Embedded Systems

Term Project

Multi Core Bare metal Matrix Multiplication

using Xilinx Zynq 7020

Μεταπτυχιακός Φοιτητής

Σιγανός Νικόλαος (mtp194)

Ιούνιος 2019

Περιεχόμενα

Πίνακας Εικόνων

Εισαγωγή

Τα ενσωματωμένα σύστημα είναι υπολογιστικά συστήματα βασισμένα σε μικροεπεξεργαστές και συνήθως αποτελούν τμήμα ενός μεγαλύτερου ηλεκτρονικού ή ηλεκτρομηχανικού συστήματος. Είναι ένας συνδυασμός υλικού και λογισμικού που σκοπό έχουν να επιτελέσουν με ακρίβεια συγκεκριμένη λειτουργία. Χαρακτηριστικά τους είναι η ταχύτητα, το μέγεθός τους, η κατανάλωση ενέργειας, η ακρίβεια, η αξιοπιστία και η ασφάλεια. Ευρεία χρήση ενσωματωμένων συστημάτων έχουμε στην αυτοκινητοβιομηχανία, στην βιομηχανική παραγωγή, στον τομέα της υγείας, τον στρατιωτικό τομέα, στον τομέα της αεροναυπηγικής κ.α.

Ο τομέας των ενσωματωμένων συστημάτων βρίσκεται σε σταθερή ανάπτυξη τις τελευταίες δεκαετίες και οι προβλέψεις δείχνουν ότι είναι ένας τομέας που διαθέτει ιδιαίτερη δυναμική.

Καθώς στα ενσωματωμένα συστήματα έχει ιδιαίτερη βαρύτητα η κατανάλωση ενέργειας και το μέγεθος, έχουμε υλοποίησή τους, με την χρήση ειδικά σχεδιασμένων κυκλωμάτων, των system on a chip (SoC). Ένα SoC συνήθως περιλαμβάνει μια κεντρική μονάδα επεξεργασίας, μνήμη, θύρες εισόδου – εξόδου, δευτερεύουσα μνήμη κ.α. σε ένα ολοκληρωμένο κύκλωμα μεγέθους ενός κέρματος. Ανάλογα με την εφαρμογή μπορεί να περιλαμβάνει και λειτουργίες επεξεργασίας ψηφιακού σήματος, αναλογικού σήματος ή και ραδιοσυχνοτήτων. Ένα SoC μπορεί να περιλαμβάνει ένα μικροελεγκτή ή ένα μικροεπεξεργαστή και σύνθετα περιφερειακά, όπως επεξεργαστή γραφικών, συνεπεξεργαστές και υπομονάδες επικοινωνίας.

Τα FPGAs (Field Programmable Gate Array) είναι ολοκληρωμένα κυκλώματα (Integrated Circuit), τα οποία επιτρέπουν στον χρήστη να σχεδιάσει ένα ολοκληρωμένο σύστημα, χρησιμοποιώντας την κατάλληλη γλώσσα προγραμματισμού. Σε αντίθεση με τα Application Specific ICs (ASIC) τα FPGAs είναι επαναπρογραμματιζόμενα. Αυτό το χαρακτηριστικό τους τα καθιστά ιδανικά για εφαρμογές που δεν έχουν οριστικοποιηθεί και οι προδιαγραφές τους μεταβάλλονται. Το κόστος τους σε σύγκριση με την παραγωγή ASIC είναι ελάχιστο. Έτσι τα FPGAs χρησιμοποιούνται ευρέως για την δημιουργία προτύπων και την επικύρωση σχεδίων σύνθετων λογικών κυκλωμάτων. Επίσης, είναι κατάλληλα για εφαρμογές όπου το τρέχον σχέδιο μπορεί να χρήζει αναβάθμισης ή τροποποίησης μετά την παράδοσή του. Το FPGA αποτελείται από Blocks προγραμματιζόμενης λογικής (logic blocks) και προγραμματιζόμενες διασυνδέσεις (interconntects). Εκτός από αυτά τα στοιχεία, όλα τα FPGAs έχουν μνήμες RAM και διαφορετικό πλήθος και είδος από εισόδους-εξόδους (IOs), καθώς και διάφορους μετατροπείς αναλογικού-ψηφιακού (ADC, DAC).

Στην παρούσα εργασία θα χρησιμοποιήσουμε ένα σύνθετο υλικό, έναν all-programmable SoC, της εταιρίας Xilinx, το Zynq 7020. Μέλος της οικογένειας Zynq®-7000 (AP SoC), συνδυάζει τη δυνατότητα προγραμματισμού επεξεργαστών ARM®, με τη δυνατότητα προγραμματισμού λογικών κυκλωμάτων ενός FGPA, ενσωματώνοντας CPU, digital signal processors (DSP) και application–specific standard parts (ASSP) σε ένα μόνο IC. Με την χρήση του hardware που ενσωματώνει ένας Zynq 7020 και των παρεχόμενων από την εταιρία Xilinx εργαλείων, θα υλοποιήσουμε διαφορετικά σενάρια χρήσης πολλαπλών επεξεργαστών, ώστε να εξετάσουμε την χρονική απόδοσή τους, σε πολλαπλασιασμό πινάκων.

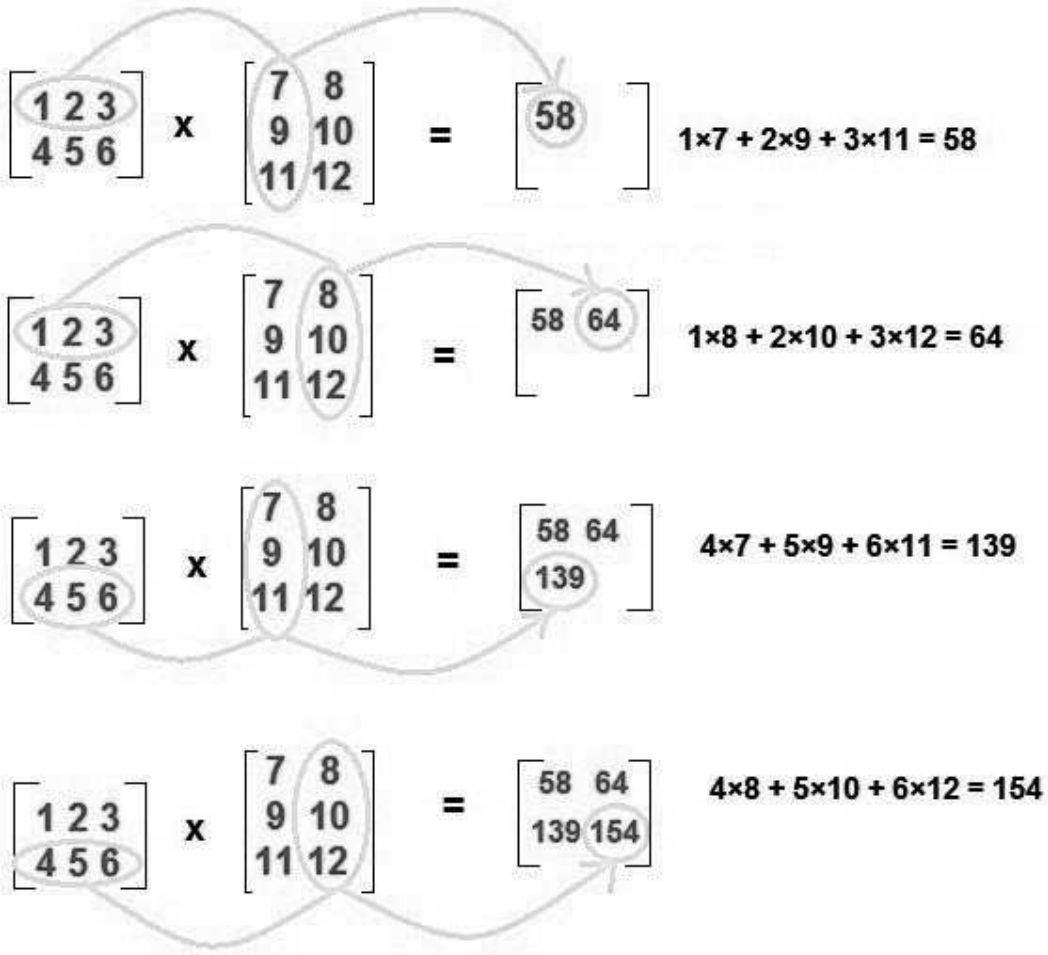
Πολλαπλασιασμός Πινάκων

Ο πολλαπλασιασμός πινάκων είναι ένα θεμελιώδες εργαλείο της γραμμικής άλγεβρας που έχει πλήθος εφαρμογών στην στατιστική, τα μαθηματικά, τα οικονομικά και την μηχανική.

