ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Optimizing ECG Signal Analysis by building FPGA-based accelerators using High Level Synthesis

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Κωνσταντίνας Ι.
Κολιογεώργη

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Αθήνα, Ιανουάριος 2016

Εθνικο Μετσοβιο Πολυτεχνειο
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ
ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Optimizing ECG Signal Analysis by building FPGA-based accelerators using High Level Synthesis

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Κωνσταντίνας Ι.
Κολιογεώργη

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή επιτροπή την 22η Ιανουαρίου 2016.

.....	.....	.....
Δημήτριος Ι. Σούντρης	Κιαμάλ Ζ. Πεχμεστζή	Γιώργος Οικονομάκος
Αναπληρωτής Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής

Αθήνα, Ιανουάριος 2016.

.....
Κωνσταντίνα Ι. Κολιογεώργη
Διπλωματούχος Φοιτήτρια
Εθνικού Μετσόβιου Πολυτεχνείου

Contents

Σύντομη περίληψη . . . . . **vi**

Abstract . . . . . **vii**

Εκτεταμένη Περίληψη . . . . . **viii**

Acknowledgements . . . . . **.xxviii**

List of Figures . . . . . **xxix**

List of Tables . . . . . **xxxii**

1 Introduction **1**

2 Problem Overview **3**

2.1 ECG Analysis Flow 3

2.2 Related Work 7

3 Theoritical Background **9**

3.1 Background Information on SVM classifier 9

3.2 High Level Synthesis 11

3.3 Zynq Evaluation and Development Board Specifications 16

4 Code Restructuring for HLS **18**

4.1 Advancing Coarse Level Parallelism in HLS 18

4.1.1 Parallelization Technique 18

4.1.2 Results 22

4.2 Advancing Instruction Level Parallelism through arithmetic operation reshaping . 25

4.2.1 Parallelization Technique 25

4.2.2 Results 28

5 Exploration of HLS Directives 30

5.1 Selection of Optimization Directives 30

5.2 Application on Original Code 32

5.2.1 Impact of each directive 33

5.3 Comparison of Implementations 44

5.4 Optimal Configurations 47

6 Implementation on Zedboard 50

6.1 Implementation Description 50

6.2 Results 55

7 Conclusion 57

7.1 Summary 57

7.2 Future Work 58

References 59

Σύντομη Περίληψη

Το ηλεκτροκαρδιογράφημα (ΗΚΓ) λόγω της στενής συνάφειάς του με τη φυσιολογία της καρδιάς είναι από τα βασικά βιοσήματα που χρησιμοποιούνται για την παρακολούθηση της κατάστασης της υγείας ενός ανθρώπου. Συνεπώς, η ανάλυσή του και η ερμηνεία του έχουν καθιερωθεί ως ένας σημαντικός κλάδος στη σύγχρονη ιατρική και αυτό έχει οδηγήσει στην εκπόνηση πολλών μελετών σχετικών με τη ψηφιακή επεξεργασία του. Λόγω της πολυπλοκότητας της δημιουργίας μοντέλων ακριβείας για την αξιολόγηση και την πρόβλεψη της κατάστασης της καρδιάς, οι Τεχνικές Μηχανικής Εκμάθησης έχουν επικρατήσει στον τομέα της Ανάλυσης του ΗΚΓ. Οι Μηχανές Διανυσμάτων Υποστήριξης (Support Vector Machines- SVM) συγκεκριμένα είναι ιδιαιτέρως διαδεδομένες λόγω της ακριβούς πρόβλεψης και της ενδιαφέρουσας υπολογιστικής δομής τους. Επιπρόσθετα η ανάγκη συνεχούς παρακολούθησης της κατάστασης της καρδιάς και μάλιστα σε πραγματικό χρόνο έχουν αυξήσει τις απαιτήσεις για επιτάχυνση της ψηφιακής ανάλυσης του ΗΚΓ και πραγματοποίησής της σε σύστημα χαμηλής κατανάλωσης ενέργειας. Στόχος αυτής της διπλωματικής εργασίας είναι η αξιοποίηση των δυνατοτήτων του HLS για τη δημιουργία αποδοτικών SVM ως επιταχυντές σε υλικό. Μελετάται ο εντοπισμός αρρυθμιών στο ΗΚΓ χρησιμοποιώντας ως βάση δεδομένων μια βάση δεδομένων για ΗΚΓ που έχει αναπτυχθεί μέσω κοινής συνεργασίας των πανεπιστημίων MIT και BIH. Σε πρώτο επίπεδο ο αρχικός κώδικας αναδομείται με κριτήριο την επιτάχυνση ώστε να δημιουργηθεί αποδοτικός επιταχυντής. Σε δεύτερο επίπεδο εξερευνώνται οι τεχνικές βελτιστοποίησης του εργαλείου HLS οι οποίες εφαρμόζονται στον αρχικό και στον τροποποιημένο κώδικα για περαιτέρω βελτίωση του ως προς μετρικές επίδοσης και χρησιμοποίησης πόρων. Ο συνδυασμός των δύο επιπέδων επιφέρει κέρδος έως και 98% σε χρόνο εκτέλεσης σε σύγκριση με το χρόνο εκτέλεσης του αρχικού κώδικα ενώ παρέχονται στο σχεδιαστή τα βέλτιστα σημεία κατά Pareto με βάση τα οποία μπορεί να επιλέξει μια υλοποίηση ανάλογα με τις απαιτήσεις της εκάστοτε εφαρμογής σε ταχύτητα εκτέλεσης και χρησιμοποίηση πόρων.

Λέξεις Κλειδιά: Σχεδιασμός Ιατρικών Ενσωματωμένων Συστημάτων, Ανάλυση Ηλεκτροκαρδιογραφήματος, Τεχνικές Μηχανικής Μάθησης, Μηχανές Διανυσμάτων Υποστήριξης, HW/SW σχεδιασμός, Αναπτυξιακή Πλακέτα Zynq Evaluation and Development Board, Εργαλεία Σύνθεσης Υψηλού Επιπέδου (HLS)

Abstract

One of the most fundamental and crucial biological signals for monitoring and assessing the health condition of a person is the Electrocardiogram (ECG) due to its inherent relation to heart physiology. Consequently, its analysis and interpretation has been established as an important field in modern medicine and this in turn has spawned various inter-disciplinary studies including digital processing analysis of the signal. Given the complexity of deriving exact models for assessing and predicting the heart's condition, machine learning techniques have recently dominated the field of ECG analysis. Support Vector Machines based classifiers especially, have grown very popular as the key element of machine learning based ECG analysis due to their capability of accurate prediction and their interesting computational structure. Last but not least, constant monitoring and real-time heart condition assessment have imposed new requirements for acceleration and low power execution of a digital ECG analysis flow system. Taking all these into consideration, in this work we focus on utilizing High Level Synthesis capabilities to produce efficient SVM hardware accelerators. Our case study is arrhythmia detection using MIT-BIH ECG signal medical database. We show that as a first step, the original code under acceleration can be re-structured in order to create instances which are efficiently transformed into a HW accelerator. As a second step, an exploration is performed on the transformed code in order to determine which HLS directives produce the best outcome in terms of various performance and resources utilization metrics. Our combined analysis shows that we can achieve results of up to 99% execution latency gain compared to the original SVM code and the designer is given a set of Pareto Optimal design points in order to decide the best trade-off between gains in latency and increase in utilized FPGA HW resources.

Keywords: Medical embedded system design, ECG analysis, machine learning, Support Vector Machines, HW/SW codesign, Zynq Evaluation and Development Board, High Level Synthesis

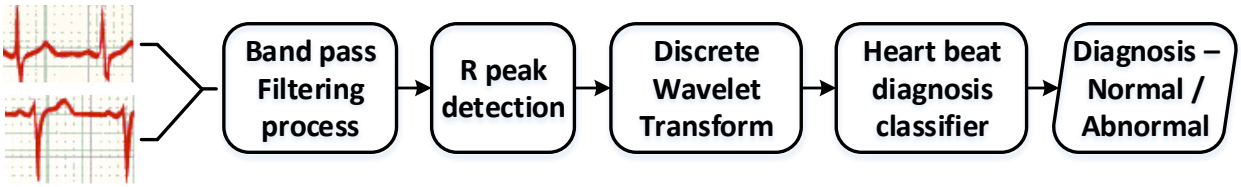
Ροή Ανάλυσης του ΗΚΓ

Η μορφή του ΗΚΓ και ο καρδιακός ρυθμός που εξάγεται από το ΗΚΓ είναι δηλωτικά της κατάστασης της καρδιάς. Στην ουσία το ΗΚΓ αποτυπώνει διαδοχικούς καρδιακούς κύκλους. Ο καρδιακός κύκλος (διαστολή, συστολή, ηρεμία) συντονίζεται από ηλεκτρικά σήματα που παράγονται από κατάλληλα κέντρα διέγερσης της καρδιάς. Υπάρχουν τρία βασικά ηλεκτρικά σήματα που εμφανίζονται στο ΗΚΓ: το έπαρμα P, το σύμπλεγμα QRS που αποτελείται από τις κορυφές Q,R,S και το έπαρμα T. Αυτά τα σήματα είναι στην πραγματικότητα μεταβολές του ηλεκτρικού δυναμικού διαφόρων περιοχών της καρδιάς και άρα το ΗΚΓ απεικονίζει την ηλεκτρική δραστηριότητα της καρδιάς. Τα επάρματα αυτά και οι αποστάσεις μεταξύ τους έχουν συγκεκριμένη χρονική διάρκεια και μορφολογία. Οποιαδήποτε παρέκκλιση από τη φυσιολογική μορφολογία τους πρέπει να μελετηθεί καθώς μπορεί να είναι δείγμα παθολογικής βλάβης. Η καρδιακή αρρυθμία είναι η πιο συνηθισμένη καρδιακή βλάβη και είναι η διαταραχή του καρδιακού ρυθμού. Η αρρυθμία μπορεί να είναι από ασυμπτωματική μέχρι κρίσιμη για την ανθρώπινη ζωή. Για αυτό το λόγο κρίνεται απαραίτητη η μελέτη του ΗΚΓ, ως μέσο διάγνωσης αρρυθμιών. Οι αρρυθμίες είναι μεμονωμένα περιστατικά που εκδηλώνονται σε τυχαίες χρονικές στιγμές. Επομένως είναι αναγκαία η μελέτη του ΗΚΓ μεγάλων χρονικών διαστημάτων. Ο μεγάλος όγκος δεδομένων προς μελέτη καθιστά απαραίτητη τη χρήση τεχνικών μηχανικής μάθησης για την επεξεργασία του. Ταξινομητές βασίζονται σε τεχνικές μηχανικής μάθησης για την εκπαίδευσή τους με αυτό το μεγάλο σύνολο δεδομένων ώστε τελικά να μπορούν να διαγνώσουν σωστά την ύπαρξη ή μη αρρυθμίας σε ένα νέο σύνολο δεδομένων ΗΚΓ. Στη συγκεκριμένη εργασία χρησιμοποιείται η βάση δεδομένων αρρυθμίας MIT-BIH Arrhythmia Database, η οποία περιλαμβάνει παλμούς για τους οποίους έχει γίνει διάγνωση από καρδιολόγους. Η διαδικασία επεξεργασίας και ανάλυσης του ΗΚΓ για την εξαγωγή των επιμέρους παλμών και των χαρακτηριστικών τους ώστε τελικά να γίνει η διάγνωση χρησιμοποιώντας μοντέλα τεχνικής μηχανικής μάθησης παρουσιάζεται ακολούθως και απεικονίζεται στο Σχ.1.

- **Αποθορυβοποίηση**: Το σήμα του ΗΚΓ φιλτράρεται για την απομάκρυνση θορύβου που προέρχεται κυρίως από την τροφοδοσία και τις κινήσεις του ασθενή.
- **Εντοπισμός κορυφών** : Σε αυτό το στάδιο ανιχνεύονται οι παλμοί που απαρτίζουν το φιλτραρισμένο πια σήμα. Η ανίχνευση των παλμών επιτυγχάνεται μέσω της ανίχνευσης των κορυφών R με τη χρήση ειδικών συναρτήσεων.
- **Εξαγωγή χαρακτηριστικών**: Σε αυτό το στάδιο γίνεται η εξαγωγή χαρακτηριστικών κάθε παλμού, βάσει των οποίων θα γίνει σε επόμενο στάδιο η διάγνωση. Στη συγκεκριμένη διπλωματική ο Μετασχηματισμός Κυματιδίων (Wavelet Transform - WT) εφαρμόζεται σε κάθε παλμό ώστε το σήμα να μελετηθεί στο πεδίο της συχνότητας και του χρόνου ταυτόχρονα. Οι συντελεστές που προκύπτουν από το μετασχηματισμό αυτό για κάθε παλμό σχηματίζουν το διάνυσμα

των χαρακτηριστικών κάθε παλμού.

- **Ταξινόμηση-Διάγνωση**: Σε αυτό το στάδιο το διάνυσμα των χαρακτηριστικών που δημιουργήθηκε στο προηγούμενο στάδιο για τον τρέχοντα παλμό χρησιμοποιείται ως είσοδος σε έναν ταξινομητή για να πραγματοποιηθεί η διάγνωση. Ο ταξινομητής προηγουμένως έχει εκπαιδευτεί χρησιμοποιώντας ένα μεγάλο σύνολο διανυσμάτων χαρακτηριστικών όμοιων με αυτά του προηγούμενου σταδίου. Σε αυτή τη διπλωματική ο ταξινομητής βασίζεται σε Μηχανές Διανυσμάτων Υποστήριξης.



Σχήμα 1: Ροή Ανάλυσης ΗΚΓ

Θεωρητικό Υπόβαθρο

Θεωρία Μηχανών Διανυσμάτων Υποστήριξης

Οι Μηχανές Διανυσμάτων Υποστήριξης (Support Vector Machines -SVM) είναι μοντέλα επιβλεπόμενης μάθησης που εκπαιδεύονται με ένα μεγάλο σύνολο δεδομένων και είναι κατάλληλη για την ταξινόμηση των νέων εισόδων σε δύο υποψήφιες κλάσεις συμπληρωματικές μεταξύ τους. Το σύνολο εκπαίδευσης αποτελείται από διανύσματα με συγκεκριμένα χαρακτηριστικά καθένα από τα οποία διαθέται μια ετικέτα δηλωτικής της κλάσης στην οποία ανήκει. Ένα σύνολο από άλλα διανύσματα με τα ίδια χαρακτηριστικά και γνωστές τις ετικέτες χρησιμοποιείται για να ελεχθεί η ακρίβεια της πρόβλεψης.

Τα SVM εφαρμόζουν αρχικά μια συνάρτηση πυρήνα που ανάγει τα διανύσματα σε ένα χώρο περισσότερων διαστάσεων, όπου είναι πιο εύκολος ο διαχωρισμός τους. Στο χώρο αυτό βρίσκουν ένα υπερεπίπεδο το οποίο αποτελείται από τα διανύσματα που απέχουν μέγιστα από τα διανύσματα που ανήκουν σε κάθε κλάση. Κάθε νέο διάνυσμα ανάγεται σε αυτόν το χώρο, υπολογίζεται η απόσταση του από το υπερεπίπεδο και άρα με βάση τη θέση του σε σχέση με αυτό ταξινομείται στην αντίστοιχη κλάση. Η συνάρτηση πυρήνα είναι καθοριστική για την ακρίβεια και την πολυπλοκότητα του μοντέλου. Λόγω των μη γραμμικών σχέσεων μεταξύ των χαρακτηριστικών του διανύσματος κάθε παλμού χρησιμοποιούμε μη γραμμική συνάρτηση πυρήνα και συγκεκριμένα εκθετικής φύσης.

Ακολουθεί η μαθηματική εξίσωση που περιγράφει τον υπολογιστικό πυρήνα του ταξινομητή και ο αντίστοιχος κώδικας C που την υλοποιεί:

$$Class = sgn(\sum_{i=1}^{N\_sv}(y_i * a_i * exp(-\gamma||\mathbf{x} - \mathbf{sup\_vector}_i||^2)) - b) \quad (1)$$

όπου K είναι η συνάρτηση πυρήνα, $\mathbf{x}$ είναι το διάνυσμα του παλμού προς ταξινόμηση, $\mathbf{sup\_vector}_i$ είναι το i-οστό διάνυσμα υποστήριξης και y_i, a_i είναι τιμές διαφορετικές για κάθε διάνυσμα υποστήριξης και προέκυψαν κατά την εκπαίδευση. Η μεταβλητή b είναι μια μεταβλητή σύγκρισης, αποτέλεσμα της εκπαίδευσης και σταθερή για όλα τα διανύσματα υποστήριξης.

Listing 1: Αρχικός κώδικας ταξινομητή.

```
const float sv_coef[N_sv];
const float sup_vectors[D_sv][N_sv];

void SVM_predict (int *y, float test_vector[D_sv]) {

loop_i: for (i=0; i<N_sv; i++){
                loop_j: for (j=0; j<D_sv; j++){
                        diff=test_vector[j]-sup_vectors[j][i];
                        norma = norma + diff*diff;
                }
                sum = sum + exp(-gamma*norma)*sv_coef[i];
                norma=0;
        }

        sum = sum - b;

        if (sum<0)
                *y = -1;
        else
                *y = 1;
}
```

Στον Κώδικα 1 η μεταβλητή sv_coef ισοδυναμεί με το γινόμενο y_i και α_i της εξίσωσης 1. Ο αριθμός των διανυσμάτων υποστήριξης N_sv και ο αριθμός των χαρακτηριστικώνD_sv όπως και η επιλογή συνάρτησης πυρήνα έχουν μεγάλη επίδραση στην πολυπλοκότητα. Στην έρευνα αυτή, η εκπαίδευση κατέληξε σε N_sv ίσο με 1274 και D_sv ίσο με 18.

High Level Synthesis

Το HLS είναι ένα σχεδιαστικό εργαλείο που δημιουργεί μονάδες hardware ειδικού σκοπού λαμβάνοντας ως είσοδο την περιγραφή της λειτουργικότητάς τους σε C. Παρέχει έτσι τη δυνατότητα στους προγραμματιστές να επιταχύνουν τα υπολογιστικά απαιτητικά κομμάτια των εφαρμογών τους υλοποιώντας τα ως ξεχωριστές μονάδες σε Field Programmable Gate Array (FPGA). Έτσι μια εφαρμογή μπορεί να εκτελείται στον επεξεργαστή του συστήματος και να καλεί τον επιταχυντή που έχει υλοποιηθεί σε hardware για το πιο απαιτητικό κομμάτι της.

Το πρώτο βήμα χρησιμοποίησης του HLS είναι η ανάπτυξη της εφαρμογής σε γλώσσα προγραμματισμού C, καθιστώντας έτσι πιο εύκολο τον έλεγχο της ορθότητας από ό,τι υλοποιώντας την εφαρμογή σε

γλώσσα περιγραφής υλικού. Ακολουθεί η διαδικασία της σύνθεσης, στο τέλος της οποίας παράγεται η περιγραφή της λειτουργικότητας σε επίπεδο καταχωρητή (Register Transfer Level - RTL). Κατά τη σύνθεση το HLS χρονοδρομολογεί τις εντολές-λειτουργίες του κώδικα και δεσμεύει τους αναγκαίους πόρους για την υλοποίησή τους. Διαθέτει τεχνικές βελτιστοποίησης τις οποίες εφαρμόζει είτε αυτομάτως είτε μετά από εντολή του χρήστη. Δημιουργεί έτσι υλοποιήσεις με υψηλές επιδόσεις και αποδοτική χρησιμοποίηση των διαθέσιμων πόρων. Οι πληροφορίες αυτές υπάρχουν στην αναφορά που παράγεται κατά τη διάρκεια τη σύνθεσης. Με βάση αυτές ο χρήστης μπορεί να διερευνήσει τις παρεχόμενες τεχνικές για να κατασκευάσει μία μονάδα που να ικανοποιεί τις προδιαγραφές σε αποδοτικότητα ταχύτητας και πόρων. Οι μετρικές που χρησιμοποιούνται για την αξιολόγηση του αποτελέσματος αφορούν τον εμβαδόν (area: LUTs, registers, block-RAM, DSPs, flip flops), το χρόνο απόκρισης του επιταχυντή (latency) και το χρόνο που πρέπει να παρέλθει μέχρι η μονάδα να μπορεί να επεξεργαστεί νέα δεδομένα. Οι τεχνικές βελτιστοποίησης αυτών των μετρικών εφαρμόζονται σε διάφορα μέρη του κώδικα, όπως σε συναρτήσεις, βρόχους, πίνακες και περιοχές που περιλαμβάνουν κάποια ή όλα τα παραπάνω μέρη. Όταν ολοκληρωθεί η διαδικασία επιτάχυνσης της μονάδας, εξάγεται σε κατάλληλο φορμάτ για να συμπεριληφθεί στην αρχιτεκτονική άλλων σχεδιαστικών εργαλείων.

Αυτά τα χαρακτηριστικά του HLS το καθιστούν ιδανική επιλογή για τη δημιουργία του ταξινομητή ως επιταχυντή στο FPGA κομμάτι του Zedboard. Χρησιμοποιώντας τον κώδικα του ταξινομητή θα εφαρμόσουμε σε αυτόν δομικές αλλαγές και τις τεχνικές βελτιστοποίησης που παρέχει το HLS προκειμένου να επιτύχουμε τις απαιτήσεις σε χρόνο εκτέλεσης της ανίχνευσης αρρυθμίας. Τα πρώτα στάδια θα εκτελούνται στον επεξεργαστή της πλακέτας και η ταξινόμηση-διάγνωση στον επιταχυντή στο FPGA.

Zedboard

Το Zedboard είναι μια αναπτυξιακή πλακέτα χαμηλού κόστους. Είναι ένα σύστημα υλοποιημένο σε ολοκληρωμένο κύκλωμα (SoC) που ανήκει στην οικογένεια Zynq-7000 της Xilinx. Συνδυάζει την ύπαρξη Υπολογιστικού Συστήματος με δύο επεξεργαστές ARM με την ύπαρξη Επαναπρογραμματιζόμενης Λογικής. Υποστηρίζει την υλοποίηση Linux, Android, Windows, OS/RTOS εφαρμογών. Τα κύρια χαρακτηριστικά του είναι:

- **Μνήμη**: δυναμική (DDR3) και στατική μνήμη (SPI Flash,SD Card Interface)
- **USB**: USB-to-UART σύνδεση, λειτουργικότητα JTAG, προστασία κυκλωμάτων USB
- **Οθόνη και Ήχος**:HDMI Transmitter, Analog Device Audio Codec, OLED Display
- **Clock Sources**: 33.3333 MHz ρολόι για το Υπολογιστικό Σύστημα ενώ το Υολογιστικό Σύστημα παράγει έως 4 ρολόγια για το Επαναπρογραμματιζόμενο μέρος της πλακέτας
- **Reset Sources**: εξωτερικοί διακόπτες για επανεκκίνηση της πλακέτας και επαναπρογραμματισμό του FPGA

- **User I/O**: 7 user GPIO push button, 8 user dip switches, 8 LEDs.
- **10/100/1000 Ethernet PHY**: Ethernet θύρα για σύνδεση στο διαδίκτυο
- **PS και PL I/O επεκτάσεις**

Στόχος της δουλειάς αυτής είναι η επιτάχυνση του λογισμικού χτίζοντας έναν επιταχυντή στην Επαναπρογραμματιζόμενη Λογική (PL). Ο επιταχυντής θα πρέπει ακόμα να επικοινωνεί με το Υπολογιστικό Σύστημα, για αυτό το ενδιαφέρον μας επικεντρώνεται στους πόρους της επαναπρογραμματιζόμενης λογικής και στα χαρακτηριστικά τοψ τρόπου διασύνδεσης των δύο μερών της πλακέτας. Συγκεκριμένα το Zedboard διαθέτει έναν δίαυλο επικοινωνίας με το υπολογιστικό τμήμα, ο οποίος εξασφαλίζει επικοινωνία κλιμακώμενης απόδοσης και υψηλών επιδόσεων (High bandwidth AMBA interconnect). Οι διαθέσιμοι πόροι του FPGA αναφέρονται στον Πίνακα 1.

Πίνακας 1: Διαθέσιμοι πόροι του Zedboard.

Πόροι	BRAM_18K	DSP48E	FF	LUT
Διαθέσιμοι	280	220	106400	53200

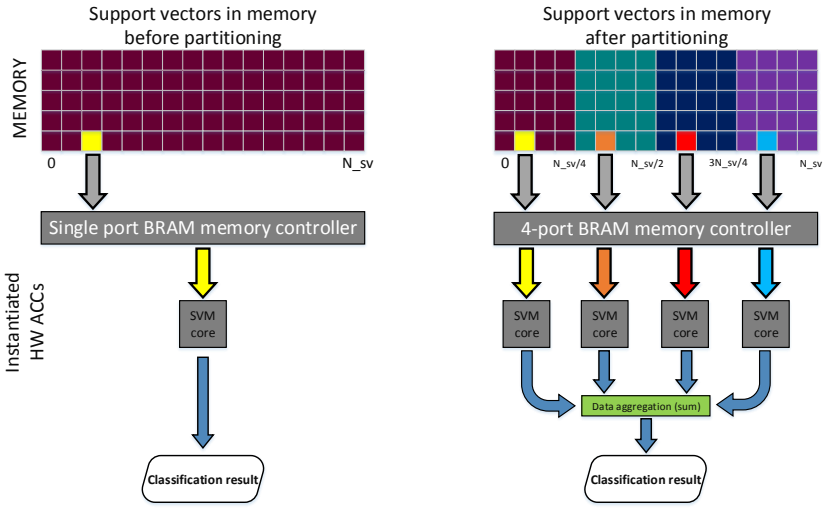
Αναδόμηση Κώδικα για το HLS

Ανάπτυξη Παραλληλισμού σε Επίπεδο Μπλοκ

Αρχικά επιδιώκουμε την εξαγωγή παραλληλισμού σε επίπεδο συνάρτησης. Για να το επιτύχουμε αυτό εκμεταλλευόμαστε τον εγγενή παραλληλισμό του αλγορίθμου. Το διάνυσμα εισόδου του τρέχοντος παλμού υλοποιείται ως ένας πίνακας-γραμμή με 18 στοιχεία-χαρακτηριστικά. Τα διανύσματα υποστήριξης υλοποιούνται ως ένας δισδιάστατος πίνακας με τόσες στήλες όσο το πλήθος των διανυσμάτων υποστήριξης ενώ κάθε στήλη έχει τόσα στοιχεία όσα είναι τα χαρακτηριστικά που μελετώνται. Σύμφωνα με τον κώδικα 1 για κάθε διάνυσμα εισόδου προς ταξινόμηση, υπολογίζεται η ευκλείδια απόστασή του από κάθε διάνυσμα υποστήριξης και υψώνεται στο τετράγωνο. Στη συνέχεια εφαρμόζεται σε αυτή τη τιμή η συνάρτηση πυρήνα και η νέα τιμή που προκύπτει πολλαπλασιάζεται με τον αντίστοιχο παράγοντα κάθε διανύσματος υποστήριξης. Οι τιμές που προκύπτουν από τους υπολογισμούς με κάθε διάνυσμα υποστήριξης αθροίζονται και το τελικό αποτέλεσμα συγκρίνεται με τη τιμή βιας για την ταξινόμηση σε μια από τις δύο κλάσεις. Οι πράξεις που απαιτούνται μεταξύ του διανύσματος εισόδου και κάθε διανύσματος υποστήριξης είναι ανεξάρτητες μεταξύ τους. Μπορούν λοιπόν να εκτελούνται παράλληλα. Σε αυτή την έμφυτη παραλληλία βασίζεται η προτεινόμενη τεχνική. Ο πίνακας των διανυσμάτων υποστήριξης μπορεί να επιμεριστεί σε μικρότερους πίνακες, καθένας από τους οποίους περιέχει λιγότερα διανύσματα υποστήριξης. Οι πράξεις για τον υπολογισμό του μερικού αθροίσματος με το οποίο συνεισφέρει το κάθε κομμάτι πίνακα στο τελικό άθροισμα εκτελούνται παράλληλα. Επιτύχαμε λοιπόν την εκτέλεση του ίδιου υπολογιστικού πυρήνα πολλές φορές παράλληλα μόνο που κάθε μία από αυτές δρα σε μικρότερο σύνολο δεδομένων. Η τεχνική απεικονίζεται στο Σχ.2.

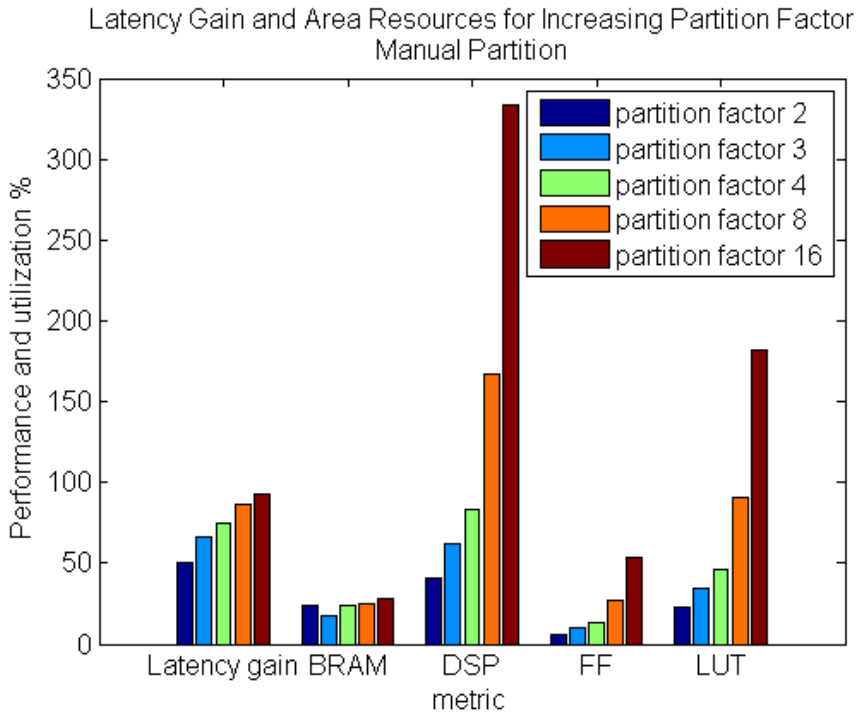
Η υλοποίηση της παραπάνω ιδέας απαιτεί αλλαγές στον κώδικα σε δομικό επίπεδο αλλά και τη χρήση των τεχνικών βελτιστοποίησης που προσφέρει το HLS. Συγκεκριμένα ο υπολογιστικός πυρήνας του ταξινομητή υλοποιείται ως συνάρτηση η οποία καλείται από την κύρια συνάρτηση τόσες φορές όσες φορές έχει επιμεριστεί ο πίνακας. Ο πίνακας των διανυσμάτων υποστήριξης και ο πίνακας των παραγόντων τους επιμερίζονται επίσης σε υποπίνακες με χρήση των κατάλληλων αυτόματων τεχνικών που παρέχει το εργαλείο. Σε διαφορετική περίπτωση θα δημιουργούνταν αντίτυπα των πινάκων για να είναι εφικτή η πρόσβαση σε πάνω από δύο στοιχεία του κάθε πίνακα τη φορά, περιορισμός που επιβάλλεται λόγω της υλοποίησης των πινάκων ως BRAM με δύο θύρες ανάγνωσης. Κάθε στιγμιότυπο της συνάρτησης έχει πρόσβαση στα στοιχεία μόνο ενός μέρους του επιμερισμένου πίνακα. Ο τροποποιημένος κώδικας παρατίθεται στον Κώδικα 2.

Αυτή η ιδέα υλοποιήθηκε για επιμερισμό του πίνακα σε 2,3,4,8 και 16 μέρη. Η βελτίωση του latency ήταν η αναμενόμενη, δηλαδή ο χρόνος εκτέλεσης διαιρέθηκε σχεδόν κατά έναν παράγοντα 2,3,4,8 και 16. Η χρησιμοποίηση σε DSP πολλαπλασιάστηκε κατά αυτόν τον παράγοντα ενώ υπήρχε σταδιακή αύξηση



Σχήμα 2: Παραλληλισμός σε Επίπεδο Μπλοκ

και στη χρησιμοποίηση LUT και Flip Flop. Η μνήμη παρέμεινε σταθερή εκτός από την τελευταία περίπτωση όπου σημειώθηκε μια απότομη αύξηση. Δοκιμάζοντας να χωρίσουμε τους πίνακες με το χέρι, δηλώνοντας τους εξαρχής χωριστά, πετύχαμε ακόμα μεγαλύτερη επιτάχυνση (ακόμα πιο κοντά στον ιδανικό παράγοντα 2,3,4,8,16 αντιστοίχως) και εξαλείφθηκε το πρόβλημα με την απότομη αύξηση σε BRAM. Τα αποτελέσματα απεικονίζονται στο Σχ.3.



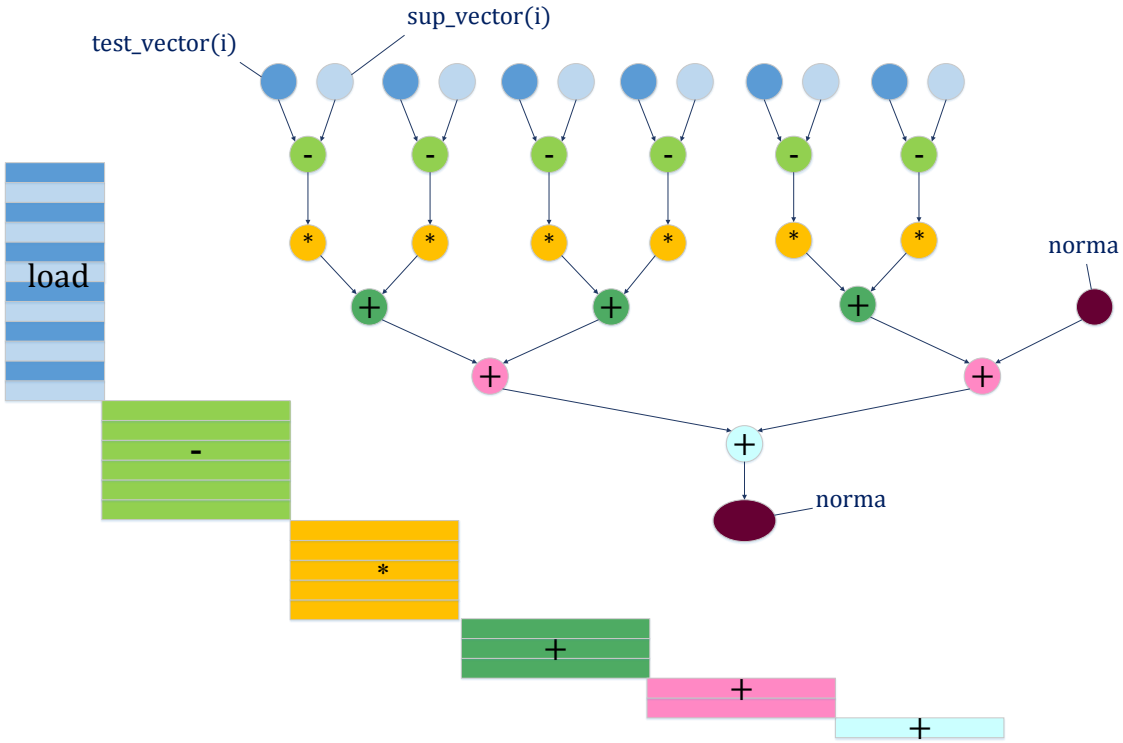
Σχήμα 3: Απόδοση και Χρησιμοποίηση Πόρων για αυξανόμενο αριθμό διαμερίσεων (μη αυτόματη)

Listing 2: Τροποποιημένος κώδικας για τον μπλοκ παραλληλισμό.

```
#include <math.h>
#include "svm.h"
#include <stdio.h>
#define gamma 8

void foo(int width,int offset,float *sum,float test_vector[D_sv],
float sv_coef[N_sv],float sup_vectors[D_sv][N_sv]){
 int i,j;
 float diff;
 float norma=0;
 *sum=0;

 loop_i:for (i=0; i<width; i++){
        loop_j:for (j=0; j<D_sv; j++){
                        diff=test_vector[j]-sup_vectors[j][i+offset];
                        norma = norma + diff*diff;
                        }
                        *sum = *sum + exp(-gamma*norma)*sv_coef[i+offset];
                        norma=0;
                }
}


void classify(int * y){

        const float sv_coef[N_sv]={
                                #include "sv_coef.dat"
                        };

        const float test_vector[D_sv]={
                                #include "test_vector.dat"
                        };

        const float support_vectors[D_sv][N_sv]={
                                #include "support_vectors.dat"
                        };

        float diff;
        float sum1,sum2,sum;

        foo(N_sv/2,0,&sum1,test_vector,sv_coef,support_vectors);

        foo(N_sv/2,N_sv/2,&sum2,test_vector,sv_coef,support_vectors);

        sum = sum1 + sum2 - b;

        if (sum<0) *y = -1;
                else    *y = 1;

}
```

Ανάπτυξη Παραλληλισμού σε Επίπεδο Εντολών μέσω μετασχηματισμού αριθμητικών υπολογισμών

Σε αυτό το κομμάτι θα εξετάσουμε την παραλληλοποίηση σε επίπεδο εντολών. Συγκεκριμένα θα ασχοληθούμε με την παραλληλοποίηση του εσωτερικού βρόχου του ταξινομητή. Αυτός ο βρόχος είναι υπεύθυνος για τον υπολογισμό της ευκλείδειας απόστασης του διανύσματος από ένα διάνυσμα υποστήριξης υψωμένης στο τετράγωνο. Σε κάθε επανάληψη υπολογίζεται η διαφορά μεταξύ των αντίστοιχων χαρακτηριστικών των δύο διανυσμάτων και υψώνεται στο τετράγωνο. Αντί να υπολογίζεται κάθε φορά μία μόνο διαφορά θα μπορούσαν να υπολογίζονται περισσότερες και να αθροίζονται σταδιακά σε μια μεταβλητή η οποία στο τέλος του βρόχου θα περιέχει την τετραγωνισμένη νόρμα. Η άθροιση όμως πολλών αριθμών κινητής υποδιαστολής συνεπάγεται μεγάλο κρίσιμο μονοπάτι επειδή οι προσθέσεις γίνονται σειριακά αν και δεν υπάρχει εξάρτηση μεταξύ των προσθετέων. Η πρόσθεση μπορεί να υλοποιηθεί αποδοτικά αν χρησιμοποιηθεί μια δενδρική μορφή. Ο εσωτερικός βρόχος εκτυλίσσεται τόσες φορές όσες διαφορές θα υπολογιστούν ταυτόχρονα. Οι διαφορές υπολογίζονται παράλληλα μεταξύ τους όπως και οι υψώσεις των διαφορών στο τετράγωνο. Στη συνέχεια οι διαθέσιμες τιμές προστίθενται ανά δύο και τα αποτελέσματα κρατώνται σε προσωρινές μεταβλητές. Αυτές προστίθενται και πάλι ανά δύο κ.ο.κ. μέχρι τον υπολογισμό της ολικής νόρμας στο τετράγωνο. Η δενδρική δομή και η χρονοδρομολόγηση του εργαλείου απεικονίζονται στο Σχ.4 ενώ οι αλλαγές στην υλοποίηση στον Κώδικα 3 στην περίπτωση που ο βρόχος εκτυλίσσεται κατά έναν παράγοντα ίσο με 6.



