HPY 519
ΑΝΑΔΙΑΤΑΣΣΟΜΕΝΑ ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

Αναφορά εργαστηριακής άσκησης

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

LAB59142122
Μανώλης Πετράκος ΑΜ 2014030009

Περιγραφή αρχικής συνάρτησης

Καλούμαστε να υλοποιήσουμε ένα επιταχυντή σε αναδιατασσόμενη λογική για την συνάρτηση *MyFunc()* που δίνεται. Δέχεται δύο πίνακες εισόδου (data0, data1) , ένα πίνακα εξόδου (data2), την οριακή τιμή των αποτελεσμάτων (threshold) και δύο μεταβλητές που ορίζουν το μέγεθος των πινάκων (size, dim). Η συνάρτηση υπολογίζει τα δεδομένα ανά dim-αδες[1] και ξεκινάει αρχικοποιώντας την έξοδο. Συνεχίζει κάνοντας τον υπολογισμό της dim-αδας και αν όλα τα δεδομένα της είναι μεγαλύτερα της οριακής τιμής, τα μηδενίζει. Από αυτήν τη διαδικασία φαίνεται ότι γίνονται μέχρι και τέσσερις προσβάσεις σε κάθε θέση του πίνακα εξόδου, τρεις εγγραφές και μία σύγκριση. Επίσης, τα δεδομένα εισόδου διαβάζονται πολλές φορές απευθείας από την μνήμη. Αν η συνάρτηση μεταφερθεί σε αναδιατασσόμενη λογική χωρίς αλλαγές, οι παραπάνω παρατηρήσεις θα δημιουργήσουν σημαντική επιβράδυνση.

Target device architecture

Η συσκευή προορισμού του επιταχυντή είναι το ZedBoard development board, το οποίο ανήκει στην οικογένεια Zynq-7000. Η οικογένεια βασίζεται στην αρχιτεκτονική System on Chip (SoC) και ενσωματώνει την δυνατότητα για προγραμματισμό λογισμικού πάνω σε έναν επεξεργαστή ARM αρχιτεκτονικής με την δυνατότητα για προγραμματισμό hardware πάνω σε μια FPGA. Εκτός από τις υπολογιστικές μονάδες (CPU & FPGA), περιέχει controllers και περιφερειακά όπως PCIe και DRAM. Σε αυτήν την αρχιτεκτονική, ο επεξεργαστής και η FPGA είναι απευθείας συνδεδεμένοι μέσω interconnect και έχουν και οι δύο πρόσβαση στην περιφερειακή μνήμη της συσκευής (DRAM).

1 Ως dim-αδα ορίζεται τα dim στοιχεία που αλληλοεξαρτούνται κατά τους υπολογισμούς.

Προετοιμασία περιβάλλοντος επιταχυντή

Πριν την υλοποίηση του επιταχυντή, πρέπει να δημιουργηθεί ένας μηχανισμός για την επαλήθευση των αποτελεσμάτων του. Αυτό γίνεται συγκρίνοντας τα, ένα προς ένα, με τα αποτελέσματα της συνάρτησης *myfunc()* όταν λειτουργούν με τα ίδια δεδομένα εισόδου. Στην σύγκριση, λαμβάνονται υπόψιν δύο δεκαδικά ψηφία για την αποφυγή false negative matches λόγω διαφορετικής ακρίβειας floats μεταξύ software και hardware. Η στρογγυλοποίηση γίνεται αξιοποιώντας την έτοιμη υλοποίηση της συνάρτησης *sprintf()*. Για την διευκόλυνση της σχεδίασης του επιταχυντή, σε περίπτωση λάθους αποτελέσματος, αυτό εμφανίζεται μαζί με την αναμενόμενη τιμή και την θέση του.

Εκτός από τον έλεγχο ορθότητας, πρέπει να δημιουργηθεί και ένας μηχανισμός για την αξιολόγηση της επίδοσης του επιταχυντή. Ως μέτρο σύγκρισης χρησιμοποιείται ο χρόνος εκτέλεσης. Αρχικά, χρονομετρούνται ξεχωριστά η αρχική συνάρτηση σε software και ο επιταχυντής σε hardware. Τέλος, εμφανίζονται οι χρόνοι εκτέλεσης, όπου μπορούν να χρησιμοποιηθούν για τον υπολογισμό του σχετικού speedup.

Σχεδιαστικές αποφάσεις

2 επιταχυντές

Η πρώτη σχεδιαστική απόφαση που λήφθηκε, είναι να δημιουργηθούν 2 επιταχυντές ίδιας αρχιτεκτονικής, ένας για dim ίσο με 4 (accel4) και ένας για dim ίσο με 16 (accel16). Αλλάζοντας το dim, αλλάζει και ο τρόπος που υπολογίζεται η έξοδος επειδή το κάθε στοιχείο εξαρτάται από την dim-αδα που ανήκει. Επειδή το hardware δεν μπορεί να γίνει allocate κατά την λειτουργία του επιταχυντή, για να εξυπηρετούνται και οι δύο περιπτώσεις ικανοποιητικά, πρέπει να σχεδιαστεί ένας επιταχυντής για την κάθε μία. Για να υλοποιηθεί αυτό, υπάρχουν δύο συναρτήσεις (accel4 και accel16) όπου έχει αντικατασταθεί το dim με την αντίστοιχη σταθερά (4 ή 16). Έτσι τα όρια των εσωτερικών loops είναι γνωστά και μπορούν εύκολα να μεταφραστούν σε hardware. Επιπλέον, το μέγεθος του πίνακα data0 γίνεται γνωστό.

