ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Accelerating Sequence-to-Graph Genome Mapping and Alignment on FPGAs using High Level Synthesis

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ειρήνη Μαρία Τζέρμπου

Επιβλέπων : Σούντρης Δημήτριος
Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Accelerating Sequence-to-Graph Genome Mapping and Alignment on FPGAs using High Level Synthesis

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ειρήνη Μαρία Τζέρμπου

Επιβλέπων : Σούντρης Δημήτριος
Καθηγητής ΕΜΠ

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

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

Αθήνα, Ιούλιος 2025

Copyright © Ειρήνη Μαρία Τζέρμπου, 2025.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

(Υπογραφή)

....

Ειρήνη Μαρία Τζέρμπου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
©2025 - All rights reserved.

Περίληψη

Ένα κρίσιμο κομμάτι της ανάλυσης αλληλουχίας γονιδιώματος περιλαμβάνει την ευθυγράμμιση τμημάτων DNA που λαμβάνονται από ένα άτομο σε μια τυπική γραμμική αλληλουχία αναφοράς γονιδιώματος, μια διαδικασία που ονομάζεται αντιστοίχιση αλληλουχίας προς αλληλουχία (sequence-to-sequence alignment). Πρόσφατα, το γραμμικό γονιδίωμα αναφοράς έχει αντικατασταθεί από τους γονιδιωματικούς γράφους, οι οποίοι αντικατοπτρίζουν καλύτερα τις γενετικές παραλλαγές και την ποικιλομορφία που υπάρχουν σε διαφορετικά άτομα μέσα σε έναν πληθυσμό. Η ευθυγράμμιση (alignment) των αλληλουχιών DNA (reads) σε γονιδιωματικό γράφο αναφοράς (γνωστή ως αντιστοίχιση αλληλουχίας προς γράφο ή sequence-to-graph alignment) βελτιώνει σημαντικά την ακρίβεια της ανάλυσης του γονιδιώματος. Ωστόσο, ενώ η αντιστοίχιση αλληλουχίας προς αλληλουχία είναι καλά εδραιωμένη με πολυάριθμα διαθέσιμα εργαλεία και επιταχυντές υλικού (hardware accelerators),η αντιστοίχιση αλληλουχίας προς γράφο αποτελεί πιο σύνθετο υπολογιστικό πρόβλημα με πολύ λιγότερες εφαρμογές λογισμικού και ακόμα λιγότερους σχετικούς επιταχυντές.

Στην παρούσα εργασία, προτείνουμε μια μέθοδο αντιστοίχισης (mapping) και ευθυγράμμισης (alignment) αλληλουχιών σε γονιδιωματικούς γράφους αναφοράς, σχεδιάζοντας, αξιολογώντας και συγκρίνοντας δύο αρχιτεκτονικές: η μία ανάγει το πρόβλημα στην αντιστοίχιση αλληλουχίας προς αλληλουχία και η άλλη ακολουθεί προσέγγιση στοίχισης αλληλουχίας προς γράφο. Επιπλέον, αναπτύσσουμε τη δική μας διαδικασία εύρεσης υποψήφιων υπογράφων αντιστοίχισης με την δοσμένη αλληλουχία (seeding) εισάγοντας αλγορίθμους φιλικούς προς το υλικό, τους οποίους αξιολογούμε τόσο ως προς την ακρίβεια όσο και την απόδοση.Ο επιταχυντής μας υλοποιείται με χρήση Υψηλού Επιπέδου Σύνθεσης (HLS) σε FPGA AMD Alveo U200, εκμεταλλευόμενοι την επαναπρογραμματισιμότητα αυτής της υπολογιστικής πλατφόρμας, για να προσφερθεί μια ευέλικτη και προσαρμόσιμη λύση. Και οι δύο προτεινόμενοι επιταχυντές, που υλοποιούν την ολοκληρωμένη ροή εργασίας, επιτυγχάνουν έως και 8.16 φορές υψηλότερη απόδοση σε σύγκριση με την αντίστοιχη υλοποίηση λογισμικού. Το έργο μας είναι το πρώτο που κατασκευάζει έναν ευθυγραμμιστή αλληλουχίας προς γράφο που ανάγει το πρόβλημα στην αντιστοίχιση αλληλουχίας προς αλληλουχία και συγκρίνει αυτές τις διαφορετικές λύσεις, ενώ ταυτόχρονα προτείνει μια αυτόνομη ροή εργασίας και αρχιτεκτονική για το βήμα του seeding έχοντας γονιδίωμα αναφοράς σε μορφή γράφου.

Keywords— Γονιδιωματική, Γονιδιωματικοί Γράφοι, Αλληλούχιση, Επιταχυντής, FPGA, Σύνθεση Υψηλού-Επιπέδου (HLS)

Abstract

A crucial part of genome sequence analysis involves aligning DNA fragments (known as reads) obtained from an individual to a standard linear reference genome sequence, a process called sequence-to-sequence alignment. Recently, this linear reference has been replaced by a graph-based representation of the genome, which better reflects genetic variations and diversity found across different individuals within a population. Aligning reads to this graph-based reference (referred to as sequence-to-graph mapping) significantly enhances the accuracy of genome analysis. However, while sequence-to-sequence mapping is well-established with numerous tools and hardware accelerators available, sequence-to-graph mapping presents a more complex computational challenge and currently has far fewer practical software solutions and even fewer proposed hardware accelerators which could have important contribution in speeding up the genome analysis pipeline. In this work, we propose a sequence-to-graph read mapping and alignment by designing, evaluating and comparing two hardware architectures, one deducing the problem to sequence-to-sequence alignment and the other following a sequence-to-graph approach. Additionally, we build our own seeding workflow introducing hardware friendly seeding algorithms, which we evaluate in terms of both accuracy and performance. Our accelerator is implemented using High-Level Synthesis (HLS) on an AMD Alveo U200 FPGA, leveraging the reconfigurability of FPGAs to offer a flexible and adaptable solution. Both proposed accelerators, implementing the integrated workflow, achieve maximum X8.16 higher performance than the software implementation on a high-end server CPU. Our work is the first to building a sequence-to-graph aligner that deduces the problem to sequence-to-sequence alignment and to compare these different solutions, while proposing a standalone seeding workflow and architecture. Our contributions pave the way to finding novel algorithms and architectures to accelerate genome mapping and alignment while using reference genome representation with a greater amount of genomic information, such as genome graphs.

Keywords— Genomics, Genome Graphs, Seeding, Sequence-to-Graph Alignment, Accelerator, FPGA, High-Level Synthesis (HLS)

Ευχαριστίες

Θα ήθελα αρχικά να ευχαριστήσω από καρδιάς τον επιβλέποντα καθηγητή μου, κύριο Δημήτριο Σούντρη, ο οποίος μου έδωσε την ευκαιρία να ασχοληθώ με ένα τόσο ιδιαίτερο και συνδυαστικό θέμα, να εργαστώ σε ένα εξαιρετικό περιβάλλον, αυτό του Εργαστηρίου Μικροϋπολογιστών και Ψηφιακών Συστημάτων(MicroLab), καθώς και να συνεργαστώ με μία ομάδα πολύ αξιόλογων μηχανικών και εξαιρετικών ανθρώπων.

Επιπλεόν, ευχαριστώ θερμά την Δρ. Κωνσταντίνα Κολιογεώργη για την συνεχή υποστήριξη, καθοδήγηση και αρωγή στην κατανόηση εννοιών και οργάνωση των ιδεών μου, βοηθώντας με πάντα να θέτω συγκεκριμένους ενδιάμεσους στόχους. Στην συνέχεια θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επίκουρο καθηγητή κύριο Σωτήριο Ξύδη και τον υποψήφιο διδάκτορα Άγγελο Φερίκογλου, για την πολύτιμη βοήθεια τους ιδιαίτερα τους τελευταίους μήνες εκπόνησης της παρούσας εργασίας. Οι συχνές συζητήσεις μας ήταν καθοριστικές τόσο στην βελτίωση και καλύτερη οργάνωση της εργασίας όσο και στην υλοποίηση και αξιολόγησή της στο FPGA. Τέλος, είμαι βαθειά ευγνώμων στον Αναστάσιο Καπετανάκη, μέλος του εργαστηρίου, για τον χρόνο, τις συζητήσεις, την υποστήριξη και την διάθεσή του να συνδράμει καθοριστικά στο τεχνικό κομμάτι της υλοποίησης.

Ευχαριστώ επίσης θερμά όλα τα μέλη του Microlab για την συμπαράσταση, τις εποικοδομητικότατες συζητήσεις και τις όμορφες στιγμές στο εργαστήριο. Εφόσον η παρούσα εργασία σηματοδοτεί και το τέλος των προπτυχιακών μου σπουδών, θα ήθελα να ευχαριστήσω τους φίλους μου για την υποστήριξή τους όλα αυτά τα χρόνια. Θα ήθελα επίσης να εκφράσω την ευγνωμοσύνη μου σε όλους τους δασκάλους, καθηγητές και παιδαγωγούς που συνέβαλαν ο καθένας με τον τρόπο του, στη διαμόρφωση της μέχρι εδώ πορείας μου και στην απόκτηση πολύτιμων γνώσεων, εμπειριών και συμβουλών. Ιδιαίτερα θα ήθελα να ευχαριστήσω την Δρ. Αλεξία-Βικτώρια Πολυσίδη για την ενθάρρυνση και την εμπιστοσύνη, καθώς και την κυρία Νίνα Πατρικίδου για τον τρόπο που με έμαθε να αντιμετωπίζω τις δυσκολίες και τις προκλήσεις. Τέλος, το μεγαλύτερο ευχαριστώ οφείλω στην οικογένεια μου, στους γονείς μου Φώτη και Νάντια και ιδιαίτερα στον παππού μου Θάνο Κωνστιάντο, για ό,τι έχουν κάνει και κάνουν για εμένα. Οφείλω πραγματικά την μέχρι τώρα πορεία μου στην αγάπη, την υποστήριξη και το ακόρεστο ενδιαφέρον τους και για αυτό τους ευχαριστώ.

Contents

List of Figures

List of Tables

Εκτεταμένη Ελληνική Περίληψη

1 Εισαγωγή

Η χαρτογράφηση γονιδιώματος αποτελεί θεμελιώδη διαδικασία στη γονιδιωματική, που επιτρέπει τον εντοπισμό και την οργάνωση των γονιδίων και άλλων σημαντικών στοιχείων στο DNA ενός οργανισμού. Αποτελεί κρίσιμο βήμα για την ανάλυση του γονιδιώματος, παρέχοντας το δομικό πλαίσιο για την ερμηνεία των δεδομένων αλληλουχίας και την εξαγωγή βιολογικών συμπερασμάτων. Η ακριβής χαρτογράφηση είναι απαραίτητη για εφαρμογές όπως η ανίχνευση μεταλλάξεων ασθενειών, εξελικτικές μελέτες και εξατομικευμένη ιατρική. Με την πρόοδο των τεχνολογιών αλληλούχησης και την αύξηση του όγκου δεδομένων, η ανάγκη για αποδοτικές και κλιμακούμενες μεθόδους χαρτογράφησης αυξάνεται σημαντικά.

Η πιο απαιτητική υπολογιστικά φάση είναι η αντιστοίχιση/στοίχιση/ευθυγράμμιση (alignment) των τμημάτων DNA (reads) με το γονιδίωμα αναφοράς, διαδικασία που απαιτεί μεγάλο χρόνο εκτέλεσης και πόρους. Οι παραδοσιακές μέθοδοι βασίζονται σε ευθυγράμμιση αλληλουχίας προς αλληλουχία (sequence-to-sequence) με γραμμικό γονιδίωμα αναφοράς, όμως παρουσιάζουν περιορισμούς λόγω της γενετικής ποικιλότητας που δεν καλύπτεται από ένα μόνο γραμμικό γονιδίωμα αναφοράς, προκαλώντας μεροληψία (reference bias) και μειωμένη ακρίβεια.

