ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Υλοποίηση ML-AI σε high-end FPGA

Διπλωματική εργασία
του
Σιδηρόπουλου Λεωνίδα

Επιβλέπων καθηγητής: Δρ. Ιωάννης Παπαευσταθίου

Θεσσαλονίκη, Νοέμβριος 2023

Περίληψη

Το HPL-AI benchmark έχει ως στόχο να ενώσει τα δύο πεδία των Υψηλής Απόδοσης Υπολογισμών και της Τεχνητής Νοημοσύνης. Ενώ τα παραδοσιακά HPC χρησιμοποιούν αριθμητική 64-bit, έχουν γίνει προσπάθειες κατά τη διάρκεια των χρόνων να μειωθεί η ακρίβεια των υπολογισμών σε αριθμητική κινητής υποδιαστολής 32-bit ή λιγότερο. Αυτή η μικρότερη ζήτηση για ακρίβεια τροφοδότησε μια αναζωπύρωση του ενδιαφέροντος για νέες πλατφόρμες υλικού που προσφέρουν ένα μείγμα πρωτοφανών επιπέδων απόδοσης και εξοικονόμησης ενέργειας για την επίτευξη των ίδιων αποτελεσμάτων με αυτά που παρέχονται από μορφές υψηλότερης ακρίβειας. Για το λόγο αυτό, υπάρχει μια εκτεταμένη χρήση των συσκευών FPGA με σκοπό τη βελτίωση της απόδοσης των υπολογιστικά πιο απαιτητικών τμημάτων του κώδικα. Αυτές οι συσκευές χρησιμοποιούν συχνά γλώσσες περιγραφής υλικού-HDL, όπως AHDL, VHDL ή Verilog, γεγονός που καθιστά την ανάπτυξη εφαρμογών πολύ πιο πολύπλοκη και χρονοβόρα. Για να κάνουμε την όλη διαδικασία πιο απλή, μπορούμε να χρησιμοποιήσουμε γλώσσες προγραμματισμού υψηλού επιπέδου, όπως C/C++, για να πετύχουμε τα ίδια αποτελέσματα, κάτι που είναι και ο κύριος στόχος της συγκεκριμένης διπλωματικής εργασίας. Αρχικά, διεξάγουμε μια χρονική ανάλυση χρησιμοποιώντας το VTune profiler της Intel για να εντοπίσουμε την πιο χρονοβόρα συνάρτηση του προγράμματος. Δεύτερον, τροποποιούμε τον κώδικα της εν λόγω συνάρτησης για να την κάνουμε πιο φιλική προς το υλικό και να βελτιώσουμε την απόδοσή της, χρησιμοποιώντας το Vitis HLS της Xilinx. Στη συνέχεια, μεταφέρουμε την τροποποιημένη συνάρτηση στο περιβάλλον Vitis, όπου εκτελούμε τον κώδικα στην FPGA Alveo U200 και παίρνουμε τα αποτελέσματα για προσομοίωση λογισμικού και έπειτα για εκτέλεση της συνάρτησης πάνω στο υλικό της FPGA. Τέλος, συγκρίνουμε τα αποτελέσματα με αυτά που λήφθηκαν νωρίτερα από το VTune και καταλήγουμε στο συμπέρασμα ότι έχουμε πετύχει μία $\times 3.7$ επιτάχυνση της συνάρτησης κατά την εκτέλεση του κώδικα στο υλικό του FPGA, χωρίς να αλλάξουμε τη λειτουργικότητά του.

Abstract

The HPL-AI benchmark strives to unite the two realms of High-Performance Computing and Artificial Intelligence. While traditional HPCs use 64-bit arithmetic there have been efforts over the years to reduce the accuracy of computations to 32-bit or less floating-point arithmetic. This lesser demand for accuracy fueled a resurgence of interest in new hardware platforms that deliver a mix of unprecedented performance levels and energy savings to achieve the same results as those by higher-accuracy formats. For this reason, there is an extensive use of FPGA devices for the purpose of improving the performance of the most compute-intensive parts of the code. These devices often use Hardware Description Languages-HDL, such as AHDL, VHDL or Verilog, which makes application development a lot more complex and time-consuming. In order to make the whole process simpler, we can use high level programming languages, such as C/C++, to achieve the same results, which is the main goal of this particular thesis. First, we conduct a time analysis using Intel's VTune profiler to identify the most time-consuming function of the benchmark. Second, we modify the code of the said function in order to make it more hardware-friendly and improve its performance, using Xilinx Vitis HLS. Then, we transfer the modified function to the Vitis environment, where we run the code on the Alveo U200 FPGA and get the results for software emulation and then by executing the function on the FPGA hardware. Lastly, we compare the results with the ones obtained earlier from VTune and we come to the conclusion that we have achieved a $\times 3.7$ acceleration of the function when running the code on the hardware of the FPGA, without altering its functionality.

Ευχαριστίες

Θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες στον κύριο Ιωάννη Παπαευσταθίου, που είναι ο κύριος εμπνευστής της παρούσας διπλωματικής, για την εμπιστοσύνη που έδειξε στο πρόσωπό μου αναθέτοντας μου την εκπόνησή της. Οι υποδείξεις του και η συμβολή του ήταν καθοριστικές για την ολοκλήρωση της εργασίας. Επίσης θα ήθελα να ευχαριστήσω τους κυρίους Νικόλαο Ταμπουρατζή και Άγγελο Αθανασιάδη για την βοήθεια που μου παρείχαν στα τεχνικά ζητήματα που προέκυψαν.

Επιπλέον θα ήθελα να απευθύνω τις ευχαριστίες μου σε όλο το διδακτικό προσωπικό του τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Αριστοτέλειου Πανεπιστημίου Θεσσαλονίκης για τις γνώσεις που μου μετέδωσαν κατά τη διάρκεια των σπουδών μου.

Τέλος, ευχαριστώ όσους βρίσκονται στο οικογενειακό και φιλικό μου περιβάλλον και με στηρίζουν τόσα χρόνια.

Περιεχόμενα

Κεφάλαιο 1: Εισαγωγή

1.1 Περιγραφή του Θέματος

Η ταχεία ανάπτυξη στο πεδίο της μηχανικής μάθησης δημιουργεί ολοένα και αυξανόμενες απαιτήσεις στο υλικό των υπολογιστών σε ενέργεια, ταχύτητα και μνήμη. Για την κάλυψη των παραπάνω αναγκών, το υλικό των σύγχρονων υπολογιστικών συστημάτων χρησιμοποιεί αριθμητικές πράξεις κινητής υποδιαστολής μικρότερης ακρίβειας, με σκοπό την επίλυση αριθμητικών πράξεων πολύπλοκων αλγορίθμων μονής και διπλής ακρίβειας. Πραγματοποιώντας, έτσι, ένα κομμάτι των πράξεων των αλγορίθμων αυτών σε μειωμένη ακρίβεια έχουμε υψηλό κέρδος σε ταχύτητα και μνήμη.

Το HPL-AI benchmark επιδιώκει να δώσει έμφαση στην αναδυόμενη σύγκλιση των φόρτων εργασίας μεταξύ των υψηλής απόδοσης υπολογισμών (HPC) και της τεχνητής νοημοσύνης (AI). Για τη μοντελοποίηση ορισμένων φαινομένων, όπως αυτά της χημείας, της φυσικής, της βιολογίας και άλλων, παρόλο που τα παραδοσιακά HPC εστιάζουν στις προσομοιώσεις, τα μαθηματικά μοντέλα τα οποία οδηγούν τους υπολογισμούς απαιτούν, σε μεγάλο βαθμό, 64-bit ακρίβεια. Ωστόσο, οι μέθοδοι μηχανικής μάθησης, οι οποίες είναι υπεύθυνες για την μεγάλη ανάπτυξη της τεχνητής νοημοσύνης, απαιτούν 32-bit ακρίβεια και χαμηλότερη αριθμητική κινητής υποδιαστολής, ώστε να επιτύχουν τα επιθυμητά αποτελέσματα. Η ολοένα και χαμηλότερη απαίτηση για ακρίβεια αποτελεί πόλο ενδιαφέροντος για νέες πλατφόρμες υλικού, οι οποίες προσφέρουν μια μίξη πρωτοφανών επιπέδων απόδοσης και εξοικονόμησης ενέργειας, επιτυγχάνοντας τα ίδια, περίπου, αποτελέσματα σε σύγκριση με τις μορφές υψηλής ακρίβειας.

Στόχος του HPL-AI είναι η ένωση των δύο αυτών πεδίων, παρέχοντας ένα συνδυασμό σύγχρονων αλγορίθμων και υλικού, ενώ παράλληλα συνδέεται με τις ήδη υπάρχουσες λύσεις παλαιών (έως και δεκαετίες)

HPL benchmarks στις μεγαλύτερες εγκαταστάσεις υπερυπολογιστών στον κόσμο. Η μέθοδος επίλυσης περιλαμβάνει ένα συνδυασμό LU παραγοντοποίησης με μία επαναληπτική βελτίωση, η οποία εκτελείται εκ των υστέρων για να επαναφέρει τη λύση σε 64-bit ακρίβεια. Η καινοτομία του HPL-AI έγκειται στην επιλογή χαμηλής ακρίβειας (έως και 16-bit) στην LU παραγοντοποίηση και, στη συνέχεια, σε μια περίπλοκη επανάληψη για την ανάκτηση της χαμένης ακρίβειας, λόγω της LU. Με αυτόν τον τρόπο γίνεται αποφυγή της απαίτησης για χρήση υψηλής ακρίβειας καθ' όλη τη διαδικασία της λύσης.

1.2 Στόχος της εργασίας

Σκοπός της εργασίας είναι να καταστήσουμε πιο αποδοτική την χρονική εκτέλεση του benchmark. Για να το πετύχουμε αυτό χρησιμοποιούμε μια συσκευή υψηλής υπολογιστικής ικανότητας, την FPGA. Σε πρώτο στάδιο, πραγματοποιούμε μια χρονική ανάλυση του benchmark, ώστε να εντοπίσουμε την πιο χρονοβόρα συνάρτηση που εκτελείται και στη συνέχεια κάνοντας τις κατάλληλες τροποποιήσεις, εκτελούμε τον κώδικα της συνάρτησης αυτής πάνω στο υλικό της FPGA.

1.3 Επισκόπηση εργαλείων

Για την χρονική ανάλυση του benchmark χρησιμοποιούμε το VTune profiler της Intel, το οποίο είναι ένα εργαλείο ανάλυσης απόδοσης για τον εντοπισμό hotspots και bottlenecks κατά τη βελτιστοποίηση του κώδικα. Στο κομμάτι εκτέλεσης του κώδικα, θα χρησιμοποιήσουμε τις γλώσσες προγραμματισμού υψηλού επιπέδου C/C++ και ορισμένα εργαλεία της Xilinx. Συγκεκριμένα, το Vitis HLS για να βελτιστοποιήσουμε τον κώδικα της πιο χρονοβόρας συνάρτησης και το Vitis, σε συνδυασμό με το περιβάλλον OpenCL για τον προγραμματισμό της FPGA και τη ρύθμιση της επικοινωνίας μεταξύ της FPGA και της CPU.

1.4 FPGA

1.4.1 Περιγραφή των FPGA

Τα FPGA ή αλλιώς Field - Programmable Gate Arrays είναι ψηφιακά ολοκληρωμένα ενσωματωμένα κυκλώματα (IC) που περιέχουν προγραμματιζόμενα λογικά μπλοκ μαζί με διαμορφώσιμες διασυνδέσεις μεταξύ των μπλοκ αυτών. Τα λογικά μπλοκ μπορούν να ρυθμιστούν κατάλληλα, ώστε να εκτελούν σύνθετες συνδυαστικές λειτουργίες ή να λειτουργούν ως απλές λογικές πύλες. Το “Field - Programmable” κομμάτι μιας FPGA αναφέρεται στο γεγονός ότι ο σχεδιαστής μπορεί να την προγραμματίσει, ακόμη και μετά την κατασκευή της, ώστε να εκτελέσει μία μεγάλη ποικιλία εργασιών. Διαθέτουν πολύ μεγάλο αριθμό τυποποιημένων πυλών, απαριθμητών, καταχωρητών μνήμης, γεννητριών PLL και άλλων. Ο προγραμματισμός των FPGA γίνεται σε γλώσσες περιγραφής υλικού (HDL), όπως για παράδειγμα Verilog, VHDL, AHDL.

Τα FPGA διαδραματίζουν πολύ σημαντικό ρόλο στην ανάπτυξη ενσωματωμένων συστημάτων, χάρη στην ικανότητά τους να ξεκινούν ταυτόχρονα την ανάπτυξη του λογισμικού συστήματος με το υλικό, να επιτρέπουν προσομοιώσεις συστήματος σε αρκετά πρώιμο στάδιο της ανάπτυξης και να επιτρέπουν διάφορες δοκιμές συστήματος και επαναλήψεις σχεδίασης πριν από την οριστικοποίηση της αρχιτεκτονικής του συστήματος.

1.4.2 Πλεονεκτήματα Χρήσης των FPGA

Μερικά πλεονεκτήματα της χρήσης των FPGA είναι:

1. **Επιτάχυνση υλικού (Hardware acceleration):** Τα FPGA μπορούν να επιταχύνουν υπολογιστικά εντατικές εργασίες, εκφορτώνοντας αυτές τις εργασίες από την CPU και ελευθερώνοντας πόρους του συστήματος. Η λειτουργία αυτή είναι πολύ χρήσιμη σε αρκετά επιστημονικά πεδία, όπως η τεχνητή νοημοσύνη (AI), η μηχανική μάθηση (machine learning) και οι υπολογισμοί υψηλής απόδοσης (HPC).

2. **Παράλληλη επεξεργασία**: Τα FPGA μπορούν να εκτελούν πολλαπλές λειτουργίες παράλληλα, γεγονός που μπορεί να οδηγήσει σε σημαντικές βελτιώσεις ταχύτητας και απόδοσης για εργασίες που μπορούν να παραλληλιστούν, όπως η επεξεργασία σήματος, κρυπτογραφία και επεξεργασία εικόνας και βίντεο.

3. **Χαμηλή καθυστέρηση:** Κατά την διάρκεια επεξεργασίας των δεδομένων οι FPGA παρέχουν χαμηλή καθυστέρηση (latency), κάτι που είναι πολύ χρήσιμο σε εφαρμογές όπου οι αποκρίσεις σε πραγματικό χρόνο είναι κρίσιμες, όπως στα αυτόνομα οχήματα, τη ρομποτική και σε συναλλαγές υψηλής συχνότητας (high-frequency trading).

4. **Μακρά διάρκεια ζωής:** Τα FPGA έχουν μεγαλύτερο κύκλο ζωής σε σύγκριση με πολλά προσαρμοσμένα ASIC, καθιστώντας τα κατάλληλα για εφαρμογές που απαιτούν εκτεταμένη υποστήριξη και συντήρηση.

5. **Προσαρμοσμένες διεπαφές I/O:** Τα FPGA επιτρέπουν στους σχεδιαστές να δημιουργούν προσαρμοσμένες διεπαφές εισόδου και εξόδου, καθιστώντας τις κατάλληλες για διασύνδεση με ένα ευρύ φάσμα αισθητήρων, περιφερειακών και πρωτοκόλλων επικοινωνίας.

6. **Ενεργειακή απόδοση:** Αναλόγως τη φύση των εργασιών που επιτελούν, τα FPGA μπορούν να είναι ενεργειακά πιο αποδοτικά σε σύγκριση με τις CPU ή τις GPU. Γενικά, η κατανάλωση ενέργειας είναι μία σημαντική παράμετρος στην οποία δίνεται μεγάλη έμφαση από πολλές εφαρμογές, όπως για παράδειγμα τα κέντρα δεδομένων και οι κινητές συσκευές.

7. **Μείωση του χρόνου ανάπτυξης εφαρμογών:** Τα FPGA παρέχουν τη δυνατότητα στους σχεδιαστές να εφαρμόζουν και να δοκιμάζουν γρήγορα και απλούστερα νέα σχέδια υλικού, χωρίς την ανάγκη ανάπτυξης προσαρμοσμένων ASIC, μειώνοντας έτσι σημαντικά τον χρόνο ανάπτυξης των εφαρμογών.

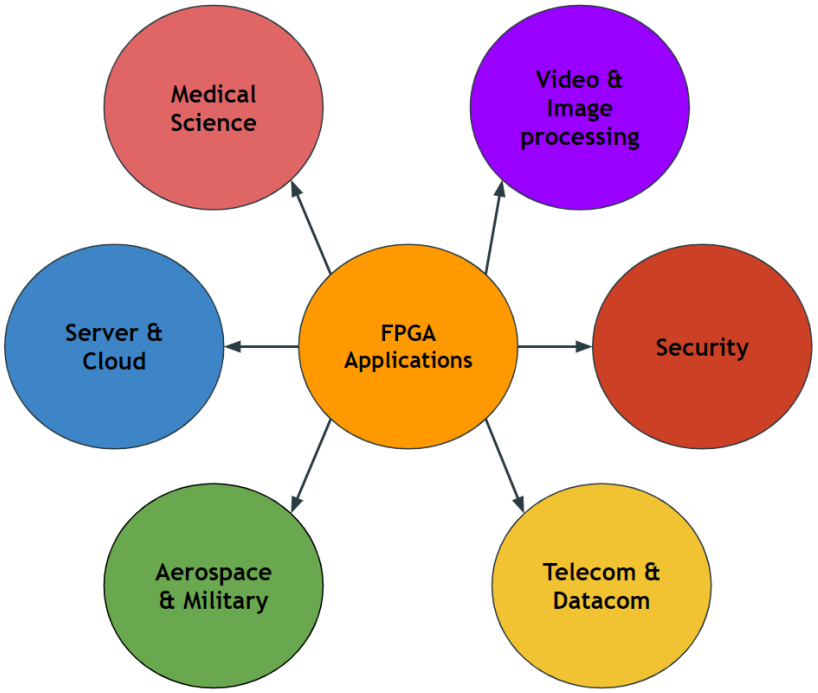
8. **Ασφάλεια:** Τα FPGA μπορούν να χρησιμοποιηθούν για εφαρμογές ασφαλείας σε επίπεδο υλικού, όπως η κρυπτογράφηση και ο έλεγχος πρόσβασης ενός συστήματος.

9. **Επαναπρογραμματισμός και επαναχρησιμοποίηση:** Ανάλογα με τις εργασίες που επιτελούν, τα FPGA μπορούν να επαναπρογραμματιστούν και επαναχρησιμοποιηθούν για να καλύψουν τις ανάγκες διαφόρων project, κάτι που τα καθιστά ως μία οικονομικά αποδοτική λύση μακροπρόθεσμα.