Διάβασμα πινάκων

Η δεύτερη σχεδιαστική αλλαγή αφορά το διάβασμα των δεδομένων εισόδων. Σκοπός της είναι η ελαχιστοποίηση των προσβάσεων στη μνήμη ανά υπολογισμό. Παρατηρώντας την λειτουργία της συνάρτησης, φαίνεται ότι ο πίνακας data0 επαναχρησιμοποιείται συνεχώς, ενώ τα δεδομένα του πίνακα data1 χρησιμοποιούνται μόνο για τον υπολογισμό της αντίστοιχης dim-αδας. Εφόσον το μέγεθος του πρώτου πίνακα είναι γνωστό και αρκετά μικρό, μπορεί να μεταφερθεί σε έναν τοπικό πίνακα. Αντίθετα, για τον δεύτερο, υπάρχει ένας τοπικός πίνακας

μεγέθους dim όπου αποθηκεύονται προσωρινά μόνο τα δεδομένα που χρειάζονται για τον επόμενο υπολογισμό. Για να υλοποιηθεί αυτό, πρέπει να μεταφέρονται τα κατάλληλα δεδομένα στους τοπικούς πίνακες και να χρησιμοποιούνται αυτοί κατά τους υπολογισμούς. Στο hardware οι πίνακες θα μεταφραστούν ως registers όπου έχουν αρκετά μικρότερο χρόνο πρόσβασης από την dram.

Αποθήκευση Αποτελεσμάτων

Η τρίτη απόφαση αλλάζει τον τρόπο που αποθηκεύονται τα αποτελέσματα στην μνήμη. Πρώτον, η μνήμη δεν χρειάζεται να αρχικοποιηθεί αν χρησιμοποιηθούν τοπικοί accumulators κατά τον υπολογισμό. Σαν αποτέλεσμα, αποφεύγεται το ένα τρίτο των εγγραφών στη μνήμη που έκανε η αρχική συνάρτηση. Επίσης, αποφεύγεται η πρόσβαση στην μνήμη κατά την σύγκριση με το threshold, αφού αυτή μπορεί να γίνει απευθείας με τους accumulators. Τέλος, γίνεται μόνο μία αποθήκευση στην μνήμη. Αν ικανοποιείται το threshold αποθηκεύονται τα αποτελέσματα των accumulator, αλλιώς μηδενίζεται. Αυτή η διαδικασία μεταφράζεται στο hardware ως παραπάνω registers όπου αποθηκεύονται τα ενδιάμεσα αποτελέσματα.

Πλάτος καναλιών πινάκων

Το εργαλείο Xilinx Vivado HLS θέτει αυτόματα το πλάτος ενός καναλιού (bus) ίσο με το μέγεθος του τύπου των δεδομένων που μεταφέρει. Αναγνωρίζει τους τύπους από την δήλωση της συνάρτησης του επιταχυντή. Σαν αποτέλεσμα, στους πίνακες μπορεί να προσπελασθεί μόνο μια θέση ανά κύκλο και εμποδίζεται η αύξηση της παραλληλίας στον επιταχυντή. Πράγματι, υλοποιώντας μόνο τις προηγούμενες αποφάσεις, το initiation interval των pipelined επιταχυντών είναι ίσο με την μεταβλητή dim. Δηλαδή, παράγει μία dim-αδα αποτελεσμάτων ανά 4 ή 16 κύκλους. Επιθυμητό είναι αυτό το νούμερο να είναι το ελάχιστο δυνατό. Για να γίνει αυτό, πρέπει να “ξεγελαστεί” το εργαλείο ώστε να μπορούν να προσπελασθούν ταυτόχρονα περισσότερες θέσεις των πινάκων. Πιο συγκεκριμένα, αυτό χρειάζεται να γίνει για τους πίνακες αγνώστου μεγέθους data1 και data2.

Για να μεγαλώσει το πλάτος των καναλιών του επιταχυντή, αλλάζει ο τύπος των δεδομένων στη δήλωση της συνάρτησης του. Οι δείκτες των κατάλληλων πινάκων, αντί να είναι τύπου float, μετατρέπονται ώστε να είναι ενός τύπου με μέγεθος ίσο με το επιθυμητό πλάτος του bus. Χρησιμοποιώντας την βιβλιοθήκη “*ap_int.h*”, κατασκευάζεται ένας τύπος ap_uint με το κατάλληλο μέγεθος. Τότε, για να χρησιμοποιηθούν τα δεδομένα εισόδου, πρέπει να γίνουν unpack σε μεταβλητές τύπου float. Επίσης, για να αποθηκευτούν τα δεδομένα εξόδου, πρέπει να γίνουν pack στο νέο τύπο δεδομένων.

Το μέγεθος του τύπου διαφέρει σε κάθε υλοποίηση του επιταχυντή λόγο διαφορετικού αρχικού initiation interval. Επίσης, περιορίζεται και από τους διαθέσιμους πόρους του target device καθώς όσο μεγαλώνει, αυξάνεται η κατανάλωση πόρων. Στους παρακάτω πίνακες φαίνεται πως το μέγεθος του τύπου επηρεάζει το initiation interval και την κατανάλωση πόρων.

