ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Exploring the Impact of Pre-alignment filtering on HLS-based DNA short read alignment Accelerators

Διπλωματική Εργασία

του

Παναγιώτη Α. Καούκη

Επιβλέπων : Δημήτριος Σούντρης
Καθηγητής ΕΜΠ

Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων
Αθήνα, Νοέμβριος 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Exploring the Impact of Pre-alignment filtering on HLS-based DNA short read alignment Accelerators

Διπλωματική Εργασία

του

Παναγιώτη Α. Καούκη

Επιβλέπων : Δημήτριος Σούντρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4η Νοεμβρίου 2025.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Δημήτριος Σούντρης	Σωτήριος Ξύδης	Γεώργιος Ζερβάκης
Καθηγητής	Επίκουρος Καθηγητής	Επίκουρος Καθηγητής
ΕΜΠ	ΕΜΠ	Πανεπιστήμιο Πατρών

Αθήνα, Νοέμβριος 2025

(Υπογραφή)

.....

Καούκης Παναγιώτης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

Genomic analysis relies heavily on transforming raw sequencing data into complete and interpretable genomic information. At the heart of this process lies the alignment step, where billions of sequencing reads are mapped to a reference genome to reconstruct the original DNA structure. Despite significant algorithmic progress, the alignment extension phase—particularly the Matrix Fill and Traceback steps—remains a major computational bottleneck due to its high time and resource demands.

This thesis explores two complementary approaches to alleviate these challenges: (i) the optimization and hardware acceleration of alignment algorithms, and (ii) the reduction of alignment workload through intelligent filtering. Field-Programmable Gate Arrays (FPGAs) are investigated as a hardware acceleration platform due to their fine-grained parallelism and configurability, which enable efficient implementation of alignment algorithms such as Smith-Waterman, GenASM, and WFA. Furthermore, the SneakySnake pre-filtering algorithm is employed to analyze datasets, identify edit distributions, and guide algorithmic optimizations.

By combining dataset-aware pre-filtering with hardware acceleration techniques, this work aims to minimize redundant computations, reduce the search space during alignment, and achieve substantial performance gains without compromising accuracy. The proposed system leverages insights from SneakySnake to dynamically adapt alignment strategies, demonstrating speedups relative to software-based implementations. Experimental results and architectural evaluations validate the effectiveness of the proposed approach and highlight promising directions for future optimization in genomic data processing pipelines. **Keywords**— Alignment, Pre-filtering, Genomics, Hardware Acceleration, Data-aware system design

Περίληψη

Η γονιδιωματικη ανάλυση βασίζεται σε μεγάλο βαθμό στη μετατροπή των αρχικών δεδομένων αλληλούχησης σε πλήρη και ερμηνεύσιμη γενετική πληροφορία. Στο επίκεντρο αυτής της διαδικασίας βρίσκεται το στάδιο της ευθυγράμμισης, κατά το οποίο δισεκατομμύρια αλληλουχίες χαρτογραφούνται σε ένα πρότυπο γονιδίωμα προκειμένου να ανακατασκευαστεί η αρχική δομή του DNA. Παρά την αξιοσημείωτη πρόοδο στους αλγορίθμους ευθυγράμμισης, η φάση της επέκτασης της ευθυγράμμισης , ιδίως τα βήματα Matrix Fill και Traceback , εξακολουθεί να αποτελεί βασικό υπολογιστικό σημείο συμφόρησης λόγω των υψηλών απαιτήσεων σε χρόνο και υπολογιστικούς πόρους.

Η παρούσα διπλωματική εργασία εξετάζει δύο συμπληρωματικές προσεγγίσεις για την αντιμετώπιση αυτών των προκλήσεων: (ι) τη βελτιστοποίηση και επιτάχυνση των αλγορίθμων ευθυγράμμισης μέσω υλικού, και (ιι) τη μείωση του φόρτου της ευθυγράμμισης μέσω έξυπνου προ-φιλτραρίσματος. Τα Field-Programmable Gate Arrays (FPGAs) διερευνώνται ως πλατφόρμα επιτάχυνσης υλικού λόγω του υψηλού βαθμού παραλληλισμού και της δυνατότητας παραμετροποίησης που προσφέρουν, κάτι που επιτρέπει την αποδοτική υλοποίηση αλγορίθμων ευθυγράμμισης όπως οι Smith-Waterman, GenASM και WFA. Επιπλέον, ο αλγόριθμος προφιλτραρίσματος SneakySnake χρησιμοποιείται για την ανάλυση των συνόλων δεδομένων, την αναγνώριση της κατανομής των διαφορών (edits) και τη βελτίωση της απόδοσης των αλγορίθμων.

Με τον συνδυασμό προ-φιλτραρίσματος βασισμένου στα χαρακτηριστικά των δεδομένων και τεχνικών επιτάχυνσης μέσω υλικού, η εργασία στοχεύει στη μείωση των περιττών υπολογισμών, στον περιορισμό του χώρου αναζήτησης κατά την ευθυγράμμιση και στην επίτευξη σημαντικών βελτιώσεων στην απόδοση χωρίς να θυσιάζεται η ακρίβεια. Το προτεινόμενο σύστημα αξιοποιεί τις πληροφορίες από το SneakySnake ώστε να προσαρμόζει δυναμικά τις στρατηγικές ευθυγράμμισης, επιτυγχάνοντας επιταχύνσεις σε σχέση με τις υλοποιήσεις αποκλειστικά σε λογισμικό. Τα πειραματικά αποτελέσματα και οι αρχιτεκτονικές αξιολογήσεις επιβεβαιώνουν την αποτελεσματικότητα της προτεινόμενης προσέγγισης και αναδεικνύουν ελπιδοφόρες κατευθύνσεις για περαιτέρω βελτιστοποίηση στις διαδικασίες επεξεργασίας γονιδιωματικών δεδομένων.

Keywords— Αντιστοίχιση, Φιλτράρισμα,Γονιδιωματική, Επιτάχυνση Υλικού,Σχεδίαση με βάση τα Δεδομένα

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή μου, κ. Δημήτριο Σούντρη, για την εμπιστοσύνη που μου έδειξε και την ευκαιρία να εκπονήσω αυτή τη διπλωματική εργασία μέσω του εργαστηρίου του MicroLab. Ένα μεγάλο ευχαριστώ στην διδάκτρορα κ. Κωνσταντίνα Κολιογεώργη και στον Επίκουρο Καθηγητή, κ. Σωτήριο Ξύδη, για την καθοδήγηση και την πολύτιμη βοήθεια που μου προσέφεραν. Θα ήθελα ακόμα να ευχαριστήσω τους φίλους μου που συνέβαλλαν στο να γίνουν τα φοιτητικά χρόνια ένα από τα πιο ευχάριστα ταξίδια της ζωής μου. Τέλος ένα τεράστιο ευχαριστώ στην οικογένεια μου για την ανιδιοτελή εμπιστοσύνη και υποστηρίξη που μου έδωσαν καθ' όλη την ακαδημαϊκή μου πορεία.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

Η γονιδιωματική είναι η συστηματική μελέτη του συνολικού γενετικού υλικού (DNA) των ζωντανών οργανισμών. Αποκαλύπτει κρίσιμες πληροφορίες που μπορούν να μεταμορφώσουν τη βιολογία, να αλλάξουν τον τρόπο με τον οποίο αντιμετωπίζουμε ασθένειες και να επηρεάσουν την κατανόησή μας για την εξέλιξη. Πιο συγκεκριμένα, επιτρέπει την ακριβή διάγνωση ασθενών, την επιτήρηση παθογόνων, και στην περίπτωση φυτικών οργανισμών, ακόμη και τη βελτίωση καλλιεργειών. Όλες αυτές οι εφαρμογές βασίζονται στη μετατροπή των αρχικών γενωμικών δεδομένων (reads) που παράγονται από πλατφόρμες αλληλούχησης σε πλήρη γονιδιώματα. Τα γονιδιώματα αυτά συγκρίνονται έπειτα με ένα πρότυπο γονιδίωμα και μέσω διαδικασιών όπως η εύρεση γενετικών παραλλαγών και η επιγενετική ανάλυση, εξάγεται χρήσιμη πληροφορία για τα υπό μελέτη δείγματα. Η διαδικασία που βρίσκεται στο επίκεντρο αυτού του υπολογιστικού σωλήνα είναι το στάδιο της ευθυγράμμισης.[1][2]

Κατά την ευθυγράμμιση, δισεκατομμύρια reads χαρτογραφούνται στο πρότυπο γονιδίωμα, ώστε να ανακατασκευαστεί η συνολική δομή του DNA. Οι περισσότεροι ευθυγραμμιστές χρησιμοποιούν μία στρατηγική που ονομάζεται «seed and extend», όπου τα reads χωρίζονται σε μικρότερα τμήματα (seeds), με στόχο την εύρεση ακριβών αντιστοιχιών στο πρότυπο γονιδίωμα και τη μείωση των πιθανών θέσεων στις οποίες μπορεί να χαρτογραφηθεί το αρχικό read. Στο στάδιο της επέκτασης, κάθε seed επεκτείνεται σε μία προσεγγιστική ευθυγράμμιση, επιτρέποντας την ύπαρξη σφαλμάτων ή διαφορών (edits).

Οι σύγχρονοι ευθυγραμμιστές υλοποιούν την επέκταση σε δύο διακριτές φάσεις. Τη φάση Matrix Fill και τη φάση Traceback. Στο στάδιο Matrix Fill οι αλγόριθμοι συμπληρώνουν έναν πίνακα ομοιότητας μεταξύ ενός read και του τμήματος του προτύπου γονιδιώματος, παράγοντας συνήθως μία βαθμολογία ευθυγράμμισης· όσο μεγαλύτερη η ομοιότητα των δύο ακολουθιών, τόσο μεγαλύτερη η βαθμολογία. Στο στάδιο Traceback χρησιμοποιείται η πληροφορία του πίνακα για να αναγνωριστούν η θέση και το είδος των διαφορών και να ανακατασκευαστεί η τελική διαδρομή ευθυγράμμισης.

Στην γονδιωματική επεξεργασία, η διαδικασία επέκτασης (δηλαδή τα Matrix Fill και Traceback) αποτελεί το κύριο σημείο συμφόρησης από άποψη απόδοσης, εξαιτίας των μεγάλων απαιτήσεων σε χρόνο και υπολογιστικούς πόρους. Στην παρούσα διπλωματική εργασία διερευνώνται δύο προσεγγίσεις: ι) η βελτιστοποίηση

της ευθυγράμμισης μέσω επιτάχυνσης, και ιι) η μείωση του πλήθους των ευθυγραμμίσεων αξιοποιώντας χαρακτηριστικά των αλγορίθμων.

Hardware acceleration and optimization of algorithms: Ο όρος επιτάχυνση μέσω υλικού περιλαμβάνει πλήθος λύσεων που στοχεύουν στην επιτάχυνση ενός υπολογιστικού έργου με εξειδικευμένο υλικό. Το στάδιο της ευθυγράμμισης έχει υλοποιηθεί σε συσκευές όπως GPUs, FPGAs και ASICs. Λόγω της παραμετροποίησης σε επίπεδο bit και της επαναπρογραμματιζόμενης φύσης τους, οι FPGAs έχουν αναδειχθεί σε ιδιαίτερα υποσχόμενες πλατφόρμες επιτάχυνσης για γονιδιωματικούς υπολογισμούς και ειδικά για το στάδιο της ευθυγράμμισης. Όταν το επιτρέπει ο αλγόριθμος, μπορούν να παραλληλοποιήσουν εκτενώς το στάδιο Matrix Fill μέσω δομών τύπου systolic arrays, μειώνοντας σημαντικά το υπολογιστικό κόστος. Επιπλέον, η επιτάχυνση μπορεί να γίνει μέσω ευριστικών τεχνικών, όπου επιτρέπεται η παράκαμψη ορισμένων εξαρτήσεων δεδομένων ώστε να αυξηθεί ο παραλληλισμός — όχι όμως χωρίς συμβιβασμούς μεταξύ απόδοσης και ακρίβειας.

Algorithm exploitation and filtering: Ένα ακόμη σημαντικό στάδιο στη γονιδιωματική ροή επεξεργασίας είναι το προ-φιλτράρισμα. Μετά το στάδιο seeding, μπορούν να φιλτραριστούν πιθανά ζεύγη reads- προτύπου γονιδιώματος, ώστε ζεύγη που δεν εμφανίζουν επαρκή ομοιότητα να απορρίπτονται νωρίς, εξοικονομώντας χρόνο και πόρους. Πολλοί αλγόριθμοι προ-φιλτραρίσματος έχουν αποδείξει ότι μειώνουν σημαντικά τον χρόνο ευθυγράμμισης. Ο SneakySnake είναι ένας αλγόριθμος προ-φιλτραρίσματος τελευταίας γενιάς, που επιτυγχάνει υψηλή ακρίβεια ευθυγράμμισης χρησιμοποιώντας μια αποδοτική τεχνική για την απόρριψη μη όμοιων επεκτάσεων. Τέτοιοι αλγόριθμοι παρέχουν επίσης πληροφορίες για την κατανομή των διαφορών σε ένα dataset, γεγονός που μπορεί να αξιοποιηθεί από αλγορίθμους ευθυγράμμισης. Για παράδειγμα, αλγόριθμοι όπως ο Banded-Smith-Waterman αποδίδουν καλύτερα όταν είναι γνωστό εκ των προτέρων το πλήθος των διαφορών, καθώς μπορούν να περιορίσουν τον χώρο αναζήτησης σε ένα στατικό εύρος (band), μειώνοντας σημαντικά την υπολογιστική πολυπλοκότητα.

Συχνά, τεχνικές επιτάχυνσης εφαρμόζονται σε γενικά datasets οδηγώντας σε μη ιδανικές λύσεις, καθώς το προ-φιλτράρισμα είναι διαδικασία εξαρτώμενη από τα δεδομένα και όσο πιο προσαρμοσμένη είναι στο dataset, τόσο καλύτερα αποτελέσματα παράγει.

Κύριος στόχος της παρούσας διπλωματικής είναι η διερεύνηση του πώς το προ-φιλτράρισμα επηρεάζει επιταχυντές ευθυγράμμισης βασισμένους σε υλικό. Η ανάλυση υπαρχόντων ευθυγραμμιστών και datasets δείχνει ότι η κατανομή των σφαλμάτων δεν είναι ομοιόμορφη, καθώς οι περισσότερες ευθυγραμμίσεις έχουν μικρό αριθμό σφαλμάτων. Τέτοιες περιπτώσεις μπορούν να επιλυθούν με λιγότερους πόρους και πράξεις σε σχέση με το τυπικό σενάριο.[3]

Η παρούσα εργασία αξιοποιεί pre-filters βασισμένα σε κατώφλια διαφορών ώστε να βελτιστοποιήσει την ευθυγράμμιση σε FPGAs σε δύο επίπεδα:

- Σε αρχιτεκτονικό επίπεδο, με τον σχεδιασμό μονάδων ευθυγράμμισης προσαρμοσμένων σε συγκεκριμένα κατώφλια διαφορών, που είναι αποδοτικές ως προς πόρους και απόδοση.
- Σε συστημικό επίπεδο, με τη δημιουργία μίας αρχιτεκτονικής πολλαπλών επιταχυντών, η οποία βελτιστοποιεί τη ροή των δεδομένων κατανέμοντας reads στους κατάλληλους ειδικευμένους επιταχυντές.

Η μεθοδολογία ξεκινά με την ανάλυση των δεδομένων — πραγματικών και προσομοιωμένων — ώστε να εξεταστεί η κατανομή των σφαλμάτων. Με βάση τα ευρήματα, επιλέγονται αλγόριθμοι που μπορούν να εκμεταλλευτούν τα κατώφλια διαφορών για να μειώσουν τους υπολογιστικούς και χωρικούς πόρους, και υλοποιούνται με High-Level Synthesis (HLS). Πολλαπλές υλοποιήσεις επιταχυντών συντίθενται για διαφορετικά κατώφλια διαφορών και δημιουργείται μία «δεξαμενή επιταχυντών» με διαφορετικές απαιτήσεις πόρων και καθυστερήσεις.

Τέλος, δημιουργείται ένα σύστημα υψηλής απόδοσης με πολλαπλούς επιταχυντές, όπου κάθε επιταχυντής επεξεργάζεται ρεαδς που αντιστοιχούν στο εύρος διαφορών του. Ένα script εξισορρόπησης, χρησιμοποιώντας δεδομένα από εκθέσεις σύνθεσης και στατιστικά του dataset, καθορίζει τον βέλτιστο συνδυασμό επιταχυντών ώστε να ελαχιστοποιηθεί ο συνολικός χρόνος ευθυγράμμισης με δεδομένους τους πόρους του FPGA.

Θεωρητικό Υπόβαθρο

Γονιδιωματική και η ροή της γονιδιωματικής επεξεργασίας

Η γονιδιωματική είναι η μελέτη του συνολικού περιεχομένου DNA ενός οργανισμού, που περιλαμβάνει όχι μόνο τη νουκλεοτιδική αλληλουχία αλλά και το πώς αυτή μεταβάλλεται μεταξύ κυττάρων, ατόμων, πληθυσμών και στον χρόνο. Επειδή τα γονιδιώματα κωδικοποιούν τις οδηγίες για τη λειτουργία των κυττάρων και την κληρονομικότητα, η γενωμική στηρίζει τη σύγχρονη βιολογία και ιατρική: αποκαλύπτει τη γενετική βάση σπάνιων και κοινών νοσημάτων, παρακολουθεί την εξέλιξη και τα ξεσπάσματα παθογόνων οργανισμών, ενημερώνει τη διάγνωση και θεραπεία του καρκίνου, καθοδηγεί τη βελτίωση ποικιλιών και τη διατήρηση στη γεωργία και την οικολογία, και επιτρέπει βασικές ανακαλύψεις για τη ρύθμιση γονιδίων, την ανάπτυξη και την εξέλιξη. Ο κλάδος έχει μετασχηματιστεί από τεχνολογίες αλληλούχησης, οι οποίες μετέτρεψαν το DNA σε ψηφιακή μορφή που μπορεί να μετρηθεί και να αναλυθεί υπολογιστικά. [4][5][6]

Μία τυπική διαδικασία επεξεργασίας γονιδιώματος ξεκινά με την αλληλούχηση, όπου τα μόρια DNA τεμαχίζονται, προετοιμάζονται σε βιβλιοθήκες και διαβάζονται από όργανα που παράγουν εκατομμύρια έως δισεκατομμύρια ίχνη σήματος. Οι πλατφόρμες μικρού μήκους αλληλουχιών (π.χ. sequencing-by-synthesis)

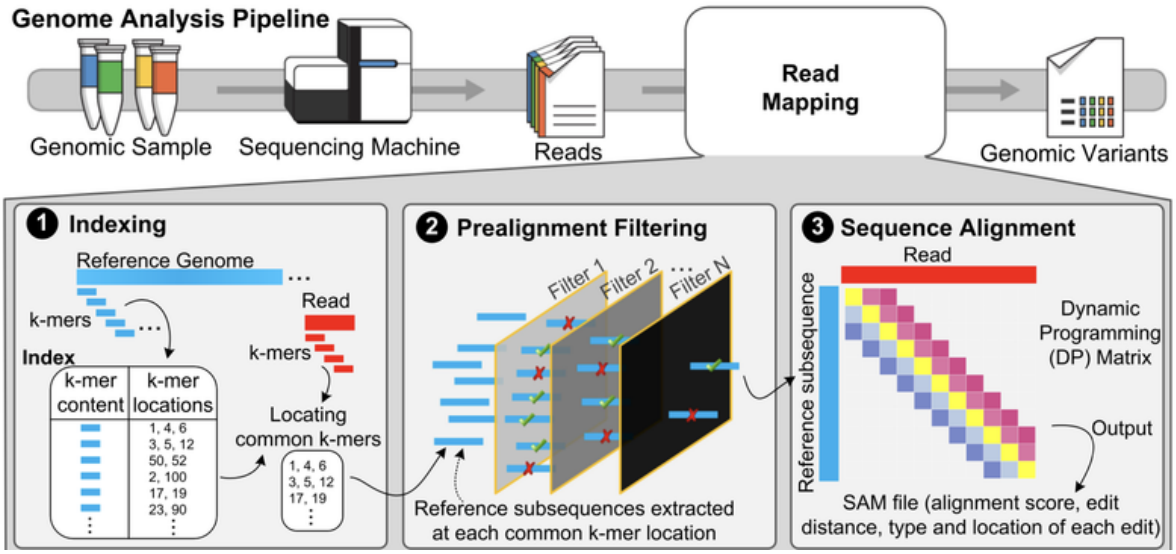


Figure 1: Τυπική επεξεργασία ανάλυσης DNA [7]

αποδίδουν υψηλής ακρίβειας αναγνώσεις μήκους περίπου 75–300 bp, ενώ οι πλατφόρμες μεγάλου μήκους (π.χ. nanopore και single-molecule real-time (SMRT) sequencing) παράγουν αναγνώσεις σε κλίμακα kilobase–megabase που επιλύουν επαναλήψεις και δομικές παραλλαγές με τίμημα υψηλότερα ακατέργαστα ποσοστά σφάλματος. Επιλογές πειραματικού σχεδιασμού — βάθος κάλυψης, μήκος ανάγνωσης και paired-end έναντι single-end — καθορίζουν την ισχύ και τις μεροληψίες των επόμενων σταδίων. [7]

Το basecalling μετατρέπει τα ακατέργαστα σήματα του οργάνου σε νουκλεοτιδικές ακολουθίες με βαθμολογίες ποιότητας ανά βάση. Πιο συγκεκριμένα, τα σήματα μεταφράζονται σε βάσεις: A (Adenine), T (Thymine), C (Cytosine), G (Guanine). Συστήματα μικρού μήκους αντιστοιχούν πρότυπα φθορισμού σε βάσεις· τα nanopore συστήματα αντιστοιχούν ίχνη ιοντικού ρεύματος σε βάσεις, συνήθως με deep neural networks. Η ακρίβεια του basecalling και η βαθμονόμηση των Phred quality scores επηρεάζουν άμεσα την ευαισθησία και τα ψευδώς θετικά στα επόμενα στάδια, γι' αυτό οι ροές συχνά διατηρούν τόσο τις ακολουθίες (FASTQ) όσο και τις βαθμολογίες για μεταγενέστερο φιλτράρισμα και μοντελοποίηση. [7]

Το seeding εντοπίζει αποδοτικά υποψήφιες γονιδιωματικές θέσεις για κάθε ανάγνωση ταιριάζοντας σύντομες υποσυμβολοσειρές. Συνήθεις στρατηγικές περιλαμβάνουν σταθερού μήκους k-mers, spaced seeds που ανέχονται ασυμφωνίες, minimizers και syncmers που υπο-δειγματοληπτούν αντιπροσωπευτικά k-mers για μείωση της πλεονάζουσας πληροφορίας, και μεθόδους βασισμένες σε ευρετήρια όπως FM-index/BWT ή πίνακες κατακερματισμού (hash tables) πάνω στο πρότυπο. Η αποτελεσματική επιλογή seeds εξισορροπεί ευαισθησία (να μη χαθούν αληθείς θέσεις) με ταχύτητα/μνήμη και συχνά παράγει πολλαπλά υποψήφια πλήγματα ανά ανάγνωση για μεταγενέστερο περιορισμό.[8][9][10]

Το filtering περιορίζει τους υποψηφίους πριν από την ακριβή ευθυγράμμιση. Τυπικά φίλτρα αφαιρούν αναγνώσεις χαμηλής ποιότητας καθώς και ιδιαίτερα ανόμοια ζευγάρια ακολουθιών.[11][12]

Το alignment βελτιώνει τη χαρτογράφηση σε ανάλυση νουκλεοτιδίου υπολογίζοντας μια βέλτιστη ή σχεδόν βέλτιστη αντιστοίχιση μεταξύ κάθε ανάγνωσης και της υποψήφιας περιοχής του προτύπου, υπό ένα μοντέλο βαθμολόγησης για ταυτίσεις, ασυμφωνίες και κενά. Αλγόριθμοι δυναμικού προγραμματισμού όπως οι Smith–Waterman (τοπική) και Needleman–Wunsch (ολική) παρέχουν ακριβείς λύσεις· banded, affine-gap και wavefront διατυπώσεις επιταχύνουν τον υπολογισμό· τα πλαίσια seed-and-extend περιορίζουν το δυναμικό πρόγραμμα στις γειτονιές που υποδεικνύει το seeding. Η επιλογή τρόπου ευθυγράμμισης (τοπική έναντι end-to-end), οι ποινές κενού και το εύρος ζώνης (bandwidth) επηρεάζουν την ανίχνευση indels, τον χειρισμό soft clipping και την απόδοση σε επαναλαμβανόμενες ή θορυβώδεις περιοχές.[10]

Το variant calling ανιχνεύει διαφορές μεταξύ δείγματος και προτύπου γονιδιώματος συγκεντρώνοντας ευθυγραμμισμένες ενδείξεις από πολλές αναγνώσεις. Οι germline callers μοντελοποιούν διπλοειδείς (ή πολυπλοειδείς) γονότυπους και εξάγουν SNPs και μικρά indels με πιθανότητες γονοτύπου και δείκτες ποιότητας· οι somatic callers συγκρίνουν ζεύγη tumor–normal ή tumor-only για τον εντοπισμό υποκλωνικών παραλλαγών σε περιβάλλον αλλαγών copy-number· τα εργαλεία μεγάλου μήκους επιλύουν επίσης μεγαλύτερα indels και δομικές παραλλαγές (SVs) όπως αναστροφές, μεταθέσεις και αλλαγές αριθμού αντιγράφων. Σύγχρονοι ανιχνευτές ενσωματώνουν base qualities, mapping qualities, προσανατολισμό/ζευγοποίηση αναγνώσεων, local assembly γύρω από ύποπτα σημεία, και μηχανική μάθηση για βαθμονόμηση. Μεταγενέστερα στάδια περιλαμβάνουν joint genotyping, phasing για ανακατασκευή απλοτύπων, σχολιασμό σε βάσεις γονιδίων και πληθυσμών, και στρατηγικές φιλτραρίσματος που ελέγχουν τα ψευδώς θετικά χωρίς να χάνουν αληθείς παραλλαγές σε δύσκολες περιοχές.[13][14]

High Level Synthesis

Το High-Level Synthesis (HLS) αποτελεί μία μεθοδολογία σχεδίασης που μεταφράζει κώδικα υψηλού επιπέδου (π.χ. C/C++, SystemC, OpenCL) σε υλοποιήσιμη περιγραφή υλικού (RTL). Αντί ο σχεδιαστής να γράφει εξ αρχής Verilog/VHDL σε επίπεδο καταχωρητών, περιγράφει τον αλγόριθμο αφηρημένα και το εργαλείο αυτοματοποιεί τον χρονοπρογραμματισμό, την κατανομή/δέσμευση πόρων και τη χαρτογράφηση σε κυκλωματικές δομές. Έτσι μειώνεται ο χρόνος ανάπτυξης και διευκολύνεται η διερεύνηση εναλλακτικών αρχιτεκτονικών.

Η τυπική ροή ξεκινά από την περιγραφή του αλγορίθμου και τον καθορισμό περιορισμών (συχνότητα, καθυστέρηση, ισχύς, χρήση πόρων). Το εργαλείο εφαρμόζει βελτιστοποιήσεις (π.χ. pipelining, unrolling, διαχείριση μνήμης) και παράγει

RTL μαζί με αναφορές εκτίμησης απόδοσης/εμβαδού (latency, initiation interval, LUT/FF/BRAM/DSP). Η επαλήθευση γίνεται συχνά με co-simulation, συγκρίνοντας τη συμπεριφορά του RTL με το μοντέλο αναφοράς λογισμικού.

Directives στο HLS

Σύντομες οδηγίες που εισάγονται στον κώδικα ή στο εργαλείο για να κατευθύνουν τον μετασχηματισμό σε υλικό χωρίς να αλλάζει η λειτουργική συμπεριφορά του αλγορίθμου. Ελέγχουν π.χ. τον παραλληλισμό βρόχων, το pipelining, την κοινοχρησία/διπλασιασμό πόρων, και τη διάταξη μνημών. Οι κυριότερες που θα χρησιμοποιηθούν στην παρούσα διπλωματική είναι:

- Loop unroll: Αντιγράφει τα σώματα των επαναλήψεων (μερικώς ή πλήρως) ώστε πολλές επαναλήψεις να εκτελούνται παράλληλα. Κερδίζει χρόνο εκτέλεσης με κόστος περισσότερους πόρους.
- Pipeline: Διασπά μια υπολογιστική ροή σε στάδια που επεξεργάζονται ταυτόχρονα διαφορετικά δεδομένα. Κρίσιμη μετρική το Initiation Interval (II)—όσο μικρότερο, τόσο μεγαλύτερη ροή (throughput).
- Array partition: «Σπάει» έναν πίνακα σε πολλαπλές ανεξάρτητες μνήμη ες για να αυξηθούν οι ταυτόχρονες προσπελάσεις.

