίαση της λογικής. Έτσι ώστε να αποτυπωθεί σε υλικό ο αλγόριθμος (υλοποιημένος σε κώδικα C) που μας δόθηκε, χρειάστηκαν διάφορες αλλαγές στον κώδικα, έτσι ώστε το εργαλείο Vivado HLS να σχεδιάσει την επιθυμητή αρχιτεκτονική. Ο αρχικός κώδικας C του αλγορίθμου:

```
void myFunc (unsigned int size, unsigned int dim, dataType_t threshold, dataType_t
* data0, dataType_t * data1, dataType_t * data2) {
      unsigned int i, k, l;
      for ( i = 0 ; i < size ; i ++ )  {
            for ( k = 0 ; k < dim ; k ++ )  {
                  data2 [ i*dim + k ] = 0.0 ;
            }
            for ( k = 0 ; k < dim ; k ++ )  {
                  for ( l = 0 ; l < dim ; l ++ )  {
                        data2 [ i*dim + k ] += data0 [ k * dim + l ] *
data1[ i*dim+ l ];
                  }
            }
            int r = 1 ;
            for ( l = 0 ; r && ( l < dim ) ; l ++ )  {
                  r = ( data2 [ i*dim + l ] > threshold ) ;
            }
            if ( r ) {
                  for ( l = 0 ; l < dim ; l ++ )  {
                        data2 [ i*dim + l ] = 0.0;
                  }
             }
      }
}
```

Αρχικά, τοποθετήθηκε η αποθήκευση όλων των δεδομένων εισόδου του πίνακα data0 σε ένα δευτερεύον πίνακα, τον οποίο το εργαλείο Vivado HLS θα αποτυπώσει ως BRAM πάνω στην FPGA, έτσι ώστε να δημιουργήσουμε την cache της αρχιτεκτονικής. Επίσης, τα δεδομένα κάθε γραμμής των πινάκων αποθηκεύονται σε buffers, τα οποία αρχικοποιούνται ανά iteration του κεντρικού loop.

Στη συνέχεια, καθώς θέλουμε να έχουμε παράλληλο έλεγχο ορίου για κάθε στοιχείο, το εργαλείο δε μπορεί να δημιουργήσει τα παράλληλα αυτά modules εάν υπάρχουν μεταβλητά όρια σε κάθε loop. Έτσι, αλλάζουμε τον κώδικα και σε αυτό το σημείο, προσθέτοντας μία λογική συνάρτηση για κάθε στοιχείο του buffer με τα δεδομένα που υπολογίστηκαν. Τέλος, τα δεδομένα του buffer αυτού εγγράφονται στον πίνακα εξόδου data2.

Για τη σχεδίαση της αρχιτεκτονικής, χρησιμοποιήθηκαν τρία HLS directives. Πρώτα, το HLS pipeline, με initiation interval = 4 για dim = 4 και initiation interval = 16 για dim = 16, έτσι ώστε να έχουμε μία pipelined αρχιτεκτονική, η οποία μας δίνει ένα αποτέλεσμα ανά 4 ή 16 κύκλους αντίστοιχα, δηλαδή υπολογίζει ένα στοιχείο πίνακα data2 ανά κύκλο, και μία ολόκληρη γραμμή σε 4 ή 16 κύκλους, ανάλογα την τιμή του dim. Έπειτα, το directive HLS unroll, το οποίο αποτυπώνει σε λογική το σώμα του loop τόσες φορές όσες θα γινόταν η επανάληψη στον κώδικα C (και για αυτό το λόγο πρέπει να μην υπάρχουν variable loop bounds), έτσι ώστε τα modules λογικής που θα προκύψουν να μπορούν να λειτουργούν παράλληλα. Εκμεταλλευόμενοι την ιδιότητα του αλγορίθμου να μην υπάρχουν εξαρτήσεις μεταξύ των δεδομένων που υπολογίζονται, ήταν δυνατή η επιλογή του full unroll, έτσι ώστε να έχουμε όσο περισσότερο παραλληλισμό γίνεται. Τέλος, χρησιμοποιήθηκε επίσης το HLS array_partition με factor = 4 για την cache με τα στοιχεία του data0, καθώς και για τα δύο buffers με τα στοιχεία του data1 και data2, με σκοπό την παράλληλη επεξεργασία κάθε κομματιού των πινάκων, εφόσον μας το επιτρέπει η λογική του αλγορίθμου. Η χρήση αυτού του directive έχει ως αποτέλεσμα την αύξηση των ports των μνημών που θα αποτυπωθούν σε λογική για κάθε ένα από τις τρεις αυτές μεταβλητές στο κώδικα, και επομένως θα αυξηθεί και το throughput της αρχιτεκτονικής. Αυτό οδηγεί επίσης και στη δημιουργία περισσότερων instances από BRAMs και καταχωρητές, αλλά καθώς σε αυτή την αρχιτεκτονική έχουμε δεδομένη χειρότερη περίπτωση για το μέγεθος αυτών των μνημών, μπορούμε να αποφανθούμε πως το επιπλέον κόστος των περισσότερων instances θα είναι αμελητέο.