Σχήμα 4: Δενδρικής δομής υπολογισμοί και χρονοδρομολόγηση.

Listing 3: Κώδικας με εκτύλιξη του εσωτερικού βρόχου και δενδρική υλοποίηση των υπολογισμών.

```
#define gamma 8
const float sv_coef[N_sv];
const float sup_vectors[D_sv][N_sv];

void SVM_predict (int *y, float test_vector[D_sv]){

loop_i:for (i=0; i<N_sv; i++) {
        loop_j:for (j=0; j<D_sv; j=j+6) {
                d1=test_vector[j]-sup_vectors[j][i];
                d2=test_vector[j+1]-sup_vectors[j+1][i];
                d3=test_vector[j+2]-sup_vectors[j+2][i];
                d4=test_vector[j+3]-sup_vectors[j+3][i];
                d5=test_vector[j+4]-sup_vectors[j+4][i];
                d6=test_vector[j+5]-sup_vectors[j+5][i];

                sq_prod1=d1*d1;
                sq_prod2=d2*d2;
                sq_prod3=d3*d3;
                sq_prod4=d4*d4;
                sq_prod5=d5*d5;
                sq_prod6=d6*d6;

                tmp_sum1=sq_prod1+sq_prod2;
                tmp_sum2=sq_prod3+sq_prod4;
                tmp_sum3=sq_prod5+sq_prod6;

                tmp_sum4=tmp_sum1+tmp_sum2;
                norma = norma + tmp_sum3;

                norma = norma + tmp_sum4;
        }

        sum = sum + exp(-gamma*norma)*sv_coef[i];
        norma=0;
}
        sum = sum - b;
        if (sum<0)
                *y = -1;
        else
                *y = 1;
}
```

Η παραπάνω ιδέα υλοποιήθηκε για εκτύλιξη του εσωτερικού βρόχου 3,6 και 18 φορές που αντιστοιχεί σε πλήρη εκτύλιξη. Τα αποτελέσματα συγκεντρώνονται στον Πίνακα 2, όπου πραγματοποιείται σύγκριση μεταξύ εκτύλιξης του βρόχου με το χέρι, χρησιμοποιώντας δενδρική δομή και εκτύλιξης του βρόχου με τις αυτόματες τεχνικές του εργαλείου.

Πίνακας 2: Σύγκριση μετρικών μεταξύ αυτόματης και προτεινόμενης εκτύλιξης βρόχου

εκδοχή	Παράγοντας εκτύλιξης	Αυτόματη εκτύλιξη					Προτεινόμενη εκτύλιξη				
		latency (cycles)	BRAM (%)	DSP (%)	FF (%)	LUT (%)	latency (cycles)	BRAM (%)	DSP (%)	FF (%)	LUT (%) (%)
initial	-	412783	24	20	3	11	412783	24	20	3	11
unrolled	3	252259	24	21	3	11	214039	47	26	4	14
unrolled	6	206395	24	23	3	11	149065	70	34	5	18
unrolled	18	173271	27	20	3	11	90461	27	50	8	29

Παρατηρείται σημαντική βελτίωση στο latency όταν η εκτύλιξη γίνεται με το χέρι και μάλιστα η διαφορά μεγαλώνει όσο μεγαλώνει και ο παράγοντας της εκτύλιξης. Η χρησιμοποίηση των DSP, LUTs και Flip Flop αυξάνεται καθώς η αντιγραφή του σώματος του εσωτερικού βρόχου οδηγεί στην δέσμευση περισσότερων πόρων προκειμένου οι πράξεις να δρομολογηθούν ταυτόχρονα. Ένα μη αναμενόμενο αποτέλεσμα είναι οι διακυμάνσεις στη χρησιμοποίηση της μνήμης. Οφείλονται ωστόσο στη δημιουργία αντιγράφων των πινάκων από το HLS για να είναι δυνατές πολλές προσβάσεις στον ίδιο πίνακα ταυτόχρονα. Στην τελευταία περίπτωση δεν υπάρχει αύξηση επειδή το HLS σπάει αυτόματα τον πίνακα των διανυσμάτων υποστήριξης κατα μήκος των γραμμών. Το πρόβλημα επιλύεται με σπάσιμο του πίνακα σε όλες τιε περιπτώσεις στην κατάλληλη διάσταση και με τον κατάλλληλο τρόπο με τις κατάλληλες τεχνικές που προσφέρει το HLS στο χρήστη.

Εξερεύνηση HLS directives

Επιλογή directives

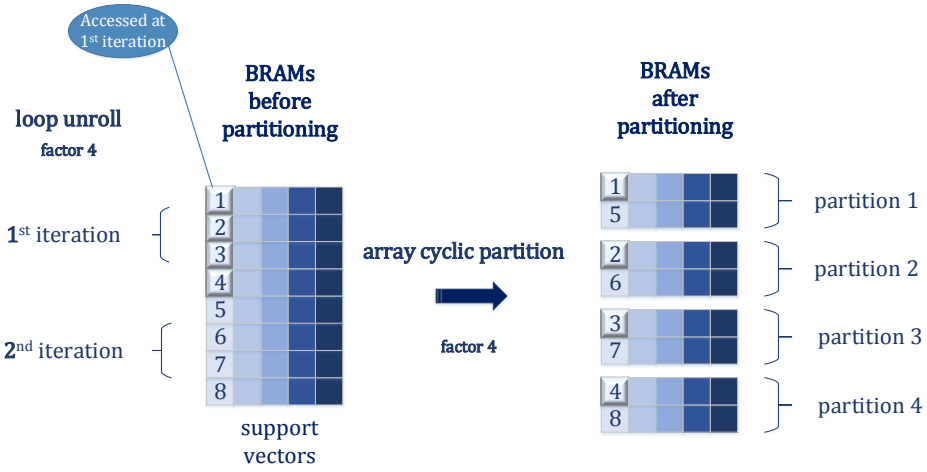
Οι τεχνικές που παρουσιάστηκαν εξασφάλισαν ένα πρώτο επίπεδο παραλληλισμού. Η απόδοση μπορεί να βελτιωθεί περαιτέρω από το συνδυασμό των προηγούμενων τεχνικών με τις ενσωματωμένες αυτόματες τεχνικές βελτιστοποίησης που παρέχει το HLS και ονομάζονται directives.

Η επιλογή αυτών εξαρτάται από την εγγενή παραλληλία του αλγορίθμου και τον τρόπο με τον οποίο αυτή μπορεί να αξιοποιηθεί. Στον συγκεκριμένο ταξινομητή οι τεχνικές που επιλέγονται στοχεύουν στην παραλληλοποίηση του εσωτερικού βρόχου. Όπως έχει αναφερθεί τα τετράγωνα της διαφοράς των στοιχείων των διανυσμάτων εισόδου και υποστήριξης συνθέτουν την ευκλείδια νόρμα και μπορούν να υπολογιστούν παράλληλα. Αυτό προϋποθέτει την ξετύλιξη του βρόχου και οδηγεί στην ανάγκη ανάγνωσης περισσότερων από δύο στοιχείων κάθε πίνακα τη φορά. Απαιτείται έτσι η αλλαγή της δομής των πινάκων για να καταστεί αυτό δυνατό. Με το ίδιο σκεπτικό μπορούμε να ξετυλίξουμε τον εξωτερικό βρόχο και να τροποποιήσουμε τους πίνακες που προσπελάζονται σε αυτό για να αυξήσουμε την παραλληλία. Ακολουθούν τα directives και ο λόγος για τον οποίο επιλέχθηκαν.

Pipeline: Αυτή η τεχνική εφαρμόζεται σε όλους τους βρόχους. Οι πράξεις των επαναλήψεων εκτελούνται παράλληλα κι όχι σειριακά χρησιμοποιώντας όλους τους πόρους κάθε χρονική στιγμή.

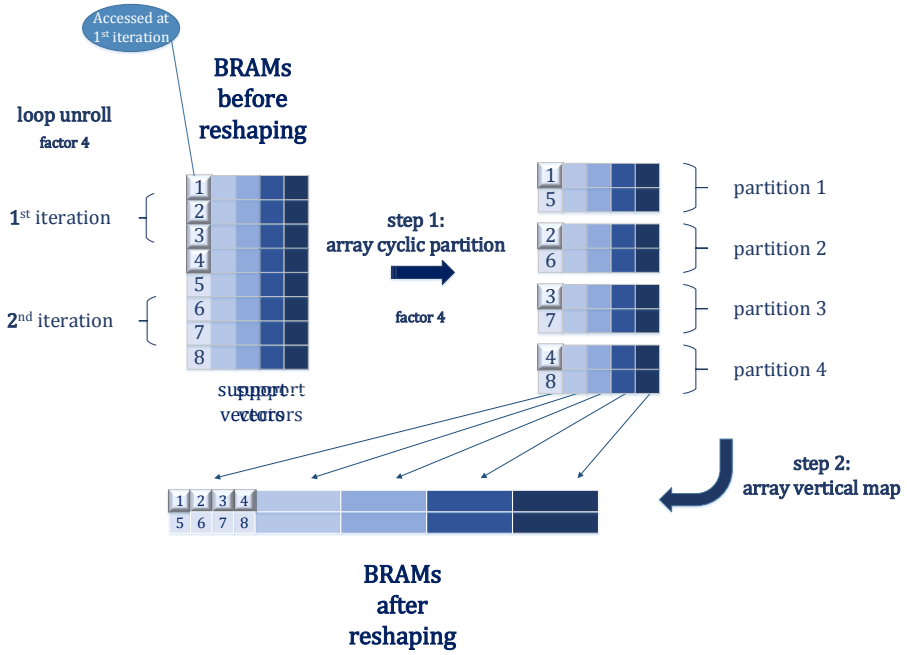
Εκτύλιξη βρόχου: Εφαρμόζεται σε όλους τους βρόχους. Δημιουργούνται αντίγραφα του σώματος του βρόχου ενώ μειώνεται ο αριθμός εκτελέσεων.

Διαίρεση Πίνακα: Εφαρμόζεται στους πίνακες sup_vector και sv_coef arrays. Διαιρεί τους πίνακες σε πίνακες μικρότερου μεγέθους κι άρα αυξάνεται ο αριθμός των θυρών ανάγνωσης.Έτσι όταν ξετυλίγεται ο εσωτερικός βρόχος είναι δυνατή η πρόσβαση σε περισσότερα από δύο στοιχεία του πίνακα και άρα μπορεί να παραλληλοποιηθεί ο υπολογισμός της ευκλείδιας απόστασης των διανυσμάτων. Η διαίρεση γίνεται κυκλικά (ανά κάποιο παράγοντα τα στοιχεία ανήκουν στην ίδια υποδιαίρεση πίνακα) ώστε να είναι δυνατή η ταυτόχρονη πρόσβαση σε διαδοχικά στοιχεία του αρχικού πίνακα με τη σειρά που αυτά χρειάζονται και στο βρόχο. Στο Σχ.5 απεικονίζεται η διαίρεση του πίνακα.



Σχήμα 5: Σχηματική Αναπαράσταση της Διαίρεσης Πίνακα

Μορφοποίηση Πίνακα: Εφαρμόζεται στους ίδιους πίνακες με την προηγούμενη τεχνική και για τον ίδιο σκοπό. Η διαφορά είναι ότι οι μικρότεροι πίνακες ενώνονται και πάλι σε έναν πίνακα ώστε ένα στοιχείου του νέου πίνακα να αποτελείται από όλα τα αντίστοιχα στοιχεία των μικρότερων πινάκων. Έτσι μειώνεται ο αριθμός των BRAM ενώ παράλληλα με μια πρόσβαση έχουμε στη διάθεσή μας περισσότερα στοιχεία. Η τεχνική απεικονίζεται στο Σχ.6



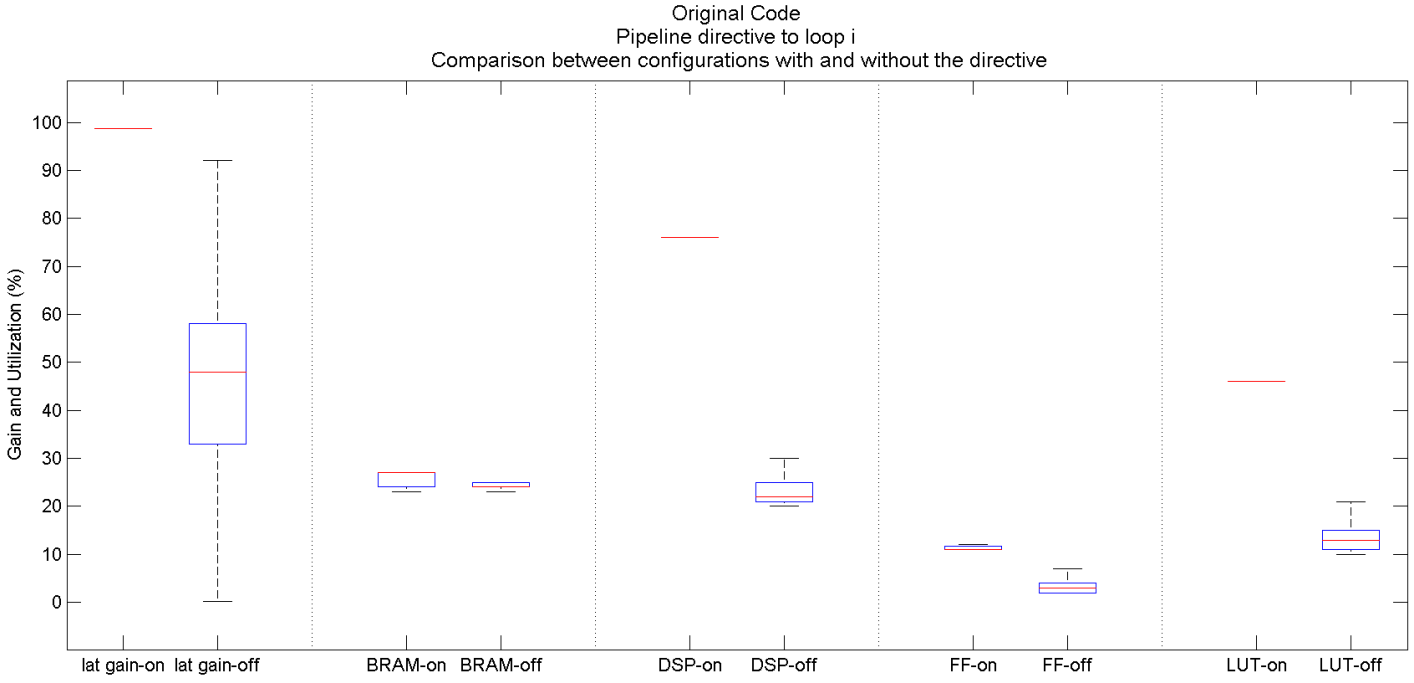
Σχήμα 6: Σχηματική Αναπαράσταση της Μορφοποίησης Πίνακα

Έχουν διερευνηθεί όλοι οι συνδυασμοί των επιλεγμένων τεχνικών που έχουν νόημα και αυτοί που δεν αποκλείονται λόγω δικών μας παραδοχών. Επίσης κάθε τεχνική εξετάζεται και ως προς την αλλαγή των τιμών των παραμέτρων της. Οι τελικοί συνδυασμοί που προκύπτουν εφαρμόστηκαν σε τέσσερις

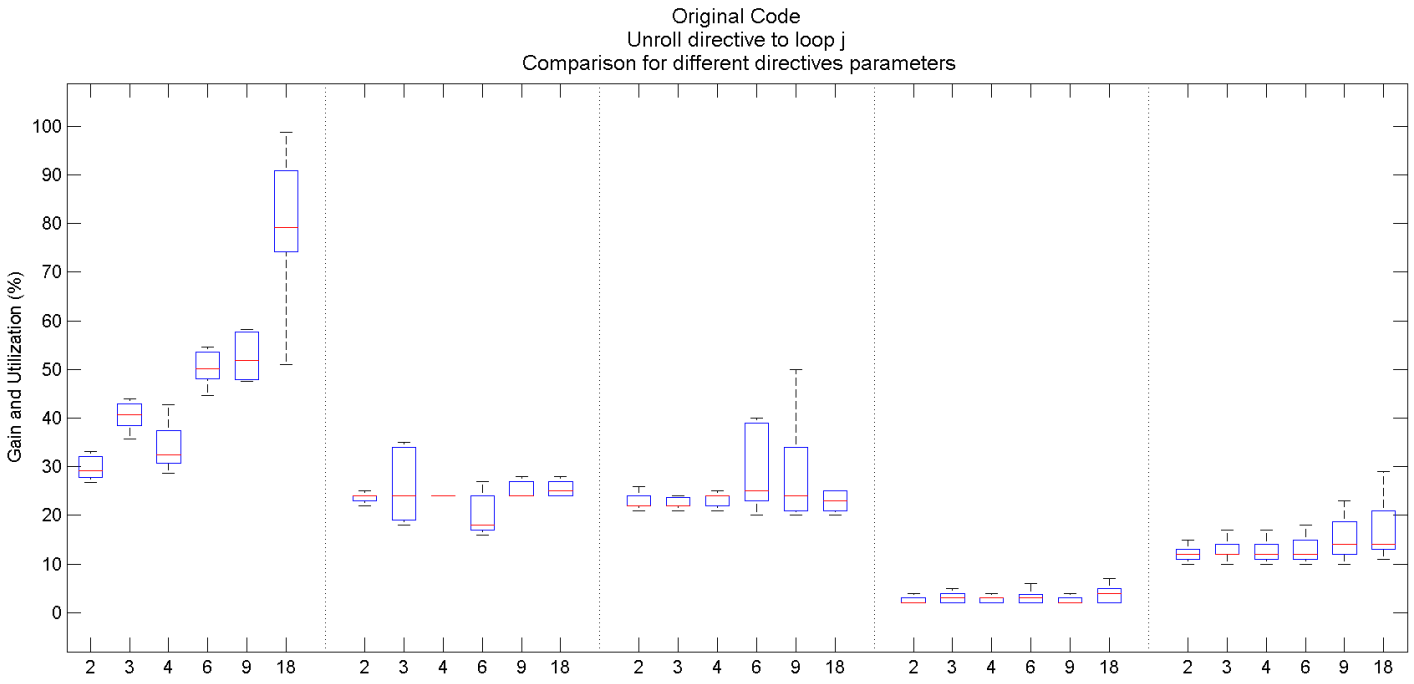
εκδοχές του αλγορίθμου:την αρχική και τρεις εκδοχές στις οποίες έχει εκτυλιχθεί ο εσωτερικός βρόχος με τον τρόπο που προτάθηκε στο Κεφάλαιο κατά 3, 6 και 18 φορές αντίστοιχα.

Αποτελέσματα

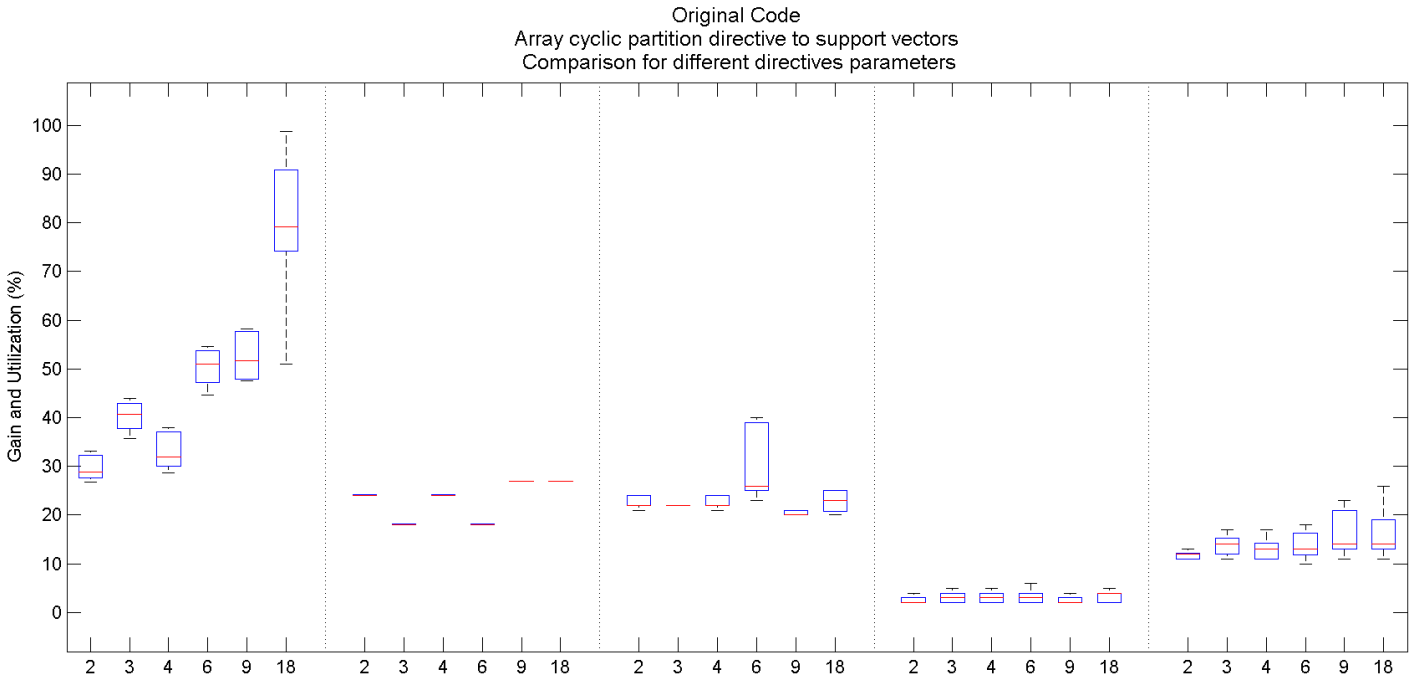
Στα Σχ.7-9 αναπαριστώνται τα αποτελέσματα των βασικότερων τεχνικών που χρησιμοποιήθηκαν.



Σχήμα 7: Pipeline εξωτερικού βρόχου

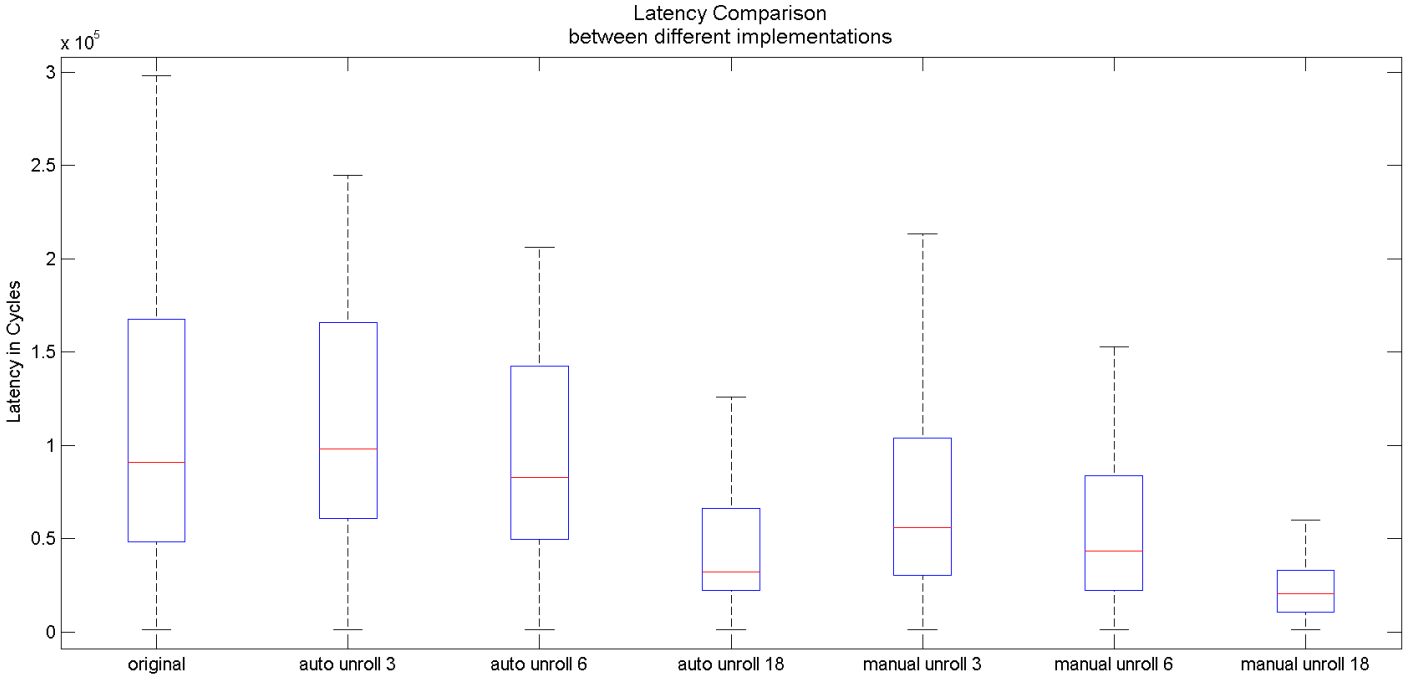


Σχήμα 8: Εκτύλιξη εσωτερικού βρόχου κατά διαφορετικό παράγοντα. Στήλες αριστερά προς τα δεξιά:κέρδος χρόνου εκτέλεσης, BRAM,DSP,FF,LUT.



Σχήμα 9: Διαίρεση πίνακα support vector. Στήλες αριστερά προς τα δεξιά:κέρδος χρόνου εκτέλεσης, BRAM,DSP,FF,LUT

Η διαφορά στα αποτελέσματα ανά διαφορετική υλοποίηση φαίνεται στο Σχ.10, όπου απεικονίζεται ο

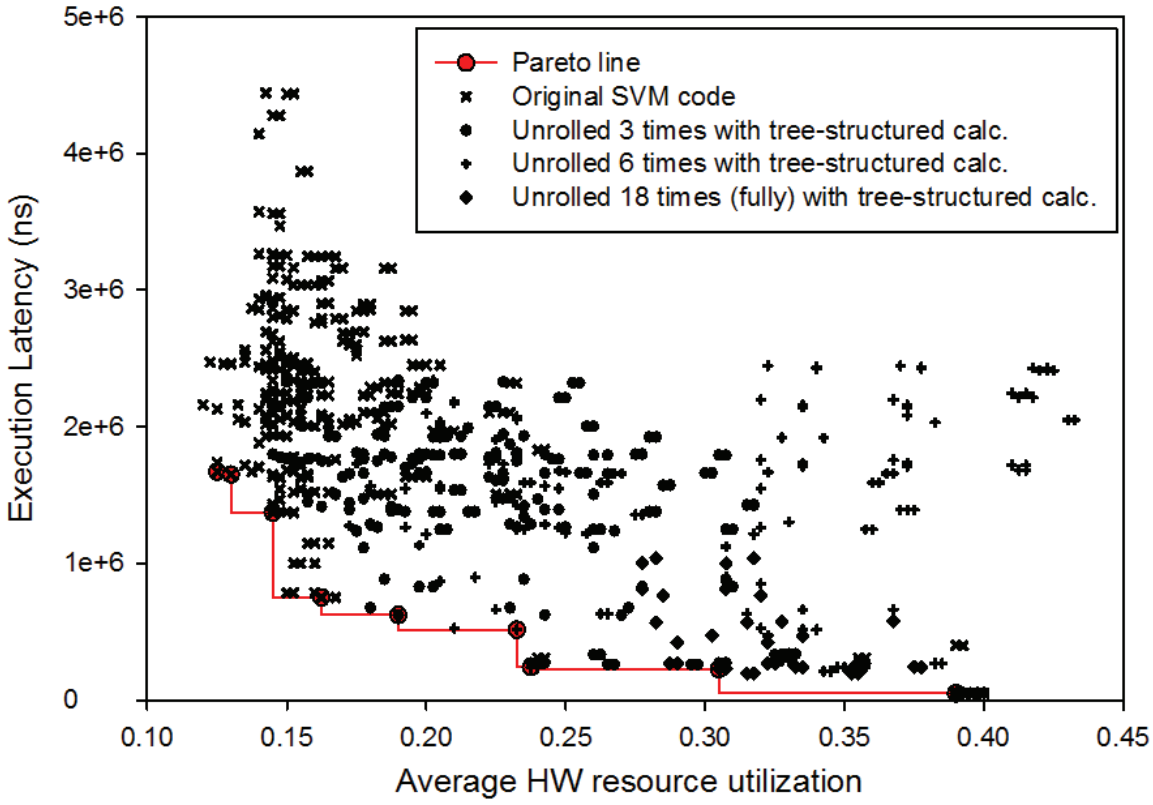


Σχήμα 10: Σύγκριση επιτάχυνσης μεταξύ όλων των υλοποιήσεων

χρόνος εκτέλεσης σε κύκλους ανά υλοποίηση. Η πρώτη υλοποίηση είναι η αρχική. Ακολουθούν τρεις εκδοχές αυτής που έχουν δημιουργηθεί έχοντας εκτυλίξει τον εσωτερικό βρόχο με τον αυτόματο τρόπο που παρέχεται από το εργαλείο. Οι τρεις επόμενες αντιστοιχούν στις εκτυλίξεις του βρόχου με τον προτεινόμενο τρόπο που συνδυάζει την εκτύλιξη με δενδρική υλοποίηση των αριθμητικών υπολογισμών. Είναι προφανές ότι η εκτύλιξη του εσωτερικού βρόχου έχει καθοριστικό αντίκτυπο στη μείωση του χρόνου εκτέλεσης. Όσο μεγαλύτερος είναι ο παράγοντας εκτύλιξης τόσο πιο έντονη είναι η επίδρασή του. Τα boxplot για μεγαλύτερους παράγοντες είναι μετατοπισμένα σε χαμηλότερους χρόνους, έχουν μικρότερο εύρος ενώ ο μέσος έχει φθίνουσα πορεία. Είναι ακόμα προφανές ότι ο προτεινόμενος τρόπος εκτύλιξης είναι πιο αποδοτικός. Επιτυγχάνει ακόμα μικρότερο εύρος boxplot, τιμές που κυμαίνονται σε χαμηλότερους χρόνους και μικρότερους μέσους. Είναι χαρακτηριστικό ότι η προτεινόμενη υλοποίηση για εκτύλιξη κατά 3 έχει καλύτερα αποτελέσματα από την αυτόματη κατά παράγοντα 6.

Ιδιαίτερα σημαντικό είναι το ότι παρέχεται στο σχεδιαστή μια μεγάλη ποικιλία αρχιτεκτονικών επιλογών, από τις οποίες μπορεί να διαλέξει ανάλογα με τις απαιτήσεις και τους περιορισμούς της εκάστοτε εφαρμογής. Στο Σχ.11 παρουσιάζεται ο χώρος εξερεύνησης για όλες τις εκδοχές υλοποιήσεων που χρησιμοποιήσαμε. Στον άξονα X μετράται η χρησιμοποίηση πόρων από κάθε προτυτοποίηση ενώ στον άξονα Υ η απόδοση ως προς το χρόνο εκτέλεσης. Η κάθε προτυτοποίηση έχει διαφορετικά χαρακτηριστικά. Για παράδειγμα τα βέλτιστα σημεία προς τα αριστερά εξασφαλίζουν ελάχιστη χρησιμοποίηση πόρων και ικανοποιητική επιτάχυνση. Αυτά προκύπτουν κυρίως από τον αρχικό κώδικα που δεν είναι εξίσου βελτιστοποιημένος με τους άλλους. Ο χρόνος εκτέλεσης είναι μικρότερος αλλά λόγω απλής δομής απαιτείται περιορισμένος αριθμός πόρων. Το αντίθετο συμβαίνει με τους τροποποιημένους κώδικες,

που επιτυγχάνουν υψηλά επίπεδα επιτάχυνσης με μια επιβάρυνση όμως στη χρησιμοποίηση πόρων.



Σχήμα 11: Καμπύλη Pareto

Υλοποίηση στην πλακέτα

Ακολουθεί η περιγραφή της υλοποίησης της εφαρμογής στην πλακέτα και τα αποτελέσματα αυτής. Τα βήματα που ακολουθήσαμε και τα εργαλεία που χρησιμοποιήθηκαν σε κάθε σχήμα είναι τα εξής:

- **High Level Synthesis**: Σε αυτό το στάδιο γίνεται η κατασκευή του επιταχυντή. Χρησιμοποιείται το εργαλείο Vivado High Level Synthesis 2014.4 της Xilinx. Το εργαλείο λαμβάνει ως είσοδο κώδικα C, τροποποιημένο πρώτα από το σχεδιαστή ή όχι, περιορισμούς του χρήστη και directives. Η έξοδος είναι ο επιταχυντής σε γλώσσα περιγραφής υλικού. Μετά από εξερεύνηση ο σχεδιαστής καταλήγει στην προτυτοποίηση που πληροί τις απαιτήσεις και τις προδιαγραφές της εφαρμογής του. Εκτός από τα directives που έχουν σκοπό την βελτιστοποίηση του επιταχυντή, εφαρμόζονται και directives για την ένταξη και επικοινωνία του επιταχυντή με ένα ευρύτερο αρχιτεκτονικό σύστημα και το υπολογιστικό σύστημα. Εξάγεται λοιπόν ως έξοδος ο επιταχυντής και σε κατάλληλο φορμάτ ώστε να μπορεί να χρησιμοποιηθεί σε κατάλληλο περιβάλλον για τη δημιουργία της συνολικής αρχιτεκτονικής.

- **Κατασκευή πλήρους αρχιτεκτονικής**: Σε αυτό το στάδιο κατασκευάζεται η πλήρης αρχιτεκτονική του συστήματος, που αποτελείται από το Υπολογιστικό Σύστημα, τον επιταχυντή στην Προγραμματιστική Λογική και το δίαυλο επικοινωνίας αυτών μεταξύ τους. Χρησιμοποείται το Xilinx Vivado Design Suite 2014.4. Στο τέλος της διαδικασίας παράγεται το hw description file.