Εάν το Α είναι ένας μ-ν πίνακας και Β είναι ένας ν-ρ πίνακας, τότε το γινόμενο του πίνακα ΑΒ είναι ο μ-ρ πίνακας του οποίου τα στοιχεία δίνονται από το γινόμενο της αντίστοιχης σειράς του Α και της αντίστοιχης στήλης Β:

$$[\mathbf{AB}]_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \cdots + A_{i,n}B_{n,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j},$$

Σχηματικά ο πολλαπλασιασμός πινάκων φαίνεται στην παρακάτω εικόνα



Εικόνα 1: Πολλαπλασιασμός πινάκων

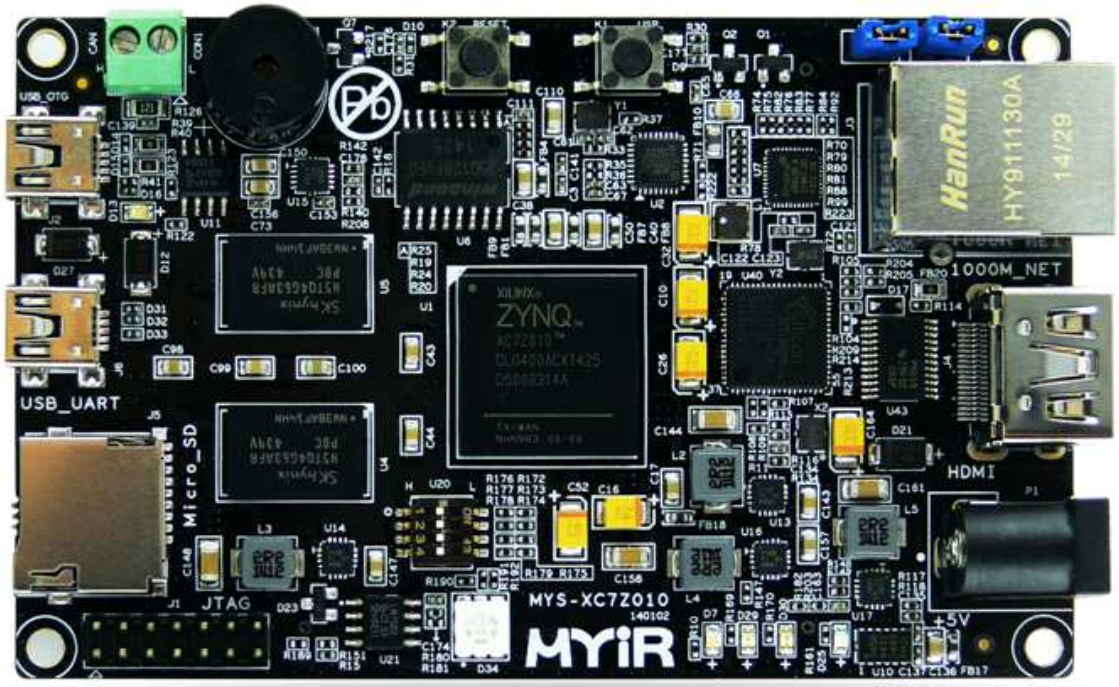
Κάθε στοιχείο του πίνακα είναι ένα άθροισμα γινομένων. Πιο συγκεκριμένα, υπολογίζεται από *n* πολλαπλασιασμούς και *n-1* προσθέσεις. Συνεπώς προκύπτει ότι για το σύνολο των στοιχείων του πίνακα, έχουμε n^3 πολλαπλασιασμούς και $n^2(n-1)$ προσθέσεις.
Η πολυπλοκότητα του αλγορίθμου του πολλαπλασιασμού πινάκων, όπως προκύπτει από τον ορισμό του, είναι **O(n^3)** για πίνακες διαστάσεων nxn. Σημειώνεται ότι υπάρχουν και διαφορετικοί αλγόριθμοι με άλλη πολυπλοκότητα.

Περιγραφή αλγορίθμου:

```
Input: matrices A and B
Let C be a new matrix of the appropriate size
For i from 1 to n:
        For j from 1 to p:
                Let sum = 0
                For k from 1 to m:
                        Set sum ← sum + Aik × Bkj
                Set Cij ← sum
Return C
```

Zynq 7020 All Programmable System-on-Chip

Όπως αναφέραμε στην εισαγωγή, οι μετρήσεις μας έγιναν με την χρήση του Zynq-7020 (XC7Z020) All Programmable System-on-Chip (SoC). Πιο συγκεκριμένα χρησιμοποιήσαμε ένα high-performance Single Board Computer (SBC), χαμηλού κόστους, το Z-turn Board (MYS-7Z020) το οποίο φαίνεται στην Εικόνα 2.



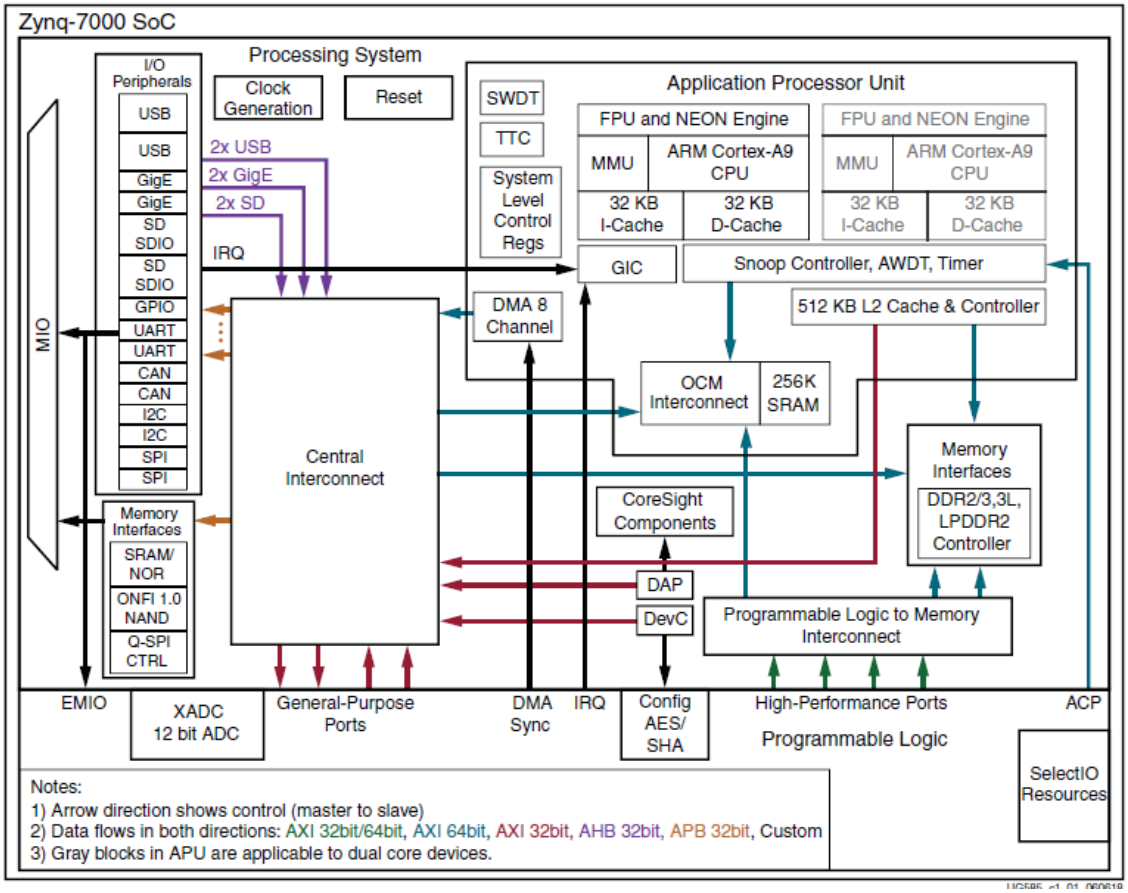
Εικόνα 2: Z-turn board

Οι επεξεργαστές ARM Cortex A9 είναι πολυπύρηνοι επεξεργαστές 32- bit που υλοποιούν την αρχιτεκτονική ARMv7-A. Η υλοποίησή του, από την Xilinx στον XC7Z020 είναι στα 28nm και περιλαμβάνουν NEON SIMD και Vector Floating Point Unit. Η συνύπαρξη του Programmable System (PS) των επεξεργαστών ARM και της Programmable Logic (PL) του FPGA, παρέχει επεξεργαστική ισχύ και ευελιξία σε χαμηλό κόστος και καθιστούν το Zynq 7020 ιδανικό για μεγάλο πλήθος εφαρμογών.