Μεθοδολογία

Κύριος σκοπός της διπλωματικής είναι η διερεύνηση της επίδρασης των φίλτρων στο βήμα της αντιστοίχησης υλοποιημένο από επιταχυντές υλικού. Μία γονιδιωματική βάση δεδομένων έχει μεγάλο ποικιλομορφία όσον αφορά την κατανομή των δεδομένων της βάσει των αριθμό των σφαλμάτων που περιέχονται σε κάθε στοιχείο της. Συνήθως πολλά δεδομένα έχουν μικρό αριθμό σφαλμάτων και αυτό συχνά σημαίνει ότι μπορεί να βρεθεί μία αντιστοίχηση με λιγότερους πόρους από το βασικό σενάριο. Σε αυτή τη διπλωματική θα χρησιμοποιήσουμε φίλτρα που ξεσκαρτάρουν δεδομένα βάσει ενός κατωφλίου σφάλματος, με σκοπό να κατηγοριοποιήσουμε τα δεδομένα μας βάσει των αριθμό των σφαλμάτων που εμπεριέχονται σε αυτά. Στη συνέχεια θα χρησιμοποιήσουμε αλγορίθμους αντιστοίχησης οι οποίοι μπορούν να επωφεληθούν από τη γνώση για τον αριθμό των σφαλμάτων τους οποίους θα υλοποιήσουμε σε FPGA και θα ειναι παραμετροποιημένοι βάσει των κατωφλίων που χρησιμοποιήσαμε κατά το φιλτράρισμα. Κάθε ομάδα επιταχυντών θα αντιστοιχεί κατάλληλα δεδομένα, δηλαδή δεδομένα που υπάρχουν στην κατηγορία που ταιριάζει με το κατώφλι της.
Θα χρησιμοποιηθεί ένα πραγματικό datasetκαι ένα dataset προσωμοίωσης. Θα χρησιμοποιηθούν δύο αλγόριθμοι αντιστοίχησης. Ο Smith-Waterman ο οποίος είναι ένα κλασσικός αλγόριθμος για αντιστοίχηση συμβολοσειρών και είναι από τους

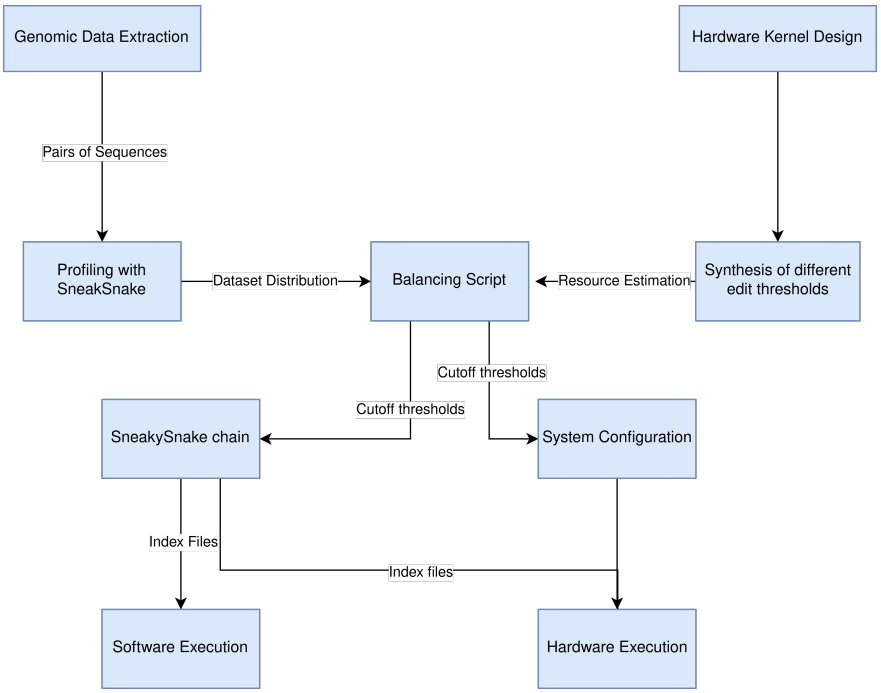


Figure 2: Προτεινόμενη Μεθοδολογία

πρώτους που χρησιμοποιήθηκε στην γονιδιωματική επιστήμη, και ο GenASM ,ο οποίος είναι ένας ιδιαίτερος αλγόριθμος που χρησιμοποιεί κατά κόρων πράξεις σε επίπεδο bitώστε να τρέξει ιδιαίτερα αποδοτικά στο hardware. Ως φίλτρο θα χρησιμοποιήσουμε το SneakySnake ,ένα φίλτρο το οποίο δουλεύει με κάποιο κατώφλι σφάλματος και έχει πάρα πολύ καλά αποτελέσματα όσον αφορά την ακρίβειά του. Η διαδικασία που θα ακολουθηθεί είναι η ακόλουθη. Αφού αναλύσουμε τα δεδομένα μας ως πρός το πλήθος των σφαλμάτων και υλοποιήσουμε την αρχιτεκτονική των επιταχυντών μας μέσω High Level Synthesis θα εισάγουμε τα αποτελέσματα για την κατναομή των σφαλμάτων αλλά και την χρήση πόρων των επιταχυντών σε έναν κώδικα ισορόπησης. Ο κώδικας αυτός θα προσπαθήσει να υπολογίσει την βέλτιστη κατανομή από πλήθος και είδη επιταχυντών , οι οποίοι θα χρησιμοποιηθούν στο τελικό σύστημα. Αφού λάβουμε αυτή την κατανομή θα δημιουργήσουμε μία σειρά από φίλτρα η οποία βάσει των κατωφλίων που προέκυψαν από το πρόγραμμα θα κατηγοριοποιεί τα δεδομένα βάσει τον αριθμώ των σφαλμάτων τους. Επίσης και πάλι βάσει του προγράμματος αυτού θα υλοποιήσουμε την αρχιτεκτονική πολλαπλών επεξεργαστών που προτείνουμε. Τέλος, σε επίπεδο προσωμοίωσης, θα πάρουμε μετρήσεις για τους χρόνους που χρειάζονται να εκτελεστούν τόσο για την αρχιτεκτονική μας, όσο και για το λογισμικό αλλά και για επιταχυντές που δεν χρησιμοποιούν την αρχιτεκτονική μας. Ένα σχηματικό διάγραμμα φαίνεται παρακάτω.

SneakySnake

Ο αλγόριθμος SneakySnake [15] αντλεί έμπνευση από το πρόβλημα SNR routing σε VLSI κυκλώματα. Το πρόβλημα SNR επιλύεται με χρήση ενός neighborhood map και ο αλγόριθμος προσπάθει να μεταβεί από την αρχή του πίνακα προς το τέλος του, συναντώντας όσο το δυνατόν λιγότερα εμπόδια, όπως φαίνεται στο σχήμα.

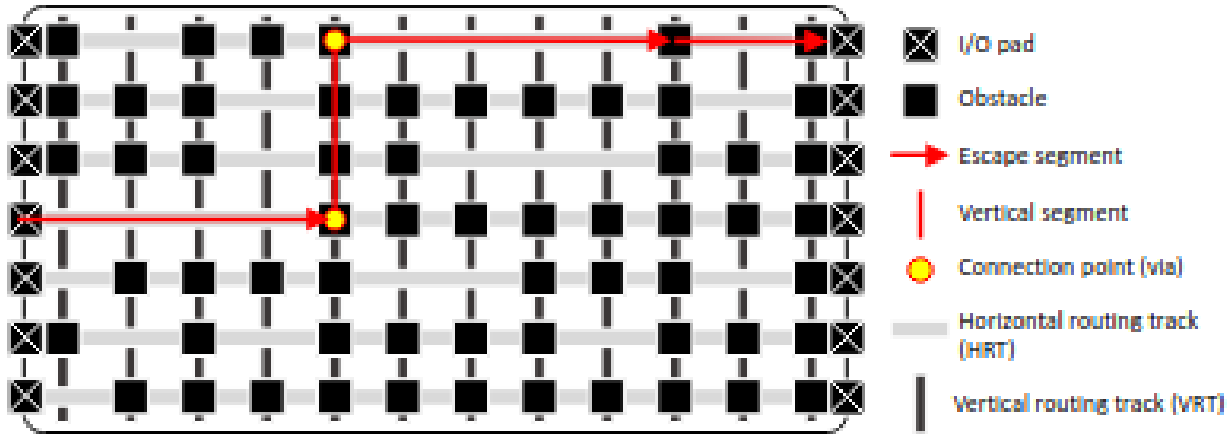


Figure 3: Το πρόβλημα SNR στα VLSI [16]

Στο SneakySnake αυτό το είδος πίνακα υπολογίζεται ως εξής. Μετά την ολοκλήρωση του seeding στην διαδικασία επεξεργασίας του γονιδιώματος, τα δεδομένα διαχωρίζονται σε ζεύγη ακολουθιών read–reference. Τα reads είναι οι ακολουθίες που προκύπτουν από τη μηχανή αλληλούχησης και οι reference ακολουθίες είναι τμήματα του προτύπου γονιδιώματος που μπορεί να αποτελούν καλούς υποψήφιους τόπους ευθυγράμμισης. Το SneakySnake λειτουργεί ιδανικά όταν οι δύο ακολουθίες έχουν το ίδιο μήκος, καθώς διαφορές στο μήκος μπορούν να ερμηνευτούν ως edits.

Για την κατασκευή του neighborhood map, η ακολουθία read μετατοπίζεται και συγκρίνεται με την reference. Ο αριθμός μετατοπίσεων είναι ίσος με το κατώφλι διαφορών που ορίζει ο σχεδιαστής προς τα αριστερά και προς τα δεξιά. Έτσι, αν το κατώφλι είναι E, ο πίνακας θα έχει 2*E+1 οριζόντιες γραμμές (μία χωρίς μετατόπιση, E για αριστερή μετατόπιση και E για δεξιά). Η σύγκριση γίνεται σε επίπεδο νουκλεοτιδικών βάσεων (A, C, G, T) και ο πίνακας γεμίζει με 0 όταν οι ακολουθίες ταιριάζουν και με 1 όταν δεν ταιριάζουν. Τα 1 αποτελούν τα «εμπόδια». Ο τελικός πίνακας μοιάζει έντονα με τον αντίστοιχο του SNR. Ο πίνακας έχει δύο πολύ σημαντικές ιδιότητες: (α) κάθε κελί του μπορεί να υπολογιστεί με παράλληλο τρόπο· (β) όλες οι οριζόντιες γραμμές μπορούν να επεξεργαστούν χωρίς εξαρτήσεις δεδομένων. Αυτά τα χαρακτηριστικά επιτρέπουν εκτεταμένο παραλληλισμό σε FPGAs, GPUs και ακόμη και σε πολυνηματικές CPUs αρχιτεκτονικές.

Αφού ολοκλρωθεί ο πίνακας ξεκινά ο κύριος αλγόριθμος, κατά τον οποίο ελέγχουμε

για συνεχόμενες συμβολοσειρές απο μηδενικά. Αφού βρούμε τα περισσότερα συνεχόμενα 0 σε μία γραμμή του πίνακα και τελικά βρούμε ένα εμπόδιο (ένα 1 δηλαδή) τότε σταματάμε προσμετράμε και προσπερνάμε το εμπόδιο και συνεχίζουμε από εκείνο το σημείο του πίνακα πάλι ψάχνοντας την μεγαλύτερη συνεχομένη συμβολοσειρά απο 0. Ο αλγόριθμος τελειώνει ή όταν φτάσουμε στην άλλη άκρη του πίνακα, το οποίο σημαίνει ότι το ζευγάρι έχει τον απαραίτητο αριθμό σφαλμάτων ή λιγότερο, ή όταν εντοπίσουμε ένα πλήθος εμποδίων που ξεπερνά το όριο, όπου τότε ο αλγόριθμος τερματίζει απορρίπτοντας το ζευγάρι δεδομένων.
Ο συγκεκριμένος αλγόριθμος εγκυάται ότι δεν απορρίπτει ποτέ λανθασμένα κάποιο ζευγάρι. Δεν μπορεί όμως να εγκυηθεί ότι δεν θα αποδεχτεί λανθασμένα κάποιο ζεύγος δεδομένων. Αυτό θα οδηγήσει σε μία μικρή πτώση της ακρίβειας της αρχιτεκτονικής μας όπως θα αναφερθεί στη συνέχεια.

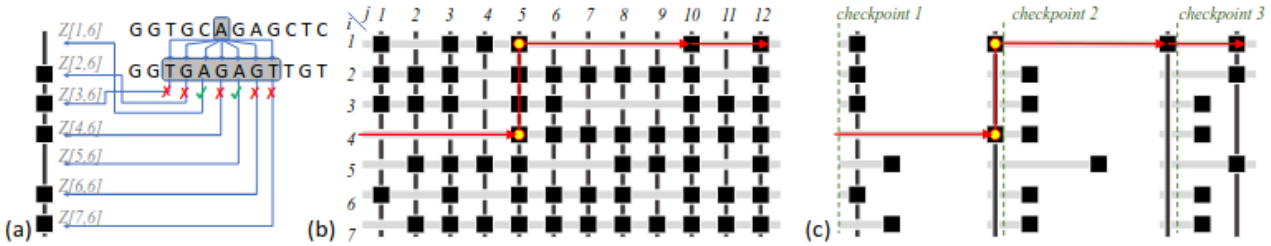


Figure 4: Ο αλγόριθμος SneakySnake. (α) Ο πίνακας υπολογίζεται συγκρίνοντας το μετατοπισμένο read με το reference. (β) Ο αλγόριθμος βρίσκει τα περισσότερα 0 μέχρι να συναντήσει εμπόδιο και επαναλαμβάνει. (ς) Εντοπίζεται διαδρομή με edits μικρότερα ή ίσα του κατωφλίου (E=3), οπότε το ζεύγος προχωρά στην ευθυγράμμιση [16].

Ανάλυση δεδομένων

Από την ανάλυση των δεδομένων με τη βοήθεια του παραπάνω φίλτρου προκύπτουν οι κατανομές σφαλμάτων πρώτα για το προσομοιωμένο και μετά για το πραγματικό dataset.

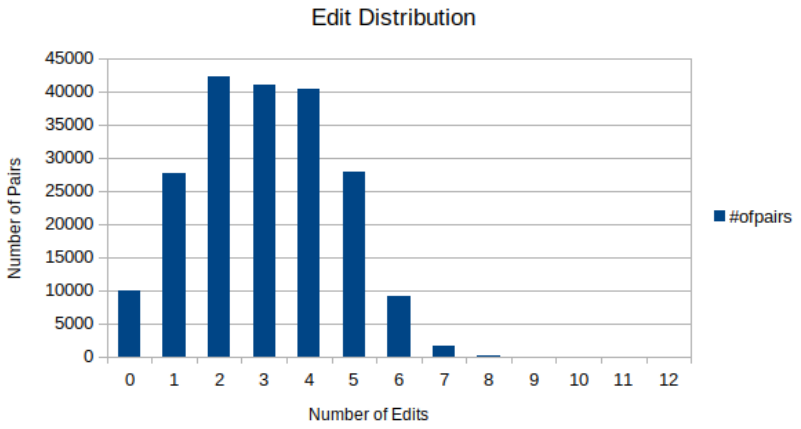


Figure 5: Κατανομή σφαλμάτων προσομοιωμένου dataset

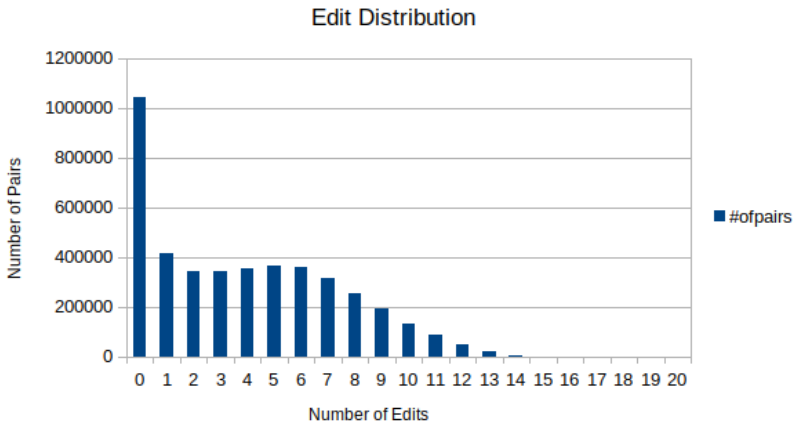


Figure 6: Κατανομή σφαλμάτων πραγματικού dataset

Αλγόριθμοι Αντιστοίχησης και υλοποίηση τους στο FPGA με χρήση HLS

Smith Waterman

Ο Smith Waterman είναι ένας αλγόριθμος δυναμικού προγραμματισμού ο οποίος επιχειρεί να αντιστοιχήσει δύο ακολουθίες με τη βοήθεια ενός πίνακα ομοιότητας. Δρα σε δύο φάσεις. Την φάση υπολογισμού του πίνακα και την φάση Traceback . Στην πρώτη φάση βάσει της παρακάτω μαθηματικής σχέσης:

```
    H[i][j] = max(
    0,
    H[i-1][j-1] + match_or_mismatch(P[i-1], T[j-1]),  // diagonal (match/
    substitution)
    H[i-1][j]   - gap_penalty,                         // deletion
    H[i][j-1]   - gap_penalty                          // insertion
)
```

υπολογίζονται τα κελιά του πίνακα. Κάθε κελί του πίνακα έχει εξάρτηση από το κελί στα αριστερά του, το κελί από πάνω και το κελί το οποίο βρίσκετα αριστερά πάνω απο αυτό. Αυτό επιτρέπει σε κάθε αντιδιαγώνιο να μπορεί να υπολογιστεί παράλληλα, αφού τα κελιά αυτής δεν έχουν μεταξύ τους εξαρτήσεις.

Κατά την πρώτη αυτή φάση αποθηκεύεται και η πληροφορία για το ποιο όρισμα του max operation που συμβαίνει σε κάθε κελί έλαβε μέρος. Αυτή τη πληροφορία θα λάβει ο αλγόριθος του traceback για να ανακτασκευάσει την τελική αντιστοίχηση.

Για τον συγκεκριμένο αλγόριθμο ισχύει ότι αν ένα ζεύγος δεδομένων έχει μέχρι E σφάλματα τότε μπορούμε να περιορίσουμε τον πίνακα ομοιότητας στις διαγωνίου του πίνακα που απέχουν E από την κύρια διαγώνιο. Έτσι μπορούμε να εκμεταλευτούμε την πληροφορία για το πλήθος των σφαλμάτων την οποία παίρνουμε μέσω του φίλτρου για να μειώσουμε τους συνολικούς πόρους που χρειάζεται ο

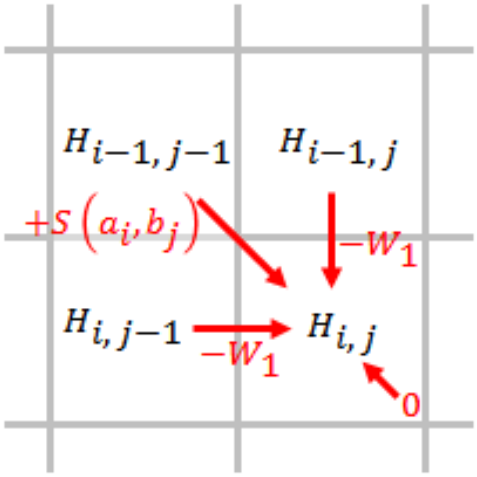


Figure 7: Εξαρτήσεις Smith Waterman

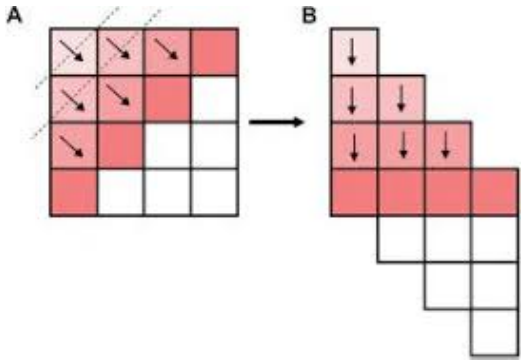


Figure 8: Διαγονιωποίηση του Smith Waterman

επιταχυντής μας. Αυτή η παραλλαγή του αλγορίθμου ονομάζεται Banded Smith Waterman.

A

		A	T	G	T	C	A
	0	0	0	0	0	0	0
A	0	1	0	0	0	0	1
A	0	1	0	0	0	0	1
G	0	0	0	1	0	0	0
T	0	0	1	0	2	1	0
A	0	1	0	0	1	1	2
A	0	1	0	0	0	0	2

B

allowed number of gaps

		A	T	G	T	C	A
	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0
A	0	1	0	0	0	0	0
G	0	0	0	1	0	0	0
T	0	0	0	0	2	1	0
A	0	0	0	0	1	1	2
A	0	0	0	0	0	0	2

Figure 9: Μείωση των πόρων για γνωστό αριθμό σφαλμάτων

Παρακάτω φαίνεται μία σχηματική αναπαράσταση της αρχιτεκτονικής που υλοποίησαμε στο FPGA. Στην αρχή γίνεται μία αποκωδικοποίηση των δεδομένων σε έναν κύκλο μηχανής μέσω του Unroll directive. Στη συνέχεια με τη βοήθεια του pipeline directive πετυχαίνουμε ένα systolic array design ώστε να παραλληλοποι-

ηθεί ο υπολογισμός των αντι-διαγωνίων. Στην εικόνα φάινονται τα processing elemenets μίας αντι-διαγωνίου τα οποία γράφουν παράλληλα στην BRAM. Για να γίνει αυτό πρέπει να χρησιμοποιήσουμε το array partition directive ώστε να χωρίσουμε τον πίνακα σε διαφορετικά κομμάτια τα οποία μπορούν να προσπελαστούν ταυτόχρονα. Στην συνέχεια το traceback module διαβάζει την πληροφορία από τον πίνακα και κατασκευάζει την τελική αντιστοίχηση. Αυτό επιτυγχάνεται με τη χρήση του pipeline directive το οποίο επικαλύπτει τις πράξεις τοης προσπέλασης σε πίνακα και την πράξη λήψης αποφάσεων για την κατασκευή της αντιστοίχησης.

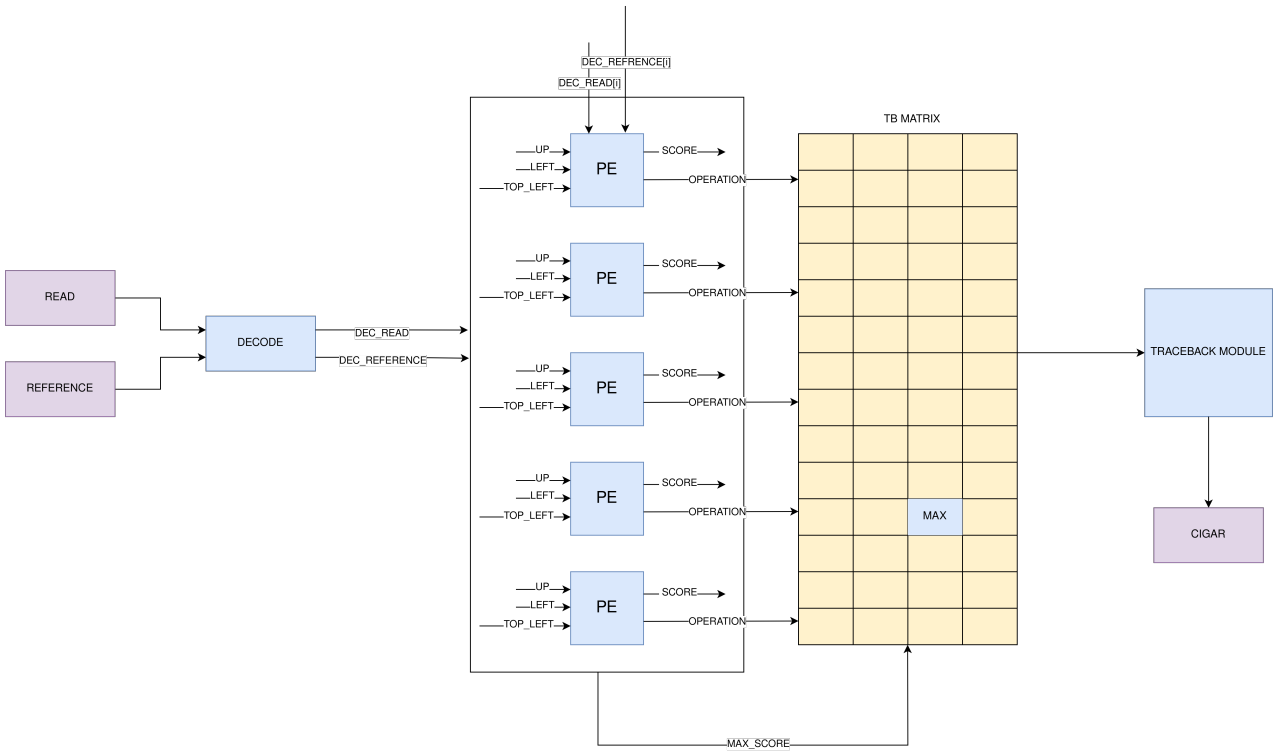


Figure 10: Αρχιτεκτονική Banded Smith Waterman

GenASM

Ο GenASMείναι ένας αλγόριθμος ο οποίος χρησιμοποιεί πράξεις σε επίπεδο bit αποθηκεύοντας διανύσματα αντί για σκορ, όπως οι συμβατικοί αλγόριθμοι αντιστοίχισης. Χρησιμοποιεί μάσκες και διανύσματα κατάστασης για να κωδικοποιήσει πληροφορίες τις οποίες αποθηκεύει σε έναν μεγάλο πίνακα τον οποίον μία traceback διαδικασία η οποία διαβάζοντας τον αποθηκευμένο πίνακα κατσκευάζει την τελική αντιστοίχηση. Ο πίνακας έχει 4 διαστάσεις, μία μεγέθους 4 (ένα για κάθε είδος πράξης), μία μεγέθους όσα τα σφάλματα συν 1, μία μεγέθους όσο το μήκος του read και μία μεγέθους όσο το μήκος του reference. Το κύριο κομμάτι του αλγορίθμου είναι μία επαναληπτική δομή η οποία τρέχει ανάποδα στο reference sequence.Σε κάθε βήμα αυτής της επανάληψης, υπολογίζονται νέα διανύσματα

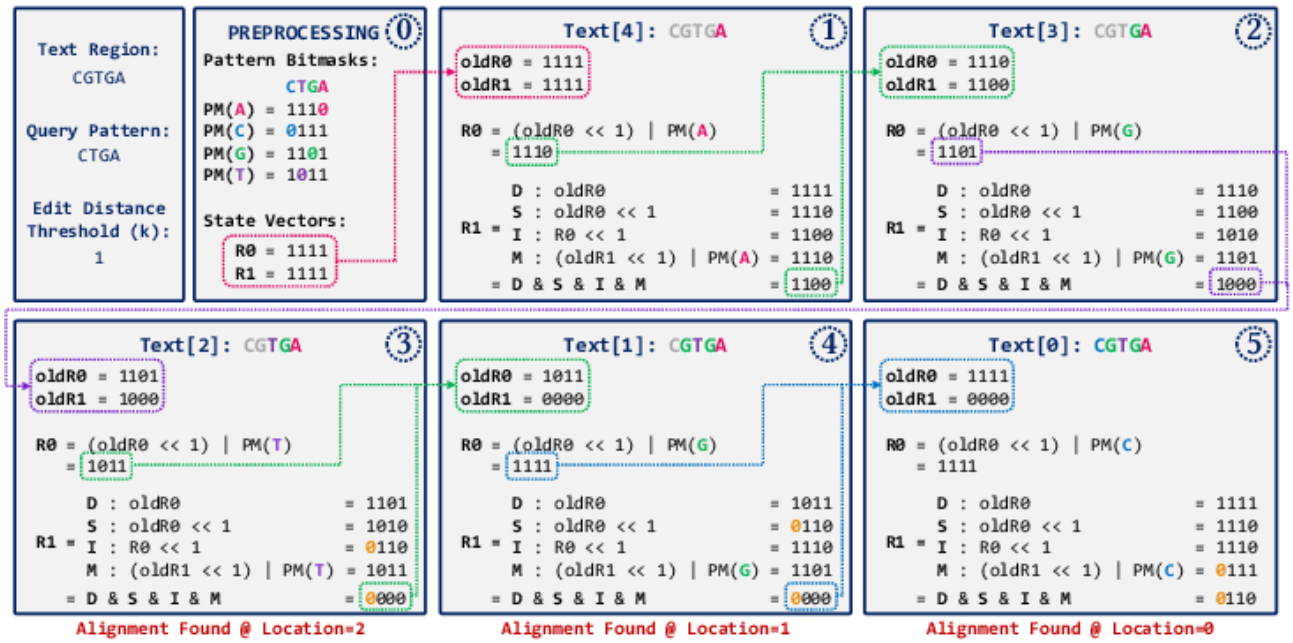


Figure 11: Παράδειγμα της κύριας δομής επανάληψης του GenASM

κατάστασης και αποθηκεύονται 4 υπο-διανύσματα (ένα για κάθε πράξη) στον πίνακα που θα χρησιμοποιήσει το traceback . Οι πράξεις που συμβαίνουν σε κάθε επανάληψη είναι κατά κύριο λόγο πράξεις ολίσθησης , λογικά ΚΑΙ και λογικά Ή, όπως φαίνεται και στην εικόνα. Όταν ο αλγόριθμος βρεί κάποιο διάνυσμα κατάστασης με το MSB να βρίσκεται στην κατάσταση 0, τότε ο αλγόριθμος αναφέρει ότι βρήκε κάποια αντιστοιχήσης με E σφάλαμτα , όπου E ο αύξων αριθμός του διανύσματος κατάστασης. Το traceback στη συνέχεια παρακολουθεί αυτό το 0 στα διάφορα κελιά του πίνακα αποθήκυεσης για να κατασκευάσει την τελική αντιστοίχηση.
Ο συγκεκριμέος αλγόριθμος λειτουργεί εξορισμού με κάποιο κατώφλι σφαλμάτων. Αυτό φαίνεται και στην διάσταση του πίνακα που χρησιμοποιείται στο Traceback.Η μία του διάσταση είναι ίση με τον αριθμό των σφαλμάτων συν 1 οποτε αν γνωρίζουμε το πόσα σφάλματα έχει ένα ζευγάρι μπορούμε να περιορίσουμε την διάσταση αυτού του πίνακα κερδίζοντας πόρους.