Accel4 synthesize estimations

Size (bits)	Initiation interval (cycles)	BRAM_18K %	DSP48E %	FF %	LUT %	URAM %
32	4	0	9	3	8	0
64	2	0	18	4	14	0
128	1	0	36	7	24	0

Accel16 synthesize estimations

Size (bits)	Initiation interval (cycles)	BRAM_18K %	DSP48E %	FF %	LUT %	URAM %
32	16	0	36	27	40	0
64	8	0	72	40	69	0
128	4	0	144	63	120	0

Από τους πίνακες φαίνονται οι κατάλληλες τιμές για το πλάτος του bus. Στον accel4 το ελάχιστο δυνατό initiation interval επιτυγχάνεται με μέγεθος ίσο με 128 bits. Δηλαδή, διαβάζονται και γράφονται τέσσερις floats ανά κύκλο. Στον accel16, με πλάτος 128 bits και παραπάνω, οι ζητούμενοι πόροι ξεπερνάνε τους διαθέσιμους. Άρα το μεγαλύτερο υλοποιήσιμο μέγεθος για την πλατφόρμα που χρησιμοποιείται είναι 64 bits. Τότε, διαβάζονται και γράφονται δύο floats ανά κύκλο.

Μεταβλητές σταθερής υποδιαστολής

Παρατηρώντας την κατασκευή των δεδομένων εισόδου στο test-bench, φαίνεται ότι οι πιθανές τιμές τους είναι από 0.0 έως 9.9 με βήμα 0.1. Επίσης, οι τιμές των αποτελεσμάτων για τον accel4 είναι από 0.00 μέχρι 392.04 [2] με βήμα 0.01. Μπορούν να χρησιμοποιηθούν αριθμοί σταθερής υποδιαστολής για τους υπολογισμούς με σκοπό την μείωση των αναγκαίων πόρων. Χρησιμοποιώντας την βιβλιοθήκη “*ap_fixed.h*”, οι μικρότεροι lossless τύποι δεδομένων που μπορούν να κατασκευαστούν είναι της μορφής ap_ufixed<17,4> για την είσοδο και ap_ufixed<34,10> για την έξοδο. Ο πρώτος αριθμός ορίζει πόσα bits είναι πριν την υποδιαστολή και ο δεύτερος πόσα την ακολουθούν. Όλα τα δεδομένα εισόδου πρέπει να μετατραπούν από float στον νέο τύπο για να γίνουν οι υπολογισμοί και τα αποτελέσματα πρέπει να μετατραπούν αντίστροφα για να γίνει ο έλεγχος εγκυρότητας. Στον παρακάτω πίνακα φαίνεται η απόδοση και η κατανάλωση πόρων του accel4 με fixed και floating point αριθμούς, χωρίς να έχει γίνει η προηγούμενη βελτιστοποίηση.

2 $dim * 9.9^2$

Data type	Estimated clock	BRAM_18K %	DSP48E %	FF %	LUT %	URAM %
Floating point	7.974 ± 1.25	0	9	3	8	0
Fixed point	8.868 ± 1.25	0	7	~0	~0	0

Πράγματι, η κατανάλωση πόρων μειώνεται, αλλά δημιουργούνται δύο νέα προβλήματα. Πρώτον, το ρολόι γίνεται αισθητά πιο αργό. Επίσης, με κάθε αλλαγή στον επιταχυντή, όπως η προηγούμενη βελτιστοποίηση, η απόδοση του μειώνεται απαγορευτικά καθώς παραβιάζεται ο περιορισμός των 100MHz συχνότητας λειτουργίας. Δεύτερον, ο επιταχυντής παράγει σωστά αποτελέσματα μόνο για test-benches με το ίδιο εύρος δεδομένων σαν αυτό που δίνεται. Εφόσον, δεν υπάρχει εγγύηση ότι θα χρησιμοποιηθεί μόνο με τέτοιου του είδους δεδομένα, η λύση είναι ελλιπής. Για αυτούς τους λόγους η παραπάνω αλλαγή **απορρίφθηκε**.

HLS directives

Το εργαλείο HLS, για να μπορεί να κάνει synthesize και C/RTL cosimulation, χρειάζεται να του προσδιοριστεί με πιο πρωτόκολλο επικοινωνίας λειτουργούν οι θύρες του επιταχυντή. Για αυτό χρησιμοποιείται η ντιρεκτίβα INTERFACE. Τα σήματα dim, size και threshold έχουν γίνει bundle σε ένα port που λειτουργεί με την μέθοδο ap_stable. Με αυτό, το εργαλείο καταλαβαίνει ότι τα σήματα είναι σταθερά και του επιτρέπει να κάνει εσωτερικές βελτιστοποιήσεις για να αφαιρέσει περιττούς καταχωρητές. Ο πίνακας data0 λειτουργεί με την ίδια μέθοδο καθώς το μέγεθος του είναι γνωστό και τα δεδομένα του δεν αλλάζουν κατά την λειτουργία του επιταχυντή.

Οι πίνακες data1 και data2 είναι αγνώστου μεγέθους και χρειάζονται διαφορετικό μέθοδο επικοινωνίας. Επίσης, η πρόσβαση των δεδομένων γίνεται σειριακά και μόνο μια φορά το καθένα. Η μέθοδος που προσομοιώνει καλύτερα αυτήν την συμπεριφορά είναι η ap_axis. Επίσης, το εργαλείο πρέπει να γνωρίζει κατά το RTL simulation των αριθμό των προσβάσεων που γίνεται στη μνήμη για τον κάθε πίνακα. Του προσδιορίζεται με το argument depth=<int>. Για να λειτουργήσει το simulation πρέπει η τιμή του να είναι ίση με την μεταβλητή size.