Για την αντιμετώπιση αυτού του προβλήματος, χρησιμοποιούνται πλέον δομές γονιδιωμάτων σε μορφή γράφων (genome graphs), που συνδυάζουν το γραμμικό γονιδίωμα με γνωστές γενετικές παραλλαγές σε ολόκληρο τον πληθυσμό. Αν και αυξάνουν την πολυπλοκότητα και τους πόρους, έχουν αναπτυχθεί λογισμικά όπως το GraphAligner που εκτελούν ευθυγράμμιση αλληλουχίας προς γράφο (sequence-to-graph) με δυναμικό προγραμματισμό, βελτιώνοντας την ακρίβεια. Παρ' όλα αυτά, η ευθυγράμμιση προς γράφο παραμένει το βασικό bottleneck, καταναλώνοντας μεγάλο ποσοστό χρόνου και πόρων, με προβλήματα όπως υψηλά ποσοστά cache misses και περιορισμένη δυνατότητα πλήρους παραλληλοποίησης.

Υπάρχει επομένως ανάγκη για ειδικά σχεδιασμένους, ισορροπημένους και κλιμακούμενους συνδυασμούς υλικού και λογισμικού που θα επιταχύνουν τις φάσεις της σποράς (seeding) και ευθυγράμμισης (alignment). Ο πρώτος καθολικός επιταχυντής υλικού SeGraM προσφέρει σημαντικές βελτιώσεις στην επιτάχυνση και μείωση κατανάλωσης ενέργειας, αλλά παρουσιάζει περιορισμούς στην ακρίβεια λόγω ορίου στο μέγεθος των υπογράφων που ευθυγραμμίζονται.

Η παρούσα διπλωματική εργασία εστιάζει στην εξερεύνηση διαφορετικών αλγορίθμων και αρχιτεκτονικών υλικού για την επιτάχυνση της χαρτογράφησης γονιδιώματος με γράφους, σχεδιάζοντας και αξιολογώντας δύο προσεγγίσεις: μία με ευθυγράμμιση αλληλουχίας προς αλληλουχία και μία με αλληλουχία προς γράφο, ενώ προτείνει και έναν φιλικό προς το υλικό αλγόριθμο για τη διαχείριση υπογράφων κατά το seeding. Οι διάφορες εκδοχές του επιταχυντή, προορισμένες για συγκεκριμένη κατηγορία υπολογιστικών συστημάτων, τα Field Programmable Gate Arrays (FPGAs), αναπτύχθηκαν χρησιμοποιώντας Σύνθεση Υψηλού Επιπέδου (HLS).

2 Θεωρητικό Υπόβαθρο

2.1 Γονιδίωμα και Γονιδιωματική

Η γονιδιωματική, ως το πιο ανεπτυγμένο πεδίο, ασχολείται με τη μελέτη της δομής, λειτουργίας, εξέλιξης, χαρτογράφησης και επεξεργασίας των γονιδιωμάτων, στοχεύοντας στη συνολική κατανόηση όλων των γονιδίων και των αλληλεπιδράσεών τους, σε αντίθεση με τη γενετική που επικεντρώνεται σε μεμονωμένα γονίδια. Το γονιδίωμα περιλαμβάνει όλη τη γενετική πληροφορία ενός οργανισμού, τόσο τα γονίδια που κωδικοποιούν πρωτεΐνες όσο και τα μη κωδικοποιητικά, καθώς και ρυθμιστικές αλληλουχίες και αχρησιμοποίητο DNA. Η ανάλυση και αλληλούχιση του γονιδιώματος έχει οδηγήσει σε σημαντική πρόοδο στη βιοϊατρική και τις βιοεπιστήμες, χάρη στις νέες τεχνολογίες αλληλούχισης και την ανάπτυξη βιοπληροφορικών εργαλείων που διαχειρίζονται τον τεράστιο όγκο δεδομένων. Αυτές οι εξελίξεις έχουν καταστήσει τη γονιδιωματική απαραίτητη για τη σύγχρονη βιολογία και την ιατρική.

Οι πρώτες προσπάθειες για τον προσδιορισμό της αλληλουχίας του DNA ξεκίνησαν το 1970 από τον Ray Wu, ο οποίος ανέπτυξε μια μέθοδο εκκινητή ειδικής θέσης με τη χρήση DNA πολυμεράσης και ειδικά επισημασμένων νουκλεοτιδίων. Η τεχνική εξελίχθηκε από τον Frederick Sanger, ο οποίος χρησιμοποίησε την ενσωμάτωση αλυσίδων-τερματιστών (ddNTPs) κατά την in vitro αντιγραφή του DNA, επιτρέποντας έτσι την ταχύτερη αλληλούχιση. Το 1977, αλληλουχήθηκε για πρώτη φορά το DNA βακτηριοφάγου με τις τεχνικές των Sanger και Maxam. Η αλληλούχιση του ανθρώπινου γονιδιώματος ολοκληρώθηκε τη δεκαετία του 2000 με κόστος περίπου 3 δισεκατομμύρια δολάρια, εγκαινιάζοντας την "πρώτη γενιά" αλληλούχισης. Έκτοτε, η τεχνολογία έχει εξελιχθεί σημαντικά, αυξάνοντας την ταχύτητα και μειώνοντας το κόστος ανά νουκλεοτίδιο.[1]

Η παραγωγή δεδομένων αλληλούχισης περιλαμβάνει τρία βασικά στάδια: συλλογή δείγματος, προετοιμασία βιβλιοθήκης και αλληλούχιση, με τα δύο πρώτα να γίνονται στο εργαστήριο πριν τη χρήση ειδικών μηχανημάτων αλληλούχισης. Τα μηχανήματα αυτά διαφέρουν ως προς την απόδοση, το μήκος των reads, το ποσοστό λαθών, το μέγεθος και το κόστος. Τα reads χωρίζονται σε τρεις κατηγορίες: μικρά (μερικές εκατοντάδες βάσεις), πολύ μεγάλα (εκατοντάδες έως εκατομμύρια βάσεις) και ακριβή μεγάλα (έως μερικές χιλιάδες βάσεις). Υπάρχει συνήθως αντιστάθμιση μεταξύ μήκους και ακρίβειας, με τα μικρότερα reads να είναι πιο ακριβή.

Οι τεχνολογίες αλληλούχισης διακρίνονται σε τρεις βασικές κατηγορίες: Τεχνολογίες μικρών reads (δεύτερη γενιά, π.χ. Illumina) με υψηλή ακρίβεια και πολύ μεγάλη απόδοση, πολύ μεγάλων reads (τρίτη γενιά, π.χ. nanopore) που επιτρέπουν ανάγνωση πολύ μεγάλων αλληλουχιών αλλά με μεγαλύτερο ποσοστό λαθών και τεχνολογίες ακριβών μεγάλων reads (π.χ. PacBio), που προσφέρουν μεγάλη ακρίβεια αλλά σε υψηλότερο κόστος.

Τα reads χρησιμοποιούνται είτε για "χαρτογράφηση" σε γνωστό γονιδίωμα είτε για de novo συναρμολόγηση, δηλαδή δημιουργία του γονιδιώματος χωρίς αναφορά. Τα μικρά reads είναι ιδανικά για χαρτογράφηση, ενώ τα μεγάλα και ακριβή reads διευκολύνουν τη συναρμολόγηση νέων γονιδιωμάτων, αν και απαιτούν περισσότερους υπολογιστικούς πόρους και έχουν μεγαλύτερο κόστος.

2.2 Τυπική Διαδικασία Αλληλούχισης και Χαρτογράφησης του Γονιδιώματος

Οι περισσότερες τεχνολογίες αλληλούχισης δεν παρέχουν άμεσα τις αλληλουχίες σε μορφή DNA, αλλά σε ένα εγγενές σήμα που πρέπει να μετατραπεί σε αλληλουχία βάσεων (A, T, G, C) μέσω αλγορίθμων basecalling, που αποτελούν το πρώτο υπολογιστικό βήμα της ανάλυσης γονιδιώματος. Το basecalling μεταφράζει τα ηλεκτρικά ή φθορίζοντα σήματα σε αλληλουχίες DNA και η ακρίβεια και ταχύτητά του επηρεάζουν ολόκληρη τη διαδικασία. Κάθε τεχνολογία (Illumina, PacBio, Nanopore) χρησιμοποιεί διαφορετικά σήματα και μεθόδους basecalling, με τις πιο σύγχρονες προσεγγίσεις να βασίζονται σε τεχνητά νευρωνικά δίκτυα, μοντέλα Markov και εξειδικευμένες αρχιτεκτονικές υλικού για βελτιωμένη απόδοση. Μετά το basecalling, γίνεται έλεγχος ποιότητας (QC) κάθε αλληλουχίας και των επιμέρους βάσεων, ώστε να εντοπιστούν και να αποκοπούν περιοχές χαμηλής αξιοπιστίας, διασφαλίζοντας την ποιότητα των δεδομένων που θα χρησιμοποιηθούν στα επόμενα στάδια της ανάλυσης.[2]

Εφόσον έχει "μεταφραστεί" η αλληλουχία των reads σε χαρακτήρες, σειρά έχει η αντιστοίχισή τους με κομμάτια του γονιδιώματος αναφοράς από τα οποία πιθανότατα αυτά έχουν προέλθει. Πριν από αυτό, πρέπει όμως να διαθέτουμε μία οικονομική κωδικοποίηση και οργάνωση του γονιδιώματος αναφοράς, πράγμα που επιτυγχάνεται μέσω του indexing. Η διαδικασία του indexing αποτελεί βασικό στάδιο στην ανάλυση γονιδιωμάτων, όπου δημιουργείται μια μεγάλη βάση δεδομένων ευρετηρίου από υποαλληλουχίες (seeds) που εξάγονται από το γονιδίωμα αναφοράς, ώστε να επιτυγχάνεται γρήγορη και αποδοτική αναζήτηση. Ο mapper χρησιμοποιεί αυτό το ευρετήριο για να εντοπίσει πιθανές περιοχές του γονιδιώματος που ταιριάζουν με κάθε read, συγκρίνοντας τα seeds των reads με αυτά του ευρετηρίου. Η εξαγωγή των seeds γίνεται με τον ίδιο αλγόριθμο τόσο για το γονιδίωμα αναφοράς όσο και για τα reads, με τη διαφορά ότι το indexing αποθηκεύει τα seeds του αναφοράς, ενώ το seeding χρησιμοποιεί τα seeds των reads για αναζήτηση. Το indexing εκτελείται μία φορά και δεν αποτελεί σημείο συμφόρησης στην ανάλυση. Η δομή του ευρετηρίου περιλαμβάνει τα seeds ή αντίστοιχες τιμές κατακερματισμού (hash) μαζί με τις θέσεις τους, και η επιλογή του πλήθους, μήκους και συχνότητας των seeds επηρεάζει σημαντικά τη μνήμη, την απόδοση και την ακρίβεια της χαρτογράφησης.

Για τη βελτίωση της αποδοτικότητας, εφαρμόζονται τρεις βασικές στρατηγικές: καλύτερο sampling, πιο αποδοτικές δομές δεδομένων και μείωση της μετακίνησης δεδομένων μέσω εξειδικευμένου υλικού. Το sampling στοχεύει στην αποφυγή πλεονασμού, καθώς η επιλογή όλων των k-mers (υποαλληλουχιών μήκους k) δημιουργεί μεγάλο όγκο δεδομένων και πολλαπλές ταυτίσεις. Για αυτό, σύγχρονοι αλγόριθμοι όπως το minimap2 [3] επιλέγουν ένα υποσύνολο k-mers με τη μέθοδο των minimizers, όπου από κάθε ομάδα w επικαλυπτόμενων k-mers επιλέγεται το με τη μικρότερη τάξη (π.χ. λεξικογραφική ή βάση hash). Αυτή η μέθοδος διασφαλίζει ότι κάθε παράθυρο περιέχει τουλάχιστον ένα seed, μειώνοντας την απώλεια πληροφορίας. Εναλλακτικά, η μέθοδος των syncmers επιτυγχάνει πιο ομοιόμορφη κατανομή των seeds, επιλέγοντας minimizers με βάση υποαλληλουχίες σε σταθερή θέση, βελτιώνοντας την ανίχνευση θέσεων χαρτογράφησης που μπορεί να χάνονται με τους minimizers. Άλλες τεχνικές περιλαμβάνουν sampling minimizers χωρίς παράθυρα με χρήση διαφορετικών hash συναρτήσεων.