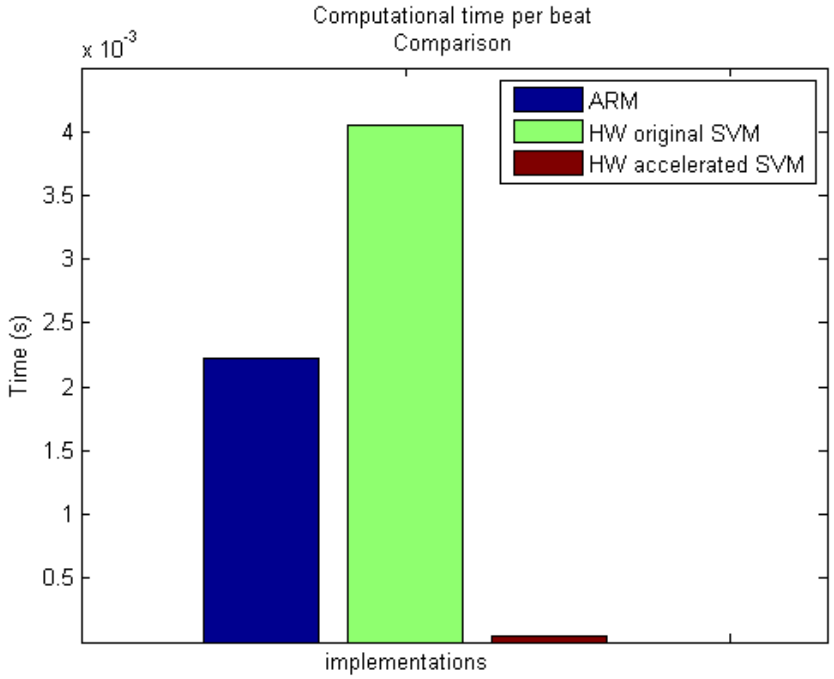
- **Δημιουργία Λειτουργικού Συστήματος**: Το επόμενο βήμα είναι η δημιουργία λειτουργικού συστήματος linux που να εκκινεί στην πλακέτα. Χρησιμοποιείται το Petalinux 2014.4. Για το χτίσιμο του είναι απαραίτητες πληροφορίες για την αρχιτεκτονική του Zedboard και ό,τι υλοποιήθηκε από το χρήστη στο FPGA, επομένως χρειάζεται το hw description file που παρήχθηκε από το Vivado. Ο πυρήνας των linux μπορεί να προτυτοποιηθεί αναλόγως με τις απαιτήσεις του συστήματος. Στο τέλος αυτού του βήματος παράγεται το image του πυρήνα και το BOOT.BIN αρχείο οπότε μπορεί να εκκινήσει το λειτουργικό στην πλακέτα και να προτυποποιηθεί το FPGA.

- **Ανάπτυξη εφαρμογής**: Αυτό είναι το τελευταίο βήμα κατά το οποίο αναπτύσσεται η εφαρμογή. Τα χαρακτηριστικά κάθε παλμού στέλνονται στον επιταχυντή και αυτός επιστρέφει το αποτέλεσμα. Ο επιταχυντής ελέγχεται και χειρίζεται από το Υπολογιστικό Σύστημα ως συσκευή που γίνεται map σε χώρο διευθύνσεων του χρήστη και ανοίγει ως αρχείο.

Ακολουθώντας τα παραπάνω βήματα υλοποιούμε στην πλακέτα την αρχική και τη βέλτιστη υλοποίηση ενώ εκτελούμε και τον αλγόριθμο μόνο στο Υπολογιστικό Σύστημα. Η βέλτιστη υλοποίηση είναι πιο γρήγορη και από τον αρχική και από αυτήν που τρέχει μόνο σε sw. Τα αποτελέσματα είναι κοντά στα θεωρητικά αναμενόμενα.

Πίνακας 3: Μετρήσεις χρόνου για τις υλοποιήσεις

	SW		HW αρχική		HW βέλτιστη	
	Χρόνος Επικοινωνίας (δευτερόλεπτα)	Χρόνος Υπολογισμού (δευτερόλεπτα)	Χρόνος Επικοινωνίας (δευτερόλεπτα)	Χρόνος Υπολογισμού (δευτερόλεπτα)	Χρόνος Επικοινωνίας (δευτερόλεπτα)	Χρόνος Υπολογισμού (δευτερόλεπτα)
ανά παλμό	-	0.002223635	0.00000449943	0.004047181	0.0000110643	0.0000521259
συνολικά	-	116.2761016	0.2352798	211.6311248	0.5785634	2.7257132



Σχήμα 12: Χρόνος υπολογισμών για SW και HW υλοποιήσεις.

Acknowledgments

The current thesis is the result of my work in collaboration with the Microprocessors and Digital Systems Laboratory (MicroLab) of NTUA. I would like to thank my supervisor, Prof. Dimitrios Soudris for the trust he showed in me and for his guidance and encouragement throughout the conduction of the thesis. The educational opportunities that he offered me undoubtedly helped me evolve both on a professional and personal level. I would also like to sincerely thank Postdoctoral Researcher Sotiris Xydis for his contribution and precious guidance. His insightful comments and constructive criticism were determinative for the completion of my work. I am also grateful to Doctoral Student Vasileios Tsoutsouras for his invaluable contribution. His guidance and advice as well as his constant support and encouragement helped me cope with the challenges that occurred. Both their work has been a true inspiration for me. I would also like to thank all the members of the laboratory with whom I have interacted during the course of my thesis conduction. Their friendliness and sincere will to help have made these last months one of the fondest memories I have from my studies in NTUA.

I would also like to thank my fellow students and especially my closest friends for being an integral part of my life during these last five years. Finally I would like to wholeheartedly thank my family for their love and support throughout all the challenges of my life.

List of Figures

1 Ροή Ανάλυσης ΗΚΓ ix

2 Παραλληλισμός σε Επίπεδο Μπλοκ xv

3 Απόδοση και Χρησιμοποίηση Πόρων για αυξανόμενο αριθμό διαμερίσεων (μη αυτόματη) xv

4 Δενδρικής δομής υπολογισμοί και χρονοδρομολόγηση. xvii

5 Σχηματική Αναπαράσταση της Διαίρεσης Πίνακα xxi

6 Σχηματική Αναπαράσταση της Μορφοποίησης Πίνακα xxi

7 Pipeline εξωτερικού βρόχου xxii

8 Εκτύλιξη εσωτερικού βρόχου κατά διαφορετικό παράγοντα. Στήλες αριστερά προς τα δεξιά:κέρδος χρόνου εκτέλεσης, BRAM,DSP,FF,LUT. xxiii

9 Διαίρεση πίνακα support vector. Στήλες αριστερά προς τα δεξιά:κέρδος χρόνου εκτέλεσης, BRAM,DSP,FF,LUT xxiii

10 Σύγκριση επιτάχυνσης μεταξύ όλων των υλοποιήσεων xxiv

11 Καμπύλη Pareto xxv

12 Χρόνος υπολογισμών για SW και HW υλοποιήσεις. xxvii

2.1 Heart Physiology [1] 4

2.2 ECG Waveform Typical Morphology [2] 5

2.3 ECG analysis flow 6

2.4 Classification energy scales with N_sv [3]. 7

2.5 Classification energy scales with D_sv [3]. 7

3.1 SVM based classification 12

3.2 Algorithmic C to Co-Processing Accelerator Integration [4] 13

3.3 Vivado HLS Tiered Verification Flow [4] 14

3.4 HLS Proposed Work Flow 15

4.1 Parallelism in Computations 19

4.2 Coarse Level Parallelism 20

4.3 Performance and utilization for increasing number of partitions (automatic) . 22

4.4 Performance and utilization for increasing number of partitions (manual) . . 23
4.5 Speedup gain comparison (automatic vs manual) 24
4.6 Tree based computations for manual unrolling and HLS scheduling 26
4.7 Scheduling Comparison between manual and automatic unrolling 26
4.8 Latency in Cycles for Automatic vs Manual Unroll. 29
4.9 Latency Gain for Automatic vs Manual Unroll. 29
4.10 BRAM Utilization for Automatic vs Manual Unroll 29
4.11 DSP Utilization for Automatic vs Manual Unroll 29
4.12 Flip Flop Utilization for Automatic vs Manual Unroll 29
4.13 LUT Utilization for Automatic vs Manual Unroll 29

5.1 Array partition schematically 31
5.2 Array reshape schematically 32
5.3 Pipelining loop_i. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 33
5.4 Unrolling loop_i. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 34
5.5 Changing unroll factor on loop_i. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization. 35
5.6 Pipelining loop_j. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 35
5.7 Unrolling loop_j. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 36
5.8 Changing unroll factor on loop_j. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization. 36
5.9 Partitioning sv_coef. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 37
5.10 Changing partition factor on sv_coef. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization. 38
5.11 Reshaping sv_coef. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 38
5.12 Changing reshape factor on sv_coef. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization. 39
5.13 Partitioning sup_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 40

5.14 Changing partition factor on sup_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization. 40

5.15 Reshaping sup_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 41

5.16 Changing reshape factor on sup_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization. 41

5.17 Partitioning test_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 42

5.18 Changing partition factor on test_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization. 42

5.19 Reshaping test_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive. 43

5.20 Changing reshape factor on test_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization. 43

5.21 Latency comparison between all implementations 44

5.22 BRAM Utilization comparison between all implementations 45

5.23 DSP Utilization comparison between all implementations 46

5.24 FF Utilization comparison between all implementations 46

5.25 LUT Utilization comparison between all implementations 47

5.26 Optimal Configurations vs the original implementations 48

5.27 Pareto Curve 49

6.1 Implementation Flow 51

6.2 System Architecture Build in Vivado Design Suite 53

6.3 Customized IP in Vivado 53

6.4 Computation time for HW and SW implementations. 55

6.5 Gain comparison between simulation and implementation 56

List of Tables

1 Διαθέσιμοι πόροι του Zedboard. xiii

2 Σύγκριση μετρικών μεταξύ αυτόματης και προτεινόμενης εκτύλιξης βρόχου . . . xix

3 Μετρήσεις χρόνου για τις υλοποιήσεις xxvii

3.1 HLS directives [5] 15

3.2 Zedboard Available Resources 17

4.1 Evaluated metrics for automatic vs manual unrolling 28

5.1 Applied directives and their parameters 32

6.1 Time measurements for different implementations. 55

6.2 Simulation vs Implementation Results 56

Chapter 1
Introduction

One of the most fundamental and crucial biological signals for monitoring and assessing the health condition of a person is the Electrocardiogram (ECG) [6]. ECG has an inherent relation to heart physiology. Consequently, its analysis and interpretation has been established as an important field in modern medicine and this in turn has spawned various inter-disciplinary studies including digital processing analysis of the signal. Given the complexity of deriving exact models for assessing and predicting the heart's condition, machine learning techniques have recently dominated the field of ECG analysis. Support Vector Machine [7] based classifiers specifically have grown to be the key element of machine learning based ECG analysis due to their capability of accurate prediction. In addition, constant monitoring and real-time heart condition assessment have imposed new requirements for acceleration and low power execution of a digital ECG analysis flow. In this work, we focus on Arrhythmia [8] detection in a real-time operating ECG analysis flow, using Support Vector Machines for beat classification and design tools for building efficient hardware accelerators for the classification module.

Support Vector Machines are widely used as classifiers in many systems and there is much work in their incorporation in ECG arrhythmia detection flow [9]. The popularity behind these classifiers is twofold. On the one hand they ensure very high classification accuracy even in problems that manifest complex non-linear distribution in the extracted features space. On the other hand their structure, based on stencil computation operations, forms a perfect candidate for applying acceleration techniques. This has been shown in [3] where it has been proven that when the classification problem becomes very complex and the computational requirements of SVM classifier are multiplied, HW acceleration can be the key for meeting both time and power constraints of the ECG signal analysis flow. Towards this direction, they propose a custom HW architecture with which the algorithm is efficiently accelerated.

Hardware acceleration is a fundamental technique for creating systems which execute complex algorithms efficiently with reduced power budget constraints. However, designing a HW accelerator in HDL can be a time consuming and difficult to debug task. An approach on mitigating these issues is High Level Synthesis [10] where the functional description of an algorithm in the form of a C function is automatically translated into a hardware description. A HW accelerator in the majority of the cases is more efficient compared to an HLS defined one, however HLS enables very quick HW implementation and exploration of a variety of architectural characteristics which is extremely difficult to achieve in HDL in a small amount of time. This is why HLS is ever increasingly utilized as HW design mechanism either for architectural exploration or fast prototype.

In this diploma thesis we focus on the last stage of the ECG analysis flow, the diagnosis classification. It has already been stressed that for monitoring chronic patients in respect to their heart's condition, a real-time ECG acquisition system has to be developed, usually implemented on wearable or implantable devices. This leads to inevitable timing, energy and resources constraints. Working towards meeting these specifications, we propose a coprocessor-based architecture in which the most time consuming and energy demanding part in the flow is implemented as a hardware coprocessor or accelerator. The ECG analysis

flow is implemented in software except for the Support Vector Machine classifier which is implemented as a hardware accelerator targeting FPGAs. This accelerator is built using High Level Synthesis design tool. HLS receives the functional description of the SVM classifier as input and translates it into RTL design. Moreover it provides standard techniques, called directives, that optimize the hardware description in terms of time, energy and area utilization efficiency. What is important to state is that available tools utilized in creating an HLS originated HW accelerator are not always capable of producing the best possible solution due to limitations in their analysis of the structure of the code. We have identified this problem and as a part of this work, we propose two strategies of manually restructuring the original code under acceleration to assist the HLS flow to fully utilize the inherent parallelization capabilities of the algorithms. These strategies have proven to be capable of producing very high gains in latency by only a small sacrifice in hardware resources. To further optimize the generated hardware description, we apply directives which are the HLS infrastructure to explore different architectural options and we examine how these directives affect the produced accelerator. The described architecture is implemented on the Zynq Evaluation and Development Board. The first stages of the flow are executed on the Processing System of the board and the executable communicates when necessary with the classifier that is implemented as an FPGA-accelerator by configurating the Programmable Logic of the board.

In Chapter 2, a brief overview of the ECG Analysis Flow is presented. The stages of the flow are explained to inform the reader of the steps required for successful arrhythmia detection and of the challenges emerging. Related work progress on the field is also included, to highlight the contribution of our suggested approach. In Chapter 3, the thesis presents the theoritical background of the Support Vector Machine classifier and the design tools and platforms used for its acceleration. The Support Vector Machine theory is described in a way that thoroughly explains its functionality and complexity and stresses the need for its efficient acceleration. High Level Synthesis design tool and its capabilities is also presented along with the specifications of the chosen development board. In Chapter 4 the developed strategies for code restructuring are demonstrated. The techniques followed in each strategy are explained in detail and measurements of the metrics of interest are provided. In Chapter 5 an exploration of HLS built-in optimization techniques is carried out using the initial code as well as the restructured ones as baseline. The selection of the optimizations applied is justified at length and the impact of their application is analysed. The chapter concludes with a comparative study of the results of all optimized implementations and of the optimal configurations that derived from the directives exploration. In the next chapter, Chapter 6, the implementation of the accelerator on the Zynq Evaluation and Development Board is presented and the measurements acquired are presented. In the final chapter, Chapter 7, important conclusions valuable to the reader are discussed, as well as ideas for improving and extending the existing work in the future.

Chapter 2
Problem Overview

2.1 ECG Analysis Flow

Electrocardiography is an important tool in diagnosing the condition of the heart. The electrocardiogram (ECG) is the record of variation of bioelectric voltage with respect to time as the human heart beats. The state of cardiac health is generally reflected in the shape of ECG waveform and heart rate.

The heart is a four-chambered organ consisting of right and left halves. The upper two chambers, the left and right atria, are entry-points into the heart, while the lower two chambers, the left and right ventricles, are responsible for contractions that send the blood through the circulation. The role of the right ventricle is to pump deoxygenated blood to the lungs through the pulmonary trunk and pulmonary arteries. The role of the left ventricle is to pump newly oxygenated blood to the body through the aorta. The physiology of the heart is depicted in Fig.2.1. The cardiac cycle refers to a complete heartbeat from its generation to the beginning of the next beat, and so includes the repetition of some stages. The first stage, " diastole," is when the semilunar valves (the pulmonary valve and the aortic valve) close, the atrioventricular (AV) valves (the mitral valve and the tricuspid valve) open, and the whole heart is relaxed. The second stage, "atrial systole," is when the atrium contracts, and blood flows from atrium to the ventricle. The third stage, "isovolumic contraction" is when the ventricles begin to contract, the AV and semilunar valves close, and there is no change in volume. The fourth stage, "ventricular ejection," is when the ventricles are contracting and emptying, and the semilunar valves are open. During the fifth stage, "isovolumic relaxation time", pressure decreases, no blood enters the ventricles, the ventricles stop contracting and begin to relax, and the semilunar valves close due to the pressure of blood in the aorta [11].

The cardiac cycle described above is coordinated by a series of electrical impulses that are produced by specialised pacemaker cells. A typical ECG tracing is a repeating cycle of three electrical entities: a P wave, a QRS complex that consists of the Q,R and S peaks and a T wave [12]. These waves are created by voltage fluctuations that depict the electrical activity of the heart and thus represent the cardiac cycle. Electrical systole of the atria begins with the onset of the P wave on the ECG. The wave of bipolarization (or depolarization) stimulates the cardiac muscle of the heart chambers to make them contract. This is soon followed by mechanical systole, which is the mechanical contraction of the heart. During atrial systole, the contraction of the atrial syncytium of cardiac muscle cells of the left and right atria forces additional blood into the ventricles. The electrical wave eventually reaches the atrioventricular node, leading to the emergence of the QRS complex. The electrical systole of the ventricles begins at the onset of the QRS complex. This electrical systole leads in turn to a mechanical systole, and thus the verticles contract in order to eject blood into circulation. Following their contraction, the ventricles recover and relax in preparation for refilling with circulating blood. The T wave represents the repolarization (or recovery) of the ventricles [13].

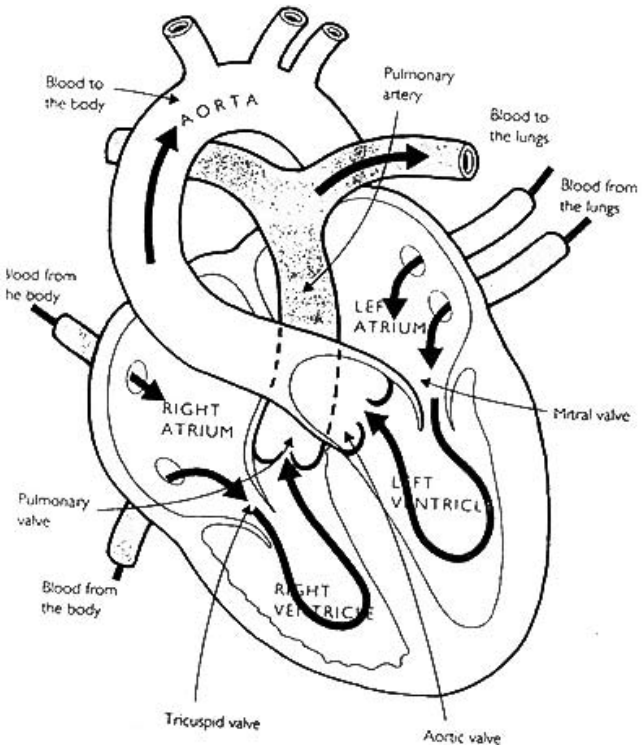


Figure 2.1: Heart Physiology [1]

All of the waves on the ECG and the intervals between them have a predictable time duration, a range of acceptable amplitudes (voltages), and a typical morphology. This morphology is depicted in Fig.2.2. Any deviation from the normal tracing is potentially pathological and therefore of clinical significance. Arrhythmia is considered as one of the most commonly encountered heart malfunctions. Cardiac arrhythmia, also known as cardiac dysrhythmia or irregular heartbeat, is a group of conditions in which the heartbeat is irregular, too fast, or too slow. Some arrhythmias do not cause symptoms, and are not associated with increased mortality. However, some asymptomatic arrhythmias are associated with adverse events. Examples include a higher risk of blood clotting within the heart and a higher risk of insufficient blood being transported to the heart because of weak heartbeat. Other increased risks are of embolisation and stroke, heart failure and sudden cardiac death. Medical assessment of the abnormality using an electrocardiogram is one way to diagnose and assess the risk of any given arrhythmia [14].

Taking into account the critical condition of a person suffering from arrhythmia episodes, the field of detecting signs of arrhythmia in an ECG signal has been highly investigated. Since the ECG is a non stationary signal, arrhythmia incidences may occur at random in the time scale. Thus, the disease symptoms may not show up all the time, but manifest at certain irregular intervals during the day. Therefore, for effective diagnosis, the study of ECG pattern and heart rate variability signal may have to be carried out over several hours. This means that an enormous data set needs to be processed in order for a diagnosis to be reached. As a result machine learning techniques [15] are ideal for solving the diagnosis problem. The data set is used as the training set required by machine learning solutions, that can deliver a diagnosis after their training is completed.

There is a number of open databases of ECG signals to choose from in order to form the training set. A well-known and rather frequently used database as such is the MIT-BIH Arrhythmia Database [16] – a combined effort of MIT and Beth Israel Deaconess Medical Center. This database is composed of 48 half-hour two-lead ECG signals with the collabora-

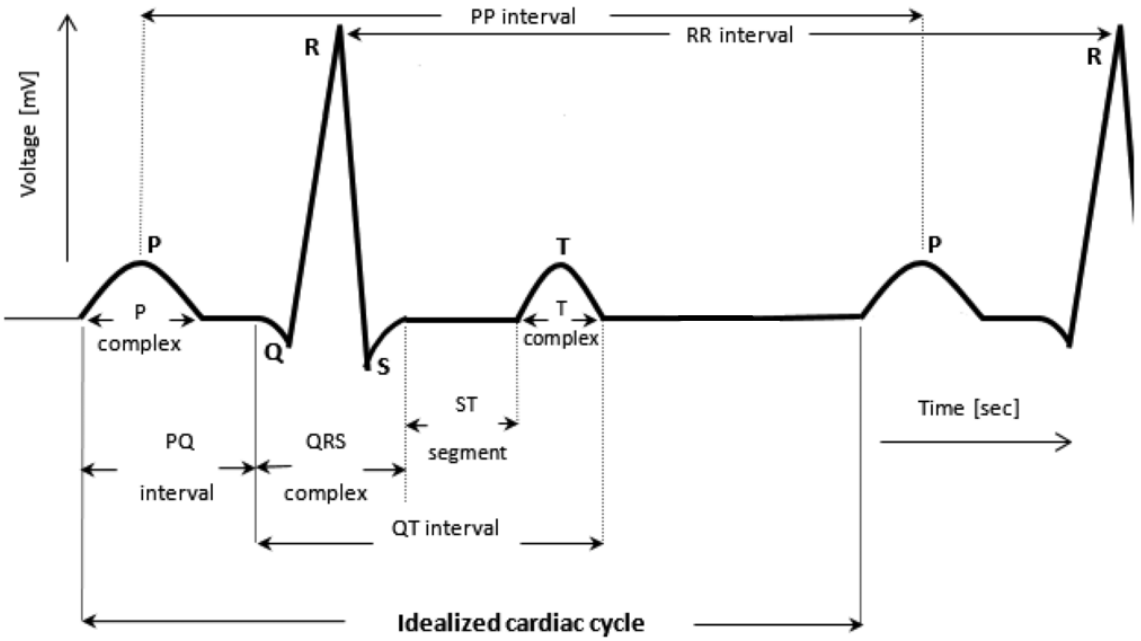


Figure 2.2: ECG Waveform Typical Morphology [2]

tion of patients of different medical files and physiology characteristics. Additionally, all the signals of this database have been fully annotated by medical experts. The heart beats of the half-hour ECG signal were initially detected by a simple beat detector and then verified by cardiologists working independently. The abnormal beats were identified and the type of the arrhythmia diagnosed was used to annotate the beats after the cardiologists reached consensus. As a result this database forms an ideal starting point for creating a training data set for the detection problem. A classifier based on machine learning techniques can be trained using this data set and can then be used to detect arrhythmia in individual beats, ideally in a real time system.

The process of acquiring and processing an ECG signal in order to extract these individual beats and their corresponding features is composed of various stages with distinct characteristics and requirements. It consists of three main stages: a preprocessing stage(noise removal),a processing stage(R peak detection,feature extraction), and a classification stage. A simplified overview of this processing flow is depicted in Fig. 2.3 where the following key features are included:

- **Noise removal**: In this stage the signal is filtered for noise removal, usually using a band-pass filter. The artifact signals that are removed include baseline wander, power line interference and high-frequence noise. Artifacts resulting from patient breathing and movement also have to be removed. The filtered ECG signals are used in all subsequent processing.
- **R peak detection**:In this stage the ultimate goal is to detect the heart beats that compose an ECG signal of a longer duration. In order to do that usually peaks of the QRS complex have to be identified and possibly P wave, T wave and QRS onsets and offsets [17]. The heart beat exact location and duration can be deducted from this information. MIT-BIH Arrhythmia Database [16] provides the user with function implementations that locate these fiducial points . These functions can be applied to the ECG signals available and the results can be verified with the help of the doctors' annotation files. That way we manage to isolate beats and thus construct the beats which will be included in the training data set. In a real-time ECG acquisition system,

the heart beat is not known a priori and points of interest such as the R peak have to be defined relying solely on the QRS detectors available in order for a new heart beat to be identified. Smaller fractions of the ECG signal are now processed for beat segmentation and as a result fewer beats are detected each time.

– **Feature extraction process**: Having determined a new heart beat, a feature extraction process is imposed on it in order to extract its characteristics. These characteristics refer to specific signal parameters that are indicative of the physiological state of interest. For arrhythmia detection there is a variety of types of characteristics that can be used, each of them introducing various clinical trade-offs .
The most complete way to display the information included in the ECG signal is to perform spectral analysis. The wavelet transform (WT), an extension of the classic Fourier transform, can be applied to extract the wavelet coefficients of discrete time signals, such as the ECG. The WT works on both time and frequency domain and allows the decomposition of a signal into a number of scales, each scale representing a particular coarseness of the signal under study. WT also has the ability to compute and manipulate data in compressed parameters which are called features. Thus using the WT transform, the ECG signal, consisting of many data points, can be compressed into a few parameters that characterize its behaviour and are not apparent from the original time domain signal. The heart beats detected at the previous stage, are decomposed into time–frequency representations using discrete wavelet transform (DWT) and wavelet coefficients are calculated to represent the signals [9]. The outputs derived at this stage form the feature vectors that are used for classification.

– **Diagnosis classification**: The final stage of the analysis flow is actually detecting whether the heart beat exhibits arrhythmia signs or not. This is performed using a classification algorithm, which detects the pattern of problematic beat. The classifier has been trained on the data set that includes the feature vectors of the isolated beats. Given a new feature vector the classifier can decide on whether the corresponding beat displays signs of arrhythmia.

  Support Vector Machines [7] are the machine-learning based classifiers that are used in this study. SVMs are popular machine-learning classifiers for data-driven modeling and classification and can be efficiently trained offline. Their training process results in a set of vectors, called support vectors (SVs), which are used to model the data by representing a decision boundary. This decision boundary is then used to classify a new instance, the feature vector of an unclassified beat. The number of support vectors and the feature-vector dimensionality can have a major impact on classifier complexity [3].

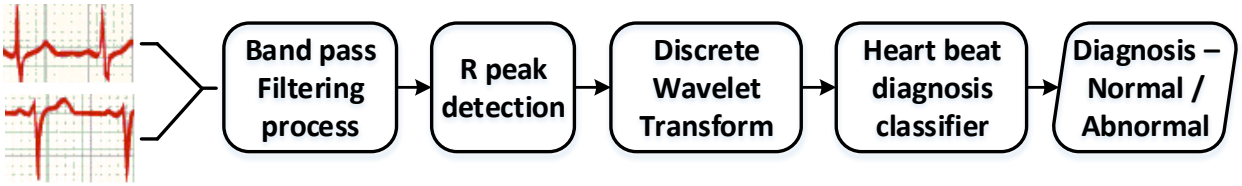


Figure 2.3: ECG analysis flow

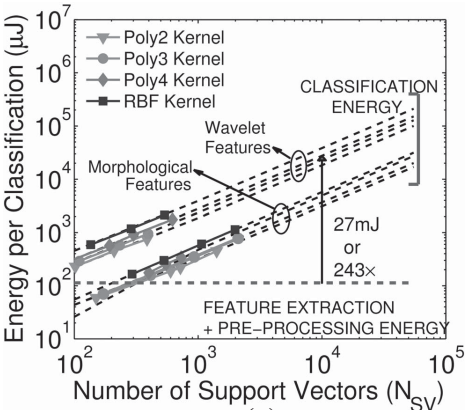


Figure 2.4: Classification energy scales with N_sv [3].

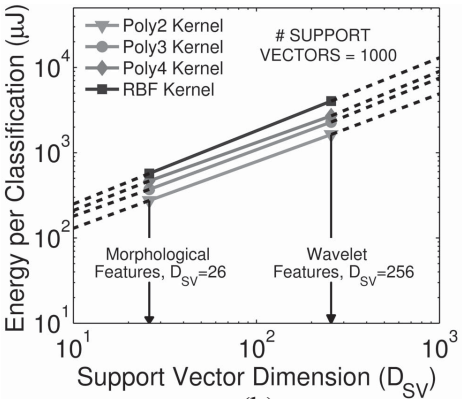


Figure 2.5: Classification energy scales with D_sv [3].

2.2 Related Work

Most biomedical devices used for monitoring chronic patients and detection of abnormalities in biomedical signals aim to provide accurate results in real time. To achieve that they need to process an enormous amount of signal data with extremely complex correlations. On this ground in [3], the writers address these issues by proposing an algorithm-driven architectural design space exploration of domain-specific medical-sensor processors. According to them, data-driven modeling techniques are emerging as a powerful approach for overcoming the mentioned challenges [18] since there has been significant development of machine-learning techniques that are capable of exploiting large amounts of data to model specific correlations and then use the models within a decision function [19]. The biodemical devices though are mainly wearable, thus they require very low power levels. To that direction the writers propose to employ application-specific architectures for low energy [20], [21]. As a case study they use arrhythmia detection and the ECG analysis flow explained in section 2.1. They also use Support Vector Machine based classifiers in the classification stage and RBF (exponential) kernel function in the classification core since linear kernels are not suitable for the degree of complexity of the correlation of medical signals. The kernel function along with the number of support vectors and feature vector dimensionality can have a major impact on classifier complexity. This was proven by implementing the entire arrhythmia detection algorithm on an embedded low-power base processor using high-order detection models for accurate signal classification and performing energy analysis. Figures 2.4 and 2.5 show the energy of classification versus number of support vectors and feature vector dimensionality, respectively. It can be seen that, because of energy scaling, classification energy rapidly dominates that of feature extraction. It is found that classification poses the energy bottleneck due to the complexity of the models required.

Thus, in their study they aim to optimize the classification stage in terms of throughput and energy efficieny. To achieve that they explore the development of a co-processor based architecture suitable for the analysis flow of various biomedical signals that require classification. A general-purpose processor is employed for feature computation, while an optimized co-processor is employed for kernel-based SVM classification. The specifications for the platform are meeting the constraints for real-time detection, energy efficiency and flexibility so that it supports various biomedical applications. The architecture has three main blocks: buffers for the support and test vectors, MAC units and a programmable polynomial kernel core. The support vectors are loaded to the buffers after the offline training is finished and the test vectors are dynamically loaded to their respective buffers. Each MAC unit is re-

sponsible for the dot product computations of the SVM classifier between a test vector and the support vectors in one support vector buffer. Once multiplication over all the support vectors is complete, the dot products are multiplexed to the programmable polynomial kernel core, where a second-, third-, or fourth-order polynomial transformation is computed as well as the kernel of the SVM classifier selected (such as RBF,sigmoid, etc). The results are scaled and summed by a final accumulator whose output sign determines the classification result. The computations are performed on integer values. The real-time constraints are met by parallelizing the dot product computations with the use of multiple MAC units and energy efficiency is achieved through voltage scaling in the MAC units.

The approach in our thesis is similar to the above in respect of developing a hardware FPGA co-processor for the classification stage. However there are some distinct differences in the suggested approach. Our co-processor is built using High Level Synthesis Design tools and is intended for the arrhythmia detection study case. It is thus optimized for this case only. For that reason the architecture is fixed concerning the implementation of the kernel function and the data related to the SVM model (such as support vectors and their coefficients) are hardcoded into the bitstream instead of being loaded. Furthermore, operations are performed on floating point values, not integer. Most importantly, the focus of our work lies in optimizing the IP in terms of performance and an energy analysis is not performed. Performance optimization is achieved by exploiting the inherent parallelism of the algorithm. However instead of building separate hardware IPs that execute in parallel, we build one IP that is parallelized on its inside. To achieve parallelization we both modify the structure of the code to derive coarse level parallelism and increase instruction level parallelism, but also utilize the optimization directives of the tool. The coarse level parallelism of our work is based on the same principle as the one described above. This principle is the lack of data dependencies between the computations related to each support vector. The instruction level parallelism though is not exploited in cite and is based on the parallelism within these computations.

Chapter 3
Theoritical Background

3.1 Background Information on SVM classifier

Support Vector Machines (SVMs) in machine learning are supervised learning models that are used for data-driven modeling and classification. They are suitable only for binary classification problems. The classification process requires that the data is separated into training and testing test. The instances in the training set have the form of a feature vector consisting of the attributes that are being observed and a label indicating the class each instance belongs to. The instances in the testing set consist only of the attributes. The goal of the SVM classification technique is to train a model that can predict the label (class) of an instance of the testing set given only the attributes of the corresponding instance [22].

This goal is met thanks to the ability of the SVM to find a hyperplane that divides samples into two classes with the widest margin between them. A mapping function is used to project each feature vector of the training set to a feature space of higher dimension where the data will be easier to classify. The SVM is used to find the optimal hyperplane for classifying the data according to their attributes. This optimal hyperplane maximizes the distance between the hyperplane and the feature vectors that belong to each class and are closest to the hyperplane. These feature vectors closest to the hyperplane represent the decision boundary between the classes and are called support vectors. The distance between support vectors and a feature vector from the testing set is used to classify the new feature vector. The function that is used for computing the distance between this unclassified feature vector and the support vectors by firstly projecting them to a higher dimensional feature space is called kernel function [19]. The hyperplane decision function for classifying feature vector x is of the following form:

$$Class = sgn(\sum_{i=1}^{N\_sv}(y_i * a_i * K(\mathbf{x}, \mathbf{sup\_vector}_i)) - b) \tag{3.1}$$

where K is the kernel function, $\mathbf{x}$ is the feature vector, $\mathbf{sup\_vector}_i$ is the i-th support vector and y_i, a_i are values related to it and result from the classifier training process. Coefficient b is a bias value, also a result of the training process and is constant for all support vectors.

The kernel function is very important to the accurate prediction of testing data. Depending on the characteristics of the dataset, different kernel functions are able to provide the desired classification accuracy.

The most popular kernels are the following four:

- **linear**: $K(\mathbf{x}, \mathbf{sup\_vector}_i) = \mathbf{x}^T * \mathbf{sup\_vector}_i$
- **polynomial**: $K(\mathbf{x}, \mathbf{sup\_vector}_i) = (\gamma * \mathbf{x}^T * \mathbf{sup\_vector}_i + r)^d$
- **radial basis function (RBF)**: $K(\mathbf{x}, \mathbf{sup\_vector}_i) = exp(-\gamma||\mathbf{x} - \mathbf{sup\_vector}_i||^2)$
- **sigmoid**: $K(\mathbf{x}, \mathbf{sup\_vector}_i) = tanh(\gamma * \mathbf{x} * \mathbf{sup\_vector}_i + r)$

If the feature vectors of our data set were linearly separable, a linear kernel function could be used for classification. The test vector $\mathbf{x}$ could then be pulled out of the summation in 3.1, allowing the summation to be precomputed over all of the support vectors into a single decision vector. As a result, even if the number of support vectors N_sv scaled, the classification energy would remain constant. However, biomedical applications have shown to perform poorly when linear decision functions are used with medical datasets [3]. Non linear functions, such as high-order polynomials, RBFs, or sigmoidal kernels, are thus needed for acceptable classifier accuracies.

In this work, we turn our attention to RBF kernel function since the complex correlations between the attributes of our feature vector and the physiological states of interest typically require the flexibility afforded by non linear kernel functions. The advantage of the RBF kernel over the other non linear kernels is that RBF has fewer parameters and fewer numerical difficulties [22]. The RBF kernel for test vector x and the i-th support vector $\mathbf{sup\_vector}_i$ is defined as:

$$K(\mathbf{x}, \mathbf{sup\_vector}_i) = exp(-\gamma||\mathbf{x} - \mathbf{sup\_vector}_i||^2) \tag{3.2}$$

The combination of equations 3.1 and 3.2 provide us the final decision function 3.3 which will be the target of our HW acceleration process:

$$Class = sgn(\sum_{i=1}^{N\_sv}(y_i * a_i * exp(-\gamma||\mathbf{x} - \mathbf{sup\_vector}_i||^2)) - b) \tag{3.3}$$

Listing 3.1, provides the actual implementation of equation (3.3) in a C-language based form that will be used as the base for the construction of the HLS based HW accelerator. The code is also schematically depicted in Fig.3.1.

Listing 3.1: SVM original prediction code

```
const float sv_coef[N_sv];
const float sup_vectors[D_sv][N_sv];

void SVM_predict (int *y, float test_vector[D_sv]) {

loop_i:for (i=0; i<N_sv; i++){
                loop_j:for (j=0; j<D_sv; j++){
                        diff=test_vector[j]-sup_vectors[j][i];
                        norma = norma + diff*diff;
                }
                sum = sum + exp(-gamma*norma)*sv_coef[i];
                norma=0;
        }

        sum = sum - b;

        if (sum<0)
                *y = -1;
        else
                *y = 1;
}
```

In Listing 3.1 *sv_coef* stands for the product of y_i and α_i of equation 3.1. The reader should observe that the number of the support vectors N_sv and the length of the feature vector D_sv along with the kernel selected have a great impact on classifier complexity. In the presented case study, the training phase resulted in N_sv equal to 1274 and D_sv equal to 18. The training phase was conducted using the Matlab [23] interface of LIBSVM library for Support Vector Machines [24].