Κατά την σύνθεση HLS του κώδικα C, χρησιμοποιήθηκαν directives HLS interface, για τις εισόδους data0, data1 και data2, έτσι ώστε η προσομοίωση HLS να παράξει δεδομένα για την απόδοση της αρχιτεκτονικής. Αυτά τα directives δεν έχουν κάποια πραγματική χρήση και θα αφαιρεθούν κατά τη διάρκεια του επόμενου βήματος στη σχεδίαση.

Ο κώδικας με τις αλλαγές και τα directives:

```
void myFuncAccel (unsigned int size, unsigned int dim, dataType_t threshold,
dataType_t * data0, dataType_t * data1, dataType_t * data2)  {
#pragma HLS INTERFACE ap_fifo depth=2000000 port=data2
#pragma HLS INTERFACE ap_fifo depth=2000000 port=data1
#pragma HLS INTERFACE ap_fifo depth=2000000 port=data0
unsigned int i, k, l;
dataType_t data0buf[maxdim*maxdim], data1buf[maxdim], data2buf[maxdim];
int r[maxdim];
#pragma HLS ARRAY_PARTITION variable=data2buf block factor=4 dim=1
#pragma HLS ARRAY_PARTITION variable=data1buf block factor=4 dim=1
#pragma HLS ARRAY_PARTITION variable=data0buf block factor=4 dim=1

data0_loop:for ( k = 0 ; k < maxdim ; k ++ ) {
#pragma HLS PIPELINE II=4
      data0_loop_inner:for ( l = 0 ; l < maxdim; l ++ )  {
      #pragma HLS UNROLL
            data0buf [ k * maxdim + l ] = data0 [ k * maxdim + l ];
            }
      }
      i_loop:for ( i = 0 ; i < size ; i ++ )  {
      #pragma HLS LOOP_TRIPCOUNT min=100 max=1000 avg=1000
```

```
	#pragma HLS PIPELINE II=4
		init_loop:for ( k = 0 ; k < maxdim ; k ++ )  {
		#pragma HLS UNROLL
			data2 [ i * maxdim + k ] = 0.0;
			data2buf [ k ] = 0.0;
		}
		l_loop:for ( l = 0 ; l < maxdim ; l ++ )  {
		#pragma HLS UNROLL
			data1buf [ l ] = data1 [ i * maxdim + l ];
		}
		k_loop_outer:for ( k = 0 ; k < maxdim ; k ++ )  {
			k_loop_inner:for ( l = 0 ; l < maxdim ; l ++ )  {
			#pragma HLS UNROLL
				data2buf [ k ] += data0buf [k * maxdim + l] * data1buf [l];
			}
		}
		for ( l = 0 ; l < maxdim ; l ++ )  {
		#pragma HLS UNROLL
			r[ l ] = 1;
			r[ l ] = ( data2buf [ l ] > thres_f );
		}
		int rf = ( r[0] == 1 && r[1] == 1 && r[2] == 1 && r[3] == 1 );
		if ( rf )  {
			thres_apply:for ( l = 0 ; l < maxdim ; l ++ )  {
				#pragma HLS UNROLL
				data2buf [ l ] = 0.0;
			}
		}
		output:for( k = 0 ; k < maxdim ; k ++ )  {
		#pragma HLS UNROLL
			data2 [ i * maxdim + k ] = data2buf [ k ];
		}
	}
}
```