Προκαθορισμένα, στο Vivado HLS ο κάθε πίνακας υλοποιείται σε μνήμη με ένα port. Σαν αποτέλεσμα, ανά κύκλο μπορεί να γίνει διάβασμα ή γράψιμο μόνο σε μία θέση κάθε πίνακα. Ο επιταχυντής απαιτεί να διαβάζονται περισσότερες ή και όλες οι θέσεις των πινάκων του για να λειτουργήσει αποδοτικά. Για αυτό χρησιμοποιείται η ντιρεκτίβα array_partition, με το argument complete. Έτσι, το εργαλείο υλοποιεί τον πίνακα ως μια συλλογή από καταχωρητές και μπορεί να γίνει ταυτόχρονη πρόσβαση σε όλες τις θέσεις του. Η ντιρεκτίβα χρησιμοποιείται για τους εσωτερικούς πίνακες που χρησιμοποιούνται στους υπολογισμούς.

Επειδή τα δεδομένα του data0 επαναχρησιμοποιούνται συνεχώς, μεταφέρονται σε έναν τοπικό πίνακα πριν ξεκινήσει η κανονική λειτουργία του επιταχυντή. Η αντιγραφή των δεδομένων του γίνεται με ένα loop στην αρχή του προγράμματος το οποίο προσθέτει μια καθυστέρηση. Για να μειωθεί αυτό το overhead, χρησιμοποιείται η ντιρεκτίβα unroll. Σαν αποτέλεσμα, ο πίνακας αντιγράφεται ολόκληρος σε ένα κύκλο. Επίσης, το εργαλείο βλέποντας ότι γίνονται στατικά ταυτόχρονες εγγραφές στον τοπικό πίνακα, καταλαβαίνει ότι το καταλληλότερο είδος μνήμης είναι οι καταχωρητές και δεν χρειάζεται να εφαρμοστεί η προηγούμενη ντιρεκτίβα σε αυτόν.

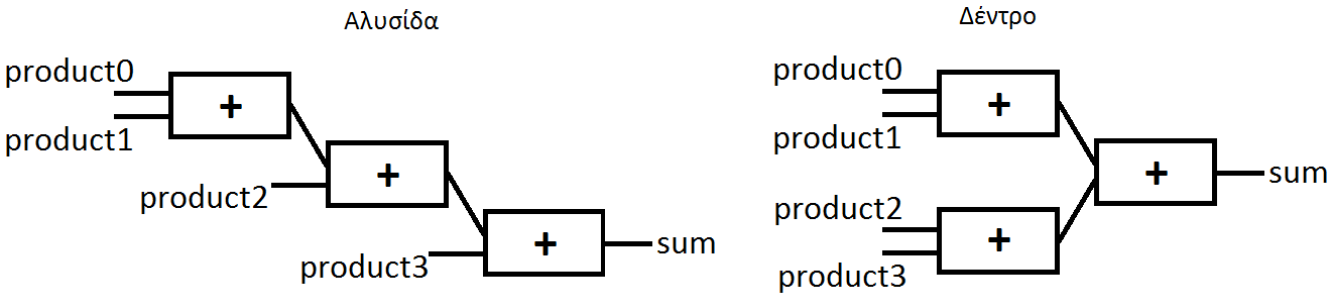
Το Vivado HLS μπορεί να εξάγει το συνολικό latency κάθε loop εφόσον γνωρίζει τον πλήθος των επαναλήψεων του. Στις περιπτώσεις όπου η καθυστέρηση του βρόχου είναι άγνωστη ή δεν μπορεί να υπολογιστεί, χρησιμοποιείται η ντιρεκτίβα loop_tripcount για να καθοριστούν ο ελάχιστος και ο μέγιστος αριθμός επαναλήψεων. Τότε, το εργαλείο μπορεί να αναλύσει πως το loop συμβάλει στην συνολική καθυστέρηση του επιταχυντή. Συνεπώς, κατά τον σχεδιασμό μπορούν να βρεθούν και να συγκριθούν βελτιστοποιήσεις κατάλληλες για το loop. Η ντιρεκτίβα δεν επιφέρει καμία αλλαγή στην απόδοση ή την λειτουργία του επιταχυντή αλλά βοηθάει κατά την σχεδίαση του.

Η σημαντικότερη ντιρεκτίβα για την αποδοτική λειτουργία του επιταχυντή είναι το PIPELINE. Ελαττώνει το initiation interval (II) [3] μιας συνάρτησης ή ενός βρόγχου επιτρέποντας την ταυτόχρονη εκτέλεση των επιμέρους λειτουργιών του. Επίσης, κάνει unroll σε όλα τα εμφωλευμένα loops της συνάρτησης ή του βρόγχου που βρίσκεται. Το επιθυμητό II καθορίζεται από την επιλογή II=<int> με την τιμή 1 και το εργαλείο προσπαθεί να ολοκληρώσει μια επανάληψη του loop ανά κύκλο. Αυτό είναι δυνατό στον επιταχυντή accel4 αλλά όχι στο επιταχυντή accel16 όπου σε αυτήν την περίπτωση δημιουργεί μια σχεδίαση με το ελάχιστο δυνατό II το οποίο είναι ίσο με 8 κύκλους.