Για την αποθήκευση και αναζήτηση των seeds χρησιμοποιούνται κυρίως δομές κατακερματισμού (hash tables), όπου το hash κάθε seed λειτουργεί ως κλειδί και οι θέσεις ως τιμές. Η επιλογή της hash συνάρτησης είναι κρίσιμη για τη μείωση συγκρούσεων αλλά και για τη βελτίωση της ευαισθησίας μέσω ομαδοποίησης παρόμοιων seeds. Επιπλέον, προτείνονται και πιο συμπιεσμένες δομές, όπως ο FM-index, που μειώνουν το αποτύπωμα μνήμης επιτρέποντας αναζητήσεις seeds μεταβλητού μήκους. Τέλος, δεδομένου ότι indexing και seeding απαιτούν μεγάλη μνήμη και μετακίνηση δεδομένων, αναπτύσσονται ειδικές αρχιτεκτονικές με επεξεργασία εντός μνήμης (processing-in-memory) και υλικό επιτάχυνσης, όπως επιταχυντές σε DRAM ή SSD (GenStore [4]), για τη βελτίωση της απόδοσης και μείωση της κατανάλωσης ενέργειας.

Μετά τον εντοπισμό πιθανών θέσεων χαρτογράφησης ενός read στο γονιδίωμα αναφοράς, ο read mapper αξιολογεί την ομοιότητα μεταξύ του read και των αντίστοιχων τμημάτων του γονιδιώματος. Για να αποφευχθούν οι ακριβοί υπολογιστικά αλγόριθμοι ευθυγράμμισης σε μη όμοιες αλληλουχίες, χρησιμοποιούνται προ-ευθυγραμμιστικά φίλτρα (pre-alignment filters). Αυτα εκτιμούν γρήγορα την απόσταση επεξεργασίας (edit distance) μεταξύ δύο αλληλουχιών και αποφασίζουν αν απαιτείται λεπτομερής ευθυγράμμιση. Αν η απόσταση επεξεργασίας υπερβαίνει ένα όριο, οι αλληλουχίες θεωρούνται μη όμοιες και παρακάμπτεται η ευθυγράμμιση, εξοικονομώντας χρόνο.

Οι βασικές μέθοδοι προ-ευθυγράμμισης βασίζονται σε αρχές όπως αυτή της περιστεροφωλιάς, την καταμέτρηση βάσεων, το q-gram filtering και τη σποραδική δυναμική προγραμματισμό (sparse DP). Για παράδειγμα, η αρχή της περιστεροφωλιάς υποστηρίζει ότι αν δύο αλληλουχίες διαφέρουν κατά E επεξεργασίες, τότε πρέπει να έχουν κοινή υποαλληλουχία σε οποιοδήποτε σύνολο E+1 μη επικαλυπτόμενων υποαλληλουχιών. Ο αλγόριθμος SneakySnake [5] εφαρμόζει αυτή την αρχή μετατρέποντας το πρόβλημα σε πρόβλημα δρομολόγησης σε ολοκληρωμένο κύκλωμα, επιτυγχάνοντας αποδοτική προ-ευθυγράμμιση.

Η καταμέτρηση βάσεων συγκρίνει τον αριθμό κάθε βάσης (A, T, C, G) μεταξύ read και αναφοράς, απορρίπτοντας γρήγορα μη όμοιες αλληλουχίες. Το q-gram filtering εξετάζει όλα τα επικαλυπτόμενα υπο-strings μήκους q και υπολογίζει τον ελάχιστο αριθμό κοινών q-grams, επιτρέποντας γρήγορη απόρριψη διαφορών. Τέλος, οι αλγόριθμοι sparse DP χρησιμοποιούν ακριβείς κοινές περιοχές (seeds/anchors) για να περιορίσουν την ευθυγράμμιση μόνο στις μη καλυπτόμενες περιοχές, ενώ η τεχνική chaining συνδέει επικαλυπτόμενα seeds σε μεγαλύτερες αλυσίδες, μειώνοντας τον χώρο αναζήτησης για την τελική ευθυγράμμιση. Αυτή η διαδικασία αποτελεί συχνά ξεχωριστό στάδιο μεταξύ προ-ευθυγράμμισης και ευθυγράμμισης, όπως υλοποιείται στο minimap2.

Μετά την απόρριψη των μη όμοιων περιοχών χαρτογράφησης, η διαδικασία read mapping υπολογίζει την ευθυγράμμιση της αλληλουχίας του read με τα αντίστοιχα τμήματα του γονιδιώματος αναφοράς. Οι ευθυγραμμίσεις βασίζονται συνήθως σε αλγορίθμους δυναμικού προγραμματισμού με affine gap scoring, όπως ο Smith-Waterman, που προσφέρουν ευελιξία αλλά έχουν υψηλό υπολογιστικό κόστος. Για τη μείωση του χρόνου υπολογισμού, χρησιμοποιούνται τεχνικές όπως η επιβολή ορίου στην απόσταση επεξεργασίας (edit distance threshold) και ο περιορισμός του υπολογισμού σε συγκεκριμένες διαγώνιες περιοχές του πίνακα DP μειώνοντας την πολυπλοκότητα.

Για την επιτάχυνση, έχουν αναπτυχθεί υλοποιήσεις με επιτάχυνση σε υλικό (CPU, GPU, FPGA, ASIC) που επιτρέπουν παράλληλο υπολογισμό πολλών ευθυγραμμίσεων, ενώ άλλοι αλγόριθμοι θυσιάζουν την ακρίβεια για ταχύτητα, όπως οι Myers' bit-vector, Darwin και GenASM. Εναλλακτικές μέθοδοι περιορίζουν τα σημεία υπολογισμού στον πίνακα DP με τεχνικές sparse DP ή greedy, αλλά ενδέχεται να παράγουν υποβέλτιστες ευθυγραμμίσεις. Στο τέλος, μέσω της διαδικασίας traceback προσδιορίζεται η βέλτιστη ευθυγράμμιση και περιγράφεται με τη μορφή CIGAR string, που αναπαριστά τις θέσεις και τους τύπους των ταυτίσεων, υποκαταστάσεων, εισαγωγών και διαγραφών. Συνολικά, η ευθυγράμμιση αποτελεί το πιο απαιτητικό υπολογιστικά στάδιο της ανάλυσης γονιδιώματος και η επιτάχυνσή της είναι κρίσιμη για τη βελτίωση της συνολικής απόδοσης της διαδικασίας.

2.3 Γονιδιωματικοί Γράφοι

Η κυρίαρχη αναπαράσταση του γονιδιώματος αναφοράς είναι η γραμμική, αλλά αυτό παρουσιάζει περιορισμούς λόγω της μεγάλης γενετικής ποικιλότητας μεταξύ των ανθρώπινων πληθυσμών. Ένα τυπικό γονιδίωμα περιέχει εκατομμύρια μικρές παραλλαγές (SNPs, indels) και χιλιάδες μεγάλες δομικές μεταβολές (SVs) σε σχέση με το αναφοράς, οι οποίες μπορούν να προκαλέσουν σφάλματα στη χαρτογράφηση των reads, ιδιαίτερα όταν αυτά επικαλύπτουν σημεία θραύσης SVs. Αυτό οδηγεί σε μειωμένη ακρίβεια χαρτογράφησης, με συνέπειες όπως ψευδώς αρνητικά ή θετικά αποτελέσματα και προβλήματα σε επόμενα βήματα ανάλυσης.

Για την αντιμετώπιση αυτών των προκλήσεων, έχουν εισαχθεί τα γονιδιωματικά γραφήματα (genome graphs), που ενσωματώνουν το αναφοράς γονιδίωμα μαζί με τη γενετική ποικιλότητα και πολυμορφισμούς, προσφέροντας μια πιο ολοκληρωμένη αναπαράσταση της γενετικής διαφοροποίησης. Τα γραφήματα αυτά βελτιώνουν την ευθυγράμμιση των reads και την ανάλυση των αλληλόμορφων, ιδιαίτερα σε πληθυσμούς με μεγάλη γενετική ποικιλότητα, όπως οι Αφρικανικοί.

Σήμερα, πολλά εργαλεία ανάλυσης γονιδιωμάτων χρησιμοποιούν γραφήματα ως αναφορά αντί για γραμμικές αλληλουχίες. Η κατασκευή του γραφήματος γίνεται συνδυάζοντας το αναφοράς γονιδίωμα με παραλλαγές από βάσεις δεδομένων, δημιουργώντας εναλλακτικές διαδρομές που αντιπροσωπεύουν τις γενετικές παραλλαγές. Υπάρχουν διάφορα εργαλεία για την κατασκευή αυτών των γραφημάτων, όπως το VG toolkit [6] και το minigraph. Στην παρούσα εργασία, όπως και σε πολλές ευθυγραμμιστικές μεθόδους, δεν λαμβάνονται υπόψη οι αναστροφές και διπλασιασμοί, καθώς δεν περιγράφονται εύκολα με τη μορφή CIGAR.

2.4 Προγραμματιζόμενες Πύλες Πεδίου (FPGAs)

Οι Προγραμματιζόμενες Πύλες Πεδίου (FPGAs) είναι υπολογιστές πλατφόρμες χρησιμοποιούμενες για την επιτάχυνση υπολογιστικά εντατικών εργασιών. Αποτελούνται από διάφορα διαμορφώσιμα block (τα οποία περιλαμβάνουν πίνακες αναζήτησης (LUTs), καταχωρητές, block RAMs και DSPs) σχεδιασμένα γα την υλοποίηση Boolean συναρτήσεων. Για τον προγραμματισμό τους χρησιμοποιούνται οι γλώσσες περιγραφής υλικού όπως οι VHDL και Verilog, αλλά και η Υψηλού Επιπέδου Σύνθεση η οποία επιτρέπει στους προγραμματιστές τη χρήση γλωσσών υψηλού επιπέδου όπως η C και η C++, την οποία μετατρέπει σε γλώσσα περιγραφής υλικού.

Η Υψηλού Επιπέδου Σύνθεση (High-Level Synthesis - HLS) είναι μια αυτοματοποιημένη διαδικασία που μετατρέπει μια αφηρημένη περιγραφή συμπεριφοράς ενός ψηφιακού συστήματος σε μια υλοποίηση σε επίπεδο καταχωρητών (register-transfer level - RTL). Η HLS επιτρέπει στον σχεδιαστή να εργάζεται σε υψηλότερο επίπεδο αφαίρεσης, διευκολύνοντας τη δημιουργία υψηλής απόδοσης υλικού με καλύτερο έλεγχο της βελτιστοποίησης. Η διαδικασία περιλαμβάνει τρία βασικά στάδια: κατανομή πόρων (allocation), δέσμευση λειτουργιών σε πόρους (binding) και προγραμματισμό εκτέλεσης λειτουργιών (scheduling). Το τελικό αποτέλεσμα είναι μια RTL περιγραφή που μετατρέπεται σε βέλτιστο κύκλωμα πύλης.