Παρακάτω φαίνεται μία σχηματική αναπαράσταση της αρχιτεκτονικής που υλοποιήθηκε στο HLS.Στην αρχή παράγονται τα bitmasks που είναι απαραίτητα για την διεκπεραίωση του αλγορίθμου. Ο υπολογισμός τους γίνεται σε έναν κύκλο αφού δεν υπάρχουν εξαρτήσεις με τη βοήθεια του unroll directive.Στη συνέχεια μία παράλληλη συστοιχία από μικρά επεξεργαστικά τμήματα υπολογίζει παράλληλα κάθε κύκλο τα επόμενα διανύσματα κατάστασης. Επειδή ο υπολογισμός τους απαιτεί γνώση και για τα τωρινά αλλά και για τα διανύσματα του προηγούμενου βήματος, χρησιμοποιήθηκε το pipeline directive ώστε να επικαλυφθεί η ανάγνωση των παλιών με την δημιουργία των καινούριων διανυσμάτων. Επίσης χρησιμοποιήθηκε το un-

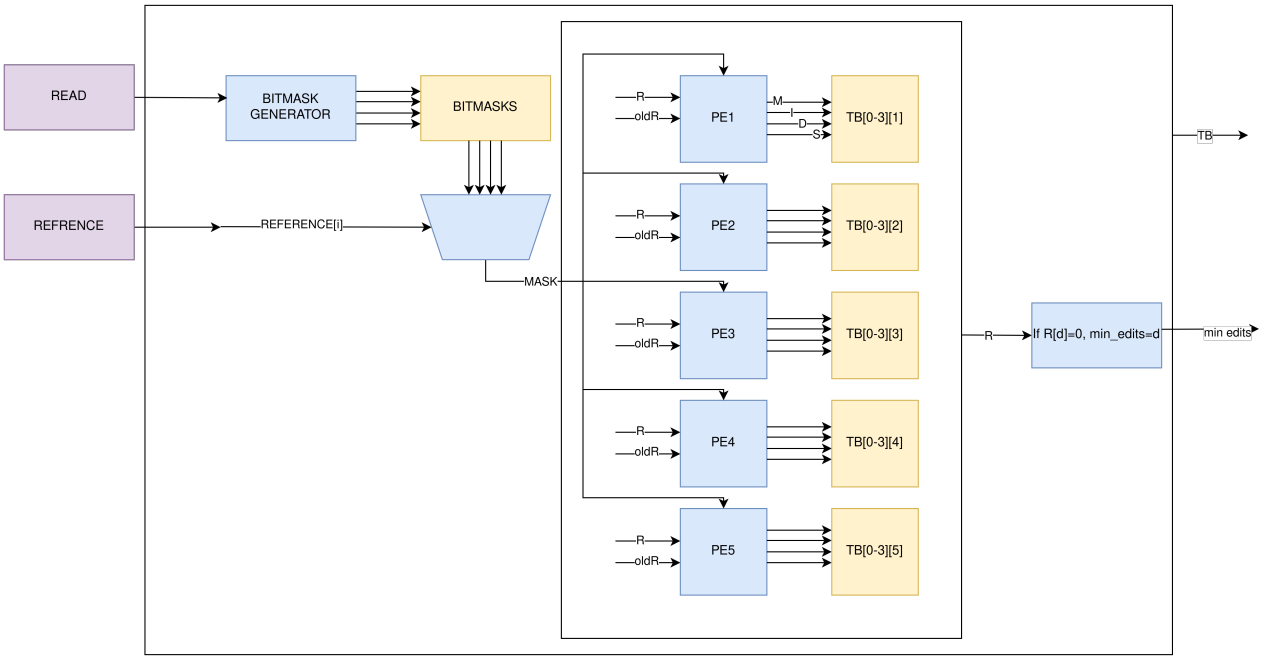


Figure 12: Σχηματική αναπαράσταση της αρχιτεκτονικής του GenASM

roll directive για τον ταυτόχρονο υπολογισμό όλων των υπο-διανυσμάτων. Για να επιτευχθεί η προαναφερθείσα παραλληλία ήταν απαραίτητο να χρησιμοποιηθεί το array partition directive τόσο στην διάσταση των σφαλμάτων , όσο και στην διάσταση των πράξεων ώστε κάθε PE να μπορεί να γράφει κάθε υπο-διάνυσμα παράλληλα. Τέλος το traceback module διαβάζει τον πίνακα και κατασκευάζει την τελική αντιστοίχηση. Αυτό για να γίνει σε ένα κύκλο ανά απόφαση έπρεπε να χρησιμοποιηθεί το pipeline directive ώστε να επικαλυφθεί η προσπέλαση στον πίνακα με την λήψη της απόφασης.

Προτεινόμενη Αρχιτεκτονική

Στην εικόνα φαίνεται η προτεινόμενη αρχιτεκτονική, κατά την οποία ένα σετ δεδομένων αφού φιλτραριστεί και κατηγοριοποιηθεί σύμφωνα με το πλήθος των σφαλμάτων του σε λογισμικό, μεταφέρεται στην πλακέτα μέσω streaming FIFOs οι οποίες καταλήγουν σε επιταχυντές αντίστοιχου μεγέθους. Εκεί πραγματοποιείται παράλληλα από κάθε εεπιταχυντή η αντιστοίχιση των δεδομένων.

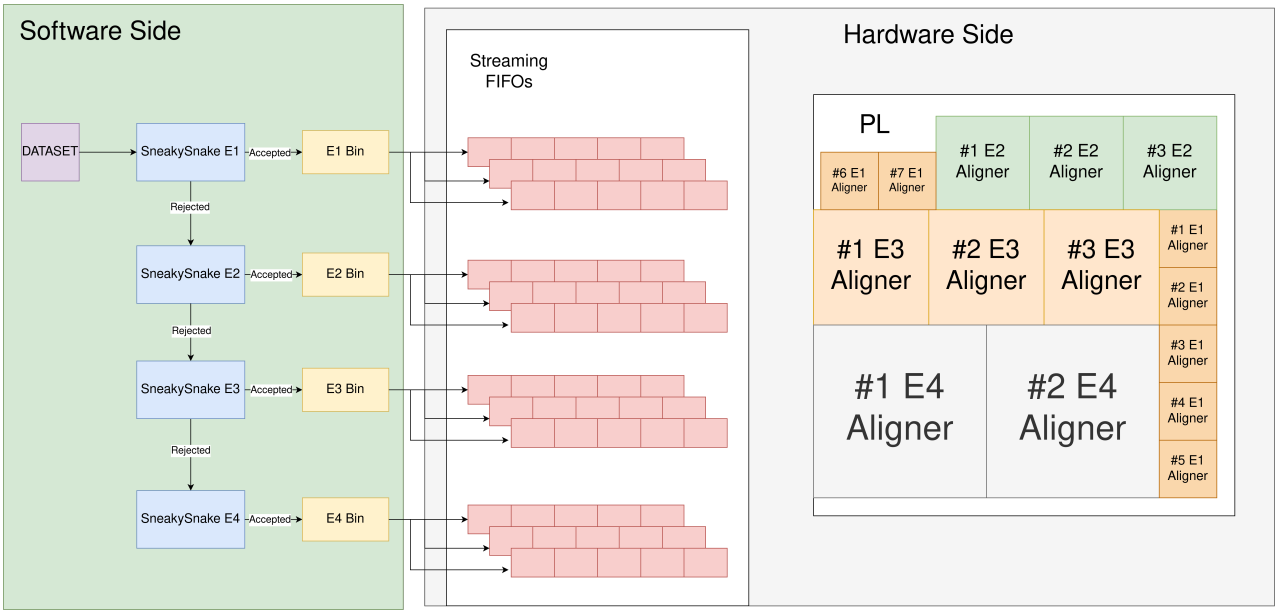


Figure 13: Προτεινόμενη Αρχιτεκτονική

Πειραματικά Αποτελέσματα

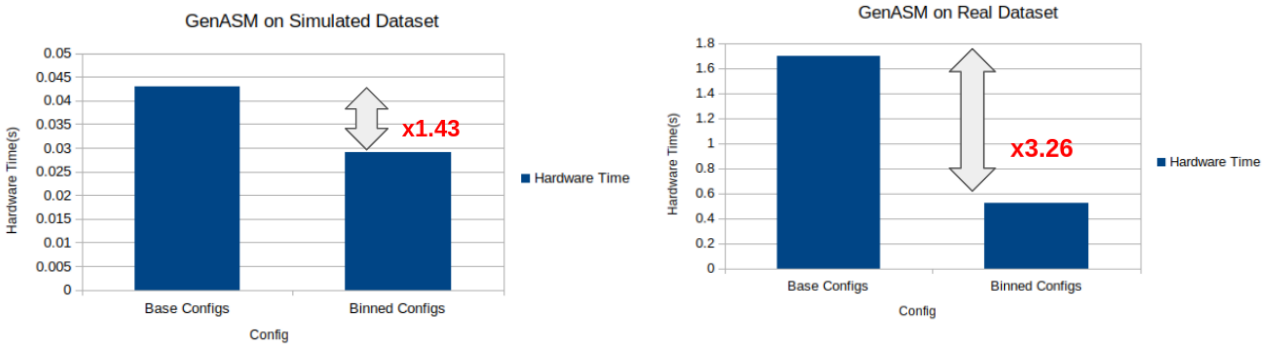
Διεξήχθησαν 4 πειράματα. Ένα για κάθε αλγόριθμο σε καθένα από τα δύο dataset.Αφού πρώτα δόθηκε η χρήση πόρων του κάθε αλγορίθμου, οι συνολικοί πόροι της πλακέτας και η κατανομές σφαλμάτων των δεδομένων στο πρόγραμμα ισσορόπησης, προέκυψαν οι αρχιτεκτονικές που υλοποιήθηκαν στο HLS Οι συγκρίσεις όσον αφορά τον χρόνο εκτέλεσης γίνανε μεταξύ της αρχιτεκτονικής μας και λογισμικού και μεταξύ της αρχιτεκτονικής μας και επιταχυντών χωρίς να έχει προ-υπάρξει φιλτράρισμα. Επίσης για το λογισμικό υπάρχουν δύο εκδόσεις. Η πρώτη αφορά λογισμικό που έχει παραμετροποιηθεί βάσει του φιλτραρίσματος και το δεύτερο είναι λογισμικό που λειτουργεί χωρίς φιλτράρισμα. Παρακάτω φαίνονται συγκεντρωτικά τα αποτελέσματα.

Σύγκριση μεταξύ βασικού επιταχυντή με την προτεινόμενη αρχιτεκτονική

Παρατηρούμε σημαντική επιτάχυνση σε σχέση με το βασικό Hardware.

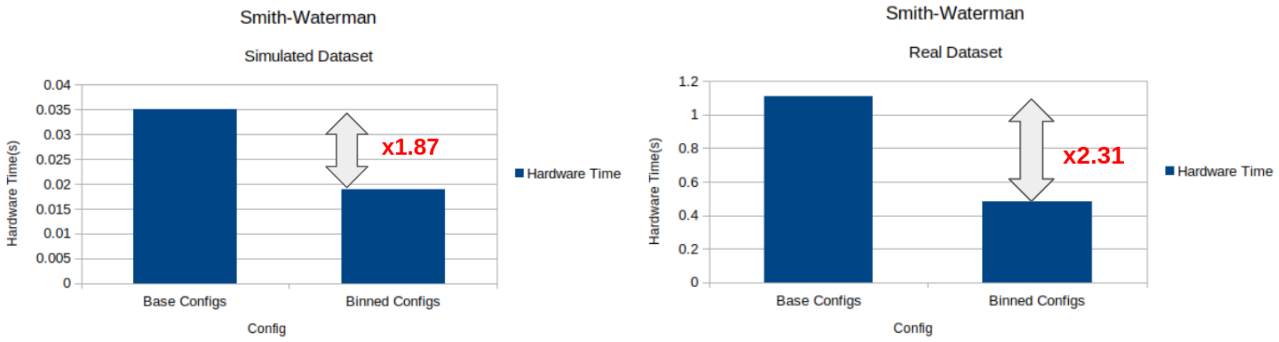
Σύγκριση μεταξύ λογισμικού και προτεινόμενης αρχιτεκτονικής

Παρατηρούμε σημαντική επιτάχυνση τόσο μεταξύ της αρχιτεκτονικής μας και του παραμετροποιημένου λογισμικού αλλά και μεταξύ αυτού με το βασικό λογισμικό , το οποίο δείχνει ότι η έρευνα είναι προς τη σωστή κατεύθυνση.



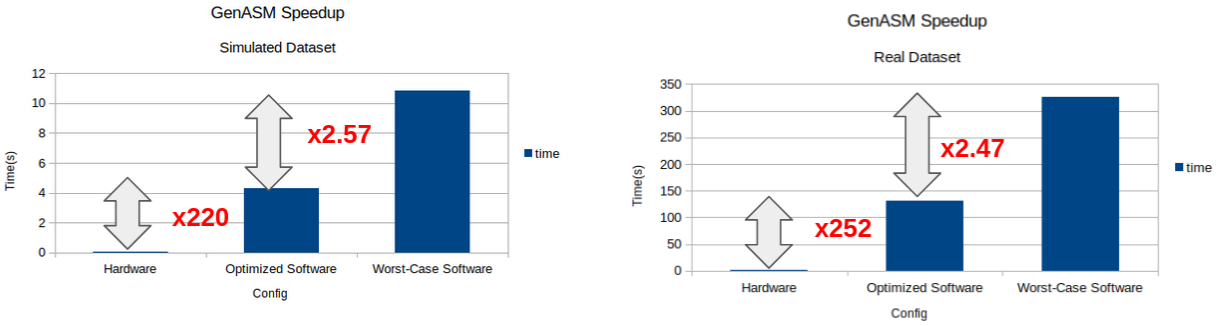
(a) Simulated (b) Real

Figure 14: Συγκριση δύο αρχιτεκτονικών για τον αλγόριθμο GenASM



(a) Simulated (b) Real

Figure 15: Συγκριση δύο αρχιτεκτονικών για τον αλγόριθμο Banded Smith Waterman

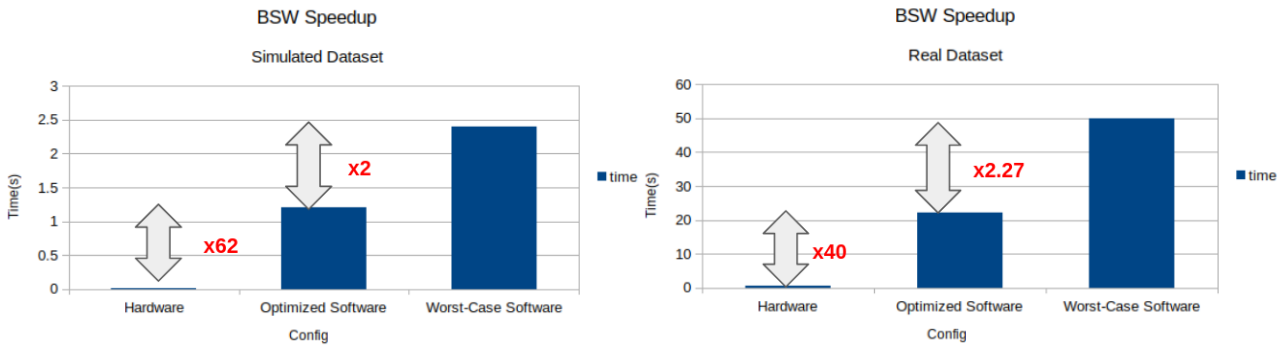


(a) Simulated (b) Real

Figure 16: Σύγκριση αρχιτεκτονικής με λογισμικό για τον αλγόριθμο GenASM

Σύγκριση ακρίβειας

Ο Banded Smith Waterman έχει καλύτερα αποτελέσματα από τον GenASM. Αυτό οφείλεται στο γεγονός ότι ο GenASM απορρίπτει αυτόματα ζεύγη που έχουν παραπάνω σφάλματα από το κατώφλι του, ενώ ο Banded Smith Waterman ενώ εγκυάται ότι θα βρει αντιστοίχιση αν το ζευγάρι έχει σφάλματα λιγότερα από το μέγεθος του band δεν αποκλίεται να βρει αντιστοίχιση με περισσότερα σφάλματα



(a) Simulated (b) Real

Figure 17: Σύγκριση αρχιτεκτονικής με λογισμικό για τον αλγόριθμο Banded Smith Waterman

αν αυτά δεν βγαίνουν εκτός του περιορισμένου χώρου. Οπότε σε κάποιες περιπτώσεις που το φίλτρο υπο-εκτίμησε τα σφάλματα ενός ζευγαριού πάλι μπορεί ο Banded Smith Waterman να βρει κατάλληλη αντιστοίχιση.

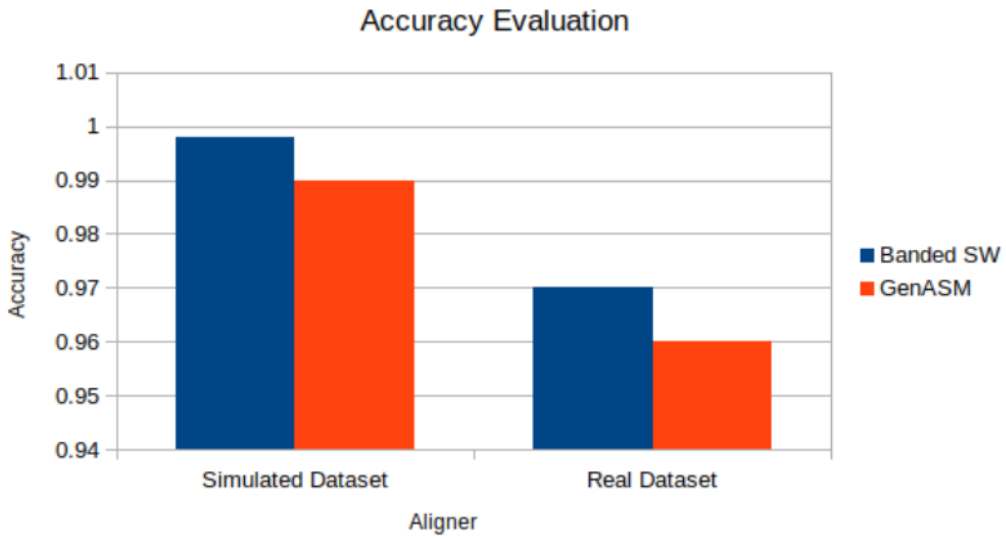


Figure 18: Σύγκριση ακρίβειας μεταξύ GenASM και Smith Waterman

Chapter 1

Introduction

Genomics is the systematic study of the complete DNA content of living organisms. It reveals crucial information that can revolutionize biology, change how we treat diseases and how we think about evolution. More specifically, it enables precise diagnosis of patients, pathogen surveillance, and in the case of plant organisms, even crop improvements. These applications depend on the transformation of raw genomic data (reads) generated by sequencing platforms into complete genomes. These genomes are then compared to a reference genome and through processes like variant calling and epigenetics we can extract useful information about our subjects[6]. The procedure that lies in the heart of this pipeline is the alignment step[1][2]. During the alignment billions of reads are mapped to a reference genome in order to reconstruct a complete DNA structure. Most aligners use a strategy called "seed and extend", in which reads are fragmented into smaller pieces called seeds in an attempt to find exact matches in the reference genome and narrow down the possible places the starting read could be mapped to. In the extension step each seed is extended into an approximate alignment allowing the existence of errors or edits.
Most modern aligners actualize the extension step through two distinct phases. The Matrix Fill step and the traceback step. During Matrix Fill the algorithms fill a similarity matrix between a read and a reference ,usually producing an alignment score. The more similar the sequences are, the greater the score. During Traceback information is used from the Matrix Fill step to identify the position and the kind of all the edits and thus reconstructs the alignment path. In the genomic pipeline, the extension process (both Matrix Fill and Traceback) is a major performance bottleneck, because of the excessive time requirements and computational intensity of the algorithms used. In this thesis the following approaches are explored: i) optimizing the alignment and thus accelerating the task , and ii) reducing the volume of alignment tasks by exploiting key features of algorithms.

Hardware acceleration and optimization of algorithms: Hardware acceleration is a broad term that encapsulates many solutions to accelerate a task with

specified hardware. The alignment step has been developed numerous times in devices such as GPUs, FPGAs and ASICs. Due to their bit-level customization and re-configurability, FPGAs have emerged as very promising hardware acceleration platforms for genomic pipelines and specifically to the alignment step. If the algorithm allows it, they can vastly parallelize the Matrix Fill step using systolic array structures, alleviating the computational bottleneck of this step. Another way hardware acceleration is enabled is through heuristics. By modifying some parts of an algorithm or by knowingly ignore some key data dependencies, further parallelization can be achieved, not without the introduction of a performance-accuracy tradeoff,which the hardware designer must confront.
Algorithm exploitation and filtering: Another optional but rather important step of a genomic pipeline is Filtering. After seeding, the potential pairs of reads and reference sequences can be filtered so that pairs that are not as similar as wanted, can be discarded early, so that no time and resources will be wasted on their alignment. Several pre-filtering algorithms produce great results in reducing alignment time. SneakySnake is state-of-the-art pre-fitering algorithm that achives great accuracy alignment result by using a very efficient technique to discard dissimilar extensions. SneakySnake and such filters can give great insights about the distribution of edits in a dataset and that can be exploited in many algorithms. Algorithms like Banded-Smith-Waterman thrive under conditions where the number of edits is known as a static band can be used to find the optimal alignment reducing computational intensity of the alignment step. These opportunities will be explored further in this thesis. Often these acceleration methods are used on generic datasets and produce suboptimal solutions, as pre-filtering is a data-aware procedure and the more it is customized to the dataset, the better results it can produce.

The main objective of this thesis is to investigate how pre-alignment filtering affects hardware-based sequence alignment accelerators. Profiling of existing aligners and datasets shows a heterogeneous distribution of edit distances, with most alignments containing few edits. Such cases can be resolved with fewer resources and operations than the baseline scenario typically assumed[3].

This work leverages edit-threshold-based pre-filters to optimize FPGA-based alignment on two levels:

At the architectural level, by designing individual alignment units that are resource- and performance-efficient for specific edit thresholds.

At the system level, by constructing a throughput-optimized multi-accelerator architecture composed of aligners tuned to different edit thresholds.

The methodology begins with profiling simulated and real datasets using state-of-the-art aligners to analyze edit distributions. Based on these insights,

algorithms capable of exploiting edit thresholds to reduce computational and spatial demands are selected and implemented using High-Level Synthesis (HLS). Multiple accelerator variants are synthesized for different thresholds, forming a pool of designs with varying latency and resource footprints.

Finally, a high-throughput multi-accelerator system is assembled, where each accelerator processes reads matching its threshold range. A balancing script, informed by synthesis reports and dataset statistics, determines the optimal combination and number of accelerators to minimize total alignment latency within available FPGA resources.

The structure of the rest of this thesis is as follows. In Chapter 2 a genomic pipeline review is presented along with an explanation of the aligners used as long as an explanation for SneakySnake. Also, there will be a brief review of the tools used. In Chapter 3 the methodology that was followed will be explained, as well the designs of the hardware kernels will be explained as well the overall architecture that was achieved within the platform. In Chapter 4 the results will be shown with comprehensive graphs and finally in Chapter 5 some thoughts for future work will be mentioned.

Chapter 2

Theoretical Background

2.1 Genomics and the genomics pipeline

Genomics is the study of the complete DNA content of an organism, encompassing not only the sequence of nucleotides but also how that sequence varies across cells, individuals, populations, and time. Because genomes encode the instructions for cellular function and inheritance, genomics underpins modern biology and medicine: it reveals the genetic basis of rare and common diseases, tracks pathogen evolution and outbreaks, informs cancer diagnosis and therapy, guides breeding and conservation in agriculture and ecology, and enables basic discoveries about gene regulation, development, and evolution. The field has been transformed by high-throughput sequencing, turning DNA into a digital substrate that can be measured at scale and analyzed computationally[4][5][6].

A typical genomics pipeline begins with sequencing, where DNA molecules are fragmented, prepared into libraries, and read by an instrument that produces millions to billions of signal traces. Short-read platforms (e.g., sequencing-by-synthesis) yield highly accurate reads of 75–300 bp, while long-read platforms (e.g., nanopore and single-molecule real-time sequencing) produce kilobase-to-megabase reads that resolve repeats and structural variation at the cost of higher raw error rates. Experimental design choices here—coverage depth, read length, and paired-end vs single-end—determine downstream power and biases[7].

Basecalling converts the instrument's raw signals into nucleotide sequences with per-base quality scores. To be more percise during basecalling the signals from the sequencing machines are translated into the following nucleotide bases: A for Adenine, T for Thymine, C for Cytosine and G for Guanine. In short-read systems this maps fluorescence intensity patterns to bases; in nanopore systems it maps ionic current traces to bases, typically using deep neural networks. Basecalling accuracy and calibration of Phred quality scores directly affect downstream sensitivity and false discovery, so pipelines often retain both sequences (FASTQ) and qualities for later filtering and modeling[7].

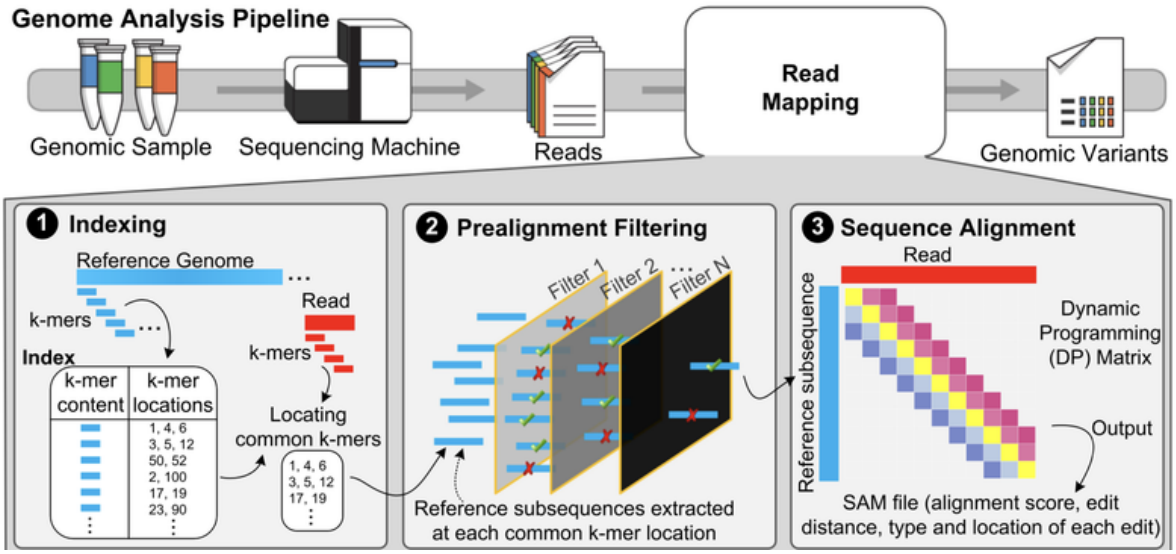


Figure 2.1: Typical Genomics Pipeline for DNA analysis [7]

Seeding identifies candidate genomic locations for each read efficiently by matching short substrings. Common strategies include fixed-length k-mers, spaced seeds that tolerate mismatches, minimizers and syncmers that subsample representative k-mers to reduce redundancy, and index-based methods using FM-index/BWT or hash tables over the reference. Effective seeding balances sensitivity (not missing true loci) against speed and memory, and often generates multiple candidate hits per read for later pruning[8][9][10].

Filtering narrows candidates before expensive alignment. Typical filters remove low-quality reads, adapter or low-complexity sequences, and exact or near-exact duplicates; they prune seed hits by distance constraints, chaining heuristics across colinear seeds, or quick Hamming/edit-distance screens. Lightweight estimators (e.g., Jaccard similarity from sketches) and mapping-quality models are used to discard unlikely mappings early, protecting throughput while preserving sensitivity[11][12]].

Alignment refines the mapping at nucleotide resolution by computing an optimal or near-optimal match between each read and its candidate reference region under a scoring model for matches, mismatches, and gaps. Dynamic-programming algorithms such as Smith–Waterman (local) or Needleman–Wunsch (global) provide exact solutions; banded, affine-gap, and wavefront formulations accelerate computation; seed-and-extend frameworks restrict DP to neighborhoods suggested by seeding. Choice of alignment mode (local vs end-to-end), gap penalties, and bandwidth influences indel detection, handling of soft clipping, and performance on repetitive or error-prone regions[10].

Variant calling infers differences between the sample and a reference genome by aggregating aligned evidence across reads. Germline callers model diploid (or polyploid) genotypes and output SNPs and small indels with genotype

likelihoods and quality metrics; somatic callers compare tumor–normal pairs or tumor-only data to identify subclonal variants amid copy-number changes; long-read callers additionally resolve larger indels and structural variants (SVs) such as inversions, translocations, and copy-number changes. Modern callers integrate base qualities, mapping qualities, read orientation and pairing, local assembly around candidate sites, and machine-learned calibration. Post-calling steps include joint genotyping across cohorts, phasing to reconstruct haplotypes, annotation against gene and population databases, and filtering strategies that control false discovery while retaining true positives in challenging regions[13][14].
Across all stages, rigorous quality control and reproducibility are essential. Pipelines typically track run metrics (yield, quality score distributions, coverage, duplication, insert size), use reference standards for benchmarking, and encode processing steps in workflow systems to ensure portability, versioning, and provenance. Choices at each step propagate to downstream accuracy and computational cost, so well-designed pipelines make explicit trade-offs among sensitivity, precision, runtime, and memory in light of the biological question and the sequencing technology[17].
In this particular thesis the main focus are the filtering and alignment steps. In the following sections these two steps will be explained in great detail along with the actual algorithms that were used in this work[18][19].

2.2 Filtering

2.2.1 General

In a seed-and-extend mapper, filtering is the triage step between seeding and full dynamic-programming alignment. Its job is to discard candidate read–reference loci that cannot possibly be within a user-specified edit-distance (or scoring) threshold, so that only a small fraction of hard cases reach the expensive aligner. Classical pre-filters rely on combinatorial or sketching arguments: q-gram counting guarantees that two strings within k edits must share at least a certain number of k-mers, enabling quick rejection when the shared count is too low; this idea underlies many early filters and remains a baseline for sensitivity–speed trade-offs. Advantages include simplicity and determinism; disadvantages are memory for large q and reduced power on indel-rich reads because indels disrupt many consecutive q-grams[20][21]. More modern filters use bit-parallel encodings and SIMD to approximate edit distance quickly. Shifted Hamming Distance (SHD) slides multiple bit-masks to tolerate up to k edits and rejects pairs that provably exceed k; it is fast and comprehensive for short reads but can pass more false positives as divergence grows, leaving extra

work for the aligner[22].

Hardware-centric designs such as GateKeeper move this screening onto FPGAs (and recently GPUs), achieving very high throughput with low latency; they excel when the seeding stage yields many candidates, but require device-specific implementations and careful batching to hide I/O[23].

Other ecosystem filters operate earlier on seeds themselves, e.g., GRIM-Filter prunes seed locations using bloom-like region summaries to cut down extension load, and BLEND tolerates near-exact seed matches in one hash lookup, improving sensitivity at similar compute cost; both reduce the number of loci that even reach pre-alignment filtering, though their effectiveness depends on the seeding scheme and reference indexing strategy[24][24].

SneakySnake is a pre-alignment filter that recasts approximate string matching as a shortest-path problem on a "neighborhood map" grid, equivalent to single-net routing in VLSI [15]. It seeks a monotone path from the origin to the opposite corner whose cost does not exceed the edit threshold; if no such path exists, the pair is safely rejected without running dynamic programming. This formulation yields several practical advantages: first, accuracy—SneakySnake reduces false accepts by up to orders of magnitude versus prior filters like SHD, Shouji, and GateKeeper at the same thresholds, which directly lowers wasted aligner work and improves end-to-end throughput; second, universality and portability—the same algorithm maps well to CPUs, GPUs, and FPGAs with similar core logic, making it straightforward to integrate into heterogeneous pipelines; third, scalability—the grid walk inspects only a thin band around the main diagonal proportional to the edit budget, so runtime and memory scale with k rather than read length, benefiting both short and long reads; and fourth, non-intrusiveness—because it is a filter rather than a substitute aligner, you retain the full scoring models and features of downstream aligners. Reported accelerations include double- to triple-digit speedups for popular libraries (e.g., Edlib) and substantial reductions in total mapping time, with accompanying Snake-on-Chip and Snake-on-GPU implementations demonstrating high throughput on real datasets. The main trade-offs are that, like all threshold-based filters, sensitivity depends on a correct or conservative edit budget, and extremely noisy reads may still pass to alignment more often[25][23][8][26][27].