1.4.3 Εφαρμογές FPGA

Τα FPGA είναι ευρέως διαδεδομένα στην εποχή μας και βρίσκουν εφαρμογή σε πολλά επιστημονικά πεδία. Τα κυριότερα από αυτά είναι:

- Ιατρική
- Επεξεργασία βίντεο και εικόνας
- Τηλεπικοινωνίες και επικοινωνία μεταξύ των δεδομένων
- Server και cloud
- Ασφάλεια συστημάτων
- Αμυντική βιομηχανία και αεροδιαστημική



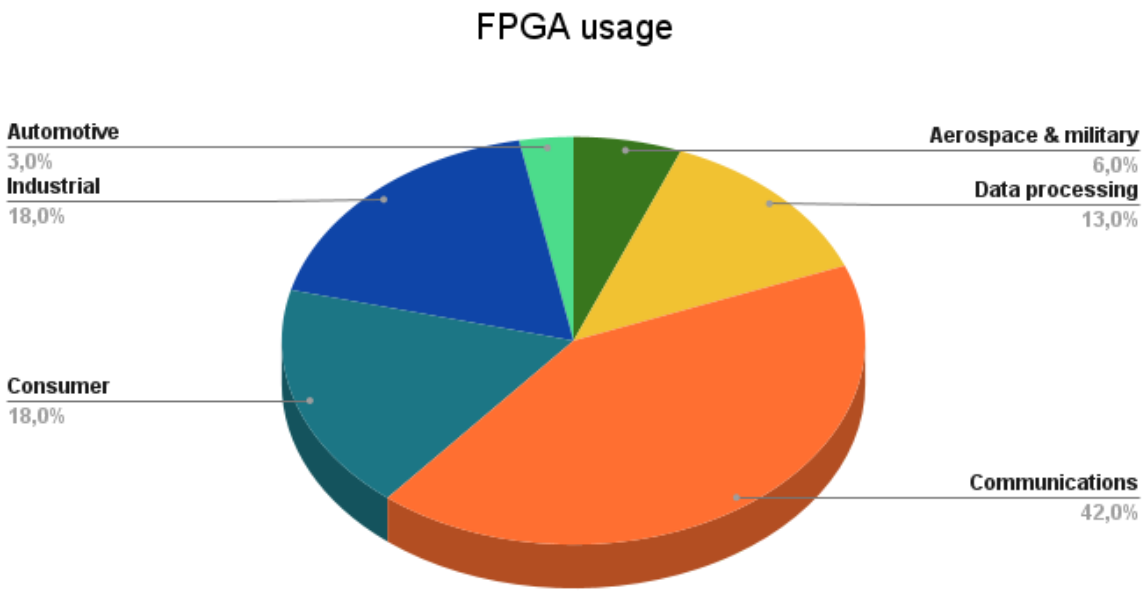
Εικόνα 1.1: Πεδία εφαρμογής των FPGA

Στην εικόνα 1.2 παρατηρούμε ότι η μεγαλύτερη χρήση των FPGA γίνεται στον τομέα των επικοινωνιών, όπου χρησιμοποιούνται τόσο σε ενσύρματες όσο και σε ασύρματες επικοινωνίες. Στις ενσύρματες επικοινωνίες χρησιμοποιείται σε σειριακά backplanes και στις ασύρματες επικοινωνίες, χρησιμοποιείται για λύσεις δικτύωσης και διευθυνσιοδότηση προτύπων WiMAX και στα 5G/6G. Τα FPGA χρησιμοποιούνται στην πλευρά της υποδομής βοηθώντας στη σύντομη σύναψη δεδομένων σε υψηλή ταχύτητα.

Στο πεδίο της ιατρικής, τα FPGA χρησιμοποιούνται σε ιατρικό εξοπλισμό για την επεξεργασία δεδομένων, καθώς και για διαγνωστικούς σκοπούς και σκοπούς παρακολούθησης.

Στον τομέα της αεροδιαστημικής και των αμυντικών εφαρμογών, τα FPGA χρησιμοποιούνται για επεξεργασία εικόνας, μερικές αναδιαμορφώσεις για SDR, καθώς και για δημιουργία κυματομορφών.

Τα FPGA χρησιμοποιούνται ευρέως για την ανάπτυξη νευρωνικών δικτύων, τα οποία τελικά θα οδηγήσουν στην παραγωγή και τη συντήρηση συστημάτων τεχνητής νοημοσύνης. Τα FPGA υψηλής απόδοσης μπορούν να βοηθήσουν ακόμη περισσότερο μια εφαρμογή σε σύγκριση με τις GPU, και επομένως προτιμώνται για τους σκοπούς της ανάπτυξης τεχνολογίας μηχανικής μάθησης.



Εικόνα 1.2: Ποσοστά χρήσης των FPGA ανά πεδίο

1.5 Δομή της Εργασίας

Στο **κεφάλαιο 1** γίνεται μια εισαγωγή στο θέμα και το σκοπό της εργασίας, μια συνοπτική περιγραφή του HPL-AI benchmark και των FPGA και μια σύντομη παρουσίαση των εργαλείων που θα χρησιμοποιηθούν.

Στο **κεφάλαιο 2** γίνεται αναλυτική περιγραφή της λειτουργίας του HPL-AI benchmark και μία αναφορά στη δομή και τη λειτουργικότητα των αρχείων πηγαίου κώδικα που το απαρτίζουν.

Στο **κεφάλαιο 3** πραγματοποιείται χρονική ανάλυση του benchmark, μέσω του λογισμικού ανάλυσης απόδοσης VTune profiler και επιλέγεται η πιο χρονοβόρα συνάρτηση του benchmark ως υποψήφια συνάρτηση για μεταφορά και εκτέλεση στην FPGA.

Στο **κεφάλαιο 4** γίνεται μεταφορά του κώδικα της υποψήφιας προς βελτιστοποίηση συνάρτησης-kernel στο Vitis HLS, πραγματοποιείται C synthesis του kernel και σχολιασμός των αποτελεσμάτων.

Στο **κεφάλαιο 5** γίνεται μεταφορά του κώδικα του benchmark και του kernel στο Vitis και δημιουργείται ο κώδικας του host σε OpenCL. Γίνεται ανάλυση και σχολιασμός των παραχθέντων αποτελεσμάτων για Software Emulation και Hardware και έπειτα ακολουθεί σύγκριση και σχολιασμός των αποτελεσμάτων του VTune και του Vitis.

Στο **κεφάλαιο 6** γίνεται μία σύνοψη της διπλωματικής εργασίας με προσωπικές παρατηρήσεις και παροτρύνσεις για μελλοντική επέκταση και βελτίωση της παρούσας εργασίας.

Στο **παράρτημα 1** είναι καταγεγραμμένες όλες οι συντομογραφίες και τα ακρωνύμια της παρούσας εργασίας.

Κεφάλαιο 2: HPL-AI benchmark

2.1 Ανάλυση του benchmark

Η βασική ιδέα του HPL-AI benchmark είναι η επίλυση ενός συστήματος γραμμικών εξισώσεων σε ακρίβεια κινητής υποδιαστολής 64-bit, χρησιμοποιώντας, αρχικά, τη μέθοδο LU παραγοντοποίησης ενός πίνακα σε χαμηλή ακρίβεια, με $\frac{2}{3}n^3 + O(n^2)$ πράξεις κινητής υποδιαστολής και στη συνέχεια ανάκτηση της χαμένης, υψηλής ακρίβειας των 64-bit από μία μέθοδο επαναληπτικής διόρθωσης, την GMRES.

2.2 Παραγοντοποίηση LU

Στην περίπτωση της LU παραγοντοποίησης, το προς επίλυση γραμμικό σύστημα εξισώσεων είναι της μορφής:

$$Ax = b$$

όπου:

- Α είναι ένας τετραγωνικός πίνακας διαστάσεων $n \times n$ με ακρίβεια κινητής υποδιαστολής 64-bit.
- b και x είναι μονοδιάστατοι πίνακες μεγέθους n με ακρίβεια κινητής υποδιαστολής 64-bit.

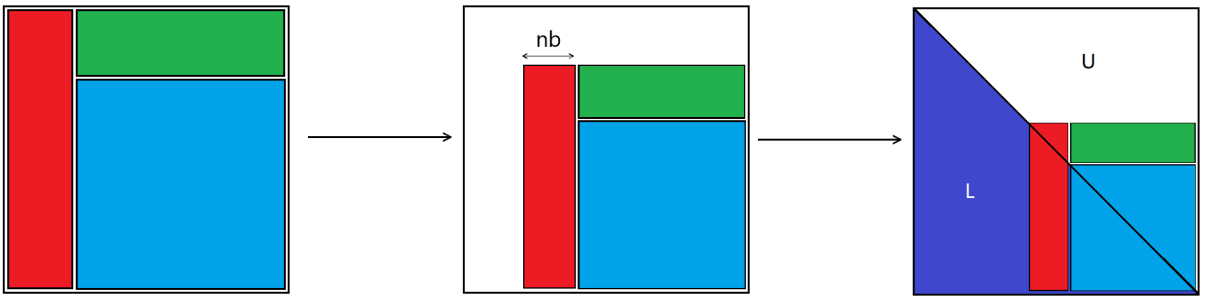
Ο πίνακας Α μπορεί να αναλυθεί σε γινόμενο 2 πινάκων, ενός κάτω τριγωνικού πίνακα L και ενός άνω τριγωνικού πίνακα U. Επομένως το σύστημα μετασχηματίζεται ως εξής:

$$Ax = b \Leftrightarrow LUx = b$$

Θέτουμε $y = Ux$ και έχουμε:

$$Ly = b$$

Η διαδικασία της LU παραγοντοποίησης χρησιμοποιεί μικτή ακρίβεια (32 και 16 bit) και πραγματοποιείται σταδιακά. Σε κάθε στάδιο εκτελούνται 2 λειτουργίες ταυτόχρονα. Η πρώτη λειτουργία αφορά την παραγοντοποίηση ενός τμήματος του πίνακα, το οποίο ονομάζεται panel και έχει μέγεθος nb. Στη δεύτερη λειτουργία γίνεται ενημέρωση των στοιχείων του πίνακα με χρήση υποπρογραμμάτων-ρουτίνων βασικής γραμμικής άλγεβρας (BLAS) επιπέδου 3 για πράξεις μεταξύ πινάκων. Μερικές από αυτές τις ρουτίνες είναι η TRSM για την επίλυση τριγωνικών πινάκων και η GEMM για πολλαπλασιασμό πινάκων. Σε αντίθεση με την πρώτη λειτουργία, η δεύτερη είναι αυτή που απαιτεί τους περισσότερους υπολογισμούς και άρα καταλαμβάνει το μεγαλύτερο μέρος του χρόνου εκτέλεσης της LU παραγοντοποίησης. Στο παρακάτω σχήμα παρουσιάζονται τα στάδια της LU παραγοντοποίησης, όπου με κόκκινο χρώμα συμβολίζουμε την πρώτη λειτουργία (περιοχή panel) και για την δεύτερη λειτουργία χρησιμοποιούμε τα χρώματα πράσινο και γαλάζιο για τις ρουτίνες TRSM και GEMM αντίστοιχα.



Εικόνα 2.1: Στάδια LU παραγοντοποίησης μεγέθους μπλοκ nb

Σημαντικό ρόλο στην απόδοση της παραγοντοποίησης παίζει και το μέγεθος μπλοκ nb, το οποίο η HPL χρησιμοποιεί για τη διανομή δεδομένων, καθώς και για την υπολογιστική ευαισθησία. Από την άποψη της διανομής δεδομένων, όσο μικρότερη είναι η τιμή του nb, τόσο καλύτερη είναι η ισορροπία φορτίου. Από υπολογιστική άποψη, μια πολύ

μικρή τιμή nb μπορεί να περιορίσει σε μεγάλο βαθμό την υπολογιστική απόδοση, επειδή δεν θα συμβεί σχεδόν καμία επαναχρησιμοποίηση δεδομένων στο υψηλότερο επίπεδο της ιεραρχίας της μνήμης. Οι αποτελεσματικές ρουτίνες πολλαπλασιασμού πινάκων συχνά αποκλείονται εσωτερικά. Μικρά πολλαπλάσια αυτού του παράγοντα αποκλεισμού είναι πιθανό να είναι “καλά” μεγέθη μπλοκ για την HPL. Η ουσία είναι ότι τα "καλά" μεγέθη μπλοκ βρίσκονται σχεδόν πάντα στο διάστημα [32,....,256]. Οι καλύτερες τιμές εξαρτώνται από την αναλογία απόδοσης υπολογισμού/επικοινωνίας του εκάστοτε συστήματος. Στην παρούσα εργασία το μέγεθος μπλοκ ορίζεται ως nb = 32.

2.3 Διαδικασία Επαναληπτικής Διόρθωσης GMRES

Κατά την επίλυση ενός γραμμικού συστήματος $Ax = b$, λόγω της συσσώρευσης σφαλμάτων στρογγυλοποίησης, η υπολογισμένη λύση $\hat{x}$ μπορεί να διαφέρει από την πραγματική λύση x_*. Για τη βελτίωση της ακρίβειας της λύσης του παραπάνω συστήματος σε 64 bit, γίνεται χρήση της μεθόδου Iterative Refinement (IR). Ορίζουμε την $x_1 = \hat{x}$ ως αρχική λύση και στη συνέχεια η IR υπολογίζει μία ακολουθία $\{x_1, x_2, x_3,...\}$, η οποία συγκλίνει στην x_*, όταν ικανοποιούνται ορισμένες παραδοχές. Απαραίτητη προϋπόθεση για την εκτέλεση της IR μεθόδου είναι η χρησιμοποίηση των παραγόντων που προκύπτουν από την διαδικασία της LU παραγοντοποίησης.

Για $n = 1, 2, 3,...$ η n-οστή επανάληψη της IR αποτελείται από τα εξής βήματα:

1. Υπολογισμός του υπολειπόμενου σφάλματος r_n από τη σχέση:

$$r_n = b - Ax_n$$

2. Επίλυση του συστήματος $Ac_n = r_n$ με άγνωστο τη διόρθωση c_n για την απαλοιφή του υπολειπόμενου σφάλματος r_n.
3. Προσθήκη της διόρθωσης c_n στην ήδη υπάρχουσα λύση για να λάβουμε την αναθεωρημένη επόμενη λύση: $x_{n+1} = x_n + c_n$

Η GMRES αποτελεί μία βελτιωμένη έκδοση της μεθόδου IR, η οποία προσεγγίζει τη λύση με το διάνυσμα σε ένα υποχώρο Krylov με ελάχιστο υπόλοιπο. Με αυτόν τον τρόπο εξασφαλίζεται μια πιο ικανοποιητική λύση του συστήματος $Ac_n = r_n$ σε σύγκριση με τη μέθοδο IR. Ο μέγιστος αριθμός επαναλήψεων στην GMRES είναι 50. Αν οι επαναλήψεις ξεπεράσουν το όριο αυτό, τότε το αποτέλεσμα δεν είναι έγκυρο. Το backward error δεν μπορεί να είναι μεγαλύτερο από 16 και υπολογίζεται από τον τύπο:

$$\frac{||Ax-b||_\infty}{||A||_\infty||x||_\infty+||b||_\infty} \times n \times \epsilon^{-1}$$

όπου:

- $Ax = b$ είναι το γραμμικό σύστημα προς επίλυση
- n είναι το μέγεθος του πίνακα A.
- $\epsilon \approx 1.1 \times 10^{-16}$ είναι η ακρίβεια μηχανής σε 64-bit αριθμητική κινητής υποδιαστολής (στις μηχανές IEEE $\epsilon \approx 2^{-53}$).

2.4 Οργάνωση αρχείων πηγαίου κώδικα

Το benchmark αποτελείται από τα παρακάτω αρχεία:

1. **blas.c:** Περιέχει τις συναρτήσεις βασικής γραμμικής άλγεβρας για πράξεις μεταξύ πινάκων.

2. **convert.c:** Περιέχει συναρτήσεις για τη μετατροπή των πινάκων από μονή σε διπλή ακρίβεια και το αντίστροφο.

3. **gmres.c:** Περιέχει τη συνάρτηση υλοποίησης της μεθόδου GMRES.

4. **hpl-ai.c:** Αποτελεί το κύριο αρχείο του προγράμματος, περιέχει τη main() συνάρτηση, η οποία εκτελεί τις βασικές λειτουργίες του benchmark, όπως την LU παραγοντοποίηση και την GMRES, επαληθεύει τα αποτελέσματα και τα τυπώνει στην κονσόλα.

5. **hpl-ai.h:** Σε αυτό το αρχείο γίνεται ορισμός όλων των συναρτήσεων του benchmark.

6. **matgen.c:** Περιέχει συναρτήσεις για το γέμισμα με τυχαίες τιμές των πινάκων και δημιουργείται ο διπλής ακρίβειας πίνακας A.

7. **print.c:** Εκτυπώνει στην κονσόλα τα στοιχεία των πινάκων για μονή και διπλή ακρίβεια.

8. **sgetrf_nopiv.c:** Σε αυτό το αρχείο εμπεριέχεται η υλοποίηση της LU παραγοντοποίησης.

9. **timer.c:** Μετράει τον χρόνο εκτέλεσης των συναρτήσεων του προγράμματος.

Λόγω της μεγάλης έκτασής τους, οι αρχικοί και τροποποιημένοι κώδικες όλων των αρχείων είναι συγκεντρωμένοι εδώ.

Κεφάλαιο 3: Intel VTune Profiler

3.1 Εισαγωγή στο VTune

Το VTune είναι ένα εργαλείο της Intel, το οποίο πραγματοποιεί χρονική ανάλυση ενός προγράμματος και παρέχει τη δυνατότητα στον χρήστη να εντοπίζει hotspots και bottlenecks στον κώδικα. Συγκεκριμένα, μερικές από τις αναλύσεις και μετρικές απόδοσης που διαθέτει το VTune και με τις οποίες θα ασχοληθούμε είναι οι εξής:

- **Performance Snapshot:** Καταγράφει μια εικόνα των διαφόρων τύπων ανάλυσης που είναι προσαρμοσμένοι για να εξετάζουν τις διάφορες πτυχές απόδοσης και παρουσιάζει μια επισκόπηση των λειτουργιών του προγράμματος. Με άλλα λόγια παρέχει στον χρήστη μία σύνοψη των ζητημάτων που επηρεάζουν την εφαρμογή ενδιαφέροντος.

- **Hotspots:** Εντοπίζει τα πιο χρονοβόρα τμήματα-συναρτήσεις του προγράμματος και τα ταξινομεί ανάλογα με τον χρόνο που δαπανάται για την εκτέλεση τους, από το μεγαλύτερο προς το μικρότερο.

- **Microarchitecture and Memory bottlenecks:** Εντοπίζει τα πιο σημαντικά προβλήματα που υπάρχουν στο κομμάτι του υλικού και επισημαίνει τα ζητήματα που σχετίζονται με την πρόσβαση στη μνήμη, όπως απώλεια μνήμης cache και προβλήματα υψηλού εύρους ζώνης (bandwidth).