Η HLS χρησιμοποιείται συχνά για την υλοποίηση αλγορίθμων σε επαναπρογραμματιζόμενες πλατφόρμες, όπως τα FPGAs, που προσφέρουν υψηλή ταχύτητα λόγω της παράλληλης φύσης τους και της βέλτιστης χρήσης πυλών. Τα FPGAs αποτελούνται από πίνακες λογικών μονάδων (CLBs), I/O pads και κανάλια δρομολόγησης, και προγραμματίζονται μέσω γλωσσών περιγραφής υλικού όπως Verilog και VHDL. Ένα δημοφιλές εργαλείο HLS είναι το Vivado HLS της Xilinx, που αυτοματοποιεί τη διαδικασία μετατροπής της περιγραφής σε RTL και στη συνέχεια σε bitstream για FPGA. Κύριο πλεονέκτημα των εργαλείων Υψηλού Επιπέδου Σύνθεσης (HLS) είναι η αποτελεσματική χρήση πόρων σε συνδυασμό με υψηλή απόδοση, επιτρέποντας στον σχεδιαστή να προσθέτει ειδικές οδηγίες (pragmas) που καθοδηγούν τη σύνθεση για τη δημιουργία βελτιστοποιημένων IP blocks. Αυτές οι οδηγίες μπορούν να εφαρμοστούν σε συναρτήσεις, βρόχους και πίνακες και επηρεάζουν παραμέτρους όπως ο χώρος (area), η καθυστέρηση (latency), το διάστημα εκκίνησης (initiation interval) και η κατανομή πόρων. Για παράδειγμα, οι οδηγίες που σχετίζονται με συναρτήσεις επιτρέπουν την παράλληλη εκτέλεση (dataflow, pipeline) ή την κατάργηση της ιεραρχίας για μείωση του overhead (inline). Οι οδηγίες για βρόχους μπορούν να μειώσουν το κόστος μετάβασης και να επιτρέψουν παράλληλη εκτέλεση πολλαπλών επαναλήψεων (unroll, pipeline), ενώ οι οδηγίες για πίνακες (partition, reshape) βελτιώνουν την πρόσβαση αφαιρώντας περιορισμούς των block-RAM.

3 Προτεινόμενη Αλγοριθμική Ροή και Αρχιτεκτονικές του Επιταχυντή

3.1 Ανάλυση των Γονιδιωματικών Γράφων Αναφοράς και του Συνόλου των Reads προς Αλληλούχιση

Η ανάλυση (profiling) των γονιδιωματικών γράφων αναφοράς είναι απαραίτητη για την ποσοτική και ποιοτική αξιολόγηση της δομής, του μεγέθους και της πολυπλοκότητας τους, καθώς και για τη βέλτιστη επιλογή παραμέτρων στη μικροαρχιτεκτονική που θα αναπτυχθεί. Επίσης, εξετάζεται η κατανομή των τελικών alignments των reads στο γράφημα, όπως προκύπτουν από το state-of-the-art λογισμικό sequence-to-graph alignment, GraphAligner. Τα γονιδιωματικά γραφήματα αναφοράς βασίζονται στην ανθρώπινη γονιδιωματική συναρμολόγηση GRCh38 και σε επιπλέον αρχεία παραλλαγών (VCFs) από το GIAB project, παράγοντας 24 γραφήματα, ένα για κάθε αυτοσωμικό χρωμόσωμα και δύο για τα φυλετικά χρωμοσώματα X και Y. Κάθε κόμβος του γραφήματος περιέχει πληροφορίες για την αλληλουχία του, τους γονείς και τα παιδιά του, καθώς και τον αριθμό αυτών. Παρατηρείται ότι ο αριθμός των κόμβων και το συνολικό μήκος της αλληλουχίας μειώνονται με την αύξηση του αριθμού του χρωμοσώματος, με εξαίρεση το χρωμόσωμα X. Το γράφημα του χρωμοσώματος Υ είναι γραμμικό, με κόμβους συνδεδεμένους μόνο με τους άμεσους γείτονές τους. Η ανάλυση του χρωμοσώματος 22 δείχνει ότι οι περισσότεροι κόμβοι έχουν μικρή αλληλουχία (συνήθως ένα νουκλεοτίδιο), αν και υπάρχουν και κόμβοι με πολύ μεγάλες αλληλουχίες που δεν βρίσκονται σε συγκεκριμένες περιοχές του γράφου. Αυτό σημαίνει ότι κατά την αντιστοίχιση μικρών reads δεν χρειάζεται να λαμβάνονται ολόκληρες μεγάλες αλληλουχίες κόμβων, κάτι που βελτιώνει την αποδοτικότητα. Η κατανομή των γονέων και παιδιών κάθε κόμβου είναι συνήθως μικρή, με τους περισσότερους κόμβους να έχουν μόνο έναν γονέα και ένα παιδί, λόγω της φύσης των παραλλαγών (SNVs). Επίσης, εξετάστηκε η κατανομή των hops (άλματα μεταξύ κόμβων), που είναι κυρίως μικρά, γεγονός που υποστηρίζει την ακρίβεια του σχεδιασμού χωρίς την ανάγκη περιορισμού τους. Σχετικά με τους minimizers (μικρές υποαλληλουχίες που χρησιμοποιούνται για γρήγορη αναζήτηση), οι περισσότεροι εμφανίζονται μόνο μία φορά στον γράφο αναφοράς, αλλά κάποιοι υπάρχουν σε χιλιάδες θέσεις, γεγονός που απαιτεί φιλτράρισμα ή περιορισμό του αριθμού των seeds για διατήρηση της ακρίβειας. Επιπλέον, η κατανομή των minimizers δεν είναι ομοιόμορφη, με τα περισσότερα να ξεκινούν από το γράμμα A, γεγονός που δημιουργεί προκλήσεις στην ισορροπημένη παράλληλη αναζήτηση. Τέλος, η ανάλυση των τελικών ευθυγραμμίσεων διαφορετικών συνόλων reads (μεγάλου μήκους από PacBio και μικρού από Illumina) δείχνει ότι τα άλματα αυξάνονται με το μήκος των reads, αλλά μόνο λίγα reads μεγάλου μήκους ευθυγραμμίζονται σε μεγάλα υπογραφήματα. Για τα μικρά reads, η επιβολή ορίου στα hops δεν επηρεάζει την ακρίβεια, καθώς τα πραγματικά hops στις ευθυγραμμίσεις είναι πολύ μικρότερα από το μέγιστο δυνατό που βρίσκουμε στο γράφημα. Τα παραπάνω συμπεράσματα για την μορφή του γράφου και του index του με χρήση των minimizers, συμβάλουν στον καλύτερο σχεδιασμό του επιταχυντή ενώ δίνει ποσοτικές πληροφορίες οριών δομών μέσα στον kernel που θα χτίσουμε στο hardware εφόσον τα μεγέθη π.χ. των τοπικών πινάκων πρέπει να είναι από πριν δοσμένα.

3.2 Ροή Εργασίας και Αλγόριθμοι

3.2.1 Κατασκευή, Οργάνωση και Indexing του Γράφου Αναφοράς

Η κατασκευή του γονιδιωματικού γράφου βασίζεται στη συνένωση πολλαπλών γραμμικών γονιδιωμάτων, η οποία γίνεται με χρήση του εργαλείου vg. Για να είναι δυνατή η ορθή χαρτογράφηση και ευθυγράμμιση, το γράφημα πρέπει να είναι τοπολογικά ταξινομημένο, δηλαδή οι κόμβοι να ακολουθούν μια γραμμική σειρά όπου κάθε ακμή οδηγεί από έναν κόμβο με μικρότερο δείκτη (nodeID) σε έναν με μεγαλύτερο. Η τοπολογική ταξινόμηση επιτυγχάνεται με αλγορίθμους γραμμικής πολυπλοκότητας που εφαρμόζονται σε κατευθυνόμενα ακυκλικά γραφήματα πάλι με χρήση του vg [6].

Μετά την κατασκευή του γραφήματος, πραγματοποιείται η διαδικασία ευρετηρίασης (indexing) που γίνεται μία φορά για κάθε γράφημα αναφοράς. Το γράφημα αποτελείται από κόμβους, καθένας με μοναδικό αναγνωριστικό (nodeID), λίστες γειτονικών κόμβων παιδιών (out_Nodes) και προγόνων (in_Nodes), καθώς και την αλληλουχία DNA που αντιπροσωπεύει. Για την αποθήκευση και αποτελεσματική πρόσβαση στις πληροφορίες αυτές, χρησιμοποιούνται πίνακες που περιλαμβάνουν τα nodeIDs, τις αλληλουχίες, τα nodeIDs των γειτόνων και τα μήκη αυτών. Για τη βελτίωση της αποδοτικότητας στην αναζήτηση πιθανών περιοχών ευθυγράμμισης, από τις αλληλουχίες του γραφήματος εξάγονται minimizers, δηλαδή μικρές υποαλληλουχίες που επιλέγονται ως οι λεξικογραφικά μικρότερες k-mers σε κάθε παράθυρο. Οι minimizers αποθηκεύονται μαζί με τη θέση τους (nodeID και μετατόπιση/offset) και οργανώνονται σε ομάδες ώστε να μειωθεί ο χρόνος αναζήτησης και η πρόσβαση στη μνήμη. Αντί να απορρίπτονται οι πιο συχνοί minimizers, τίθεται όριο στον αριθμό των seeds που μπορεί να ρυθμιστεί από τον χρήστη, με την προειδοποίηση ότι η μείωση αυτού του ορίου μπορεί να επηρεάσει αρνητικά την ακρίβεια του alignment. Τέλος, για την αποθήκευση των αλληλουχιών όλων των κόμβων σε μία ενιαία δομή, οι αλληλουχίες συνενώνονται σε έναν πίνακα χαρακτήρων και χρησιμοποιούνται αθροιστικά μήκη για να εντοπίζονται οι αρχικές και τελικές θέσεις της αλληλουχίας κάθε κόμβου μέσα σε αυτόν τον πίνακα. Όλες αυτές οι διαδικασίες αποτελούν τα προ-επεξεργαστικά βήματα που εκτελούνται μία φορά για κάθε γράφημα αναφοράς και φορτώνονται στη μνήμη για χρήση κατά τη διάρκεια της στοίχισης.

3.2.2 Το βήμα του Seeding και Εξαγωγή Υποψήφιου Υπογράφου προς Στοίχιση

Η διαδικασία εξαγωγής minimizers από τη σειρά reads γίνεται με τον ίδιο τρόπο όπως και για το γονιδιωματικό γράφημα αναφοράς, επιλέγοντας το λεξικογραφικά μικρότερο k-mer μέσα σε κάθε παράθυρο. Το πλήθος των minimizers που εξάγονται από ένα read μπορεί να υπολογιστεί με βάση το μήκος του read, το μήκος του k-mer και το μέγεθος του παραθύρου, λαμβάνοντας υπόψη ότι το παράθυρο μετακινείται με βήμα μεγαλύτερο του ενός για μείωση της επανάληψης. Κάθε minimizer αποθηκεύεται μαζί με τη θέση εκκίνησής του μέσα στο read. Στη συνέχεια, πραγματοποιείται αναζήτηση των minimizers στον ευρετήριο του γονιδιωματικού γραφήματος, η οποία γίνεται μέσα σε ομάδες (buckets) που ομαδοποιούν minimizers με κοινά πρώτα δύο γράμματα ή το πρώτο γράμμα, μειώνοντας τον χρόνο αναζήτησης και τις προσβάσεις στη μνήμη. Όταν βρεθεί ένας minimizer στο ευρετήριο, καταγράφονται τρεις ακέραιες τιμές που προσδιορίζουν τη θέση του στο read, τη θέση μέσα στον κόμβο του γραφήματος και το αναγνωριστικό του κόμβου. Επειδή ένας minimizer μπορεί να εμφανίζεται σε πολλές θέσεις του γραφήματος, τίθεται όριο στον αριθμό των θέσεων που διατηρούνται για περαιτέρω επεξεργασία, το οποίο μπορεί να ρυθμιστεί από τον χρήστη. Οι θέσεις στις οποίες βρίσκεται ένα minimizer στο γονιδίωμα αναφοράς ονομάζονται seed hits. Μετά τον εντοπισμό των seed hits, υπολογίζονται οι σχετικές θέσεις έναρξης και λήξης μέσα στον κόμβο, ώστε να προσδιοριστεί η περιοχή του γραφήματος που θα χρησιμοποιηθεί για την στοίχιση. Αυτή η περιοχή πρέπει να περιλαμβάνει αρκετούς χαρακτήρες για να ευθυγραμμιστεί ολόκληρο το read, λαμβάνοντας υπόψη ένα όριο λαθών (edit threshold). Οι σχετικές θέσεις έναρξης και λήξης ορίζονται με βάση τις θέσεις του minimizer μέσα στο read και στον κόμβο, προσαρμοσμένες ώστε να καλύπτουν ολόκληρη την αλληλουχία. Στη συνέχεια, προσδιορίζονται τα υπογραφήματα (subgraphs) που είναι υποψήφια για αντιστοίχιση. Υπάρχουν τέσσερις βασικές περιπτώσεις:

1. Όταν ο κόμβος του seed hit έχει αρκετούς χαρακτήρες για να ευθυγραμμίσει ολόκληρο το read.
2. Όταν απαιτείται επέκταση του υπογραφήματος προς τα αριστερά, επειδή δεν υπάρχουν αρκετοί χαρακτήρες πριν από το seed hit.
3. Όταν απαιτείται επέκταση προς τα δεξιά, επειδή δεν υπάρχουν αρκετοί χαρακτήρες μετά το seed hit.
4. Όταν απαιτείται επέκταση και προς τις δύο κατευθύνσεις, για να καλυφθεί ολόκληρο το read.