Αποτύπωση σε FPGA

Το επόμενο βήμα στη σχεδίαση του επιταχυντή ήταν η μεταφορά της σχεδίασης που υλοποιήθηκε μέσω των αλλαγών στον κώδικα C και του εργαλείου Vivado HLS και των directives του, σε μία πραγματική FPGA. Για αυτό το σκοπό χρησιμοποιήθηκε το εργαλείο Xilinx SDx, μέσω του οποίου γίνεται το place & route της λογικής που έχει σχεδιαστεί, και παράγεται το αρχείο με το οποίο προγραμματίζεται η FPGA που χρησιμοποιείται.

Όπως και στο προηγούμενο εργαλείο, έτσι και εδώ χρειάζονται κάποια directives τα οποία καθοδηγούν το SDx στην επιθυμητή αποτύπωση της αρχιτεκτονικής.

Έτσι ώστε να αποφύγουμε καθυστερήσεις λόγω μεταφοράς δεδομένων σε bus από μία εξωτερική μνήμη, χρησιμοποιούμε το directive SDS copy, το οποίο καθοδηγεί το εργαλείο στο να δημιουργήσει αντίγραφα των data0, data1 και data2 σε μνήμη πάνω στην FPGA. Η αντιγραφή των δεδομένων αυτών γίνεται με data movers του SDx. Για τη σχεδίαση αυτή, και σύμφωνα με το εγχειρίδιο βελτιστοποίησης σχεδιασμού της Xilinx, έγινε χρήση τριών παράλληλων data movers τύπου AXIDMA_SIMPLE, καθώς η διατεθειμένη φυσική μνήμη για τα δεδομένα είναι συνεχής, και το μέγεθος των πινάκων είναι της τάξης των μερικών MB.

Για τη προσπέλαση των δεδομένων εισόδου/εξόδου, χρησιμοποιούμε σειριακή προσπέλαση, μέσω του SDS data access_pattern, και για τους τρεις πίνακες. Καθώς ο αλγόριθμος επεξεργάζεται σειριακά γραμμή ανά γραμμή τα δεδομένα, μπορούμε να αποφύγουμε το σημαντικό κόστος της τυχαίας προσπέλασης.

Η μνήμη στην οποία τα δεδομένα αποθηκεύονται δε μπορεί να διανεμηθεί με malloc, καθώς με αυτό το τρόπο, δε μπορούμε να αποφανθούμε για τη θέση των δεδομένων στη φυσική μνήμη, και επομένως, δε μπορούμε να κάνουμε χρήση των παραπάνω SDS directives. Έτσι, από τις βοηθητικές βιβλιοθήκες του sds, χρησιμοποιούμε την συνάρτηση sds_alloc, η οποία διανέμει στα δεδομένα μία φυσικά συνεχή περιοχή μνήμης. Για την απελευθέρωση των χώρων μνήμης που διανέμεται με τη συνάρτηση sds_alloc, χρησιμοποιούμε τη συνάρτηση sds_free.