2.2.2 SneakySnake

As mentioned the SneakySnake [15] filtering algorithm draws inspiration from the SNR routing problem in VLSI chips. The SNR problem is solved by using a neighborhood map and trying to reach from the beginning to the end encountering as few obstacles as possible as shown in Figure 2.2. In SneakySnake this kind of matrix is calculated in the following manner. After the seeding step of the

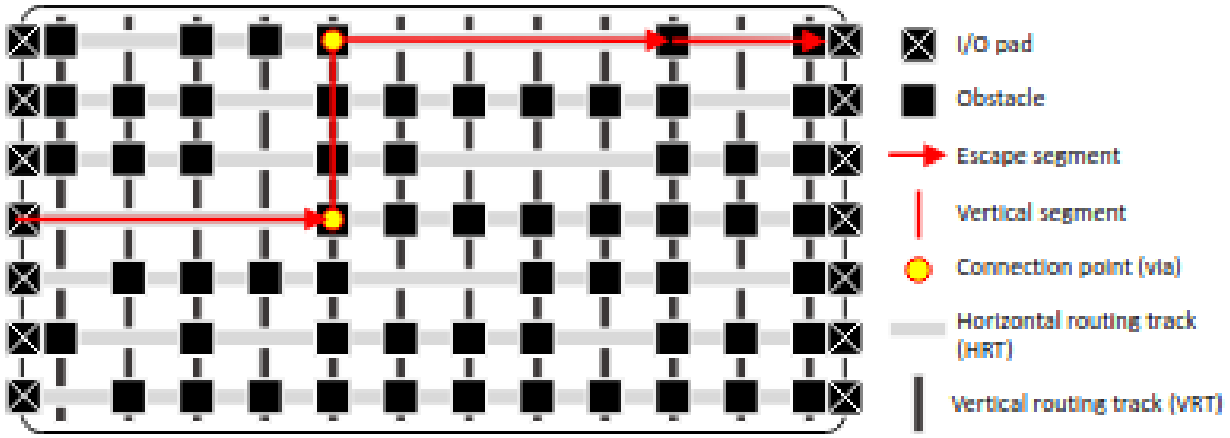


Figure 2.2: SNR problem in VLSI [16]

pipeline is complete the data is seperated into pairs of read-reference sequences. The reads are the sequences that ultimately came out of the sequencing machine and the reference sequences are parts of the reference genome that may be a good candidate spot for alignment. SneakySnake works well when the two sequences have the same length, as differences in the length of the sequences could be interpreted as edits.
To construct the neighborhood map, the read sequence is shifted and compared to the reference sequence. The number of shifts is equal to the edit threshold given by the designer to the left and to the right. So, if the edit threshold is E, the matrix will have 2*E+1 horizontal lines (one for no shifting , E for left-shift and E for right-shift). The comparison is being done, comparing nucleotide bases (A,C,G,T) and filling the matrix with a 0 where the sequences match and with an 1 where the sequences do not match. The ones are the obstacles in this case. So, the final matrix looks a lot like the SNR equivalent. This matrix has two very important aspects. First, every cell of it can be computed in parallel. Second, all horizontal lines of the matrix can be processed without any data dependencies. These two feature enable great parallelization in various platforms, like FPGAs, GPUs and even multi-threaded CPUs.
After the matrix is complete, the SneakySnake algorithm starts by counting zeroes (matches) simultaneously in all lines of the matrix,trying to find the longest segment of 0s before reaching an obstacle (an 1). Then it stops, skips the obstacle and repeats this process until either of two things happen:

- The algorithm found obstacles greater than the edit threshold: The pair is too dissimilar and is discarded never reaching to the alignment step
- The algorithm reaches the end of the matrix: The algorithm found equal or less obstacles than the edit threshold, so the pair is similar enough to reach the alignment step

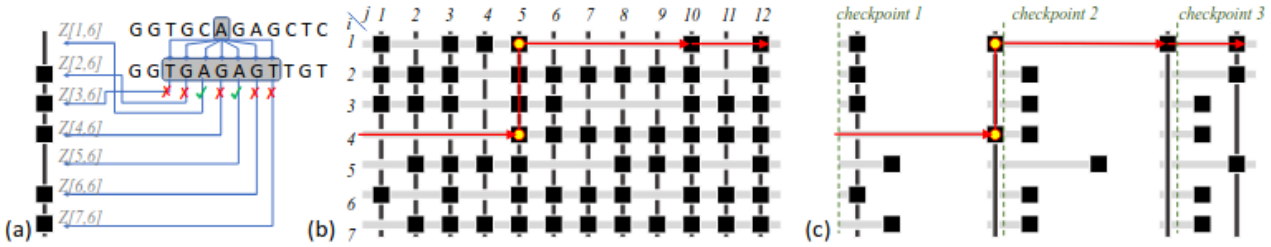


Figure 2.3: The SneakySnake algorithm. In a) the matrix is calculated by comparing the shifted read to the reference. In b) the algorithm finds the most zeroes until it finds an obstacle and repeats the algorithm. In c) The algorithm found a path with edits less than or equal to the edit threshold (E=3) so the pair will reach the alignment step [16]

This procedure is shown in Figure 2.3
The SneakySnake algorithm guarantees the following two qualities:

- The SneakySnake algorithm is guaranteed to find a signal net that interconnects the source terminal and the destination terminal when one exists.
- When a signal net exists between the source terminal and the destination terminal, using the SneakySnake algorithm, a signal from the source terminal reaches the destination terminal with the minimum possible latency

This means that SneakySnakes never overestimates the edits between two sequences. This is a highly important feature as no other filtering algorithm can ensure that it will not have false rejections. Despite that, as it is common in pre-filtering, SneakySnake occasionally underestimates the number of edits between two sequences, but even then it is extremely better than other filters. This typically does not raise an issue as the only downside of false accepted pairs is that an alignment will be attempted at slightly more pairs than needed, which overall does not impact greatly the speedup it offers to the pipeline. As it will be explained in Chapter 3 this raises a problem in the workflow of the current thesis and will introduce an accuracy loss on the complete system that needs to be addressed.

2.3 Alignment

Alignment is the step that turns a set of sequenced reads into precise hypotheses about where those reads came from in a reference genome and how they differ from it. Formally, alignment finds, for each read, one or more reference loci and an edit script—matches, mismatches, insertions, and deletions—that maximizes a scoring function under biological constraints. Conceptually there are three classical modes. Global (Needleman–Wunsch) alignment forces both sequences to align end-to-end, which is appropriate for assembled contigs or

amplicons. Local (Smith–Waterman) alignment finds the highest-scoring subsequence match and is robust to adapters, chimeras, and partial matches. Semi-global (end-to-end read, local reference) is common in mapping, penalizing internal gaps but tolerating clipping at read ends to handle residual adapters and variable insert sizes. Most modern mappers embed local or semi-global alignment inside a larger "seed-and-extend" framework[28][29][30][31].

The scoring model encodes biological plausibility. Matches receive positive scores, mismatches and gaps receive penalties. Affine-gap costs are nearly universal: opening a gap carries a larger penalty than extending it, reflecting the fact that single multi-base indels are more likely than many tiny ones. Some aligners modulate mismatch penalties with base quality scores and known variable sites; others incorporate context-dependent gap costs to better handle homopolymers. The dynamic-programming (DP) matrix that computes the optimal score has quadratic worst-case time, so practical mappers restrict where they fill it. Common accelerations include banded DP around the main diagonal, X-drop or Z-drop termination that abandons unpromising extensions, bit-parallel algorithms, SIMD-vectorized kernels, and the wavefront algorithm (WFA) whose cost scales with the edit distance rather than read length. The output is typically emitted in SAM/BAM/CRAM with a CIGAR string describing the edit script, alignment score tags, mapping quality (MAPQ), and flags that distinguish primary, secondary, and supplementary alignments[1][32][33][29][34].

Because genomes are large and repetitive, alignment begins with indexing and seeding to narrow candidates. FM-index/BWT based mappers (e.g., BWA-MEM, Bowtie2) search the reference implicitly and are very memory-efficient; hash-based and minimizer-based mappers (e.g., minimap2) sample representative k-mers and chain co-linear seeds into long anchors before DP. For RNA-seq, "spliced" aligners add an intron model that permits long deletions consistent with splice junctions and can leverage annotations to improve sensitivity[35][28][36][37].

Short-read mainstays include BWA-MEM and Bowtie2. Their strengths are mature indexing, good speed–accuracy balance, careful MAPQ models, and broad ecosystem support. Weaknesses include difficulties in highly diverged regions, long tandem repeats, and complex structural variation; they also rely on heuristics that can occasionally mis-prioritize among many near-identical loci. SNAP and STAR (for RNA-seq) emphasize high throughput, with STAR excelling at splice junction detection in exchange for substantial memory. For long-read data, minimap2 is the de facto standard across DNA and RNA. It is fast, versatile, and robust to high error rates, with strong chaining that reduces DP work and good SV/split-read handling; trade-offs include heuristic tuning across modes and occasional under-penalization of certain gap patterns,

which can affect precise indel placement in low-complexity regions. Specialized long-read aligners such as NGMLR, GraphMap, LRA, and Winnowmap target structurally complex or highly repetitive genomes; they often improve alignment contiguity and SV breakpoint placement but can be slower or more memory-hungry and may require careful parameterization for different read accuracies (raw nanopore vs PacBio HiFi)[28][35][38][39][40].

At the core, many tools reuse or expose optimized DP engines. Libraries like ksw2 (used by BWA-MEM/minimap2), Edlib, and WFA provide SIMD or wavefront primitives for global, local, and semi-global alignment with affine gaps. Their strengths are speed and exactness within the restricted band or error budget; their limitations are inherited from the surrounding heuristics: if seeding or chaining misses the right locus, no downstream DP can fix the mapping. Hardware acceleration (GPU/FPGA) increasingly offloads extension DP, traceback, or pre-alignment filtering to achieve order-of-magnitude throughput gains, but integration must account for I/O and batching to avoid new bottlenecks[35][32][26][41].

Two practical issues dominate downstream correctness. First, multi-mapping and ambiguity: in repetitive regions, multiple loci produce near-identical scores. Aligners report one "primary" location and may emit "secondary" or "supplementary" alignments for alternatives and split mappings. The MAPQ field tries to summarize uniqueness but is model-dependent; analyses should avoid over-interpreting high MAPQ in repeats and should retain secondary/supplementary records for SV and fusion discovery. Second, scoring and clipping choices shape variant representation. Aggressive soft-clipping can hide real indels at read ends; overly narrow DP bands can misplace indels; affine parameters that penalize gap opens too strongly can fragment true long indels into runs of mismatches. Best practice is to keep base qualities, tune presets to the read technology, and validate with truth sets; for variant calling, many pipelines realign locally around candidate sites or perform graph-based assembly to mitigate alignment artifacts.

In summary, alignment is an optimization problem wrapped in engineering compromises: indexes and seeds find plausible loci, chaining and filters cut the search, and a DP kernel produces base-level edits under a biologically motivated score. Short-read aligners deliver speed and small-variant precision at scale but struggle in repeats and with long indels; long-read aligners span repeats and resolve structural variation with modest trade-offs in runtime and, for noisier reads, per-base accuracy. Understanding these strengths and weaknesses—and how parameters, heuristics, and reporting fields interact—lets you design pipelines that are both computationally efficient and faithful to the biology you aim to infer.

In this thesis three aligners will be examined:

- Smith-Waterman: the classic DP algorithm used in a banded approach[42]
- GenASM: an efficient algorithm based on BITAP][43]
- WFA: a newer, ambitious algorithm with cost that scales to edit distance rather than read length[32][44]

2.3.1 Smith-Waterman

2.3.1.1 Base Algorithm

The Smith–Waterman algorithm [45][42]performs local sequence alignment; that is, for determining similar regions between two strings of nucleic acid sequences or protein sequences. Instead of looking at the entire sequence, the Smith–Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure.

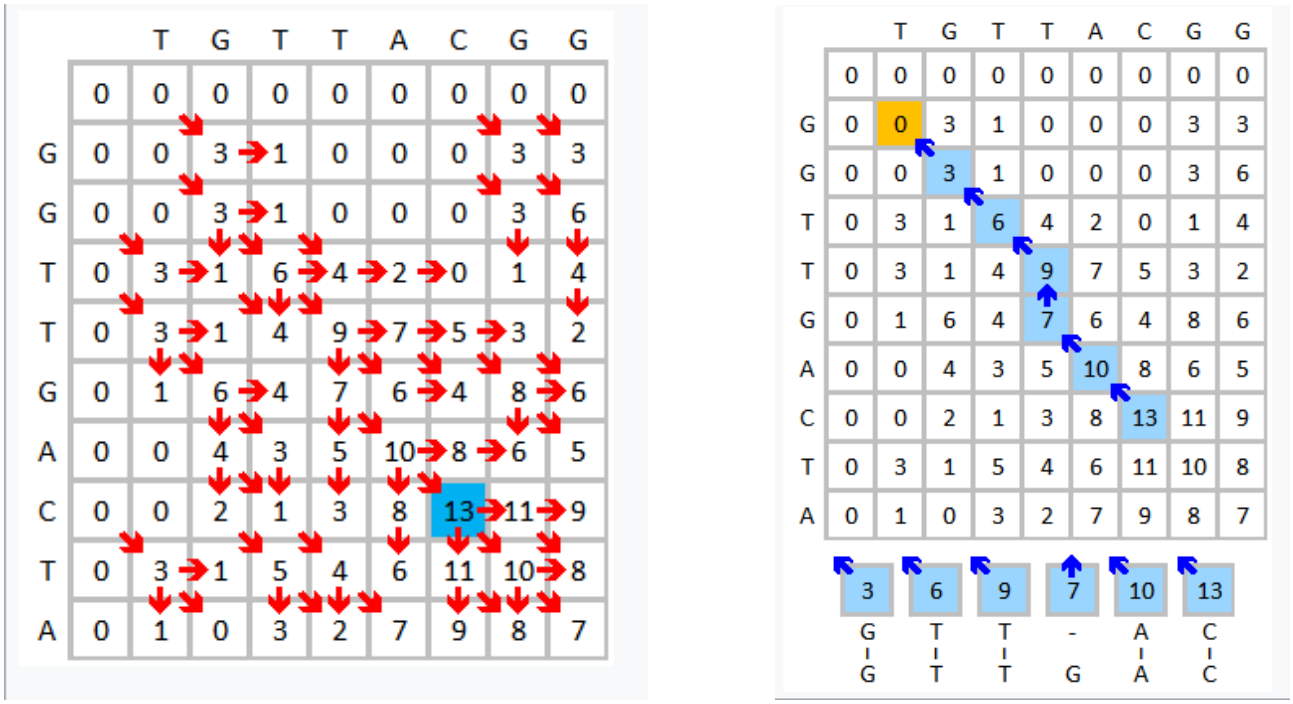
As many aligners have in common, Smith-Waterman has a matrix fill and a traceback step. During matrix fill, a similarity matrix H is computed which has (n+1)x(m+1) dimensions , where n is the length of the reference sequence and m is the length of the read sequence. Before the computation of the similarity matrix a similarity score must be assigned which entails how much the algorithm rewards a match between the two sequences, and how much it penalizes mismatches and gaps. These constants often occur from biological constraints of the organism that is studied. The algorithm is as follows:

- Let $A = a_0, a_1, a_2, ..., a_n$be the reference sequence and $B = b_0, b_1, b_2, ...b_m$ the read sequence
- Initialize the first row and first column of the matrix with zeroes
- Fill the rest of the matrix with the following formula:

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ \max_{k \geq 1}\{H_{i-k,j} - W_k\}, \\ \max_{l \geq 1}\{H_{i,j-l} - W_l\}, \\ 0 \end{cases} \qquad (1 \leq i \leq n, 1 \leq j \leq m) \qquad (2.1)$$

  where,

  - W_k is the penalty score of gap of length k
  - s(a,b) the similarity score (positive for match, negative for mismatch)
  - $H_i - k, j$ - W_k is the score if a_i is at the end of a gap with length k
  - $H_i, j - l$ - W_l is the score if b_j is at the end of a gap with length l



(a) Matrix Fill

(b) Traceback

Figure 2.4: Smith-Waterman Algorithm [46]

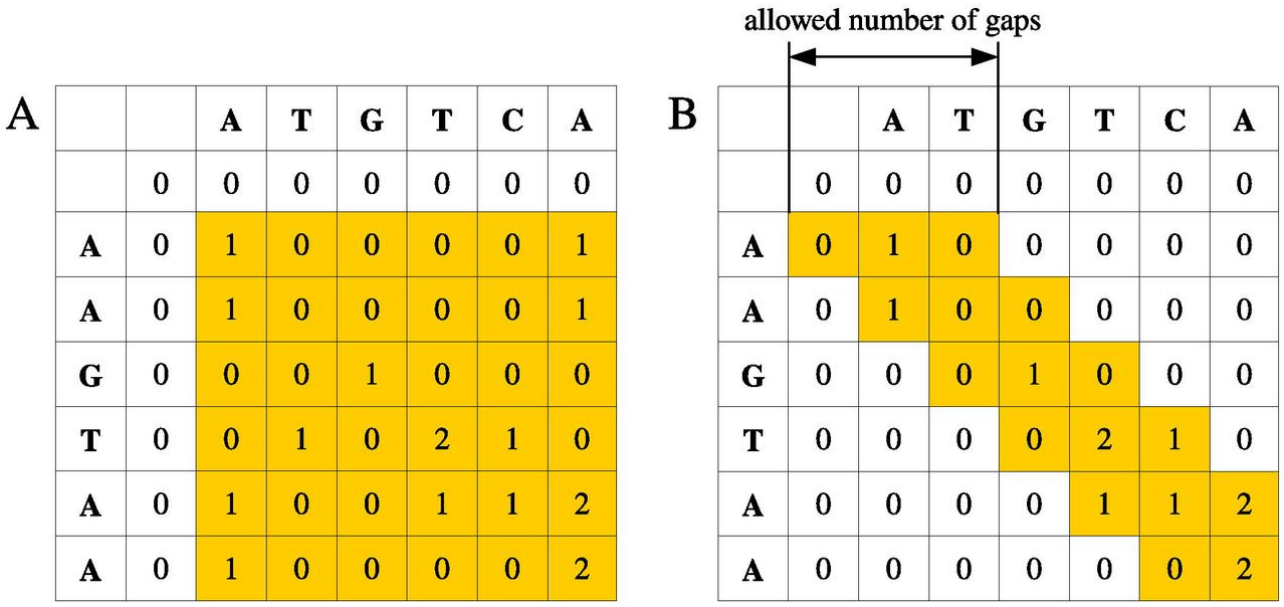
  - 0 means there is no similarity between a_i and b_j

- At the same time as the computation of the matrix, information is also stored about the branch of the equation (2.1) that was used to get to the number in each cell of the matrix. This way a traceback algorithm starting at the highest scoring cell can find an alignment path tracing back to the first cell it finds with score 0. An example of the matrix fill and traceback steps are shown in Figure 2.4. A diagonal arrow means that a match or a substitution error exists in that place. An upwards arrow means that a gap in the reference sequence is inserted (an insertion error) and a sideways arrow means that a gap in the read sequence is inserted (a deletion error).

The Smith-Waterman algorithm has square spatial and time complexity as it needs to compute n*m values and it needs to store them all for traceback. These two points make it a very resource and time consuming DP algorithm. In Chapter 3 it will be discussed how it can be parallelized and optimized via the use of systolic arrays and diagonalization.

2.3.1.2 Banded Smith-Waterman

As explained in (insert reference), it is true that if a pair of sequences has E edits, the max-score in the Smith-Waterman similarity matrix will be within the 2*E+1 diagonals with the main diagonal of the matrix as the center diagonal of these. That means that for a given edit threshold Smith-Waterman can be



		A	T	G	T	C	A
	0	0	0	0	0	0	0
A	0	1	0	0	0	0	1
A	0	1	0	0	0	0	1
G	0	0	0	1	0	0	0
T	0	0	1	0	2	1	0
A	0	1	0	0	1	1	2
A	0	1	0	0	0	0	2

		A	T	G	T	C	A
	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0
A	0	1	0	0	0	0	0
G	0	0	0	1	0	0	0
T	0	0	0	0	2	1	0
A	0	0	0	0	1	1	2
A	0	0	0	0	0	0	2

Figure 2.5: Standard vs Banded Smith Waterman [47]

reduced to a specific band within the matrix and eliminate the need for storing and searching the entire score matrix. As shown in Figure 2.5, for E = 1, only the main diagonal and one diagonal on each side must be computed to find an alignment for the two sequences. It needs to be mentioned that for traceback to be executed, the algorithm needs one additional diagonal on each side because it also checks neighboring cells forming a "halo" around the yellow area in Figure 2.6. As an example as it will be explained in Chapter 3, for two sequences which are 100 nucleotide bases long, the matrix would have 10000 cells to be computed and stored for traceback. With Banded Smith-Waterman for few edits (eg. E = 3), only (2*3+3)*199 = 1791 cells must be computed and store which is less than 20% of what the standard algorithm needed (The number 199 is the number of diagonals of the score matrix for sequences of length 100).
This observation makes the Banded Smith-Waterman a very efficient alignment algorithm and with the systolic array optimization it can be a great aligner for platforms like FPGAs, as it will be detailed later in this thesis.

2.3.2 GenASM

GenASM [43]is a co-designed algorithm–hardware framework that accelerates approximate string matching (ASM)—the main bottleneck in read mapping and several downstream genomics tasks. Instead of quadratic-time dynamic programming, GenASM builds on a reworked, bit-vector form of the Bitap algorithm, chosen for its simple, word-parallel bitwise operations. The authors remove loop-carried dependencies to expose intra-alignment parallelism, extend Bitap to handle long as well as short reads, and introduce the first Bitap-

compatible traceback method, enabling full alignments (not just distances).

The hardware consists of two tightly integrated units: GenASM-DC for distance computation using parallel, on-chip bit-vector updates (match/ins/del/-sub) and GenASM-TB for high-throughput traceback. By matching compute with on-chip SRAM bandwidth and capacity in a systolic-style architecture, performance scales linearly with the number of compute units while keeping power and area low.

Across three use cases—read alignment, pre-alignment filtering, and edit-distance computation—GenASM consistently outperforms state-of-the-art software and prior accelerators, often by large margins (e.g., >100× speedups vs. leading software on both long and short reads, multi-× gains over FPGA/ASIC baselines), while substantially reducing power. Beyond these evaluated settings, the same engine is applicable to other genome analysis tasks (e.g., WGA/MSA) and even general text processing. In sum, GenASM demonstrates that a Bitap-based, memory-balanced accelerator can deliver high throughput, low power, and flexibility for modern sequencing workloads.

2.3.2.1 BITAP Algorithm

Given a text (reference) T and a pattern (read) P of length m, Bitap scans T to find all end positions where some substring of T matches P with at most k edits (Levenshtein distance). When k=0, it finds exact matches. The algorithm works by turning P into bitmasks and then updating a small set of status bit-vectors with only bitwise ops and shifts as it sweeps through T.

To create the bitmasks, the Bitap algorithm preprocess P once to build four masks PM[a] (length m bits) for every alphabet symbol a from A,C,G,T. By convention, 0 means "match" in Bitap, so PM[a][i]=0 iff P[i]==a; otherwise it is 1. These masks let the algorithm compare the next text character to all pattern positions in parallel using simple bitwise operators.

The algorithm maintains m-bit status bit-vectors R[d] one for each number of edits. Intuitively, at text index i, the j-th bit of R[d] summarizes whether the suffix of P starting at position j can match the text suffix starting at i with at most d edits—again where again 0 is a match. All R[d] are initialized with all-ones.

After the pre-process and initialization steps the main algorithm begins. The main loop of the algorithm traverses the Text sequence (reference), so for each text character c=T[i]:

- It retrieves its precomputed mask curPM = PM[c].
- It updates R[0] (exact matching) by shifting left one bit (consume one pattern char) and doing an OR operation with curPM.

- For each d=1..k, it forms intermediate bit-vectors that capture the ways we could incur up to d edits at this text step, then combines them:
  - Deletion (D): consume a pattern character without consuming text (delete in pattern).
  - Insertion (I): consume a text character without consuming pattern (insert in pattern).
  - Substitution (S): consume both but allow a mismatch.
  - Match (M): consume both only where curPM indicates equality.

  All four are expressed with left shifts and AND/ORs on the previous iteration's R[·] (and curPM). The combined result is the new R[d]. This is the key to Bitap's speed: we update m positions at once with a handful of word-level operations.

Then after the main loop is finished Bitap checks the most significant bit (MSB) of each R[d]: if it is 0 for any d less than or equal to k, then the pattern matches a text substring ending at position i with edit distance d. Finally it records the position and the minimal d. An example is presented as follows:

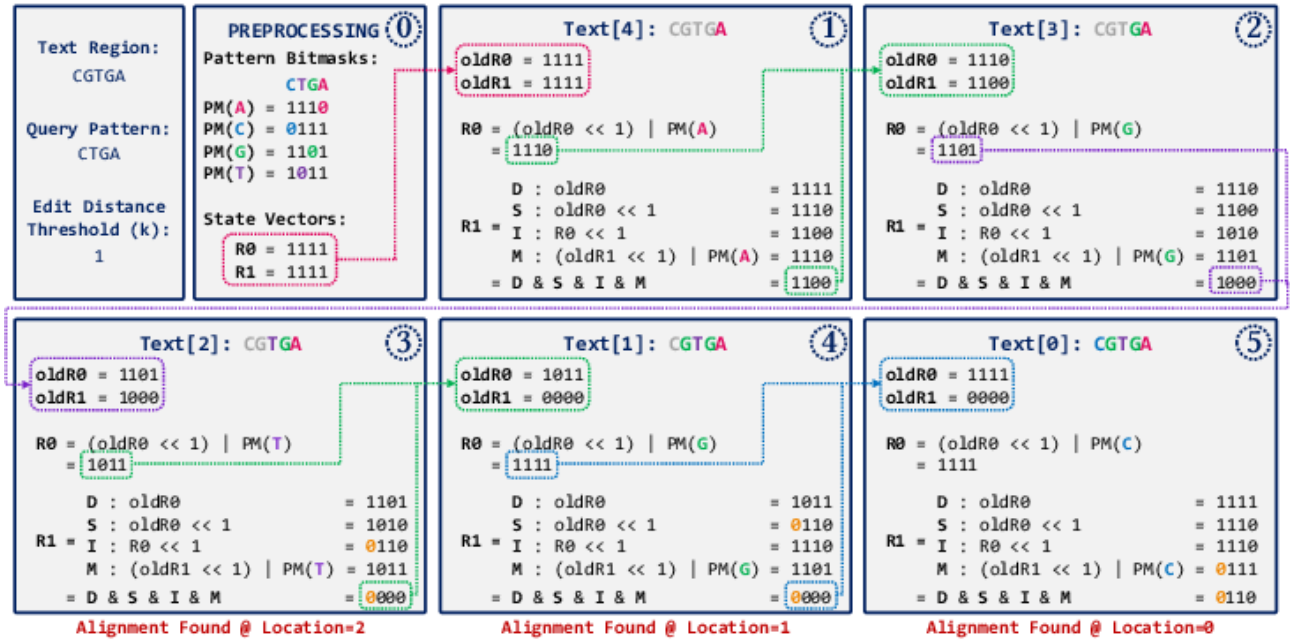


Figure 2.6: Example of Bitap Algorithm [48]

The Bitap algorithm is very hardware-friendly. The inner loop uses only shifts, ANDs, and ORs over compact bit-vectors, it has no branches and regular memory accesses so it maps well to SIMD/SIMT and systolic hardware. The GenASM algorithm builds on this, but the baseline Bitap already illustrates why bit-parallel ASM has such high throughput potential.
The reasons the authors saw the need to evolve Bitap into GenASM, are these 5 pitfalls:

- No support for long reads. Bit-vectors R have length equal to the read length. So for larger reads with potential higher edit thresholds, the storage requirements are utilizing more and more of a platforms resources.
- Data Dependency between Iterations. The computed bitvectors of each iteration depend on the bitvectors in the previous iteration.
- No Traceback support. There is no stored information that could produce a CIGAR string through a traceback process.
- Limited Parallelism. There is a computational bottleneck on compute units on modern CPUs that even after alleviating the data dependencies could prove fatal for parallelization.
- Limited Memory Bandwidth. The above problem is maybe solved by moving to GPUs , but GPUs have problems with memory bandwidth (the bitvectors need big memory bandwidth to be computed in parallel).

So GenASM is an attempt to evolve Bitap by solving all of the aforementioned problems.

2.3.2.2 GenASM-DC

The GenASM algorithm is split into to distinct steps. GenASM-DC (distance calculation) which is the evolution of Bitap by solving many of eaach problems and by providing the appropriate data to perform traceback. GenASM-TB (traceback) uses the data stored by GenASM-DC to create a CIGAR string (the alignment of the pair of sequences).
The modifications of Bitap that produce GenASM-DC are the following:

- Long Read Support. GenASM can store long reads over multiple words, further improving on Bitap, by introducing a computational overhead for operations like shifting. In this thesis only short reads are used, so this optimization will not be addressed.
- Loop Dependency Removal. GenASM is a co-designed algorithm. The main algorithm resides on a PL part of some platform to utilize its parallelization capabilities. To remove the loop dependencies, GenASM performs loop unrolling, enabling computations of independent bit-vectors in parallel. This will be shown later in Chapter 3 where the hardware-designs of this thesis will be explained.
- Text-level Parallelism. This is done by overlapping windows of the text sequence and compute those in parallel. This is advised for longer reads so it will not be addressed in this thesis.

- Traceback Support. In Bitap four vectors (insertion, deletion, substitution, match) were computed to then be saved as an single AND'ed version in R[d]. These vectors contain useful information that can be used fro traceback. So in the GenASM algorithm, instead of of saving only the R[d] vectors, all four vectors are saved in a large array.

2.3.2.3 GenASM-TB

After GenASM-DC has scanned a text window and found a match position and its edit distance d, GenASM-TB reconstructs the alignment path—i.e., the sequence of matches (M), substitutions (S), insertions (I), and deletions (D)—and emits a CIGAR string. The key idea is to reuse the intermediate bit-vectors (for M/S/I/D) that GenASM-DC produced at every text step and error level. A zero bit indicates that the corresponding operation is feasible at that position and error budget.