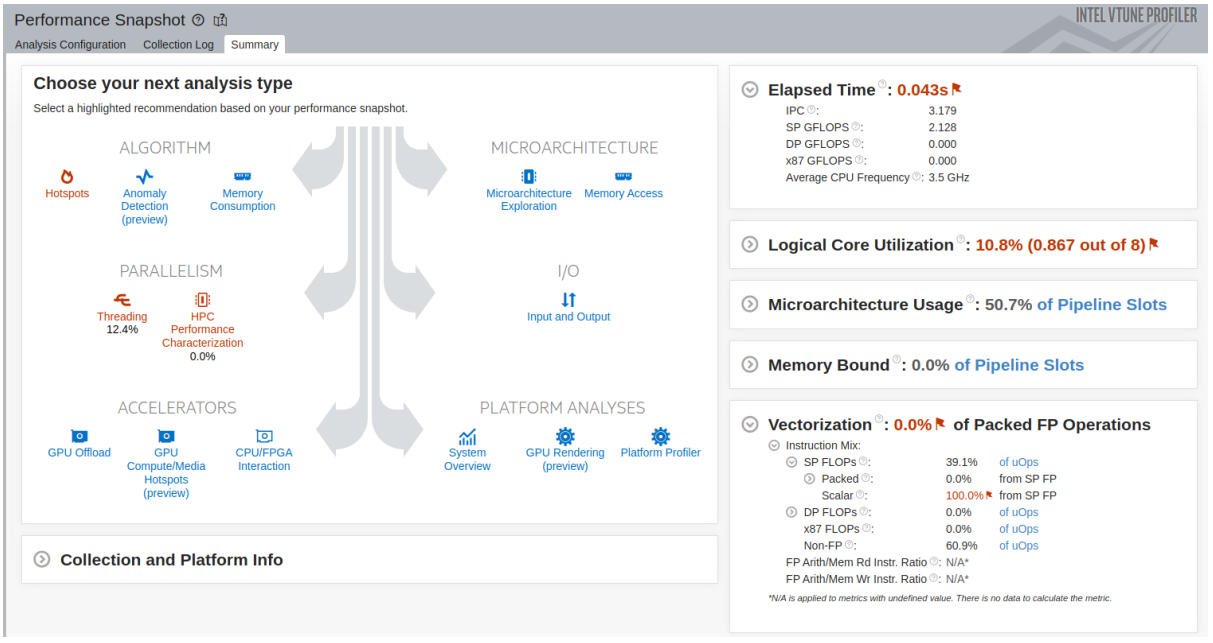
Άλλες χρήσιμες μετρικές που παρέχει το VTune είναι το **elapsed time**, δηλαδή ο συνολικός χρόνος εκτέλεσης της εφαρμογής και τα **GFLOPS**, δηλαδή πόσες δισεκατομμύρια πράξεις κινητής υποδιαστολής εκτελούνται κατά την διάρκεια του προγράμματος.

Στην παρούσα εργασία θα πραγματοποιήσουμε ένα performance snapshot για να πάρουμε κάποιες βασικές πληροφορίες για την

απόδοση του benchmark και στη συνέχεια θα κάνουμε μία ανάλυση για hotspots, ώστε να εντοπίσουμε την πιο χρονοβόρα συνάρτηση του προγράμματος. Αφού επιλέξουμε την υποψήφια προς βελτίωση συνάρτηση kernel του benchmark, στη συνέχεια θα μεταφερθούμε στο περιβάλλον του Vitis HLS, στο οποίο θα κάνουμε κάποιες τροποποιήσεις στον κώδικα της συνάρτησης, με σκοπό τη βελτίωση της απόδοσης της. Οι μετρήσεις του VTune profiler πραγματοποιούνται σε λειτουργικό σύστημα Ubuntu.

3.2 Performance Snapshot

Τα αποτελέσματα του performance snapshot για μέγεθος προβλήματος N = 512 παρουσιάζονται στην παρακάτω εικόνα:



Εικόνα 3.1: Αποτελέσματα performance snapshot για N = 512

Από την εικόνα 3.1 μπορούμε να παρατηρήσουμε ότι το VTune έχει χρωματίσει με κόκκινο χρώμα τα χαρακτηριστικά/αναλύσεις, τις οποίες πρέπει να δούμε για να βελτιώσουμε την απόδοση του προγράμματός μας.

3.3 Hotspots

Στη συνέχεια, θα πραγματοποιήσουμε μία ανάλυση hotspots για να εντοπίσουμε την πιο χρονοβόρα συνάρτηση του benchmark και η οποία θα είναι η υποψήφια προς βελτίωση συνάρτηση kernel. Επιλέγουμε μέγεθος προβλήματος N=512, “Hardware Event-Based sampling” με διάστημα δειγματοληψίας 0.01msec, το οποίο είναι και το μικρότερο δυνατό και θα μας δώσει πιο ακριβή αποτελέσματα. Παρακάτω παρουσιάζονται τα αποτελέσματα της ανάλυσης hotspots:

Function	CPU time	Instructions retired	Source file
sgemm	39,850msec	346.200.000	blas.c
strsm	2,707msec	16.000.000	blas.c
convert_double_ to_float	1,923msec	2.300.000	convert.c
matgen	1,104msec	1.800.000	matgen.c
lcg_rand_double	0,890msec	2.900.000	matgen.c
sgetrf2_nopiv	0,142msec	1.000.000	sgetrf_nopiv.c

Πίνακας 3.1: Αποτελέσματα ανάλυσης hotspots για N = 512

Πρέπει να σημειωθεί ότι τα αποτελέσματα των performance snapshot και hotspots διαφέρουν για κάθε νέα ανάλυση που πραγματοποιείται. Σε γενικές γραμμές, όμως, τα αποτελέσματα έχουν πολύ μικρή απόκλιση μεταξύ τους, επομένως μπορούν να θεωρηθούν αξιόπιστα.

3.4 Υποψήφια συνάρτηση kernel

Από τα παραπάνω αποτελέσματα είναι εμφανές ότι η πιο χρονοβόρα συνάρτηση του benchmark είναι η sgemm του αρχείου blas.c. Αναλύοντας τον κώδικα της συνάρτησης, παρατηρούμε ότι εκτελούνται οι παρακάτω πράξεις μεταξύ πινάκων και σταθερών όρων:

$$matrix(C) = beta \times matrix(C)$$
$$matrix(C) = matrix(C) + alpha \times matrix(B) \times matrix(A)$$

όπου:

- A, B, C είναι πίνακες διαστάσεων $k \times m$, $n \times k$ και $n \times m$ αντίστοιχα.
- alpha και beta είναι σταθερές.

Επίσης, οι παράμετροι transX, όπου το X συμβολίζει τον αντίστοιχο πίνακα και παίρνει τιμές X = a, b, δηλώνουν ότι για transX = “N” θα χρησιμοποιηθεί ο αντίστοιχος κανονικός πίνακας ενώ για transX = “T” θα χρησιμοποιηθεί ο ανάστροφος πίνακας, X^T. Τα m, n, k συμβολίζουν τις διαστάσεις των πινάκων ενώ τα lda, ldb, ldc συμβολίζουν τις βασικές διαστάσεις των πινάκων.

Κεφάλαιο 4: Vitis HLS

4.1 Εισαγωγή στο Vitis HLS

Το Vitis HLS της Xilinx, είναι ένα εργαλείο σχεδίασης επιταχυντών υλικού (accelerators) με χρήση μεθόδων σύνθεσης υψηλού επιπέδου (High Level Synthesis - HLS). Η σχεδίαση με μεθόδους HLS αφορά τη χρήση κάποιας γλώσσας προγραμματισμού υψηλού επιπέδου, όπως C/C++, η οποία στη συνέχεια μετατρέπεται σε μία υλοποίηση RTL. Μερικά από τα πλεονεκτήματα χρήσης των μεθόδων σχεδίασης HLS είναι η σημαντική μείωση του χρόνου ανάπτυξης της εφαρμογής, η καλή ποιότητα της παραγόμενης σχεδίασης και η πολύ μεγάλη ευκολία τροποποίησης της αρχικής σχεδίασης για προσαρμογή στις νέες απαιτήσεις.

Το Vitis HLS αποτελείται από 2 αρχεία πηγαίου κώδικα:

1. **Testbench:** Στο αρχείο testbench γίνεται προσομοίωση - compile σε C του κώδικα της συνάρτησης kernel πριν το στάδιο της σύνθεσης, προκειμένου να επαληθευτεί η σωστή λειτουργία της συνάρτησης.
2. **Kernel:** Στο αρχείο kernel βρίσκεται η υλοποίηση της συνάρτησης που θέλουμε να επιταχύνουμε. Η συνάρτηση kernel παίρνει, ως είσοδο, τα δεδομένα της από το testbench και επιστρέφει τα αποτελέσματα πίσω στο testbench για να γίνει η επαλήθευση.

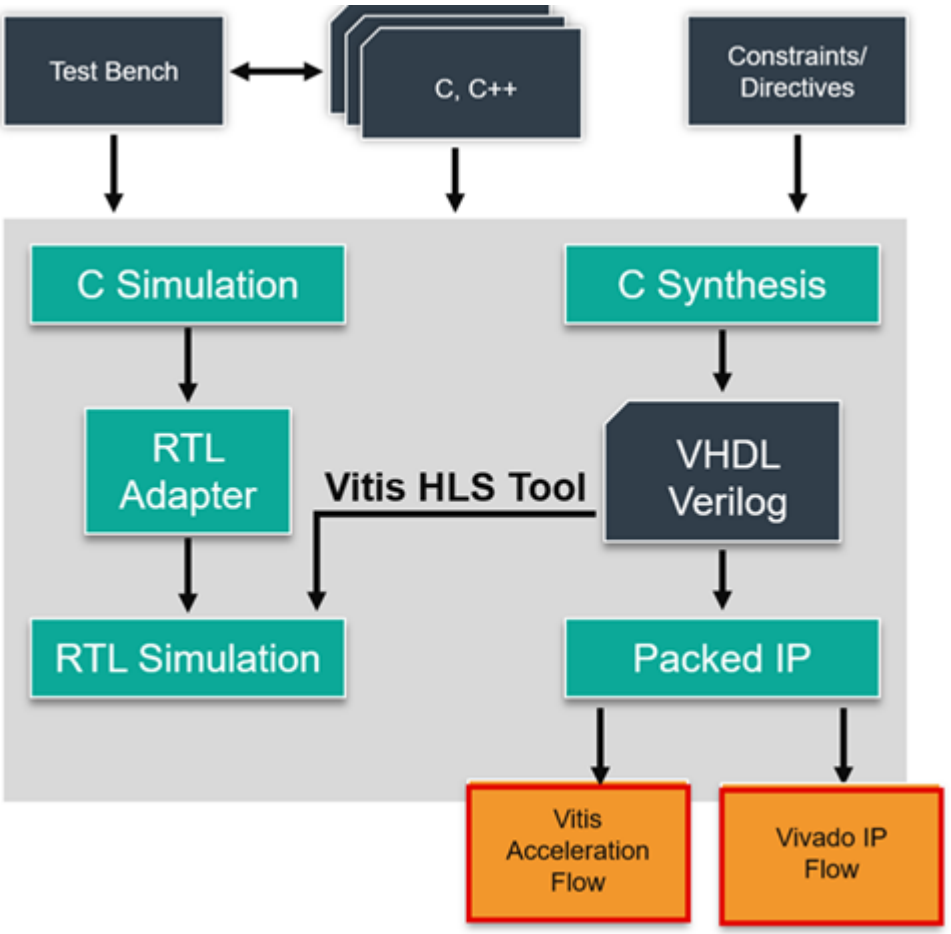
Το Vitis HLS περιλαμβάνει τα παρακάτω 3 στάδια:

- **C simulation:** Σε αυτό το στάδιο, χρησιμοποιείται ο compiler της C για τον έλεγχο της ορθής λειτουργίας της συνάρτησης kernel που θέλουμε να επιταχύνουμε. Υποστηρίζονται όλες οι δομές και όλοι οι τύποι δεδομένων, όπως integer, float, double και άλλοι, και παρέχονται βιβλιοθήκες για C based γλώσσες προγραμματισμού, κατάλληλα προσαρμοσμένες για τη λειτουργία σε FPGA.

- **C synthesis:** Μετά την επιτυχή ολοκλήρωση της προσομοίωσης σε C, περνάμε στο στάδιο της σύνθεσης, στο οποίο ο επιταχυντής μετατρέπει τον κώδικα υψηλού επιπέδου, που είναι σε C, σε περιγραφή χαμηλού επιπέδου RTL και, συγκεκριμένα, σε γλώσσες Verilog, VHDL. Επειδή το FPGA έχει προκαθορισμένο αριθμό πόρων, στο στάδιο αυτό δεν υποστηρίζονται ορισμένες δομές δεδομένων, όπως για παράδειγμα η δυναμική δέσμευση μνήμης και οι αναδρομικές συναρτήσεις. Στο τέλος της σύνθεσης, το εργαλείο παρουσιάζει μία λεπτομερή αναφορά με τα αποτελέσματα της εκτέλεσης του κώδικα, σχετικά με τη συνολική κατανάλωση πόρων (BRAMs, DSPs, FFs, LUTs) και την καθυστέρηση (latency) σε κύκλους.

- **C/RTL co-simulation:** Αποτελεί το τελικό στάδιο στο οποίο γίνεται επαλήθευση του παραγόμενου RTL κώδικα, από τη διαδικασία της σύνθεσης, με το testbench και επιστρέφεται στην main() η τιμή “0” εάν η προσομοίωση είναι επιτυχής. Σε αντίθετη περίπτωση, επιστρέφεται στη main() μία τιμή διάφορη του μηδενός.

Μετά το πέρας όλων των σταδίων μπορούμε να εξάγουμε τα αρχεία της υλοποίησης RTL σε γλώσσα περιγραφής υλικού και τα αποτελέσματα της σύνθεσης, του C/RTL co-simulation και του IP Packaging, το οποίο μπορεί να χρησιμοποιηθεί σε άλλες πλατφόρμες της Xilinx.



Εικόνα 4.1: Ροή σχεδίασης με το εργαλείο Vitis HLS

4.2 Alveo U200 Data Center Accelerator Card

Οι Alveo U200 Data Center Accelerator cards είναι συσκευές που έχουν σχεδιαστεί για να ανταποκρίνονται στις συνεχώς μεταβαλλόμενες ανάγκες του σύγχρονου Data Center, παρέχοντας έως και 90 φορές υψηλότερη απόδοση από τις CPU για βασικούς φόρτους εργασίας, συμπεριλαμβανομένων της μηχανικής μάθησης, διακωδικοποίησης βίντεο και αναζήτησης και ανάλυσης βάσεων δεδομένων. Σχεδιασμένες με αρχιτεκτονική AMD 16nm UltraScale, οι κάρτες επιτάχυνσης Alveo προσαρμόζονται στις μεταβαλλόμενες απαιτήσεις επιτάχυνσης και τα πρότυπα αλγορίθμων, είναι ικανές να επιταχύνουν οποιοδήποτε φόρτο εργασίας, χωρίς αλλαγή υλικού και να μειώσουν έτσι το συνολικό κόστος παραγωγής. Στους παρακάτω πίνακες παρουσιάζονται τα χαρακτηριστικά της κάρτας Alveo U200, καθώς και μία σύγκριση της επιτάχυνσης που μπορεί η κάρτα να επιτύχει έναντι ορισμένων CPUs.

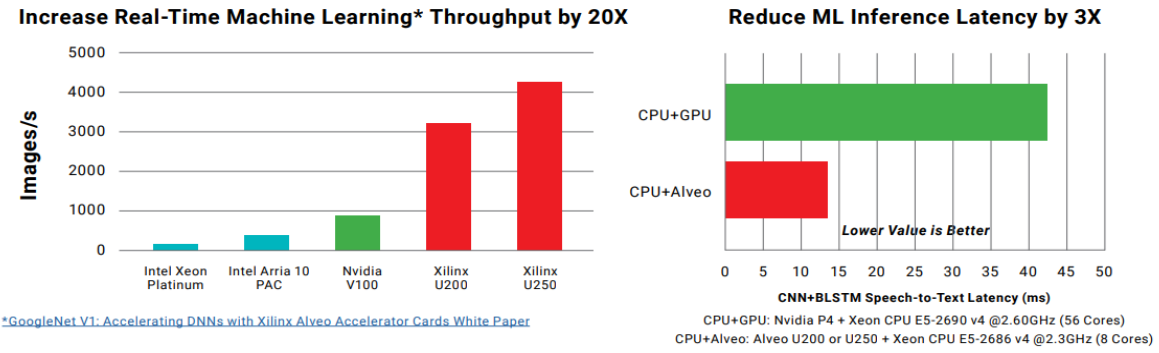
Εικόνα 4.2: Κάρτα Alveo U200 Data Accelerator με Active thermal cooling

Card Specifications	U200	
	A-U200-P64G-PQ-G	A-U200-A64G-PQ-G
Compute		
INT8 TOPs (peak)	18.6	18.6
Dimensions		
Height	Full	Full
Length	¾	Full
Width	Dual Slot	Dual Slot
Memory		
Off-chip Memory Capacity	64 GB	64 GB
Off-chip Total Bandwidth	77 GB/s	77 GB/s
Internal SRAM Capacity	35 MB	35 MB
Internal SRAM Total Bandwidth	31 TB/s	31 TB/s
Interfaces		
PCI Express	Gen3x16	Gen3x16
Network Interfaces	2x QSFP28 (100GbE)	2x QSFP28 (100GbE)
Logic Resources		
Look-up Tables (LUTs)	892,000	892,000
Power and Thermal		
Maximum Total Power	225W	225W
Thermal Cooling	Passive	Active

Εικόνα 4.3: Προδιαγραφές Alveo U200

AREA	PARTNER WORKLOAD	ALVEO ACCELERATION VS CPU
Database Search and Analytics	BlackLynx Unstructured Data Elasticsearch	90X
Financial Computing	Maxeler Value-at-Risk (VAR) Calculation	89X
Machine Learning	Xilinx Real-Time Machine Learning Inference	20X
Video Processing / Transcoding	NGCodec HEVC Video Encoding	12X
Genomics	Falcon Computing Genome Sequencing	10X

*CPU Comparisons: Xeon c4.8xlarge AWS | Xeon E5-2643 v4 3.4GHz | Xeon Platinum c5.18xlarge AWS | Dual Socket E5-2680 v3 2.5GHz | Xeon f1.16xlarge



Εικόνα 4.4: Σύγκριση επιτάχυνσης μεταξύ των συσκευών Alveo και διαφόρων CPU και GPU

4.3 Δημιουργία project στο Vitis HLS

Στο Vitis HLS δημιουργούμε ένα νέο project, επιλέγουμε την κάρτα Alveo U200 και στις ρυθμίσεις, ορίζουμε την sgemm ως την “Top-function” συνάρτηση πάνω στην οποία θα γίνει η προσομοίωση και η σύνθεση του κώδικα. Το project μας αποτελείται από 2 αρχεία πηγαίου κώδικα: το testbench και το kernel.

4.3.1 Κώδικας αρχείου testbench

Στο αρχείο testbench τροφοδοτούμε τη συνάρτηση - kernel με τυχαίες τιμές, την καλούμε με ορίσματα τους πίνακες A, B, C, τις διαστάσεις τους M, N, K και τη σταθερά alpha και στο τέλος επαληθεύουμε τη σωστή λειτουργία της μέσω της συνάρτησης test_kernel. Σε αυτό το σημείο πρέπει να τονίσουμε ότι η sgemm καλείται πολλές φορές στο αρχείο **sgetrf_nopiv.c** του benchmark από τις συναρτήσεις sgetrf_nopiv και sgetrf2_nopiv. Ωστόσο, εμάς μας ενδιαφέρει, στο στάδιο της σύνθεσης, να μετρήσουμε την απόδοση της συνάρτησης sgemm για μία μόνο κλήση της. Επομένως, έχοντας "απομονώσει" τη συνάρτηση sgemm από το υπόλοιπο benchmark, μετά το πέρας της εκτέλεσης της, καλούμε τη συνάρτηση test_kernel, η οποία ελέγχει ένα προς ένα όλα τα στοιχεία του παραγόμενου πίνακα από τον kernel (hardware) με τα στοιχεία του πίνακα που παράγονται στο testbench (software). Αν είναι ίδια τότε η προσομοίωση ήταν επιτυχής.