Στις περιπτώσεις που απαιτείται επέκταση, η διαδικασία εξετάζει αναδρομικά τους γειτονικούς κόμβους (inNodes ή outNodes) για να βρει επαρκείς χαρακτήρες ώστε να καλυφθεί ολόκληρη η αλληλουχία. Για λόγους υλοποίησης σε υλικό, η αναδρομή αντικαθίσταται με χρήση πινάκων που αποθηκεύουν τους κόμβους προς εξέταση και τα υπόλοιπα απαιτούμενα μήκη, ώστε να αποφευχθεί η χρήση σύνθετων αναδρομικών αλγορίθμων. Συγκεκριμένα, χρησιμοποιούμε τους πίνακες που εμπεριέχουν τα μήκη των ακολουθιών των γειτόνων και αφαιρούμε από αυτά τις σχετικές θέσεις έναρξης και λήξης ανάλογα με την περίπτωση στην οποία υπάγεται το seed hit.Μέχρις ώτου να έχουμε θετικό αποτέλεσμα σε αυτές τις διαφορές (δηλαδή επαρκή αριθμό χαρακτήρων για την στοίχιση του read), επαναλαμβάνουμε τη διαδικασία για τα κομμάτια των πινάκων που αντιστοιχούν στους κόμβους των γειτόνων αποθηκεύοντας τα αναγνωριστικά των κόμβων και τα offsets που οι διαφορές αυτές είναι θετικές, ως συντεταγμένες υποψήφιων υπογράφων στοίχισης. Τέλος, όλες αυτές οι δυνατές συνδυαστικές θέσεις έναρξης και λήξης που προκύπτουν από τις επεκτάσεις θεωρούνται ως ξεχωριστά υπογραφήματα υποψηφίων ευθυγράμμισης, τα οποία περνούν για περαιτέρω επεξεργασία. Με αυτόν τον τρόπο εξασφαλίζεται ότι το υπογράφημα που επιλέγεται περιλαμβάνει επαρκή αριθμό χαρακτήρων για την αξιόπιστη ευθυγράμμιση του read. Για να πραγματοποιηθεί η στοίχιση, απαιτείται ενδιάμεσο βήμα που εντοπίζει όλες τις δυνατές διαδρομές μέσα στο υπογράφημα και συνενώνει τις αλληλουχίες των κόμβων αυτών σε μία ενιαία αλληλουχία εισόδου για τον αλγόριθμο του sequence-to-sequence alignment. Στην περίπτωση του sequence-to-graph alignment το βήμα εύρεσης των μονοπατιών του υπογράφου παρακάμπτεται και όπως θα δούμε συνενώνουμε τις αλληλουχίες των κόμβων του υπογράφου ακολουθώντας την σειρά των nodeIDs τους μετά από μία διαδικασία φιλτραρίσματος. Για την εύρεση των διαδρομών, χρησιμοποιούνται οι πίνακες με τα nodeIDs των γειτονικών κόμβων που έχουν φορτωθεί από τη μνήμη. Οι διαδρομές αποθηκεύονται ως ακολουθίες nodeIDs και στη συνέχεια περνούν σε συνάρτηση που συνενώνει τις αντίστοιχες αλληλουχίες DNA των κόμβων. Ο αλγόριθμος επέκτασης του υπογραφήματος διασφαλίζει ότι τουλάχιστον μία από τις διαδρομές έχει αρκετό μήκος για να ευθυγραμμίσει ολόκληρο το read, ενώ οι διαδρομές που δεν πληρούν αυτή την προϋπόθεση απορρίπτονται Με αυτόν τον τρόπο, το σύστημα προετοιμάζει κατάλληλα τα δεδομένα ώστε να εισαχθούν στον αλγόριθμο στοίχισης αλληλουχιών, εξασφαλίζοντας την αποδοτική και ακριβή αντιστοίχιση των reads στο γονιδιωματικό γράφημα.

3.2.3 Μέθοδοι Αντιστοίχισης Αλληλουχίας

Η μέχρι εδώ διαδικασία πραγματοποιείται και για τις δύο διαφορετικές αρχιτεκτονικές, οι οποίες στη συνέχεια θα διαχωριστούν με βάση το είδος του aligner που θα χρησιμοποιηθεί. Στην πρώτη εκδοχή του προτεινόμενου συστήματος με χρήση sequence-to-sequence alignement, χρησιμοποιείται ένας αλγόριθμος ευθυγράμμισης αλληλουχιών βασισμένος στον αλγόριθμο Bitap [7], ο οποίος ανήκει στην κατηγορία των αλγορίθμων Προσεγγιστικής Ταύτισης Συμβολοσειρών (Approximate String Matching). Ο Bitap αντικαθιστά τους παραδοσιακούς αλγορίθμους δυναμικού προγραμματισμού χρησιμοποιώντας bitwise πράξεις, επιτρέποντας τον υπολογισμό της ελάχιστης απόστασης επεξεργασίας (edit distance) μεταξύ μιας αλληλουχίας αναφοράς και ενός ερωτήματος (π.χ. read), επιτρέποντας μέχρι k λάθη. Όταν k=0, ο αλγόριθμος εκτελεί ακριβή ταύτιση. Η λειτουργία του Bitap ξεκινά με μια φάση προεπεξεργασίας όπου το read μετατρέπεται σε bitmasks για κάθε χαρακτήρα του αλφαβήτου του DNA, επιτρέποντας την δυαδική αναπαράσταση της αλληλουχίας. Κατά την εκτέλεση, διατηρούνται bit vectors που αποθηκεύουν πληροφορίες μερικής ταύτισης για κάθε πιθανό αριθμό λαθών από 0 έως k. Ο αλγόριθμος επεξεργάζεται διαδοχικά κάθε χαρακτήρα του κειμένου, ενημερώνοντας τα bit vectors με bitwise λειτουργίες που αντιστοιχούν σε εισαγωγές, διαγραφές, υποκαταστάσεις και ακριβείς ταυτίσεις. Η ανίχνευση ταυτίσεων γίνεται όταν το πιο σημαντικό bit ενός bit vector γίνεται μηδέν, υποδεικνύοντας θέση με edit distance d. Μετά τον εντοπισμό της θέσης και της απόστασης επεξεργασίας, ένας δεύτερος αλγόριθμος (GenASM-TB) εκτελεί traceback, δηλαδή αναδρομική ανάλυση των bit vectors για να προσδιορίσει την ακριβή ακολουθία των τύπων αλλαγής (ταύτιση, υποκατάσταση, εισαγωγή, διαγραφή) που οδήγησαν στην αντιστοίχιση, παράγοντας την περιγραφή CIGAR. Η χρήση του Bitap και των βελτιώσεών του επιτρέπει αποδοτική, παράλληλη και χαμηλής κατανάλωσης ενέργειας υλοποίηση για την επιτάχυνση κρίσιμων βημάτων της ανάλυσης γονιδιωμάτων, όπως η ευθυγράμμιση reads, ο υπολογισμός edit distance και η προ-φιλτράρισμα ευθυγράμμισης. Για την δεύτερη εκδοχή του προτεινόμενου ολοκληρωμένου συστήματος, χρησιμοποιείται μία παραλλαγμένη μορφή του Bitap, ο αλγοριθμος BitAlign [8]. Πριν όμως από το βήμα της στοίχισης, εισάγουμε μία διαδικασία φιλτραρίσματος των κόμβων του υποψήφιου υπογράφου που έχει εξαθχεί από το βήμα του seeding. Η διαδικασία φιλτραρίσματος υπογραφήματος εκμεταλλεύεται την ταξινομημένη φύση των γραφημάτων και περιλαμβάνει την απομάκρυνση κόμβων που δεν έχουν τουλάχιστον μία ακμή, είτε εισερχόμενη είτε εξερχόμενη, που συνδέεται με κάποιον κόμβο του υπογραφήματος. Κόμβοι που δεν είναι προσβάσιμοι από άλλους κόμβους με μικρότερο αναγνωριστικό ή δεν οδηγούν σε κόμβους με μεγαλύτερο nodeID μέσα στον υπογράφο απορρίπτονται, καθώς δεν συμμετέχουν στην ευθυγράμμιση. Επιπλέον, απομακρύνονται κόμβοι των οποίων το μήκος της αλληλουχίας είναι μεγαλύτερο από τους χαρακτήρες που απομένουν να ευθυγραμμιστούν, ώστε να αποφευχθεί η επεξεργασία περιττών δεδομένων. Έτσι, δημιουργούνται μικρότερα και πιο στοχευμένα υπογραφήματα. Μετά το φιλτράρισμα, πραγματοποιείται η γραμμικοποίηση του υπογραφήματος, δηλαδή η συνένωση των αλληλουχιών όλων των κόμβων από τον κόμβο έναρξης μέχρι τον κόμβο λήξης, αξιοποιώντας την προγενέστερη ταξινόμηση. Παράλληλα, δημιουργούνται bitvectors που σημειώνουν τις θέσεις έναρξης και λήξης των hops μεταξύ κόμβων, τα οποία είναι σημαντικά για τον αλγόριθμο ευθυγράμμισης BitAlign, ο οποίος εκμεταλλεύεται τη γραμμικοποιημένη αλληλουχία του υπογραφήματος, τα bitvectors θέσεων, το όριο απόστασης επεξεργασίας και την αλληλουχία του read για να υπολογίσει την ελάχιστη απόσταση επεξεργασίας και να παράγει την περιγραφή ευθυγράμμισης (CIGAR string). Ο BitAlign λειτουργεί με bitwise λειτουργίες όπως ο Bitap, αλλά επιπλέον λαμβάνει υπόψη τις θέσεις των hops μέσω επιπρόσθετων bitvectors, ενσωματώνοντας τις διακλαδώσεις του γραφήματος στην ευθυγράμμιση. Η διαδικασία ολοκληρώνεται με το traceback, που χρησιμοποιεί τα ενδιάμεσα bitvectors για να ανακατασκευάσει την τελική ευθυγράμμιση.