Traceback begins at the most significant bit (MSB) where a zero in some R[d] indicates a successful alignment end (length m with d errors) and proceeds "backwards" toward the least significant bit (LSB), effectively undoing the bit-wise updates performed during the forward pass. At each step, GenASM-TB examines the stored per-step bit-vectors in this priority: (i) extend an ongoing insertion or deletion if possible (to coalesce runs into one CIGAR op), else (ii) take a match if available, else (iii) take a substitution, else (iv) open a new insertion, else (v) open a new deletion. Based on the chosen operation, it updates three indices:

- patternI (pattern position / bit being traced),
- textI (text position within the current window),
- curError (remaining error budget).

Transitions follow the usual Levenshtein semantics:

- M (match): consume both pattern and text; errors unchanged $(x, y, z) \rightarrow (x-1, y+1, z)$
- S (substitution): consume both; errors decrease $(x, y, z) \rightarrow (x-1, y+1, z-1)$
- I (insertion in pattern): consume pattern only; errors decrease $(x, y, z) \rightarrow (x-1, y, z-1)$
- D (deletion from pattern): consume text only; errors decrease $(x, y, z) \rightarrow (x, y+1, z-1)$

In general, GenASM aligns in windows of size W with overlap O,that is why TB runs per window: it initializes indices, walks until (W−O) characters are consumed (to keep overlap for the next window), emits the local CIGAR (merging consecutive identical ops), advances the window anchors, and repeats. Unlike GenASM-DC's regular, bit-parallel loop, TB's control flow is data-dependent (it follows chains of zeros across the saved bit-vectors), but it remains lightweight because it touches only the compact bit-state already produced during DC. The windowed heuristic is not used in this thesis as only short reads are used and the computational overhead does not give enough storage benefits to consider it.

2.3.3 WFA

Pairwise alignment is a foundational primitive in computational genomics, underpinning read mapping, variant calling, de novo assembly, and multiple sequence alignment, among other pipelines. Classical dynamic-programming (DP) approaches such as Needleman–Wunsch for gap-linear penalties and Smith Waterman for gap-affine penalties guarantee optimality but incur quadratic time and space in the sequence lengths. This quadratic cost rapidly becomes the bottleneck as modern sequencing produces both enormous volumes of reads and, with third-generation technologies like PacBio and Oxford Nanopore Technologies (ONT), read lengths often exceeding tens of kilobases. Over the years, an extensive body of work has squeezed significant constant-factor speedups from DP through vectorization techniques, clever data layouts, banded computations, integer saturation, and cut-offs, and through highly tuned libraries such as SSW, SeqAn, KSW2, and GABA. Yet, despite these advances, the core O(nm) dependence remains, vectorization tends to be architecture-specific, and heuristics can forgo optimality. A key observation is that classical DP evaluates essentially the same number of cells regardless of how similar the two sequences are, leaving potential performance gains on the table when the true optimal alignment deviates only modestly from the main diagonal.

The Wavefront Alignment algorithm (WFA) [32][44]rethinks this landscape by replacing cell-by-cell DP with a score-by-score exploration that explicitly exploits sequence similarity. WFA targets the common gap-affine scoring model with penalties p = a, x, o, e for matches, mismatches, gap-open, and gap-extend. Critically, it sets the match score to zero (a = 0) and formulates the computation in terms of wavefronts—sets of "furthest-reaching" points along each DP diagonal for a given alignment score s. Intuitively, rather than filling a two-dimensional matrix, WFA advances a one-dimensional front of candidate offsets across diagonals, increasing the score in small steps and extending along exact matches whenever possible. This design leads to a time complexity O(n*s)

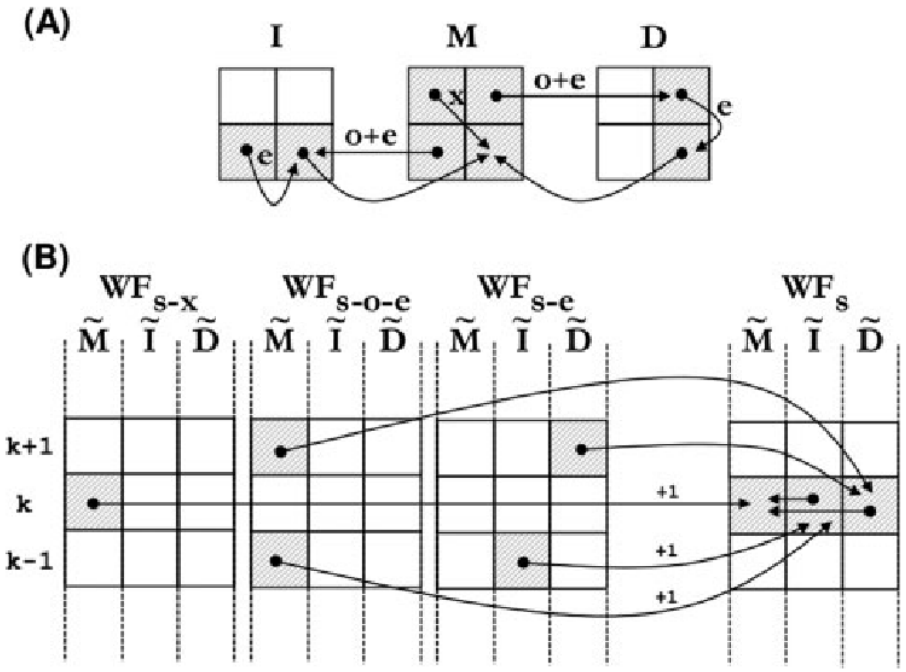


Figure 2.7: Data Dependencies in WFA. A) When computing DP cells. B) When computing wavefronts [32]

for sequences of length n with optimal alignment score s, yielding dramatic improvements when the error rate is moderate and s is much smaller than n.

At the heart of WFA are the notions of diagonals and furthest-reaching (f.r.) points. For strings q and t with indices v and h, diagonal k is defined by k=h v. For each score s and diagonal k, WFA maintains the largest offset reached on that diagonal under score s for each of the three SWG states: match/mismatch (M), insertion (I), and deletion (D). These offsets, encode how far the algorithm has progressed along each diagonal without storing the full DP scores. The algorithm begins with a trivial wavefront at score s = 0 on the main diagonal and iteratively constructs higher-score wavefronts by considering the only events that can increase the score under gap-affine penalties: a mismatch, opening a gap, or extending a gap. For a given s, the new offsets are computed from previously built wavefronts at scores s x, s o e, and s e through simple max-plus recurrences that select, for each diagonal, the predecessor state that yields the furthest reach. After these transitions, WFA performs an aggressive "extend" step that consumes all consecutive matches along each diagonal in constant-time word-level batches, thanks to bit-parallel comparisons over packed characters. This extend step is central: because matches do not increase the score, the front can surge forward cheaply across long identical stretches, compressing what would be many DP cell visits into one operation per word of sequence.

The algorithm proceeds by alternating two phases for increasing s: extend all current furthest-reaching points along exact matches, then generate the next

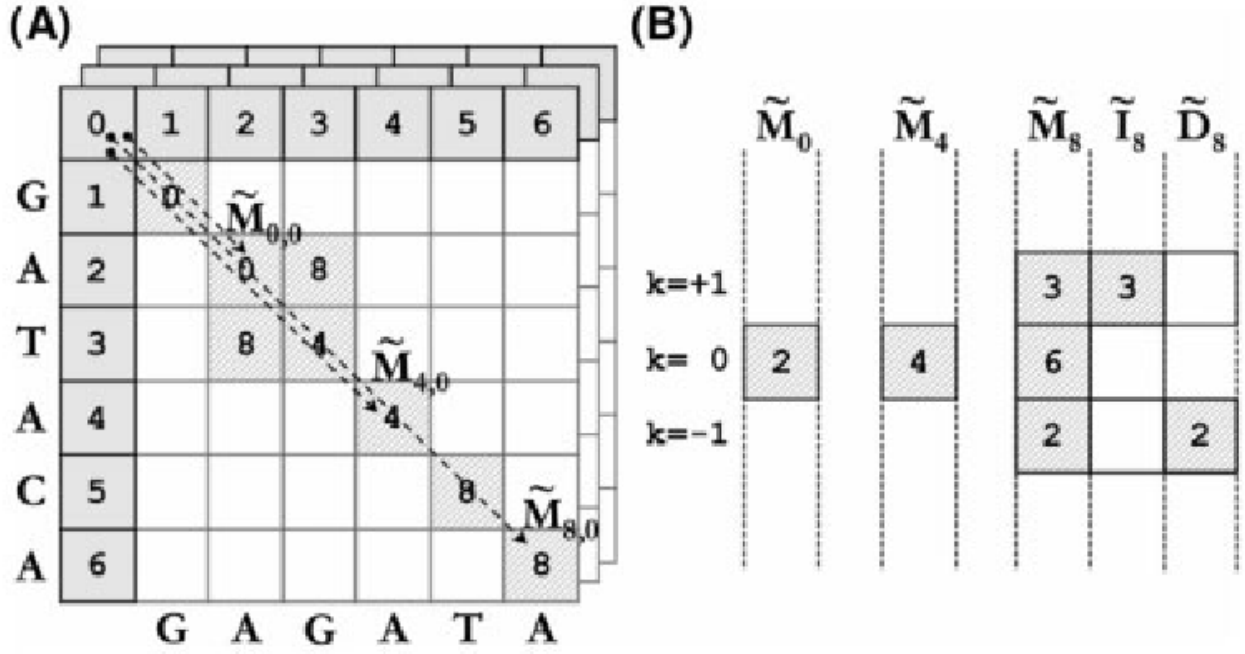


Figure 2.8: Example of WFA Algorithm. A) The DP matrix, B) The offset arrays per wavefront[32]

wavefront via the affine transitions. At each score, WFA checks whether any diagonal's offset has reached the terminal cell (n, m). The first score s for which this occurs is provably the optimal alignment score under the chosen penalties. Importantly, WFA's proof of optimality leverages the property that the constructed furthest-reaching points at score s dominate all other paths with the same score; if a strictly further point existed on the same diagonal, it would have been generated by the same recurrence and subsequent match extension, contradicting maximality. This "frontier optimality" ensures that the earliest wavefront reaching (n, m) corresponds to the globally optimal path. Once the terminal cell is reached, WFA performs backtracking not through a full DP matrix but through the sequence of wavefront offsets. The traceback follows, in reverse, which predecessor state and diagonal produced each furthest-reaching point and uses differences in offsets to infer contiguous runs of matches between successive scored events. For example, if an offset at score s on diagonal k originated from an offset at score s-x on the same diagonal, the difference encodes exactly how many matches follow the mismatch event. Because only a thin history of recent wavefronts is needed to reconstruct predecessors, backtracking avoids materializing the quadratic matrix and retains the algorithm's succinct memory footprint.

Memory usage in WFA scales with the span of the active wavefronts. As the score increases, the set of diagonals that a wavefront could occupy grows roughly linearly with s, so storing all wavefronts naïvely can require $O(s^2)$ space. In practice, WFA stores only the few wavefronts necessary for both forward pro-

gression and later traceback, and it encodes offsets in compact integer widths. For many workloads, 16-bit integers suffice, and even 8-bit integers are adequate for short Illumina-like reads. This compact representation, together with predictable access patterns, also enables compilers to auto-vectorize the core transitions, eliminating the need for architecture-specific SIMD intrinsics while still harvesting instruction-level parallelism. The extend phase benefits from bit-parallel block comparisons over packed characters, which typically terminate in a single iteration unless long runs of matches are present; in that case, the algorithm progresses even faster.

A salient strength of WFA is that its performance is governed by the optimal score s rather than by n × m. When error rates are moderate, s grows roughly with the number and size of indels and mismatches, so O(n*s) can be orders of magnitude smaller than $O(n^2)$. This behavior is exactly what practical read mappers and long-read aligners need: work proportional to the actual divergence between sequences. Moreover, WFA's formulation is inherently portable and friendly to modern compilers; the simple recurrences over contiguous arrays of offsets allow transparent SIMD across diverse instruction sets without handwritten intrinsics, and the match-extension uses standard word-level operations. These engineering advantages stand in contrast to several classic gap-affine vectorizations, which often require bespoke kernels for each SIMD width and layout. Finally, the memory profile is low and controllable; narrow integer offsets and optional pruning keep footprint modest even when aligning tens of kilobases.

In practice, integrating WFA into a pipeline invites a few considerations. First, the gap-affine parameterization should reflect the application domain; penalties that strongly favor opening versus extending gaps will shape the dynamics of wavefront transitions and the realized score s. Second, for very high error rates or extreme structural discordance, s can approach n, and WFA's asymptotic advantage diminishes; however, in these regimes classical DP is also stressed, and WFA-Adapt can still rein in resource use. Third, memory and throughput can be tuned through integer width choices for offsets and through thresholds for adaptive reduction. Finally, because WFA computes an exact alignment with a compact provenance, it is straightforward to reconstruct CIGAR strings and to interoperate with downstream components that expect standard gap-affine semantics.

Taken together, the Wavefront Alignment algorithm offers a clean, general, and highly efficient alternative to classical DP for gap-affine global alignment. By recasting alignment as the progressive expansion of score-indexed wavefronts of furthest-reaching diagonal points, it ties computational effort to the true difficulty of the instance, leverages long exact matches with bit-parallel

extensions, invites portable auto-vectorization, and greatly reduces memory. With an optional, empirically robust pruning heuristic, it scales gracefully to the long, noisy reads produced by modern sequencers while preserving optimality when desired. These properties explain its strong empirical performance relative to state-of-the-art libraries and motivate its growing adoption within high-throughput genomics workflows.

Further optimizations can be implemented into WFA as by knowing an exact (or approximately exact) edit threshold E for a pair of sequences, and by using the lemma from Banded Smith Waterman that ensures that an optimal alignment is always between 2*E+1 diagonals, we can prune the DP matrix statically to only search those diagonals for alignment paths.

2.4 High Level Synthesis

High-Level Synthesis (HLS) [49] [50]converts algorithmic C/C++ (or SystemC) into application-specific RTL, letting designers work at a higher abstraction while still delivering high-performance hardware. In a typical heterogeneous system, the bulk of the application runs on a host CPU, and the compute-intensive kernels are compiled by HLS into RTL and then into an FPGA bitstream. The result is a custom accelerator that exploits the FPGA's massive spatial parallelism and fine-grained control over data movement to achieve strong performance, cost, and energy efficiency compared with traditional processors[51][5

The overall flow begins with a C-level description of the algorithm. HLS provides fast functional verification at this level, so correctness can be established before committing to hardware. The C code is then synthesized to RTL. During synthesis, the tool applies default optimizations and also honors user-specified constraints and directives to shape the resulting micro-architecture. The primary outputs are Verilog/VHDL (and often SystemC) suitable for downstream logic synthesis, place-and-route, and bitstream generation. Post-synthesis, RTL co-simulation can be used to check that the hardware's behavior matches the C model. Finally, the generated RTL is packaged as an IP block for system integration.

In the background, HLS first compiles the functional specification into a control/data flow graph (CDFG) that makes data and control dependencies explicit. Three intertwined steps then determine the hardware: allocation (what types/quantities of resources are available), scheduling (which operations execute in which cycle, possibly chained or in parallel, while meeting timing and user constraints), and binding (which specific resource implements each scheduled operation). Choices in allocation and binding affect schedulability and vice-versa, so the tool iterates across these decisions to converge on a feasible design. Performance and area are driven by a small set of metrics that appear

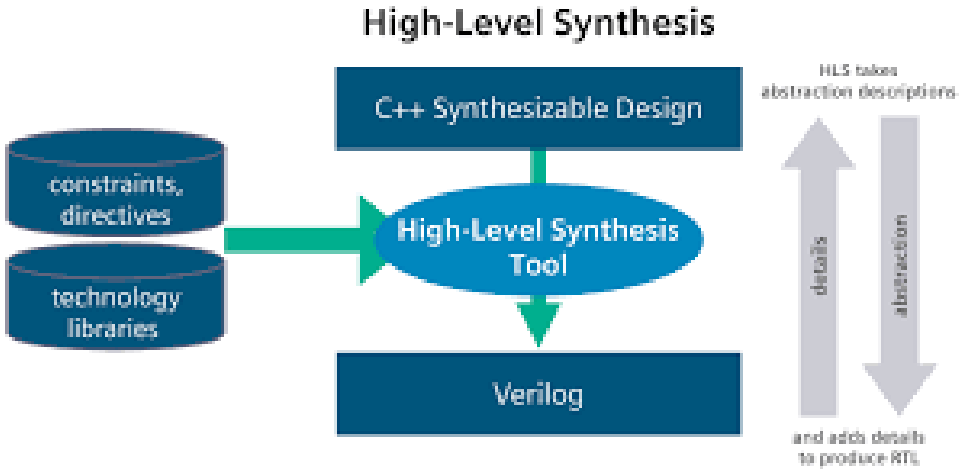


Figure 2.9: HLS Workflow [49]

in the synthesis report. Area reflects the consumption of FPGA resources such as LUTs, registers, BRAMs, and DSPs. Latency is the number of cycles to produce all outputs for one invocation of a function. Initiation interval (II) is the number of cycles between starting successive invocations—an II of 1 indicates a throughput-optimized pipeline that can accept new inputs every cycle. After examining these metrics, designers refine the micro-architecture with HLS directives and re-synthesize to approach the desired area–performance point.

Directives can be applied to functions, loops, arrays, or regions to expose and exploit parallelism. Pipelining reduces the II by overlapping loop iterations or intra-function operations. Dataflow enables task-level pipelining across producer/consumer functions and loops, so independent stages run concurrently. Inlining removes function boundaries to enable more aggressive logic optimization. Loop unrolling replicates the loop body to create parallel operators. Memory pragmas are crucial because arrays map to on-chip memories, typically with limited ports. Partitioning and reshaping adjust array organization to increase parallel access and remove BRAM bottlenecks, while array mapping can combine arrays to reduce memory footprint when bandwidth is not the limiter. Used together, these controls let the designer trade area for throughput and latency in a principled way.

HLS is particularly effective for computational genomics, where pipelines consist of multiple streaming stages with abundant parallelism and predictable access patterns. In seed-and-extend alignment, for example, seeds are generated and filtered, candidate locations are extended and scored, and the best alignments are traced back to produce compact encodings (e.g., CIGAR). HLS matches this structure well: dataflow can stream reads through seeding, filtering, alignment, and traceback kernels with on-chip FIFOs; loop pipelining and unrolling expose fine-grained parallelism within each stage; and careful ar-

ray partitioning supplies the multi-port memory bandwidth needed for parallel comparisons[53][54]].

Alignment itself—whether dynamic-programming based (e.g., Smith–Waterman, affine-gap variants) or wavefront-based—benefits directly from HLS-driven micro-architectures. Systolic or semi-systolic arrays map DP recurrences onto processing elements arranged along anti-diagonals or bands, achieving an II of 1 when BRAM and routing permit. Banded implementations exploit the observation that the optimal path stays near the main diagonal, reducing memory and compute while retaining exactness within a known edit threshold. HLS directives can unroll across the band width and pipeline across anti-diagonals, while array partitioning provides concurrent access to reference/read symbols and DP states. For wavefront alignment, diagonal-major data layouts and diagonally partitioned memories support parallel comparator arrays that extend matches, detect the first mismatch, and then advance the wavefront; static pruning can be realized by predicating updates outside the admissible diagonal window, saving both cycles and memory bandwidth. In all cases, 2-bit base packing, on-chip buffering of hot DP tiles, and carefully chosen bit-widths for scores and penalties reduce area and improve timing.

Traceback can be integrated without sacrificing throughput by storing compact predecessor hints (e.g., 2-bit codes for match/substitution/deletion/insertion) in a rolling on-chip buffer or in selectively spilled BRAM tiers, enabling a second, streaming pass that reconstructs the alignment path. HLS facilitates this design pattern by letting the forward pass and traceback pass be separate dataflow stages, each independently pipelined, so the system sustains high throughput while controlling memory footprint.

In summary, HLS provides a productive path from algorithm to RTL for genomics workloads. By exposing pipeline and memory structure at the C level—and then steering allocation, scheduling, and binding with directives,designers can realize accelerators for alignment that approach hand-tuned RTL in throughput and efficiency, while retaining the agility to iterate on scoring schemes, pruning heuristics, and I/O formats as datasets and accuracy requirements evolve[55].

2.5 Versal VCK190 Evaluation Platform — Specification

Platform overview. The AMD Versal™ AI Core VCK190 evaluation kit targets high-throughput, low-latency acceleration. A dual-Arm® processing system orchestrates heterogeneous engines (AI Engines, programmable logic, and hardened I/O) over an integrated NoC. Compute-intensive kernels are implemented in the PL/AIE while control and non-critical code runs on the PS[56].

SoC at a glance

Device / Package	XCVC1902 (VC1902 family), speed grade -*2*, SEVSVA2197
Processing System	Dual Arm Cortex-A72 (APU) + Dual Arm Cortex-R5F (RPU)
AI Engines	400 AIE tiles
DSP Engines	1,968
Integrated DDRMC	4 controllers
Programmable NoC	High-bandwidth connectivity among PS/-PL/AIE/DDR

Programmable Logic (PL) resources (VC1902 class)

System logic cells	∼1,968 K
LUTs / Flip-flops	899,840 LUTs / 1,799,680 FFs
Block RAM (BRAM)	967 blocks (≈ 34.0 Mb total)
UltraRAM (URAM)	463 blocks (≈ 130 Mb total)
Distributed RAM	∼27.5 Mb
Max general I/O (device-level)	∼770 pins (board-dependent availability)

Board-level memory & storage

- 8 GB DDR4 UDIMM (via DDRMC), up to 3200 MT/s.
- 8 GB LPDDR4 (4×16 Gb), up to 3900 MT/s.
- Dual microSD sockets (kit includes card).

Expansion & I/O

- PCIe Gen4 x8 edge connector.
- Two FMC+ (VITA 57.4) sites (multi-lane GTY per connector).
- Networking: 1×QSFP28, 2×SFP28, 3×RJ-45 (tri-speed Ethernet).
- Video: HDMI in + HDMI out.
- Control/Debug: JTAG, USB-UART, QSPI boot, system controller (PM-Bus/telemetry), SYSMON.

Clocks, power, and physical

- On-board programmable clocks for DDR, PCIe, Ethernet/video, and timing sources.
- Power: 12 V input (AC adapter included); PMBus-managed rails with telemetry.
- Form factor: $\frac{3}{4}$-length PCIe card; example envelope $\sim$9.50 in (L) $\times$ 7.48 in (H).
- Operating range (typ.): 0 °C to +45 °C (storage: -25 to +60 °C).

Notes for genomic alignment workloads

- The mix of $\sim$900k LUTs, $\sim$34 Mb BRAM, and $\sim$130 Mb URAM supports deep tiling/buffering for banded dynamic programming (SW/affine) and wavefront alignment.
- URAM-backed score tiles with BRAM line buffers enable $\mathrm{II} = 1$ systolic/semi-systolic datapaths across anti-diagonals or fixed bands.
- Dataflow across PS $\rightarrow$ PL $\rightarrow$ AIE streams seeding/filters into extension and traceback; array partitioning/reshaping removes BRAM port bottlenecks.
- PCIe Gen4 and QSFP28/SFP28 sustain high-rate ingest from sequencers or host storage; FMC+ permits custom front-ends.

Chapter 3

Methodology

3.1 Methodology Overview

The main goal of this work is to explore the impact of pre-alignment filters on hardware-based accelerators for alignment. As previously described, profiling different aligners and datasets reveals heterogeneity in the number of edits observed in the final alignments. In most cases, the alignments exhibit small edit number and could be found using less resources and performing less operations than in the baseline scenario. In this thesis, we aim at leveraging pre-alignment filters based on the edit threshold to optimize FPGA-based alignment on two levels: (i) optimize the architecture of individual alignment units in terms of resource utilization and performance, (ii) create a throughput-optimized multi-accelerator architecture comprising aligners that support different edit threshold. This section describes a systematic way to achieve this.

First, we select simulated and real datasets and perform profiling with state-of-the-art aligner to study the alignments reported based on the number of edits observed. We then perform a literature search and identify state-of-the-art alignment algorithms that could benefit from the edit threshold information,i.e., algorithms that can exploit the edit threshold in a way to reduce their computational and spatial requirements. After identifying the algorithms, we explore their parallelization capabilities and leverage High-Level Synthesis tools and tuning knobs to implement hardware-based accelerated designs. Those designs leverage the inherent parallelization of the algorithms and incorporate the information for the edit threshold to further optimize for resource utilization and performance. We synthesize different versions of the aligners with different edit thresholds and create a pool of designs with different latency and resources utilization.

We then aim to create a high-throughput multi-accelerator architecture that comprises accelerators of different thresholds to align a given dataset with increased throughput and reduced resources. The number and type of the accelerators, i.e., in terms of edit threshold, are mainly defined by the edit distribution

of the datasets. The accelerators operate in parallel on suitable reads, i.e., reads with edits less or equal to the edit threshold of their assigned accelerator. With a help of a balancing script, given the resource estimation from synthesis and the workload for each number of edits, we greedily search for a configuration that has the minimum latency in aligning the whole dataset. Therefore, the balancing script decides on the cutoff edit thresholds that the aligners will have and how many of each type of aligner we will use.

The following subsections elaborate on each step of the methodology and give insights to the design process.

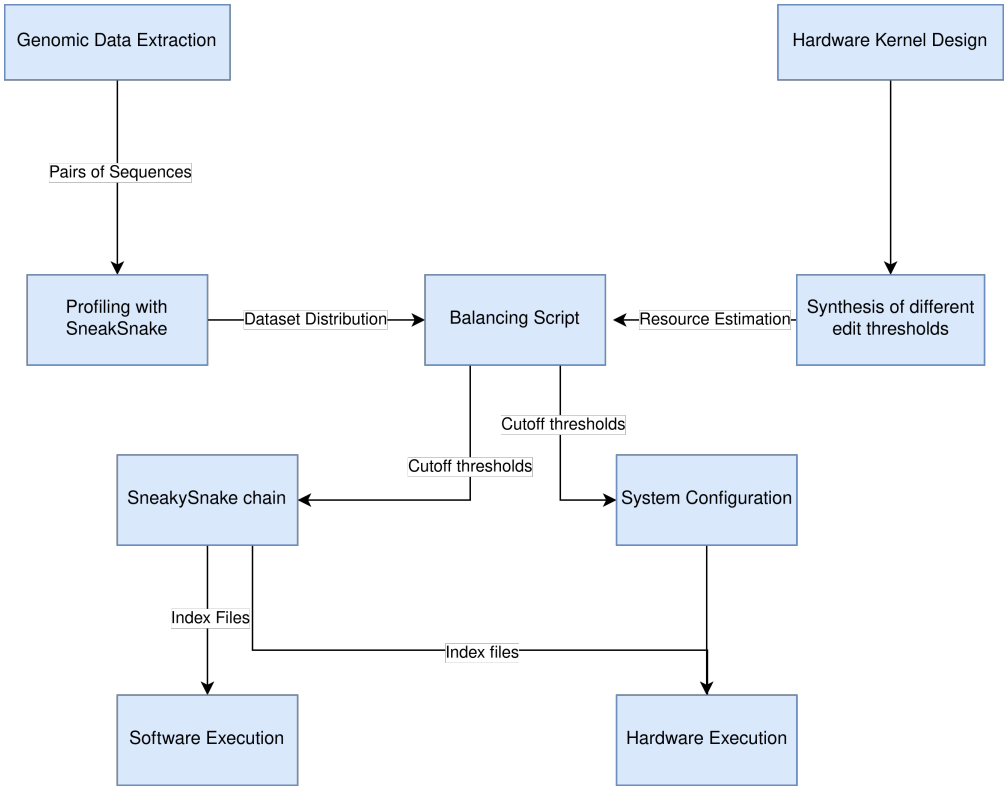


Figure 3.1: Proposed Workflow

3.2 Genomic Data Extraction

To evaluate the aforementioned kernels, genomic data are needed. For this particular thesis 100 base sequences are going to be used for read and reference sequences. Two dataset are going to be used. The first is a simulated dataset of 200000 pairs. .They are not taken from real human genome but with them corner cases can be usually explored.

The second dataset is part of the Illumina dataset. It comes from [57] and it consists of 4287748 pairs taken for chromosome 22 of the human genome. To extract this data first the reads where downloaded from official sources. Then, with the use of samtools, a software library used for alignment, these reads were

aligned to a reference genome and produced CIGAR strings. Finally, with a python script the CIGAR strings along with the reads were reverse engineered to produce 100 base pairs of read and reference sequences.

3.3 Profiling Genomic Data with SneakySnake

SneakySnake was used to filter the data into bins of different exact edit thresholds. For this procedure a series of filters was used, all with different edit thresholds ranging from 0 to 20. If a pair passed a filter, it was binned into a separate index file. If the pair was rejected, it was forwarded to the next filter. With that procedure we created filtered data distributions (Figure 3.2 and Figure 3.3).

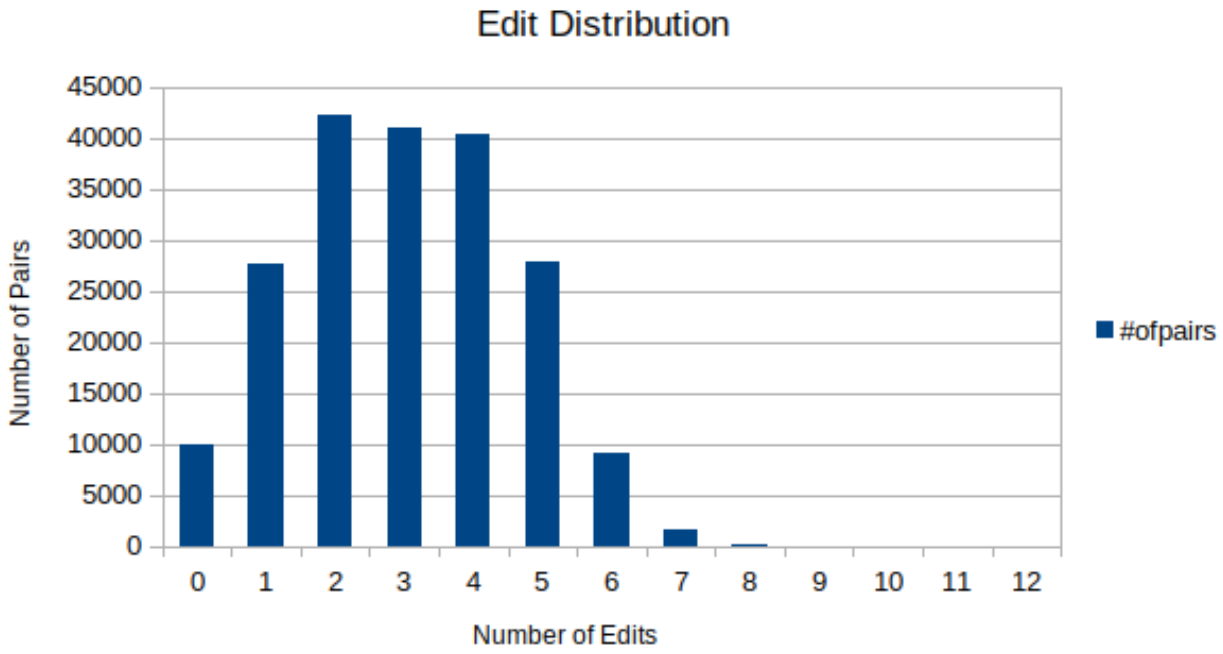


Figure 3.2: Simulated Dataset Filtered Distribution

SneakySnake can be extra helpful for the proposed architecture, as it can automatically align 0 edit pairs. That happens because the Neighborhood matrix of a SneakySnake module with 0 edits is only one line that is the result of a 1-1 comparison of the reference and read sequence. So for SneakySnake to pass that pair it must had only 0 accross that line, meaning a perfect match. This way SneakySnake aligns the pairs with 0 edits, if passed by a SneakySnake module with edit threshold E = 0. That means that when counting pairs for our proposed architecture, we can exclude those with 0 edits. That is not true when testing basic hardware implementations that do not use SneakySnake as a pre-filter.