4.3.2 Κώδικας αρχείου kernel

Στο δεύτερο αρχείο περνάμε τη συνάρτηση kernel, έχοντας κάνει τις απαραίτητες τροποποιήσεις, ώστε ο κώδικας να γίνει πιο φιλικός ως προς το υλικό και αποδοτικός, χωρίς να αλλάξει η λειτουργικότητά του. Αρχικά, ορίζουμε τις απαραίτητες βιβλιοθήκες, δύο συναρτήσεις μετατροπής δεδομένων από float σε unsigned integer και το αντίστροφο

και ορισμένους τύπους μεταβλητών που θα χρησιμοποιήσουμε. Η βασική ιδέα της υλοποίησης μας είναι ότι εκμεταλλευόμαστε τη δυνατότητα που μας παρέχει η κάρτα Alveo για μεταφορά στοιχείων έως και 512 bits από και προς τον kernel. Για το λόγο αυτό ορίζουμε έναν νέο τύπο μεταβλητής “uint512_dt”, ο οποίος περιέχει συνολικά 16 unsigned integer στοιχεία των 32 bits, δηλαδή 16 elements × 32 bits = 512 bits. Επομένως, ορίζουμε στις παραμέτρους της συνάρτησης kernel και τους 3 μονοδιάστατους πίνακες (2 input και 1 output) ως uint512_dt. Για να γίνει η μεταφορά των δεδομένων από το testbench στον kernel και αντιστρόφως, πραγματοποιούμε μία μετατροπή των στοιχείων των πινάκων με τη χρήση των δύο προαναφερθέντων, βοηθητικών συναρτήσεων και της συνάρτησης range(), η οποία παίρνει ως παραμέτρους το τελικό και αρχικό index των bits της εκάστοτε uint512_dt μεταβλητής.

Περνώντας στην υποψήφια συνάρτηση kernel, χρησιμοποιούμε ορισμένα **directives** και **πρωτόκολλα διεπαφής** για να βελτιώσουμε την απόδοσή της. Σε αντίθεση με μία συνάρτηση σε C, στην οποία η επεξεργασία των πράξεων εισόδου και εξόδου γίνεται μέσω των παραμέτρων της και σε μηδενικό χρόνο, σε ένα σχέδιο RTL για την ίδια διαδικασία είναι απαραίτητος ο καθορισμός ενός πρωτοκόλλου διεπαφής για είσοδο και έξοδο. Τα πρωτόκολλα διεπαφής χωρίζονται σε τρεις κατηγορίες:

1. **Block-level:** Κάνουν έλεγχο για το πότε το block μπορεί να δεχθεί νέες εισόδους, πότε να ξεκινήσει την επεξεργασία των δεδομένων, αν το σχέδιο είναι σε αδρανή κατάσταση ή έχει ολοκληρωθεί η λειτουργία του.
2. **Port-level:** Είναι υπεύθυνα για τη σειριακή μεταφορά των δεδομένων στις θύρες εισόδου και εξόδου. Τα πρωτόκολλα αυτά δημιουργούνται για κάθε παράμετρο της συνάρτησης kernel, που έχουμε ορίσει ως top function και εξαρτώνται από τον τύπο των ορισμάτων της συνάρτησης. Επίσης, έχουμε port-level πρωτόκολλα και για τη συνάρτηση return, αν υπάρχει επιστροφή κάποιας τιμής.
3. **Θύρα ρολογιού**(clock) και **επαναφοράς**(reset)**.**

Στον kernel μας κάνουμε χρήση των πρωτοκόλλων διεπαφής **AXI** (Advanced eXtensible Interface), τα οποία είναι μία κατηγορία on-chip πρωτοκόλλων διαύλου (bus) επικοινωνίας και διαχείρισης λειτουργικών μπλοκ κατά το σχεδιασμό SoC και αποτελούν μέρος-εξέλιξη της προδιαγραφής Advanced Microcontroller Bus Architecture (AMBA), ορισμένα από την ARM. Η αρχιτεκτονική του AXI είναι παρόμοιας φιλοσοφίας με αυτή του **AHB** (Advanced High Performance Bus).

Το AHB είναι ένας δίαυλος μονού καναλιού που χρησιμοποιούν πολλαπλοί “αφέντες” (masters) και “σκλάβοι” (slaves) για την ανταλλαγή δεδομένων. Ένας διαιτητής προτεραιότητας (priority arbiter) καθορίζει ποιος master μπορεί να χρησιμοποιήσει αυτήν τη στιγμή το δίαυλο, ενώ ένας κεντρικός αποκωδικοποιητής κάνει επιλογή του slave. Οι λειτουργίες εκτελούνται σε ριπές (bursts), που μπορεί να χρειαστούν πολλαπλοί κύκλοι διαύλου (bus cycles) για να ολοκληρωθούν. Κάθε burst μεταφορά δεδομένων αποτελείται από μια φάση διεύθυνσης και ελέγχου, που ακολουθείται από μια φάση δεδομένων.

Το AXI χρησιμοποιεί πολλαπλά και αποκλειστικά κανάλια για ανάγνωση και εγγραφή δεδομένων, βασίζεται σε burst, όπως και ο προκάτοχός του, και χρησιμοποιεί παρόμοια φάση διεύθυνσης και ελέγχου πριν από την ανταλλαγή δεδομένων. Το AXI περιλαμβάνει, επίσης, μια σειρά από νέες δυνατότητες, όπως συναλλαγές out-of-order, μη ευθυγραμμισμένες μεταφορές δεδομένων, σήματα υποστήριξης κρυφής μνήμης και μία χαμηλής κατανάλωσης διεπαφή. Ανάμεσα σε ένα master και slave του AXI υπάρχουν 5 ανεξάρτητα κανάλια και είναι τα εξής:

- Κανάλι ανάγνωσης διεύθυνσης (Read address channel).
- Κανάλι ανάγνωσης δεδομένων (Read Data channel).
- Κανάλι εγγραφής διεύθυνσης (Write address channel).
- Κανάλι εγγραφής δεδομένων (Write Data channel).
- Κανάλι εγγραφής απόκρισης (Write response channel).

Τα κανάλια διευθύνσεων χρησιμοποιούνται για την αποστολή πληροφοριών διεύθυνσης και ελέγχου και τα κανάλια δεδομένων είναι εκεί όπου τοποθετούνται οι πληροφορίες προς ανταλλαγή. Κάθε ανταλλαγή δεδομένων ονομάζεται συναλλαγή. Μια συναλλαγή

περιλαμβάνει τη διεύθυνση και τις πληροφορίες ελέγχου, τα δεδομένα που αποστέλλονται, καθώς και οποιαδήποτε πληροφορία απάντησης. Τα πραγματικά δεδομένα αποστέλλονται σε bursts που περιέχουν πολλαπλές μεταφορές. Ένας master διαβάζει δεδομένα από και γράφει δεδομένα σε ένα slave. Οι πληροφορίες απόκρισης ανάγνωσης τοποθετούνται στο κανάλι ανάγνωσης δεδομένων, ενώ οι πληροφορίες απόκρισης εγγραφής έχουν ένα αποκλειστικό κανάλι. Με αυτόν τον τρόπο ο master επαληθεύει ότι μια συναλλαγή εγγραφής έχει ολοκληρωθεί.

Εικόνα 4.5: Σύνδεση των διεπαφών master και slave μέσω των AXI καναλιών

Στη συνάρτηση kernel χρησιμοποιούμε τα πρωτόκολλα διεπαφής AXI4 master(m_axi) και AXI4-Lite(s_axilite). Το m_axi χρησιμοποιείται για ορίσματα που έχουν να κάνουν με πίνακες και pointers και είναι απαραίτητο για να ρυθμίσουμε το μέγεθός τους. Το s_axilite χρησιμοποιείται για όλα τα ορίσματα, συμπεριλαμβανομένων και των

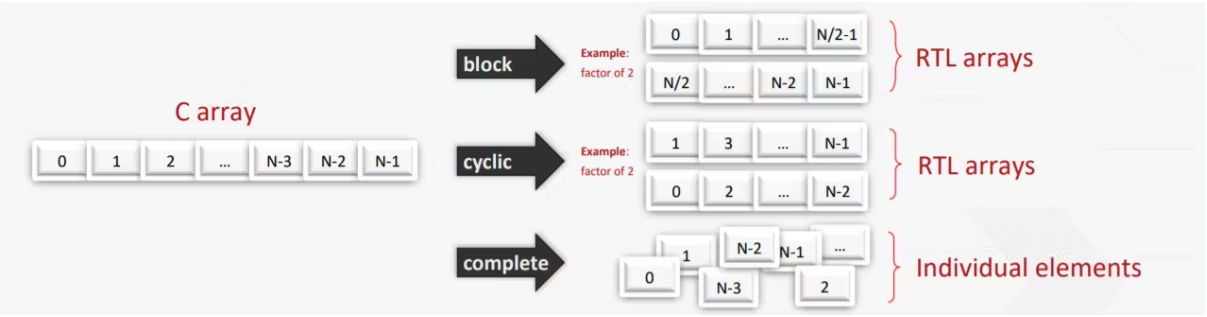
προαναφερθέντων πινάκων και pointers. Όλα τα ορίσματα AXI4-Lite χρησιμοποιούν το δίαυλο (bundle) “control”. Για τις παραμέτρους με το πρωτόκολλο διεπαφής m_axi:

- Ορίζουμε την επιλογή “offset=slave” για να κάνουμε την αντιστοίχιση με τα ορίσματα s_axilite.
- Ορίζουμε ως επιπρόσθετες διαύλους (bundles) των pointers “in1”, “in2” και “out” τους “gmem”, “gmem1” και “gmem2” αντίστοιχα για ταυτόχρονη ανάγνωση και εγγραφή δεδομένων.
- Χρησιμοποιούμε τις επιλογές “max_write_burst_length = 32” και “max_read_burst_length = 32”, για να ορίσουμε το μέγιστο αριθμό δεδομένων εγγραφής/ανάγνωσης σε μία burst συναλλαγή δεδομένων.

Τα directives που θα χρησιμοποιήσουμε για βελτίωση της απόδοσης της συνάρτησης sgemm και την παραλληλοποίηση των δεδομένων είναι οι εξής:

- **Array Partition:** Διαχωρίζει έναν πίνακα σε μικρότερους υπο-πίνακες ή μεμονωμένα στοιχεία και ως αποτέλεσμα μας παρέχει ένα σχέδιο RTL με πολλές μικρές μνήμες ή πολλαπλούς καταχωρητές αντί για μία μεγάλη μνήμη. Ακόμη, αυξάνει σημαντικά τον όγκο των θυρών ανάγνωσης και εγγραφής για την αποθήκευση των δεδομένων και βελτιώνει την απόδοση του σχεδιασμού. Τέλος, απαιτεί περισσότερες οντότητες μνήμης ή καταχωρητές. Υπάρχουν 3 είδη array partition:
    1. Block**:** Ο διαχωρισμός μπλοκ δημιουργεί μικρότερους πίνακες από διαδοχικά μπλοκ του αρχικού πίνακα.
    2. Cyclic**:** Ο κυκλικός διαχωρισμός δημιουργεί μικρότερους πίνακες παρεμβάλλοντας στοιχεία από τον αρχικό πίνακα. Ο πίνακας διαμερίζεται κυκλικά βάζοντας ένα στοιχείο σε κάθε νέο πίνακα πριν επιστρέψει στον πρώτο πίνακα για να επαναλάβει τον κύκλο, έως ότου ο πίνακας κατατμηθεί πλήρως.
    3. Complete**:** Ο πλήρης διαχωρισμός αποσυνθέτει τον πίνακα σε μεμονωμένα στοιχεία. Για ένα μονοδιάστατο πίνακα, αυτό

αντιστοιχεί στην ανάλυση μιας μνήμης σε μεμονωμένους καταχωρητές. Αυτός ο τύπος ισχύει από προεπιλογή.



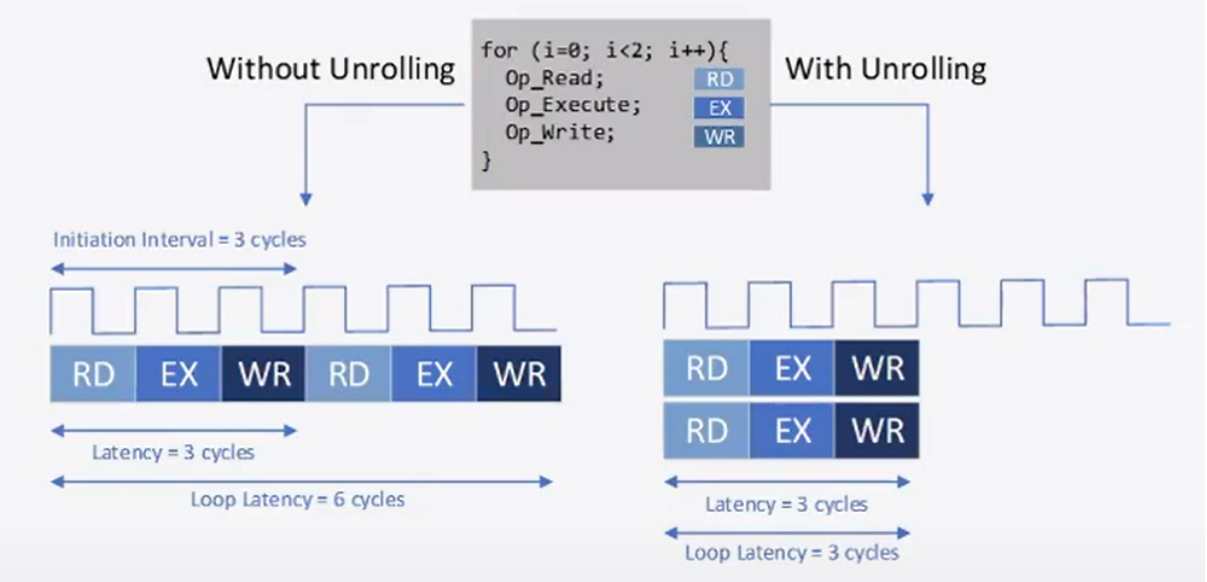
Εικόνα 4.6: Αναπαράσταση λειτουργίας των block, cyclic και complete array partitions

Στην περίπτωση των πολυδιάστατων πινάκων χρησιμοποιούμε την επιλογή “dim=<N>” , όπου N είναι η διάσταση που θέλουμε να κάνουμε partition. Για παράδειγμα, θέτοντας σε έναν δισδιάστατο πίνακα dim=2 σημαίνει ότι θα κάνουμε partition τη 2η διάσταση, ενώ αν θέσουμε dim=0 τότε όλες οι διαστάσεις του πίνακα γίνονται array partitioned. Επίσης στις περιπτώσεις των block και cyclic array partition είναι υποχρεωτικό να βάλουμε την επιλογή “factor=<N>”, όπου το N εκφράζει τον αριθμό των ίσων υπο-πινάκων που θα δημιουργηθούν από την κατάτμηση του αρχικού πίνακα. Στην περίπτωση του complete array partition, η επιλογή factor δεν χρησιμοποιείται.

- **Tripcount:** Χρησιμοποιείται για τον καθορισμό του ελαχίστου και μεγίστου αριθμού επαναλήψεων του βρόχου. Επειδή, όμως, στον kernel μας ο αριθμός των επαναλήψεων του κάθε βρόχου είναι μεταβλητός και όχι σταθερός, ορίζουμε μία ελάχιστη τιμή “1” και μία μέγιστη τιμή για την οποία θα τρέξει ο βρόχος. Πρέπει να τονίσουμε ότι το tripcount δεν επηρεάζει τα αποτελέσματα της σύνθεσης του κώδικα.

- **Unroll:** Με το loop unrolling (ξετύλιγμα βρόγχων) μπορούμε να δημιουργήσουμε πολλαπλές ανεξάρτητες λειτουργίες αντί για μια

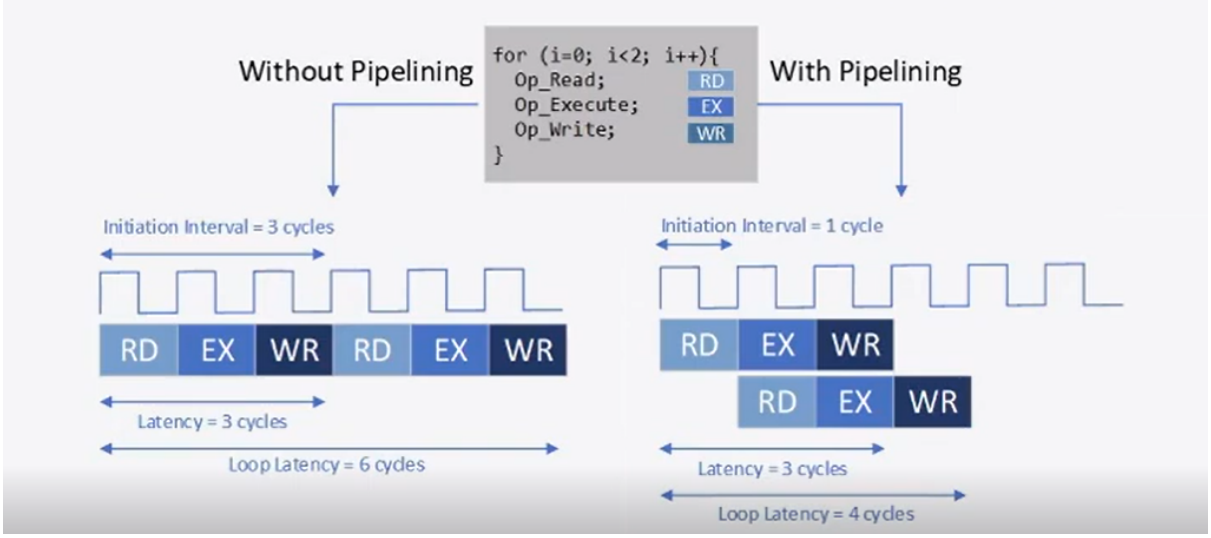
ενιαία συλλογή λειτουργιών. Το UNROLL pragma μετασχηματίζει τους βρόγχους, δημιουργώντας πολλαπλά αντίγραφα του σώματος του βρόγχου στο σχέδιο RTL, το οποίο επιτρέπει σε ορισμένες ή όλες τις επαναλήψεις βρόχων να συμβαίνουν παράλληλα, αυξάνοντας έτσι την προσβασιμότητα των δεδομένων και βελτιώνοντας την απόδοση. Υπάρχουν δύο είδη loop unrolling: ολικό (complete) και μερικό (partial). Στην πρώτη περίπτωση, δημιουργείται ένα αντίγραφο του σώματος του βρόχου στο RTL για κάθε επανάληψη βρόχου, έτσι ώστε ολόκληρος ο βρόχος να μπορεί να εκτελεστεί ταυτόχρονα. Στην δεύτερη περίπτωση, πρέπει να καθορίσουμε έναν παράγοντα (factor) N, για να δημιουργήσουμε N αντίγραφα του σώματος του βρόχου και, αναλόγως, να μειώσουμε τις επαναλήψεις του βρόχου.



Εικόνα 4.7: Αναπαράσταση λειτουργίας του UNROLL pragma

- **Pipeline:** Μειώνει το Initiation Interval (II) για μια συνάρτηση ή βρόχο επιτρέποντας την ταυτόχρονη εκτέλεση πράξεων. Μία pipelined συνάρτηση ή ένας pipelined βρόχος μπορούν να επεξεργάζονται νέες εισόδους για κάθε <N> κύκλους ρολογιού, όπου <N> είναι το II της συνάρτησης ή του βρόχου. Για II=1, επεξεργάζεται μια νέα είσοδο σε κάθε κύκλο ρολογιού. Μπορούμε να καθορίσουμε το Initiation Interval χρησιμοποιώντας την επιλογή

II για το pragma. Κατά την χρήση του pipeline pragma, από προεπιλογή, το Vitis HLS θα δημιουργήσει το ελάχιστο Initiation Interval (default II=1) για τη σχεδίαση σύμφωνα με τον καθορισμένο περιορισμό περιόδου ρολογιού. Εάν το εργαλείο Vitis HLS δεν μπορεί να δημιουργήσει ένα σχέδιο με το καθορισμένο II, εμφανίζει ένα warning στην κονσόλα και δημιουργεί ένα σχέδιο με το χαμηλότερο δυνατό II.