3.3 Αρχιτεκτονική των Επιταχυντών

3.3.1 Αρχιτεκτονική του Ολοκληρωμένου Συστήματος με χρήση Sequence-to-Sequence Alignment

Η προτεινόμενη σχεδίαση αποτελείται από τρία διακριτά blocks που εκτελούν διαδοχικά τα βασικά βήματα της διαδικασίας: την εξαγωγή seeds, τη γραμμικοποίηση υπογραφήματος και την ευθυγράμμιση. Η κεντρική μονάδα CPU παρέχει ομάδες (batches) από reads, κάθε μία με N reads μήκους m, στο κομμάτι εύρεσης seeds, το οποίο εντοπίζει όλα τα υπογραφήματα του γονιδιωματικού γραφήματος που είναι πιθανά για τη στοίχιση κάθε read. Η εξαγωγή seeds περιλαμβάνει τρία υποβήματα: εξαγωγή minimizers από τα reads, αναζήτηση αυτών στο minimizers' index του γραφήματος και εντοπισμό υπογραφήματος αρκετού μεγέθους για την ευθυγράμμιση κάθε read. Οι συντεταγμένες των υπογραφήματος (κόμβοι έναρξης και λήξης και αντίστοιχες θέσεις μέσα στις αλληλουχίες κόμβων) περνούν στο block γραμμικοποίησης (linearization block), το οποίο παράγει την ενιαία αλληλουχία αναφοράς για την ευθυγράμμιση. Για μικρά reads, η γραμμικοποίηση αφορά συνήθως υπογραφήματα ενός μόνο κόμβου, ενώ για μεγαλύτερα υπογραφήματα περιλαμβάνει και εύρεση διαδρομών πριν τη συνένωση των αλληλουχιών. Τέλος, το alignment block (aligner) χρησιμοποιεί την παραγόμενη αλληλουχία για να ευθυγραμμίσει κάθε read, παράγοντας ένα ανοιχτό CIGAR string και τις συντεταγμένες ευθυγράμμισης, τα οποία επιστρέφονται στον host. Η CPU ολοκληρώνει τη μορφοποίηση του CIGAR string. Τα block εξαγωγής seeds (seeder) και γραμμικοποίησης (linearization block) ανακτούν δεδομένα από την κύρια μνήμη, ενώ ο aligner λειτουργεί με δεδομένα που βρίσκονται σε εσωτερικούς πίνακες του πυρήνα. Όλα τα block μπορούν να λειτουργήσουν είτε ενσωματωμένα σε έναν ενιαίο πυρήνα είτε ως ανεξάρτητοι kernels, με την CPU να παρέχει τα κατάλληλα δεδομένα σε κάθε περίπτωση.

Η αρχιτεκτονική του Seeder αποτελείται από τρία βασικά blocks: εξαγωγή minimizers, αναζήτηση minimizers και επέκταση υπογραφήματος. Υπάρχουν δύο εκδόσεις του Seeder, με και χωρίς read scratchpad. Η χρήση scratchpad για την αποθήκευση ομάδων reads αυξάνει την απόδοση και τη χρήση πόρων, καθώς επιτρέπει την παράλληλη επεξεργασία πολλών reads αντί για ένα κάθε φορά. Στο block εξαγωγής minimizers, τα reads χωρίζονται παράλληλα σε παράθυρα και από κάθε παράθυρο εξάγεται ο αντίστοιχος minimizer. Η ολίσθηση του παραθύρου γίνεται με βήμα μεγαλύτερο του ενός για μείωση της επανάληψης, και μάλιστα ίσου με τη διαφορά του μεγέθους του παραθύρου και του μήκους των minimizers. Τα minimizers αποθηκεύονται σε scratchpad μαζί με τις θέσεις τους και τα αναγνωριστικά των reads από το οποίο προήλθαν. Στη συνέχεια, τα minimizers κατηγοριοποιούνται σε ομάδες βάσει των δύο πρώτων χαρακτήρων τους για αυτά που ξεκινούν από A ή C και βάση του πρώτου χαρακτήρα αν αυτός είναι G ή T,, ώστε να εξισορροπηθεί η μη ισορροπημένη κατανομή τους και να μειωθεί ο χρόνος αναζήτησης. Δημιουργούνται δέκα ομάδες minimizers, με διαφορετική scratchpad για κάθε ομάδα, οι οποίες τροφοδοτούν αντίστοιχα υπο-block αναζήτησης που λειτουργούν παράλληλα, χρησιμοποιώντας ξεχωριστές διεπαφές μνήμης για αποδοτική πρόσβαση σε διαφορετικά τμήματα της μνήμης (χρήση m_axi διεπαφής). Η αναζήτηση πραγματοποιείται σε batches, όπου ενώ γίνεται αναζήτηση στο τρέχον batch, το επόμενο προφορτώνεται, επιτυγχάνοντας διπλή ροή (double buffering). Όταν βρεθεί minimizer στο index αναφοράς, ανακτώνται οι θέσεις του στο γράφημα αναφοράς, οι οποίες περιορίζονται από όριο που ορίζει ο χρήστης για τον αριθμό των seed hits. Τα seed hits χαρακτηρίζονται από τέσσερις τιμές: θέση έναρξης στο read, θέση έναρξης στον κόμβο του γραφήματος, αναγνωριστικό κόμβου και αναγνωριστικό του read. Υπολογίζονται οι σχετικές θέσεις έναρξης και λήξης κάθε seed hit, οι οποίες καθορίζουν σε ποια από τις τέσσερις κατηγορίες επέκτασης υπογραφήματος ανήκει κάθε seed. Ανάλογα με την κατηγορία, τα seed hits αποθηκεύονται σε διαφορετικές scratchpads και περνούν σε αντίστοιχα υπο-block που εκτελούν την επέκταση προς τα αριστερά, δεξιά, και στις δύο κατευθύνσεις ή καμία επέκταση αντίστοιχα. Όλα τα υπο-block επέκτασης εκτελούνται παράλληλα, με κατάλληλη διαχείριση μνήμης ώστε να αποφευχθούν συγκρούσεις πρόσβασης, χρησιμοποιώντας διαφορετικές διεπαφές μνήμης και δείκτες για τα κοινά δεδομένα. Τα τελικά αποτελέσματα των επεκτάσεων αποθηκεύονται σε scratchpad και είτε επιστρέφονται στον host όταν ο Seeder λειτουργεί ανεξάρτητα, είτε περνούν στο επόμενο στάδιο γραμμικοποίησης όταν ο Seeder αποτελεί μέρος ολοκληρωμένης σχεδίασης.

Το δεύτερο block της ολοκληρωμένης σχεδίασης είναι το block γραμμικοποίησης, το οποίο είναι υπεύθυνο για την εύρεση των διαδρομών μέσα σε υπογραφήματα που αποτελούνται από πολλούς κόμβους και τη συνένωση των αλληλουχιών των κόμβων αυτών σε μία ενιαία γραμμική αλληλουχία αναφοράς. Επιπλέον, καλύπτει και την περίπτωση υπογραφήματος ενός μόνο κόμβου, όπου απλώς ανακτά από τη μνήμη το αντίστοιχο τμήμα της αλληλουχίας που αντιστοιχεί στον κόμβο του seed hit και τις αντίστοιχες θέσεις εκκίνησης και λήξης. Η επιλογή του υπο-block που θα χρησιμοποιηθεί για κάθε υπογράφημα εξαρτάται από το αν το υπογράφημα αποτελείται από έναν κόμβο ή από πολλούς κόμβους. Στην περίπτωση του υπογραφήματος ενός κόμβου, το αντίστοιχο υπο-block χρησιμοποιεί τον πίνακα αθροιστικών/σωρευτικών (cumulative) μηκών για να εντοπίσει τις θέσεις της αλληλουχίας που πρέπει να ανακτηθούν, προσθέτοντας την αρχική μετατόπιση στο σωρευτικό μήκος του κόμβου. Η αλληλουχία ανακτάται από την κύρια μνήμη σε έναν τοπικό πίνακα και προωθείται στον αλγόριθμο ευθυγράμμισης. Στις περιπτώσεις υπογραφήματος που έχουν προκύψει από επέκταση μόνο προς τα αριστερά ή μόνο προς τα δεξιά, χρησιμοποιούνται οι πίνακες inNodes ή outNodes αντίστοιχα για να βρεθούν οι διαδρομές μέσα στο υπογράφημα. Η εύρεση των διαδρομών γίνεται με επανάληψη πάνω στους πίνακες γειτόνων, περιορίζοντας το βάθος και τον αριθμό των διαδρομών με βάση προφίλ δεδομένων. Κάθε διαδρομή αποθηκεύεται ως πίνακας από nodeIDs και περνά στο block γραμμικοποίησης, όπου χρησιμοποιούνται οι σωρευτικές θέσεις για την ανάκτηση των αλληλουχιών και τη σωστή διαχείριση των αρχικών, ενδιάμεσων και τελικών κόμβων. Στην περίπτωση επέκτασης και προς τις δύο κατευθύνσεις, απαιτείται επιπλέον πρόσβαση στη μνήμη για την ανάκτηση τμημάτων του πίνακα outNodes που αντιστοιχούν στο υπογράφημα, καθώς δεν υπάρχουν διαθέσιμες πληροφορίες για όλους τους ενδιάμεσους κόμβους. Με τη χρήση αυτών των δεδομένων, οι διαδρομές εξάγονται με τον ίδιο τρόπο όπως στις προηγούμενες περιπτώσεις και η αλληλουχία γραμμικοποιείται σε μία ενιαία αλληλουχία αναφοράς.

Τέλος, το τελευταίο μεγάλο block της ολοκληρωμένης αρχιτεκτονικής, αυτό του sequence-to-sequence aligner, υλοποιεί τον αλγόριθμο Bitap όπως αυτός περιγράφηκε επιγραμματικά παραπάνω. Το συνολικό design επιστρέφει στον host την CIGAR αναπαράσταση της στοίχισης καθώς και τις συντεταγμένες του υπογράφου από τον οποίο εξήχθη η αλληλουχία αναφοράς.

3.3.2 Αρχιτεκτονική του Ολοκληρωμένου Συστήματος με χρήση Sequence-to-Graph Alignment

Μια εναλλακτική αρχιτεκτονική που υλοποιήθηκε για την ίδια λειτουργία με την αρχιτεκτονική που παρουσιάζεται στην προηγούμενη υποενότητα, περιλαμβάνει έναν αλγόριθμο ευθυγράμμισης απευθείας αλληλουχίας προς γράφημα (sequence-to-graph aligner), ο οποίος καταργεί την ανάγκη εξαγωγής διαδρομών πριν την ευθυγράμμιση. Σε αυτή τη σχεδίαση, το block γραμμικοποίησης αφαιρείται πλήρως, το block εξαγωγής seeds παραμένει το ίδιο, ενώ το block ευθυγράμμισης αντικαθίσταται από ένα block sequence-to-graph με ένα επιπλέον βήμα φιλτραρίσματων υποψήφιων υπογράφων. Το block του sequence-to-graph aligner περιλαμβάνει τρία υπο-block: φιλτράρισμα και γραμμικοποίηση υπογραφήματος, edit distance calculation και traceback. Ως είσοδο λαμβάνει τις συντεταγμένες υπογραφήματος που παράγονται από το block Seeder, μαζί με δεδομένα που έχουν ήδη ανακτηθεί από τη μνήμη κατά το στάδιο εξαγωγής seeds. Το φιλτράρισμα βασίζεται σε ήδη φορτωμένους πίνακες γειτόνων (inNodes και outNodes) και τα μήκη των κόμβων, απορρίπτοντας κόμβους που δεν πληρούν τα κριτήρια. Οι κόμβοι που περνούν το φιλτράρισμα αποθηκεύονται σε τοπική scratchpad και στη συνέχεια το υπο-block γραμμικοποίησης ανακτά από τη μνήμη τις αλληλουχίες των κόμβων και τις cumulative θέσεις τους, δημιουργώντας bitvectors που σημειώνουν τις θέσεις έναρξης και λήξης των κόμβων. Οι αλληλουχίες συνενώνονται σε έναν πίνακα χαρακτήρων, ενώ παράλληλα δημιουργείται ένας πίνακας που αποθηκεύει τον κόμβο προέλευσης κάθε χαρακτήρα. Η γραμμικοποιημένη αλληλουχία του υπογραφήματος, τα bitvectors των θέσεων και ο πίνακας προέλευσης περνούν στο υπο-block υπολογισμού απόστασης, όπου εφαρμόζεται ο αλγόριθμος BitAlign. Η αρχιτεκτονική αυτού του υπο-block είναι παρόμοια με αυτήν του sequence-to-sequence aligner, με τη διαφορά ότι περιλαμβάνει επιπλέον υπο-block για τη διαχείριση των τελικών χαρακτήρων των κόμβων που υποδεικνύουν την ύπαρξη hops. Χρησιμοποιώντας τους ήδη ανακτημένους πίνακες outNodes, οι bitvectors υπολογίζονται με bitwise λειτουργίες, αποφεύγοντας την επανυπολογισμό bitvectors για χαρακτήρες κόμβων κάθε φορά που εξετάζεται νέα διαδρομή. Αυτό μειώνει σημαντικά τον απαιτούμενο χρόνο εκτέλεσης σε σύγκριση με την προηγούμενη αρχιτεκτονική, όπου οι bitvectors υπολογίζονταν ξανά για κάθε διαδρομή. Περισσότερες λεπτομέρειες καθώς και σχηματικά των αρχιτεκτονικών μπορούν να βρεθούν στο αντίστοιχο αγγλικό κομμάτι του κειμένου (Κεφάλαιο 6).