3.2 High Level Synthesis

High Level Synthesis (HLS) [10] is a design tool for generating application-specific IP from algorithmic C specification and thus allowing the designer to work at a higher level of abstraction, while creating high-performance hardware. It provides software developers with an easy way to accelerate the computationally intensive parts of their algorithms on a new compilation target, a Field Programmable Gate Array (FPGA). The FPGA provides a massively paralleled architecture with benefits in performance, cost and power over traditional processors. The main part of the application thus, is executed on the system's processor while a part of it is transformed into a Register Transfer Level (RTL) implementation that synthesizes into a FPGA. This approach is depicted in Fig.3.2.

The flow of using HLS is briefly descripted in Fig.3.3. The first step is to develop an algorithm at C-level. HLS then provides the ability of verification at C-level, which allows designers to validate the functional correctness of the algorithm faster than doing so in traditional hardware description languages. The next step is the synthesis of the C implementation into an RTL design. HLS offers the ability to control the C synthesis process through optimization directives allowing the creation of specific high-performance hardware implementations. It performs some optimizations by default and also allows the user to impose directives and constraints of his own choice. The primary output from Vivado HLS is generated at the

synthesis step and it is the implementation in RTL format. The RTL can be synthesized into a gate-level implementation and an FPGA bitstream file by logic synthesis. The RTL output from Vivado HLS is provided in the industry standard Hardware Description Language (HDL) formats of Verilog and VHDL. A version of the RTL implementation is also provided in SystemC. After synthesis, verification of the RTL implementation is possible, to ensure that the same results with the software implementation are being generated. At the last step, the RTL implementation is packaged into one of the available IP formats, so that it can be integrated into the hardware system [4], [5].

HLS always begins with the compilation of the functional specification. This step transforms the input description into a formal model that exhibits the data and control dependencies through a Control and Data Flow Graph (CDFG). Allocation, scheduling and binding are the steps that follow and are the processes at the heart of High-Level Synthesis. Allocation defines the type and the number of hardware resources needed. Components can be added

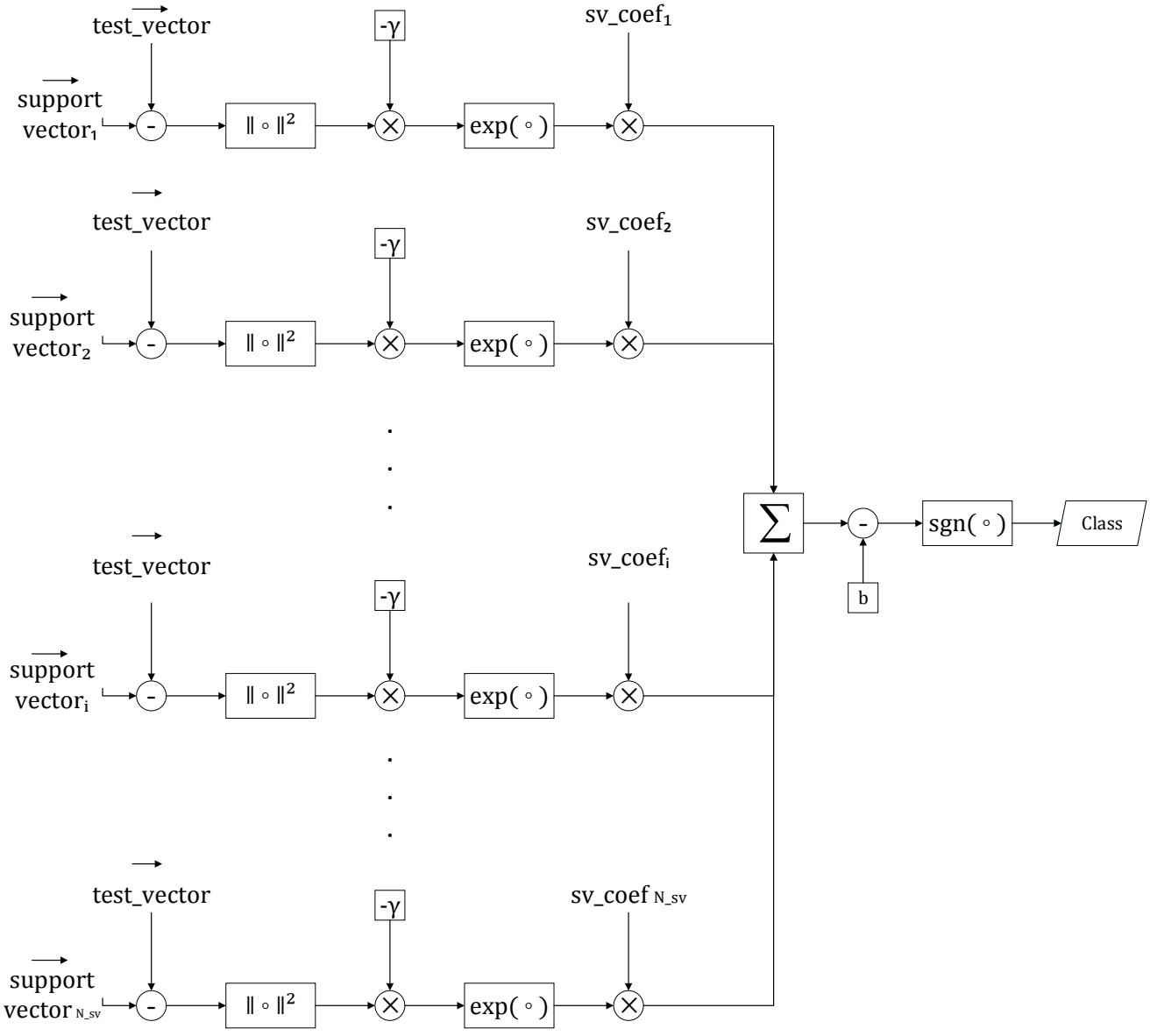


Figure 3.1: SVM based classification

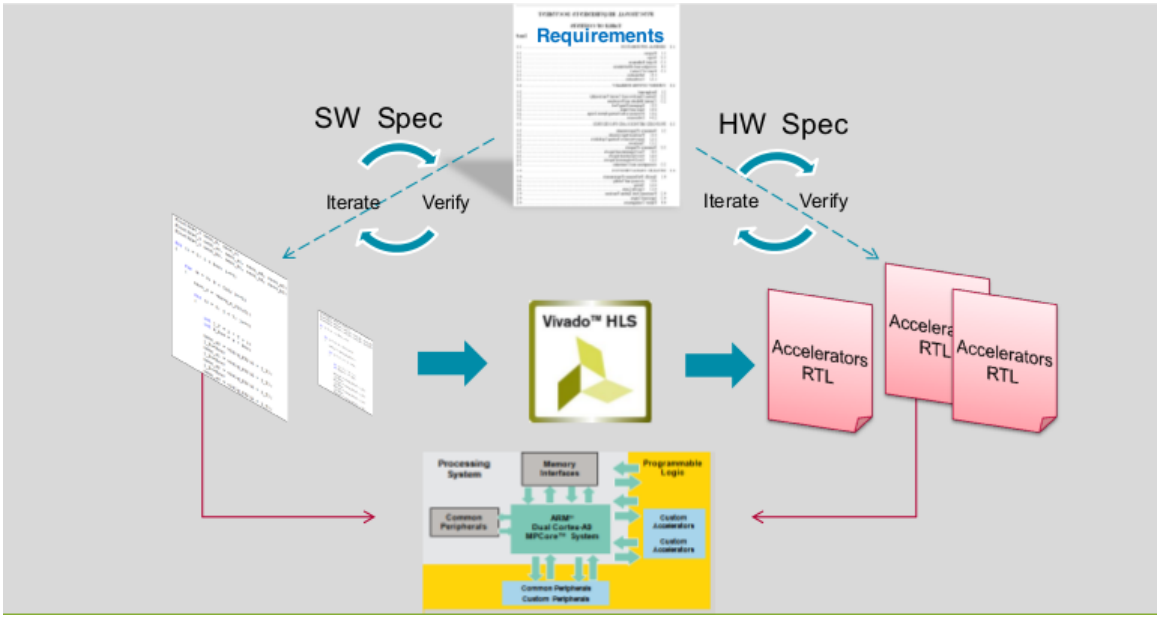


Figure 3.2: Algorithmic C to Co-Processing Accelerator Integration [4]

during the scheduling or the binding phase. During the scheduling process it is determined which operations will occur in which operation cycles. These operations can take place within one or several clock cycles, they can be chained or they can execute in parallel. The scheduling phase takes into account design, timing and user defined constraints. Binding is the process used that determines which hardware resource implements each scheduled operation. The decisions taken in the binding and allocation process influence the scheduling of the operations, thus resulting in these steps to be intertwined rather than happening in a serial fashion [25].

The main advantage of the HLS tools is that they can compile the C code into an implementation of high performance while maintaining an efficient resource usage. This is accomplished by adding HLS-defined pragma (directives) that are taken into account during the scheduling and binding process and result in an optimized IP block. High-Level Synthesis creates the most optimum implementation based on its own default behavior, the constraints, and the directives that the users specify. These optimization directives are selected so that the architecture created satisfies the desired performance and area goals.

When synthesis completes, a synthesis report is generated by High Level Synthesis. This report contains details on the performance metrics. After analyzing the report, optimization directives can be used to refine the implementation towards the desired outcome. In order to do that effectively, it is important to understand the metrics used to measure performance in a design created by HLS [5]. The main ones are area, latency and initiation interval.

- **Area**: Area is a measure of how many hardware resources are required to implement the design. Area is measured by the resources available in the FPGA: LUTs, Registers, block-RAM and DSPs. Their utilization is included in the synthesis report.
- **Latency**: The latency of a function is the number of clock cycles required for the function to compute all output values.
- **Initiation Interval**: The function Initiation Interval (II) is the number of clock cycles before the function can accept new input data, thus before the function can initiate a new set of input reads and start to process the next set of input data.

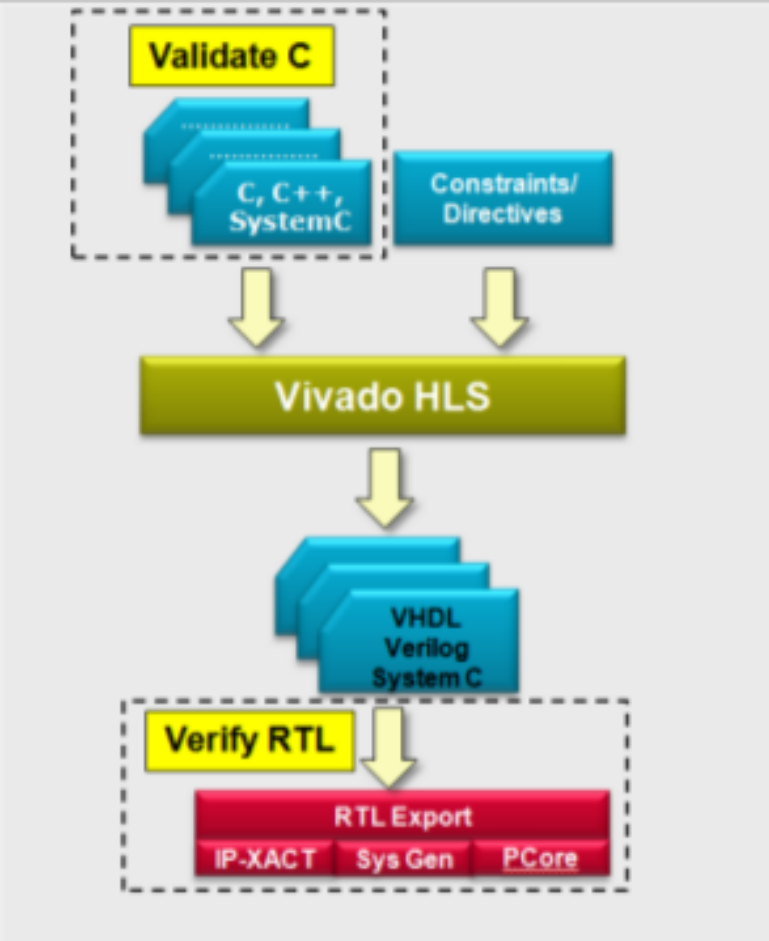


Figure 3.3: Vivado HLS Tiered Verification Flow [4]

The directives available from HLS aim at performance and area optimization. They can be applied on functions, loops, arrays and regions containing one or more of the above [4].

- **Functions**: Directives applied to functions mainly aim at enabling two or more functions to execute concurrently. They can also remove function hierarchy in order to reduce function call overhead and examine logic optimization.
- **Loops**: Directives applied to loops can reduce the cost of transition cycles between different loops and between iterations of the same loop, improve latency, reduce resources and allow the parallel execution of multiple loops within a function.
- **Arrays**: Arrays are the basic construct to express memory in HLS and are implemented using block-RAMS. A block-RAM can be at most dual-port, which means that maximum 2 elements of the same array can be accessed at the same time. This introduces a bottleneck and prevents effective parallelization. The directives that are applied to arrays mainly address this issue. They change array layout by reshaping or partitioning to remove bottlenecks without requiring changes to the original code. There are also directives that map arrays together in order to reduce area.

In Table 3.1 some of the basic HLS directives are briefly descripted.

Table 3.1: HLS directives [5]

Directive	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
UNROLL	Unroll for-loops to create multiple independent operations rather than a single collection of operations.
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block-RAM bottlenecks.
ARRAY_MAP	Combines multiple smaller arrays into a single large array to help reduce block-RAM resources.
ARRAY_RESHAPE	Reshape an array from one with many elements to one with greater word-width. Useful for improving block-RAM accesses without using more block-RAM.

These attributes of HLS make it an ideal choice for creating an efficient co-processor of the classification stage of the ECG analysis flow. Using the code in Listing 3.1 as baseline, we are going to apply some structural changes to it to bring up the inherent parallelism of the algorithm and then further improve our design by exploiting the capabilities provided by HLS. The flow of work that we are going to follow is schematically described in Fig.3.4.

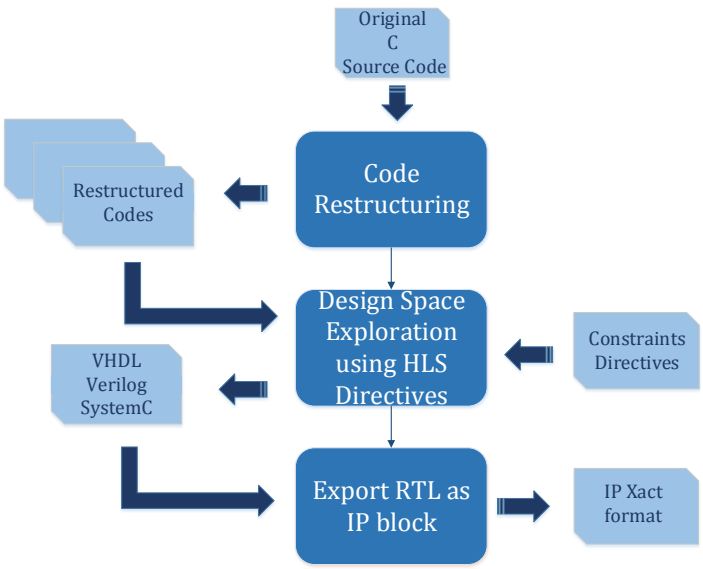


Figure 3.4: HLS Proposed Work Flow

3.3 Zynq Evaluation and Development Board Specifications

The ZedBoard [26] is a low-cost evaluation and development board based on the Xilinx Zynq -7000 All Programmable SoC (AP SoC). Combining a dual Corex-A9 Processing System (PS) with 85,000 Series-7 Programmable Logic (PL) cells, the Zynq-7000 AP SoC can be targeted for broad use in many applications. This board contains everything necessary to create a Linux, Android, Windows or other OS/RTOS-based design. Additionally, several expansion connectors expose the processing system and programmable logic I/Os for easy user access. The features [27] provided by the ZedBoard consist of:

- **Memory**: Zynq contains a hardened PS memory interface unit. The memory interface unit includes a dynamic memory controller (DDR3) and static memory interface modules (SPI Flash, SD Card Interface).
- **USB**: ZedBoard implements one of the two available PS USB OTG interfaces and a USB-to-UART bridge connected to a PS UART peripheral, and provides JTAG functionality and USB circuit protection.
- **Display and Audio**: An Analog Devices ADV7511 HDMI Transmitter provides a digital video interface to the ZedBoard. The ZedBoard also allows 12-bit color video output through a through-hole VGA connector. An Analog Devices ADAU1761 Audio Codec provides integrated digital audio processing. An Inteltronic/Wisechip UG-2832HSWEG04 OLED Display is used on the ZedBoard.
- **Clock Sources**: The Zynq-7000 AP SoC's PS subsystem uses a dedicated 33.3333 MHz clock source, IC18, Fox 767-33.333333-12, with series termination. The PS infrastructure can generate up to four PLL- based clocks for the PL system. An on-board 100 MHz oscillator, IC17, Fox 767-100-136, supplies the PL subsystem clock input.
- **Reset Sources**: The Zynq PS supports external power-on reset signals. The power-on reset is the master reset of the entire chip. A Program Push Button Switch initiates reconfiguring the PL-subsection by the processor. Power-on reset erases all debug configurations.
- **User I/O**: The ZedBoard provides 7 user GPIO push buttons to the Zynq-7000 AP SoC; five on the PL-side and two on the PS-side. It has eight user dip switches providing user input and eight user LEDs.
- **10/100/1000 Ethernet PHY**: The ZedBoard implements a 10/100/1000 Ethernet port for network connection using a Marvell 88E1518 PHY.
- **PS and PL I/O expansion**: A single low-pin count (LPC) FMC slot is provided on the ZedBoard to support a large ecosystem of plug-in modules. The ZedBoard has five Digilent PmodTM compatible headers (2x6). The XADC header provides analog connectivity for analog reference designs, including AMS daughter cards.
- **Configuration Modes**: Zynq-7000 AP SoC devices use a multi-stage boot process that supports both non-secure and secure boot. The PS is the master of the boot and configuration process. Upon reset, the device mode pins are read to determine the primary boot device to be used: NOR,NAND, Quad-SPI, SD Card or JTAG.

The Zedboard can be used for several target applications such as video processing, motor control, software acceleration, linux/Android/RTOS development, embedded ARM processing, general Zynq-7000 All Programmable SoC prototyping. The goal of this work is software acceleration by building an IP on the PL side. The acceleator will have to communicate with

the PS, thus the interest lies in the features of the PS, the PL and their interconnect. Zedboard has a complete ARM-based Processing System with Dual ARM CortexTM-A9 MPCoreTM fully autonomous to a state-of-the-art Programmable Logic. The Programmable Logic is used to extend the Processing System and is tightly integrated. High performance ARM AXI interfaces (High bandwidth AMBA interconnect) are provided for scalable and effective communication. The available resources of the PL side are also very important and critical for applications that need to process large data sets [4]. Table 3.2 contains this information for the Zedboard.

Table 3.2: Zedboard Available Resources

Name	BRAM_18K	DSP48E	FF	LUT
Available	280	220	106400	53200

Chapter 4
Code Restructuring for HLS

4.1 Advancing Coarse Level Parallelism in HLS

4.1.1 Parallelization Technique

At first we try to increase the parallelism of the classification algorithm on function level. To do that successfully, an in depth understanding of the algorithm is required. In that direction, we provide the reader with a detailed explanation of the SVM based classifier.

In the code in Listing 3.1, test_vector represents the feature vector of the pulse to be classified and has been created at the previous stage of the ECG analysis flow, feature extraction. It is implemented as an array of D_sv elements, as many as the attributes of interest are. The classification and differentiation of beats into normal and abnormal is based on these chosen features of the pulse. Array sup_vectors contains the support vectors of the hyperplane that divides the space into two classes. It has N_sv columns, one for each support vector, and each column-support vector has D_sv elements-attributes. Array sv_coef holds the values of the coefficients of the support vectors, and thus has N_sv elements, one for each support vector. Constant b is also a parameter of the derived SVM classifier and represents the bias to which the final result is compared, so as to decide the class to which the current beat belongs.

According to the decision function given in Chapter 3, the squared euclidean distance between the test vector and each support vector is computed and then the RBF kernel function is applied on it. This value is then multiplied by a weighting factor equal to the coefficient of the current support vector. The resulting value is finally added to the total sum, which is compared to the bias in order for the class to be deducted. The contribution of each support vector to the total sum is irrelevant to the contribution of the other support vectors. This means that there are no data dependencies regarding the computations performed between the test vector and each column of the support vector's array. As a result, these computations can be performed simultaneously. This is illustrated in Fig.4.1 where the use of different colours indicates that the computations performed between each coloured column and the test vector can happen in parallel with the computations of the other columns.

This is the main idea of this parallelization technique. Array sup_vectors can be partitioned into smaller arrays, each one containing fewer support vectors. These arrays will have the same number of rows, since the number of attributes does not change, but fewer columns. Each array will contribute a partial sum to the total sum used for classification. The calculations required for the partial sums to be computed can be performed in parallel. We have thus managed to divide the initial large problem into smaller ones, which are then solved at the same time. Each smaller problem is not a subtask of the initial task but the same task performed on a smaller dataset. All the smaller-scale problems are processed independently and simultaneously and their results are combined for the final computations. Thus, we have accomplished to extract coarse level parallelism from the initial problem. The coarse level parallelism is presented schematically in Fig.4.2.

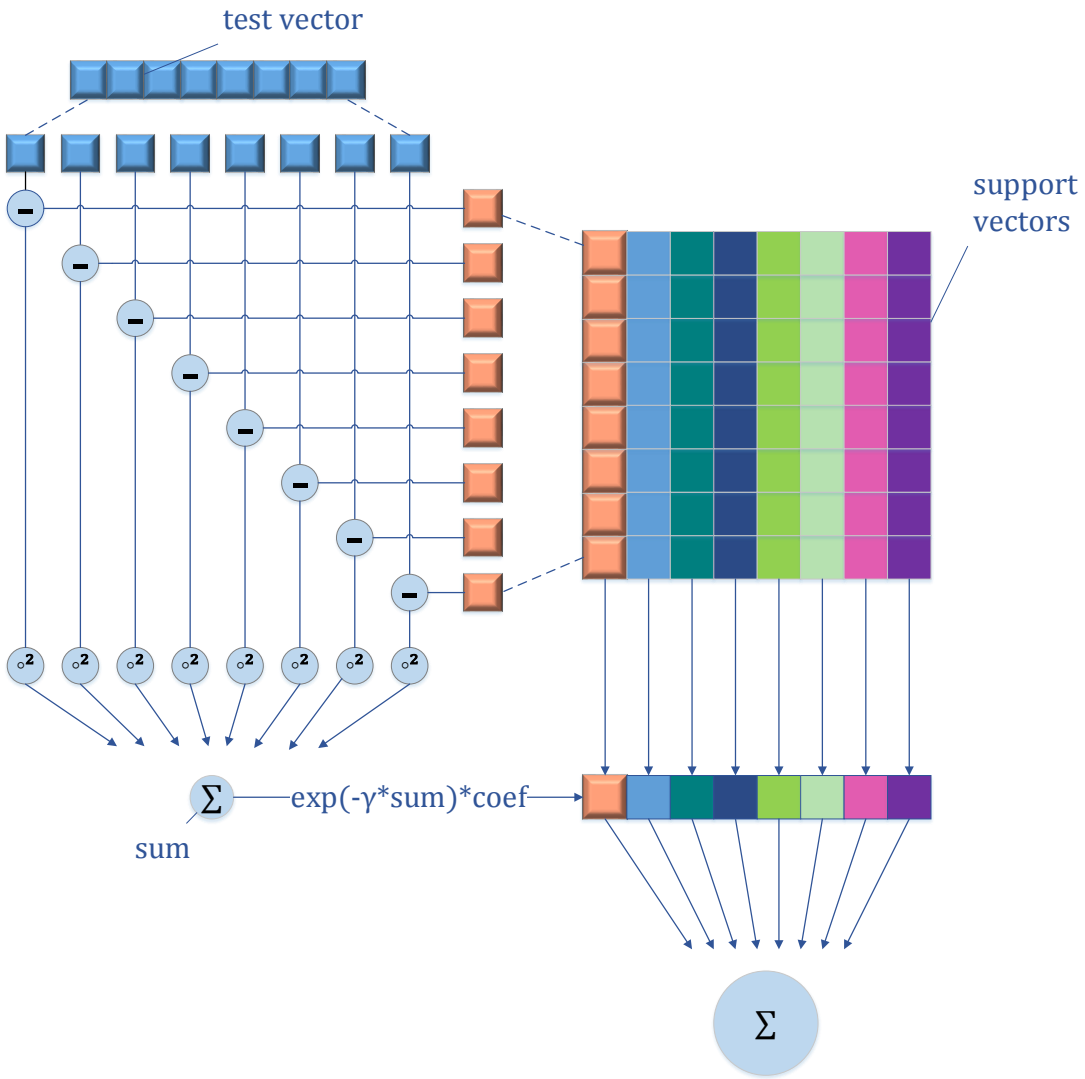


Figure 4.1: Parallelism in Computations

The above idea is going to be implemented by both modifying the structure of the code in Listing3.1 and applying the necessary HLS directives. Firstly, we are presenting the changes made to the code and then the HLS directives that are required to implement the parallelism. The main part of the code that is responsible for computing the total sum is going to be implemented as a separate function called by the main function as many times as many array partitions exist. Each instance of this function is assigned a different partition of the sup_vectors ans sv_coef arrays and computes the partial sum that its assigned partition contributes to the total sum. The main function gathers the partial sums of all partitions and makes the classification decision. The modified code is presented in Listing 4.1 for the case that the support vector array is partitioned into two smaller ones.

On this modified code, we are going to apply the HLS directives. The first directive needed is the one responsible for the automatic array partitioning. An array in HLS is implemented using block RAMs which can at most have two read ports. This means that the simultaneous computation among all the array partitions wouldn't be possible since all functions would require access to the same array at the same time. HLS would address this problem by creating a replicate for each function instance, thus leading to memory burst.

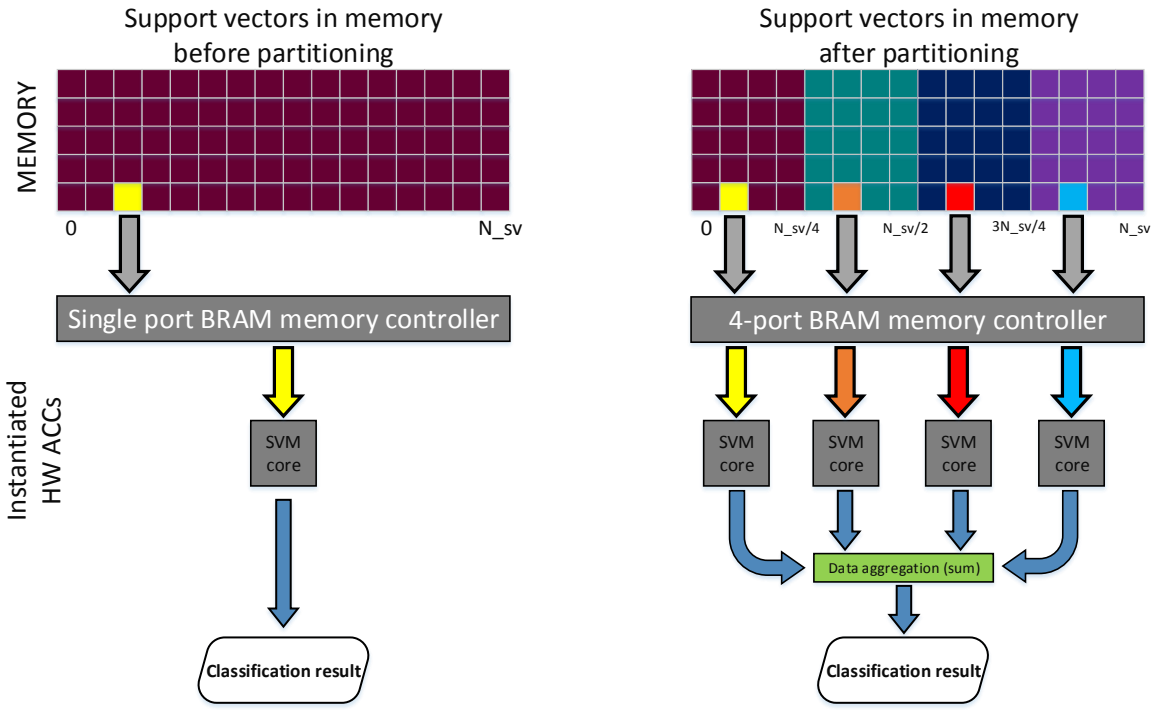


Figure 4.2: Coarse Level Parallelism

By using the array partition directive instead, we solve since different read ports for each partition are created and the parallelization is possible the problem without any adverse effects. In order to use this directive we have to specify on which array it is applied, which dimension of the array is going to be partitioned, how many partitions are needed and which elements each partition contain. We are going to partition arrays sup_vectors and sv_coef. Array sv_coef is one-dimensional, thus it is partitioned across its elements and each partition contains consecutive elements (block partitioning). We are going to partition array sup_vectors across its column dimension into a varying number of partitions and each partition will contain consecutive columns (block partitioning). In the main function we call the computational function once for each array partition. Its arguments include the pointers to the arrays, a value indicating the size of each partition in columns, an offset to identify the exact location of the partition in the initial array and a pointer to the address that holds the partial sum to be computed. Then the dataflow directive is applied on the main function and allows functions within the main's function scope to operate in parallel. HLS automatically detects that the function calls can be executed simultaneously and completes the allocation of resources and synthesis accordingly.

Listing 4.1: Partitioned version of the original code

```
#include <math.h>
#include "svm.h"
#include <stdio.h>
#define gamma 8

void foo(int width,int offset,float *sum,float test_vector[D_sv],
float sv_coef[N_sv],float sup_vectors[D_sv][N_sv]){
 int i,j;
 float diff;
 float norma=0;
 *sum=0;

 loop_i:for (i=0; i<width; i++){
        loop_j:for (j=0; j<D_sv; j++){
                        diff=test_vector[j]-sup_vectors[j][i+offset];
                        norma = norma + diff*diff;
                        }
                        *sum = *sum + exp(-gamma*norma)*sv_coef[i+offset];
                        norma=0;
                }
}


void classify(int * y){

        const float sv_coef[N_sv]={
                                #include "sv_coef.dat"
                        };

        const float test_vector[D_sv]={
                                #include "test_vector.dat"
                        };

        const float support_vectors[D_sv][N_sv]={
                                #include "support_vectors.dat"
                        };

        float diff;
        float sum1,sum2,sum;

        foo(N_sv/2,0,&sum1,test_vector,sv_coef,support_vectors);

        foo(N_sv/2,N_sv/2,&sum2,test_vector,sv_coef,support_vectors);

        sum = sum1 + sum2 - b;

        if (sum<0) *y = -1;
                else    *y = 1;

}
```

4.1.2 Results

We have implemented this idea for array partitioning by a factor of 2,3,4,8 and 16. The results for latency and resources are depicted in Fig.4.3.

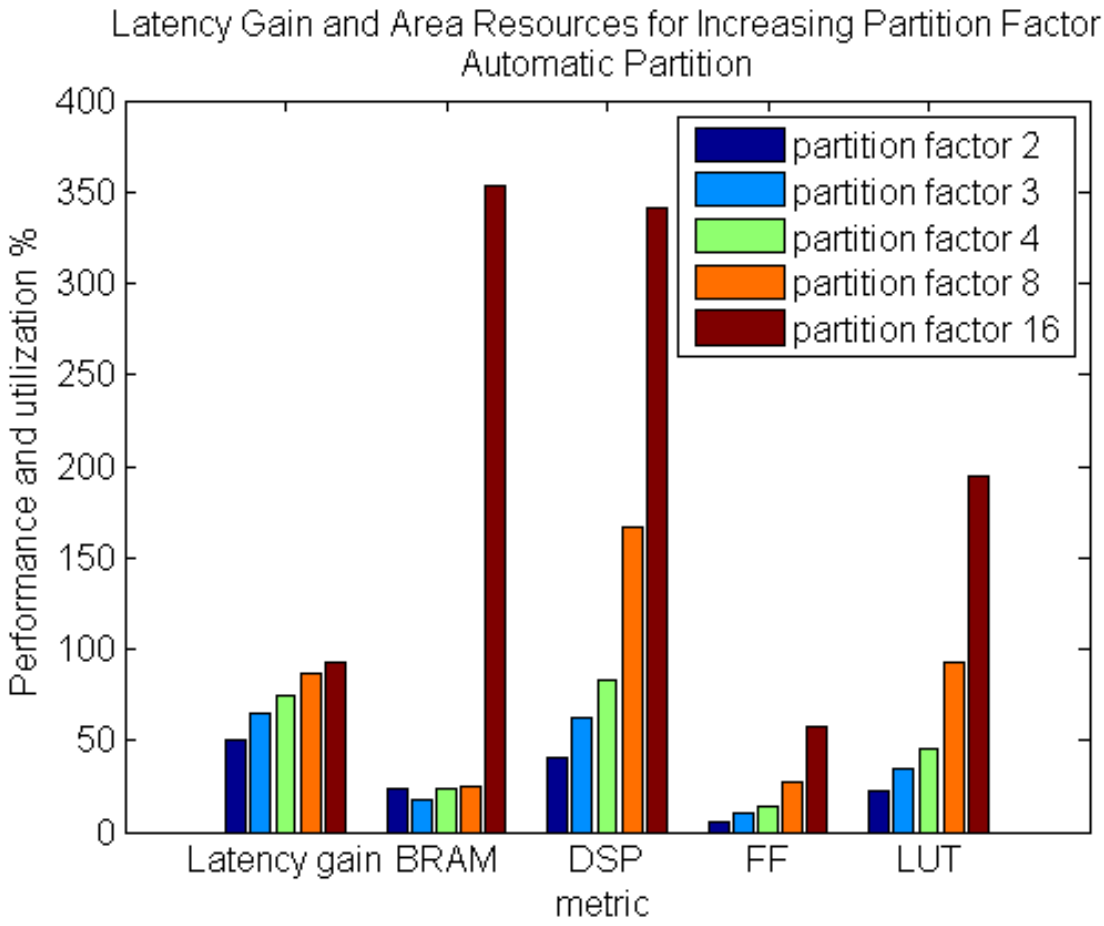


Figure 4.3: Performance and utilization for increasing number of partitions (automatic)

The latency is assessed in terms of gain, in comparison to the latency of the original code. The memory and area are measured in utilization percentages over the initial available resources. As far as the latency is concerned, it can be seen that as the number of partitions becomes greater, the latency of the design reduces accordingly. In fact, the speedup of execution time is very close to the ideal speed-up value, which is equal to the number of array partitions being used. For a partition factor of two, this translates into a speed-up of two and gain in latency equal to 50%, while a partition factor of sixteen results in a speed-up close to sixteen and thus a gain in latency close to 93.75%. There is however a trade-off between the gain in latency and the increasing area resources. In order for the function instances to run in parallel the logic used for the function body is replicated once for each instance. This explains the increase in DSP, flip flop and LUT utilization. For example, the loop body requires 45 DSP units for all the operations to be scheduled the way they are in the original code. This number remains the same for each independent instance but the total number multiplies by a factor equal to the number of instances or partitions used. Given the fact that there are 220 DSP units available in the specific development board, the utilization

percentages are formed as in Fig.4.3, exceeding availability for a partition factor greater than 4. The same principle also applies for the utilization of flip flops and LUTs. An interesting parameter is the utilization of BRAM for array implementation. By using the partitioning directive we have managed to avoid the replication of the arrays for each instance. Each function requires as many BRAMs, as many are needed to implement its partition of the array. This leads to the total number of BRAMs used remaining almost the same in all cases except for the one of partition factor of value 16. In that case replicates of the arrays are created, resulting in a burst in memory usage.

In our efforts to prevent this burst from happening, we tried to apply the same idea only this time doing it manually. The only thing that differs from the previous implementation is the way the support vector and coefficient arrays are partitioned. Instead of declaring one large array and then partitioning it into smaller ones using the array partitioning directive, we allocate several smaller arrays from the start. Now in each function call, a different array pointer is passed pointing to one of the array partitions. This means that we must manually split the array containing the support vectors and the array of their coefficients into smaller arrays containing fewer support vectors. The results are shown in Fig.4.4.