Συνοπτικά τα κύρια χαρακτηριστικά του Zynq-7020 (XC7Z020-1CLG400C)

- Up to 667MHz ARM® dual-core Cortex™-A9 MPCore processor
- Integrated Artix-7 class FPGA subsystem with with 85K logic cells, 53,200 LUTs, 220 DSP slices
- NEON™ & Single / Double Precision Floating Point for each processor
- Supports a Variety of Static and Dynamic Memory Interfaces

Ακολουθεί το μπλοκ διάγραμμα των Zynq.



Εικόνα 3: Μπλοκ διάγραμμα Zynq

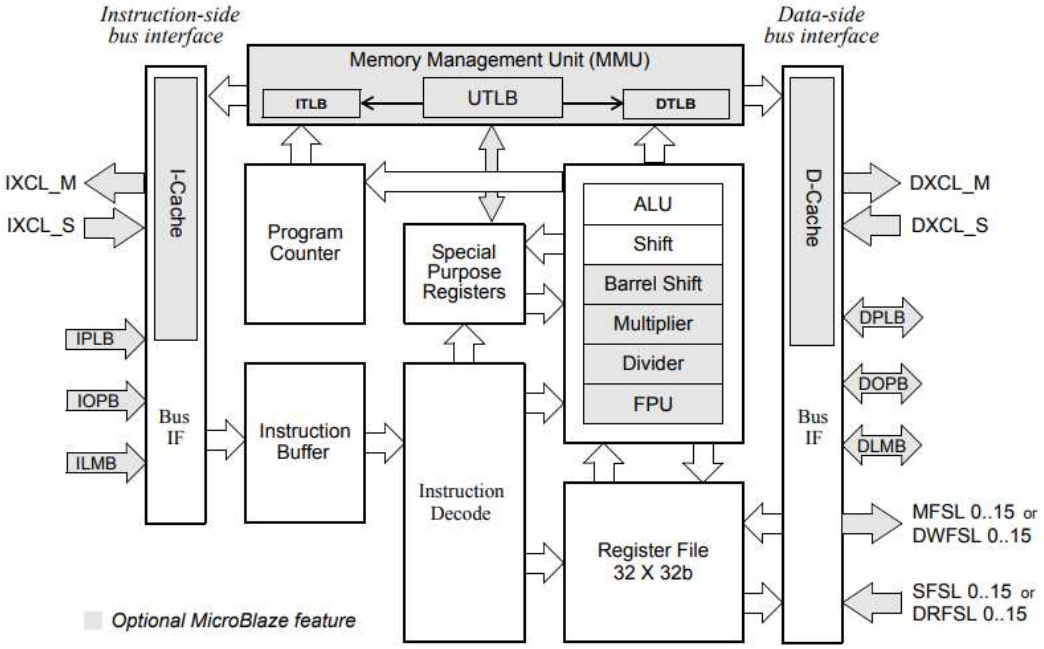
Το PL μπλοκ περιλαμβάνει διάφορες υπομονάδες. Σε αυτές συμπεριλαμβάνονται εκτός των παραμετροποιήσιμων λογικών μπλοκ (CLBs), τα παρακάτω:

- ρυθμιζόμενου εύρους block RAM (BRAM),
- DSP slices με ένα πολλαπλασιαστή 25 x 18,
- 48-bit accumulator και pre-adder (DSP48E1),
- παραμετροποιήσιμο από τον χρήστη analog to digital convertor (XADC),
- clock management tiles (CMT),
- ένα ρυθμιζόμενο μπλοκ για 256b AES decryption και SHA authentication,
- configurable SelectIO™ technology
- GTP or GTX multi-gigabit transceivers (προαιρετικό)
- ενσωματωμένο PCI Express® (PCIe) μπλοκ.

Τα PS και PL μπορεί να είναι tightly ή loosely συνδεδεμένα, χρησιμοποιώντας πολλά Interfaces και σήματα, που συνδυάζονται σε περισσότερες από 3000 συνδέσεις. Αυτό επιτρέπει στον χρήστη να δημιουργεί επιταχυντές υλικού (accelerators) ή άλλες λειτουργίες στην PL logic, οι οποίες έχουν πρόσβαση στην μνήμη του PS και είναι προσβάσιμες από τους επεξεργαστές. Τα περιφερειακά PS I/O, συμπεριλαμβανομένων και των static/flash memory interfaces μοιράζονται τα πολυπλεγμένα I/O (MIO) μέσω των 54 MIO pins. Επιπλέον μέσω των extended multiplexed I/O interface ,υπάρχει η δυνατότητα πολλά από τα I/O του PL τομέα να χρησιμοποιούνται από τα I/O peripherals του τομέα PS.
Το σύστημα περιλαμβάνει πολλές λειτουργίες ασφάλειας, δοκιμών και αποσφαλμάτωσης. Ο Zynq 7020 μπορεί να εκκινήσει με ασφάλεια ή χωρίς. Επίσης οι ρυθμίσεις bitstream της PL μπορούν να μεταφερθούν με ασφάλεια ή χωρίς. Οι λειτουργίες ασφαλείας χρησιμοποιούν τα μπλοκ 256b AES decryption και SHA authentication τα οποία είναι τμήματα της PL. Για την χρήση αυτών των λειτουργιών ο τομέας PL πρέπει να είναι σε χρήση.
Οι τομείς PL και PS έχουν αυτόνομες διαφορετικές τροφοδοσίες και έτσι ο τομέας PL μπορεί να παραμένει κλειστός για εξοικονόμηση ενέργειας. Αντιστοίχως, ο τομέας PS μπορεί να ρυθμιστεί επιπλέον για εξοικονόμηση ενέργειας.
Στο υποσύστημα FPGA μπορούν να προστεθούν soft processor MicroBlaze (uBlaze). Οι MicroBlazes είναι μια γενιά 32-bit RISC Soft Processor Core, οι οποίοι μπορούν να προστεθούν εύκολα με τη χρήση του (βασισμένου στο Eclipse) Xilinx Software Development Kit. Το Artix-7 FPGA που εμπεριέχεται στο Zynq-7020 παρέχει την δυνατότητα να υλοποιήσουμε μέχρι 4 uBlaze.
Οι επεξεργαστές MicroBlaze συνήθως χρησιμοποιούνται σε μια από τις τρείς προεπιλεγμένες επιλογές:

- ως απλός μικροελεγκτής για την εκτέλεση κάποιας bare metal εφαρμογής,
- ως επεξεργαστής πραγματικού χρόνου με προστασία μνήμης και χρήση cache με τη χρήση FreeRTOS,
- ως επεξεργαστής εφαρμογών με διαχείρηση μνήμης τρέχοντας Linux.

Το μπλοκ διάγραμμα ενός MicroBlaze παρουσιάζεται στο παρακάτω σχήμα:

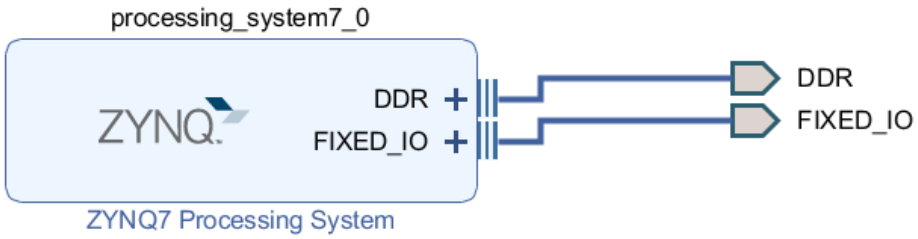


Εικόνα 4: Μπλοκ διάγραμμα microBlaze

Block Designs

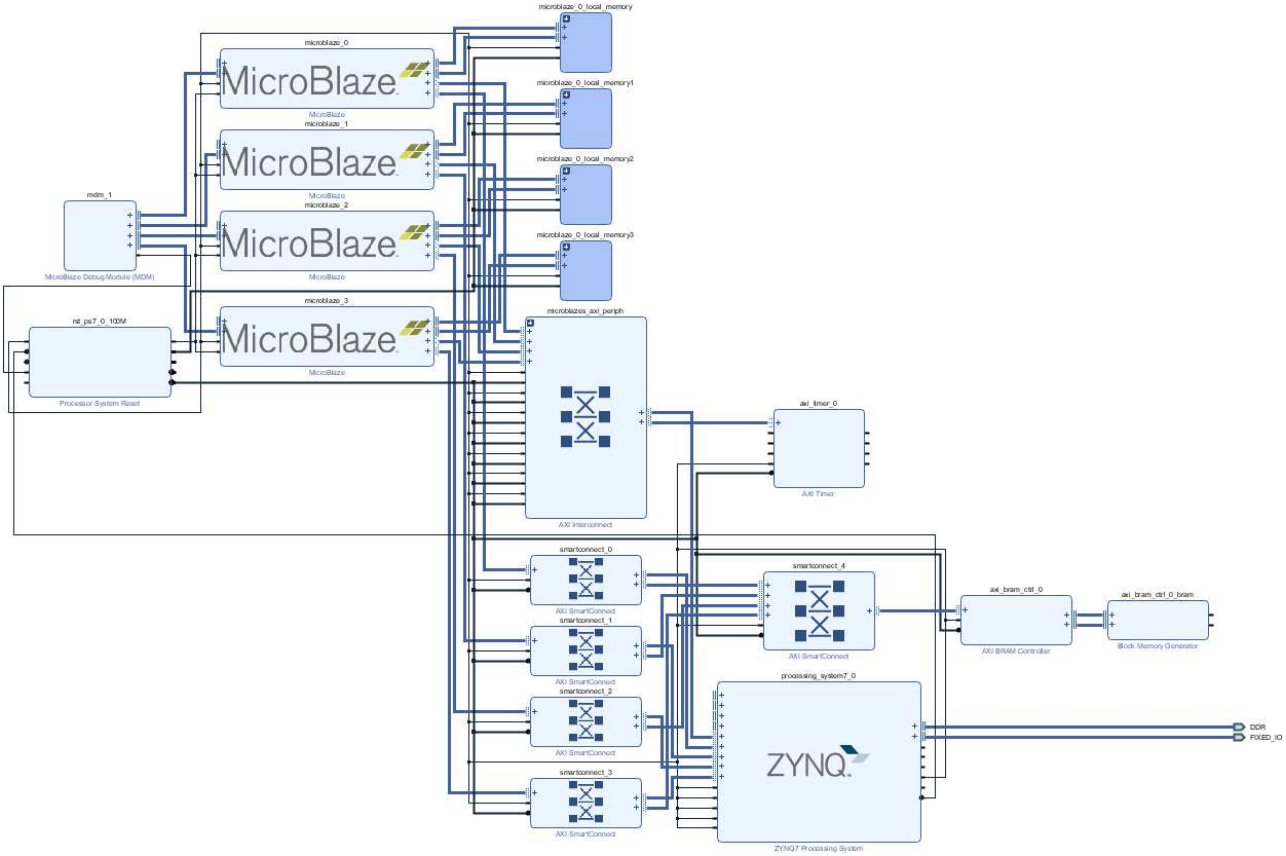
Ο προγραμματισμός του Z-turn έγινε σε περιβάλλον Xilinx Vivado 2018.3, με τη χρήση ενός Xilinx Platform Cable USB Download Cable JTAG Programmer for FPGA CPLD.

Για την πραγματοποίηση των μετρήσεων υλοποιήσαμε 2 διαφορετικά design. Ένα απλό που περιλαμβάνει το Zynq 7020 με τις ελάχιστες συνδέσεις όπως φαίνεται παρακάτω και στο οποίο έγιναν οι μετρήσεις με 1 και 2 πυρήνες.



Εικόνα 5: Απλό Zynq block design

Ένα σύνθετο design, που περιλαμβάνει επεξεργαστές 4 uBlaze επιπλέον, το οποίο φαίνεται στο παρακάτω σχήμα και έγινε με την βοήθεια του ISCA Lab. Στο design αυτό, έγιναν οι μετρήσεις με περισσότερους πυρήνες.



Εικόνα 6: Σύνθετο block design με 4 uBlaze

Περιγραφή μετρήσεων

Οι μετρήσεις μας αφορούν πολλαπλασιασμούς πινάκων τετραψήφιων ακέραιων αριθμών και έγιναν για τετραγωνικούς πινάκες με διαστάσεις 16x16 και 32x32.
Μετρήσεις έγιναν για τους 2 πίνακες για τον παρακάτω αριθμό πυρήνων:

Cores used
Single A9 Core with Cache
Single A9 Core
Dual A9 Core
Six Cores (2x A9 και 4xuBlaze)

Η χρήση της μνήμης cache επηρεάζει την χρονική επίδοση των επεξεργαστών και για τον λόγο αυτό έγιναν μετρήσεις με ενεργοποιημένη την cache για τις μετρήσεις με Single A9 Core. Σημειώνεται, ότι χρήση της cache δεν επιτρέπει την επικοινωνία των Cortex A9, μέσω της κοινόχρηστης μνήμης DDR, ώστε να κάνουμε πολλαπλασιασμό με 2 πυρήνες.

Επιπλέον, για πίνακες διαστάσεων 32x32 έγιναν μετρήσεις για:

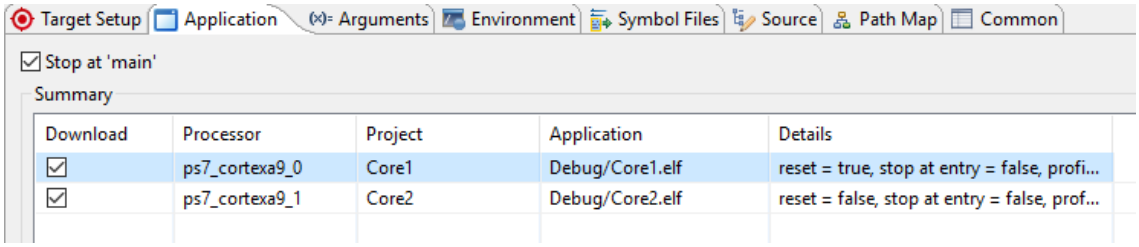
Cores used
Four uBlaze Cores with cache
Six Cores (2x A9 Cores και 4xuBlaze with Cache)

Διαδικασία μετρήσεων

Για την πραγματοποίηση μετρήσεων με ακρίβεια ακολουθούμε την παρακάτω διαδικασία:

Ο επεξεργαστής A9, Core 0 δημιουργεί τους πίνακες και καταχωρεί σε αυτούς τυχαίες τιμές. Αυτός είναι και ο επεξεργαστής, ο οποίος συγχρονίζει τους υπόλοιπους (4 uBase και Core 1) και καταγράφει τον χρόνο (σε κύκλους του) που απαιτείται για την ολοκλήρωση της διαδικασίας.

Οι επεξεργαστές εκκινούν και φορτώνουν τον κώδικα.



Εικόνα 7: Run configuration Vivado SDK - Εκκίνηση dual core application

(Σημειώνεται ότι για την εκτέλεση κώδικα από πολλούς επεξεργαστές, πρέπει να έχει παραμετροποιηθεί κατάλληλα το αρχείο linker script (lscript.id), ώστε να μην υπάρχουν επικαλύψεις στις θέσεις μνήμης που θα τοποθετηθεί ο κώδικας διαφορετικών επεξεργαστών)
Μόλις αρχίσουν την εκτέλεση του κώδικά τους, δίνουν τιμή σε ένα flag “Ready” (διαφορετικό σε κάθε επεξεργαστή) για να δηλώσουν ότι είναι έτοιμοι, να διαβάσουν τιμές για να εκτελέσουν τους πολλαπλασιασμούς και περιμένουν τον Core 0 να τους δώσει σήμα να ξεκινήσουν.

Ο Core 0 περιμένει να αλλάξουν τα ανωτέρω flags όλων των επεξεργαστών.
Μόλις γίνει αυτό, αλλάζει ένα δικό του flag “Start”, το οποίο το διαβάζουν όλοι οι επεξεργαστές με το οποίο δηλώνει:

- ότι τα data είναι έτοιμα

- όλοι οι επεξεργαστές είναι έτοιμοι να εκτελέσουν πράξεις

- να γίνει εκκίνηση των πράξεων από τον κάθε ένα

Παράλληλα ο Core 0 ενεργοποιεί τον timer και εκτελεί τους πολλαπλασιασμούς που του έχουν ανατεθεί.

```
// start of Timer
XTime_GetTime(&tStart);
time0 = XScuTimer_GetCounterValue(TimerInstancePtr);
*syncBarrier = 9999;
while (*syncBarrier2!=9998){
}
XScuTimer_Start(TimerInstancePtr);
```

Εικόνα 8: Συγχρονισμός πυρήνων (απόσπασμα κώδικα Core 0)

Όπως φαίνεται στο παραπάνω απόσπασμα κώδικα, ο Core 0 θέτει μια τιμή στον pointer *syncBarrier,* ενημερώνοντας τον Core1 (στην περίπτωση αυτή), ότι τα data είναι έτοιμα και στη συνέχεια μπαίνει σε ατέρμονο βρόχο, μέχρι να ενημερωθεί από τον Core1 -μέσω του pointer *syncBarrier2*- ότι και αυτός είναι έτοιμος, ώστε να ξεκινήσει τον counter.
Να σημειωθεί ότι το κώδικας είναι σε γλώσσα C.

Κάθε επεξεργαστής διαβάζει από την DDR, τις γραμμές του πρώτου πίνακα που του έχουν ανατεθεί, εκτελεί τις πράξεις και καταγράφει τα αποτελέσματα σε περιοχή μνήμης της DDR, που έχει οριστεί για τη σύνθεση του πίνακα των αποτελεσμάτων.