4 Πειραματική Αξιολόγηση

Για την αξιολόγηση και μέτρησης της επίδοσης των παραπάνω αρχιτεκτονικών χρησιμοποιήθηκε μία AMD Alveo U200 FPGA κάρτα, ενώ ο γράφος αναφοράς ήταν αυτός του 22ου ανθρώπινου χρωμοσώματος προερχόμενου από το GRCh38 γραμμικό ανθρώπινο γονιδίωμα και 7 VCF αρχεία από το GIABProject. Τα 10 χιλιάδες reads, μήκους 100 βάσεων προήλθαν από dataset της Illumina.

Αρχικά έγινε αξιολόγηση της ακρίβειας της προτεινόμενης μεθόδου seeding χρησιμοποιώντας τα τελικά alignments όπως αυτά δόθηκαν από τον GraphAligner. Αν οι συντεταγμένες του τελικού alignment ενός read ήταν σε εκείνες των υποψήφιων υπογράφων όπως δίνονταν από τον seeder, τότε το seeding θεωρείτο επιτυχημένο. Η ακρίβεια του προτεινόμενου σχεδιασμού έφτασε το 98.73% ενώ αύξηση της παρατηρήθηκε με την αύξηση του ορίου seed hits που εισάγουμε στη σχεδίαση. Συνεπώς παρόλο που θέσαμε ένα όριο στις τοποθεσίες που εντοπίζουμε seed hits, παρατηρούμε ότι η ακρίβεια δεν μειώνεται δραματικά και αυτό γιατί όπως είδαμε κατα το στάδιο του profiling, το 95% των minimizers αναφοράς βρίσκονται το πολύ σε 5 σημεία σε όλο τον γράφο.

Στην συνέχεια αξιολογήσαμε την επίδραση του μεγέθους της read scrathcpad, παρατηρώντας ότι με την αύξηση του ο χρόνος υπολογισμού ανά read στο hardware μειωνόταν, φτάνοντας ακόμα και στο 3.8 της τιμής χρόνου της έκδοσης χωρίς scratchpad. Σε επίπεδο χρησιμοποίησης των διαφόρων στοιχείων, με την αύξηση του μεγέθους της scratchpad αυξήθηκε η χρησιμοποίηση των BRAMs ενώ σε επίπεδο κατανάλωσης ισχύος είχαμε επίσης μία μικρή αύξηση. Συνολικά, ο Seeder επιτυγχάνει την βέλτιστη επίδοση σε χρόνο ανά read για read scratchpad μεγέθους 5, στα 150 MHz, με επιτάχυνση (speedup) στο FPGA ίσο με 4.92×.

Συγκρίνουμε επίσης την επίδοση των δύο διαφορετικών ειδών aligner οου έχουν χρησιμοποιηθεί: sequence-to-sequence και sequence-to-graph. Διαπιστώνουμε ότι ο sequence-to-graph aligner δίνει speedup κοντά σε αυτό του sequence-to-sequence για το συνολικό dataset (7.67× ο S2S και 9.9× ο S2G) και αυτό γιατί στην περίπτωση των read μικρού μήκους τα περισσότερα στοιχίζονται σε υπογράφους του ενός κόμβου, συνεπώς εκτελείται sequence-to-sequence αντιστοίχιση. Αυτό που είναι εντυπωσιακό είναι η ακόμα μεγαλύτερη διαφορά και βελτίωση που δίνει η χρήση του S2G για τα alignments σε υπογράφους πολλών κόμβων για τα οποία επιτυγχάνεται μέχρι και 26.28× alignment speedup. Συνεπώς, η χρήση του S2G aligner φαίνεται να βελτιώνει την επίδοση του aignment σε περιπτώσεις υπογράφων με πολλούς κόμβους και αναμένεται να συμβάλλει ακόμα περισσότερο στο alignment speedup μεγαλύτερων αλληλουχιών. Όσον αφορά σε επίπεδο ποσοστού χρησιμοποίησης των στοιχειών του FPGA, και οι δύο aligners έχουν σχεδόν μηδενικό για όλες τις κατηγορίες εκτός των LUTs για τα οποία ο S2G έχει ελαφρώς υψηλότερο κοντά στο 20%.

Τέλος, όσον αφορά στο ολοκληρωμένο design, παρατηρούμε ότι η φαση του seeding καθορίζει τον συνολικό χρόνο επεξεργασίας ενός read. Διαπιστώνουμε επίσης ότι η αύξηση του μεγέθους της scratchpad βελτιώνει το speedup, φτάνοντας μέχρι και 8.16× για συχνότητα λειτουργίας στα 200 MHz του ολοκληρωμένου συστήματος με S2S Aligner. Επίσης, για read scratchpad μεγέθους 5 φτάνουμε σχεδόν στο 100% της χρήσης των LUTs του ενός SLR που χρησιμοποιούμε. Στην περίπτωση του ολοκληρωμένου συστήματος με χρήση του S2G Aligner, λόγω περιορισμού των resources δεν μπορούμε να εισάγουμε read scratchpad όμως επιτυγχάνουμε ελαφρώς καλύτερο χρόνο σε σχέση με την εκδοχή του design με S2S χωρίς read scratchpad. Οι διαφορές παρόλα αυτά είναι μικρές εφόσον ο συνολικός χρόνος εκτέλεσης και στις δύο περιπτώσεις καθορίζεται από τον Seeder που είναι ο ίδιος. Οι δύο εκδοχές του συνολικού design έχουν πολύ κοντινή κατανάλωση ισχύος (1.648 και 1.715W στα 100 MHz) με εκείνη με το S2G να έχει λίγο περισσότερη.

5 Συμπεράσματα και Προεκτάσεις

Η παρούσα εργασία πρότεινε διαφορετικές ροές εργασίας και αρχιτεκτονικές για την στοίχιση αλληλουχίας σε γράφο (sequence-to-graph alignment) δίνοντας διαφορετικές εκδοχές ενός ολοκληρωμένου συστήματος που πραγματοποιεί χαρτογράφηση και στοίχιση. Παρατηρήσαμε ότι πράγματι η χρήση sequence-to-graph αλγορίθμων βελτιώνει την επίδοση του συνολικού συστήματος ενώ το βήμα του seeding έχει το μεγαλύτερο latency λόγω της εντατικής επικοινωνίας με την κύρια μνήμη. Προκειμένου να αναδειχθεί περαιτέρω η αξία του sequence-to-graph alignment και της προτεινόμενης αρχιτεκτονικής πρέπει αυτή να χρησιμοποιηθεί για την στοίχιση reads μεγάλου μήκους καθώς και το design να ανοίξει σε περισσότερα SLR στο FPGA προς αξιοποίηση όλων των πόρων του. Θα μπορούσε επίσης να εισαχθεί ένα πιο εξειδικευμένο pre-alignment φίλτρο ικανό να διαχειριστεί γράφους, ενώ θα μπορούσαν να επιταχυνθούν και άλλου είδους βιοπληροφορικές εφαρμογές που χρησιμοποιούν γράφους, όπως η de novo δημιουργία ενός γονιδιώματος.

Chapter 1

Introduction

Genome mapping is a foundational process in genomics that enables scientists to locate and organize genes and other important elements within an organism's DNA. It serves as a critical step in genome analysis, providing the structural context needed to interpret sequencing data and uncover meaningful biological insights. Accurate genome mapping is essential for a wide range of applications, including disease variant detection, evolutionary studies, and personalized medicine. As sequencing technologies continue to advance and generate massive volumes of data at unprecedented speeds, the demand for efficient and scalable genome mapping methods becomes increasingly urgent. The genome analysis pipeline includes computationally expensive steps to be able to produce the genome assembly of an organism. The most important step of them all is perhaps read mapping during which each DNA fragment produced by a sequencing machine is matched to its potential location in the reference genome so as to identify its original location. Read mapping is a computationally demanding process demanding high execution time and increased storage and resources. Unfortunately, even though sequencing technologies evolve increasing sequencing throughput, the growth in the rate that they generate reads is far outpacing the corresponding growth computational power placing greater pressure on the alignment and read mapping in general [9]. Traditional read mapping involves sequence-to-sequence (S2S) mapping and alignment of the reads with a linear reference genome. The alignment step is able to compare the read with the candidate mapping reference subsequence and find differences between them, called edits as well as the type of edits need to go from one sequence to the other (deletion, insertion, substitution). This type of alignment has been well studied, and many software tools have been developed to efficiently perform S2S using various algorithms (mainly dynamic programming and approximate string-matching approaches) and additional filtering steps trying to alleviate the read mapping bottleneck, while there are several hardware-software co designed accelerators aiming at accelerating read mapping by creating custom hardware architectures to accommodate these searching and string matching algorithms in a highly efficient manner both in terms of resources and latency. Even though these efforts have managed to accelerate and increase the performance of S2S aligners, there is still an issue of accuracy rooted in the nature of the linear reference sequence. More specifically, a single reference genome originated from a specific model organism (such as linear reference genomes) are not representative of different sets of individuals or strains in a species creating a significant bias to the mapping process due to the inability to incorporate the genetic diversity existing within a population. This bias influences greatly alignment and mapping accuracy. For instance, the African genome, with all genetic variations found in populations of African descent, contains 10% more DNA bases that the current linear reference sequence, therefore many reads coming from an African individual's sample may not be aligned at all. If the linear reference genome used was of African origin though these sequences would be aligned, and the accuracy would increase for this specific individual but decrease similarly for the case of an individual of Asian origin. The same issue arises in genomic studies when trying to identify carcinogenic mutations or to track mutations of viruses or other organisms or viruses that undergo mutagenesis with a high rate. Therefore, finding and adopting a new reference genome structure that incorporates all this genetic variation is essential to increase read mapping accuracy.