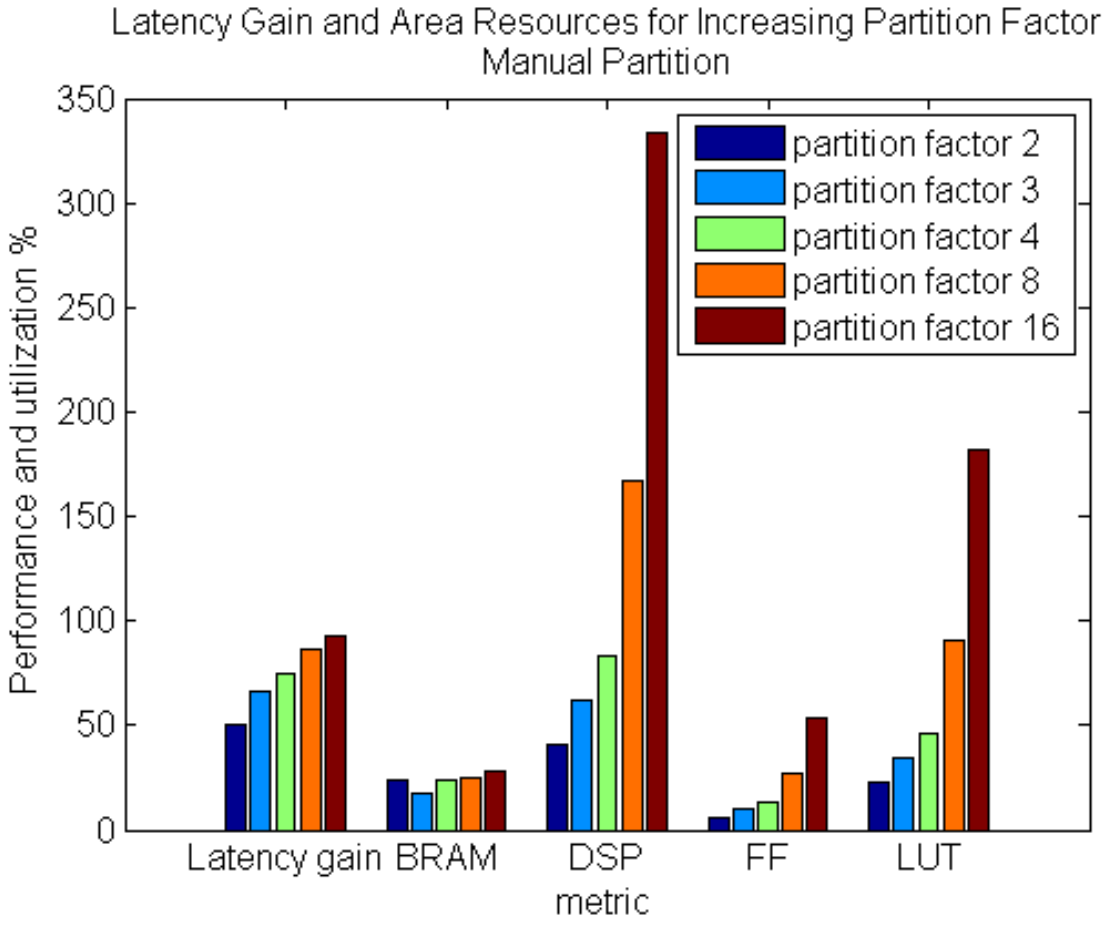


Figure 4.4: Performance and utilization for increasing number of partitions (manual)

The gradual increase in area resources (DSPs,flip flops and LUTs) remains the same. The burst in number of BRAMs when partitioning for a factor of 16 is now remedied and BRAM utilization remains practically the same regardless of the number of partitions used. We also noted that the gain in latency is even greater now, giving a speed-up practically equal to the ideal one. This is depticted in Fig.4.5.

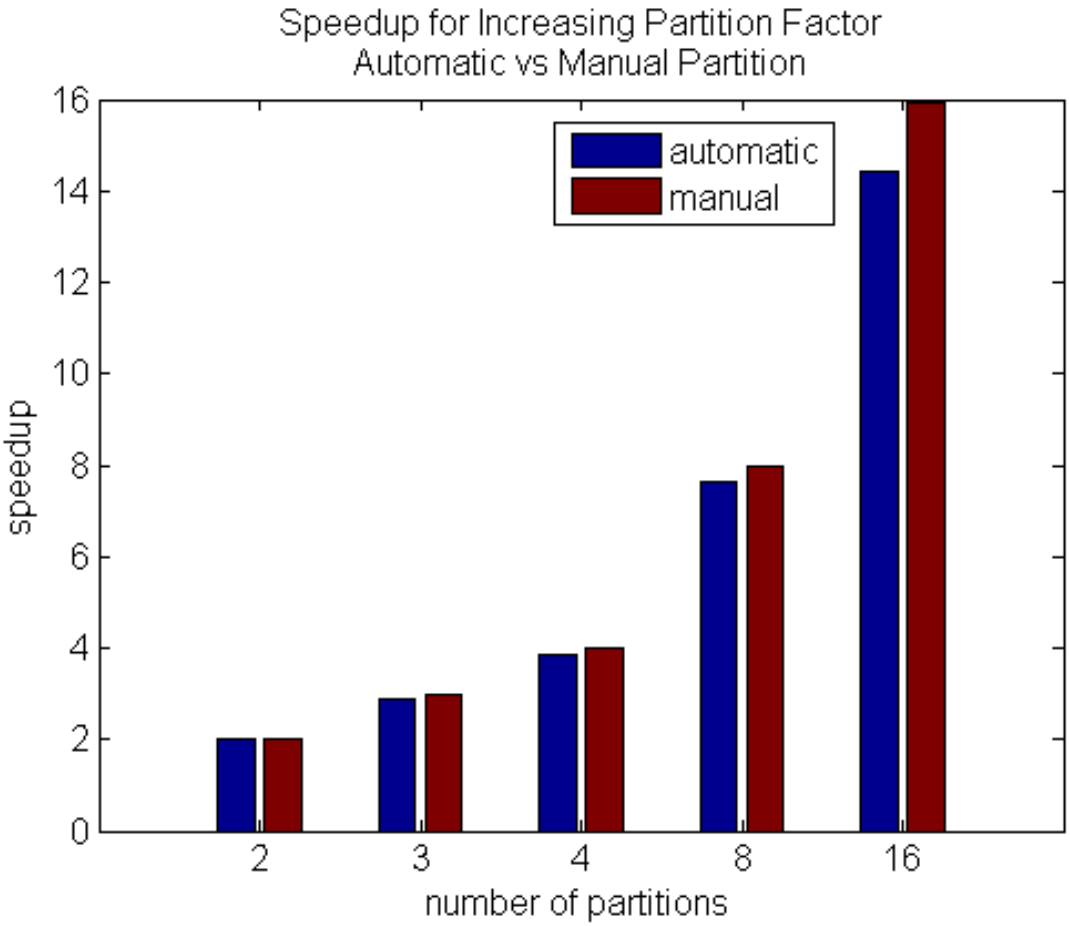


Figure 4.5: Speedup gain comparison (automatic vs manual)

4.2 Advancing Instruction Level Parallelism through arithmetic operation reshaping

4.2.1 Parallelization Technique

So far we have examined how to parallelize the SVM classifier on coarse level by performing the computations related to different support vectors simultaneously. Now we are going to examine increasing parallelism within the computations required for calculating the contribution of one support vector only. This is achieved by increasing instruction level parallelism.

We are going to parallelize the inner loop (loop_j) of the code, which is responsible for the computation of the euclidean distance. Each time this loop is executed it computes the euclidean distance between a different support vector and the test vector to be classified. It iterates over the attributes-elements of the two vectors and in each iteration the difference between the two attributes-elemeInnts is computed. Then this difference raised to the square is added to variable "norma", which holds the value of the euclidean distance at the end of loop execution. Instead of computing one squared difference in each iteration we could compute several of them at the same time. This can be achieved by unrolling the loop, storing each squared difference in a temporary variable and ultimately adding all of them to the variable "norma" holding the final distance.

Adding many floating point values in HLS increases dramatically the critical path because the additions are scheduled in a serial manner even though there is no true data dependency between the added values. However a more efficient implementation of the addition is feasible by changing the structure of the addition and transforming it into a tree-based computation of the final result. That way HLS can schedule the independent calculations to be executed in parallel allocating if needed more hardware resources.

Several changes have to be made to the code in Listing 3.1. Firstly, the inner loop is manually unrolled and as a result the number of iterations lessens. In each iteration more differences are computed (as many as the unroll factor is) and saved to temporary variables. Each one of the computed differences is raised to the power of two. Then we begin to add the squared differences following a tree-based structure, adding them in pairs to produce new temporary values. Accordingly, these values have to be added again in pairs and this continues in the same fashion until all the squared differences have been added to the variable containing the euclidean distance upon loop completion. By arranging the partial additions in this structure, we implicitly force HLS to allocate more resources so that these operations can execute in parallel.

In this work the best classifier for the arrhythmia detection was determined after performing design space exploration. This exploration resulted in a feature vector of 18 attributes. This is also the number of iterations of the inner loop. We have applied the above idea by manually unrolling the inner loop by a factor of 3,6 and 18 and arranging the computations in a tree structure. The modifications to the code are presented in Listing 4.2 for an unroll factor of 6. In Fig.4.6 the tree based structure of the calculations for an unroll factor of 6 is demonstrated. In the same figure the way HLS schedules these operations is shown. The difference between the default scheduling of HLS for automatic unrolling and our proposed implementation can be seen by comparing Fig.4.7, both of which assume an unroll factor of 6.

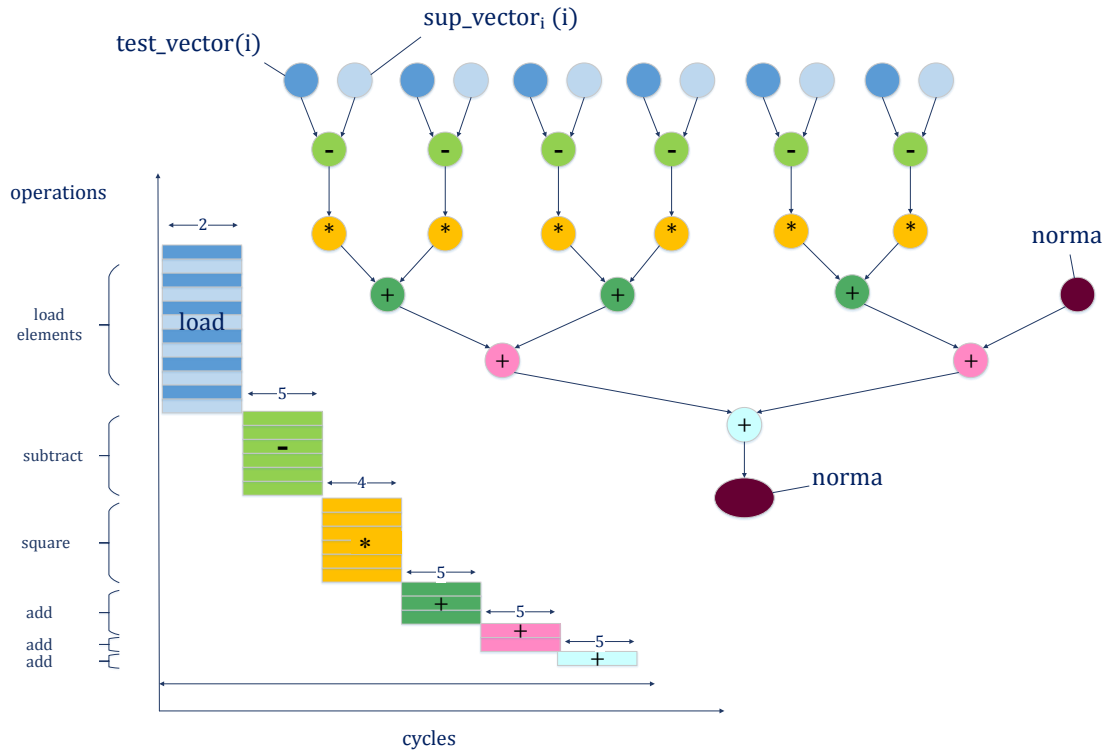


Figure 4.6: Tree based computations for manual unrolling and HLS scheduling

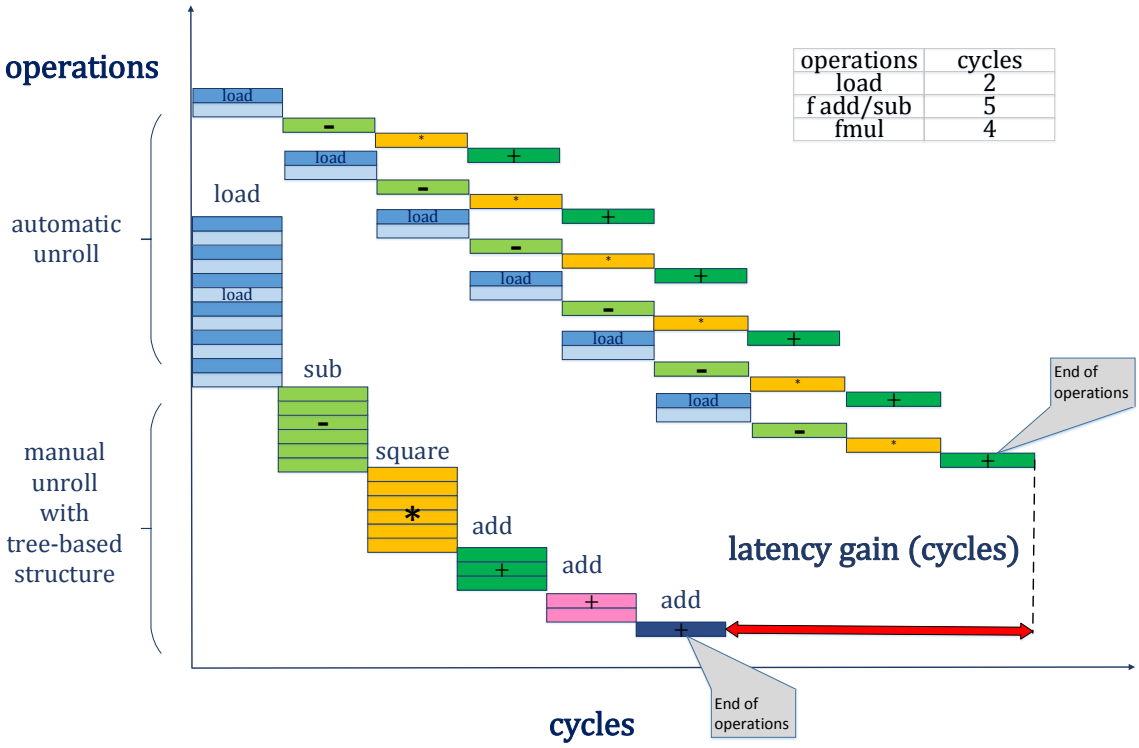


Figure 4.7: Scheduling Comparison between manual and automatic unrolling

Listing 4.2: Unrolled version of the original code

```
#define gamma 8
const float sv_coef[N_sv];
const float sup_vectors[D_sv][N_sv];

void SVM_predict (int *y, float test_vector[D_sv]){

loop_i:for (i=0; i<N_sv; i++) {
        loop_j:for (j=0; j<D_sv; j=j+6) {
                d1=test_vector[j]-sup_vectors[j][i];
                d2=test_vector[j+1]-sup_vectors[j+1][i];
                d3=test_vector[j+2]-sup_vectors[j+2][i];
                d4=test_vector[j+3]-sup_vectors[j+3][i];
                d5=test_vector[j+4]-sup_vectors[j+4][i];
                d6=test_vector[j+5]-sup_vectors[j+5][i];

                sq_prod1=d1*d1;
                sq_prod2=d2*d2;
                sq_prod3=d3*d3;
                sq_prod4=d4*d4;
                sq_prod5=d5*d5;
                sq_prod6=d6*d6;

                tmp_sum1=sq_prod1+sq_prod2;
                tmp_sum2=sq_prod3+sq_prod4;
                tmp_sum3=sq_prod5+sq_prod6;

                tmp_sum4=tmp_sum1+tmp_sum2;
                norma = norma + tmp_sum3;

                norma = norma + tmp_sum4;
        }

        sum = sum + exp(-gamma*norma)*sv_coef[i];
        norma=0;
}
        sum = sum - b;
        if (sum<0)
                *y = -1;
        else
                *y = 1;
}
```

4.2.2 Results

In Table 4.1 we can see the difference in performance and resources utilization between applying the unroll directives on the inner loop and manually unrolling it while using a tree-based expression balancing structure for the computations.

Table 4.1: Evaluated metrics for automatic vs manual unrolling

Version	Unroll factor	Automatic Unroll using directives					Manual Unroll using tree structure				
		latency (cycles)	BRAM (%)	DSP (%)	FF (%)	LUT (%)	latency (cycles)	BRAM (%)	DSP (%)	FF (%)	LUT (%) (%)
initial	-	412783	24	20	3	11	412783	24	20	3	11
unrolled	3	252259	24	21	3	11	214039	47	26	4	14
unrolled	6	206395	24	23	3	11	149065	70	34	5	18
unrolled	18	173271	27	20	3	11	90461	27	50	8	29

The same results are illustrated in Figures 4.8 to 4.13, each of which highlights the changes in a different metric for automatic versus manual unrolling of the inner loop.

There is a significant improvement in latency gain when applying manual instead of automatic unrolling and the difference becomes greater the larger the unrolling factor is. The reason is the tree based structure of the computations that allows data independent operations to execute in parallel, significantly reducing the inner loop latency and as a result the total design latency as well.

Manual loop unrolling appears to cause however an increase in BRAM utilization for some unroll factors. When unrolling for a factor of 3, three accesses at array support vectors are required at the same time but the array has at most two read ports. To achieve the reading of three elements, HLS creates a copy of the array and implements both copies using dual port RAMs since three elements can indeed be read from four ports. As a result, there is a gain in parallelism and latency but the BRAM utilization doubles. This can be crosschecked from the reports, where the implementation of the sup_vector array requires 128 BRAMs whereas in the original version exactly 64. In the case of unrolling for a factor of 6, three copies of the same array are needed to be implemented with dual port rams in order for parallelism to be achieved. Thus BRAM utilization triples and the array is implemented using 192 BRAMs in particular. For fully unrolling the loop we would expect nine copies of the array to be created in order for 18 elements of the same array to be read concurrently. When fully unrolling the loop however, HLS automatically partitions the array in 18 rows, each one being implemented as dual port, and as a result parallelism is possible without memory increase. The memory increase problem for the cases of using an unroll factor of 3 and 6 can be avoided altogether by applying array partition directives, as discussed in the next chapter 5.

An increase in the area resources utilization is also observed. The implicit declaration of parallel subtractions and multiplications as well as of the parallel additions of the tree structure result in an increase in the number of computational instances required to achieve instruction level parallelism. HLS allocates more multipliers and floating point adders to achieve parallelism while at the same time tries to limit their number by maximizing sharing during the scheduling and binding process. The result is an increase in instances which translates into an increase in DSPs, flip flops, LUTs,multiplexers and registers.

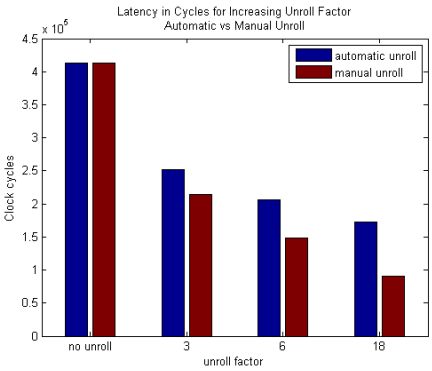


Figure 4.8: Latency in Cycles for Automatic vs Manual Unroll.

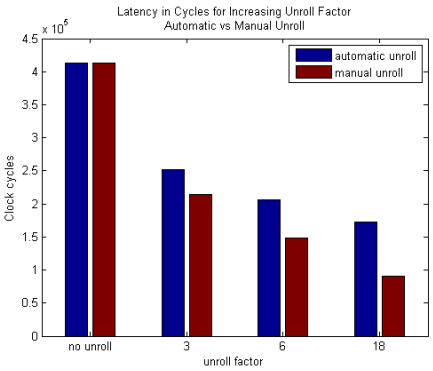


Figure 4.9: Latency Gain for Automatic vs Manual Unroll.

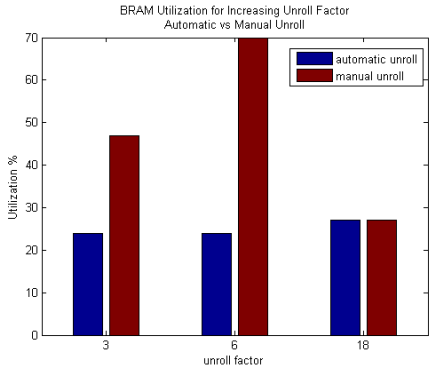


Figure 4.10: BRAM Utilization for Automatic vs Manual Unroll

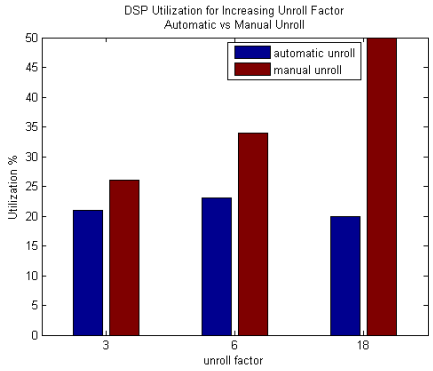


Figure 4.11: DSP Utilization for Automatic vs Manual Unroll

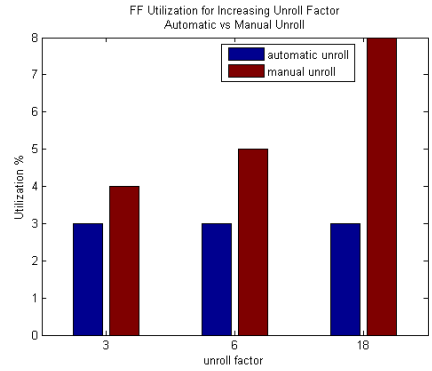


Figure 4.12: Flip Flop Utilization for Automatic vs Manual Unroll

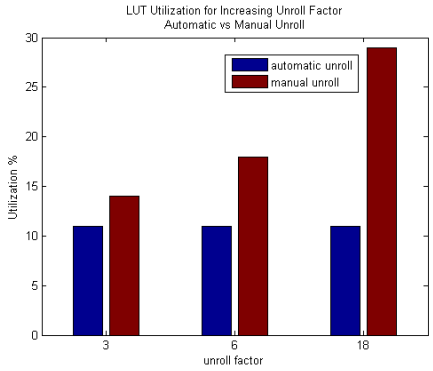


Figure 4.13: LUT Utilization for Automatic vs Manual Unroll

Chapter 5
Exploration of HLS Directives

5.1 Selection of Optimization Directives

In the previous sections we examined ways to create effective RTL based on structural modifications of the C code. These modifications offer a first level of optimization. The performance though can be further improved by combining these structural modifications with the HLS directives. In this section we are going to explore the impact the chosen directives can have on the efficiency of the accelerator.

The directives chosen for optimization differ from one application to another and depend on the nature of the algorithm under study. For the SVM classifier the selection of directives applied is based on the instruction level parallelism that can be accomplished. Most of the directives used, aim at optimizing the inner loop that computes the euclidean distance between the test and support vector. As mentioned before, the squared differences that compose the euclidean distance can be computed in parallel, which means that the inner loop should be unrolled. For an unrolling factor greater than two this leads to more than two simultaneous reads to the same array, which exceeds the maximum number of read ports of each BRAM. As a result we have to change the array structure by partitioning or reshaping so as to have more read ports and so that the accesses to the required elements can happen in parallel. In the same manner we can unroll the outer loop and partition or reshape the arrays accessed in each iteration of the outer loop to gain parallel access to their elements.

All the directives applied on the classifier and the intent of their application are explained below:

Loop pipeline: This directive can be applied on all loops. Without pipeline all loop iterations execute sequentially instead of being scheduled when the required resources are available. Thus loop pipelining allows the use of all resources at the same time.

Loop unroll: This directive can also be applied on all loops. It reduces the number of iterations and increases logic since it creates copies of the loop body.

Array partition: This directive is applied on the sup_vector and the sv_coef arrays. It partitions arrays into smaller ones providing more data ports. For array sup_vector, when combined with the unrolling of the inner loop it allows access to multiple elements of the same column in parallel. The partitioning must be performed in a cyclic way so that the consecutive elements that need to be accessed during the same iteration belong to a different partition. In Fig.5.1 the partitioning of the sup_vector array is illustrated. The array is partitioned across the row dimension, which contains the attributes. In the figure an unroll factor of 4 is assumed, meaning that the initial array is partitioned into 4 smaller arrays of the same number of columns but less rows. The rows with the same colour belong to the same array partition and the tuples of four that are highlighted are accessed at the same time. The same principle applies to the partition of the array sv_coef, with the only difference being that this array is one-dimensional. Its partition looks like the partition of only one column of array sup_vectors.

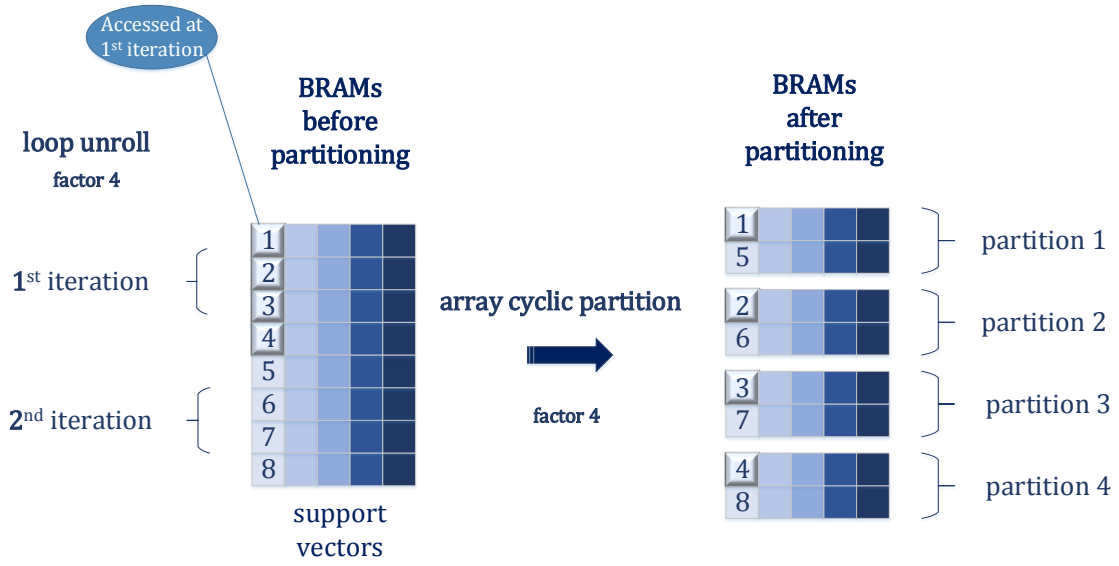


Figure 5.1: Array partition schematically

Array reshape: This optimization performs a similar task to array partitioning however instead of creating more arrays it re-combines the elements created by partitioning into a single block-RAM with wider data ports. Thus it reduces the number of block-RAM while still allowing parallel access to the data. It is applied on the sv_coef and the sup_vectors arrays. The difference from array partition is that the elements of these arrays that get accessed at the same iteration instead of belonging to different partitions, are merged into one element of wider word-width. The reshaping of array sup_vectors is demonstrated in Fig. 5.2. Again we assume an unroll factor of 4. The elements that are highlighted are merged into one element in the new array of less rows that is created.

These directives can be applied to any of the implementations mentioned at the previous sections since all of them maintain the same computational kernel regardless of any expression balancing or how large the dataset they operate on is.

We have implemented four basic versions of the SVM classifier. The initial given in Listing 3.1 and three versions which had the inner loop manually unrolled by a factor of 3,6 and 18 (fully unrolled) respectively as described in Chapter 4. These implementations were synthesized for all valid combinations of the directives above for various values of their parameters. In Table 5.1 all directives used and all the values of their parameters are included. Some combinations of directives were excluded due to directives incompatibility (for example pipeline unrolls all loops down its hierarchy so there is no point in combining these two directives) and others due to user defined constraints. For the initial implementation the main constraint applied was that there is no reason to partition an array without unrolling the loop in which it is accessed since in each iteration there's only one access needed. When the corresponding loop is unrolled by some factor then the array is partitioned or reshaped by that same factor. Also when a computation requires the elements of two arrays and one of them is partitioned or reshaped the other must be too. In the implementations where the inner loop is already unrolled we loosen the first constraint. The arrays in the inner loop can be partitioned or reshaped when the inner loop is not further unrolled but by a factor equal to the factor the loop is already manually unrolled by. When the loop is further unrolled the array is also further partitioned or reshaped by that same factor.

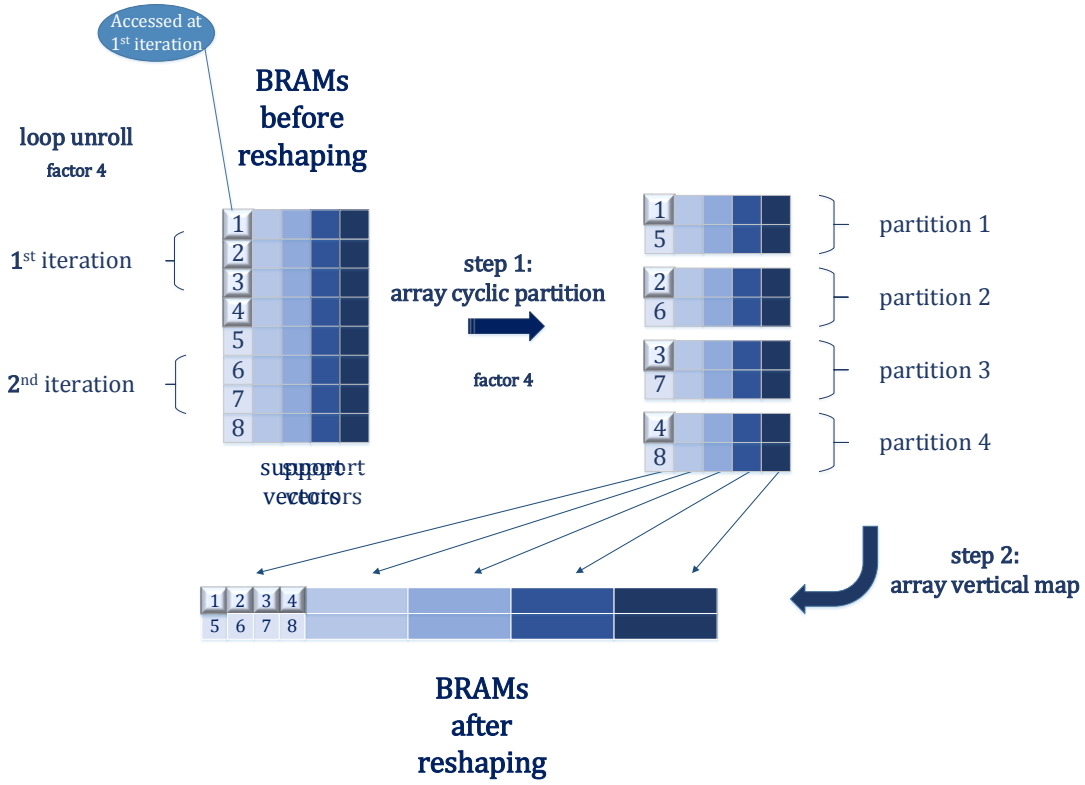


Figure 5.2: Array reshape schematically

5.2 Application on Original Code

The original SVM code was synthesized using the directives in Table 5.1 and their combinations allowed by the constraints described in the previous section. The same configurations were synthesized under different clock periods for a more thorough design space exploration. The clock periods used were those of 10, 20, 30 and 40 ns. The results were gathered and used by the boxplots of the following sections. Only the configurations that achieved critical path less than the clock period were used for the diagrams.

Table 5.1: Applied directives and their parameters

directive	variable	mode	factor	dimension
pipeline	loop_i	-	-	-
	loop_j	-	-	-
loop unroll	loop_i	-	2,7	-
	loop_j	-	2,3,4,6,9,18	-
partition	sup_vectors	cyclic	2,3,4,6,9,18	rows (dim 1)
	test_vector	cyclic	2,3,4,6,9,18	-
	sv_coef	cyclic	2,7	-
reshape	sup_vectors	cyclic	2,3,4,6,9,18	rows (dim 1)
	test_vector	cyclic	2,3,4,6,9,18	-
	sv_coef	cyclic	2,7	-

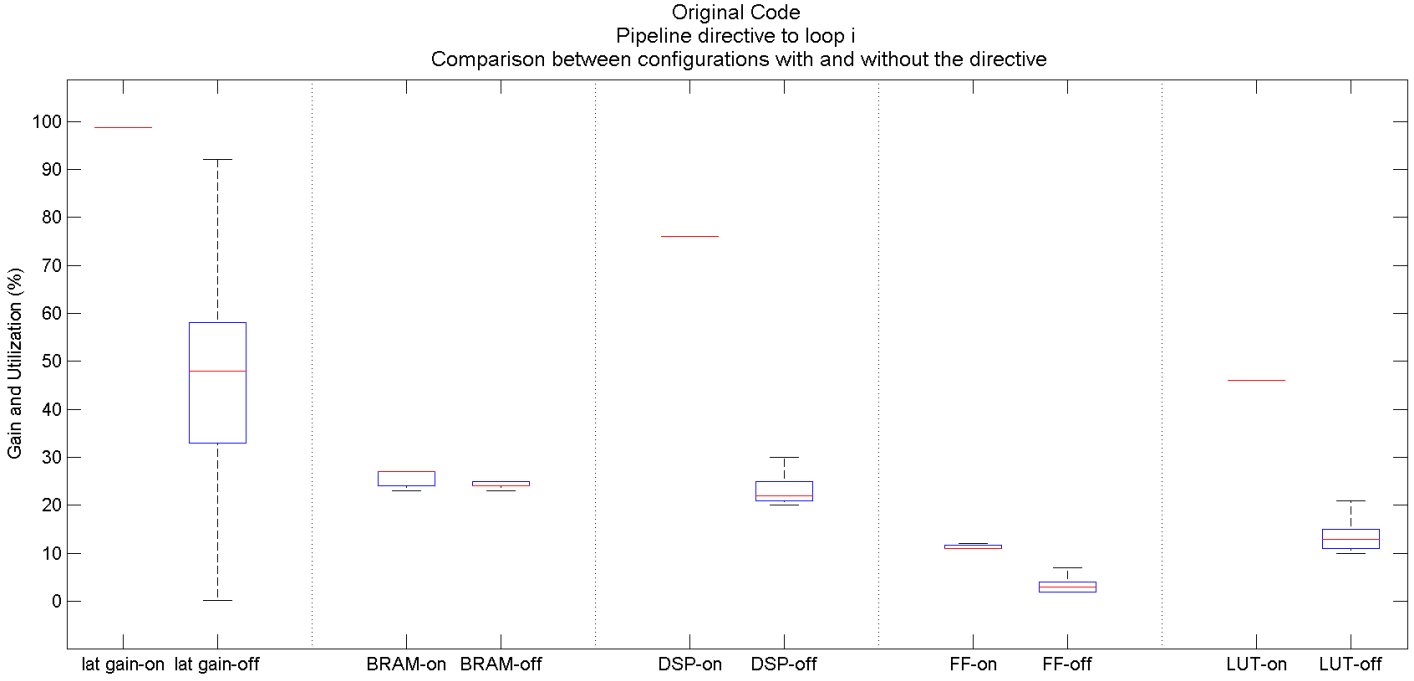


Figure 5.3: Pipelining loop_i. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

5.2.1 Impact of each directive

In this section, the impact of each directive on the various metrics is examined. Each of Figures 5.3 to 5.20 depict how all configurations score in metrics with one specific directive enabled or disabled.

In Fig.5.3 the pipeline directive in the outer loop loop_i is examined. The results show its definitive impact on latency. When it is applied latency gain reaches the value of 99%. When it is not applied latency gain can practically hold any value from 0 to 100% and all the quartiles appear to have the same distance one from another. It does lead however to a large critical path that often exceeds the time constraints. In terms of BRAM utilization, when the directive is applied there is a slightly wider range of utilization values as opposed to not being applied. However this range doesn't exceed the width of 5%. In fact, in both cases the utilization remains very close to that of the original synthesis with no directives applied at all. Upon its application, DSP, Flip Flop and LUT utilization change significantly. The range of taken values is much narrower and the median of greater value. Especially for DSPs and LUTs the utilizations have a fixed value in contrast to the 5% range within which they fluctuate when the directive is not applied. The higher values of the utilization of area resources can be explained by the fact that when pipeline is applied on a loop, it unrolls all loops inside of it. This leads to an unavoidable increase in area. Pipelining the outer loop loop_i unrolls the inner loop loop_j, thus the main loop body logic is replicated leading to an increase in resources.