Μόλις κάθε επεξεργαστής ολοκληρώσει τις πράξεις που του έχουν ανατεθεί, αλλάζει ένα δικό του flag “Fin” για να δηλώσει στον Core 0, ότι έχει τελειώσει το έργο του.

Ο Core 0 μόλις διαβάσει τα ανωτέρω flags, όλων των επεξεργαστών, ότι έχουν ολοκληρώσει, (και εφόσον έχει ολοκληρώσει και ο ίδιος) σταματά την καταμέτρηση των κύκλων, τυπώνει τους αρχικούς πίνακες των δεδομένων, τον πίνακα των αποτελεσμάτων και των αριθμό των κύκλων.

Τέλος αρχικοποιεί τα flags “Ready”, “Start” και “Fin” ώστε να μην επηρεάζουν πιθανή επανεκτέλεση του προγράμματος.

Ο κώδικας που χρησιμοποιείται για την εκτέλεση των πολλαπλασιασμών είναι σχετικά απλός και για την περίπτωση που εκτελείται από έναν μόνο επεξεργαστή, είναι όπως παρακάτω:

```
for (c = 0; c < 16; c++) {
  for (d = 0; d < q; d++) {
    for (k = 0; k < p; k++) {
      sum = sum + first[c][k]*second[k][d];
    }
    multiply[c][d] = sum;
    sum = 0;
  }
}
```

Εικόνα 9: Κώδικας πολλαπλασιασμού (16x16 Single Core)

Για την εκτέλεση του πολλαπλασιασμού με τη συνδρομή πολλών επεξεργαστών πρέπει να γίνει επιμερισμός του έργου. Στην απλή περίπτωση των 2 επεξεργαστών ίδιας ταχύτητας κάνουμε ισοκατανομή του έργου, μοιράζοντας τον αριθμό των γραμμών που θα επεξεργαστεί ο κάθε ένας, τροποποιώντας τον κώδικά τους. Για την πρόσβαση του 2ου core στα δεδομένα που δημιούργησε ο Core 0, χρησιμοποιούμε pointers. Έτσι, σε αυτή την περίπτωση, ο κώδικας πολλαπλασιασμού στους 2 επεξεργαστές, διαμορφώνεται, όπως φαίνεται παρακάτω :

```
// First iteration stops at 8 so it reads the 1st half
for (c = 0; c < 8; c++) {
  for (d = 0; d < q; d++) {
    for (k = 0; k < p; k++) {
      sum = sum + first[c][k]*second[k][d];
    }
    multiply[c][d] = sum;
    sum = 0;
  }
}
```

```
// First iteration start at 128 byte so it reads the 2nd
m=128;
for (k=m;k<256;k+=16){
    for (j=0;j<16;j++){
        for (i=0;i<16;i++){
            sum+=(*(first+i+k))*(*(second+i*16+j));
        }
        *(result+m)=sum;
        m++;
        sum=0;
    }
}
```

Εικόνα 10: Κώδικας πολλαπλασιασμού με επιμερισμό γραμμών σε Core0 και Core1 (16x16 Dual Core)

(πλήρης κώδικας για 2 πυρήνες υπάρχει στο ΠΑΡΑΡΤΗΜΑ Ι)

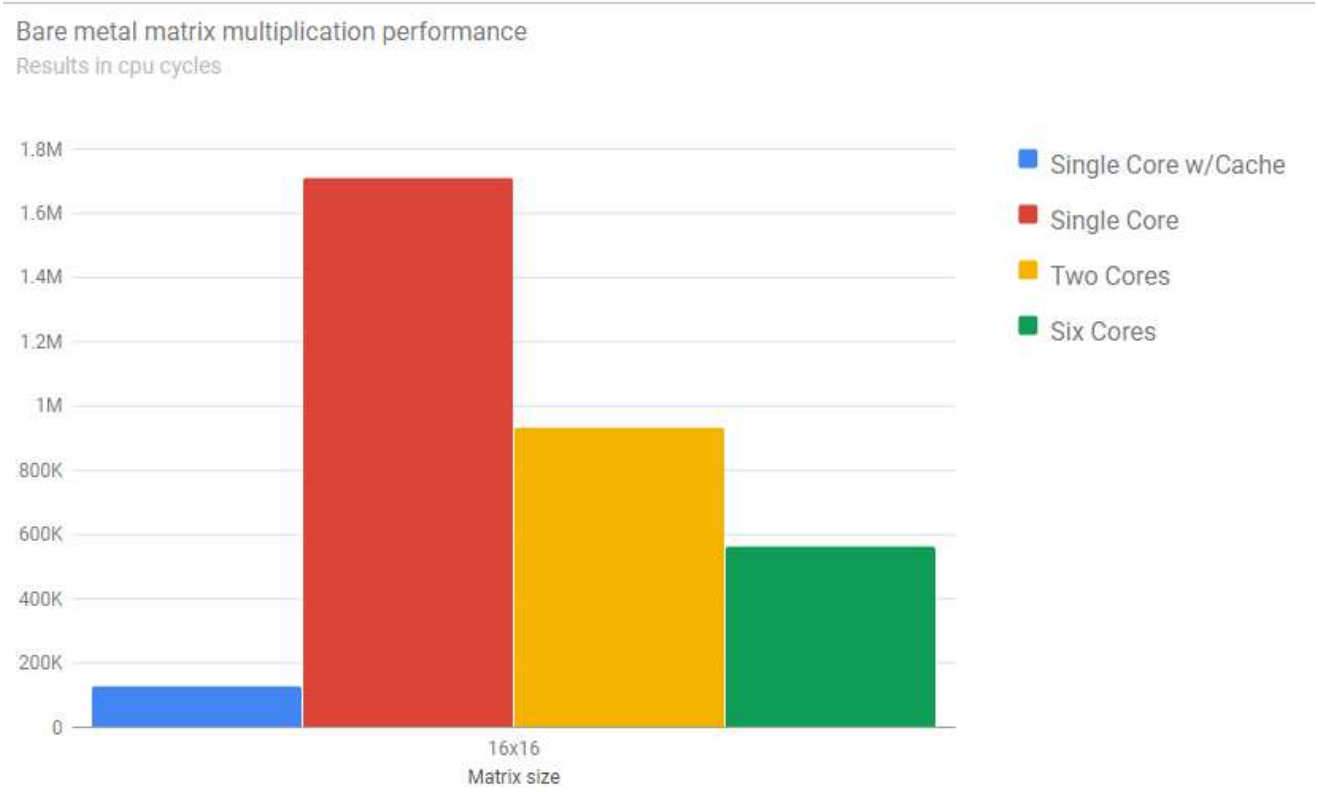
Στην πιο σύνθεση περίπτωση που έχουμε επεξεργαστές διαφορετικής ταχύτητας, πρέπει να γίνει ετεροκατανομή του έργου με στόχο το βέλτιστο αποτέλεσμα. Η γενική αρχή είναι ότι πρώτα βρίσκουμε τον χρόνο που απαιτείται από κάθε επεξεργαστή για την εκτέλεση μιας μονάδας έργου (πολλαπλασιασμός μιας γραμμής με μια στήλη) και στη συνέχεια κατανέμουμε το έργο, με στόχο ο συνολικός χρόνος εκτέλεσης να είναι το ο μικρότερος δυνατός. Για την περίπτωση των δυο επεξεργαστών με διαφορετικές ταχύτητες, κατανέμουμε το έργο αντίστροφος ανάλογα του χρόνου αυτού (κάνοντας τις απαραίτητες στρογγυλοποιήσεις).
Έτσι για χρήση των έξι πυρήνων, βέλτιστη κατανομή είναι ο επιμερισμός των γραμμών με αναλογία 2 προς 1, οπότε για πίνακα 32x32 δίνουμε 8 γραμμές στους A9 και 4 γραμμές στους uBlaze. Με την χρήση της cache των uBlaze, καθώς αυξάνεται η ταχύτητα τους βέλτιστη κατανομή είναι, 6 γραμμές στους A9 και 5 γραμμές στους uBlaze (βλέπε Εικόνα 14).
Η cache memory που έχει ενεργοποιηθεί σε όλους τους uBlaze είναι data cache και έχει ρυθμιστεί σε μέγεθος 1kB. Η ενεργοποίηση της cache στους uBlaze έγινε με την χρήση της εντολής enable_caches().