Εικόνα 4.8: Αναπαράσταση λειτουργίας του PIPELINE pragma

4.4 Σύνθεση του κώδικα και ανάλυση των αποτελεσμάτων

Αρχικά, θα πραγματοποιήσουμε μία σύνθεση του αρχικού κώδικα της συνάρτησης sgemm, στον οποίο οι μόνες αλλαγές που θα κάνουμε θα είναι να βάλουμε τα πρωτόκολλα διεπαφής για τη μεταφορά των δεδομένων από και προς στον kernel και να ορίσουμε τα όρια των επαναλήψεων κάθε βρόχου με την χρήση του TRIPCOUNT pragma. Στη συνέχεια, θα κάνουμε σύνθεση του επεξεργασμένου κώδικα της συνάρτησης sgemm με την χρήση όλων των directives και τέλος θα

συγκρίνουμε τα αποτελέσματα. Στόχος μας είναι να πετύχουμε την χρησιμοποίηση ορισμένων resources που μας παρέχει μία μονάδα υπολογισμού SLR της FPGA για να μειώσουμε το latency και να βελτιώσουμε την απόδοση της συνάρτησης, όπως είναι τα DSPs και τα BRAMs.

Τα **DSPs** είναι εξειδικευμένα μπλοκ υλικού σχεδιασμένα να εκτελούν αποτελεσματικά λειτουργίες επεξεργασίας ψηφιακού σήματος. Μπορούν να εκτελούν πολλαπλές μαθηματικές πράξεις παράλληλα, καθιστώντας τα κατάλληλα για εφαρμογές σε πεδία όπως η επεξεργασία εικόνας και σήματος.

Τα **BRAMs** ή Block RAMs είναι μικρά και γρήγορα μπλοκ μνήμης σε τσιπ, που μπορούν να χρησιμοποιηθούν για την αποθήκευση δεδομένων και την παροχή πρόσβασης υψηλής ταχύτητας σε αυτά τα δεδομένα. Μπορούν να χρησιμοποιηθούν για την αποθήκευση μεταβλητών, πινάκων αναζήτησης (LUTs) και άλλων δεδομένων που χρειάζονται γρήγορη και αποτελεσματική πρόσβαση ανάγνωσης/εγγραφής. Τα BRAMs μπορούν να διαμορφωθούν με διαφορετικούς τρόπους (single-port, dual-port ή true dual-port), ώστε να ταιριάζουν σε διάφορα μοτίβα πρόσβασης στη μνήμη.

4.5 Ανάλυση αποτελεσμάτων C synthesis

Για την αρχική συνάρτηση sgemm τα αποτελέσματα της σύνθεσης για μέγεθος προβλήματος N = 512 παρουσιάζονται στους παρακάτω πίνακες:

BRAM	DSP	FF	LUT	URAM
6	17	5936	7392	0

Πίνακας 4.1: Αποτελέσματα C synthesis της αρχικής συνάρτησης sgemm

BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
~0	~0	~0	~0	0

Πίνακας 4.2: Αποτελέσματα C synthesis με ποσοστά της αρχικής συνάρτησης sgemm

Όπως ήταν αναμενόμενο, η αρχική συνάρτηση sgemm δεν χρησιμοποιεί σχεδόν καθόλου DSPs και BRAMs και επίσης παίρνουμε αρκετά Memory Dependency Violations και warnings στην κονσόλα για τυχόν βελτιώσεις τις οποίες θα μπορούσαμε να κάνουμε για να επιταχύνουμε την εκτέλεσή της.

Στη συνέχεια περνάμε στα αποτελέσματα της τροποποιημένης συνάρτησης sgemm:

BRAM	DSP	FF	LUT	URAM
154	1284	160529	147940	0

Πίνακας 4.3: Αποτελέσματα C synthesis της τροποποιημένης συνάρτησης sgemm

BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
3	18	6	12	0

Πίνακας 4.4: Αποτελέσματα C synthesis με ποσοστά της τροποποιημένης συνάρτησης sgemm

Παρατηρούμε ότι στην συγκεκριμένη υλοποίηση πετυχαίνουμε υψηλότερο ποσοστό DSP, που σημαίνει ότι θα έχουμε καλύτερη παραλληλοποίηση δεδομένων όταν θα τρέξουμε τον κώδικα στο Hardware στην FPGA. Δυστυχώς, δεν καταφέραμε να πετύχουμε εξίσου υψηλό ποσοστό BRAM, αλλά δεν μας πειράζει τόσο, καθώς ο κύριος στόχος ήταν να πετύχουμε όσο το δυνατόν υψηλότερο DSP utilization.

Γενικά, η υλοποίηση της τροποποιημένης συνάρτησης sgemm είναι τέτοια, ώστε να μπορούμε να αυξομειώνουμε τον αριθμό των πόρων που χρησιμοποιούνται, αλλάζοντας απλά τις τιμές των μεταβλητών “BUFFER_K”, “BUFFER_M”, “UNROLL_K” και “UNROLL_M”, έτσι ώστε να πετύχουμε τον καταλληλότερο για εμάς συνδυασμό. Θεωρητικά θα μπορούσαμε να πετύχουμε μέχρι και 75% DSP utilization, μεταβάλλοντας τις παραπάνω τιμές, ωστόσο δεν θα ήταν εφικτό να βάλουμε τον kernel να τρέξει σε hardware, καθώς θα ξεπερνούσε τους διαθέσιμους πόρους της μονάδας υπολογισμού SLR. Επειδή η FPGA έχει 3 μονάδες υπολογισμού και εμείς χρησιμοποιούμε μόνο τη μία, αναγκαστικά οι διαθέσιμοι πόροι είναι περιορισμένοι.

4.6 Συμπέρασμα

Σε αυτό το κεφάλαιο, δημιουργήσαμε ένα project στο Vitis HLS, ένα High Level Synthesis εργαλείο της Xilinx, προκειμένου να βελτιώσουμε την απόδοση της υποψήφιας συνάρτησης sgemm, την οποία επιλέξαμε ως συνάρτηση kernel από το κεφάλαιο 3. Στη συνέχεια, κάναμε μία παρουσίαση των χαρακτηριστικών της κάρτας Alveo U200 και αναλύσαμε τον κώδικα και τα directives που χρησιμοποιήσαμε για να βελτιώσουμε την απόδοση του kernel. Τέλος, πήραμε τα αποτελέσματα της σύνθεσης του κώδικα για την αρχική και τροποποιημένη συνάρτηση sgemm και καταλήξαμε στο συμπέρασμα ότι η τροποποιημένη συνάρτηση sgemm είναι πιο φιλική προς το υλικό και κάνει μεγαλύτερη χρήση των διαθέσιμων πόρων της FPGA, κάτι που θα οδηγήσει σε μεγαλύτερη επιτάχυνση και βελτίωση της απόδοσης του προγράμματος στο επόμενο στάδιο της εισαγωγής του κώδικα στο hardware της FPGA.

Κεφάλαιο 5: Vitis

5.1 Εισαγωγή στο Vitis

Το Vitis είναι μια πλατφόρμα λογισμικού ανεπτυγμένη από την Xilinx, η οποία έχει σχεδιαστεί για να απλοποιεί και να επιταχύνει την ανάπτυξη εφαρμογών που εκτελούνται σε συσκευές FPGA και SoC της Xilinx. Στόχος του Vitis είναι να κάνει την ανάπτυξη FPGA και SoC πιο προσιτή στους προγραμματιστές λογισμικού και να επιταχύνει το σχεδιασμό και την ανάπτυξη εφαρμογών που μπορούν να επωφεληθούν από την επιτάχυνση υλικού. Είναι ένα πολύ χρήσιμο εργαλείο για όσους εργάζονται σε εφαρμογές πάνω σε τομείς, όπως data acceleration centers, τεχνητή νοημοσύνη, υπολογισμούς υψηλής απόδοσης και πολλά άλλα. Το Vitis προσφέρει μια ενοποιημένη πλατφόρμα λογισμικού που περιλαμβάνει διάφορα εργαλεία ανάπτυξης και βιβλιοθήκες για να βοηθήσει τους προγραμματιστές να εργαστούν αποτελεσματικά με το υλικό Xilinx. Το ενοποιημένο περιβάλλον Vitis περιλαμβάνει τα ακόλουθα:

1. **Vitis Core Development Kit (CDK):** Ένα πλήρες σύνολο αναπτυξιακών εργαλείων γραφικών και γραμμής εντολών που περιλαμβάνει μεταγλωττιστές (compilers), αναλυτές (analyzers) και προγράμματα εντοπισμού σφαλμάτων (debuggers) Vitis για τη δημιουργία εφαρμογών, την ανάλυση σημείων συμφόρησης απόδοσης (bottlenecks) και τον εντοπισμό σφαλμάτων επιταχυνόμενων αλγορίθμων, που έχουν αναπτυχθεί σε C, C++ ή OpenCL APIs.

2. **Vitis Accelerated Libraries:** Βιβλιοθήκες ανοιχτού κώδικα, βελτιστοποιημένες ως προς την απόδοση, που προσφέρουν επιτάχυνση με ελάχιστες έως μηδενικές αλλαγές κώδικα στις υπάρχουσες εφαρμογές και είναι γραμμένες σε C, C++ ή Python.

3. **Vitis AI Development Environment:** Το περιβάλλον ανάπτυξης Vitis AI είναι ένα εξειδικευμένο περιβάλλον ανάπτυξης για την επιτάχυνση των συμπερασμάτων AI σε ενσωματωμένες πλατφόρμες AMD, κάρτες επιταχυντών Alveo ή σε περιπτώσεις FPGA στο cloud. Ακόμα, υποστηρίζει τα κορυφαία deep learning frameworks, όπως το Tensorflow και το Caffe, και προσφέρει ολοκληρωμένα API για κβαντοποίηση, βελτιστοποίηση και μεταγλώττιση των εκπαιδευμένων δικτύων και μας βοηθάει να πετύχουμε την υψηλότερη απόδοση συμπερασμάτων AI για την εφαρμογή μας.

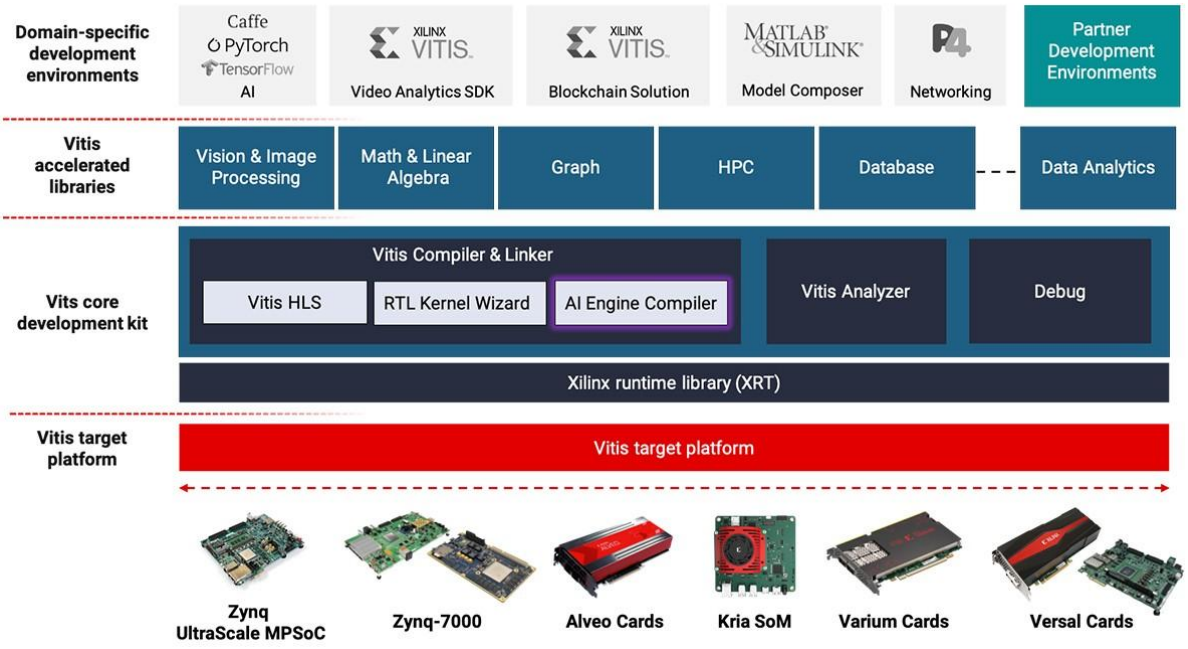
4. **Xilinx Runtime Library (XRT):** Είναι υπεύθυνη για να διευκολύνει την επικοινωνία μεταξύ του κώδικα της εφαρμογής, που εκτελείται σε ενσωματωμένο βραχίονα ή x86 host, και των επιταχυντών που αναπτύσσονται στο αναδιαμορφώσιμο τμήμα των καρτών επιτάχυνσης AMD, οι οποίες βασίζονται σε διασύνδεση PCIe, ενσωματωμένες πλατφόρμες που βασίζονται σε MPSoC ή προσαρμοστικές πλατφόρμες SoC . Περιλαμβάνει user-space βιβλιοθήκες και APIs, kernel drivers, βοηθητικά προγράμματα πλακέτας και υλικολογισμικό.

5. **Vitis Target Platforms:** Ορίζει την αρχιτεκτονική βάσης υλικού και λογισμικού καθώς και το πλαίσιο εφαρμογής για πλατφόρμες AMD, συμπεριλαμβανομένων των διεπαφών εξωτερικής μνήμης, προσαρμοσμένων διεπαφών εισόδου/εξόδου και χρόνου εκτέλεσης λογισμικού.

    - Για κάρτες επιτάχυνσης AMD σε εγκαταστάσεις ή στο cloud, η πλατφόρμα στόχου Vitis διαμορφώνει αυτόματα τις διεπαφές PCIe που συνδέουν και διαχειρίζονται την επικοινωνία μεταξύ των επιταχυντών FPGA και του κώδικα εφαρμογής x86.

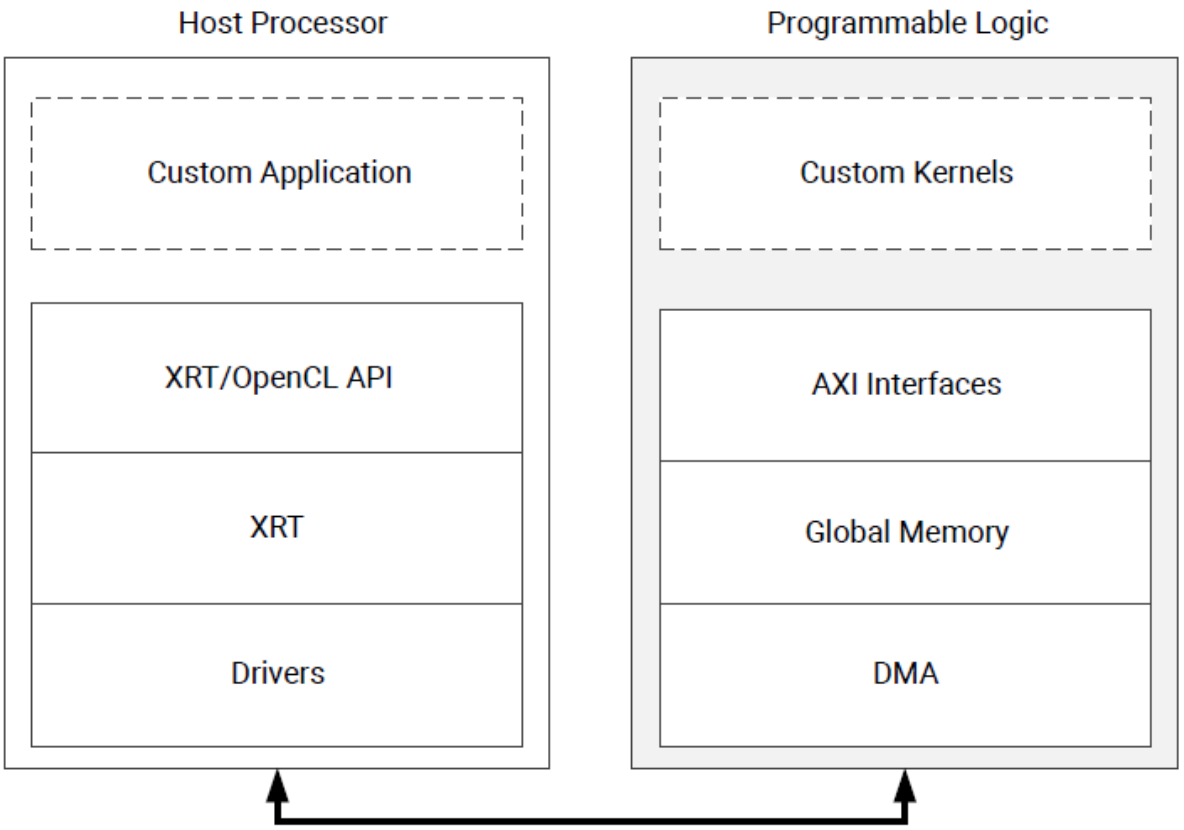
    - Για τις ενσωματωμένες συσκευές AMD, η πλατφόρμα στόχου Vitis περιλαμβάνει, επίσης, το λειτουργικό σύστημα για τον επεξεργαστή στην πλατφόρμα, τον bootloader και τους

drivers για τα περιφερειακά της πλατφόρμας και το σύστημα αρχείων root.



Εικόνα 5.1: Επισκόπηση της ενοποιημένης πλατφόρμας λογισμικού Vitis

Περνώντας στο Vitis Core Development Kit, ο κώδικας του host, δηλαδή ο κώδικας του λογισμικού, αναπτύσσεται σε έναν ενσωματωμένο επεξεργαστή ×86 και είναι γραμμένος σε γλώσσα C/C++, ενώ ο κώδικας του kernel, δηλαδή το κομμάτι του υλικού είναι γραμμένο σε γλώσσα C/C++, OpenCL και RTL. Η μεταφορά των δεδομένων εισόδου/εξόδου και γενικά η επικοινωνία μεταξύ του host και του kernel εξασφαλίζεται με την χρήση είτε ενός διαύλου PCIe είτε ενός διαύλου AXI. Τόσο ο host όσο και ο kernel έχουν πρόσβαση στα δεδομένα μέσω της καθολικής μνήμης (global memory), ενώ η μνήμη του host (RAM) είναι προσβάσιμη μόνο από τη CPU. Η πλατφόρμα στόχου Vitis αποτελείται από τους kernels, την καθολική μνήμη και την άμεσα προσβάσιμη μνήμη (Direct Memory Access ή DMA) για τη μεταφορά των δεδομένων. Γενικά, οι μεταφορές των δεδομένων από τον host στην καθολική μνήμη δημιουργούν μία καθυστέρηση, η οποία επηρεάζει την εφαρμογή μας.