Η συσκευή της σχεδίασης – το Zynq SoC

Η συσκευή για την οποία υλοποιήθηκε η παραπάνω αρχιτεκτονική είναι το ZedBoard, μία συσκευή βασισμένη στη πλατφόρμα Zynq, και συγκεκριμένα, το Zynq-7000. Το Zynq είναι ένα System-on-Chip (SoC), δηλαδή όλη η λειτουργία ενός συστήματος υλοποιείται σε ένα μόνο chip, σε αντίθεση με ένα "παραδοσιακό" σύστημα, στο οποίο υπάρχουν φυσικά διαχωρισμένες συσκευές, οι οποίες επικοινωνούν μεταξύ τους με συνδέσεις πάνω σε ένα printed circuit board και σχηματίζουν το σύστημα. Τα προτερήματα ενός SoC είναι η πιο ασφαλής και πιο γρήγορη μεταφορά των δεδομένων, χαμηλότερες ενεργειακές απαιτήσεις, το μικρότερο φυσικό μέγεθος και περισσότερη αξιοπιστία.

Το Σύστημα Επεξεργασίας

Η γενική αρχιτεκτονική του Zynq-7000 αποτελείται από δύο μέρη: το Σύστημα Επεξεργασίας (Processing System, PS) και την Αναδιατασσόμενη Λογική (Programmable Logic, PL). Τα δύο αυτά μέρη μπορούν να χρησιμοποιηθούν ανεξάρτητα ή μαζί. Το σύστημα επεξεργασίας του Zynq-7000 βασίζεται σε ένα διπύρηνο επεξεργαστή ARM Cortex-A9, χρονισμένο στα 1GHz. Το PS περιέχει και το Application Processing Unit (APU), που είναι το σύστημα το οποίο χρησιμοποιεί τους δύο πυρήνες που διατίθενται, και περιέχει cache επιπέδου L1, ένα Media Processing Engine (MPE) και μία Μονάδα Κινητής Υποδιαστολής (Floating Point Unit, FPU) για κάθε ένα από αυτούς τους πυρήνες. Επίσης περιέχει μνήμη L2 cache, περαιτέρω on-chip μνήμη τύπου SRAM, καθώς και το Snoop Control Unit (SCU), το οποίο σχηματίζει τη γέφυρα μεταξύ των δύο πυρήνων. Τέλος, το PS υλοποιεί διεπαφές επικοινωνίας μεταξύ PS και PL, καθώς και μεταξύ PS και εξωτερικών συσκευών. Τέτοιες διεπαφές είναι οι: Serial Peripheral Interface (SPI), I2C bus, SD card, USB και Gigabit Ethernet.

Η Αναδιατασσόμενη Λογική

Το δεύτερο βασικό κομμάτι της αρχιτεκτονικής Zynq-7000 είναι η Αναδιατασσόμενη Λογική (Programmable Logic, PL). Αυτή βασίζεται στην αρχιτεκτονική FPGA των Artix-7 και Kintex-7 FPGAs, και αυτή αποτελείται από slices, Configurable Logic Blocks (CLBs) και Input/Output Blocks (IOBs) για διεπαφή.

- Τα CLBs είναι μικρές ομάδες συμβατικής λογικής, η οποίες τοποθετούνται σε ένα δισδιάστατο πίνακα του PL και συνδέονται με άλλους παραπλήσιους πόρους μέσω προγραμματιζόμενων interconnect. Κάθε CLB τοποθετείται δίπλα σε ένα switch matrix, και περιέχει δύο slices λογικής.

- Ένα slice είναι μία υπο-μονάδα μέσα στο CLB, η οποία περιέχει πόρους για την υλοποίηση συνδυαστικών και ακολουθητικών λογικών κυκλωμάτων. Κάθε slice του Zynq αποτελείται από 4 Lookup Tables, 8 Flip-Flops, και άλλη λογική.

- Το Lookup Table (LUT) είναι ένας ευέλικτος πόρος ικανός να υλοποιήσει μία λογική συνάρτηση έως και έξι εισόδων, μία μικρή ROM, μία μικρή RAM ή ένα καταχωρητή ολίσθησης.

• Ένα Flip-Flop είναι ένα στοιχείο ακολουθιακού κυκλώματος που υλοποιεί ένα καταχωρητή ενός bit, με ικανότητα επαναφοράς.