Fig.5.4 depicts the results of unrolling the outer loop (loop_i). Unrolling the outer loop does not seem to be crucial in the improvement of the design in terms of latency since the range of values extends from 0 to 90%. When this loop is not unrolled latency receives values from a much narrower range but the value does not exceed the gain of 65%. When this directive is

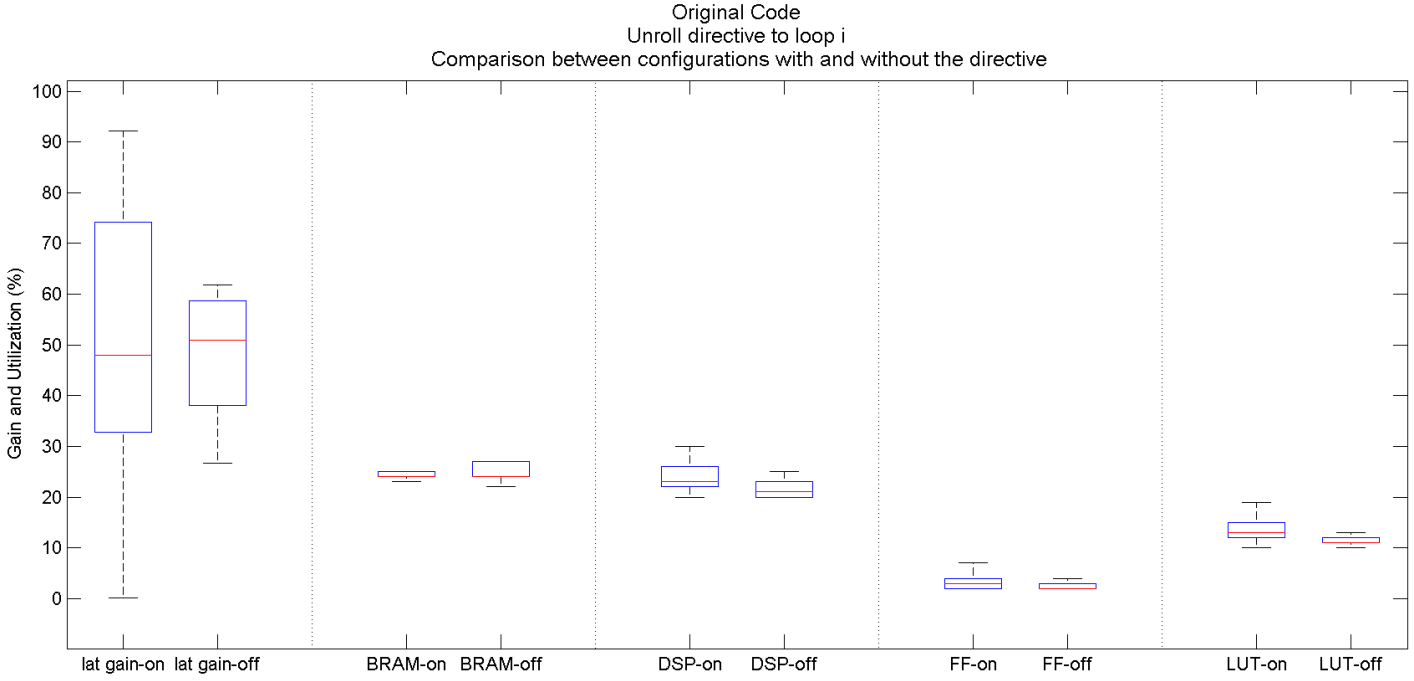


Figure 5.4: Unrolling loop_i. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

applied the BRAM utilization is not greatly affected. The median value is the same in both cases and equal to the original one. As far as DSPs, FFs and LUTs utilization is concerned their value range is slightly wider and the median value is greater when the directive is applied. This is again anticipated, since unrolling a loop in HLS creates duplicates of the loop body and thus the area is increased. In Fig.5.5 the impact of the directive is exhibited for two different values of the unroll factor. The first value is two and the second is seven. It is evident that a greater unroll factor leads to a stronger manifestation of the directive's impact. Thus we can see that BRAM, DSP, flip flop and LUT utilization receive values from a wider range and these values are greater for unroll factor equal to 7 as opposed to unroll factor equal to 2. The latency range and median value remain the same.

When utilizing the pipeline directive in the inner loop (loop_j) latency gain has a narrower range that extends from 50 to 90% and has a median value of 75% as shown in Fig.5.6. When the loop is not pipelined the range is much wider and the median value is lower. When pipelining the inner loop all the area resources utilization are of the same or slightly greater range but are characterized by the same median values, close to the ones of the original version of the SVM code. This happens because pipeline increases parallelism by using all resources at all times and not by replicating hardware. The operations are scheduled when the required resources and data are available and not necessarily sequentially. This increases instruction level parallelism and reduces the initiation interval by allowing the concurrent execution of operations within the loop. Thus the major improvement in latency gain is anticipated.

Unrolling the inner loop has a positive impact on latency as it limits the range of values taken by latency gain. It is illustrated in Fig.5.7 that when the inner loop is not unrolled the range of values of latency gain extends from 0 to 100% with a median of 50% while the majority of configurations have a gain less than 60% since that is approximately the value of the third

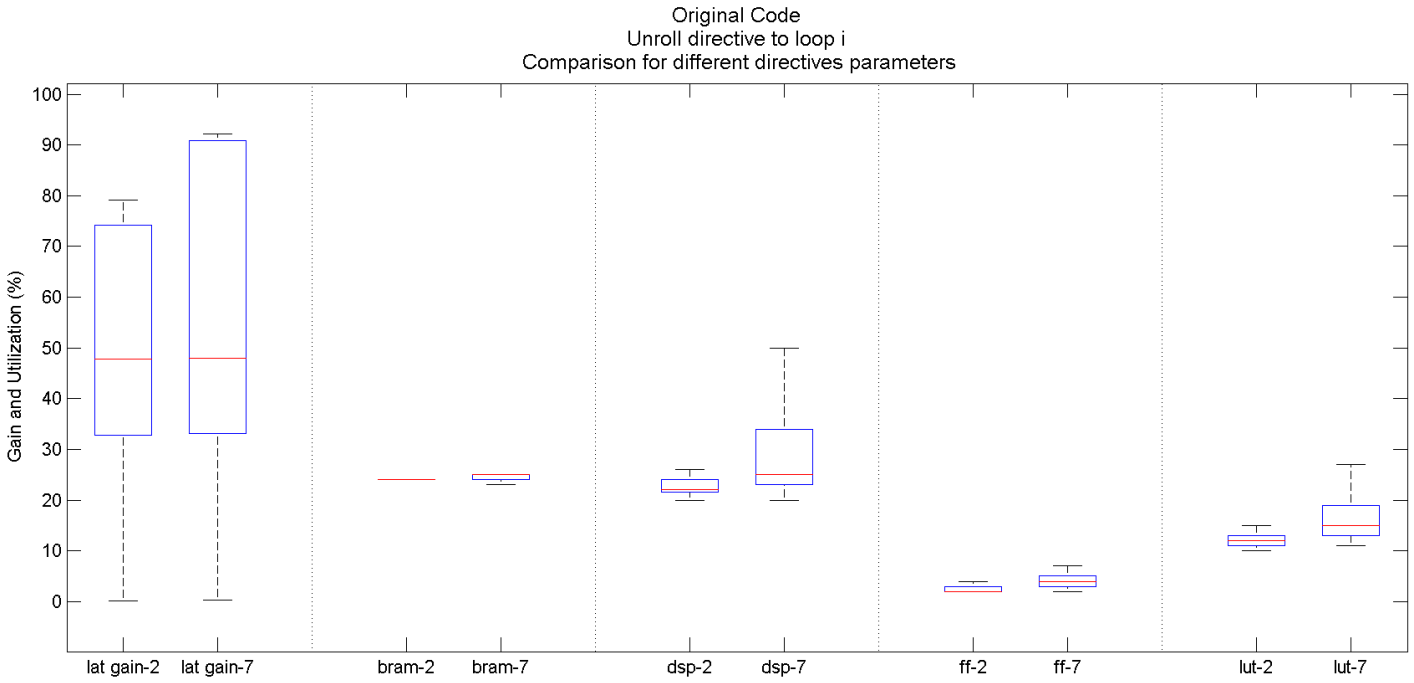


Figure 5.5: Changing unroll factor on loop_i. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization.

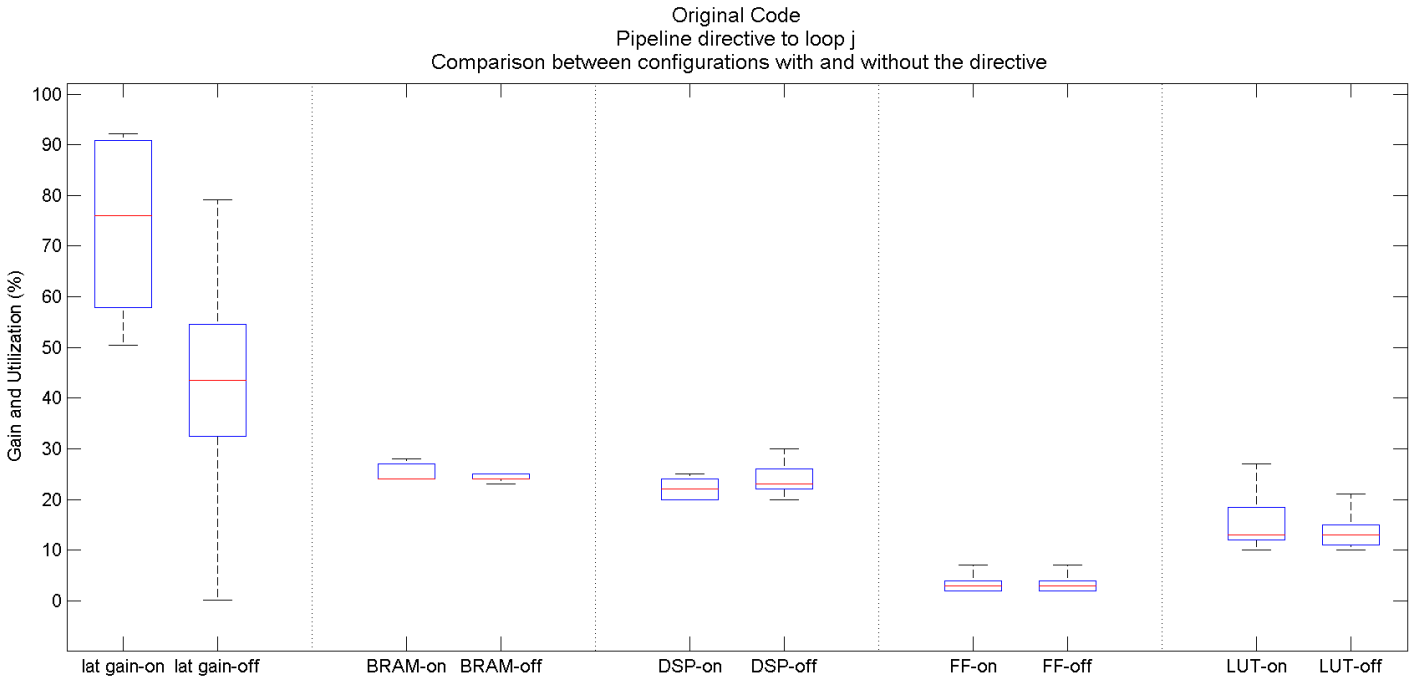


Figure 5.6: Pipelining loop_j. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

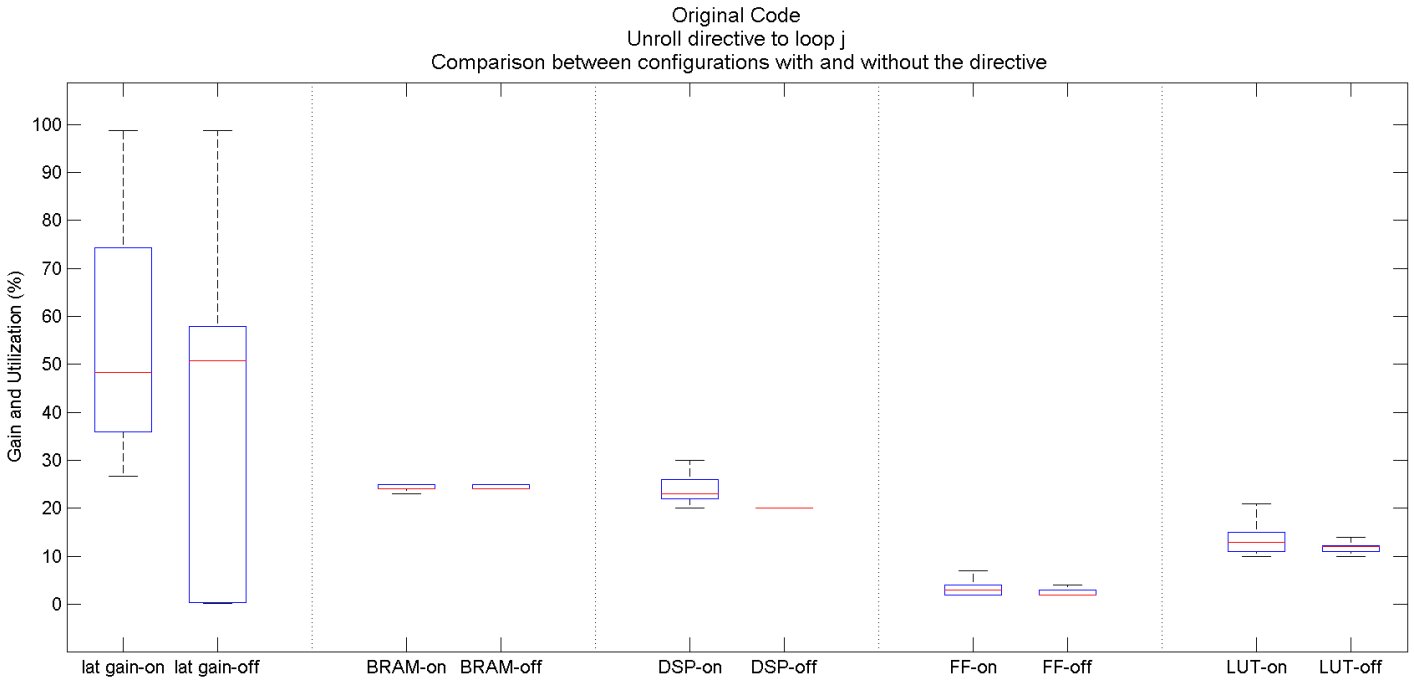


Figure 5.7: Unrolling loop_j. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

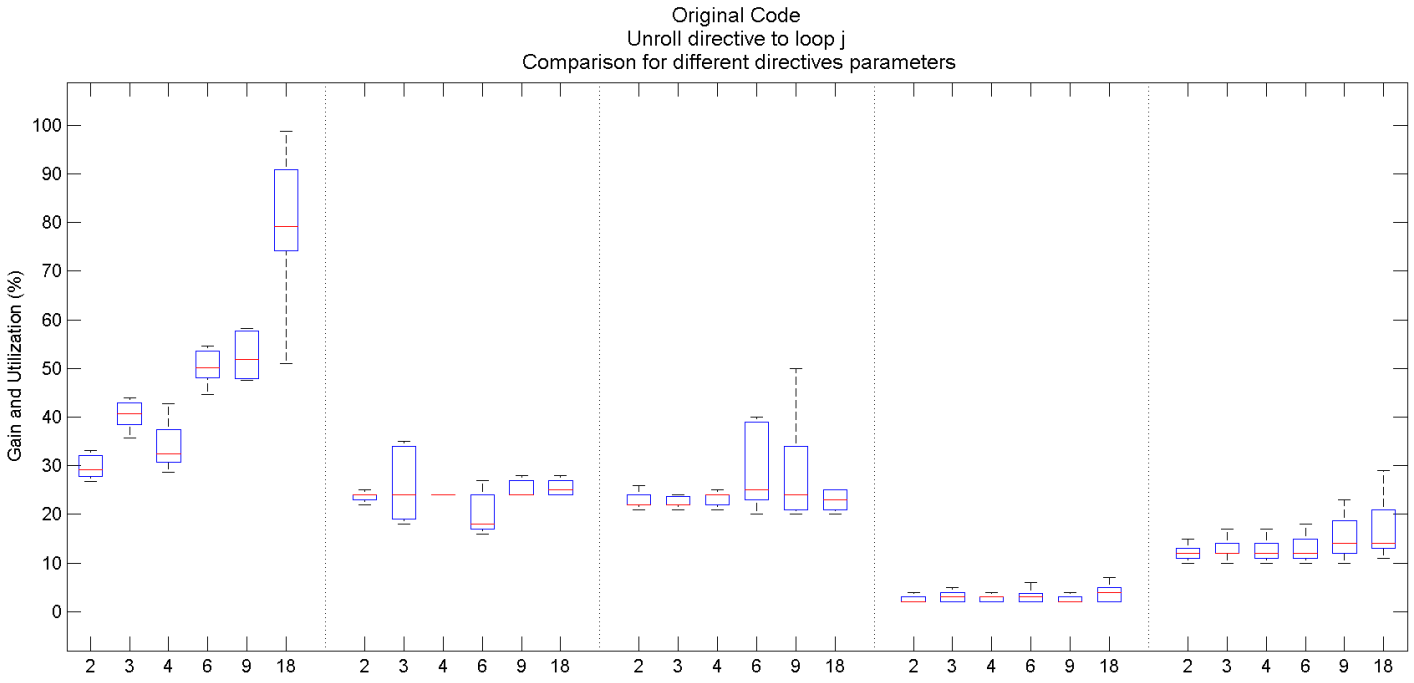


Figure 5.8: Changing unroll factor on loop_j. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization.

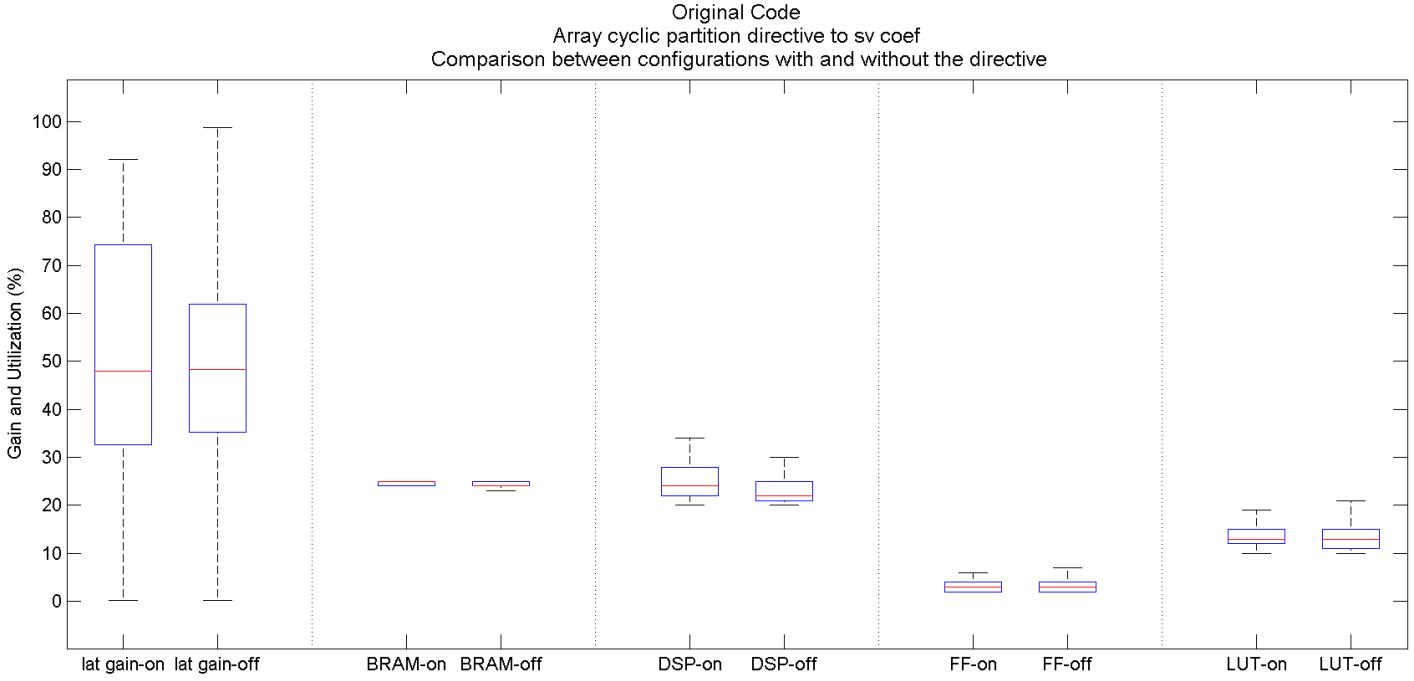


Figure 5.9: Partitioning sv_coef. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

quartile. On the other hand applying the unroll directive to the loop significantly limits the range of values taken. All configurations that include the unroll directive have a gain of at least 30% and some of them even reach a gain near 100%. The median value is close to 50% as well. Fig.5.8 presents the change in latency for an increasing value of unroll factor. It can be seen that for greater unroll factors the range of values for latency gain moves to higher regions and the median value keeps ascending. The only exception takes place for unroll factor 4, which is the only value that does not divide the total number of iterations perfectly. Especially when fully unrolling the inner loop half the configurations provide latency gain from 75 to 95% and the median value is equal to 85%. The BRAM utilization seems to be exactly the same regardless of unrolling the inner loop and it is equal to the initial utilization with none of the directives applied. In Fig.5.8 we can see a small differentiation in BRAM utilization for some factors. These increases however can be attributed to the configurations that included unrolling of the inner loop without partitioning the arrays accessed in the loop body, as explained in Chapter 4. In Fig.5.7 it can be seen that when the inner loop is unrolled DSP,FF and LUT utilization range is slightly wider and in a higher region. This is again due to the fact that unrolling a loop leads to replicating the loop body and thus to more logic and area. As the value of unroll factor becomes greater so does the area utilization and the median value increases in strictly increasing order.

Fig.5.9 depicts the impact of partitioning the coefficient array sv_coef that is accessed in each iteration of the outer loop. It appears that its application does not define the performance since the range of latency gain extends from 0 to 100% and the median is equal to 50% whether the directive is applied or not. BRAM utilization practically remains the same and equal to the initial one and is thus irrelevant to sv_coef array partitioning. DSP, flip flop and LUT utilization range is slightly wider and shifted to higher regions when the directive is applied, and the median value is greater. In Fig.5.10 we can see that when the partition

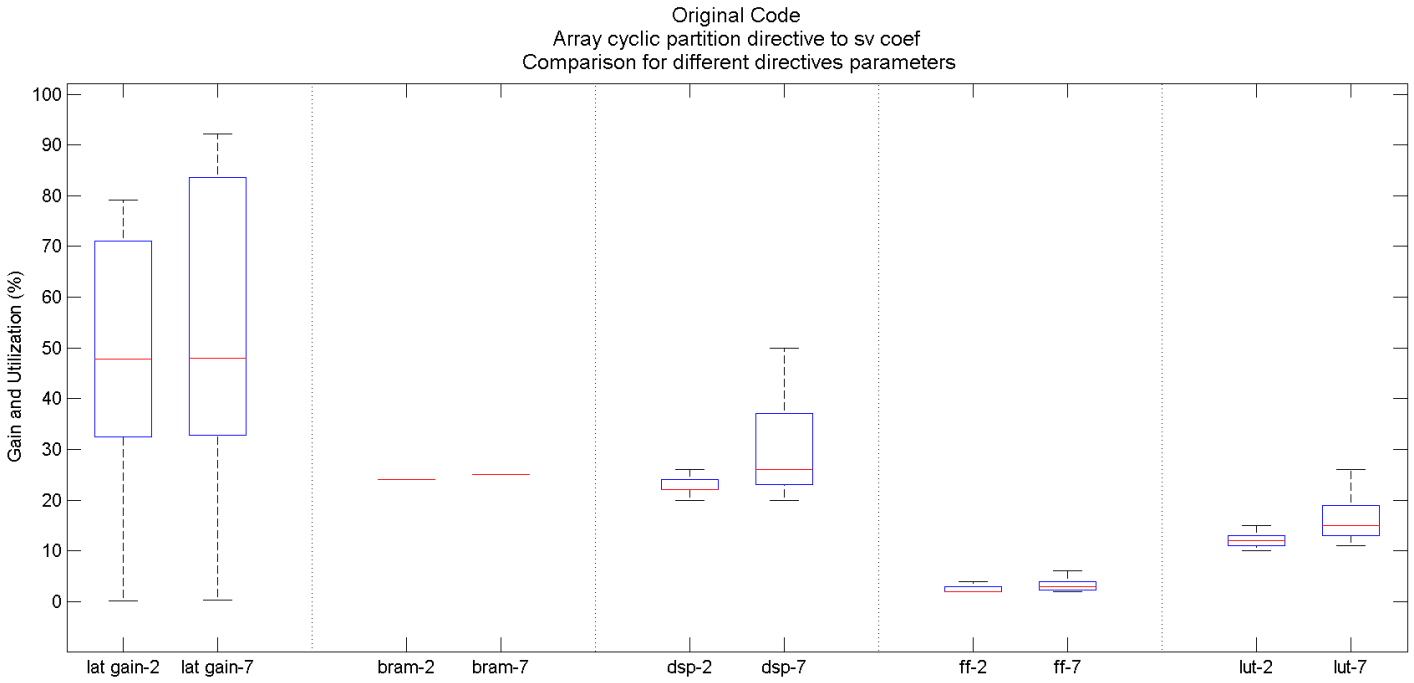


Figure 5.10: Changing partition factor on sv_coef. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization.

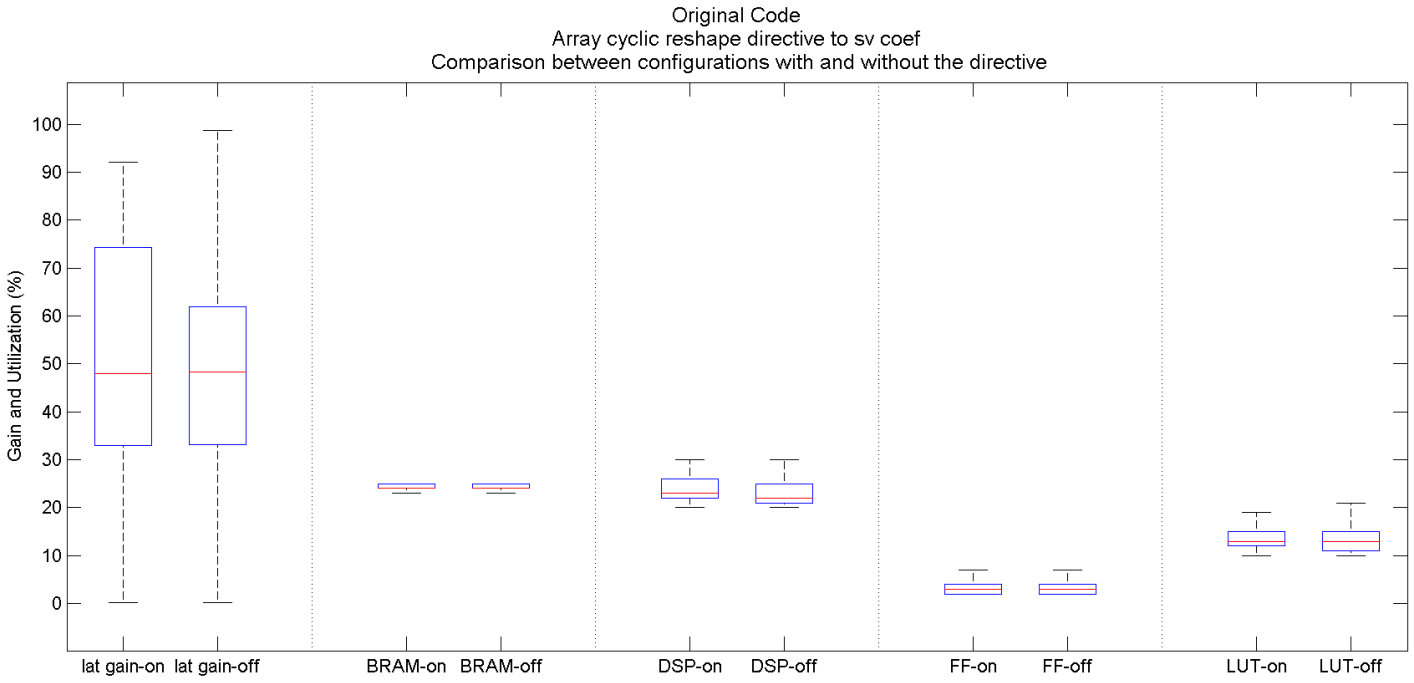


Figure 5.11: Reshaping sv_coef. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

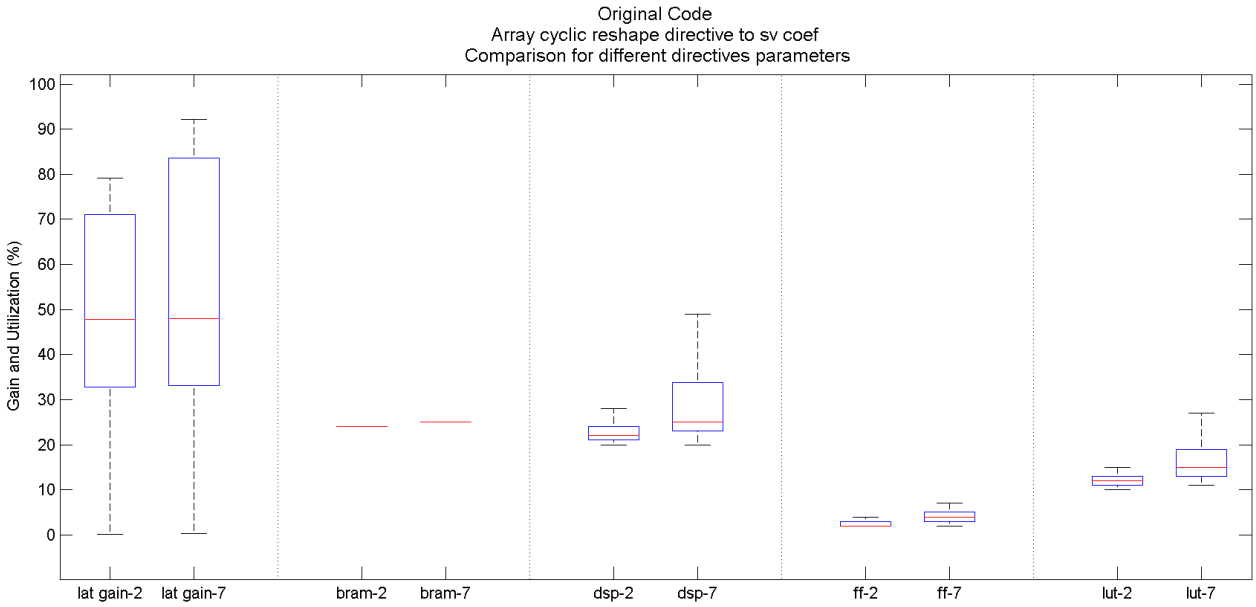


Figure 5.12: Changing reshape factor on sv_coef. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization.

factor is greater, this range becomes even wider and the median value greater. The values though remain considerably close to the ones of the initial design. The relative increase in the area, however small it might be, is due to the fact that the array partitioning is applied only when the outer loop is unrolled as well and the unroll and partition factors hold the same value. This happens because more than two accesses in an array is required for the partition directive to help increase parallelism. The unrolling of the outer loop leads to replication of the inside of the loop and thus to the increase in area illustrated in Fig.5.9. Applying the reshape directive instead for thw partition one inflict the same results as it can be seen from Fig.5.11 and Fig.5.12.

Fig.5.13 depicts the performance and resource metrics that result when applying the partition directive to the arrays accessed in the inner loop. The directive is applied only when the inner loop is unrolled and thus multiple accesses to the same array are necessary and more read ports are required. The arrays are partitioned by a factor equal to the unroll factor. The results of the configurations that do not include this directive do not provide us with useful information, since they include all possible latency gain values and the boxplot is perfectly symmetrical. The application of the directive though guarantees that the latency gain does not drop below 30% and three quarters of the configurations achieve a gain up to 70%. These results do not differentiate between various partition factors. In Fig.5.14 it shows that the greatest the partition factor is, the greatest the gain in latency is. For fully partitioning the sup_vectors array across the row dimension a gain in the range of 75 up to 95% is achieved for at least half the configurations. Having however examined all the directives, we reach the conclusion that the unrolling of the loop is actually the definitive factor in improving the latency but partitioning the arrays prevents memory burst and thus allows the configurations to be implementable. BRAM utilization has a wider range when the partition directive is on but it doesn't exceed a width of 5% and the median value remains the same and close to the initial one of the design. DSP, flip flop and LUT manifest a slightly wider range of utilization but half the configurations remain inside a narrow width of 5% at most. In Fig.5.14 where the results for different factors are depicted there are small differentiations

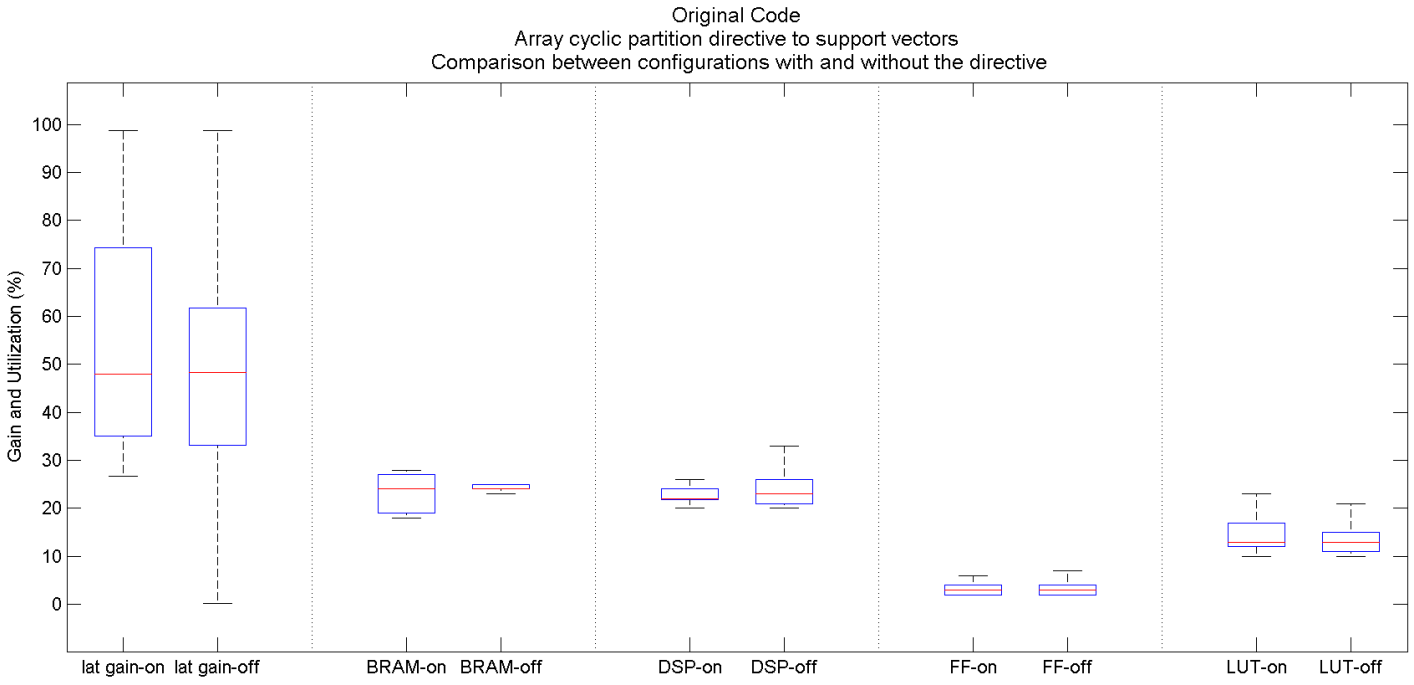


Figure 5.13: Partitioning sup_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

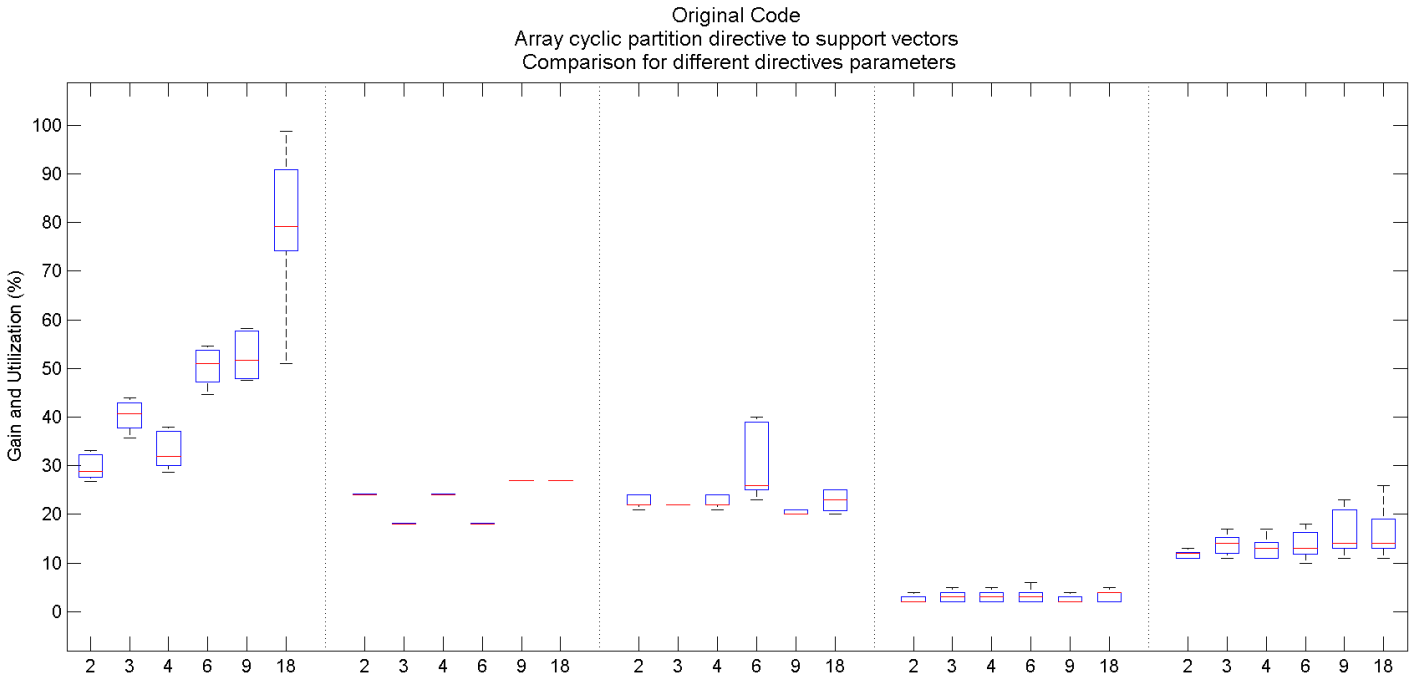


Figure 5.14: Changing partition factor on sup_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization.