Εικόνα 5.2: Επικοινωνία μεταξύ του host και των kernels

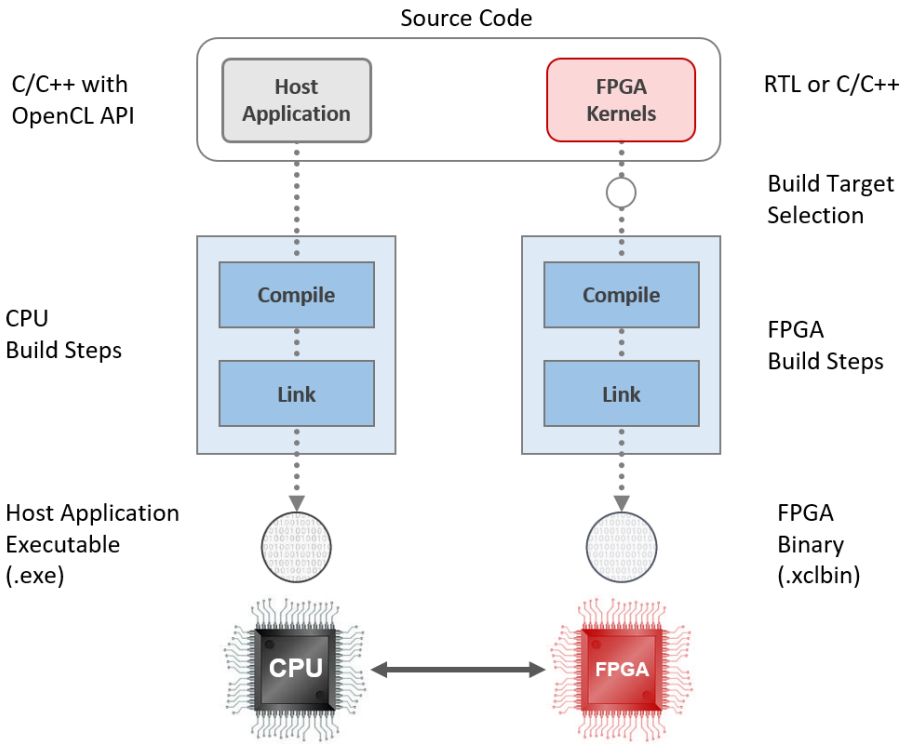
Το μοντέλο της ροής εκτέλεσης του Vitis μπορεί να αναλυθεί στα εξής στάδια:

1. Ο κώδικας του host που εκτελείται στη CPU γράφει τα δεδομένα που απαιτούνται από έναν kernel στην καθολική μνήμη της συνδεδεμένης συσκευής μέσω της διεπαφής PCIe σε μια κάρτα επιταχυντή Alveo (Data Center) ή μέσω του διαύλου AXI σε μια ενσωματωμένη πλατφόρμα.

2. Το τμήμα του κώδικα που εκτελείται στη πλευρά του host ρυθμίζει τον kernel με τις παραμέτρους εισόδου του και ενεργοποιεί την εκτέλεση της λειτουργίας kernel στην FPGA.

3. Ο kernel που εκτελείται στην FPGA εκτελεί τον απαιτούμενο υπολογισμό ενώ διαβάζει δεδομένα από την καθολική μνήμη.

4. Στη συνέχεια γράφει τα δεδομένα πίσω στην καθολική μνήμη και ειδοποιεί τη CPU ότι έχει ολοκληρώσει την εργασία του. Το πρόγραμμα της CPU μεταφέρει τα δεδομένα από την καθολική μνήμη στη μνήμη RAM της CPU και συνεχίζει την επεξεργασία.



Εικόνα 5.3: Ροή σχεδίασης στο Vitis

Ο compiler του Vitis μας παρέχει τρεις διαφορετικές λειτουργίες, δύο προσομοίωσης που χρησιμοποιούνται για τον εντοπισμό σφαλμάτων και επικύρωσης και μια που χρησιμοποιείται για τη δημιουργία του πραγματικού πυρήνα FPGA:

- **Software Emulation (sw_emu):** Τόσο ο κώδικας εφαρμογής που προορίζεται για τη CPU όσο και ο κώδικας του kernel της FPGA γίνονται compile στη CPU. Η λειτουργία αυτή είναι χρήσιμη για τον εντοπισμό σφαλμάτων σύνταξης (syntax error), τον εντοπισμό σφαλμάτων του κώδικα του kernel που εκτελείται μαζί με την

εφαρμογή και την επαλήθευση της συμπεριφοράς του συστήματος και ολόκληρης της εφαρμογής.

- **Hardware Emulation (hw_emu):** Ο κώδικας του kernel μετατρέπεται σε ένα μοντέλο υλικού (RTL), το οποίο εκτελείται σε έναν ειδικό προσομοιωτή. Αυτή η διαδικασία απαιτεί περισσότερο χρόνο, αλλά παρέχει μια λεπτομερή, ακριβή προβολή της δραστηριότητας του kernel. Αυτή η λειτουργία είναι χρήσιμη για τον έλεγχο της λειτουργικότητας της λογικής που θα πάει στην FPGA και για τη λήψη αρχικών εκτιμήσεων απόδοσης.

- **Hardware:** Ο κώδικας του kernel μεταγλωττίζεται σε ένα μοντέλο υλικού RTL και στη συνέχεια υλοποιείται στην FPGA, με αποτέλεσμα να δημιουργείται ένα binary file (.xclbin) που θα τρέχει στην πραγματική FPGA.

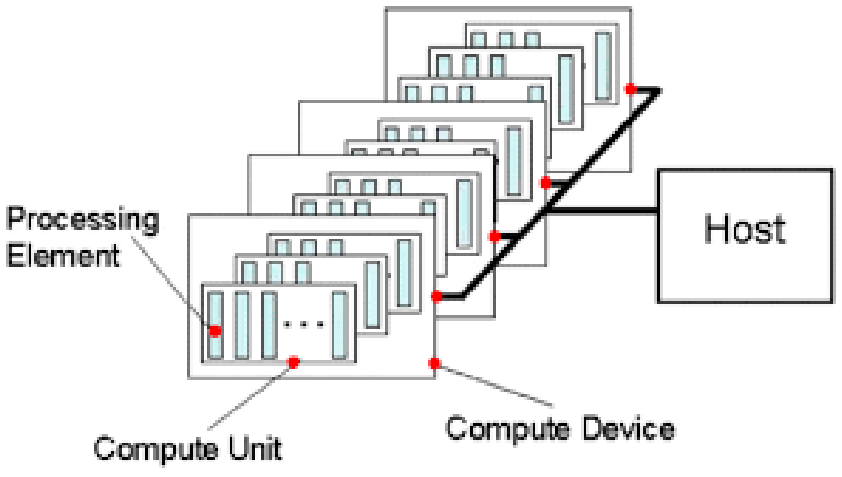
Στην παρούσα εργασία θα πραγματοποιήσουμε software emulation για να επαληθεύσουμε ότι οι κώδικες του host και του kernel δεν έχουν συντακτικά λάθη και ότι ο kernel μας είναι λειτουργικά ο ίδιος με την αρχική συνάρτηση sgemm. Επίσης, δεν θα ασχοληθούμε με Hardware Emulation, καθώς έχουμε ήδη επιβεβαιώσει, από το Vitis HLS, ότι η λογική του kernel μας είναι σωστή. Τέλος, θα τρέξουμε τον κώδικα στο Hardware της FPGA από το οποίο θα πάρουμε τις πραγματικές μετρήσεις και τις οποίες θα συγκρίνουμε με τα αποτελέσματα που βγάλαμε από το VTune για να πάρουμε την επιτάχυνση του kernel.

5.2 OpenCL Framework

Το OpenCL (Open Computing Language) είναι ένα framework για τη σύνταξη προγραμμάτων που εκτελούνται σε ετερογενείς πλατφόρμες που αποτελούνται από κεντρικές μονάδες επεξεργασίας (CPU), μονάδες επεξεργασίας γραφικών (GPU), επεξεργαστές ψηφιακού σήματος (DSP), FPGA και άλλα είδη επεξεργαστών ή επιταχυντών υλικού. Το OpenCL καθορίζει γλώσσες προγραμματισμού (C99-based, C++14 και C++17) για τον προγραμματισμό αυτών των συσκευών και των APIs για τον

έλεγχο της πλατφόρμας και την εκτέλεση προγραμμάτων στις υπολογιστικές συσκευές. Το OpenCL διαθέτει μια διεπαφή για παράλληλους υπολογισμούς, χρησιμοποιώντας παραλληλισμό που βασίζεται σε εργασίες και δεδομένα. Ο προγραμματισμός του OpenCL γίνεται βάσει των παρακάτω τριών μοντέλων:

- **Platform Model:** Ορίζει τη λογική αναπαράσταση όλου του υλικού που είναι ικανό να εκτελέσει ένα πρόγραμμα OpenCL. Οι πλατφόρμες OpenCL ορίζονται από την ομαδοποίηση ενός κεντρικού επεξεργαστή και μιας ή περισσότερων υπολογιστικών συσκευών OpenCL. Ο host, στον οποίο εκτελείται το λειτουργικό σύστημα για το σύστημα, είναι επίσης υπεύθυνος για την εκκίνηση διεργασιών που σχετίζονται με την εκτέλεση των εφαρμογών OpenCL. Η υπολογιστική συσκευή είναι το στοιχείο υλικού στο σύστημα στο οποίο εκτελούνται οι kernels μιας εφαρμογής OpenCL. Κάθε συσκευή χωρίζεται περαιτέρω σε ένα σύνολο υπολογιστικών μονάδων. Ο αριθμός των υπολογιστικών μονάδων εξαρτάται από το υλικό-στόχο. Μια υπολογιστική μονάδα υποδιαιρείται περαιτέρω σε στοιχεία επεξεργασίας (compute units). Ένα στοιχείο επεξεργασίας είναι η θεμελιώδης μηχανή υπολογισμού στην υπολογιστική μονάδα, η οποία είναι υπεύθυνη για την εκτέλεση των λειτουργιών ενός αντικειμένου εργασίας.



Εικόνα 5.4: Αναπαράσταση μοντέλου πλατφόρμας OpenCL

- **Memory Model:** Ορίζει τη συμπεριφορά και την ιεραρχία της μνήμης που μπορεί να χρησιμοποιηθεί από τις εφαρμογές OpenCL. Σε όλες τις υλοποιήσεις OpenCL η ιεραρχική αναπαράσταση της μνήμης είναι κοινή, αλλά αναλόγως τον προμηθευτή καθορίζεται ο τρόπος με τον οποίο το μοντέλο μνήμης OpenCL αντιστοιχίζεται σε συγκεκριμένο υλικό. Το μοντέλο της μνήμης OpenCL αποτελείται από 5 επιμέρους τμήματα μνήμης:

    1. Host Memory: Η μνήμη του host ορίζεται ως η περιοχή της μνήμης του συστήματος που είναι άμεσα και αποκλειστικά προσβάσιμη από τον host. Οποιαδήποτε δεδομένα χρειάζονται οι kernels θα πρέπει να μεταφέρονται προς και από την καθολική μνήμη, χρησιμοποιώντας το OpenCL API.

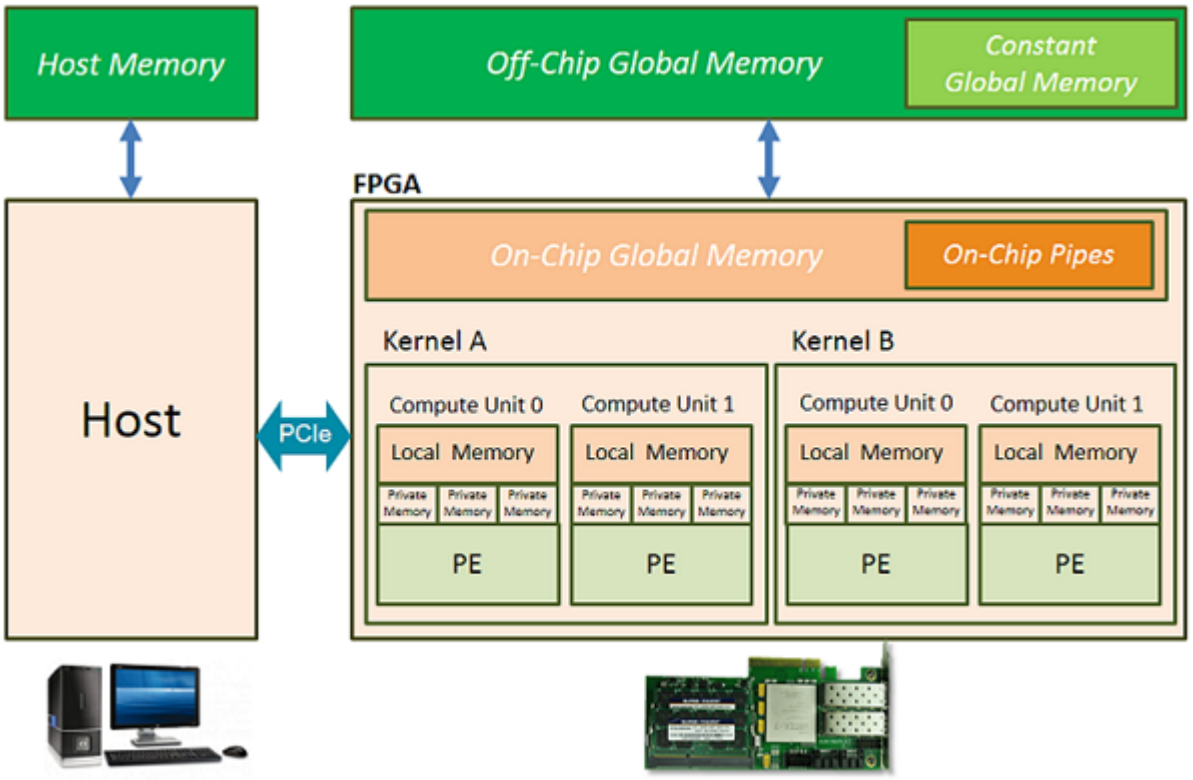
    2. Global Memory: Η καθολική μνήμη ορίζεται ως η περιοχή της μνήμης της συσκευής που είναι προσβάσιμη τόσο από τον host όσο και από την συσκευή. Η καθολική μνήμη επιτρέπει την πρόσβαση ανάγνωσης/εγγραφής στον επεξεργαστή του host, καθώς και σε όλες τις υπολογιστικές μονάδες της συσκευής. Ο host είναι υπεύθυνος για την κατανομή και την αποδέσμευση των buffers σε αυτόν τον χώρο μνήμης. Ο επεξεργαστής του host μεταφέρει δεδομένα από τον χώρο μνήμης του host στον καθολικό χώρο μνήμης. Στη συνέχεια, μόλις ενεργοποιηθεί ένας kernel για την επεξεργασία των δεδομένων, ο host χάνει τα δικαιώματα πρόσβασης στο buffer στην καθολική μνήμη. Τότε η συσκευή αναλαμβάνει και είναι σε θέση να διαβάζει και να γράφει από την καθολική μνήμη μέχρι να ολοκληρωθεί η εκτέλεση του kernel. Με την ολοκλήρωση των λειτουργιών του kernel , η συσκευή επιστρέφει τον έλεγχο του καθολικού buffer μνήμης στον επεξεργαστή του host. Μόλις ανακτήσει τον έλεγχο ενός buffer, ο host μπορεί να διαβάσει και να γράψει δεδομένα στον buffer, να μεταφέρει δεδομένα πίσω στη μνήμη του host και τέλος να αποδεσμεύσει τον buffer.

    3. Constant Global Memory: Η σταθερή καθολική μνήμη ορίζεται ως η περιοχή της μνήμης συστήματος που είναι

προσβάσιμη με δικαιώματα ανάγνωσης και εγγραφής για τον host και με δικαίωμα μόνο για ανάγνωση για τη συσκευή OpenCL. Η τυπική χρήση αυτής της μνήμης είναι η μεταφορά σταθερών δεδομένων που απαιτούνται από τον υπολογισμό του kernel από τον host στη συσκευή.

4. Local Memory: Αποτελεί μία τοπική περιοχή μνήμης σε μια μεμονωμένη υπολογιστική μονάδα. Ο host δεν έχει πρόσβαση σε αυτόν τον χώρο μνήμης. Αυτό το επίπεδο μνήμης επιτρέπει λειτουργίες ανάγνωσης και εγγραφής από όλα τα στοιχεία επεξεργασίας με υπολογιστικές μονάδες και χρησιμοποιείται, κατά κύριο λόγο, για την αποθήκευση δεδομένων που πρέπει να μοιράζονται πολλά work-items. Οι λειτουργίες στην τοπική μνήμη είναι αποδιοργανωμένες μεταξύ των work-items, αλλά ο συγχρονισμός και η συνοχή μπορούν να επιτευχθούν χρησιμοποιώντας λειτουργίες φραγμού και φράχτη.

5. Private Memory: Αποτελεί μία περιοχή της μνήμης που είναι ιδιωτική σε ένα μεμονωμένο αντικείμενο εργασίας που εκτελείται μέσα σε ένα στοιχείο επεξεργασίας OpenCL. Όπως και με την τοπική μνήμη, ο host δεν έχει πρόσβαση σε αυτήν την περιοχή μνήμης. Αυτός ο χώρος μνήμης μπορεί να διαβαστεί και να εγγραφεί σε όλα τα στοιχεία εργασίας, αλλά οι μεταβλητές που ορίζονται στην ιδιωτική μνήμη ενός work-item δεν είναι προσβάσιμες σε ένα άλλο work-item.



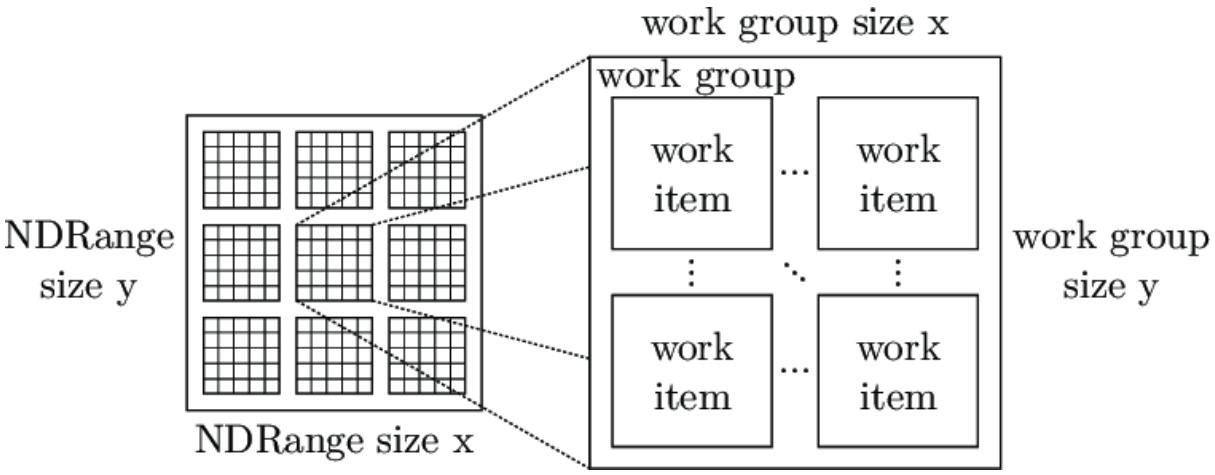
Εικόνα 5.5: Αναπαράσταση μοντέλου μνήμης OpenCL

- **Execution Model:** Καθορίζει τον τρόπο με τον οποίο εκτελούνται οι kernels. Όταν οι OpenCL kernels υποβάλλονται για εκτέλεση σε μία συσκευή OpenCL, εκτελούνται σε ένα index space. Ένα παράδειγμα index space είναι ένας βρόχος for στη C/C++. Στον βρόχο for που ορίζεται από την πρόταση "for(int i=0; i<10; i++)", όλες οι εντολές εντός αυτού του βρόχου θα εκτελεστούν 10 φορές, με i = 0,1,2...,9. Σε αυτή την περίπτωση το index space του βρόχου είναι [0,1,2...,9]. Στο OpenCL, οι index spaces ονομάζονται NDRanges και μπορούν να έχουν 1, 2 ή 3 διαστάσεις.