Η θέση της ντιρεκτίβας στον κώδικα επηρεάζει σε πολύ μεγάλο βαθμό την RTL σχεδίαση που παράγεται. Αν εφαρμοστεί σε ένα πολύ εμφωλευμένο loop προσφέρει ελάχιστη βελτίωση καθώς όλο το υπόλοιπο σύστημα δουλεύει σειριακά. Αντίθετα, αν εφαρμοστεί σε όλο τον επιταχυντή, το παραγόμενο RTL κύκλωμα μπορεί να γίνει υπερβολικά μεγάλο για να χωρέσει στο target device. Επίσης, δεν είναι δυνατόν για το εργαλείο να το εφαρμόσει σε loop αγνώστου μεγέθους. Για αυτούς τους λόγους, εφαρμόστηκε στο σώμα του size loop, όπου δημιουργεί pipeline με όλα τα περιεχόμενα του και τα περιεχόμενα των εμφωλευμένων loops τα οποία έχει κάνει unroll.

3 Ως II ορίζεται ο χρόνος μεταξύ δύο διαδοχικών επαναλήψεων.

Η τελευταία HLS ντιρεκτίβα που χρησιμοποιείται είναι το expression_balance. Ο κώδικας του επιταχυντή για τους υπολογισμούς οδηγεί σε μια μακριά αλυσίδα RTL λειτουργιών. Με αυτήν την ντιρεκτίβα, ζητείται ρητά από το εργαλείο να αναδιατάξει τις λειτουργίες χρησιμοποιώντας προσεταιριστικές και αντιμεταθετικές ιδιότητες. Αυτή η αναδιάταξη δημιουργεί ένα ισορροπημένο δέντρο πράξεων με αποτέλεσμα την μείωση του latency του επιταχυντή και των αναγκαίων πόρων. Στο παρακάτω σχεδιάγραμμα φαίνεται η διαφορά μεταξύ αλυσίδας και δέντρου πράξεων για dim = 4.