• Το switch matrix βρίσκεται δίπλα από κάθε CLB και προσφέρει ευέλικτη δρομολόγηση έτσι ώστε να δημιουργούνται συνδέσεις είτε μεταξύ στοιχείων μέσα στο ίδιο CLB, είτε από ένα CLB σε κάποιον άλλο πόρο στο PL.

• Τα Input/Output Blocks (IOBs) είναι πόροι που προσφέρουν διεπαφή μεταξύ των λογικών πόρων του PL και των εισόδων των φυσικών συσκευών που χρησιμοποιούνται για την σύνδεση εξωτερικών κυκλωμάτων. Κάθε IOB μπορεί να χειριστεί ένα σήμα εισόδου ή εξόδου του ενός bit.

Πέραν αυτών των πόρων, η πλατφόρμα του Zynq-7000 προσφέρει και δύο ειδικούς πόρους: τα DSP48E1 και τις μνήμες Block RAM (BRAM).

Οι Block RAM χρησιμοποιούνται για απαιτήσεις πυκνής μνήμης, και κάθε BRAM μπορεί να αποθηκεύσει έως και 36Kb πληροφορίας, και μπορεί να διαμορφωθεί ως μία μνήμη 36Kb ή δύο ανεξάρτητες μνήμες των 18Kb. Επίσης μπορεί να να "αναπλασθεί" έτσι ώστε να περιέχει περισσότερα δεδομένα μικρότερου μεγέθους ή λιγότερα δεδομένα μεγαλύτερου μεγέθους. Μπορούμε να δημιουργήσουμε μνήμες μεγαλύτερου μεγέθους συνδυάζοντας δύο ή περισσότερες Block RAMs.

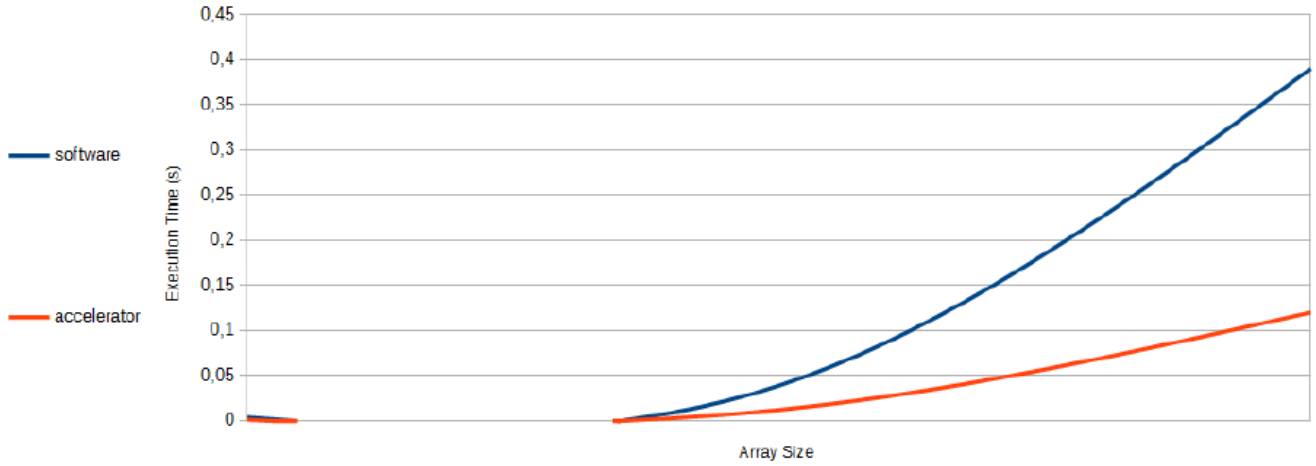
Τα DSP48E1 είναι slices τα οποία χρησιμοποιούνται για αριθμητικές πράξεις υψηλής ταχύτητας. Και οι δύο πόροι αυτοί βρίσκονται σε μικρή απόσταση μεταξύ τους στο πλέγμα του PL, καθώς οι εντατικοί υπολογισμοί και η ταχεία αποθήκευση και ανάγνωση δεδομένων είναι συχνά έννοιες αλληλένδετες.