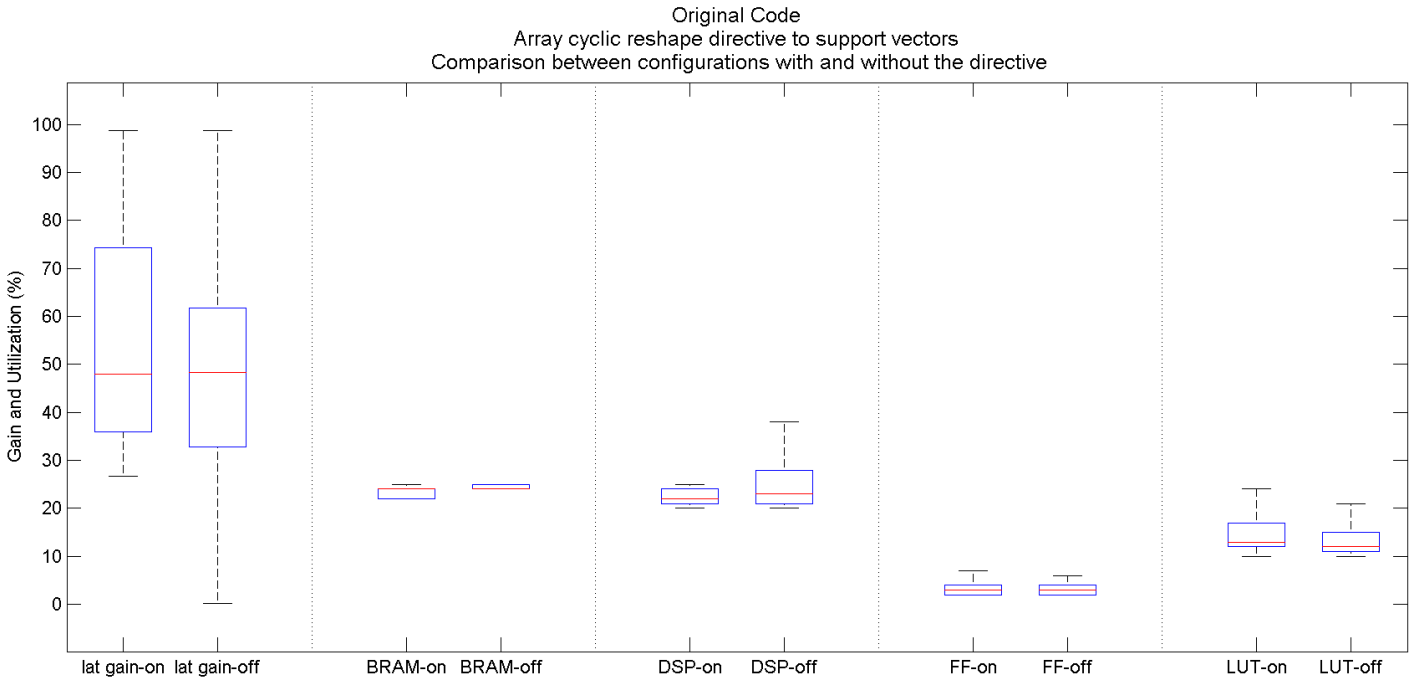


Figure 5.15: Reshaping sup_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

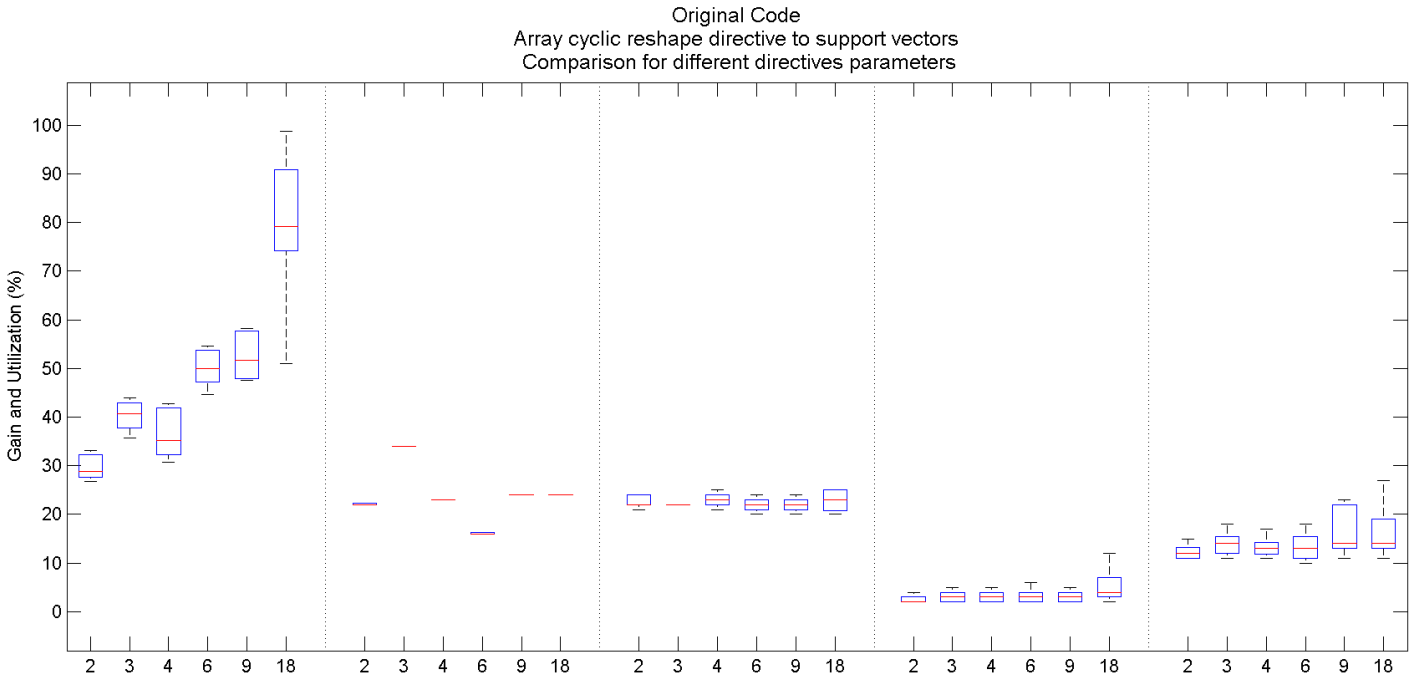


Figure 5.16: Changing reshape factor on sup_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization.

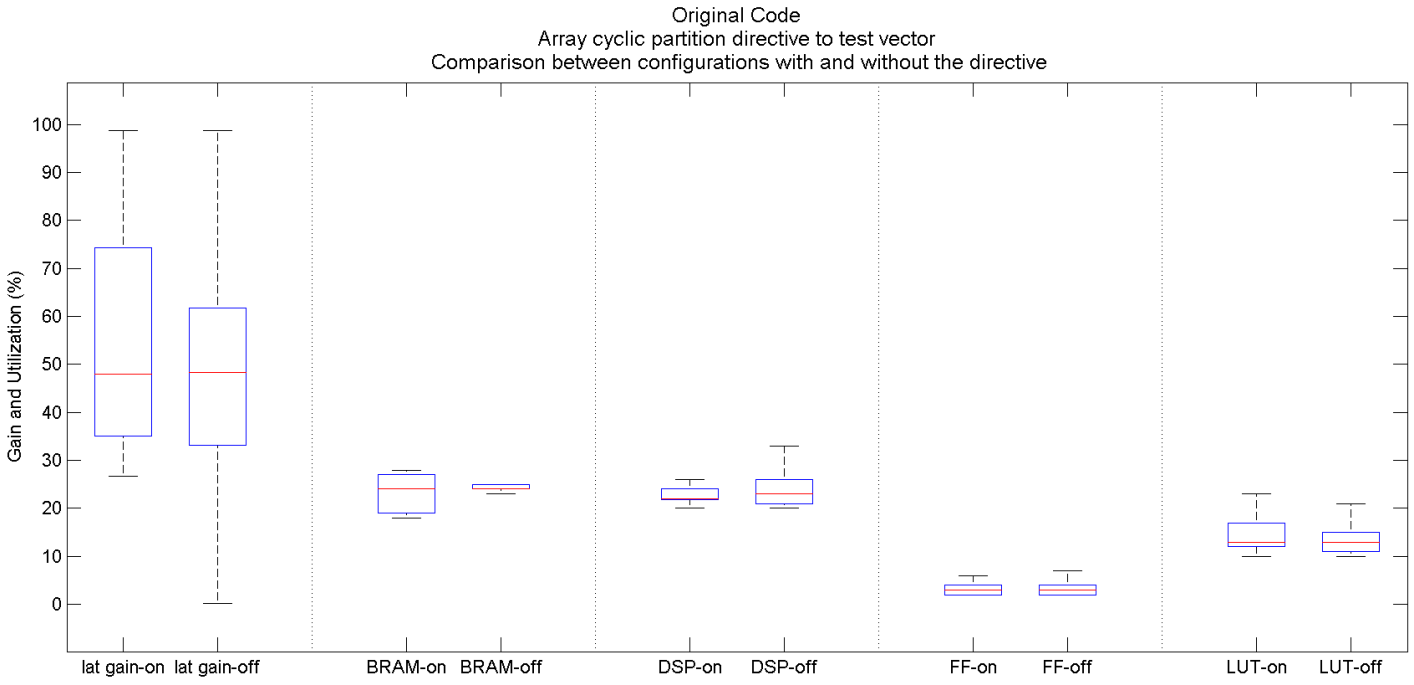


Figure 5.17: Partitioning test_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

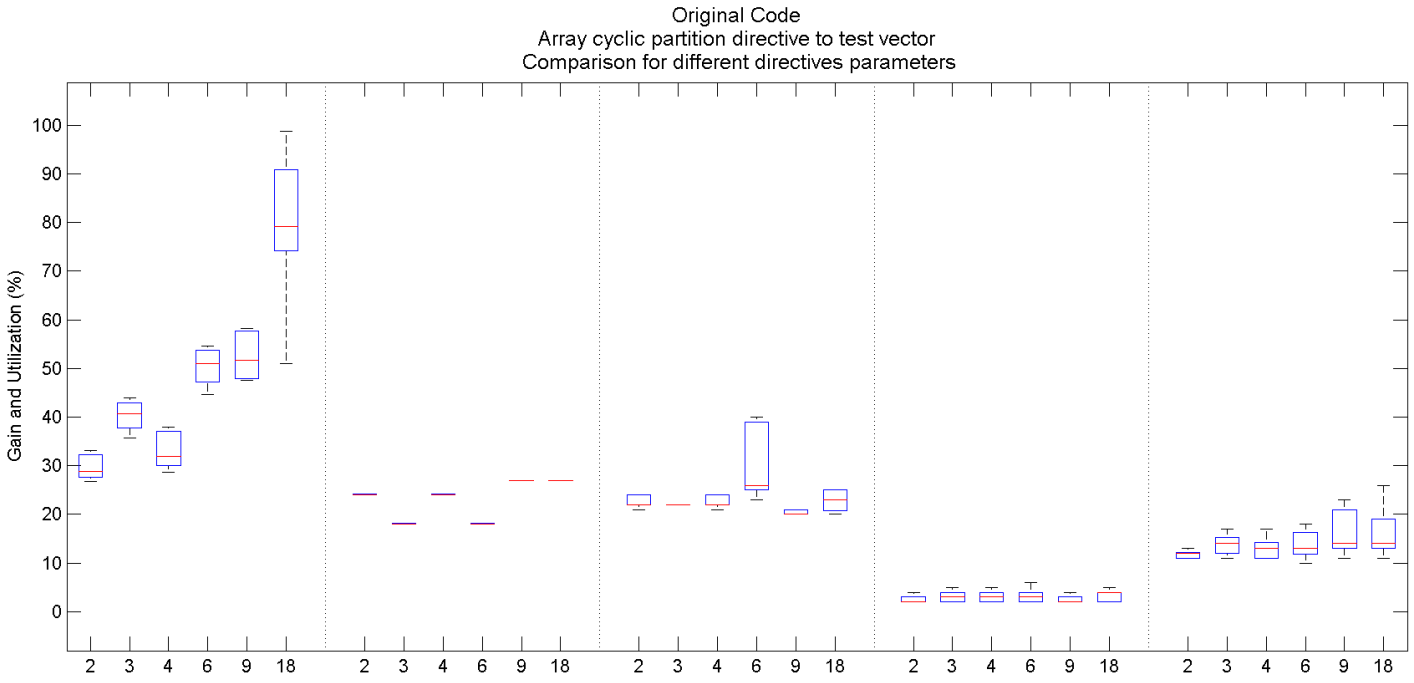


Figure 5.18: Changing partition factor on test_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization.

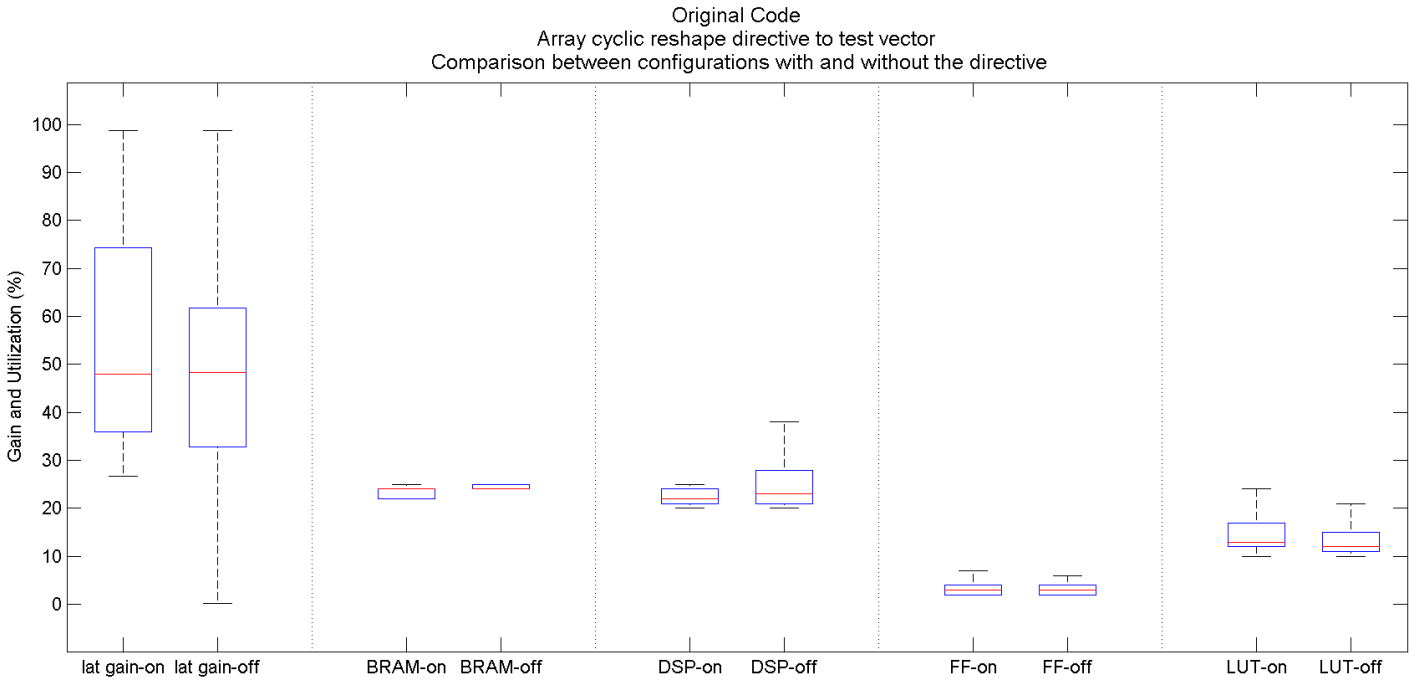


Figure 5.19: Reshaping test_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization with and without the directive.

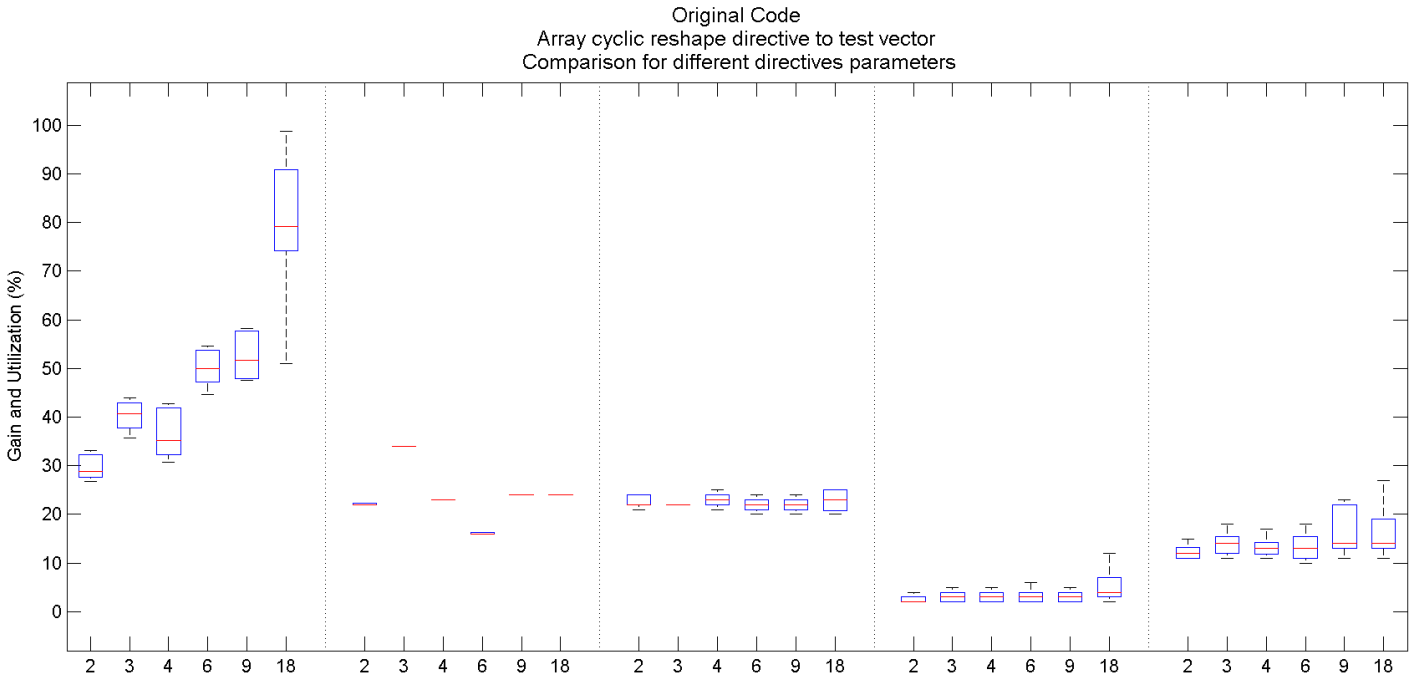


Figure 5.20: Changing reshape factor on test_vector. Columns from left to right:Latency gain and BRAM,DSP,FF,LUT utilization.

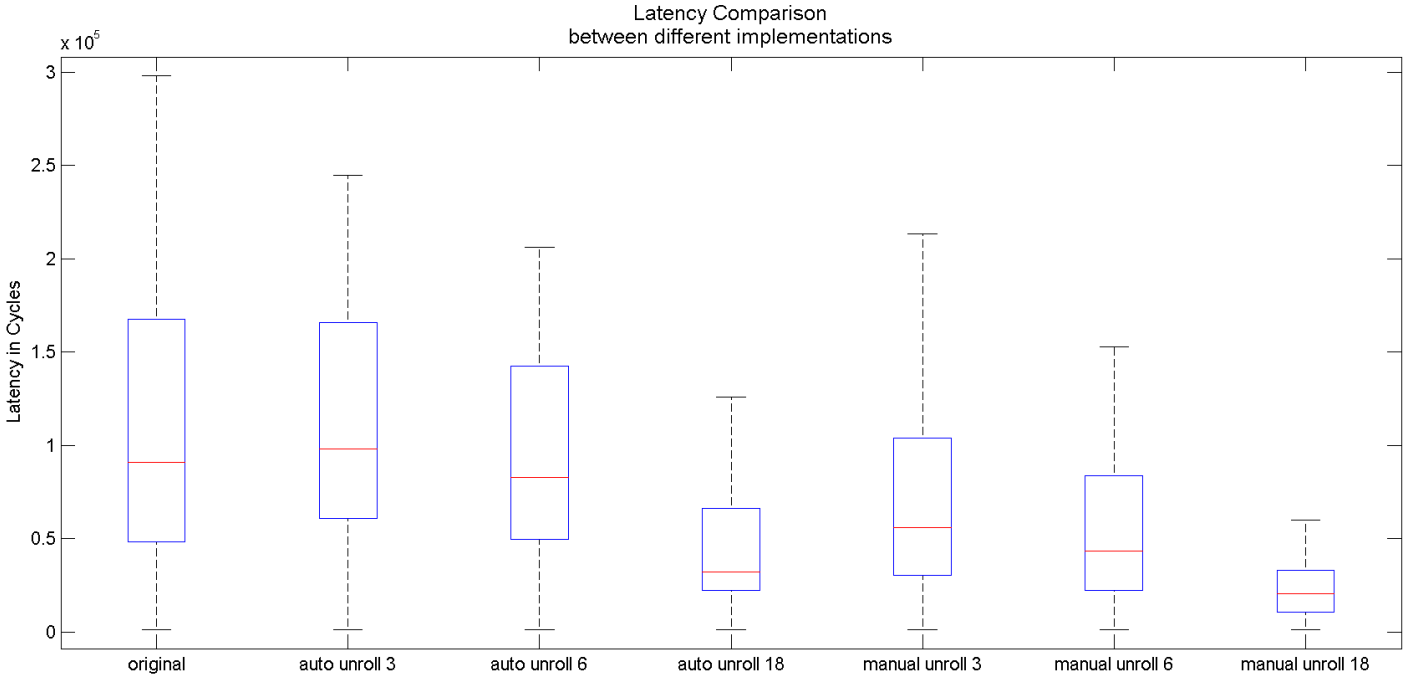


Figure 5.21: Latency comparison between all implementations

but they can be explained by the unrolling of the loop and the replication of logic that lead to an area increase. The same conclusions are drawn when the reshape directive is applied instead of the partition one.

5.3 Comparison of Implementations

The focus of this section is a comparison between the performance and area resources resulting from applying the directives described to different implementations. There are four basic versions of the classifier code: the original one and three versions with unrolled the inner loop by a factor of 3, 6 and 18 respectively. The conclusions are drawn from boxplots. Each boxplot represents a different implementation and contains all the configurations resulting from applying the directives to the corresponding version. In order to highlight the difference between the automatic and the proposed manual unrolling, there are also boxplots that include a subset of the configurations of the original version. In particular, each one contains only the configurations that include automatic unrolling by a factor of 3, 6 and 18 respectively.

In Fig.5.21 the results for latency measured in cycles are presented. The definitive effect of unrolling the inner loop is apparent. The greater the unroll factor is, the narrower is the range of latency values, the boxplot is shifted to smaller values and the median values follow a strictly decreasing path. The advantage of choosing the proposed manual over the automatic unrolling in regards to performance is also highlighted. Manual unrolling achieves even narrower boxplot width, smaller values and a lower median. It is worth mentioning that applying manual unrolling of factor 3 has better effects than automatic unrolling of factor 6.

Fig.5.22 illustrates the comparison for BRAM utilization. For automatic loop unrolling the median remains the same, the values fluctuate at the same levels and the width presents small

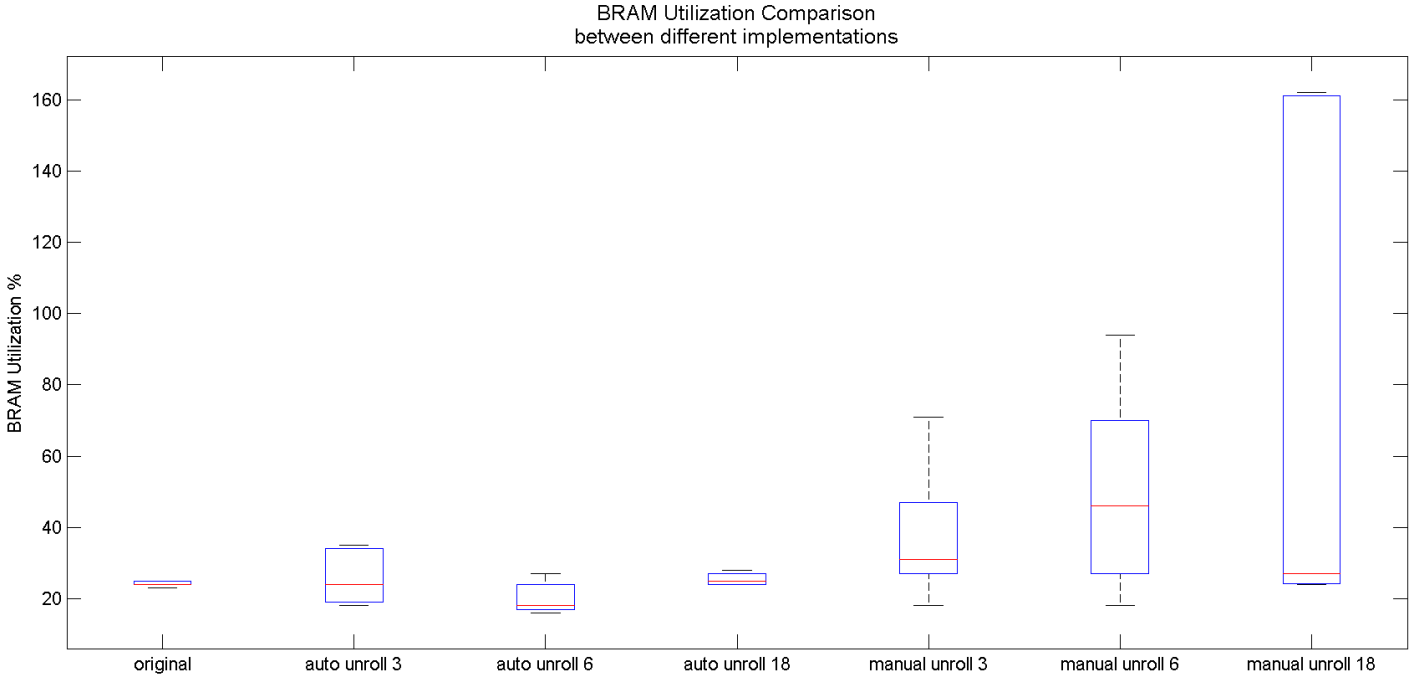


Figure 5.22: BRAM Utilization comparison between all implementations

changes. Manual unrolling though causes an increase in the width of each range, an increase in the values of utilization and in the median. These increases become more prominent the greater the unroll factor gets. They could be attributed to the configurations that include unrolling of the inner loop and not array partitioning or reshaping, leading to a replication of the arrays to achieve parallelism. It has to be stressed though that for fully unrolling the inner loop the median is equal to the initial BRAM utilization which means that 50% of the configurations exhibit BRAM utilization equal or less that that.

In Figures 5.23 to 5.25 the area resources of the different implementations are compared. In Fig.5.23 we can see that DSP utilization holds the same median value for automatic unrolling. For manual unrolling though there is a gradual increase of the values, the median and the width of the range of values. This is due to the replication of the loop body that introduces a trade-off between increase in parallelism and increase in area. For FF and LUT in case of automatic unrolling the median is of the same value and the range of values does not change significantly. However in case for manual unrolling there is an increase in the width of values, the absolute values taken and the median. In fact the greater the unroll factor is, the greater this increase is exhibited. This is again attributed to the replication of the loop body and the allocation of more resources from HLS to achieve the desired increase in parallelism.

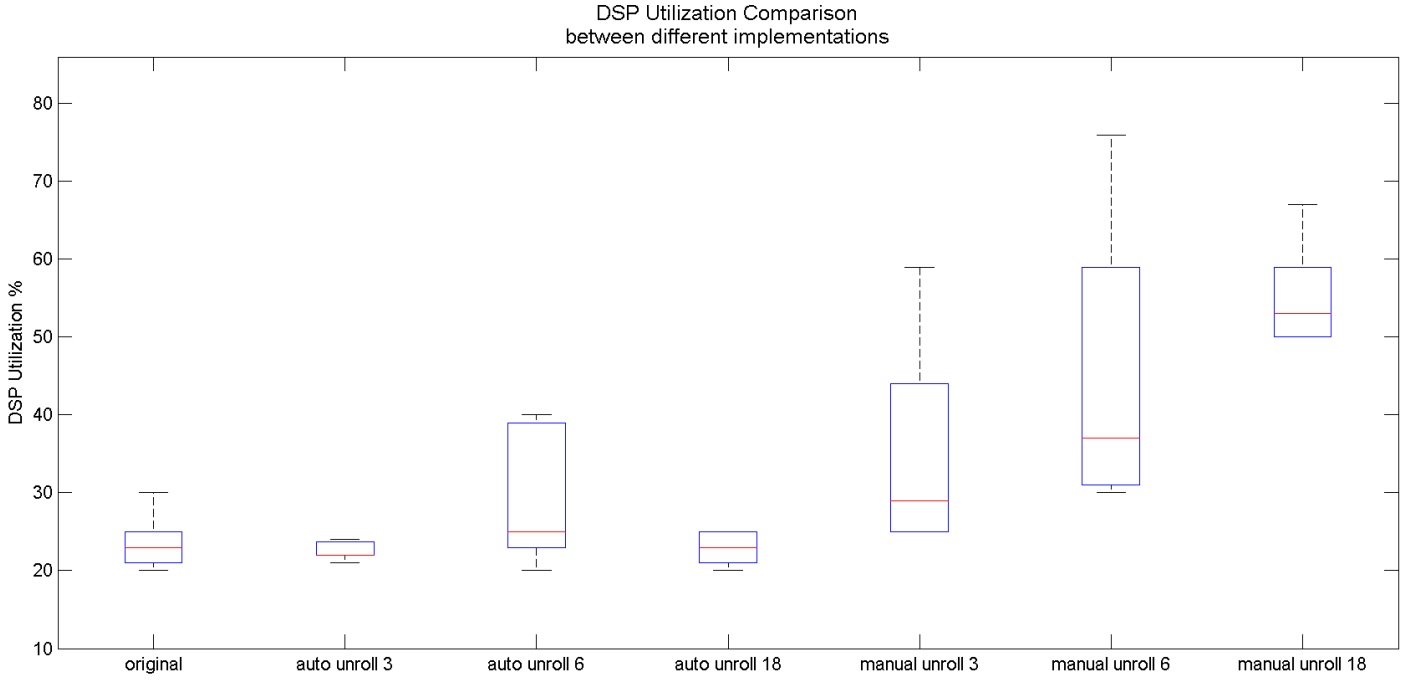


Figure 5.23: DSP Utilization comparison between all implementations

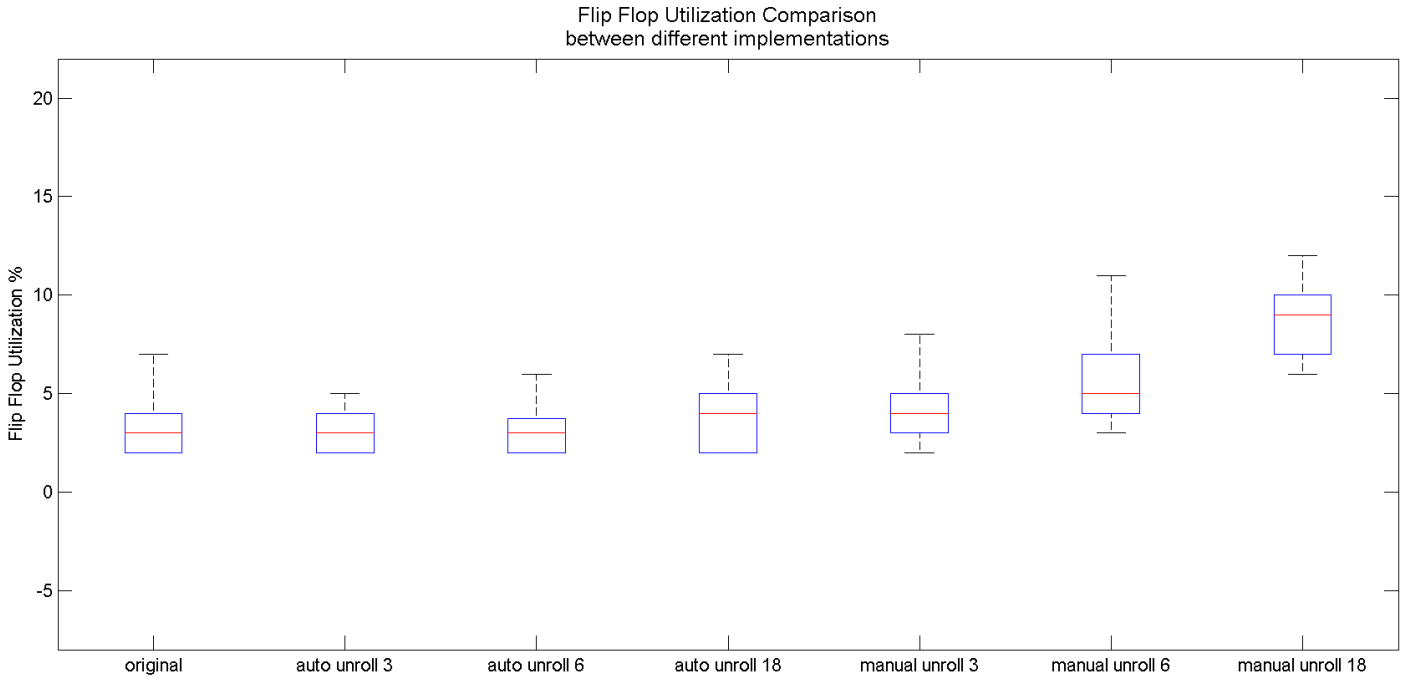


Figure 5.24: FF Utilization comparison between all implementations

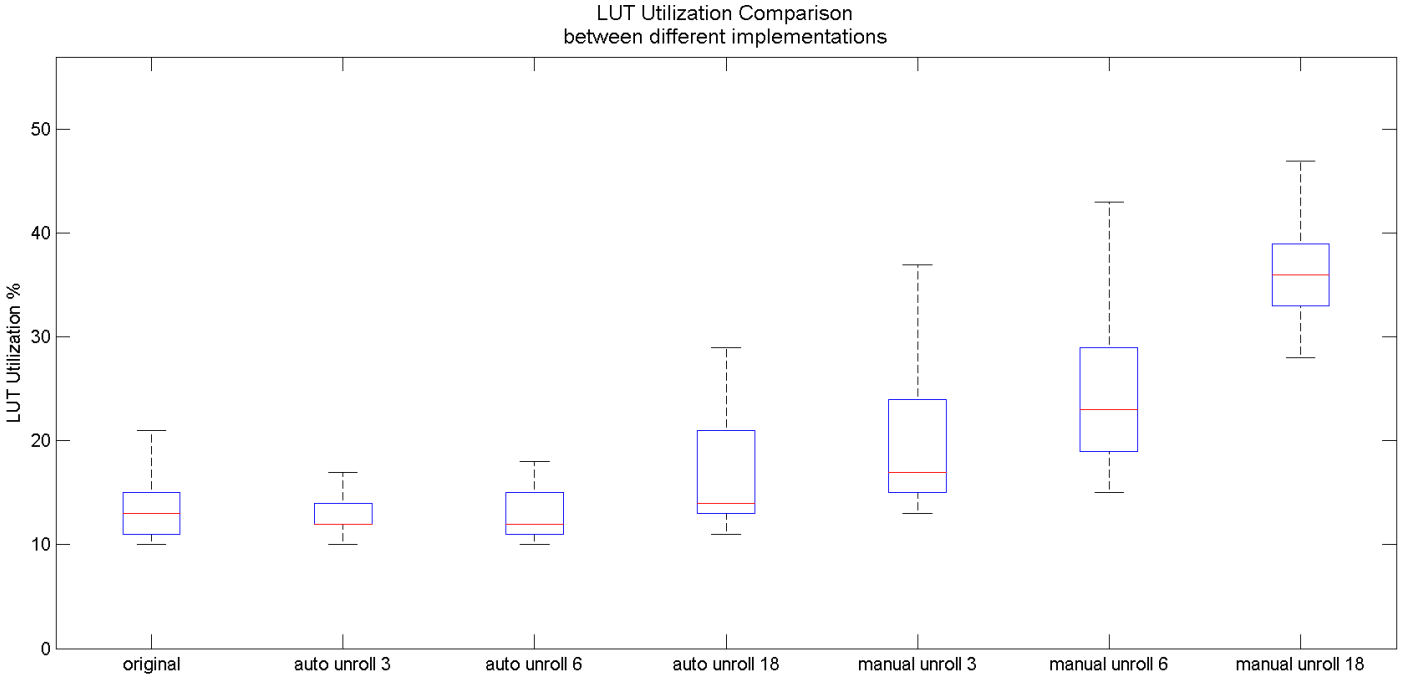


Figure 5.25: LUT Utilization comparison between all implementations

5.4 Optimal Configurations

Fig.5.26 displays the performance and area utilization comparison between the initial versions of the classifier and the optimal ones. The first four configurations correspond to the original classifier and the three versions with the unrolled loop. The following configuration is the optimal one of those applied to the original code, and the last one is the optimal of those applied to the restructured versions. For the three restructured implementations we observe a gradual increase in latency gain as the unroll factor becomes greater. This comes with fluctuations in BRAM utilization and an increase in area resources, both of which have been explained in detail in 4.2.2. The optimal solution of the original code depicted in this figure derives from several configuration. All of these configurations include fully unrolling the inner loop and then unrolling the inner loop by a factor of 2 combined with pipeling the inner loop and partitioning or reshaping the accessed arrays in various combinations. There is a further improvement in latency and a small increase in DSP utiliation. The latency gain reaches 80%. Finally the last case corresponds to the optimal synthesis among all the manual versions. It is based on manually and fully unrolling the inner loop, applying the pipeline directive to the outer loop and the array reshape directive to array sup_vectors. It achieves the best latency gain of all, equal to 98%, and it is accompanied with a significant increase in DSP and LUT utilization.