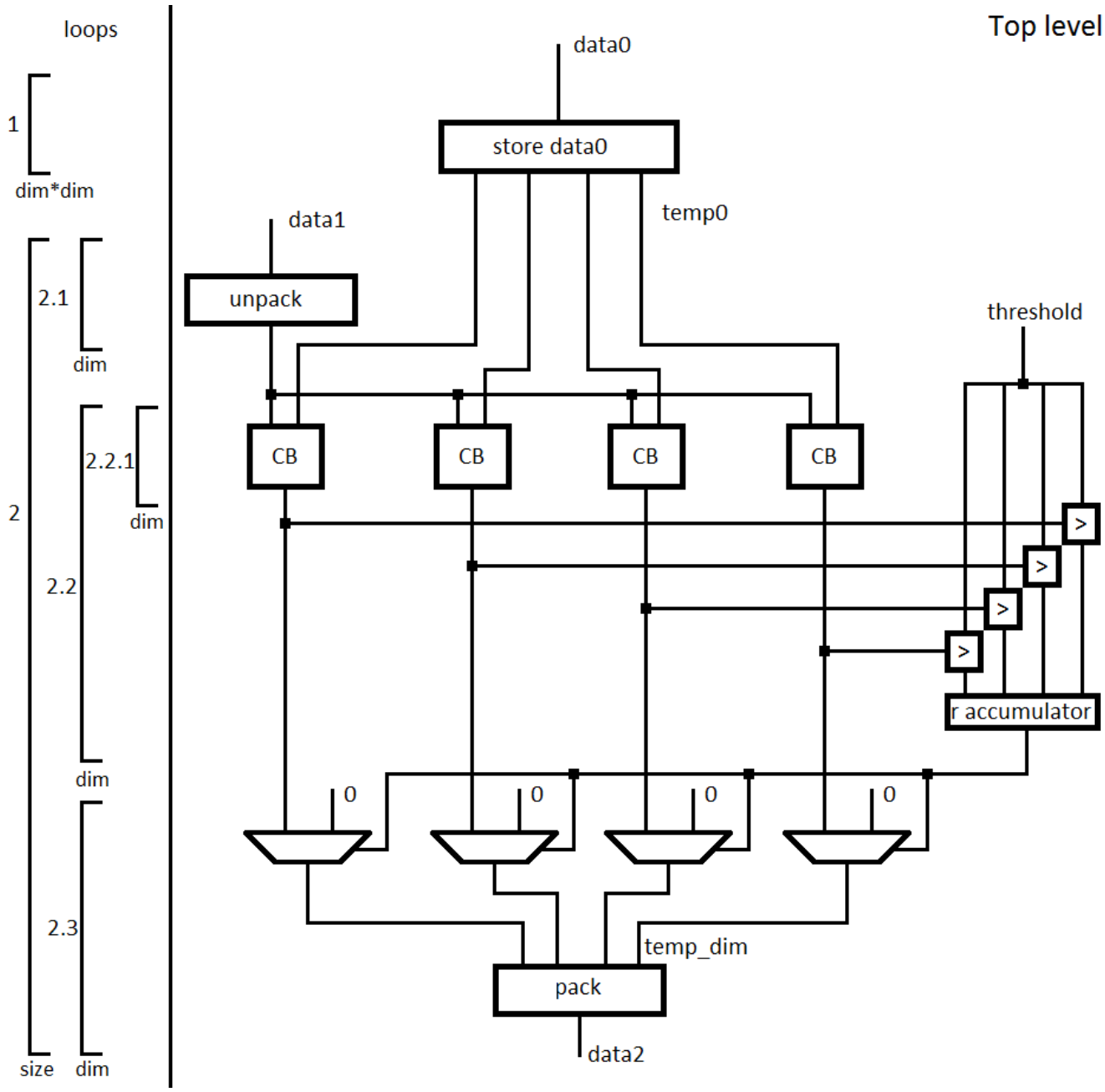


Περιγραφή λειτουργίας επιταχυντή

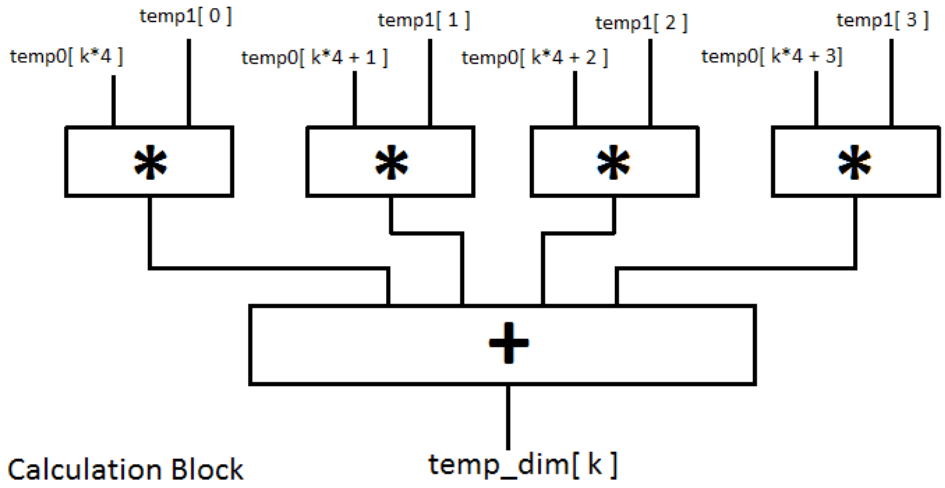
Συνοπτικά η λειτουργία του επιταχυντή είναι ως εξής:

1. Διάβασμα και μεταφορά πίνακα data0 σε τοπική μνήμη. Γίνεται μόνο μια φορά ανά κάλεσμα του επιταχυντή. (loop 1)
2. Διάβασμα dim-άδας πίνακα data1 και unpack από ap_uint σε floats. (loop 2.1)
3. Υπολογισμός αποτελεσμάτων. (loop 2.2.1)
4. Υπολογισμός μεταβλητής r που δείχνει αν όλα τα αποτελέσματα έχουν περάσει το threshold ή όχι. (loop 2.2)
5. Επιλογή κατάλληλων τιμών σύμφωνα με το r, πακετάρισμα τους από floats σε ap_uint και αποθήκευση στην μνήμη. (loop 2.3)

Τα βήματα 2 με 5 (loop 2) επαναλαμβάνονται μέχρι να ολοκληρωθεί η λειτουργία του επιταχυντή. Ακολουθεί μια αφαιρετική σχηματική αναπαράσταση του σε αντιστοιχία με τα loops του κώδικα.


loops
Top level
data0
store data0
1
dim*dim
data1
temp0
unpack
2.1
dim
threshold
CB
CB
CB
CB
2.2.1
dim
2
2.2
dim
r accumulator
0
0
0
0
2.3
temp_dim
pack
data2
size
dim

Όπου CB:


temp1[0]
temp1[1]
temp1[2]
temp1[3]
temp0[k*4]
temp0[k*4 + 1]
temp0[k*4 + 2]
temp0[k*4 + 3]
*
*
*
*
+
Calculation Block
temp_dim[k]

SDSoC directives

Μέχρι αυτό το σημείο, ο επιταχυντής έχει υλοποιηθεί αλλά δεν έχει ενσωματωθεί στην αρχιτεκτονική SoC. Αυτό επιτυγχάνεται χρησιμοποιώντας το εργαλείο SDSoC με τις κατάλληλες ντιρεκτίβες και συναρτήσεις. Το εργαλείο καθορίζει το ρολόι λειτουργίας, προσδιορίζει πόσοι και ποιοι επιταχυντές θα εφαρμοστούν στο προγραμματιζόμενο υλικό και τέλος κατασκευάζει τα απαραίτητα data movers για την επικοινωνία μεταξύ επιταχυντή και επεξεργαστή ή μνήμης. Αφαιρούνται οι HLS ντιρεκτίβες INTERFACE καθώς δεν προσφέρουν τίποτα μετά το στάδιο του HLS και εφαρμόζονται SDSoC ντιρεκτίβες που στοχεύουν στην μεταφορά δεδομένων.

Καταρχάς, χρησιμοποιείται το SDSoC Environment API για το memory allocate των πινάκων. Αν αυτό γίνει με malloc, οι πίνακες είναι φυσικά κατακερματισμένοι καθώς το λειτουργικό σύστημα τους έχει αποθηκεύσει σύμφωνα με το page table του. Δηλαδή, ο κάθε πίνακας είναι διάσπαρτος στην μνήμη και το διάβασμα του από τον επιταχυντή επιβραδύνεται. Για αυτό αντικαθίσταται με την συνάρτηση sds_alloc η οποία αποθηκεύει τους πίνακες σε φυσικά συνεχόμενο χώρο. Σαν αποτέλεσμα ο επιταχυντής μεταφέρει τους πίνακες γρηγορότερα.