Αξιολόγηση απόδοσης

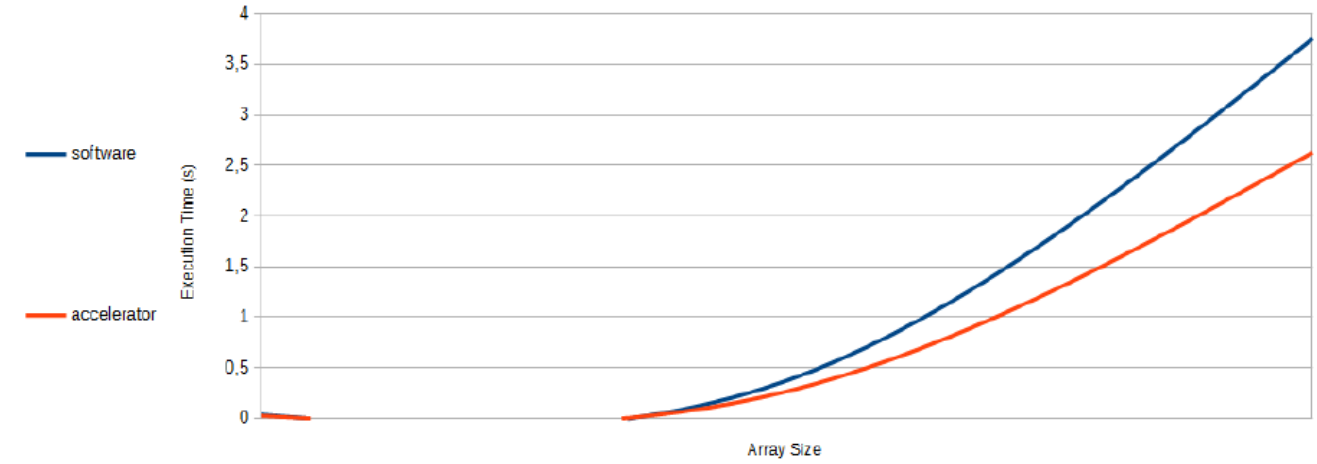
Η συνάρτηση που υλοποιεί τον αλγόριθμο που θέλουμε να επιταχύνουμε δέχεται 4 ορίσματα. Το μήκος του πίνακα size, το πλάτος κάθε γραμμής του πίνακα dim, το όριο threshold, καθώς και τον αριθμό αρχικοποίησης της συνάρτησης παραγωγής τυχαίων αριθμών (seed), η οποία δίνει τα δεδομένα στους πίνακες data0 και data1. Λόγω του σχεδιασμού της αρχιτεκτονικής, χρειαζόμαστε δεδομένο πλάτος πινάκων για τη σωστή δέσμευση πόρων από τα εργαλεία που χρησιμοποιούμε, δηλαδή κατά το προγραμμα- τισμό της FPGA. Επομένως δημιουργήσαμε δύο αρχιτεκτονικές, μία για δεδομένο dim = 4, και μία για δεδομένο dim = 16.

Για τη πειραματική αξιολόγηση της απόδοσης πάνω στο ZedBoard, η συνάρτηση είχε τα εξής ορίσματα: seed = 10, threshold = 100, size = 100000 – 1000000.

Μετράμε την απόδοση ως τον χρόνο που χρειάζεται για την ολοκλήρωση των υπολογισμών για αυξανόμενο μέγεθος πινάκων, και συγκρίνουμε την απόδοση του επιταχυντή με μία software single-thread υλοποίηση του αλγορίθμου.



Εικόνα 2: *Απόδοση για dim = 4*



Εικόνα 3: *Απόδοση για dim = 16*

Παρατηρούμε από τα αποτελέσματα, πως για την αρχιτεκτονική με δεδομένο dim = 4, πετυχαίνουμε μία επιτάχυνση 3.2x σε σχέση με την υλοποίηση σε λογισμικό. Στη περί- πτωση με δεδομένο dim = 16, πετυχαίνουμε μία μικρότερη επιτάχυνση, της τάξης του 1.5x.