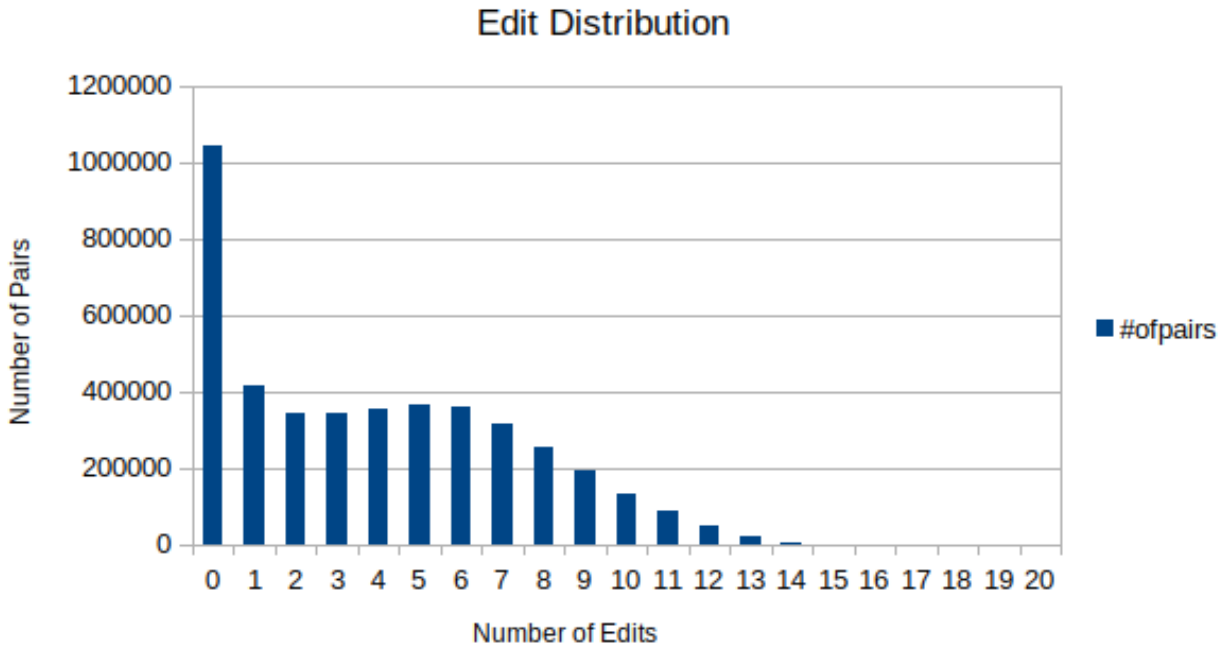


Figure 3.3: Illumina Dataset Filtered Distribution

In the simulated dataset, there are pairs with up to 10 edits and for the Illumina dataset up to 19. In the Illumina dataset given the large amount of pairs, pairs with more than 15 edits are very few and they are likely to be poor quality reads given from the sequencing machine. Despite that, they can be treated like viable reads and configurations will be considered to encompass them into the proposed designs.

It is important, not to neglect, that these are results form the SneakySnake filter. That means that in some cases the filter might have underestimated the number of edits and binned a pair into an index file of less edits than it should. That will impose a problem as the aligners will not align this pair properly (they are bound by edit threshold) and thus introducing an accuracy metric that will be discussed later.

3.4 SneakySnake on Hardware

SneakySnake can also be implemented on Hardware using HLS and can be greatly parallelized. SneakySnake executes on two steps. First, it creates a Neighborhood matrix as described in Chapter 2. Then it runs the actual algorithm on the matrix finding longest trails of zeros until it finds a 1 (obstacle). If it finds a number of obstacles greater than a user given threshold it rejects the pair, otherwise it accepts it. The hardware SneakySnake acts on those two procedures to accelerate the algorithm.

3.4.1 Neighborhood Parallelization

The neighborhood matrix is made of 2*E+1 horizontal lines , each LENGTH cells long , where LENGTH is the length of the pairs of sequences. Because each line is simply a comparison between the read and reference sequences ,each at different shifts, each line is fully independent of each other and so they can be computed in parallel (Figure 3.4). Also, because the comparisons are between apuint variables, the comparisons are bit-wise operations that can also be computed in parallel.

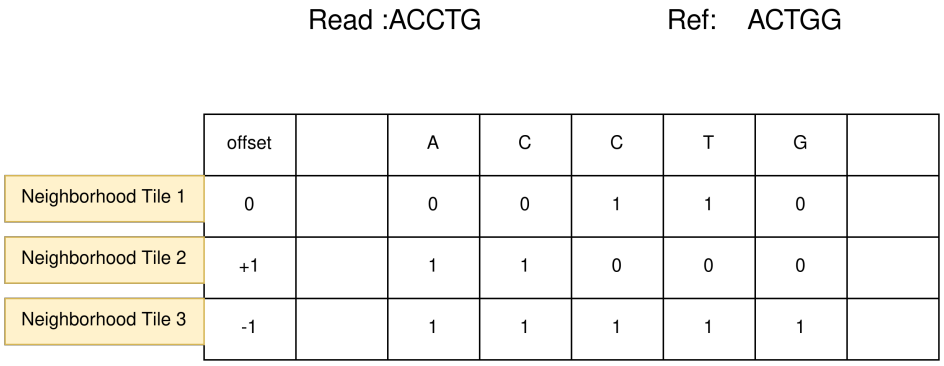


Figure 3.4: Neighborhood Matrix Parallelization

So, each neighborhood tile computes one line of the matrix and each tile contains LENGTH parallel 1-bit comparators (Figure 3.5).

The second optimization it the Tiling of the neighborhood matrix. After computing it, we tile the matrix into sub-matrices of length t. Then each sub-matrix, becomes a sub-problem. Each sub-problem can run in parallel (Figure 3.6).

Each After Neighborhood Tile consists of several Leading Zero counters and comparators. Each leading zero counter counts the leading zero of each line of he sub-matrix. The comparators, compare the results in a tree-like structure to find the biggest trail of zeros (Figure 3.7).

After this step all the edits found are added from each sub-problem and if the total surpasses the edit thresholds then the pair is rejected. Otherwise it passes the SneakySnake test.

The major problem with this implementation is the accuracy loss. Because each sub-problem is treated as new, it is very likely to underestimate the number of edits. An example is shown in Figure 3.8 , to visually interpret how that is possible. When we process the full matrix it is obvious that he SneakySnake finds 1 obstacle. When we tile the matrix it finds 0 obstacles. In this case it is not really a problem as the set edit threshold was E=1, but it shows how it can underestimate edits and lead to estimation that can really damage the

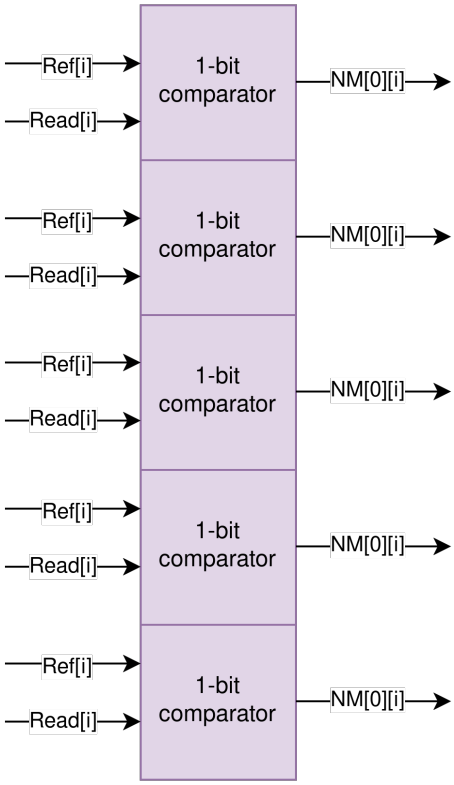


Figure 3.5: Neighborhood Tile 1

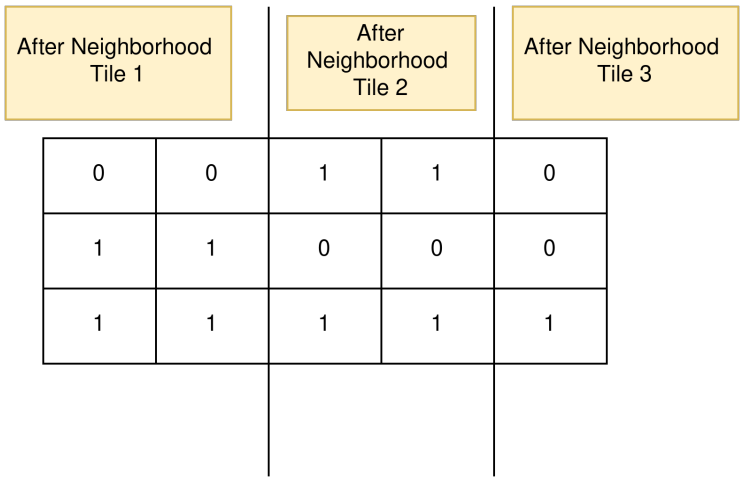


Figure 3.6: After Neighborhood Procedure

accuracy of our proposed design, as we depend on SnealySnake to make as accurate predictions as possible. As it will be mentioned in Chapter 5, it is important to develop an alternative hardware design for SneakySnake in order to exploit its parallel structure and include it in an end-to-end pipeline. Despite that, in the current work SneakySnake will serve as on off-line pre-processing step.

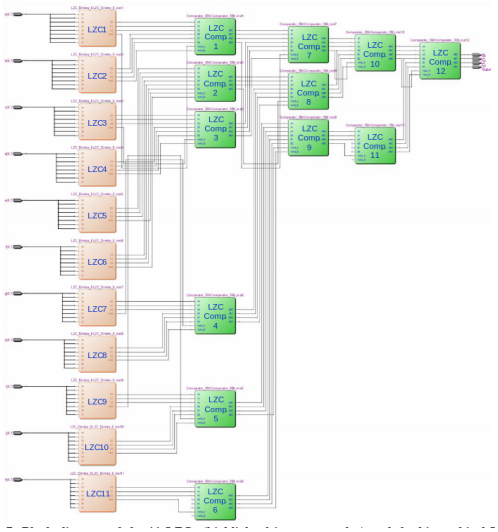


Figure 3.7: After Neighborhood Tile [7]

Read :ACCTG Ref: ACTGG

offset		A	C	C	T	G	
0		0	0	1	1	0	
+1		1	1	0	0	0	1 edit
-1		1	1	1	1	1	

offset		A	C	C	T	G	
0		0	0	1	1	0	0 edits
+1		1	1	0	0	0	
-1		1	1	1	1	1	

Figure 3.8: SneakySnake example. On the full matrix it finds 1 obstacle. On the tiles matrix it finds 0 obstacles

3.5 GenASM

3.5.1 Overview

The GenASM (Genome Approximate String Matching) algorithm is a bit-parallel method for approximate sequence matching, designed to efficiently align a short pattern (read) against a longer text (reference). Unlike classical dynamic programming algorithms that fill a two-dimensional matrix cell by cell, GenASM encodes the same computation using bit-vectors and simple bitwise operations, which can be performed in parallel and efficiently implemented on hardware such as FPGAs.

The main idea is to represent each position of the pattern as a bit within a machine word. Each bit indicates whether a partial match is possible at that position for a given number of allowed edits (mismatches, insertions, or deletions). The algorithm then updates these bit-vectors as it scans through the reference sequence, keeping track of which positions can still yield a valid alignment within the allowed error limit. First, a bit mask is created for each symbol of the alphabet (for DNA, A, C, G, T). Each mask stores the locations in the pattern where that character appears. For example, for letter A, the mask has a 0 bit at positions where A occurs and 1 otherwise. These masks allow quick comparison between the current text character and the pattern.
The algorithm maintains a small set of bit-vectors R[e], one for each allowed edit distance e (from 0 up to a user-defined limit E). Initially, all bits are set to 1, meaning that no match has been established yet.
The algorithm processes the reference text one character at a time. For each character T[j]:

The corresponding pattern mask M = MASK[T[j]] is retrieved.

Using bit shifts and logical operations, the algorithm updates the bit-vectors to reflect:

Match/Extend: advancing when characters match,

Substitution: allowing a mismatch,

Insertion and Deletion: handling gaps between the read and reference. These updates efficiently emulate the recurrence relations of the edit-distance dynamic programming algorithm.
After each character is processed, the algorithm checks whether the pattern could fully align ending at that position within any allowed number of edits. This is done by examining a specific bit (corresponding to the last pattern character) in each R[e]. If this bit indicates a valid alignment (a 0 in the proper encoding), a match is reported.
When traceback is enabled, small control bits are stored during updates to record the operation that led to each state (match, insertion, deletion, or sub-

stitution). Once a match is found, these bits are used to reconstruct the alignment path and generate a CIGAR string (a compact format that encodes the sequence of edit operations).

3.5.2 Exploiting SneakySnake

GenASM is an algorithm that inherently uses an upper edit threshold. The main bottleneck of GenASM is the storage utilization as it needs a rather large matrix to encode information for traceback. The size of the matrix depends on the upper edit threshold. To be precise, the 4-dimensional matrix used is of size: LENGTH * LENGTH * (EDITS+1) * 4 , where LENGTH is the length of the sequences used (e.g. 100bp), EDITS is the edit threshold and 4 is the number of possible operations (match, mismatch, insertion, deletion). So, for example a GenASM kernel with edit threshold E = 2, will need to store 100*100*4*3 = 120000 cells of that matrix, whereas if E = 10, will need to store 400000 cells. It is obvious that we can fit more kernels of lower edit thresholds into a platform, so we are going to exploit this information given by SneakySnake to instantiate many GenASM kernels of different sizes.

3.5.3 Optimizations

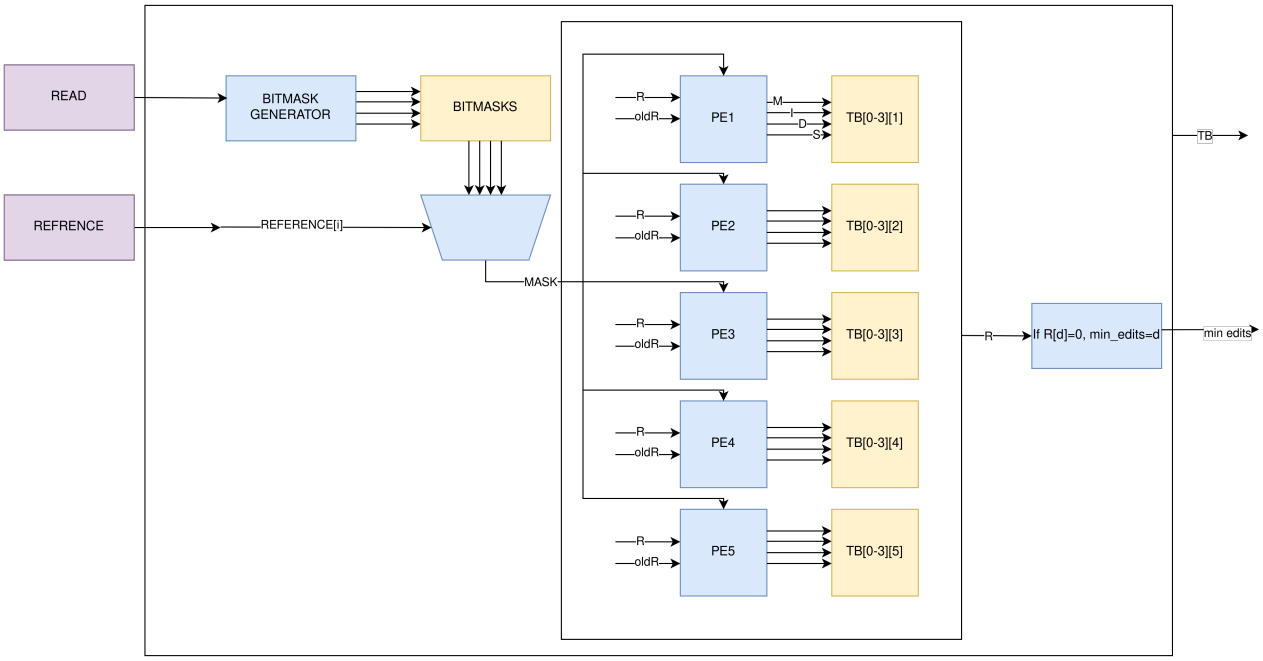


Figure 3.9: GenASM DC

The GenASM-DC algorithm runs a main for loop over all Text characters in the read sequence. For each character, it runs a for loop that goes over all possible number of edits and stores 4 values for each number of edits, one for each possible operation. There is also a data dependency between iterations

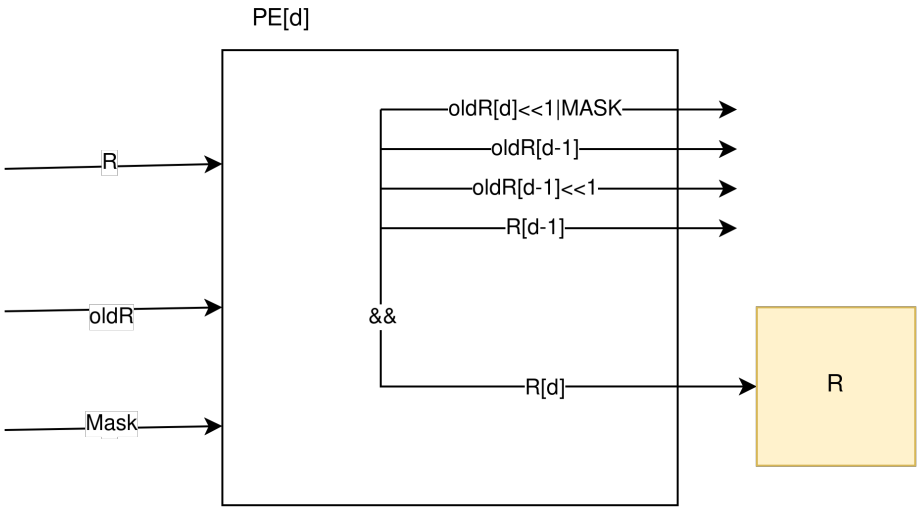


Figure 3.10: GenASM DC Processing Element

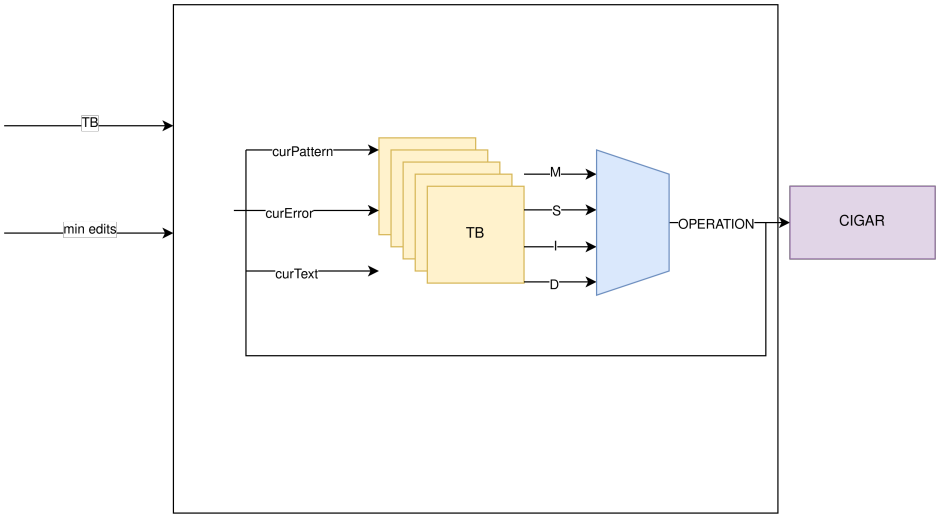


Figure 3.11: GenASM TB

as information from the previous character of the text sequence is required to compute values for the next character. So, by running this loop without directives it would take roughly LENGTH*(Edits+1)*4 cycles to complete the DC step.

To achieve the design in Figure 3.9 ,Figure 3.10 and Figure 3.11, we use a combination of pipeline, unroll and partition directives. We fully unroll the bitmask generation so that it happens all in 1 cycle , as every character of the bitmask can be computed in parallel. Then main loop has a latency of 2 cycles. The first cycle is used to compute the next state vectors and the second cycle is used to write the information into the TB matrix in the BRAM. We pipeline this loop so we can achieve II=1. Each nested loop inside the main loop is also unrolled so that all computations inside the PE can be executed independently. At first those directives did not have any result, as the TB matrix was stores

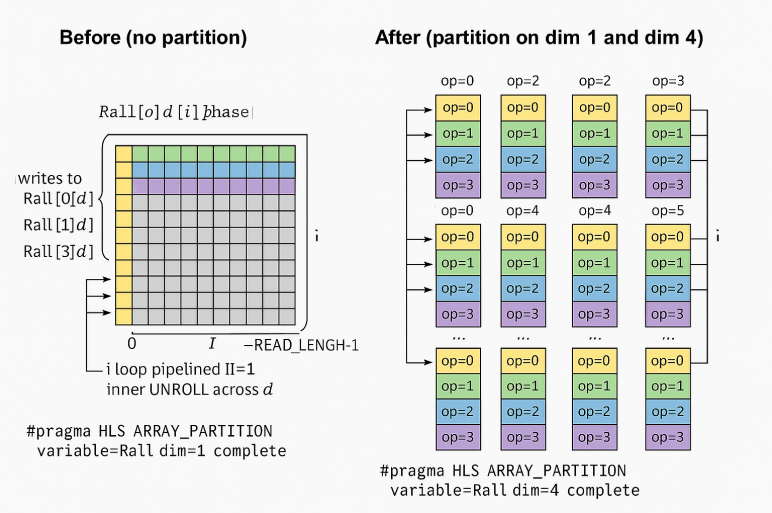


Figure 3.12: Array Partition on traceback matrix

as a whole inside one block. So, the array partition directive was used on the TB matrix to partition it along the dimension of size Edits+1 and along the dimension of the operations. This ways each PE (one for each edit) can write at the same time all four operations states to the TB matrix at the same time (Figure 3.12).

Below is the code that shows the main loop:

```
for(int i=READ_LENGTH-1;i>=0;i--){
    #pragma HLS PIPELINE II=1
        curChar = ref.range(2*i+1,2*i);
        curMask = masks[(int)curChar];

        for(int d=0;d<MAX_EDITS+1;d++){
        #pragma HLS UNROLL
            oldR[d]=R[d];
        }

        R[0] = (oldR[0]<<1) | curMask;
        Rall[0][0][i]=R[0];
        Rall[1][0][i]=all1;
        Rall[2][0][i]=all1;

        for(int d=1; d<MAX_EDITS+1; d++){
        #pragma HLS UNROLL
            ap_uint<READ_LENGTH> m   = (oldR[d]<<1)|curMask;
            ap_uint<READ_LENGTH> del = oldR[d-1];
            ap_uint<READ_LENGTH> ins = (R[d-1]<<1);
            ap_uint<READ_LENGTH> sub = (oldR[d-1]<<1);

            R[d] = del & (del<<1) & ins & m;
            Rall[0][d][i] = m;
            Rall[1][d][i] = del;
            Rall[2][d][i] = ins;
            Rall[3][d][i] = sub;
        }
    }
```

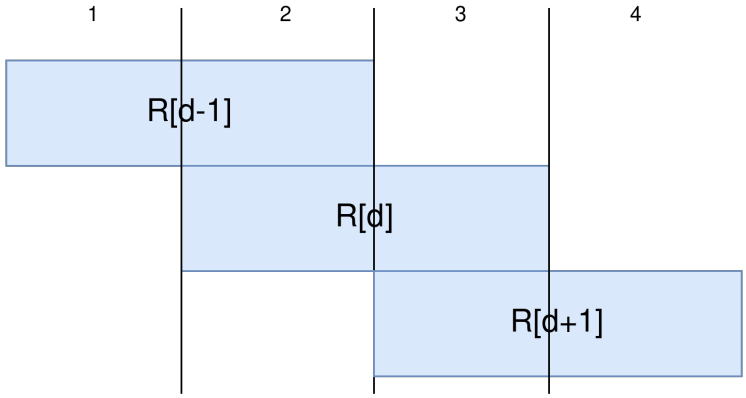


Figure 3.13: Main Array Pipelining. In the first cycle a part of the array is written, and in the following cycle it is read, achieving II=1.

The end product is a loop that takes LENGTH time instead of LENGTH*(EDITS+1)*4 to complete. The algorithm finishes with a small loop that finds the final minimum number of edits which is Edits+1 cycles long. So, the whole DC algorithm takes LENGTH+Edits+1 cycles to complete which is roughly equal to LENGTH (LENGTH»Edits). During traceback the TB algo-

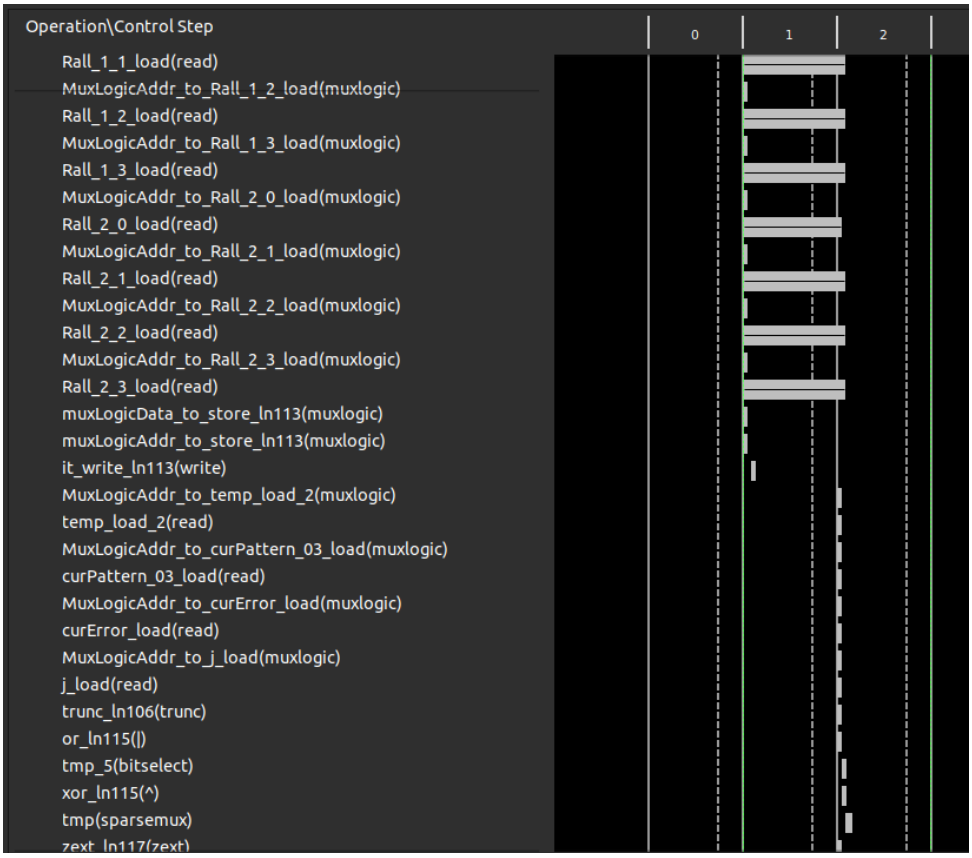


Figure 3.14: Traceback. All reads happen at first cycle. All mux operation happen on the second cycle

rithm starts form the end of the traceback matrix figuring out which operation took place for each text character. This requires that for each character four

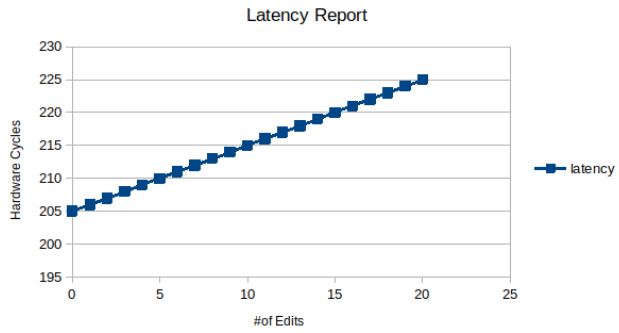
decisions must be taken. By partitioning the traceback array, the TB algorithm can access every cell needed. Each iteration takes 2 cycles to complete, 1 to read the matrix and one to take the decision (Figure 3.14). These two operations can be pipelined so TB can achieve II = 1 and so, max throughput. The optimizations used are shown on the following listing.

```
    template<int MAX_EDITS>
void GenASM_TB(ap_uint<READ_LENGTH> Rall[4][MAX_EDITS+1][READ_LENGTH],
               int min_edits,
               ap_uint<2*(READ_LENGTH+MAX_EDITS)> &CIGAR,
               bool &valid){
#pragma HLS INLINE off
    ap_uint<2*(READ_LENGTH+MAX_EDITS)> temp=0;
    int curPattern = READ_LENGTH-1;
    int curText=0;
    int curError = min_edits;
    int j=0;
    bool temp_valid=false;

    for(int it=0; it<(READ_LENGTH+MAX_EDITS); it++){
    #pragma HLS PIPELINE II=1
        if((curPattern>=0)&&(curError>=0)){
            bool step_valid = true;
            bool bit_match = !Rall[0][curError][curText][curPattern];
            bool bit_sub   = !Rall[3][curError][curText][curPattern];
            bool bit_del   = !Rall[1][curError][curText][curPattern];
            bool bit_ins   = !Rall[2][curError][curText][curPattern];

            if(bit_match){
                temp.range(j+1,j)=0; curPattern--; curText++;
            } else if(bit_sub){
                temp.range(j+1,j)=3; curPattern--; curText++; curError--;
            } else if(bit_del){
                temp.range(j+1,j)=2; curPattern--; curError--;
            } else if(bit_ins){
                temp.range(j+1,j)=1; curText++; curError--;
            } else{
            }
            temp_valid |= step_valid;
            j+=2;
        }
    }

    CIGAR=temp;
    valid=temp_valid;
}
```

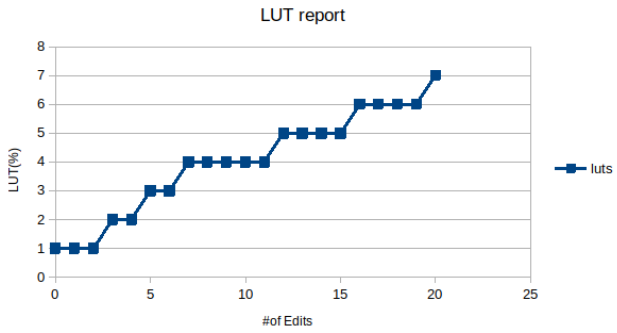
The final optimization is that ap uint types are used for all variables. Each base is encoded in 2 bits (00 = A, 01 = T, 10 = C, 11 = G) and each operation is also encoded in 2 bits (00 = match, 11 = mismatch, 01 = insertion, 10 = deletion).