Αποτελέσματα

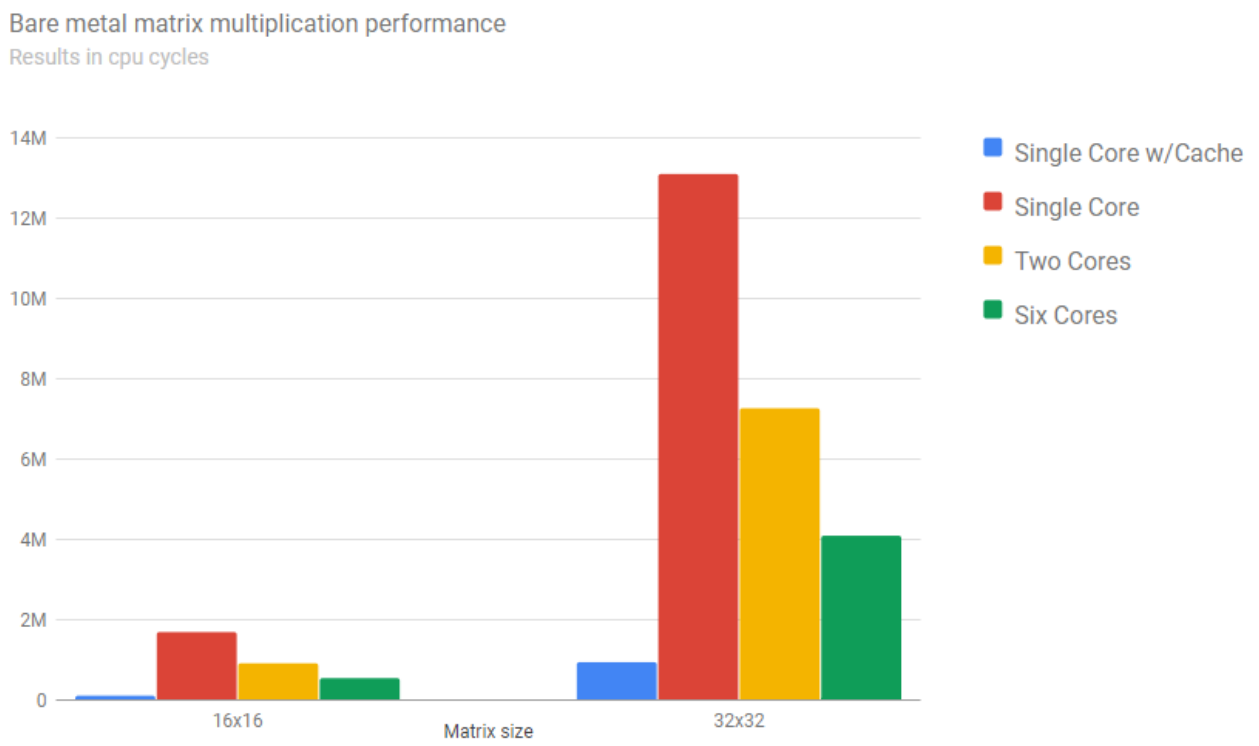
Ακολουθούν πίνακες μετρήσεων και διαγράμματα που προκύπτουν.

Cores used	Matrix 16x16		Matrix 32x32	
	Αριθμός κύκλων Core A9	% μεταβολή του χρόνου εκτέλεσης	Αριθμός κύκλων Core A9	% μεταβολή του χρόνου εκτέλεσης
Single Core with Cache	130030	-	964544	-
Single Core without Cache	1712524	0	13120718	0
Dual Core	935030	-45,4 %	7285932	-44,5 %
Six Cores (2 A9 & 4 MicroBlaze)	565194	-66,9 %	4112298	-68.5 %

(τιμή αναφοράς Single Core without Cache)

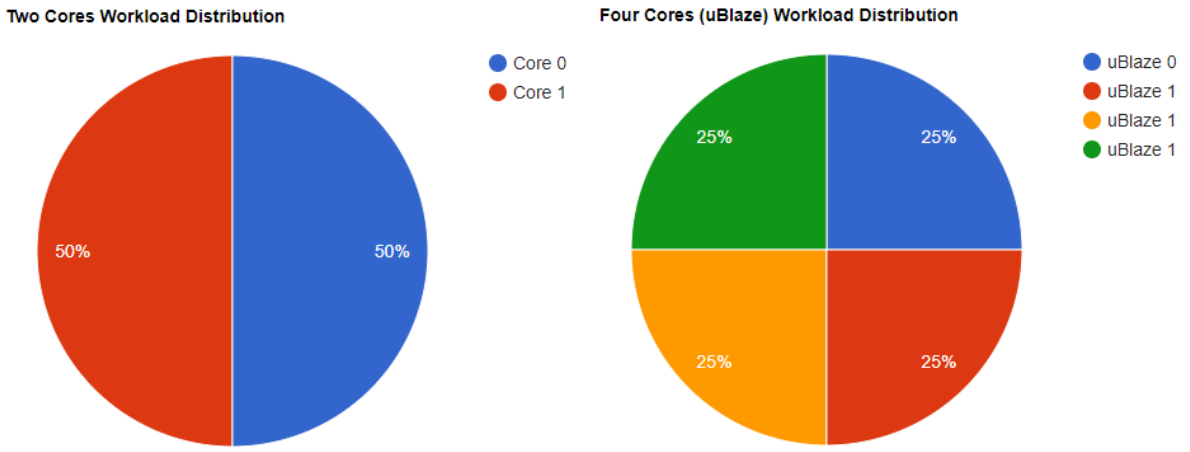


Εικόνα 11: Συγκριτικό διάγραμμα απόδοσης για 1,2 και 6 πυρήνες

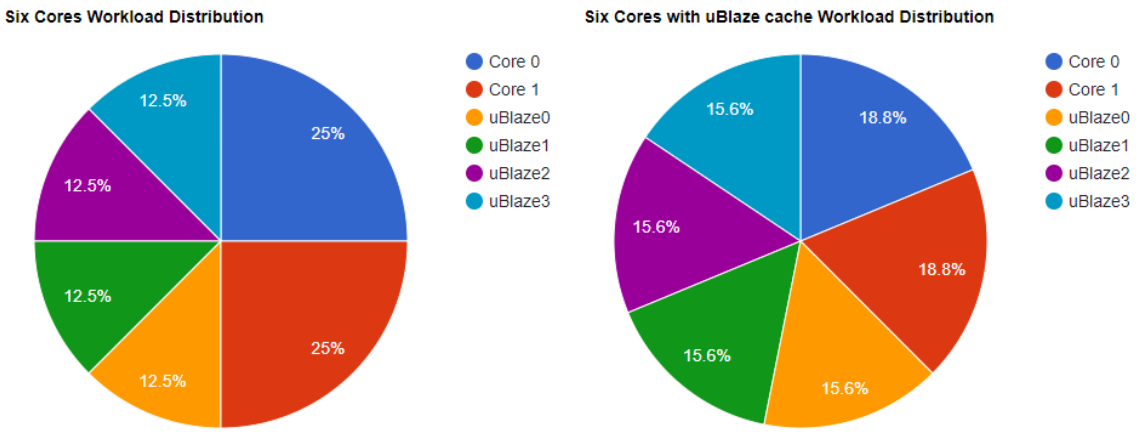


Εικόνα 12: Σύγκριση διαγραμμάτων απόδοσης για πίνακες 16x16 και 32x32

Όπως προαναφέραμε, για τον πολλαπλασιασμό πινάκων 32x32 έγιναν και μετρήσεις με χρήση cache στους uBlaze. Η κατανομή του φόρτου εργασίας μεταξύ των πυρήνων, φαίνεται στα διαγράμματα που ακολουθούν.