  Οι συναρτήσεις kernel της OpenCL εκτελούνται ακριβώς μία φορά για κάθε σημείο στο NDRange. Αυτή η μονάδα εργασίας για κάθε σημείο στο NDRange ονομάζεται αντικείμενο εργασίας (work-item). Σε αντίθεση με τους βρόγχους στη C, όπου οι επαναλήψεις εκτελούνται διαδοχικά και με τη σειρά, ένας χρόνος εκτέλεσης (runtime) και μια συσκευή OpenCL είναι ελεύθεροι να εκτελούν work-items παράλληλα και με οποιαδήποτε σειρά. Το χαρακτηριστικό αυτό του μοντέλου εκτέλεσης OpenCL επιτρέπει

στον προγραμματιστή να εκμεταλλευτεί τους παράλληλους υπολογιστικούς πόρους.

Τα work-items δεν έχουν προγραμματιστεί για μεμονωμένη εκτέλεση σε συσκευές OpenCL. Αντίθετα, οργανώνονται σε ομάδες εργασίας (work-groups). Οι ομάδες εργασίας ορίζουν επίσης το σύνολο των work-items που ενδέχεται να μοιράζονται δεδομένα χρησιμοποιώντας τοπική μνήμη.



Εικόνα 5.6: Αναπαράσταση μοντέλου εκτέλεσης OpenCL

5.3 Δημιουργία project στο Vitis

Αρχικά δημιουργούμε ένα project στο Vitis, επιλέγοντας την πλατφόρμα “xilinx_u200_gen3x16_xdma_base_2-0”. Στο project ορίζουμε ως kernel την τροποποιημένη συνάρτηση sgemm ακριβώς όπως την υλοποιήσαμε στο Vitis HLS. Στο project αποφασίσαμε να εκτελέσουμε μόνο τα κομμάτια του κώδικα του HPL-AI benchmark που σχετίζονται με:

- Τη δημιουργία του πίνακα A, την αρχικοποίηση του με τυχαίες τιμές και τη μετατροπή του από διπλή σε μονή ακρίβεια (συναρτήσεις: matgen, convert_double_to_float).

- Τη διαδικασία της LU παραγοντοποίησης, η οποία είναι και η πιο χρονοβόρα (συναρτήσεις: sgetrf_nopiv, sgetrf2_nopiv, sgemm, strsm).

5.4 Περιορισμοί

Έχοντας συμπεριλάβει στο project μας τις συναρτήσεις sgetrf_nopiv και sgetrf2_nopiv του benchmark, εισάγονται κάποιοι περιορισμοί στην υλοποίησή μας οι οποίοι καθιστούν τη διαδικασία της βελτιστοποίησης του kernel αρκετά πιο δύσκολη. Συγκεκριμένα, η συνάρτηση kernel sgemm καλείται μία φορά από την sgetrf_nopiv και πολλές φορές από την sgetrf2_nopiv, η οποία με τη σειρά της καλείται επίσης πολλές φορές αναδρομικά και από την sgetrf_nopiv. Σε κάθε κλήση της, η sgemm δέχεται ως ορίσματα 3 δείκτες, οι οποίοι δείχνουν σε διαφορετικά σημεία του ίδιου πίνακα A και με διαφορετικά μεγέθη κάθε φορά. Τα μεγέθη αυτά εξαρτώνται από το αρχικό μέγεθος του προβλήματος N που έχουμε ορίσει εξ αρχής (N=512) και σε ποια κλήση της συνάρτησης βρισκόμαστε, καθώς μετά από κάθε εκτέλεση της συνάρτησης τα μεγέθη αυτά αλλάζουν, όπως φαίνεται στο παρακάτω τμήμα κώδικα:

```
sgemm('N', 'N', m - j - jb, n - j - jb, jb, -1.0, &A(j + jb, j),
      lda, &A(j, j + jb), lda, 1.0, &A(j + jb, j + jb), lda);
```

Εικόνα 5.7: Κλήση της sgemm από τη συνάρτηση sgetrf_nopiv

```
sgemm('N', 'N', m - n1, n2, n1, -1.0, &A(n1, 0), lda, &A(0, n1), lda,
      1.0, &A(n1, n1), lda);
```

Εικόνα 5.8: Κλήση της sgemm από τη συνάρτηση sgetrf2_nopiv

Ακόμη, από τη φύση του αλγορίθμου, η επεξεργασία των στοιχείων των pointers που δείχνουν στον πίνακα A γίνεται ανά στήλη και όχι ανά γραμμή όπως συνηθίζεται, διότι η διαδικασία αυτή είναι ταχύτερη.

```
#define HPLAI_INDEX2D(PTR, R, C, LDIM) ( ((PTR) + (R)) + sizeof(char) * (C)
                    / sizeof(char) * (LDIM) )
#define A(i, j) *HPLAI_INDEX2D(A, (i), (j), lda)
```

Εικόνα 5.9: Ορισμός πίνακα A με την μορφή HPLAI_INDEX2D

```
void sgemm(char transa, char transb, int m, int n, int k,
           float alpha, float *A, int lda, float *B, int ldb,
           float beta, float *C, int ldc) {
    int i, j, l;

    for (j = 0; j < n; j++) {
        for (l = 0; l < k; l++) {
            float temp = alpha * B(l, j);
            for (i = 0; i < m; i++) {
                C(i, j) += temp * A(i, l);
            }
        }
    }
    return;
}
```

Εικόνα 5.10: Συνάρτηση sgemm με πίνακες A, B, C της μορφής HPLAI_INDEX2D

```
void sgemm(char transa, char transb, int m, int n, int k,
           float alpha, float *A, int lda, float *B, int ldb,
           float beta, float *C, int ldc) {
    int i, j, l;

    for (j = 0; j < n; j++) {
        for (l = 0; l < k; l++) {
            float temp = alpha * B[j*ldb+l];
            for (i = 0; i < m; i++) {
                C[j*ldc+i] += temp * A[l*lda+i];
            }
        }
    }
    return;
}
```

Εικόνα 5.11: Συνάρτηση sgemm με κανονικούς πίνακες A, B, C

Στον κώδικα της εικόνας 5.9, παρέχεται η δυνατότητα ο μονοδιάστατος πίνακας A να μπορεί να εκφραστεί ως δισδιάστατος, όπου το i αντιπροσωπεύει τις στήλες, το j τις γραμμές και το lda τον αριθμό των στηλών. Οι κώδικες των εικόνων 5.10 και 5.11 υλοποιούν την λειτουργικότητα της συνάρτησης sgemm και είναι ισοδύναμοι μεταξύ τους.

Ως συνέπεια των παραπάνω περιορισμών η μεταφορά των δεδομένων των πινάκων από τον host στον kernel και το αντίστροφο γίνεται πιο πολύπλοκη, πιο δύσκολη και εισάγει κάποια καθυστέρηση στην εφαρμογή μας.

5.5 Ανάλυση κώδικα host

Η υλοποίηση του κώδικα του host μπορεί να αναλυθεί στα παρακάτω βήματα:

- Αρχικά, δημιουργούμε 2 πίνακες τους οποίους αρχικοποιούμε με τα ίδια στοιχεία. Οι πίνακες αυτοί αντιπροσωπεύουν τον πίνακα A του προβλήματος, όπου ο ένας πίνακας αντιστοιχεί στην εκτέλεση της LU παραγοντοποίησης με την αρχική, μη τροποποιημένη συνάρτηση sgemm, ενώ ο δεύτερος αντιστοιχεί στην εκτέλεση της LU παραγοντοποίησης με την τροποποιημένη συνάρτηση-kernel sgemm. Στο τέλος της εφαρμογής, θα ακολουθήσει μία σύγκριση των δύο πινάκων για επαλήθευση των αποτελεσμάτων.
- Προγραμματίζουμε την συσκευή FPGA καθορίζοντας το ID και φορτώνοντας το αρχείο xclbin.
- Καλούμε την συνάρτηση sgetrf_nopiv για να πάρουμε τα αποτελέσματα της LU παραγοντοποίησης με την αρχική συνάρτηση sgemm και αμέσως μετά καλούμε μία κατάλληλα τροποποιημένη συνάρτηση sgetrf_nopiv_hw από την οποία θα πάρουμε τα αποτελέσματα της LU παραγοντοποίησης με την τροποποιημένη συνάρτηση-kernel sgemm.
- Δέσμευση buffers στην καθολική μνήμη για τη μεταφορά των δεδομένων από τη συσκευή στον host.

- Δημιουργία του kernel και καθορισμός των παραμέτρων του.
- Μεταφορά των δεδομένων ανάμεσα στον host και στον kernel και εκτέλεση του kernel.
- Επιστροφή αποτελεσμάτων από τον kernel και επαλήθευσή τους.

Λόγω των περιορισμών του αλγορίθμου που εξηγήθηκαν προηγουμένως και για να απλουστεύσουμε τη διαδικασία κλήσης του kernel, θα δημιουργήσουμε μία συνάρτηση “call_kernel”, η οποία θα δημιουργεί τους buffers και τον kernel, τον οποίο και θα καλεί με ανανεωμένες παραμέτρους κάθε φορά. Με αυτόν τον τρόπο δεσμεύεται όση μνήμη είναι μόνο απαραίτητη για τους buffers σε κάθε κλήση του kernel και οι δείκτες A, B και C δείχνουν στη σωστή περιοχή του πίνακα A κάθε φορά. Μόλις οριστούν οι παράμετροι και εκτελεστεί ο kernel, τα δεδομένα επιστρέφονται πίσω στην καθολική μνήμη και η συνάρτηση τερματίζει.

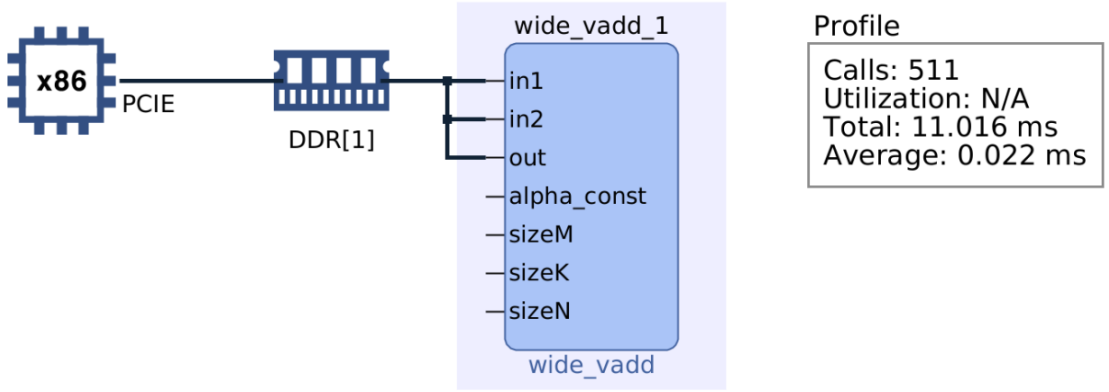
5.6 Software Emulation

Όπως έχουμε ήδη προαναφέρει, ο σκοπός της εκτέλεσης του software emulation είναι να επιβεβαιώσουμε την ορθή λειτουργία του κώδικα του kernel και του host και να εντοπίσουμε τυχόν συντακτικά λάθη. Τα αποτελέσματα που θα λάβουμε σε αυτό το στάδιο δε μας απασχολούν. Κατά την διαδικασία του software emulation θα τρέξει πρώτα η συνάρτηση sgetrf_nopiv και έπειτα η συνάρτηση sgetrf_nopiv_hw που περιέχει την συνάρτηση call_kernel η οποία καλεί την τροποποιημένη sgemm, όπως αναφέρθηκε προηγουμένως. Στο τέλος, γίνεται επαλήθευση των στοιχείων των δύο πινάκων με τη χρήση βοηθητικών συναρτήσεων, οι οποίες μεταφέρουν τα δεδομένα των πινάκων σε 2 αρχεία και τα συγκρίνουν για να διαπιστωθεί ότι είναι ίδια. Χωρίς να πάρουμε οποιοδήποτε error ή warning στην κονσόλα, η διαδικασία του software emulation ολοκληρώθηκε επιτυχώς.

5.7 Hardware

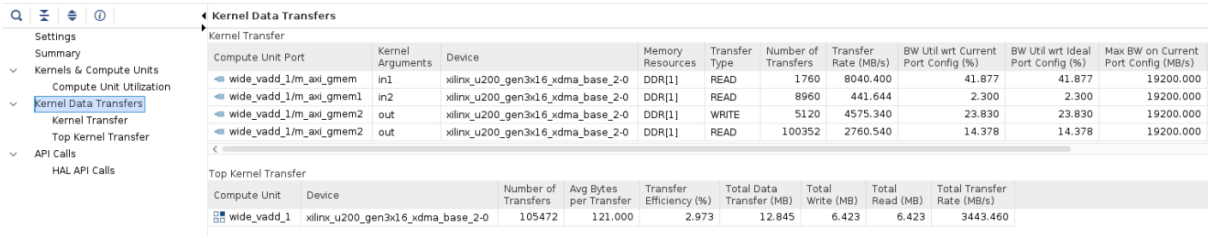
5.7.1 Αποτελέσματα Hardware για 1 SLR

Σε αυτό το στάδιο, για να μεταφερθεί ο κώδικας της εφαρμογής και να γίνει build το project στο Hardware της FPGA χρειάστηκε αρκετός χρόνος. Στο συγκεκριμένο design γίνεται χρήση ενός memory bank από μία υπολογιστική μονάδα SLR. Τα αποτελέσματα που προέκυψαν παρουσιάζονται παρακάτω:

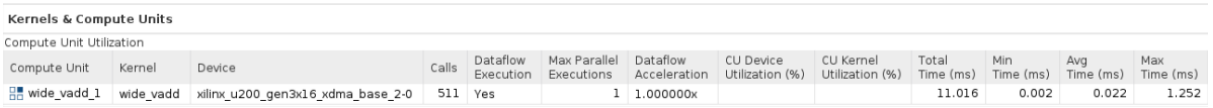


Εικόνα 5.12: System diagram

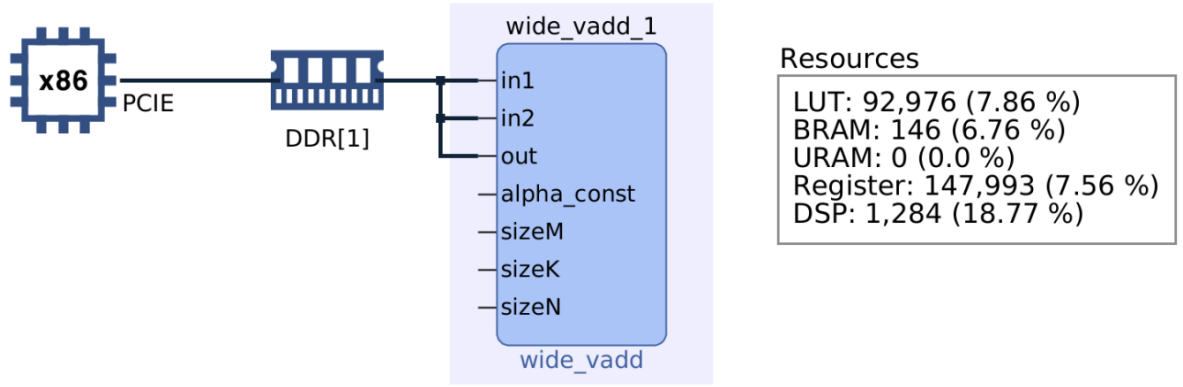
Εικόνα 5.13: Timeline trace



Εικόνα 5.14: Kernel data transfers



Εικόνα 5.15: Compute unit utilization

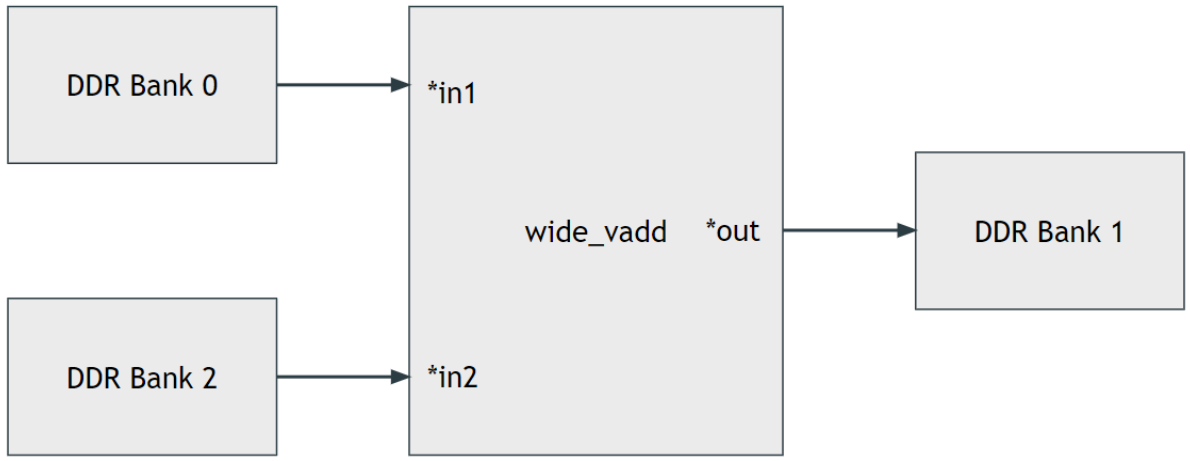


Εικόνα 5.16: System diagram resources utilization

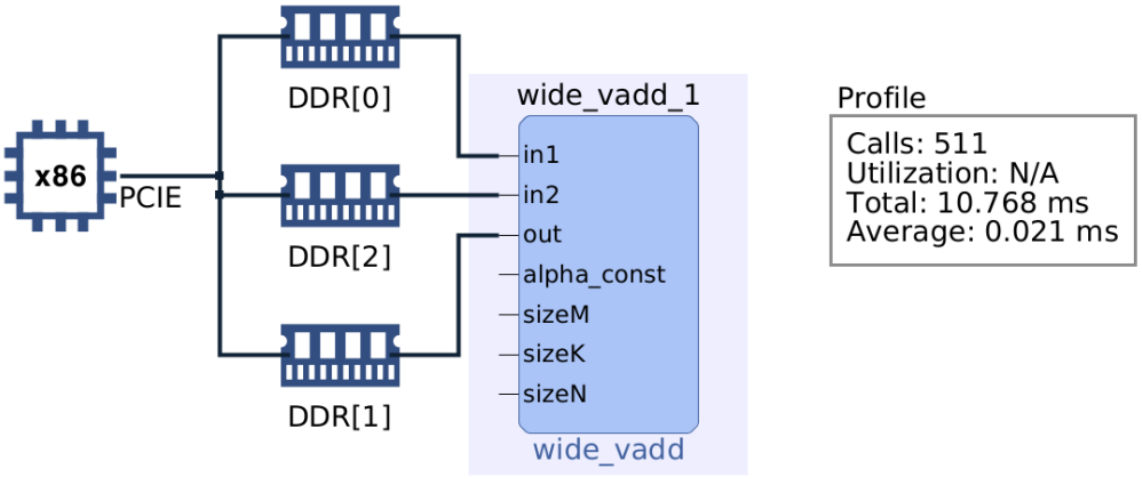
Σύμφωνα με τα παραπάνω αποτελέσματα, ο συνολικός χρόνος εκτέλεσης για 511 κλήσεις του kernel είναι 11,016ms, ενώ χρησιμοποιείται μόνο μία από τις 4 memory banks. Επίσης, τα ποσοστά χρήσης των πόρων σύμφωνα με την εικόνα 5.16 είναι σχετικά χαμηλά.