Για την αρχιτεκτονική με δεδομένο dim = 4, υπολογίσαμε επίσης το θεωρητικό μέγιστο throughput της αρχιτεκτονικής, το οποίο είναι 25 εκατομμύρια υπολογισμοί ανά δευτε- ρόλεπτο. Στην υλοποίηση μας πετύχαμε ένα throughput 8,5 εκατομμύρια υπολογισμούς ανά δευτερόλεπτο, δηλαδή το 34% του θεωρητικά μέγιστου throughput.

Όσον αναφορά την χρήση των πόρων της FPGA, τα ποσοστά χρήσης του κάθε τύπου πόρου παρουσιάζονται παρακάτω, αρχικά για dim = 4:

Πόρος	Εν χρήση	Σύνολο διαθέσιμων	Ποσοστό εν χρήση
DSP	20	220	9,09%
BRAM	24	140	17,14%
LUT	17569	53200	33,02%
FF	21173	106400	19,9%

Για dim = 16:

Πόρος	Εν χρήση	Σύνολο διαθέσιμων	Ποσοστό εν χρήση
DSP	80	220	36,36%
BRAM	28	140	20%
LUT	36453	53200	68,52%
FF	57148	106400	53,71%

Από τα ποσοστά χρήσης πόρων, καθώς και από την επιτάχυνση που πετυχαίνουμε, παρατηρούμε πως η αρχιτεκτονική με δεδομένο το dim = 4 είναι η πιο αποδοτική εκ των δύο. Για αυτή την αρχιτεκτονική επίσης παρατηρούμε πως οι πόροι της FPGA που έχουμε στη διάθεσή μας αρκούν για μία λύση με δύο instances του επιταχυντή. Έτσι, χωρίζοντας τους πίνακες data1 και data2 στο ενδιάμεσο τους, καλούμε δύο instances του επιταχυντή από το πρόγραμμα ελέγχου του, το οποίο βρίσκεται στο PS. Η παράλληλη ασύγχρονη κλήση των δύο instances του επιταχυντή γίνεται μέσω των SDS directives async και wait, με το πρώτο να δημιουργεί ένα instance του επιταχυντή για τα ορίσματα που δίνουμε και το δεύτερο να δημιουργεί ένα barrier στο οποίο το πρόγραμμα σταματά έως ότου η κλήση του επιταχυντή να ολοκληρώσει τους υπολογισμούς.

Με αυτή τη λύση διπλού instance, παρατηρήσαμε πειραματικά πως πετυχαίνουμε περίπου τη διπλάσια επιτάχυνση σε σχέση με τη λύση με ένα instance για dim = 4, με επίσης διπλάσια χρήση πόρων πάνω στην FPGA για τη λογική των instances του accelerator.

Συμπεράσματα

Κλείνοντας, παρατηρούμε πως η ταχύτερη λύση είναι μία που χρησιμοποιεί όσους περισσότερους από τους διαθέσιμους πόρους γίνεται για ουσιαστικούς υπολογισμούς. Η λύση η οποία παρουσιάζεται εδώ θα μπορούσε να βελτιστοποιηθεί για μικρότερη χρήση πόρων ανά instance του επιταχυντή, έτσι ώστε να μπορέσουμε να έχουμε παραπάνω από δύο instances κάθε φορά.

Όσον αναφορά την αρχιτεκτονική για δεδομένο dim = 16, μία λύση για βελτίωση της απόδοσης είναι η κάθετη διαίρεση των υπολογισμών και όχι μόνο η οριζόντια, δηλαδή η κάθε γραμμή του πίνακα να διαιρείται για επεξεργασία σε περισσότερα από 4 ξεχωριστά κομμάτια.

Σε κάθε περίπτωση, παρατηρούμε τη σημαντική επιτάχυνση του αλγορίθμου με την χρήση των επιταχυντών βασισμένων σε FPGA.