Εικόνα 13: Ποσοστιαίο διάγραμμα καταμερισμού γραμμών σε 2 και 4 πυρήνες

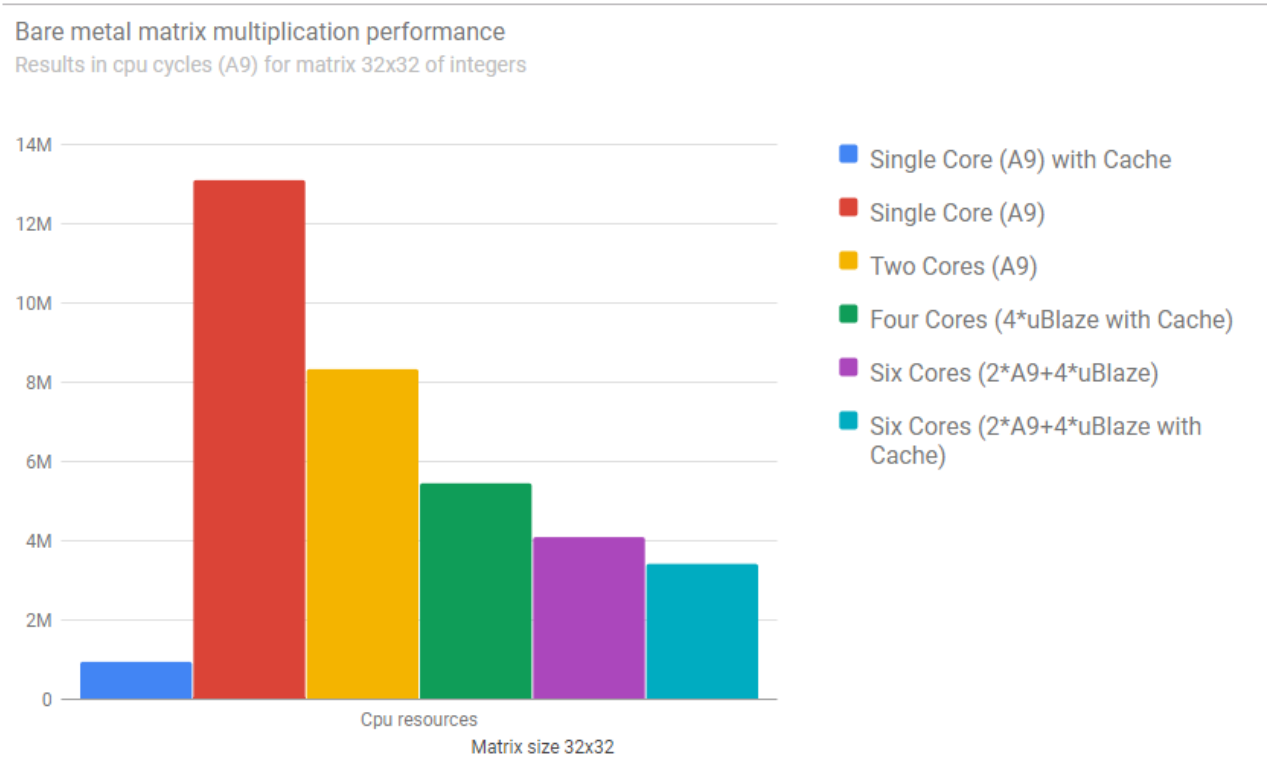


Εικόνα 14: Ποσοστιαίο διάγραμμα καταμερισμού γραμμών για 6 πυρήνες, χωρίς χρήση και με χρήση cache στους uBlaze

Μετρήσεις με χρήση πολλαπλών πύρινων για πίνακες 32x32.

Cores used	Matrix 32x32	
	Αριθμός κύκλων Core A9	% μεταβολή του χρόνου εκτέλεσης
Single Core with Cache	964544	-
Single Core without Cache	13120718	0
Dual Core	7285932	-44,5 %
Four MicroBlaze with Cache	5470084	-58,31%
Six Cores (2 A9 & 4 MicroBlaze)	4112298	-68.5 %
Six Cores (2 A9 & 4 MicroBlaze with cache)	3437410	-73,8%

(τιμή αναφοράς Single Core without Cache)



Εικόνα 15: Συγκριτικό διάγραμμα απόδοσης για πίνακα 32x32

Συμπεράσματα

Από τις μετρήσεις μας προκύπτει ότι η χρήση της cache στον A9, έχει ως αποτέλεσμα την εκτέλεση του κώδικα 13 φορές ταχύτερα από την ταχύτητα του ίδιου επεξεργαστή, χωρίς την χρήση της. Το αποτέλεσμα αυτό είναι το ίδιο για τα δύο μεγέθη πίνακα, για τους οποίους κάναμε μετρήσεις και είναι η καλύτερη επίδοση που μπορούμε να έχουμε. Η μέτρηση όμως αυτή αποτελεί μια ιδανική περίπτωση καθώς ο κώδικάς μας έχει αποκλειστική χρήση ολόκληρης της cache. Σημειώνουμε ότι το μέγεθος της cache είναι των Cortex A9 είναι L1: 32KB Instruction και 32 KB Data per processor και L2: 512KB.

Η χρήση της cache έχει μειονεκτήματα, στα οποία αναφερόμαστε παρακάτω:

- Αν ο πολλαπλασιασμός αποτελεί μέρος ενός μεγαλύτερου προγράμματος και χρήση της cache γίνεται και από άλλα τμήματα του κώδικα, η χρήση της στο τμήμα του πολλαπλασιασμού δεν θα είναι τόσο αποδοτική, όσο αυτή που καταγράψαμε.
- Το ίδιο ισχύει και για πολλαπλασιασμό πινάκων μεγάλων διαστάσεων, για τους οποίους η cache δεν μπορεί να αποθηκεύσει όλα τα προς χρήση δεδομένα.
- Η χρήση της cache μπορεί να προκαλέσει «αστάθεια» στον χρόνο εκτέλεσης κάτι το οποίο μπορεί να μην είναι αποδεκτό από τις προδιαγραφές της εφαρμογής μας.
- Μας αφαιρεί τη δυνατότητα επικοινωνίας με το άλλο A9 Core μέσω της DDR.

Όπως είδαμε, η πολυπλοκότητα του αλγόριθμου είναι $O(n^3)$. Έτσι για πίνακες διπλάσιου μεγέθους θα έχουμε διαφορά στον χρόνο εκτέλεσής τους κατά $2^3 = 8$ φορές.
Συγκρίνοντας τα αποτελέσματα των μετρήσεων για τα 2 διαφορετικά μεγέθη πινάκων, καταγράφουμε ότι ο πολλαπλασιασμό του πίνακα 16x16, εκτελείται περίπου 7.7 φορές γρηγορότερα από τον 32x32 (βλέπε Εικόνα 12), κάτι που είναι συμβατό με τη θεωρητική προσέγγιση. Οι μετρήσεις λοιπόν, έχουν μια σταθερή αναλογία και οι συγκρίσεις του χρόνου εκτέλεσης με διαφορετική σύνθεση υλικού, μας οδηγούν στα ίδια συμπεράσματα.

Με τη χρήση 2 επεξεργαστών έχουμε μείωση του χρόνου κατά 45%, αποτέλεσμα κοντά στο θεωρητικά βέλτιστο 50%. Με τη χρήση των 6 πυρήνων έχουμε μείωση του χρόνου περίπου κατά 68%, σε σχέση με τον χρόνο ενός A9. Έτσι για να έχουμε επιπλέον μείωση του χρόνου για περίπου 25% σε σχέση με τους 2 πυρήνες ή διατυπωμένο διαφορετικά, στο 1/3 περίπου του χρόνου ενός μόνο A9, πρέπει να διαθέσουμε σημαντικά αυξημένους πόρους.

Σχετικά με την σταθερότητα του χρόνου εκτέλεσης, παραθέτουμε μετρήσεις των uBlaze για το ίδιο έργο:

uBlaze_0: 4019876 clock cycles.
uBlaze_1: 4014282 clock cycles.
uBlaze_2: 4016796 clock cycles.
uBlaze_3: 4016190 clock cycles.

Παρατηρώντας την απόκλιση των μετρήσεων, είναι φανερή η σταθερότητα του χρόνου εκτέλεσης, χαρακτηριστικό πλεονέκτημα των εφαρμογών bare metal.

Για διάσταση πίνακα 32x32 (όπως προαναφέραμε, υπάρχει σταθερή αναλογία των μετρήσεων για διάσταση 16x16), κάναμε χρήση της data cache των uBlaze με μέγεθος ορισμένο στο 1 kB. Στην περίπτωση αυτή με τη χρήση των 6 πυρήνων, έχουμε το μεγαλύτερο ποσοστό μείωσης του χρόνου, περίπου στο 74%. Ενώ με τη χρήση μόνο των 4 uBlaze με cache, έχουμε μείωση 58% και αποτελεί μια σχετικά αποδοτική επιλογή καθώς αποδεσμεύει τους Cortex A9. Η μείωση του χρόνου είναι σημαντική αλλά όχι τόσο εντυπωσιακή όσο στην πρώτη μέτρηση (Single A9 Core with cache) καθώς το μέγεθος της cache είναι περιορισμένο. Περαιτέρω αύξηση της data cache και χρήση της instruction cache θα προκαλούσε επιπλέον μείωση του χρόνου.

Να σημειωθεί ότι τα αποτελέσματα των μετρήσεων, επηρεάζονται σημαντικά από τον τρόπο υλοποίησης του αλγόριθμου. Έτσι ο χρόνος για τον πολλαπλασιασμό 2 πυρήνων, με χρήση στον 2^{ο} Core, των εντολών ανάγνωσης και εγγραφής μνήμης, της Xilinx, Xil_In32() και Xil_Out32() (όπως προσπαθήσαμε αρχικά), είχε τελικό αποτέλεσμα, 2364286 κύκλους, ενώ με την χρήση pointer της γλώσσας C, 1712524 κύκλους.