5.7.2 Αποτελέσματα Hardware για 2 SLRs

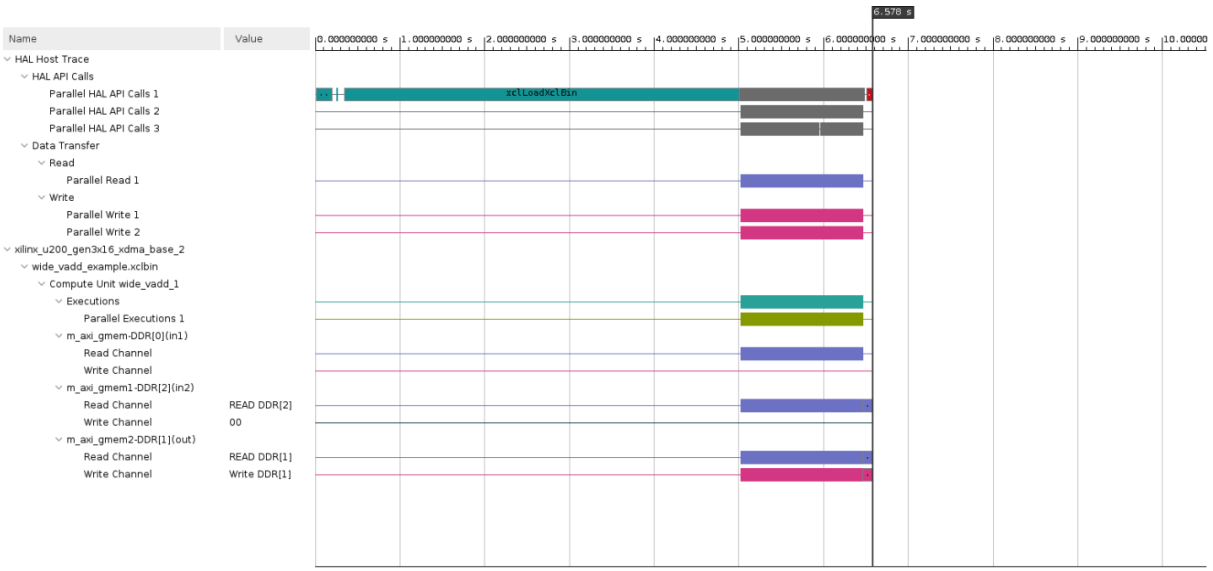
Στην προηγούμενη ενότητα, χρησιμοποιήσαμε μόνο μία memory bank από μία υπολογιστική μονάδα SLR. Σε αυτό το design, θα χρησιμοποιήσουμε 3 memory banks από 2 SLRs. Ο επιταχυντής Alveo U200 διαθέτει συνολικά 4 memory banks DDR4. Επειδή στον kernel μας έχουμε 3 παραμέτρους μεταφοράς δεδομένων (δύο ανάγνωσης και μία εγγραφής) μπορεί να είναι δυνατή η βελτίωση της απόδοσης εάν χρησιμοποιούνται περισσότερες memory banks ταυτόχρονα, παρέχοντας μεγιστοποίηση του εύρους ζώνης (bandwidth) που είναι διαθέσιμο σε καθεμία από τις διεπαφές. Με αυτόν τον τρόπο δίνεται η δυνατότητα να εκτελούνται συναλλαγές δεδομένων υψηλού εύρους ζώνης ταυτόχρονα με διαφορετικές, εξωτερικές memory banks. Επομένως, είναι δυνατό να χρησιμοποιήσουμε την τοπολογία του παρακάτω σχήματος:



Εικόνα 5.17: System diagram



Εικόνα 5.18: System diagram



Εικόνα 5.19: Timeline trace

Kernel Data Transfers

Kernel Transfer

Compute Unit Port	Kernel Arguments	Device	Memory Resources	Transfer Type	Number of Transfers	Transfer Rate (MB/s)	BW Util wrt Current Port Config (%)	BW Util wrt Ideal Port Config (%)	Max BW on Current Port Config (MB/s)
wide_vadd_1/m_axi_gmem	in1	xilinx_u200_gen3x16_xdma_base_2-0	DDR[0]	READ	1760	8154.140	42.470	42.470	19200.000
wide_vadd_1/m_axi_gmem1	in2	xilinx_u200_gen3x16_xdma_base_2-0	DDR[2]	READ	8960	579.812	3.020	3.020	19200.000
wide_vadd_1/m_axi_gmem2	out	xilinx_u200_gen3x16_xdma_base_2-0	DDR[1]	WRITE	5120	4692.750	24.441	24.441	19200.000
wide_vadd_1/m_axi_gmem2	out	xilinx_u200_gen3x16_xdma_base_2-0	DDR[1]	READ	100352	3120.520	16.253	16.253	19200.000

Top Kernel Transfer

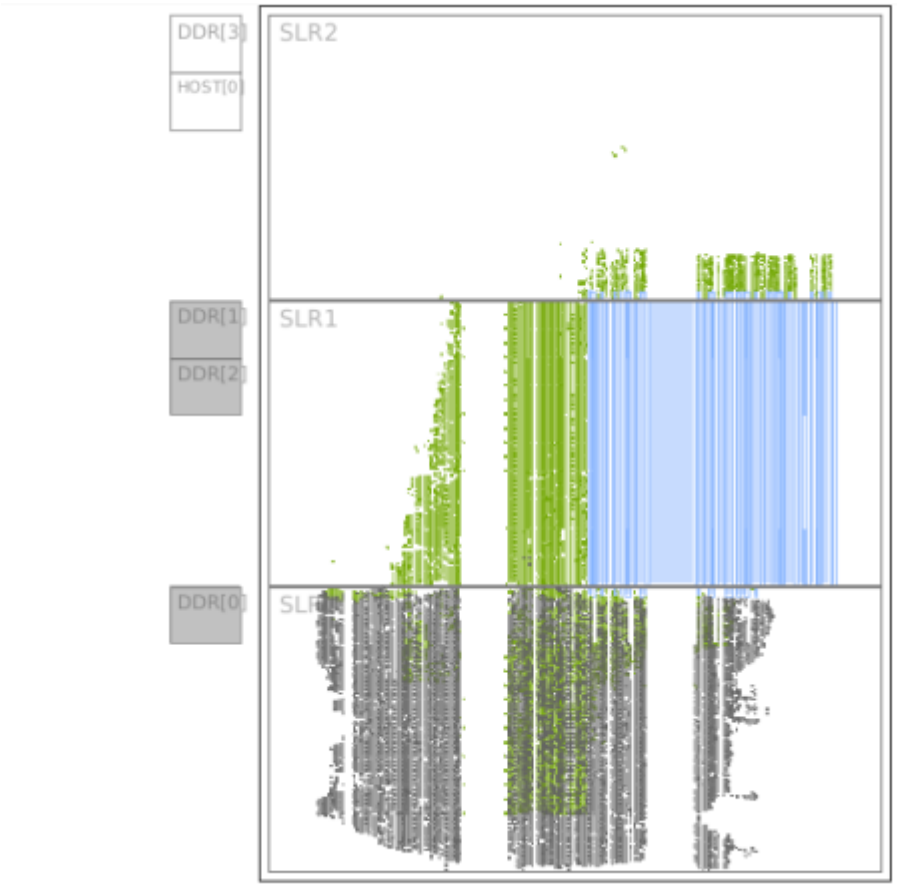
Compute Unit	Device	Number of Transfers	Avg Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)
wide_vadd_1	xilinx_u200_gen3x16_xdma_base_2-0	105472	121.000	2.973	12.845	6.423	6.423	3748.450

Εικόνα 5.20: Kernel data transfers

Compute Unit Utilization

Compute Unit	Kernel	Device	Calls	Dataflow Execution	Max Parallel Executions	Dataflow Acceleration	CU Device Utilization (%)	CU Kernel Utilization (%)	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
wide_vadd_1	wide_vadd	xilinx_u200_gen3x16_xdma_base_2-0	511	Yes	1	1.000000x			10.768	0.002	0.021	1.216

Εικόνα 5.21: Compute unit utilization



Εικόνα 5.22: Device map

Σύμφωνα με τα αποτελέσματα της εικόνας 5.18, ο συνολικός χρόνος εκτέλεσης για 511 κλήσεις του kernel μειώθηκε στα 10,768ms με την χρήση τριών memory banks από 2 SLRs. Στην εικόνα 5.22 παρουσιάζεται ο χάρτης κατανομής των δεδομένων στις 3 SLRs της συσκευής.

5.8 Σύγκριση αποτελεσμάτων

Τα αποτελέσματα του VTune και του Vitis στο hardware της FPGA παρουσιάζονται συγκεντρωμένα στον παρακάτω πίνακα:

Συνολικός χρόνος εκτέλεσης της sgemm	
VTune	39,850ms
Vitis Hardware 1 SLR	11,016ms
Vitis Hardware 2 SLRs	10,768ms

Πίνακας 5.1: Σύγκριση αποτελεσμάτων του συνολικού χρόνου εκτέλεσης της sgemm

Παρατηρούμε ότι με τις προσθήκες των διάφορων directives και interfaces που κάναμε για παραλληλοποίηση των δεδομένων, μειώσαμε το συνολικό χρόνο εκτέλεσης της sgemm από τα 39,850ms στα 11,016ms με την χρήση μίας μόνο memory bank από μία SLR. Χρησιμοποιώντας 3 διαφορετικές memory banks από 2 διαφορετικές SLRs καταφέραμε να μειώσουμε περαιτέρω το συνολικό χρόνο εκτέλεσης του kernel στα 10,768ms πετυχαίνοντας έτσι τελική επιτάχυνση $\times$**3,7**.

5.9 Συμπέρασμα

Σε αυτό το κεφάλαιο πραγματοποιήσαμε μία ανάλυση του προγράμματος Vitis και του OpenCL framework και εκτελέσαμε software emulation και hardware για τον kernel της τροποποιημένης συνάρτησης sgemm, όπως τη φτιάξαμε στο Vitis HLS στο κεφάλαιο 4. Έχοντας επαληθεύσει τη σωστή λειτουργία του kernel τρέχοντας software emulation, στη συνέχεια τρέξαμε τον kernel σε hardware και, τέλος, συγκρίναμε τα αποτελέσματα που πήραμε με εκείνα από το VTune στο κεφάλαιο 3 και διαπιστώσαμε ότι ο συνολικός χρόνος της συνάρτησης sgemm μειώθηκε σημαντικά με τις αλλαγές που κάναμε, πετυχαίνοντας έτσι μία $\times$**3.7** επιτάχυνση. Έτσι, λοιπόν, καταφέραμε να επιταχύνουμε σε έναν ικανοποιητικό βαθμό το πιο χρονοβόρο τμήμα του κώδικα του HPL-AI benchmark, χρησιμοποιώντας μόνο υψηλού επιπέδου γλώσσες προγραμματισμού και τα εργαλεία της Xilinx.

Κεφάλαιο 6: Σύνοψη

6.1 Σύνοψη διπλωματικής εργασίας

Στην παρούσα διπλωματική εργασία καλούμαστε να πραγματοποιήσουμε μία χρονική ανάλυση του HPL-AI benchmark προκειμένου να βρούμε την πιο χρονοβόρα συνάρτηση του προγράμματος και στη συνέχεια να επιταχύνουμε την εκτέλεσή της πάνω σε έναν high-end FPGA επιταχυντή, με την χρήση των κατάλληλων directives στα εργαλεία Vitis HLS και Vitis.

Στο **κεφάλαιο 1**, κάναμε μία εισαγωγή στο HPL-AI benchmark, μία σύντομη περιγραφή των FPGA, καθώς και μία επισκόπηση των εργαλείων που θα χρησιμοποιήσουμε στην πορεία της διπλωματικής.

Στο **κεφάλαιο 2**, κάναμε μία θεωρητική ανάλυση του HPL-AI benchmark και των μεθόδων της LU παραγοντοποίησης και της επαναληπτικής διόρθωσης GMRES.

Στο **κεφάλαιο 3**, χρησιμοποιήσαμε το εργαλείο VTune profiler της Intel προκειμένου να κάνουμε μία χρονική ανάλυση του benchmark και να βρούμε το πιο χρονοβόρο τμήμα της εφαρμογής. Πραγματοποιώντας ένα performance snapshot και μία hotspots ανάλυση, επιλέξαμε την συνάρτηση sgemm ως την υποψήφια συνάρτηση προς επιτάχυνση.

Στη συνέχεια, στο **κεφάλαιο 4**, μεταφέραμε την συνάρτηση sgemm στο εργαλείο Vitis HLS και κάναμε τις απαραίτητες τροποποιήσεις, ώστε ο κώδικας να γίνει πιο φιλικός ως προς το υλικό και να βελτιωθεί η απόδοσή της, χωρίς να αλλάξει η λειτουργικότητά της. Έχοντας επιβεβαιώσει επιτυχώς την ορθή λειτουργία της συνάρτησης στο στάδιο C simulation, περάσαμε στο στάδιο της C synthesis όπου με τη χρήση των κατάλληλων directives και interfaces κάναμε τη συνάρτηση πολύ πιο αποδοτική από ότι ήταν προηγουμένως.

Στο **κεφάλαιο 5**, δημιουργήσαμε ένα project στο Vitis με επιταχυντή την πλατφόρμα “xilinx_u200_gen3x16_xdma_base_2-0”, στο οποίο βάλαμε για kernel την τροποποιημένη συνάρτηση sgemm του προηγούμενου κεφαλαίου και αναπτύξαμε τον κώδικα του host. Στη συνέχεια, πραγματοποιήσαμε software emulation, στον επεξεργαστή του host, για να ελέγξουμε ότι η εφαρμογή μας λειτουργεί σωστά και, έπειτα, τρέξαμε την εφαρμογή στο hardware της FPGA, από το οποίο πήραμε μία ×**3.7** επιτάχυνση του χρόνου εκτέλεσης της συνάρτησης sgemm σε σύγκριση με τον χρόνο εκτέλεσης της ίδιας συνάρτησης στο VTune.

6.2 Κάλυψη στόχων της διπλωματικής εργασίας

Οι στόχοι της παρούσας διπλωματικής εργασίας ήταν οι εξής:

1. Εντοπισμός της πιο χρονοβόρας συνάρτησης του benchmark.
2. Κατάλληλη τροποποίηση του κώδικα της συνάρτησης, ώστε να μπορεί να μεταφερθεί στο εργαλείο Vitis.
3. Εκτέλεση του κώδικα της συνάρτησης kernel στο Vitis με σκοπό την επιτάχυνση της εφαρμογής.

Ο πρώτος στόχος ολοκληρώθηκε με επιτυχία, καθώς μέσω ενός performance snapshot και μίας ανάλυσης hotspots εντοπίσαμε τη συνάρτηση sgemm ως την πιο χρονοβόρα συνάρτηση του benchmark.

Ο δεύτερος στόχος, επίσης, επετεύχθη, αφού με την χρήση ορισμένων directives και interfaces στο Vitis HLS, κατορθώσαμε να τροποποιήσουμε τη συνάρτηση sgemm, βελτιώνοντας την απόδοσή της, χωρίς να αλλάξει η λειτουργικότητά της.

Τέλος, όσον αφορά τον τρίτο στόχο, καταφέραμε να πετύχουμε μία ×**3.7** επιτάχυνση του χρόνου εκτέλεσης του kernel στο hardware της FPGA σε σύγκριση με τον χρόνο εκτέλεσης της ίδιας συνάρτησης στο VTune. Ωστόσο, επειδή έχουμε έναν kernel, χρησιμοποιούμε μόνο μία από τις 3 SLRs, που σημαίνει ότι οι διαθέσιμοι πόροι είναι περιορισμένοι στο 1/3 των συνολικών πόρων της FPGA.

6.3 Προτάσεις για μελλοντική επέκταση της εργασίας

Για βελτίωση των αποτελεσμάτων της παρούσας εργασίας, θα μπορούσαμε να χρησιμοποιήσουμε παραπάνω από μία υπολογιστικές μονάδες SLR, ώστε να έχουμε στη διάθεσή μας περισσότερους πόρους, όπως BRAMs και DSPs, για παραλληλοποίηση μεγαλύτερου τμήματος των δεδομένων και άρα να πετύχουμε ακόμα μεγαλύτερη επιτάχυνση της εφαρμογής. Ακόμη, θα μπορούσε να γίνει μία μικρή τροποποίηση στον κώδικα του host για ταχύτερη μεταφορά των δεδομένων στους buffers.

Παράρτημα 1: Ακρωνύμια και συντομογραφίες

AHB -Advanced High Performance Bus
AHDL -Altera Hardware Description Language
AI -Artificial Intelligence
AMBA -Advanced Microcontroller Bus Architecture
API -Application Programming Interface
AXI -Advanced eXtensible Interface
BLAS -Basic Linear Algebra Subprograms
BRAM -Block Random Access Memory
CPU -Central Processing Unit
DMA -Direct Memory Access
DSP -Digital Signal Processor
FF -Flip Flop
FPGA -Field Programmable Gate Array
GEMM -General Matrix Multiply
GFLOPS -Giga floating point operations
GMRES -Generalized Minimal Residual Method
GPU -Graphics Processing Unit
HDL -Hardware Description Language
HLS -High Level Synthesis
HPC -High Performance Computing
HPL -High Performance Linpack
IC -Integrated circuit
IR -Iterative Refinement
LU -Lower-Upper
LUT -Look Up Table
ML -Machine Learning
MPSoC -MultiProcessor System on Chip
PCIe -Peripheral Component Interconnect Express
PLL -Phase-Locked Loop
RAM -Random Access Memory
RTL -Register Transfer Level
SDR -Software Defined Radio
SLR -Super Logic Region
SoC -System on Chip

TRSM **-**Triangular solve matrix
XRT **-**Xilinx Runtime Library

Βιβλιογραφία

[1] HPL-AI benchmark code reference implementation [https://bitbucket.org/icl/hpl-ai/src/main/].

[2] HPL-MxP Mixed-Precision Benchmark [https://hpl-mxp.org/].

[3] Innovative Computing Laboratory (ICL) HPL-AI Benchmark research [https://icl.utk.edu/hpl-ai].

[4] Netlib HPL Algorithm documentation and explanation [https://www.netlib.org/benchmark/hpl/algorithm.html].

[5] Wikipedia Iterative refinement [https://en.wikipedia.org/wiki/Iterative_refinement].

[6] Wikipedia Generalized Minimal Residual Method-GMRES [https://en.wikipedia.org/wiki/Generalized_minimal_residual_method].

[7] Wikipedia LU decomposition [https://en.wikipedia.org/wiki/LU_decomposition].

[8] Wikipedia FPGA definition and how it works [https://el.wikipedia.org/wiki/FPGA].

[9] Intel VTune Profiler [https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html].

[10] AMD Xilinx Alveo U200 Data Center Accelerator Card [https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#overview].

[11] AMD Xilinx OpenCL framework documentation [https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/pet1504034296131.html].

[12] AMD Xilinx Vitis application acceleration documentation [https://docs.xilinx.com/v/u/2019.2-English/ug1393-vitis-application-acceleration].

[13] AMD Xilinx Vivado Design Suite User Guide High-Level Synthesis [https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2020_2/ug902-vivado-high-level-synthesis.pdf].