By synthesizing the above design we obtain the following metrics about latency, bram and luts.

The main bottleneck of GenASM is the BRAM utilization as expected as the traceback matrix is expected to use a lot of BRAM and this is what will



(a) Latency



(b) LUTs(%)

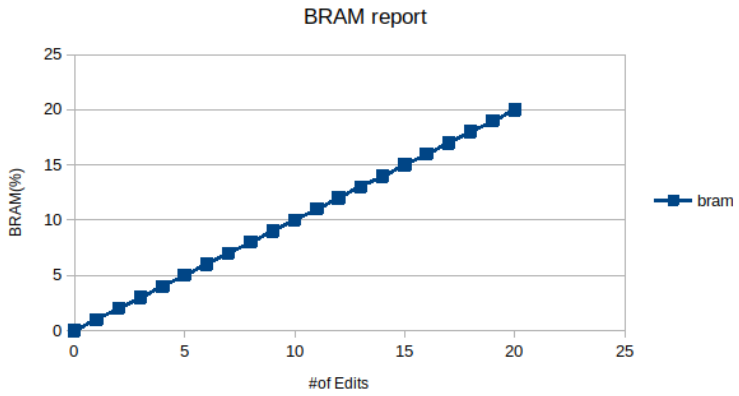


Figure 3.15: BRAM utilization

limit the design of running too many kernels at the same time.

3.6 Smith-Waterman

3.6.1 Overview

The Smith–Waterman algorithm is a classic dynamic programming method used for local sequence alignment, which identifies the most similar subsections between two sequences — typically a read and a reference segment in genomics. Unlike global alignment algorithms such as Needleman–Wunsch, Smith–Waterman focuses on finding the best local match, meaning that it can align subsequences even when the overall sequences differ significantly.

This algorithm is known for producing optimal alignments but is computationally expensive, which is why many modern accelerators and optimizations aim to improve its performance or reduce its memory footprint.
A two-dimensional scoring matrix H is created with dimensions (m+1) × (n+1), where m is the length of the query (pattern) and n is the length of the reference (text). All entries in the first row and first column are initialized to 0, since local alignment allows the alignment to start anywhere in the sequences.
A two-dimensional scoring matrix H is created with dimensions (m+1) × (n+1),

where m is the length of the query (pattern) and n is the length of the reference (text). All entries in the first row and first column are initialized to 0, since local alignment allows the alignment to start anywhere in the sequences.
The matrix H is filled cell by cell, where each cell H[i][j] represents the best alignment score for the prefixes P[0..i-1] and T[0..j-1]. The recurrence relation is:

```
    H[i][j] = max(
    0,
    H[i-1][j-1] + match_or_mismatch(P[i-1], T[j-1]),  // diagonal (match/
    substitution)
    H[i-1][j]   - gap_penalty,                        // deletion
    H[i][j-1]   - gap_penalty                         // insertion
)
```

As the matrix is filled, the algorithm keeps track of the highest score and its position (i*, j*). This represents the endpoint of the best local alignment between the two sequences. Once the maximum cell is identified, the traceback begins from (i*, j*) and proceeds backward:

Move diagonally for matches or substitutions,

Move up for deletions,

Move left for insertions. The traceback stops when a cell with score 0 is reached, marking the start of the optimal local alignment. This path is then converted into a CIGAR string describing the sequence of operations (M, I, D).

3.6.2 SneakySnake Exploitation

As it was explained in Chapter 2, the Smith Waterman DP matrix can be restricted, if the edit threshold is known or roughly estimated. If the edit threshold is E, only the main diagonal and E diagonals on each side are needed to find an alignment. This algorithm is also known as Banded Smith-Waterman. Banded Smith-Waterman when executing on CPUs has two major variations. Adaptive Banded Smith-Waterman and Static Banded Smith Waterman. The Adaptive algorithm starts its search around the main diagonals and if it needs it expands to further diagonals to always find an optimal alignment. The static algorithm is designed using a maximal band which is 2*E+3 diagonals wide, where is a user provided edit threshold. The adaptive algorithm is dynamic and not ideal for hardware design as it has many run time unknowns. Also, by using SneakySnake we can provide the algorithm with exact edit thresholds, so bands can easily defined. This way banded SW can be used to always find an optimal alignment using minimal search time and space. In the example of this thesis 100bp sequences are used for both read and reference. This means that the matrix has 199 anti-diagonals to compute. So, if we use a static band for E = 2, which is 2*2+3= 7 cells wide, 7*199 = 1393 cells must be computed. If E = 10, the band is 23 cells wide and 23*199 = 4577 cells must be computed

and stored. So, if SneakySnake can give good estimates for the edit threshold, Banded Smith Waterman can exploit this to instantiate different kernels of different sizes and achieve greater speedup.

3.6.3 Optimizations

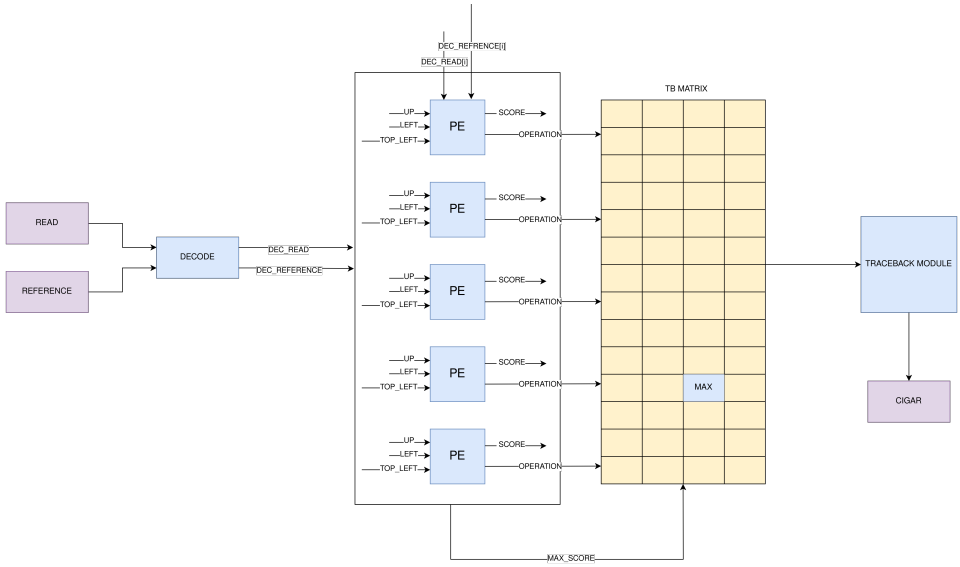


Figure 3.16: Banded Smith Waterman Hardware Architecture

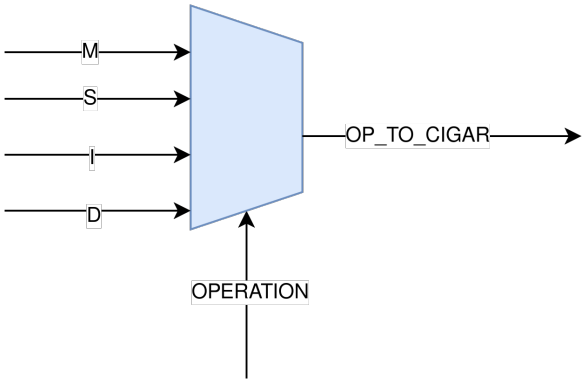


Figure 3.17: BSW-TraceBack

The main hardware optimizations done on the banded Smith Waterman can be summarized into 3 main axis.

- Diagonalization of the algorithm and use of systolic arrays
- Using custom types of variables to reduce area utilization
- Pre- and post-computing to avoid unnecessary lut replication

As it also true for regular Smith Waterman, the banded version can be diagonalized. Each cell of the matrix is dependent on the cell on top, the cell on the left and the cell in the upper left corner.

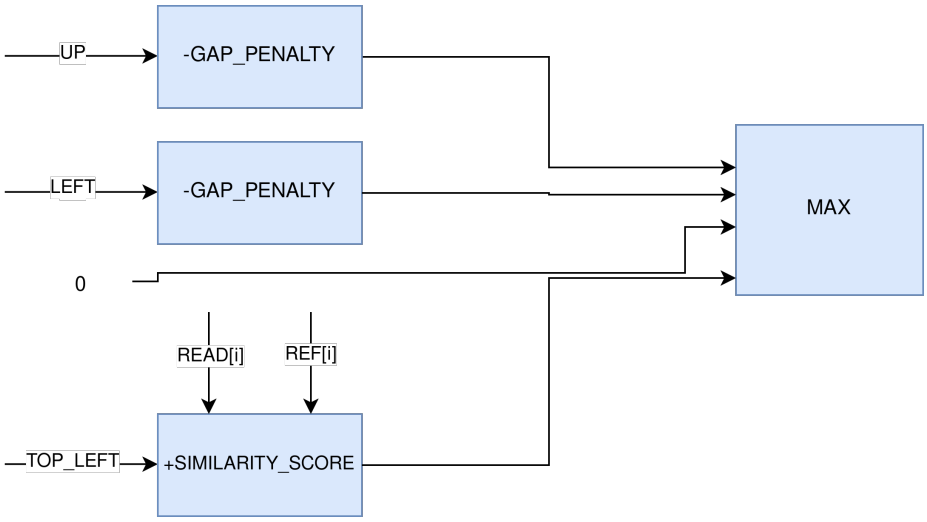


Figure 3.18: BSW-PE

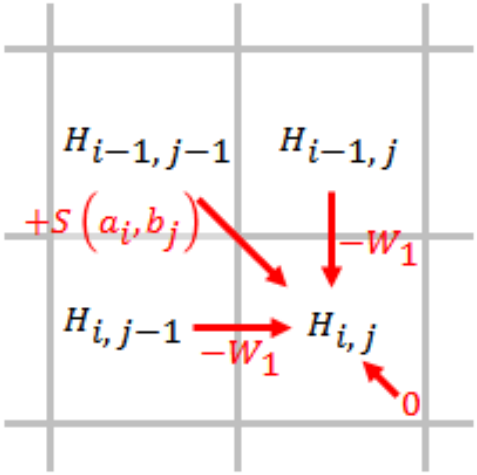


Figure 3.19: SW dependencies

This means that each cell of an antidiagonal can be computed at the same time, essentially reducing the time to run the main loop of the algorithm to O(n+m) where n and m are the lengths of the read and reference sequences. This is done by using systolic arrays. By creating process elements (PEs) that use as inputs the outputs of the top, left and topleft cells the algorithm can compute each independent cells (each antidiagonal) one cycle at a time. The main part of the PE is shown in the following listing.

```
        ap_int<10> diff_ji = (ap_int<10>)j - (ap_int<10>)i;
            bool in_core = in_mat &&
                           (diff_ji <= (ap_int<10>)BAND) &&
                           (-diff_ji <= (ap_int<10>)BAND);

            // Halo band: |j-i| <= BAND+1
            bool in_halo = in_mat &&
                           (diff_ji <= (ap_int<10>)(BAND+1)) &&
                           (-diff_ji <= (ap_int<10>)(BAND+1));

```

```
            // Neighbors from previous diagonals (gated to halo)
            score_t Hdiag = in_halo                ? H_prev2[idx]
     : Z;
            score_t Hup   = (in_halo && (idx+1<HALO_BANDW)) ? H_prev1[idx+1]
    : Z;
            score_t Hleft = (in_halo && (idx>0))  ? H_prev1[idx-1]
     : Z;

            // Bases from predecoded arrays (only if in halo)
            ap_uint<2> br = 0, bq = 0;
            if (in_halo) {
                br = rbase[(int)i];
                bq = qbase[(int)j];
            }

            // PE compute on halo; clamp out-of-core result to zero (score
    not stored)
            score_t    H_ij;
            ap_uint<2> dcode;
            sw_pe_linear_argmax(in_halo, Hdiag, Hup, Hleft, br, bq, H_ij,
    dcode);
            if (!in_core && in_halo) { H_ij = Z; /* keep dcode to guide TB
    if desired */ }

            H_curr[idx] = in_halo ? H_ij : Z;

```

Here Hdiag, Hup, Hleft are the values from topleft, top and left cells of the current PE, and i,j are the indexes of the matrix (where i - j is an antidiagonal). Finally, to achieve diagonalization all loops are internally unrolled and pipelined with the appropriate directives.

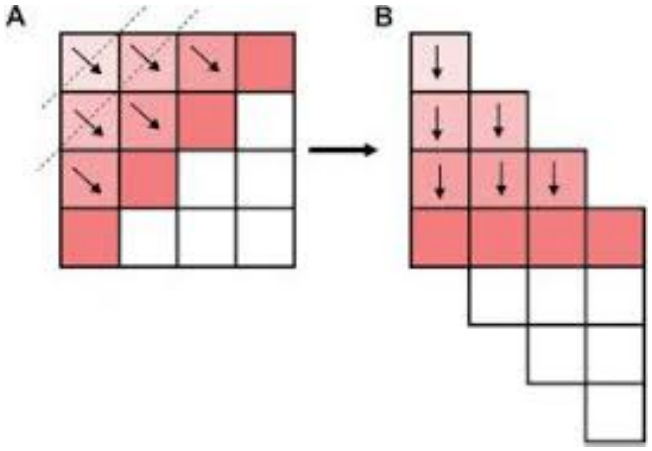


Figure 3.20: Diagonalization of SW

Then using apuints for all variables and especially encoding CIGAR operations and read and reference sequences as in the GenASM algorithm, the total area used is reduced.

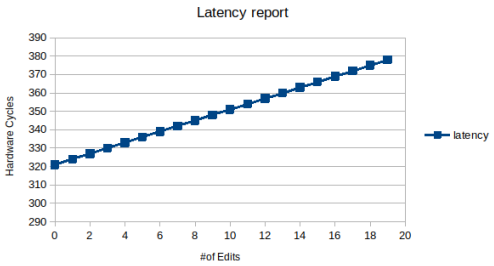
Lastly, instead of updating the max score to find the best cell at the end, each diagonal saves its personal best score and the comparison is done after the algorithm is complete introducing a small area and computational overhead. Otherwise, huge critical paths are created and the design misses timing. Also,

the bit-packed input sequences are decoded into small arrays so that no apuint slicing is taking place in each PE. This way the LUT utilization is almost dropped in half, enabling almost double the kernels to be instantiated.

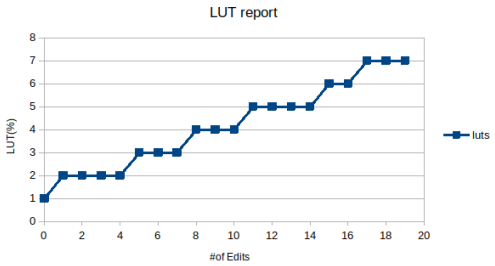
```
PRE-DECODING INPUT SEQUENCES
    DECODE_READ: for (idx7_t i=0; i<(idx7_t)LEN; ++i) {
#pragma HLS PIPELINE II=1
        int ii = (int)i;
        rbase[ii] = read_enc.range(2*ii+1, 2*ii);
    }
DECODE_REF: for (idx7_t j=0; j<(idx7_t)LEN; ++j) {
#pragma HLS PIPELINE II=1
        int jj = (int)j;
        qbase[jj] = ref_enc.range(2*jj+1, 2*jj);
    }
```

The latency of each kernel is almost the same and it is around 350 cycles, as it needs 200 cycles for the main loop, and 100 cycles for traceback, plus some overhead cycles for writing the CIGAR string and initialize some arrays.

These are the utilization results for edit thresholds ranging from 0 to 20.



(a) Latency



(b) LUTs(%)

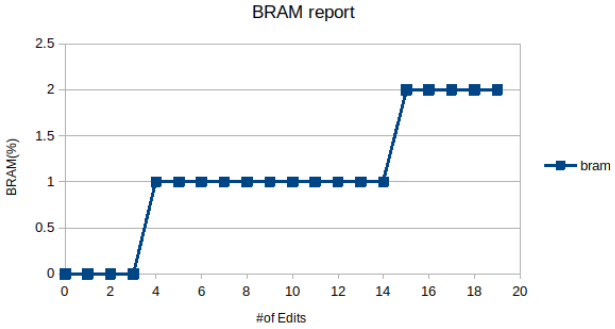


Figure 3.21: BRAM(%)

Here the main bottleneck that will limit the use of many kernels will be the LUT utilization and that is why it was needed to reduce its usage by unpacking the input sequences.

3.7 WFA

The WFA algorithm was proven a daunted task for this thesis, with no satisfiable results. That was because of two key reasons. First, it is a dynamic algorithm. The extend matches step of the algorithm is not pre-determined as to when it will find the next mismatch, running for an amount of cycles that range from 1 cycle up to the while length of a diagonal. Also, the intermediate scores are not determined either. For that reason, and in order to have static bounds for HLS synthesis, loop tripcounts are maxed at max length, or max score respectively, resulting in very large execution times. The second reason is that despite its large execution time, because of the high complexity computations needed for determining the next wavefront and traceback the resources are not minimum. So, it is not feasible to instantiate too many kernels to alleviate the high latency reports. In this particular case, the lack of flexibility HLS sometimes have , has proven detrimental to the realization of this algorithm for the current task. As implemented in [58] a more custom solution to this problem exists that targets WFA in FPGAs that writes its kernels into low-level VHDL in order to bypass the flexibility restrictions imposed by HLS.

WFA remains a great algorithm for CPU and GPU implementations that has benefits over other algorithms in these platforms. For FPGAs it is either reformed into a more static version that eventually resembles Banded Smith-Waterman, or written in more low-level languages like VHDL and verilog that allow for custom solutions.

The results form the aforementioned work for 100bp inputs gave x6 speedup in reference to the single-threaded cpu-wfa, by fitting 100 aligners in a single FPGA. That could be managed by cleverly mapping resources into the FPGA with VHDL. This is a great motivation for future work, to try to implement the banded-WFA algorithm on FPGA using hardware description languages and achieve greater speedups.

3.8 Multi-accelerator Throughput-Optimized Architecture

The proposed architecture is shown in Figure 3.22. The real and the simulated datasets are passed through a chain of SneakySnake filters. These data are cetegorized into bins according to their number of edits. These bins are streamed through FIFOs to the PL of our system. There, each FIFO feeds an accelerator custom to edit threshold that matches the bin the FIFO originated from.

The data in the used datasets are not equally distributed between the different number of reads in regard to edit threshold. If that was the case, we could instantiate an equal amount of aligners for each edit threshold as long we have

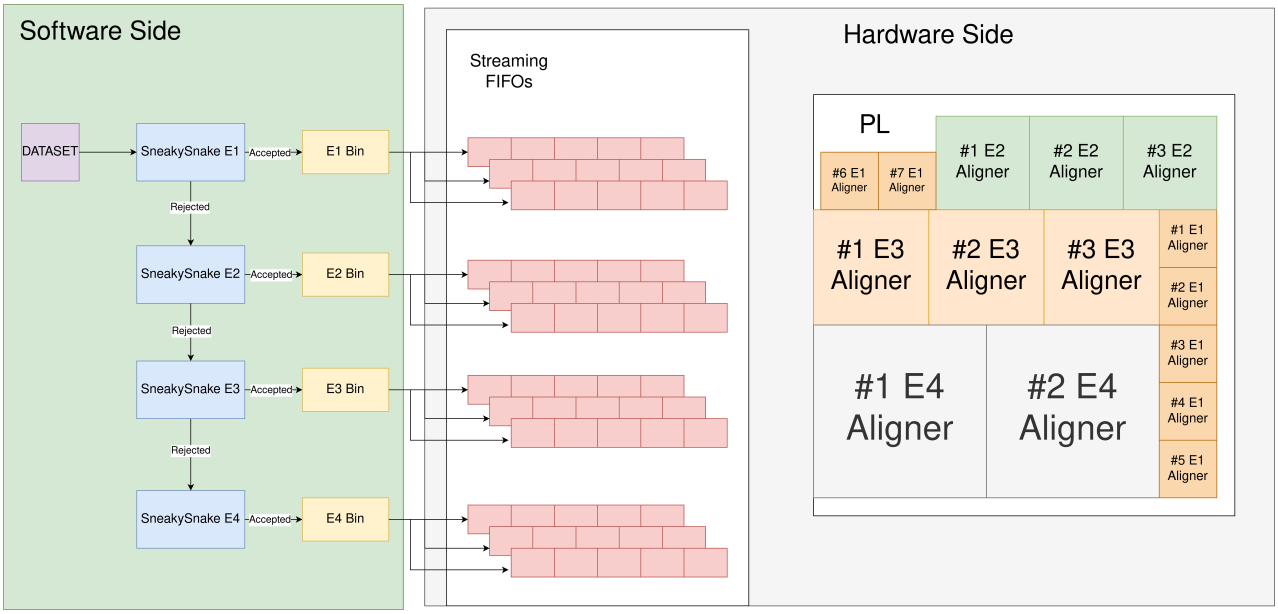


Figure 3.22: Proposed Architecture

resources. Because in all of our used algorithms the latency is roughly equal despite the difference in edit thresholds, the whole system of equally distributed accelerators would finish when the bin with the most pairs would finish, while the rest remain idle. This is a sub-optimal behavior and we propose a balancing heuristic.

By binning the data into ranges of thresholds, instead of individual exact thresholds, we can divide the workload between different classes of aligners by instantiating a -proportional to workload - number of them. If the workload for a range of threshold is X and the number of aligner instantiate for that workload is Y, we want X/Y to be the same for all classes of aligners. To achieve that, we created a python script that takes as parameters the number of workload, according to SneakySnake, for each exact threshold and the critical resource utilization for the particular kernel (BRAM for GenASM, LUTs for Banded Smith-Waterman). This script greedily tries all possible configurations and outputs the optimal configuration of aligners to be used by estimating their resource utilization.

We ensure this configuration is synthesizable, through the vitis-hls environment. If it has less than 100% utilization , it is a valid configuration. Otherwise, the python script must be restricted to a smaller budget and rerun the script.

Chapter 4

Experiments and Results

We have four configurations to realize:

- A GenASM configuration for the simulated dataset
- A GenASM configuration for the Illumina dataset
- A Banded-SW configuration for the simulated dataset
- A Banded-SW configuration for the Illumina dataset

All of the experiments were executed as follows:

- First, given the critical resource estimation (BRAM for GenASM and LUTs for Banded Smith-Waterman) a balancing script was executed to propose a combination of aligners of different edit thresholds.
- The proposed combination was synthesized in order to check, if it actually fits inside the FPGA. In case it fitted, the design was used to estimate speedup. Otherwise, the balancing script was run anew with a lower resource budget, until a suitable combination was found.
- Then SneakySnake was run on software to correctly bin the dataset into index files.
- Then the alignment algorithms were executed in software for each of the bins to extract the appropriate metrics (time in seconds and number of aligned pairs). Only the alignment and traceback parts were measured in terms of time.
- Finally, an estimate of cycles and time using synthesis was extracted to calculate the speedup of alignment

The goal is to run SneakySnake on software and the acceleration process on hardware. The speedup is computed by comparing the hardware version with the software version of the aligners. It is also important to calculate the speedup

our proposed design has in respect to a design that did not implement the above methodology and run a max threshold aligner.
The software experiments were executed on single thread Intel i7 CPU 1.8Ghz. For the hardware implementation, a 10ns clock was used with a clock uncertainty of 27% meaning that the design was estimated to run on 7.3ns, but to respect the fact that the hardware experiments were not executed on hardware the 10ns will be used to estimate speedup.
As mentioned in Chapter 2 the SneakySnake pre-filter can falsely accept sequences of higher edit thresholds. In this proposed system, these sequences are not aligned, if the aligner they were assigned to, cannot support them. This introduces an accuracy metric that essentially shows how many pairs of sequences SneakySnake falsely accepted and it is the same number as the pairs of sequences that were not aligned properly, or not aligned at all by the proposed design.

4.1 GenASM with simulated dataset

After running the balancing python script, the following combination of GenASM aligners was proposed.

Edit Threshold	Aligners	Workload	Workload/Aligner
1	5	27710	5542
2	5	42141	8428
3	5	41061	8212
4	5	40321	8064
5	4	27899	6974
10	2	10941	5470

Table 4.1: GenASM configuration with simulated dataset

The Workload is the number of pairs to be aligned for each number of edits. The Workload/Aligner is the number of pairs that each individual aligner will have to align in the final design. Because for each number of edits the latency is roughly the same, it is important to balance the Workload/Aligner metric for each aligner to achieve minimum latency. The balancing script tries to minimize the maximum Workload/Aligner metric given a resource constraint. The above configuration of aligners has a total resource estimation of:

BRAM(%)	LUTs(%)	FFs(%)
99	87	6

Table 4.2: Total Resources of GenASM on simulated dataset

After binning the data to the proposed bins with the help of SneakySnake, the GenASM algorithm with an appropriate edit threshold was executed for

each bin. The results are the following, in respect to time and accuracy. Time(s) for each bin is the individual time each run needed to execute. Total time is the sum of all individual times.

Bin	Time(s)	Accuracy
1	0.33601	0.51
2	0.675876	0.65
3	0.853457	0.58
4	1.016036	0.60
5	0.82732	0.69
6-10	0.586214	0.99
Total	4.294913	0.63

Table 4.3: Software Metrics

For hardware estimation, the maximum Workload/Aligner was used to estimate how many times the hardware kernels must be executed. To ensure that data transfer overhead between runs of the same kernel was minimized, an axi-stream interface was used to stream data from a FIFO queue to the kernels. The max Workload/Aligner is 8428 on the aligner of edit threshold E=2 which needs 207 cycles to fully execute. So, given a 10ns clock period time, the hardware needed to fully run was 0.0174 seconds, resulting in a x246 speedup in respect to software alignment.
The main issue with this configuration is the accuracy. 63% accuracy means that 37% of the pairs were not aligned. This happens for two reasons. First, SneakySnake seems to underestimated many of the pairs' edit thresholds. Secondly, the GenASM algorithm works with a very strict rule, that if a pair of sequences has more edits than its edit threshold, GenASM will automatically reject it. That is not happening with algorithms like Banded-Smith Waterman because in some cases they can find an alignment with more edits than the reserved band as it will be discussed in the next section.
To solve the aforementioned problem there are two solutions. Either keep track of the indexes that were not aligned and realign them in a later step, or raise the edit threshold of all the GenASM aligners by some amount. Running the same balancing script but for greater thresholds that bin pairs into categories of higher edits. The following results were recorded (GenASM+X means that the pairs were binned X edits higher than SneakySnake predicted).

Aligner	Accuracy	Software Time(s)	Hardware Time(s)	Speedup
GenASM+1	0.72	4.84	0.023	x211
GenASM+2	0.94	6.72	0.029	x227
GenASM+3	0.99	6.71	0.03	x220

Table 4.4: GenASM+X Aligners

Edit Threshold	Aligners	Workload	Workload/Aligner
4	2	27710	13855
5	3	42141	14047
6	3	41061	13687
7-8	5	68220	13644
9-10	1	10941	10941

Table 4.5: GenASM+3 configuration with simulated dataset

A Hardware Time(s)-Accuracy (total for the whole architecture) plot for the four designs is shown in Figure 4.1:

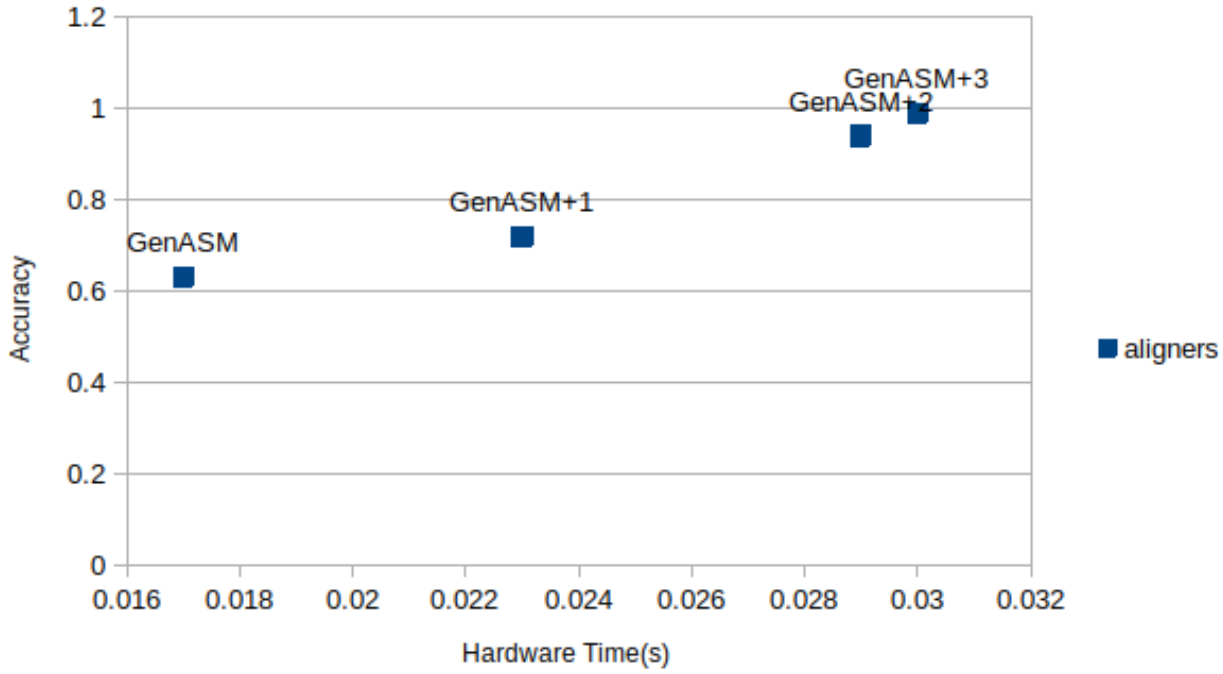


Figure 4.1: GenASM aligners

Because we want to maximize accuracy and minimize time, the Pareto optimal aligners are the original GenASM and the GenASM+3. Because in genomics accuracy is very important and the difference in time is negligible, GenASM+3 is the best aligner for the simulated dataset. This final configuration as calculated by the balancing script has 11941 max Workload/Aligner with a resource utilization as shown in the following table.