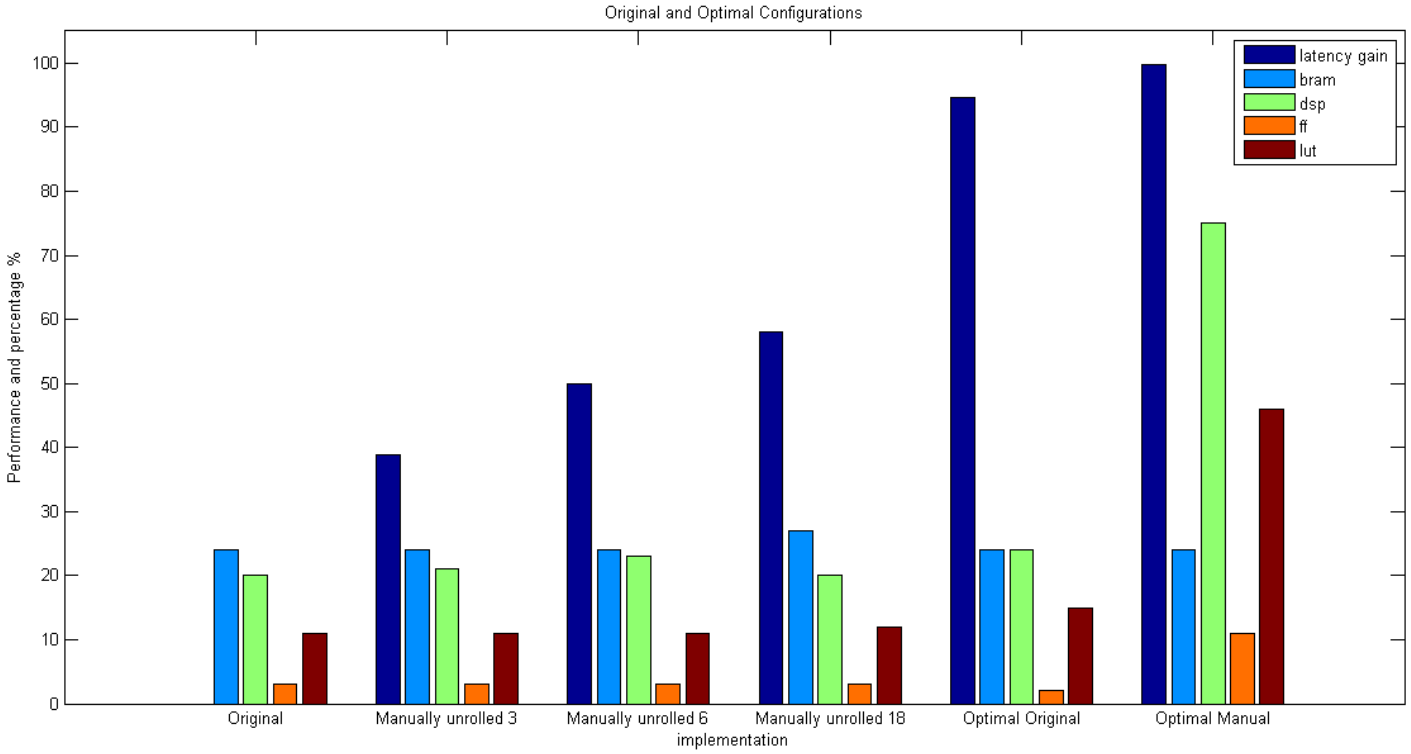


Figure 5.26: Optimal Configurations vs the original implementations

What is important apart from the exploration per directive is to understand and focus on the different architectural choices provided to the system designer in an effort of meeting different constraints and requirements. Towards this goal, in Fig.5.27 we present the design space created by the different versions of manually restructured code which emphasized on unrolling loop_j and utilize tree-structured performance of calculations. All manually created versions of the code are explored with all aforementioned HLS directives for each one of them. The figure includes only designs which managed to achieve their clock target since we distinguish the opposite case as infeasible. The X axis is a combined and weighted value of the utilization in all available metrics (BRAMs, DSPs, FFs and LUTs) while the Y axis is the estimated execution latency of the accelerator in ns derived from multiplying the required cycles for its execution with its operating clock period.

We can see that as indicated by the Pareto curvein Fig.5.27, there are many different non-dominated design choices which exhibit different characteristics. For example, optimum points in the left side of the design space exhibit the least HW resources utilization and fare relatively high on execution latency. As expected, such design points result from the original SVM code which is the least optimized and thus its execution is slow but its structure is simple thus requiring less resources. On the opposite side, the restructured versions of the code are on the right side of the figure meaning that the exhibit low execution latency and increased requirements in HW resources. This derives from their more complex structure and replicated functional units which enables parallel execution of calculations while imposing a toll on resources utilization.

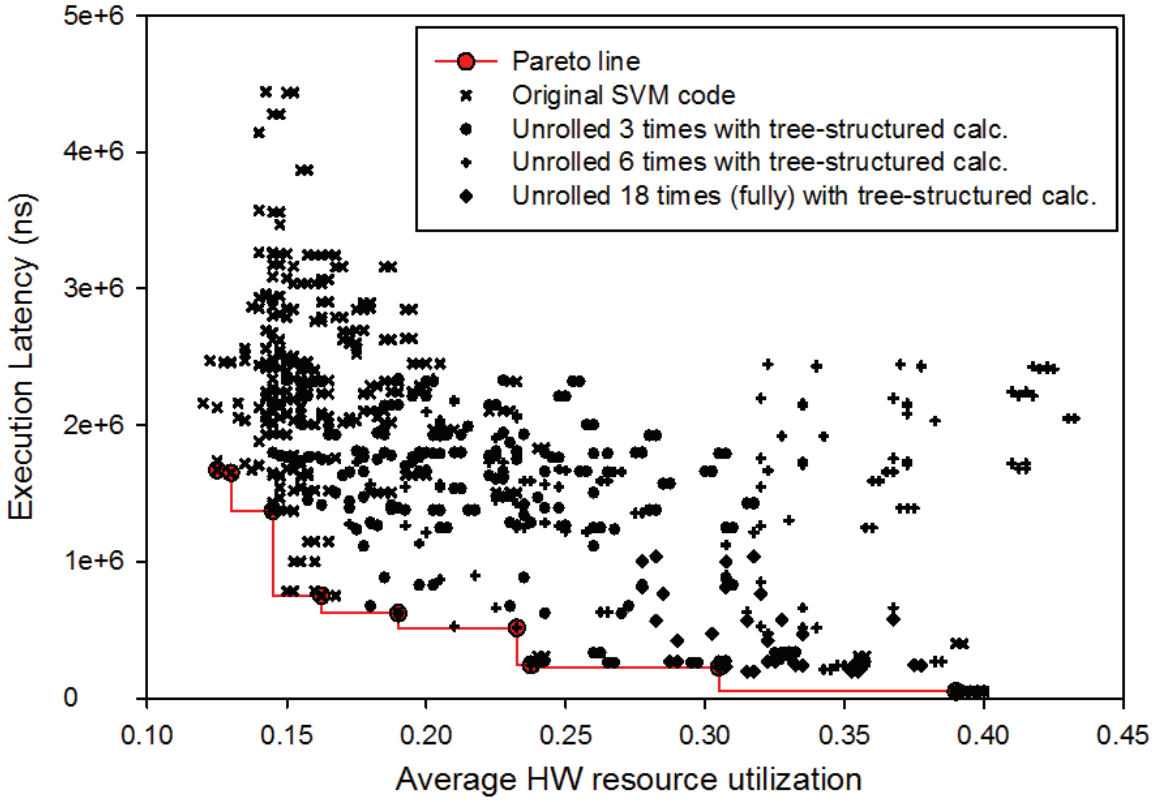


Figure 5.27: Pareto Curve

Chapter 6
Implementation on Zedboard

This Chapter presents the implementation of the accelerator on the Zedboard. The goal is to configure the Programmable Logic of the Zedboard with the IP that was created and communicate with it through a linux application that executes on the Processing Unit of the board. The linux on the board will boot from an SD card and the application will execute through a remote connection. The design tools that were used are mentioned in the following section and an elaborate description of their use is provided. The flow followed to build this infrastructure is given in Fig.6.1.

6.1 Implementation Description

The first stage includes the manufacturing of the accelerator using High Level Synthesis design tools. In this work the Vivado High Level Synthesis 2015.2 was used for the exploration of the design space formed by the different combinations of the chosen directives. When the exploration is conducted, a configuration of directives is decided to be applied to the original or the modified code. After the synthesis is completed, the RTL implementation is packaged into the chosen IP format. The packaged IP is intended to be synthesized using logic synthesis into the bitstream used to program an FPGA and includes the Verilog and VHDL design files. The arguments of the top-level function to be synthesized are implemented as I/O (input-and-output) ports in the final RTL design. Optimization directives allow these I/O ports to be implemented with a selection of I/O protocols. The I/O protocol should be selected to ensure the final design can be connected to other hardware blocks with the same I/O protocol. The SVM classifier IP in this work has two arguments: an input array that includes the features of the current beat used for classification and a pointer to an integer value where the IP writes the classification result at the end of the computation. Vivado HLS creates three types of ports on the RTL design:

- **Clock and Reset ports**: These ports are used as inputs to signals ap_clk and ap_rst which are the clock signal and the reset signal respectively.
- **Block-Level interface protocol**: By default, a block-level interface protocol is added to the design. The signals that this protocol includes control the block. They control when the block can start processing data (ap_start), indicate when it is ready to accept new inputs (ap_ready) and indicate if the design is idle (ap_idle) or has completed operation (ap_done). In this work, the default block level protocol is applied to the function.
- **Port Level interface protocol**:The final group of signals are the data ports. The I/O protocol created depends on the type of C argument and on the default. After the block-level protocol has been used to start the operation of the block, the port-level IO protocols are used to sequence data into and out of the block. An AXI4 slave interface is a port level interface protocol and is typically used to allow the design to be controlled

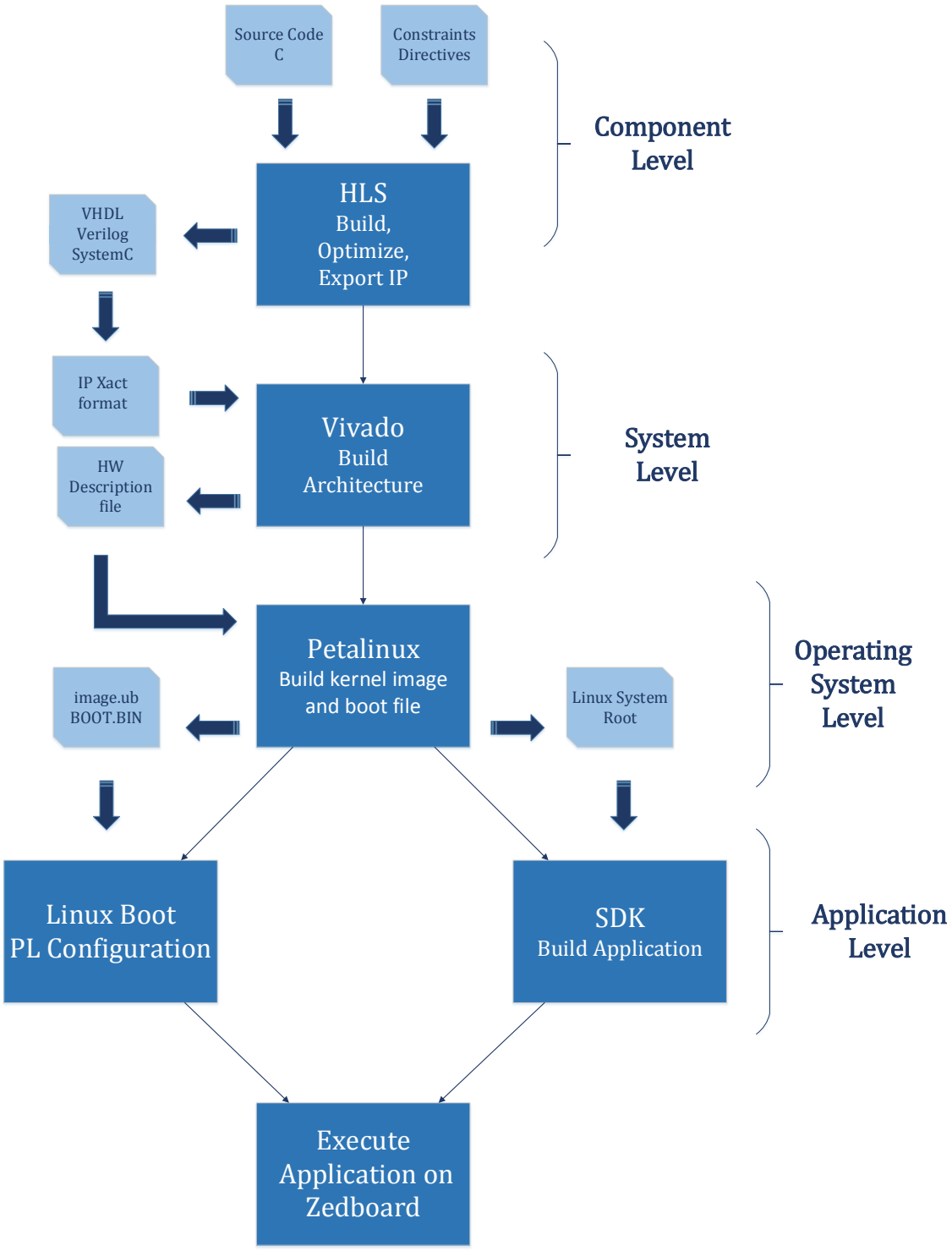


Figure 6.1: Implementation Flow

by some form of CPU or micro-controller. This is the chosen protocol applied to the arguments of the classifier IP. It is also applied to the function return argument leading to the creation of an interrupt port. This interrupt is very useful since it is driven from the block-level protocol ap_done port which indicates when the function has completed operation. If the function return is also specified as an AXI4-Lite interface (s_axilite) all the ports in the block-level interface are grouped into the AXI4-Lite interface. This is a common practice when another device, such as a CPU, is used to configure and control when this block starts and stops operation. Since the IP in this work was built as a software accelerator with the intent of communicating with the rest of the software application running on the PS, this strategy is also followed. Thus all the ports of the IP including the function return are grouped into the same AXI4 Slave Lite interface. Another advantage of this protocol is that when it is implemented, a set of C driver files are automatically created. These C driver files provide a set of APIs that can be integrated into any software running on a CPU and used to communicate with the device via the AXI4 Slave Lite interface. The most important file generated during this process is the hardware header file. This file is used at a later stage to build the application that runs on the CPU since it provides a complete list of the memory mapped locations for the ports grouped into the AXI4 Slave Lite interface [5].

The next stage is the implementation of the hardware system that includes both the PS and PL part of the board. It is completed using the Vivado Design Suite 2014.4 [28]. In Vivado a new project is created after specifying the Zedboard as the target board. Since the main part of the application needs to be executed on the PS side we select the Zynq7 Processing Unit from the available IPs to include in our architecture. A default configuration for the Zynq All Programmable SoC is applied. This configuration can be modified to meet the application requirements. In order for a Zynq hardware project to boot Linux one triple time counter (TTC), an external memory controller (DDR controller), UART for serial console, non-volatile memory (QSPI Flash,SD for example) and ethernet are the necessary PS requirements. In this work it is decided that Linux boot from an SD card, so the SD card peripheral has to be selected with SD card detection and write options enabled. The clock under which the PL side operates is determined in the Clock Configuration part of the PS customization process. The interrupts from PL to PS side also have to be enabled [29].

The PS IP added must communicate with the classifier IP on the PL side. This IP is the one exported from Vivado HLS in a Vivado IP Catalog format. The IP can be imported into the Vivado IP catalog for use in the Vivado Design Suite by adding the repository that contains the IP in the IP Catalog. An instance of the HLS created IP can then be added to the architecture under development. The signal connections between the two IPs are automatically done by Vivado, except for the connection between the interrupt port of the Classifier IP and the input interrupt port of the PS which is done manually. The PS now can communicate with the PL through the AXI, an advanced microcontroller bus architecture that is a part of AMBA. The AXI Interconnect of the IP with the PS uses AXI4 Memory Mapped Interfaces that are automatically converted to the AXI4 Lite Protocol which was added as an interface to the accelerator when it was built in HLS. One of the GP Master interfaces of the Zynq are used as ports. The Processing System implements the Master Interface and the IP the Slave Interface, which is controlled by the Master through the block level signals mentioned earlier. Fig.6.2 presents the system architecture as it is developed in Vivado and Fig.6.3 focus on the built IP and the interface of signals that it uses. After the architecture is built we build an HDL wrapper, run synthesis, implementation and generate the bitstream. We also export the hardware specification file including the bitstream. This file is going to be used in subsequent stages [5], [4].

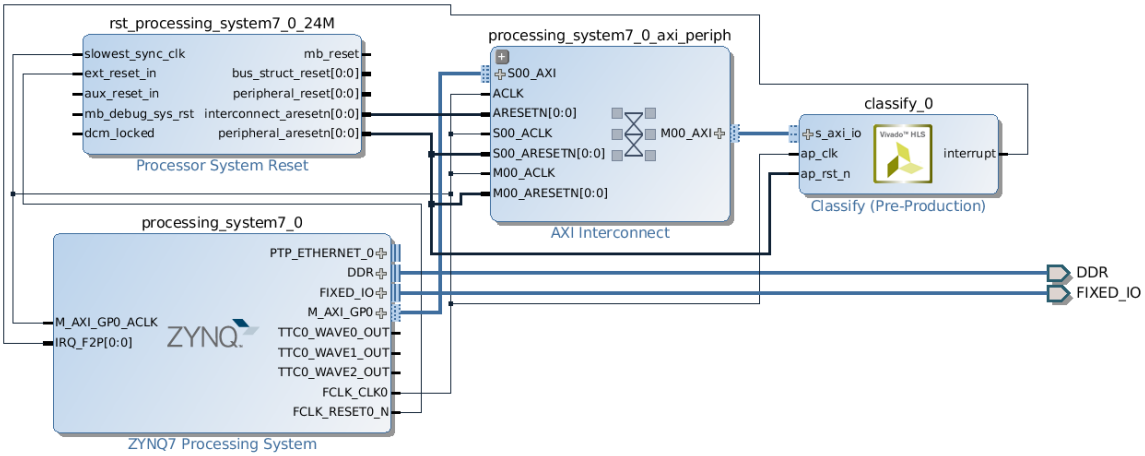


Figure 6.2: System Architecture Build in Vivado Design Suite

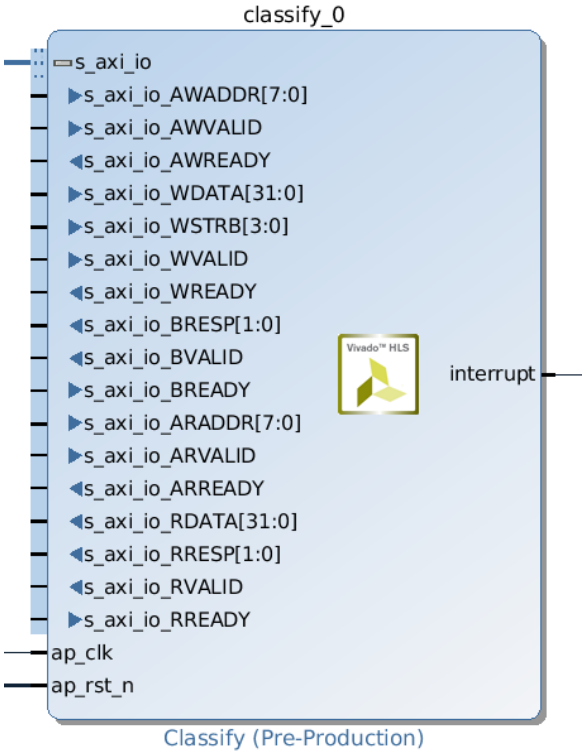


Figure 6.3: Customized IP in Vivado

The next step is to create a new PetaLinux software platform, ready for building a Linux system customized to the new hardware platform. The PetaLinux Tools offer everything necessary to customize, build and deploy Embedded Linux solutions on Xilinx processing systems. Tailored to accelerate design productivity, the solution works with the Xilinx hardware design tools to ease the development of Linux systems for Zynq-7000 All Programmable SoC. In this work Petalinux 2014.4 is used. First we create a new embedded Linux platform for the Zedboard. Then we customize the software platform template to precisely match the hardware system built in Vivado at the previous step. This is done by copying and merging the platform configuration files generated during the hardware build phase into the newly created software platform. The tool parses the hardware description file to get the hardware information to update the device tree, PetaLinux u-boot configuration files and the kernel config files. During configuration we also set the SD card as the boot device and configure the linux kernel to support built-in userspace I/O device drivers. Now that the device tree is updated to our hardware specifications we can see that a classifier IP instance is included to the file that contains the modules of the PL side. We also have to manually include it to a file with all the userspace I/O drivers. Finally we modify the file system configuration to support dropbear SSH and thus allow remote connection to the linux running on the Zedboard. Once the configuration is complete we build the system image. Having configured to use the SD card as the primary boot device, we create a boot image file that contains the Zynq All Programmable SoC FSBL, the BIT file for the programmable logic (PL) configuration from the Vivado project, u-boot, and the Linux image for the SD card boot. We copy the kernel image and the boot binary file to the SD card and use it to boot linux on the Zedboard. The PL is configurated and we connect a terminal to the Zynq Processing system through Gtkterm, which is a simple terminal used to communicate with a serial port [29].

The last stage includes building the application using Xilinx SDK 2014.4. The hardware specification file including the bitstream is exported from Vivado and imported to the SDK working directory. This way SDK has all the information needed for the hardware platform where the application will execute. At this point we choose to create a new application, select Linux as the OS platform and insert the Petalinux path for system root. The IP is included in the file system in directory "/dev/" as a userspace I/O device "uio0". "/dev/uio0" is a virtual file representing the memory map of the whole system. To access the device from user space, we can open "/dev/uio0", use mmap() to map the device to memory, and then we can access the device by using the pointer which points to the mapped memory [30]. To read or write to each separate argument of the IP we add the offsets provided at the HLS hardware header file to the pointer returned from mmap. To get the classification result for a beat the following process is followed:

- use memcpy to write the feature vectors to the given address space so that the IP has access to them
- set the ap_start block level signal to start the IP operation and reset it after a while
- wait until the control signal of the address where the IP writes the result is valid, which happens when the computation is over
- read the result from the address that holds the value of the result

6.2 Results

Two versions of the SVM classifier were implemented on the Zedboard following the steps described above. The first one was the original version with no structural modifications or optimization directives applied to it. The second one was the optimal configuration of the pareto curve. It included manually unrolling the inner loop, applying the pipeline directive to the outer loop and reshaping the sup_vectors array. A software only version of the classification was executed on the ARM Processor for comparison.

The testing set included 52291 test vectors, which were read from a file. Measurements were taken for the computation time per beat and for the time necessary for the test vector of each beat to be transferred from the PL to PS side. Also the total execution time, total transfer and total computation time were taken. The execution was repeated 5 times and the mean values were computed to eliminate mistakes.

The results show that the computation time is significantly reduced in the accelerated version. It is faster than both the original HW implementation and the SW one on ARM. The exact results are presented in Table 6.1 and demonstrated in Fig.6.4.

Table 6.1: Time measurements for different implementations.

	SW version		HW original version		HW accelerated version	
	Communication time(s)	Computation time(s)	Communication time(s)	Computation time(s)	Communication time(s)	Computation time(s)
per beat	-	0.002223635	0.00000449943	0.004047181	0.0000110643	0.0000521259
total	-	116.2761016	0.2352798	211.6311248	0.5785634	2.7257132

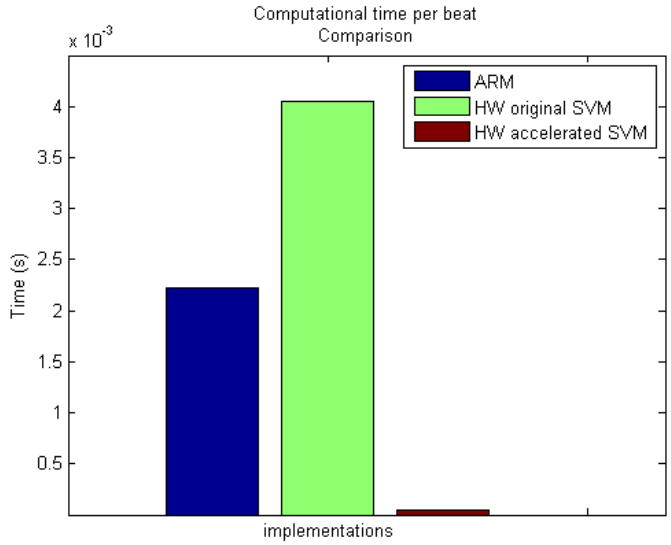


Figure 6.4: Computation time for HW and SW implementations.

The results confirm the ones anticipated, according to HLS synthesis reports. This is depicted in Fig.6.5 and the exact numbers are included in Table 6.2.

Table 6.2: Simulation vs Implementation Results

HLS synthesis			Zedboard implementation		
original (s)	accelerated (s)	gain (s)	original (s)	accelerated (s)	gain (s)
0.00412783	0.00005172	79.81	0.004047181	0.0000521259	77.64

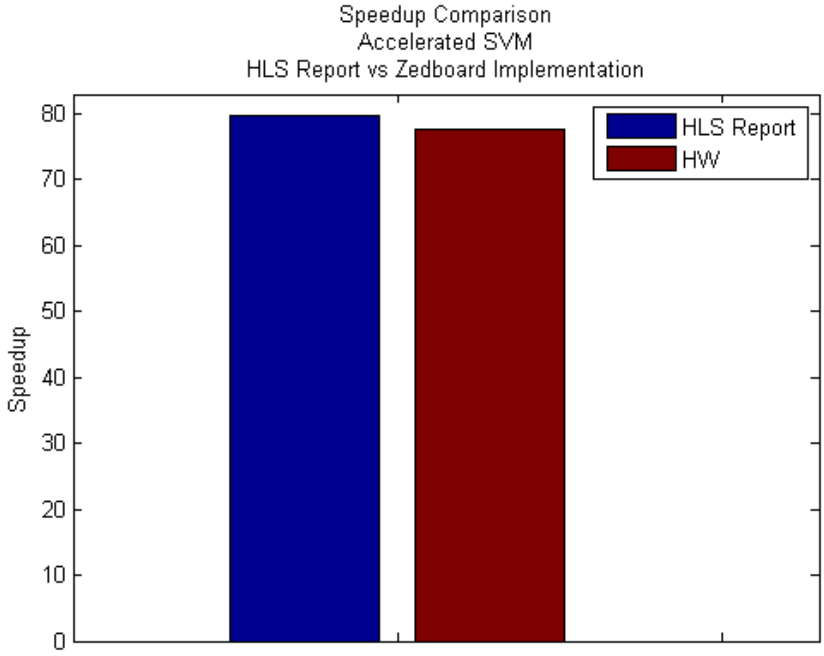


Figure 6.5: Gain comparison between simulation and implementation

Chapter 7
Conclusion

7.1 Summary

Arrhythmia detection for chronic patients suffering from various cardiovascular problems requires constant monitoring by recording the ECG signal and thus processing an enormous data set characterized by complex non-linear distribution among its samples. Given the complexity of deriving exact models for assessing the ECG signals and predicting the heart's condition, machine learning techniques have recently dominated the field of ECG analysis. Support Vector Machines particularly are widely used as classifiers and are often incorporated in ECG arrhythmia detection flow. In the detection algorithms, classification is found to pose the primary energy and performance bottleneck and is thus targeted for optimization.

In this work we have presented a methodology for creating efficient HLS based HW accelerators targeting Support Vector Machine based classifiers. The methodology relies on two parts. First the original code under acceleration is structurally transformed in order to assist the HLS tool to maximize the parallelization of the computational parts of the algorithm. Two code restructuring techniques are proposed, which are possible thanks to the dual inherent parallelism of the algorithm on a macroscopic and microscopic level. The first one is based on advancing coarse level parallelism and relies on the idea of executing multiple instances of the computational kernel at the same time, each instance operating on a subset of the initial set. The second technique includes implementing modifications to the code manually, instead of utilizing the provided HLS directives, combined with reshaping of arithmetic calculations. After these techniques have been applied, different baselines of the SVM classifier are available, apart from the initial one, each one accelerated to some extent. To further improve this performance we exploit the application of HLS directives to these baselines. The selection of these directives is again based on the inherent parallelism of the algorithm. We perform an exploration by combining all these directives and exclude configurations due to lack of compatibility or user defined constraints.

Results of applying this methodology achieve up to 99% execution latency gain compared to the original SVM code. This corresponds to the optimal configuration and comes with a significant increase in area resources such as DSP, LUTs and flip flops. However there are configurations that achieve a latency gain up to 80% without an extreme overhead in HW resources. Depending on the application requirements the configuration with the desired effect can be implemented. The original and the pareto optimal configuration were actually implemented on the Zedboard and the measurements were in agreement with those of the synthesis.

These results prove that HW acceleration and thus SW-HW codesign is in fact a valid solution when software acceleration techniques meet their limit. Another important conclusion though, drawn from the methodology described, is that designers should not rely solely on the available tools to optimize their accelerators. There is a limit to the capabilities of the tool. A preprocessing of the algorithm to extract a sufficient degree of parallelism could prove to be a key to the final extent of parallelization achieved. Furthermore it was observed that

HLS does not always implement the alterations required in the most effective way possible. Manually implementing some modifications annihilates problems that HLS directives cause on their own, and at the same time allows them to be implemented in a more performance and area efficient manner.

7.2 Future Work

The current work could be expanded towards various directions:

As far as the design space exploration is concerned, a more thorough study could be conducted. The most interesting choice would be examining the combination of the two restructuring strategies proposed. There are indications to believe that their combination would be the most effective one although an important overhead in area utilization is anticipated. Secondarily, an exploration around the optimal points derived from pareto analysis could be performed, to validate that the user defined constraints in the selection of the directives and their combinations did not lead to pruning of better solutions.

Another direction would be to assess the derived co-processor in terms of energy requirements. An energy analysis could be performed to find out if the proposed architecture meets the energy requirements in addition to the performance ones.

Apart from the classification application it could be interesting to examine the development of co-processors for other parts of the ECG analysis flow as well. The feature extraction stage that is based on Discrete Wavelet Transform would be an ideal candidate. In that case, except for the optimization of the IP using HLS, exploiting Zedboard capabilities to achieve communication between the IPs without the intervention of the Processing System could prove to be both time efficient and energy saving.

Finally the application of the suggested restructuring strategies to other cores with stencil computations would be an interesting expansion of the current work.

References

[1] "Heart anatomy." http://freecoloringpages.co.uk/?q=heart+anatomy. xxixxxix, 4

[2] A. Szczepański and K. Saeed, "A mobile device system for early warning of ecg anomalies," *Sensors*, vol. 14, no. 6, pp. 11031–11044, 2014. xxixxxix, 5

[3] M. Shoaib, N. K. Jha, and N. Verma, "Algorithm-driven architectural design space exploration of domain-specific medical-sensor processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 10, pp. 1849–1862, 2013. xxixxxix, xxixxxix, 1, 6, 7, 10

[4] S. Neuendorffer and F. Martinez-Vallina, "Building zynq® accelerators with vivado® high level synthesis.," in *FPGA*, pp. 1–2, 2013. xxixxxix, xxixxxix, 12, 13, 14, 17, 52

[5] *Vivado Design Suite, User Guide, High-Level Synthesis*. May 2014. xxxiixxxii, 12, 13, 15, 52

[6] R. M. Rangayyan and N. P. Reddy, "Biomedical signal analysis: a case-study approach," *Annals of Biomedical Engineering*, vol. 30, no. 7, pp. 983–983, 2002. 1

[7] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995. 1, 6

[8] F. I. Marcus, J. N. Ruskin, and B. Surawicz, "Arrhythmias," *Journal of the American College of Cardiology*, vol. 10, no. 2s1, pp. 66A–72A, 1987. 1

[9] E. D. Übeyli, "Ecg beats classification using multiclass support vector machines with error correcting output codes," *Digital Signal Processing*, vol. 17, no. 3, pp. 675–684, 2007. 1, 6

[10] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, 1990. 1, 11

[11] "Cardiac cycle." https://en.wikipedia.org/wiki/Cardiac_cycle. 3

[12] "Electrocardiography." https://en.wikipedia.org/wiki/Electrocardiography. 3

[13] "T wave." https://en.wikipedia.org/wiki/T_wave. 3

[14] "Cardiac arrhythmia." https://en.wikipedia.org/wiki/Cardiac_arrhythmia. 4

[15] A. Gacek and W. Pedrycz, *ECG signal processing, classification and interpretation: a comprehensive framework of computational intelligence.* Springer Science & Business Media, 2011. 4

[16] G. B. Moody and R. G. Mark, "The impact of the mit-bih arrhythmia database," *Engineering in Medicine and Biology Magazine, IEEE*, vol. 20, no. 3, pp. 45–50, 2001. 4, 5

[17] J. Pan and W. J. Tompkins, "A real-time qrs detection algorithm," *Biomedical Engineering, IEEE Transactions on*, no. 3, pp. 230–236, 1985. 5

[18] D. T.-W. Hau and E. W. Coiera, "Learning qualitative models from physiological signals," Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1994. 7

[19] G. Meyfroidt, F. Güiza, J. Ramon, and M. Bruynooghe, "Machine learning techniques to examine large patient databases," *Best Practice & Research Clinical Anaesthesiology*, vol. 23, no. 1, pp. 127–143, 2009. 7, 9

[20] B. Gyselinckx, R. J. Vullers, C. Van Hoof, J. Ryckaert, R. F. Yazicioglu, P. Fiorini, and V. Leonov, "Human++: Emerging technology for body area networks.," in *VLSI-SoC*, pp. 175–180, 2006. 7

[21] Z. Nie, L. Wang, W. Chen, T. Zhang, and Y. Zhang, "A low power biomedical signal processor asic based on hardware software codesign," in *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*, pp. 2559–2562, IEEE, 2009. 7

[22] C.-W. Hsu, C.-C. Chang, C.-J. Lin, *et al.*, "A practical guide to support vector classification," 2003. 9, 10

[23] M. U. Guide, "The mathworks," *Inc., Natick, MA*, vol. 5, p. 333, 1998. 11

[24] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm. 11

[25] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, no. 4, pp. 8–17, 2009. 13

[26] F. Digilent's ZedBoard Zynq, "Dev. board documentation." 16

[27] "Zedboard hardware user's guide." `http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf`. 16

[28] T. Feist, "Vivado design suite," *White Paper*, 2012. 52

[29] "Embedded linux on zynq using vivado." `http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-embedded-linux-zynq.html`. 52, 54

[30] "Zynq design from scratch. sven andersson's blog." `http://svenand.blogdrive.com/archive/160.html#.VphCXpOli1E`. 54