Η πρώτη ντιρεκτίβα που χρησιμοποιείται είναι η data mem_attribute με την επιλογή PHYSICAL_CONTIGUOUS για τους πίνακες data0, data1 και data2. Με αυτή, ο compiler καταλαβαίνει ότι οι πίνακες έχουν γίνει allocate μέσω της συνάρτησης sds_alloc και επιλέγει το κατάλληλο data mover.

Η δεύτερη ντιρεκτίβα που χρησιμοποιείται είναι η data zero_copy πάνω στους ίδιους πίνακες. Αυτή δείχνει στον compiler ότι η συνάρτηση του hardware προσεγγίζει τα δεδομένα των πινάκων απευθείας από την κοινόχρηστη μνήμη μέσω μιας διεπαφής AXI master bus και δεν αντιγράφονται στην μνήμη της αναδιατασσόμενη λογικής. Αυτό γίνεται γιατί τα δεδομένα των πινάκων διαβάζονται ή γράφονται μία φορά και δεν χρειάζεται να αποθηκευτούν στην μνήμη της αναδιατασσόμενης λογικής ή έχουν αποθηκευτεί σε registers από το προηγούμενο στάδιο.

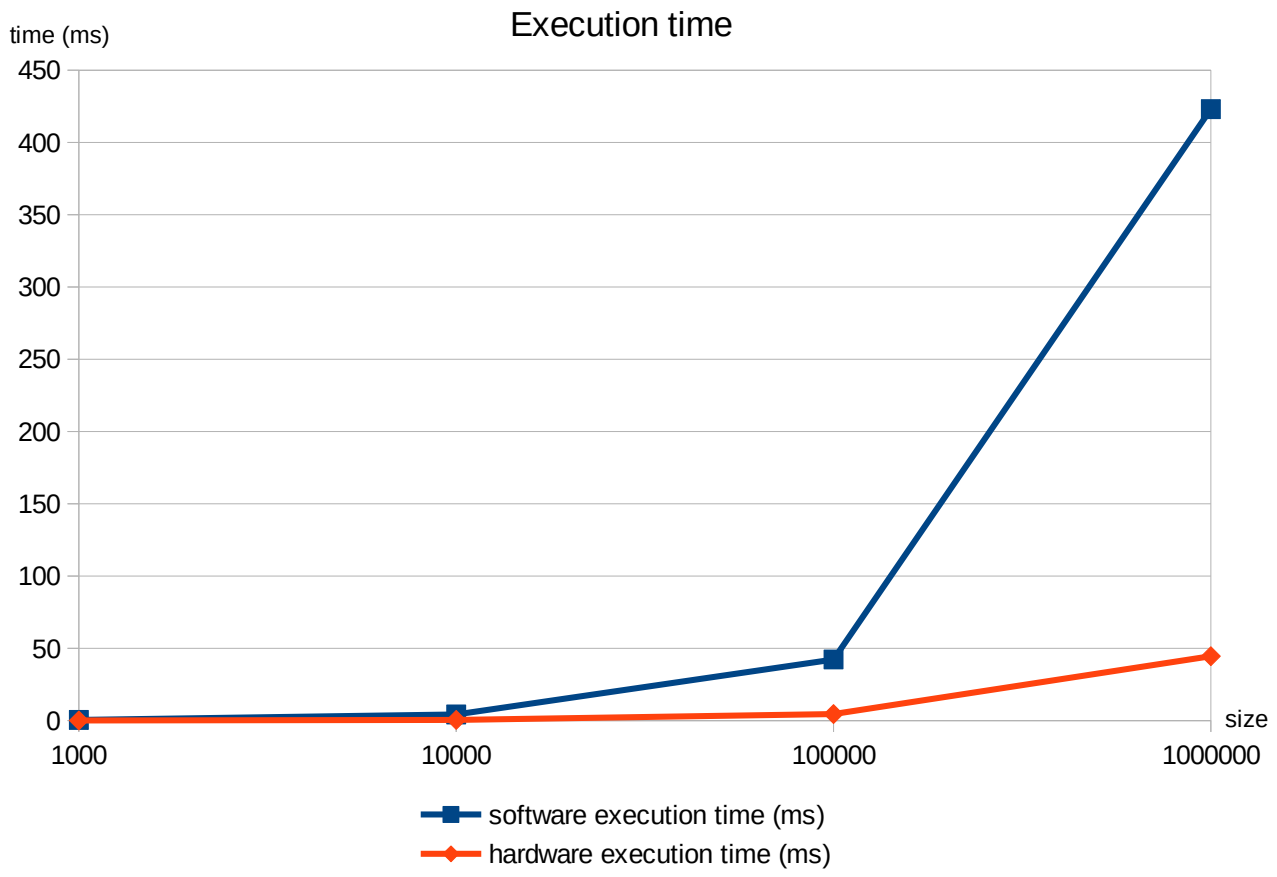
Η τελευταία ντιρεκτίβα που χρησιμοποιείται είναι η data access_patern με την επιλογή SEQUENTIAL για τους ίδιους πίνακες. Υποδεικνύει στον compiler ότι η πρόσβαση στα δεδομένα τους γίνεται σειριακά, ώστε αυτός να υλοποιήσει τα κατάλληλα πρωτόκολλα επικοινωνίας.

Με τις παραπάνω ντιρεκτίβες εξασφαλίζεται ότι ο compiler υλοποιεί τα κατάλληλα data movers και πρωτόκολλα επικοινωνίας για την βέλτιστη απόδοση του επιταχυντή.