BRAM(%)	LUTs(%)	FFs(%)
93	56	4

Table 4.6: GenASM+3 Resources

Another important comparison, is the speedup between the binned version proposed by this thesis against a configuration of GenASM aligners that their edit threshold was not exploited by filtering. Because the simulated dataset

has an upper edit threshold of 10 edits, the FPGA was filled with as many GenASM aligners it could fit with an edit threshold of ten. The result was, that 10 aligners could fit with a re source utilization of:

BRAM(%)	LUTs(%)	FFs(%)
99	49	3

Table 4.7: Baseline Genasm Resources

That means, that the Workload/Aligner is 20000 and the Hardware Time(s) is 0.043s yielding a x250 speedup in respect to a software implementation running at 10 edit threshold for the whole dataset. The results that compare the GenASM+3 aligner against the base GenASM are shown in the following table:

Aligner	Hardware Time(s)	Speedup(in respect to Base)	Accuracy
Baseline GenASM	0.043	1	0.999
GenASM+3	0.03	1.6	0.99

Table 4.8: Comparison between baseline GenASM and GenASM+3

As shown above, there is a x1.6 speedup between the base version and the proposed aligner with a very slight loss in accuracy (only 0.009).

4.2 Banded Smith-Waterman with simulated dataset

The same reasoning was used for this experiment. After using a balancing script but with LUTs as critical resource instead of BRAM the following configuration of Banded Smith-Waterman kernels was proposed.

Edits	Aligners	Workload	Workload/Aligner
4	26	151233	5816
6	5	28899	5579
10	2	10941	5470

Table 4.9: Banded Smith-Waterman(BSW) configuration with simulated dataset

The above configuration of aligners has a total resource estimation of:

BRAM(%)	LUTs(%)	FFs(%)
27	98	4

Table 4.10: Total Resources of BSW on simulated dataset

After binning the data to the proposed bins with the help of SneakySnake, the Banded Smith-Waterman algorithm with an appropriate edit threshold was

Bin	Time(s)	Accuracy
1-4	0.844	0.997
5-6	0.337	0.999
7-10	0.0224	1
Total	1.20	0.998

Table 4.11: Software Metrics for BSW

executed for each bin. The results are the following, in respect to time and accuracy.

For hardware estimation, the maximum Workload/Aligner was used to estimate how many times the hardware kernels must be executed. To ensure that data transfer overhead between runs of the same kernel was minimized, an axi-stream interface was used to stream data from a FIFO queue to the kernels. The max Workload/Aligner is 5816 on the aligner of edit threshold E=4 which needs 333 cycles to fully execute. So, given a 10ns clock period time, the hardware needed to fully run was 0.019 seconds, resulting in a x62 speedup in respect to software alignment.
A major benefit of Banded Smith-Waterman is the accuracy in respect to SneakySnake. Banded Smith-Waterman does not inherently reject pairs that have more edits than its band. Banded Smith-Waterman only rejects pairs that attempt to search the DP matrix in areas that extend further than its band. That happens only if the pair has many insertions or deletions. Insertions make the algorithm search cells further from the main diagonal on one side and deletions to the other. So, it needs many gaps of the same type, in order not to be able to align two pairs. That results in great accuracy and there is no need to explore +X aligners as in GenASM algorithm.

To compare the binned version proposed by this thesis against a configuration of Banded Smith-Waterman aligners that their edit threshold was not exploited by filtering. Because the simulated dataset has an upper edit threshold of 10 edits, the FPGA was filled with as many Banded Smith-Waterman aligners it could fit with an edit threshold of ten. The result was, that 20 aligners could fit with a resource utilization of:

BRAM(%)	LUTs(%)	FFs(%)
27	96	6

Table 4.12: Baseline BSW Resources

That means, that the Workload/Aligner is 10000 and the Hardware Time(s) is 0.035s yielding a x68 speedup in respect to a software implementation running at 10 edit threshold for the whole dataset. The results that compare the Banded

Smith-Waterman aligner against the base BSW are shown in the following table:

Aligner	Hardware Time(s)	Speedup(in respect to Base)	Accuracy
Baseline BSW	0.035	1	1
BSW	0.019	1.84	0.998

Table 4.13: Comparison between Base BSW and proposed BSW

As shown above, there is a x1.84 speedup between the base version and the proposed aligner with a very slight loss in accuracy (only 0.002).

Comparing all the aligners for the simulated dataset, we conclude these results.

Aligner	Accuracy	Hardware Time(s)
GenASM	0.63	0.017
GenASM+1	0.72	0.023
GenASM+2	0.94	0.029
GenASM+3	0.990	0.028
BSW	0.998	0.019

Table 4.14: Comparison between the two aligners for the simulated dataset

We see that Banded Smith-Waterman performs better than GenASM in terms of latency and accuracy, even so we tried to optimize GenASM. Although some GenASM variations have better latency than BSW, they have very poor accuracy which make them not valid candidates for a good aligner. In every case, the proposed design performs better at alignment than the base algorithms.

4.3 GenASM with real dataset

The same reasoning was used for this experiment. After using a balancing script but with BRAM as critical resource, the following configuration of GenASM kernels was proposed.

Edits	Aligners	Workload	Workload/Aligner
1	2	413460	206730
3	3	689054	229684
5	3	722505	240835
7	3	675909	225303
9	2	446219	223109
15	2	164311	164311

Table 4.15: GenASM configuration with real dataset

The above configuration of aligners has a total resource estimation of:

BRAM(%)	LUTs(%)	FFs(%)
97	70	5

Table 4.16: Total Resources of GenASM on real dataset

After binning the data to the proposed bins with the help of SneakySnake, the GenASM algorithm with an appropriate edit threshold was executed for each bin. The results are the following, in respect to time and accuracy.

Bin	Time(s)	Accuracy
1	5.32	0.999
2-3	17.9	0.994
4-5	26.13	0.968
6-7	29.73	0.926
8-9	24.9	0.878
10-15	27.51	0.999
Total	131.55	0.96

Table 4.17: Software Metrics for GenASM on real dataset

For hardware estimation, the maximum Workload/Aligner was used to estimate how many times the hardware kernels must be executed. To ensure that data transfer overhead between runs of the same kernel was minimized, an axi-stream interface was used to stream data from a FIFO queue to the kernels. The max Workload/Aligner is 240835 on the aligner of edit threshold E=5 which needs 216 cycles to fully execute. So, given a 10ns clock period time, the hardware needed to fully run was 0.52 seconds, resulting in a x252 speedup in respect to software alignment.
This time GenASM has a much better accuracy than before. This shows that pre-filtering is very dataset dependent and the accuracy error SneakySnake can introduce depends heavily on the dataset and the distribution of edits within that dataset.

To compare the binned version proposed by this thesis against a configuration of GenASM aligners that their edit threshold was not exploited by filtering. Because the real dataset has an upper edit threshold of 15 edits, the FPGA was filled with as many GenASM aligners it could fit with an edit threshold of fifteen. The result was, that 6 aligners could fit with a resource utilization of:

BRAM(%)	LUTs(%)	FFs(%)
84	20	6

Table 4.18: Base GenASM Resources for real dataset

That means, that the Workload/Aligner is 760000 and the Hardware Time(s) is 1.7s yielding a x266 speedup in respect to a software implementation run-

ning at 15 edit threshold for the whole dataset. The results that compare the GenASM aligner against the base GenASM are shown in the following table:

Aligner	Hardware Time(s)	Speedup(in respect to Base)	Accuracy
Baseline GenASM	1.7	1	0.999
GenASM	0.52	3.26	0.96

Table 4.19: Comparison between Baseline BSW and proposed BSW

As shown above, there is a x3.26 speedup between the baseline version and the proposed aligner with a small loss in accuracy (0.03).

4.4 Banded Smith-Waterman with real dataset

The same reasoning was used for this experiment. After using a balancing script but with LUTs as critical resource instead of BRAM the following configuration of Banded Smith-Waterman kernels was proposed.

BSW size	Num. of Aligners	Workload	Workload/Aligner
4	11	1457776	132525
7	8	1043152	130394
8	2	255347	127673
11	3	412811	137603
15	1	76924	76924

Table 4.20: Banded Smith-Waterman(BSW) configuration with real dataset

The above configuration of aligners has a total resource estimation of:

BRAM(%)	LUTs(%)	FFs(%)
27	96	4

Table 4.21: Total Resources of BSW on simulated dataset

After binning the data to the proposed bins with the help of SneakySnake, the Banded Smith-Waterman algorithm with an appropriate edit threshold was executed for each bin. The results are the following, in respect to time and accuracy.

For hardware estimation, the maximum Workload/Aligner was used to estimate how many times the hardware kernels must be executed. To ensure that data transfer overhead between runs of the same kernel was minimized, an axi-stream interface was used to stream data from a FIFO queue to the kernels. The max Workload/Aligner is 137603 on the aligner of edit threshold E=11 which needs 354 cycles to fully execute. So, given a 10ns clock period time, the hardware needed to fully run was 0.48 seconds, resulting in a x47 speedup in respect to software alignment.

Bin	Time(s)	Accuracy
1-4	6.41	0.99
5-7	7.92	0.96
8	2.37	0.90
9-11	4.91	0.95
12-15	1.13	0.99
Total	22.76	0.97

Table 4.22: Software Metrics for BSW on real dataset

To compare the binned version proposed by this thesis against a configuration of Banded Smith-Waterman aligners that their edit threshold was not exploited by filtering. Because the simulated dataset has an upper edit threshold of 15 edits, the FPGA was filled with as many Banded Smith-Waterman aligners it could fit with an edit threshold of fifteen. The result was, that 15 aligners could fit with a resource utilization of:

BRAM(%)	LUTs(%)	FFs(%)
22	94	5

Table 4.23: Base BSW Resources

That means, that the Workload/Aligner is 304000 and the Hardware Time(s) is 1.11s yielding a x45 speedup in respect to a software implementation running at 15 edit threshold for the whole dataset. The results that compare the Banded Smith-Waterman aligner against the base BSW are shown in the following table:

Aligner	Hardware Time(s)	Speedup(in respect to base)	Accuracy
Baseline BSW	1.11	1	0.99
BSW	0.48	2.31	0.97

Table 4.24: Comparison between base BSW and proposed BSW on real dataset

As shown above, there is a x2.31 speedup between the base version and the proposed aligner with a slight loss in accuracy (0.02).

Comparing the two aligners for the real dataset, we conclude these results.

Aligner	Accuracy	Hardware Time(s)
GenASM	0.96	0.52
BSW	0.96	0.48

Table 4.25: Comparison between the two aligners for the simulated dataset

The two aligners have very similar accuracy on the real dataset. In terms of hardware latency, GenASM is 0.04s slower on the real dataset than Banded Smith-Waterman.

All the aforementioned results are presented in Figure 4.2 and Figure 4.3

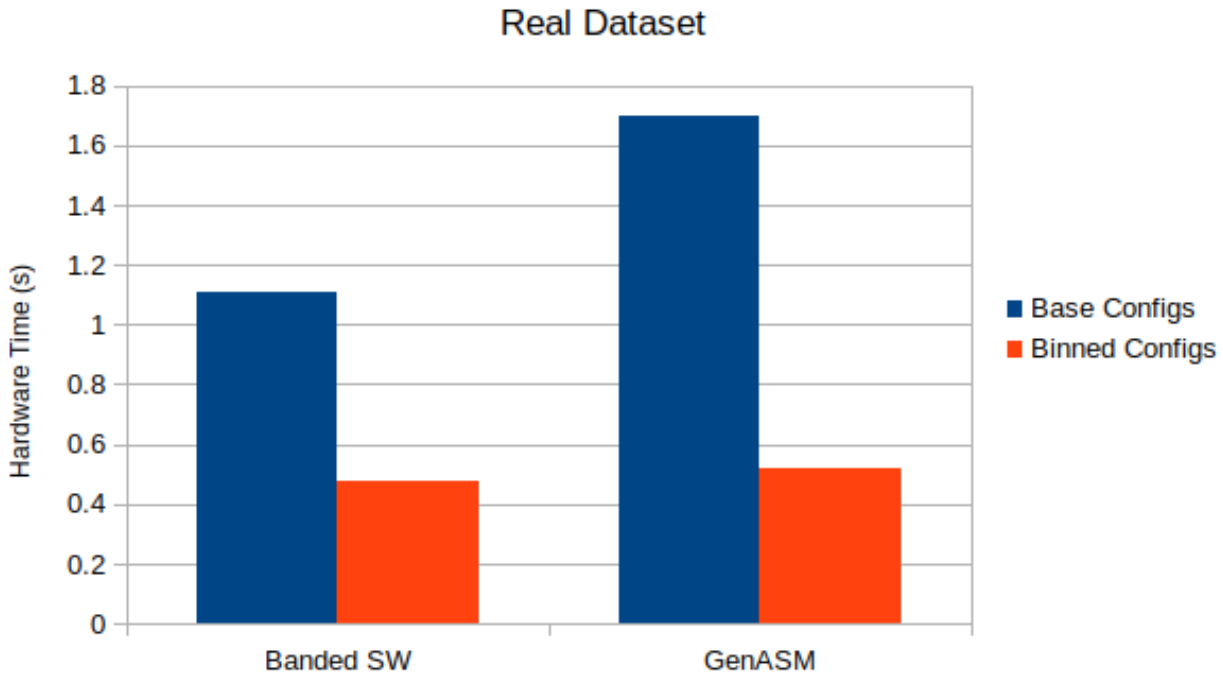


Figure 4.2: Hardware Times of aligners for real dataset

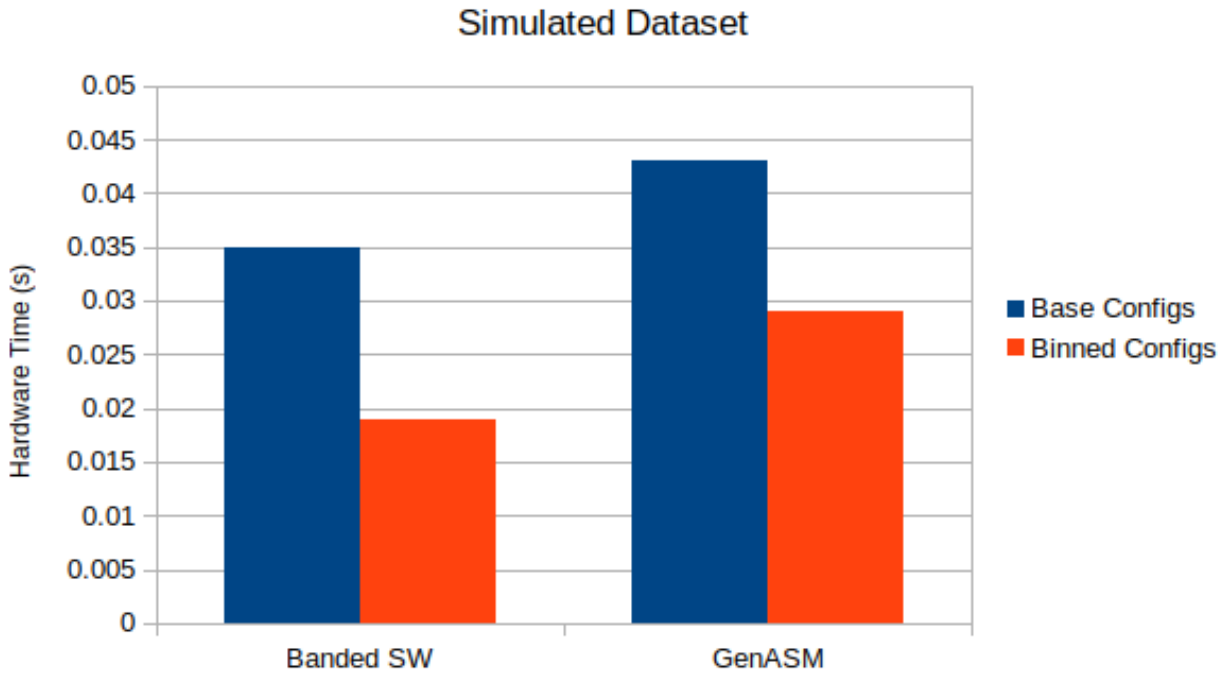


Figure 4.3: Hardware Times of aligners for simulated dataset

4.5 SneakySnake Overhead

Despite using SneakySnake as an offline profiling tool, it is important to mention the time overhead it introduces when discussing the speedup of the proposed architecture. We compare the end to end workflow that includes both filtering and alignment. We compare the following configurations:

- software filter + accelerated aligner against software pipeline
- software filter + accelerated aligner against baseline aligner
- software filter + binned software against baseline software

SneakySnake time for the simulated dataset is 0.016s and for the Illumina dataset is 0.35s.
When comparing the software binned version to the hardware binned version, the SneakySnake time is just added to the two latencies and then find the new speedup. This produces the following results.

Experiment	Dataset	Original Speedup against software	Speedup after overhead
GenASM	simulated	x220	x144
	real	x252	x151
Banded SW	simulated	x36	x23
	real	x47	x29

Table 4.26: Measuring Speedup against software including the SneakySnake overhead

When comparing the proposed architecture (hardware binned version) to the baseline scenario aligners, the SneakySnake overhead is only added to the proposed architecture.

Experiment	Dataset	Original Speedup against Baseline Hardware	Speedup after overhead
GenASM	simulated	x1.48	x0.955(Slowdown)
	real	x3.26	x1.95
Banded SW	simulated	x1.8	x1.09
	real	x2.29	x1.46

Table 4.27: Measuring Speedup against base hardware including the SneakySnake overhead

In most cases we observe a speedup even after the SneakySnake overhead is introduced. Two key observations can be made. First, the more data we process the greater the gain from SneakySnake. Second, various aligners can behave different on different datasets. GenASM has better speedup on the real dataset, and Banded Smith-Waterman has better speedup on the simulated dataset. Finally, these results, especially the slowdown, call for further research on SneakySnake and pre-filtering algorithms in general, as well for a full hardware implementation that can speedup this process without losing too much

accuracy.

When comparing the software versions of our binned proposed architecture to the base software the overhead time is only added to the binned architecture.

Experiment	Dataset	Software Speedup against Baseline Software
GenASM	simulated	x2.1
	real	x2.47
Banded SW	simulated	x2.2
	real	x2.29

Table 4.28: Software Comparison

These results show the importance of data-aware procedures and architectural design, and that those techniques can benefit both Hardware and Software implementations.

Chapter 5

Conclusion

5.1 Summary

This thesis explored the design of different genomic aligners, deployed on FPGA-based accelerators using a data-aware approach. This approach was achieved by using the pre-filtering algorithm SneakySnake, enabling a more custom-to-dataset design methodology of the alignment step of the genomics pipeline. Also, High Level Synthesis tools were used as a method to speedup the design process and testing if such tools are good candidates for genomics hardware design.

First, the proposed methodology exposed the benefits of using pre-filtering algorithms to speedup alignment. By efficiently discarding dissimilar pairs and by giving insights of exact edit thresholds, a custom configuration of aligners could be deployed to achieve speedups of over x2 for certain aligners. These results highlight the importance of the pre-filtering step, and show how important is to develop good pre-filters with great accuracy.

Second, this thesis showed how useful High Level Synthesis can be in designing hardware for genomic algorithms. It significantly sped up the design and testing procedure, as writing in more human understandable languages like C, instead of hardware description languages like VHDL or Verilog, can make the design process that much easier. Despite its more limited capabilities, in respect to HDLs, it proved quite useful and capable in designing complex DP algorithms with great results.

Finally, this work emphasizes the need of data-aware approaches when designing systems for genomic pipelines. Genomic datasets can vary in many aspects, e.g. edits distribution, distribution of edits and types of edits within the sequences. This information can be exploited by designers to create more data-aware hardware kernels and systems that greatly improve the accuracy

and latency of such designs.

5.2 Future Work

The current work can be expanded in various directions:

- Deployment of the designs mentioned in the current thesis on an actual board to extract real measurements and not estimates
- Mapping the above designs to Alveo boards' multiple SLRs to be able to fit even more aligners into a system to achieve greater speedup
- Executing experiments on whole human genomes
- SneakySnake can be specifically designed to run on hardware, creating a complete end-to-end pipeline that will drastically reduce the time needed for both pre-filtering and alignemtn to be executed
- Designing a Verilog/VHDL implementation of banded WFA with the data-aware workflow of this thesis to explore if it can produce great results as it achieves on CPUs and GPUs
- Verilog/VHDL implementations of GenASM and Banded Smith Waterman that can achieve greater hardware times and maybe better area utilization.
- Explore even more data-aware workflows for genomic pipelines

Bibliography

[1] K.-M. Chao, W. R. Pearson, and W. Miller, "Aligning two sequences within a specified diagonal band," *Bioinformatics*, vol. 8, no. 5, pp. 481–487, 1992.

[2] H. Suzuki and M. Kasahara, "Acceleration of nucleotide semi-global alignment with adaptive banded dynamic programming," *BioRxiv*, p. 130633, 2017.

[3] M. G. Ross, C. Russ, M. Costello, A. Hollinger, N. J. Lennon, R. Hegarty, C. Nusbaum, and D. B. Jaffe, "Characterizing and measuring bias in sequence data," *Genome biology*, 2013.

[4] Wikipedia contributors, "Genome," 2025, accessed: 2025-07-03. [Online]. Available: https://en.wikipedia.org/wiki/Genome

[5] ——, "Genomics," 2025, accessed: 2025-07-03. [Online]. Available: https://en.wikipedia.org/wiki/Genomics

[6] C. L. Richards, O. Bossdorf, and K. J. Verhoeven, "Understanding natural epigenetic variation," *The New Phytologist*, vol. 187, no. 3, pp. 562–564, 2010.

[7] M. Alser, Z. Bingol, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, " Accelerating Genome Analysis: A Primer on an Ongoing Journey ," *IEEE Micro*, vol. 40, no. 05, pp. 65–75, Sep. 2020. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MM.2020.3013728

[8] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, 1970.

[9] C. Moeckel, M. Mareboina, M. A. Konnaris, C. S. Chan, I. Mouratidis, A. Montgomery, N. Chantzi, G. A. Pavlopoulos, and I. Georgakopoulos-Soares, "A survey of k-mer methods and applications in bioinformatics," *Computational and Structural Biotechnology Journal*, vol. 23, pp. 2289–2303, 2024.

[10] K. Prousalis, E. Kartsakli, C. Antonopoulos, and S. Papavassiliou, "A survey on sequence alignment algorithms and state-of-the-art," *ACM Computing Surveys*, 2025.

[11] M. Ebel, R. Wittler, K. Reinert, and T. H. Dadi, "Global, highly specific and fast filtering of alignment seeds," *BMC Bioinformatics*, vol. 23, no. 1, p. 47, 2022.

[12] Z. Bingöl, C. Alkan, M. Alser, O. Ozturk *et al.*, "Fast and accurate pre-alignment filtering in short read mapping," *arXiv preprint arXiv:2103.14978*, 2021.

[13] D. C. Koboldt, "Best practices for variant calling in clinical sequencing," *Genome Medicine*, vol. 12, no. 1, p. 91, 2020.

[14] S. Zhao, Y. Zhang, W. Gordon, J. Quan, and R. Xi, "Accuracy and efficiency of germline variant calling pipelines evaluated on whole genome sequencing data," *Scientific Reports*, vol. 10, p. 20285, 2020.

[15] M. Alser, T. Shahroodi, J. Gómez-Luna, C. Alkan, and O. Mutlu, "Sneakysnake: a fast and accurate universal genome pre-alignment filter for cpus, gpus and fpgas," *Bioinformatics*, vol. 36, no. 22-23, pp. 5282–5290, 2020.

[16] ——, "Sneakysnake: a fast and accurate universal genome pre-alignment filter for cpus, gpus and fpgas," *Bioinformatics*, vol. 36, no. 22–23, p. 5282–5290, Dec. 2020. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btaa1015

[17] G. G. Nardone, D. Raimondi, L. Chessa, and M. Pala, "A hitchhiker guide to structural variant calling," *Biomedicines*, vol. 13, no. 8, p. 1949, 2025.

[18] M. Alser, O. Mutlu, and O. Ozturk, "From molecules to genomic variations: Accelerating genome analysis," *Bioinformatics*, 2022.

[19] A. Arjona, A. Gabriel-Atienza, S. Lanuza-Orna, X. Roca-Canals, A. Bourramouss, T. K. Chafin, L. Marcello, P. Ribeca, and P. García-López, "Scaling a variant calling genomics pipeline with faas," in *Proceedings of the 9th International Workshop on Serverless Computing (WoSC '23)*, 2023, pp. 59–64.

[20] X. Cao, S. C. Li, and A. K. H. Tung, "Indexing dna sequences using q-grams," in *Database Systems for Advanced Applications*, L. Zhou, B. C. Ooi, and X. Meng, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 4–16.

[21] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Shifted hamming distance: a fast and accurate simd-friendly filter to accelerate alignment verification in read mapping," *Bioinformatics*, vol. 31, no. 10, pp. 1553–1560, 2015.

[22] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 2017.

[23] ——, "Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping," *Bioinformatics*, vol. 33, no. 21, p. 3355–3363, May 2017. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btx342

[24] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "Grim-filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies," *BMC genomics*, vol. 19, no. 2, pp. 23–40, 2018.

[25] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: a fast and efficient pre-alignment filter for sequence alignment," *Bioinformatics*, vol. 35, no. 21, pp. 4255–4263, 2019.

[26] M. Šošić and M. Šikić, "Edlib: a c/c++ library for fast, exact sequence alignment using edit distance," *Bioinformatics*, vol. 33, no. 9, pp. 1394–1395, 2017.

[27] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[28] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, p. 357, 2012.

[29] E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, vol. 64, no. 1–3, pp. 100–118, 1985.

[30] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

[31] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint*, 2013.

[32] Q. Aguado-Puig, M. Doblas, C. Matzoros, A. Espinosa, J. C. Moure, S. Marco-Sola, and M. Moreto, "Wfa-gpu: gap-affine pairwise read-alignment using gpus," *Bioinformatics*, vol. 39, no. 12, p. btad701, 11 2023. [Online]. Available: https://doi.org/10.1093/bioinformatics/btad701

[33] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, 1982.

[34] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A greedy algorithm for aligning dna sequences," *Journal of Computational Biology*, vol. 7, no. 1-2, pp. 203–214, 2000.

[35] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

[36] ——, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[37] A. Dobin, C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras, "Star: ultrafast universal rna-seq aligner," *Bioinformatics*, vol. 29, no. 1, pp. 15–21, 2013.

[38] N. Trifunovic, V. Milutinovic, N. Korolija, and G. Gaydadjiev, "An app-gallery for dataflow computing," *Journal of Big Data*, vol. 3, no. 1, pp. 1–30, 2016.

[39] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler, "Faster and more accurate sequence alignment with snap," *arXiv preprint*, 2011.

[40] F. J. Sedlazeck, P. Rescheneder, M. Smolka, H. Fang, M. Nattestad, A. von Haeseler, and M. C. Schatz, "Accurate detection of complex structural variations using single-molecule sequencing," *Nature Methods*, vol. 15, no. 6, pp. 461–468, 2018.

[41] Q. Aguado-Puig, M. Doblas, C. Matzoros, A. Espinosa, J. C. Moure, S. Marco-Sola, and M. Moreto, "Wfa-gpu: gap-affine pairwise read-alignment using gpus," *Bioinformatics*, vol. 39, no. 12, p. btad701, 2023.

[42] Y.-L. Liao, Y.-C. Li, N.-C. Chen, and Y.-C. Lu, "Adaptively banded smith-waterman algorithm for long reads and its hardware accelerator," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–9.

[43] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand *et al.*, "Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020.

[44] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, vol. 37, no. 4, pp. 456–463, 2021.

[45] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*. ACM, 2007, pp. 39–48.

[46] Wikipedia contributors, "Genomics," 2025, accessed: 2025-07-03. [Online]. Available: https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm

[47] X. Hu and I. Friedberg, "Swiftortho: a fast, memory-efficient, multiple genome orthology classifier," *bioRxiv*, 2019. [Online]. Available: https://www.biorxiv.org/content/early/2019/02/17/543223

[48] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," 2020. [Online]. Available: https://arxiv.org/abs/2009.07692

[49] Siemens, "Hls," 2025, accessed: 2025-07-03. [Online]. Available: https://resources.sw.siemens.com/en-US/white-paper-working-smarter-not-harder-nvidia-closes-design-complexity-gap-with-

[50] XILINX, "Hls," 2025, accessed: 2025-07-03. [Online]. Available: https://docs.amd.com/r/2022.1-English/ug1399-vitis-hls/Basics-of-High-Level-Synthesis

[51] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: Successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, 2022.

[52] A. Cornu, S. Derrien, and D. Lavenier, "Hls tools for fpga: Faster development with better performance," in *Reconfigurable Computing: Architectures, Tools and Applications (ARC 2011)*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 6578, pp. 67–78.

[53] Y. Cao, A. Gupta, J. Liang, and Y. Turakhia, "Dp-hls: A high-level synthesis framework for accelerating dynamic programming algorithms in bioinformatics," *arXiv preprint arXiv:2411.03398*, 2024.

[54] Y. Feng, Z. Li, G. G. Akbulut, V. Narayanan, M. T. Kandemir, and C. R. Das, "Fpga-based accelerator for adaptive banded event alignment in nanopore sequencing data analysis," *BMC Bioinformatics*, vol. 26, 2025.

[55] S. F. Schifano *et al.*, "High-throughput edit distance computation on fpga-based accelerator combining metaprogramming and high-level synthesis," *The Journal of Systems Architecture*, 2025.

[56] XILINX, "Hls," 2025, accessed: 2025-07-03. [Online]. Available: https://docs.amd.com/r/en-US/ug1366-vck190-eval-bd

[57] https://www.illumina.com/products/by-type/informatics-products/dragen-bio-it-platform.html, "Illumina DRAGEN Bio-IT Platform."

[58] A. Haghi, S. Marco-Sola, L. Alvarez, D. Diamantopoulos, C. Hagleitner, and M. Moreto, "Wfa-fpga: An efficient accelerator of the wavefront algorithm for short and long read genomics alignment," *Future Generation Computer Systems*, vol. 149, pp. 39–58, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X2300256X