Όπως είδαμε, υπάρχουν πολλά σενάρια χρήσης των διαθέσιμων πόρων, τα οποία επιτυγχάνουν διαφορετικά αποτελέσματα (Εικόνα 15), ενώ υπάρχουν πολλά ακόμη σενάρια, τα οποία δεν εξετάστηκαν. Οι απαιτήσεις της εφαρμογής και οι διαθέσιμοι πόροι, σε συνδυασμό με άλλες παραμέτρους, όπως ο συνολικός κώδικας της εφαρμογής, η κατανάλωση ενέργειας, ο τύπος των αριθμών κ.α. καθορίζουν την επιλογή της καταλληλότερης λύσης.

References

[1] D. A. Richie and J. A. Ross. “Opencl+ openshmem hybrid programming model for the adapteva epiphany architecture,” presented at the Third workshop on OpenSHMEM and Related Technologies, Baltimore, Maryland, Aug. 2-4, 2016.

[2] G. Kornaros, *Multi-core Embedded Systems*. Taylor and Francis Group, CRC Press, 2010.

[3] G. Kornaros and M. Coppola, "Enabling Efficient Job Dispatching in Accelerator-Extended Heterogeneous Systems with Unified Address Space," *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Lyon, France, 2018, pp. 180-188.
doi: 10.1109/CAHPC.2018.8645945

[4] Xilinx Inc., *Zynq-7000 All Programmable SoC: Embedded Design* Tutorial, 2015.1 ed, p. 14-41 [E-book], Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug1165-zynq-embedded-design-tutorial.pdf [Accessed: 2-MAY- 2019]

[5] Xilinx Inc., *Zynq-7000 SoC Technical Reference Manual*, v1.12.2, p.10-40 [E-book], Available: [Accessed: 25-APR-2019]
https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[6] Xilinx Inc., *Zynq Architecture,* 14.2, 2012, p. 1-40 [E-book], Available: http://www.ioe.nchu.edu.tw/Pic/CourseItem/4468_20_Zynq_Architecture.pdf
[Accessed: 2-MAY- 2019]

[7] Xilinx Inc., *Bare-Metal System Running on Both Cortex-A9 Processors*, 1.0.1 ed, 2014, p. 4-30 [E-book], Available: https://www.xilinx.com/support/documentation/application_notes/xapp1079-amp-bare-metal-cortex-a9.pdf. [Accessed: 25-APR-2019]

[8] Xilinx Inc., *Vivado Design Suite User Guide System-Level Design Entry*, 2017.2 ed, p.10-32 [E-book], Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug895-vivado-system-level-design-entry.pdf [Accessed: 28-APR-2019]

[9] Xilinx Inc., *MicroBlaze Processor Reference Guide*, 2017.2 ed, p.7 [E-book], Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug984-vivado-microblaze-ref.pdf

[10] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, *The Zynq Book Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC*, 1st ed, Glasgow, Scotland, UK: Strathclyde Academic Media, 2014, p 47-75 [E-book], Available: http://www.zynqbook.com/download-book.php [Accessed: 19-APR-2019]

[11] R. Kastner, J.Matai, and S.Neuendorer, *Parallel Programming for FPGAs*, 2018. p. 117-144 [E-book], Available: http://kastner.ucsd.edu/wp-content/uploads/2018/03/admin/pp4fpgas11.12.2018.pdf [Accessed: 25-MAY-2019]

ΠΑΡΑΡΤΗΜΑ Ι: Πλήρης κώδικας πολλαπλασιασμού για 2 πυρήνες

```
/*************************************************************************
****
* Core #1
*************************************************************************
***/
#include <stdio.h>
#include "sleep.h"
#include <time.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters_ps.h"
#include "xil_io.h"
#include "xtime_l.h"
#include "xil_cache.h"
#include "xscutimer.h"

#define TIMER_DEVICE_ID		XPAR_XSCUTIMER_0_DEVICE_ID
#define TIMER_LOAD_VALUE	0xFFFFFFFF

XScuTimer Timer;		/* Cortex A9 SCU Private Timer Instance */
XScuTimer_Config *ConfigPtr;
XScuTimer *TimerInstancePtr = &Timer;

int main()

{
      int c, d, k,l=0, sum = 0;
      int m=16, n=16, p=16,q=16;
      int * f=0x110000;
      int * s=0x110004;
      int * r=0x110008;
      int * flag=0x110012;
      int * syncBarrier=0x110016;
      int * syncBarrier2=0x110020;

    Xil_DCacheDisable();
    int first[16][16], second[16][16], multiply[16][16];
    *f=*first;
    *s=*second;
    *r=*multiply;
    * flag=0;
    XTime tStart, tEnd;
    int time0, time1;

    init_platform();
    timerInit();

    for (c = 0; c < m; c++)
          for (d = 0; d < n; d++)
          first[c][d] = (rand() % 10000);


      if (n != p)
                printf("The matrices can't be multiplied with each
other.\n");
              else
              {
```

```
                  printf("\nRunning on A9 Core #0");
                printf("\nMatrix populated with random numbers less than
10000\n");

                for (c = 0; c < p; c++)
                  for (d = 0; d < q; d++)
                  second[c][d] = (rand() % 10000);

                // Initialize multiply array
                for (c = 0; c < m; c++)
                  for (d = 0; d < q; d++)
                    multiply[c][d]=0;


                // start of Timer
                XTime_GetTime(&tStart);
                time0 = XScuTimer_GetCounterValue(TimerInstancePtr);
                *syncBarrier = 9999;
                while (*syncBarrier2!=9998){
                }
                XScuTimer_Start(TimerInstancePtr);

                // First iteration stops at 8 so it reads the 1st half of
the first array
                for (c = 0; c < 8; c++) {
                  for (d = 0; d < q; d++) {
                    for (k = 0; k < p; k++) {
                      sum = sum + first[c][k]*second[k][d];
                    }
                    multiply[c][d] = sum;
                    sum = 0;
                  }
                }

                while (*flag==0){
                }
                //end of Timer
                XScuTimer_Stop(TimerInstancePtr);
                time1 = XScuTimer_GetCounterValue(TimerInstancePtr);

                printf("1st matrix:\n");

                for (c = 0; c < m; c++) {
                  for (d = 0; d < q; d++)
                    printf("%d ", first[c][d]);

                  printf("\n");
                }

                        printf("2nd matrix:\n");

                        for (c = 0; c < m; c++) {
                          for (d = 0; d < q; d++)
                              printf("%d ", second[c][d]);

                          printf("\n");
                        }

                        printf("Product of the matrices:\n");

                for (c = 0; c < m; c++) {
                  for (d = 0; d < q; d++)
```

```
                    printf("%d ", multiply[c][d]);

                  printf("\n");
                }
              }
              printf("Output took %lu clock cycles.\n", 2*(time0 - time1));
//            printf("Output took %.6f us.\n",
//                        (float)(time0 - time1) / (COUNTS_PER_SECOND));

    cleanup_platform();
    return 0;
}

int timerInit()
{
      int Status;

      /*
       * Initialize the Scu Private Timer so that it is ready to use.
       */
      ConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);

      /*
       * This is where the virtual address would be used, this example
       * uses physical address.
       */
      Status = XScuTimer_CfgInitialize(TimerInstancePtr, ConfigPtr,
                        ConfigPtr->BaseAddr);
      if (Status != XST_SUCCESS) {
            return XST_FAILURE;
      }

      /*
       * Load the timer counter register.
       */
      XScuTimer_LoadTimer(TimerInstancePtr, TIMER_LOAD_VALUE);

      return XST_SUCCESS;
}
```

```
/*****************************************************************************
****
* Core #2
*****************************************************************************
***/
#include <stdio.h>
#include "sleep.h"
#include <time.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters_ps.h"
#include "xil_io.h"
#include "xtime_l.h"
#include "xil_cache.h"
int main()
{
     int i, j , k , l ,m ,n, sum=0;
     int * f=0x110000;
     int * s=0x110004;
     int * r=0x110008;
     int * flag=0x110012;
     int * syncBarrier=0x110016;
     int * syncBarrier2=0x110020;
     int * first=* f;
     int * second=* s;
     int * result=* r;


    Xil_DCacheDisable();


    // Signal from core0 that matrices are populated and ready to start
calculation
    while (*syncBarrier!=9999){
//                l++;
    }
    // Signals core0 that calculation have just  started
    * syncBarrier2 = 9998;
    // First iteration start at 128 byte so it reads the 2nd half of the
first array
    m=128;
      for (k=m;k<256;k+=16){
            for (j=0;j<16;j++){
                  for (i=0;i<16;i++){
                  sum+=(*(first+i+k))*(*(second+i*16+j));
                  }
                  *(result+m)=sum;
                  m++;
                  sum=0;
        }
      }
      *flag=1;
      sleep(1);
//   printf("\nCORE 1 is used 2nd half of the first array");
     cleanup_platform();
    return 0;
}
```