Αξιολόγηση αποδόσεων

Για την αξιολόγηση της αρχιτεκτονικής, έγινε ένα benchmark μεταξύ του επιταχυντή accel4 στο hardware και της αρχικής συνάρτησης στο software. Στον παρακάτω πίνακα παρουσιάζονται οι χρόνοι εκτέλεσης τους συναρτήσει της μεταβλητής size που ορίζει το πλήθος των δεδομένων. Επίσης, φαίνεται και το σχετικό speedup μεταξύ software και hardware. Τέλος, οι χρόνοι παρουσιάζονται σε διάγραμμα για να φανεί γραφικά η διαφορά τους.

Size	1000	10000	100000	1000000
Software execution time (ms)	0.419	4.247	42.283	423.105
Hardware execution time (ms)	0.099	0.438	4.537	44.554
Speedup	4.23	9.696	9.32	9.496



Από τις παραπάνω μετρήσεις φαίνεται πως το speedup κυμαίνεται γύρω από το 9.5, δηλαδή για κάθε αποτέλεσμα που προκύπτει από software έχουν προκύψει σχεδόν δέκα από το hardware. Για λίγα δεδομένα (size = 1000), η διαφορά μεταξύ software και hardware είναι μικρότερη καθώς δεν υπάρχουν αρκετοί υπολογισμοί για να υπερκαλύψουν το overhead της κλήσης του επιταχυντή.

Για τον επιταχυντή accel16 δεν έγιναν μετρήσεις αλλά μπορεί να γίνει ένας θεωρητικός υπολογισμός του speedup του. Ο επιταχυντής accel4 παράγει 4 αποτελέσματα ανά κύκλο ενώ ο accel16 παράγει 2, άρα χρειάζεται περίπου[4] τον διπλάσιο χρόνο για τον υπολογισμό του ίδιου αριθμού δεδομένων[5]. Επίσης, ο χρόνος εκτέλεσης της αρχικής συνάρτησης τριπλασιάζεται όταν το dim αυξάνεται από 4 σε 16 και το πλήθος των δεδομένων παραμένει ίδιο. Αφού από dim 4 σε 16 ο επιταχυντής καταναλώνει το διπλάσιο χρόνο και η αρχική συνάρτηση καταναλώνει τον τριπλάσιο χρόνο, το speedup θα είναι 3/2 φορές μεγαλύτερο. Άρα το speedup του accel16 αναμένεται να είναι περίπου 15.

Περαιτέρω βελτιώσεις

Ο επιταχυντής accel4 χρησιμοποιεί περίπου το 36% των DSP48E του target device και μπορούν να χωρέσουν δύο από αυτούς. Έγινε προσπάθεια να υλοποιηθεί αυτό στο στάδιο του SDSoC χρησιμοποιώντας τις ντιρεκτίβες resourse και async. Με την πρώτη υλοποιούνται δύο επιταχυντές στην αναδιατασσόμενη λογική και με την δεύτερη καλούνται ασύγχρονα. Πράγματι, οι καταναλωθέντες πόροι διπλασιάστηκαν αλλά δεν εμφανίστηκε βελτίωση στις επιδόσεις. Το επόμενο βήμα για την ανάπτυξη του επιταχυντή, είναι η περαιτέρω εξερεύνηση αυτού του μέρους καθώς αναμένεται να διπλασιάσει το speedup αν λειτουργήσει σωστά.

Συμπεράσματα

Το περιβάλλον ανάπτυξης των HLS και SDSoC προσφέρουν μια πληθώρα από εργαλεία και τεχνικές για την ανάπτυξη επιταχυντών σε SoC συσκευές, χωρίς να χρειάζεται να υλοποιηθούν σε μια γλώσσα μοντελισμού hardware όπως η verilog και η VHDL. Σαν αποτέλεσμα, μειώνουν σημαντικά τον χρόνο ανάπτυξης και την πολυπλοκότητα σχεδίασης.

Επίσης, λόγω της αρχιτεκτονικής System on Chip, η επικοινωνία μεταξύ επεξεργαστή και επιταχυντή γίνεται trivial. Με μερικές αλλαγές στον αλγόριθμο, μπορεί μια software συνάρτηση να μεταφερθεί σε αναδιατασσόμενη λογική με ικανοποιητικό speedup χωρίς να παρεμβάλλεται η λειτουργία του υπόλοιπου προγράμματος. Γενικά, οι επιταχυντές είναι κατάλληλη λύση για computation-heavy αλγόριθμους.

4 Ο θεωρητικός υπολογισμός δεν λαμβάνει υπόψιν τα διάφορα overhead.
5 Έχουν τον ίδιο αριθμό δεδομένων όταν το size στον accel16 είναι ίσο με το 1/4 του size στον accel4.

Σημειώσεις

Το project υλοποιήθηκε σε αυτό το repository. Πέρα από τα αρχεία που έχουν σταλεί, υπάρχουν κώδικες για τις υλοποιήσεις που απορρίφθηκαν (ap_fixed, resource κ.α.) καθώς και synthesis reports για όλες.

Πηγές

HLS pragmas:

https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623-2.html

SDS pragmas:

https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/nmc1504034362475.html

SDSoC Environment API:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/api/reference_sdsoc_api.html

Vivado Design Suite User Guide HLS:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

Vivado HLS Optimization Methodology Guide:

https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/gkv1510957295794.html

Increasing Local Memory Bandwidth:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_increasing_local_memory_bandwidth.html

HLS co-simulation debug:

https://www.xilinx.com/Attachment/AR61063_VHLS_cosim_debug.pdf