An increasingly popular reference structure used for read mapping is that of genome graphs. Genome graphs enable a compact representation of the linear reference genome combined with the known genetic variations in the entire population as a graph-based structure. Of course, the use of a graph in the whole pipeline increases complexity, latency and resources' need, however, there have been some software tools developed to efficiently perform sequence to graph read mapping and alignment proposing ways of organizing and indexing the big graph structures while decreasing heuristic algorithms' complexity. Most of these limited tools, with the most used one being GraphAligner [10], utilize dynamic programming algorithms to perform sequence to graph alignment and calculate the edits between a reference subgraph found during the seeding step and the read. Even though GraphAligner has very good performance in terms of accuracy and satisfactory efficiency, the alignment step of the sequence to graph read mapping pipeline still is a major bottleneck requiring 50-95% of the end-to-end execution time, urging thus for acceleration. Another major issue when it comes to sequence to graph alignment is the high cache miss rates, with GraphAligner reaching up to 41% of cache misses due to the high amount of intermediate data that is generated and reused as part of the alignment step while another sequence-to-graph mapper and aligner vg, to tackle this issue, divides the read into overlapping chunks to reduce the size of the dynamic programming table, thus the size of the intermediate data. Additionally, during the seeding step of the pipeline there is a DRAM latency bottleneck, since seeding requires a big number of random main memory accesses while querying the index for the seed locations. Finally, these software tools scale sublinearly, meaning that due to the amount of intermediate data needed to be accessed in the caches during the alignment step when the number of threads reaches the number of physical cores in a CPU system, two threads sharing the same core experience severe cache and main memory interference with each other. Thus, full thread-level parallelization is not possible [8]. Due to the above issues arising in the existing software sequence-to-graph mapping and alignment tools, there is a need to develop specialized, balanced and scalable design for compute units, on-chip memory and main memory accesses for all the steps of the pipeline, mainly seeding and alignment step which require acceleration the most. Thus, the need to introduce hardware/software co-designed tools to enable high performance, efficient, scalable and low-cost sequence-to-graph mapping is great. The first universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping SeGraM provides significant improvements for multiple steps of the genomic assembly pipeline for both a graph and a linear reference genome providing acceleration in both the seeding and alignment steps outperforming both GraphAligner and vg, while reducing power consumption. However, as explained in detail in Chapter 2 the SeGraM accelerator introduces a limitation concerning the size of the candidate alignment subgraph, through the imposition of a hop limit within the proposed algorithm and hardware architecture, decreasing alignment accuracy and not being able to accommodate all possible alignment positions and subgraphs in the graph.

This thesis aims at exploring different algorithms and hardware architectures to accelerate sequence-to-graph genome mapping and alignment. Specifically, we design, implement and evaluate two different approaches to this problem, the one involving the use of a sequence-to-sequence aligner and the second one a sequence-to-graph. Apart from comparing the performance and characteristics of the proposed designs, we also build our seeding workflow and suggest a hardware friendly algorithm to manipulate candidate alignment subgraphs during the seeding step.

Chapter 2

Related Work

This chapter presents a brief overview of the existing software tools and hardware accelerators implementing sequence-to-sequence and sequence-to-graph aligners explaining some of them in more detail together with some of their limitations.

1 Existing Sequence-to-Sequence Alignment Tools

Sequence-to-Sequence genome alignment has been better examined with more software and software/hardware co-designed tools having been designed to accelerate it. This section includes some of the existing CPU, GPU, FPGA and ASIC accelerators of sequence-to-sequence alignment together with a short overview of the algorithms they aim to accelerate.

1.1 CPU Aligners

Minimap2 and Bowtie2 are two of the CPU implemented algorithms which are broadly used for the task of sequence-to-sequence alignment, including both the mapping and alignment steps.

1.1.1 Minimap2

Minimap2 [3] is a general-purpose alignment program for mapping DNA or long mRNA sequences against large reference databases. Published in Bioinformatics in 2018, this tool addresses the critical need for efficient alignment of increasingly long sequencing reads produced by modern technologies like PacBio SMRT and Oxford Nanopore. The software demonstrates remarkable versatility by handling diverse sequence types ranging from short reads of 100 bp with high accuracy to long genomic reads of 1 kb at 15% error rate, full-length noisy Direct RNA or cDNA reads, and even assembly contigs or chromosomes of hundreds of megabases in length.

The tool's architecture follows a seed-chain-align procedure that begins with indexing by collecting minimizers from reference sequences and storing them in a hash table. During the seeding phase, query minimizers serve as seeds to find exact matches called anchors, which are then organized into colinear chains. Finally, dynamic programming extends chains and closes gaps between anchors when base-level alignment is requested. The chaining component represents a significant technical advancement, using dynamic programming with the formula $f(i) = \max\{\max_{i>j>1}[f(j) + a(j,i) - \beta(j,i), w_i]\}$ where a(j,i) represents matching bases between anchors, b(j,i) is the gap cost, and w_i is the anchor weight. A key optimization reduces time complexity from O(N^2) to O(hN) through a heuristic that stops evaluation after h iterations without improvement.

For base-level alignment, minimap2 implements the breakthrough Suzuki-Kasahara difference-based formulation, which enables 16-way SSE vectorization regardless of peak alignment scores. This innovation allows efficient processing of long sequences that would exceed traditional score thresholds, representing a major advancement over previous methods limited to 4-way parallelization. The algorithm employs a 2-piece affine gap cost scheme, which effectively recovers longer insertions and deletions compared to standard affine gap penalties.

Minimap2 excels in specialized applications through adaptive modifications of its core algorithm. For spliced sequence alignment in RNA-seq data, the tool modifies gap costs to distinguish insertions from deletions, treating deletions longer than a threshold as introns with no extension cost while implementing reference-dependent costs that penalize non-canonical splicing signals. The software models GT-AG and CT-AC splicing signals with different penalties and can align each chain twice to infer strand orientation for unstranded libraries. For short read alignment, minimap2 adapts its chaining algorithm for paired-end reads by treating read pairs as single fragments with unknown gap lengths, applying different gap costs for seeds on the same versus different reads.

The performance achievements of minimap2 are exceptional across multiple benchmarks. The tool operates 3-4 times faster than mainstream short-read mappers at comparable accuracy and demonstrates 30-fold speed improvement over existing long-read mappers while maintaining higher accuracy. Memory usage remains competitive at 6.8GB peak for human genome alignment, and the software provides superior mapping quality estimation with reduced spurious alignments. On simulated human PacBio reads, minimap2 achieved higher mapping accuracy than established tools like BLASR, BWA-MEM, and GraphMap while requiring only 200 CPU seconds compared to over 6000 seconds for competing tools.

Technical innovations distinguish minimap2 from other alignment tools. The software adopts homopolymer compression for indexing, which contracts consecutive identical bases to single bases, improving overlap sensitivity for SMRT reads from 90.9% to 97.4% compared to standard minimizers. The Z-drop heuristic prevents forced alignment of unrelated sequences by breaking alignments when scores drop too rapidly, similar to X-drop in BLAST but more robust to single long gaps. Additionally, minimap2 filters out misplaced anchors that would lead to suboptimal alignments and employs sophisticated primary chain identification to handle repetitive regions effectively.

1.1.2 Bowtie2

Traditional read aligners using full-text minute indices (like the original Bowtie) excelled at fast, memory-efficient ungapped alignment of short reads. However, they struggled with gapped alignments, which are essential for detecting insertions, deletions, and handling sequencing errors. The challenge arises because gaps substantially increase the size of the search space and reduce the effectiveness of pruning strategies such as double indexing and bidirectional Burrows-Wheeler transform that make ungapped alignment so efficient. This limitation means that purely index-based approaches become computationally inefficient for gapped alignment, often failing to align reads that span gaps and missing important biological evidence for insertion and deletion events.

Bowtie 2's key innovation is a hybrid two-stage approach that strategically combines the strengths of full-text minute indices with the flexibility of hardware-accelerated dynamic programming algorithms [11]. The first stage maintains the speed and memory efficiency advantages of full-text minute index-assisted seed finding, while the second stage enables sensitive gapped alignment through SIMD-accelerated dynamic programming. This combination achieves an effective balance of speed, sensitivity, and accuracy across a range of read lengths and sequencing technologies.

The Bowtie 2 algorithm proceeds through four distinct steps for each read. In the first step, the software extracts seed substrings from both the read and its reverse complement. The second step involves aligning these extracted substrings to the reference genome in an ungapped fashion, assisted by the full-text minute index. During the third step, seed alignments are prioritized and their positions in the reference genome are calculated from the index. Finally, in the fourth step, seeds are extended into full alignments by performing SIMD-accelerated dynamic programming.

The algorithm incorporates several key technical features that enhance its performance. SIMD processing leverages single-instruction multiple-data parallel processing capabilities available on modern processors, providing substantial computational acceleration. The system uses bidirectional Burrows-Wheeler transform for efficient pruning during the seed-finding phase. Bowtie 2 supports flexible alignment modes, offering both end-to-end alignment (where the entire read must be aligned) and local alignment (where portions of the read extremes can be soft-clipped). Additionally, the algorithm is quality-aware, incorporating mapping quality scores that estimate the probability of incorrect alignment.

The performance results demonstrate significant improvements across multiple metrics. In terms of speed, Bowtie 2's default mode was more than 2.5 times faster than BWA's default for unpaired reads and more than 3 times faster for paired-end reads. The software consistently outperformed BWA-SW on longer reads from 454 and Ion Torrent platforms. Memory efficiency was well-balanced, with peak memory usage of 3.24-3.39 gigabytes, positioned between BWA's 2.39 gigabytes and SOAP2's 5.34 gigabytes.

Regarding sensitivity and accuracy, Bowtie 2 aligned more reads than BWA, SOAP2, and the original Bowtie across all test scenarios. When evaluated on simulated data where correct alignments were definitively known, Bowtie 2 demonstrated superior accuracy. The software showed particularly better performance on longer reads (250-400 nucleotides) compared to BWA-SW, with fewer reads requiring trimming during the alignment process.

Bowtie 2 offers several distinct technical advantages over its predecessors and competitors. Unlike the original Bowtie, it provides robust gapped alignment capability, effectively handling insertions and deletions that are crucial for comprehensive genomic analysis. The hardware acceleration through SIMD optimization provides substantial speed improvements without sacrificing accuracy. The algorithm maintains efficiency across different read lengths and sequencing technologies, making it versatile for various experimental designs. Additionally, Bowtie 2 demonstrates improved robustness in handling sequencing errors, which is particularly important for emerging single-molecule technologies where error profiles differ significantly from traditional platforms.

1.2 GPU Aligners

Less aligner designs have been proposed for GPU acceleration, two of which, GASAL2 and AGAThA are presented in this subsection. The first one is designed to align high throughput data while the second one long reads.

1.2.1 GASAL2

GASAL2 is a high-performance, GPU-accelerated library specifically designed for pairwise sequence alignment of DNA and RNA, primarily for high-throughput Next-Generation Sequencing (NGS) data [12]. It aims to overcome the computational intensity of sequence alignment algorithms by leveraging the massive parallelism of GPUs, addressing limitations found in traditional CPUs and even other GPU-based libraries like NVBIO. Sequence alignment, a fundamental process in genomics, involves editing two sequences with gaps and substitutions to identify similarities. This is typically done using dynamic programming, with the Needleman-Wunsch algorithm for global alignment and the Smith-Waterman algorithm for local alignment, both of which have been improved with affine-gap penalties. Global alignment seeks to align sequences in their entirety, maximizing the score, while local alignment finds the most similar sub-regions. Semi-global alignment identifies overlaps, allowing unpenalized gaps at the ends of sequences. GASAL2 supports all types of global, local, and semi-global alignments, including one-to-one, all-to-all, and one-to-many pairwise alignments.

A core innovation in GASAL2 is its GPU-based sequence packing. Unlike NVBIO, which packs sequences on the CPU and loses significant time (around 80% of total execution time for some datasets), GASAL2 performs this process entirely on the GPU. Each DNA/RNA base (A, C, G, T/U, N) is represented by 4 bits, and 8 bases are packed into a 32-bit unsigned integer by each GPU thread. This GPU packing is dramatically faster, achieving at least 580x speedup over NVBIO and effectively eliminating data packing time.

After packing, sequences reside in GPU memory, and subsequent operations, including optional reverse-complementing, are performed on the GPU. The alignment kernel itself utilizes inter-sequence parallelization, assigning each GPU thread a pair of sequences for independent alignment. For affine-gap penalties, the algorithm typically computes three dynamic programming matrices (H, E, F). To optimize memory access and reduce memory requirements from $O(n^2)$ to $O(n)$, GASAL2 processes these matrices in 8x8 tiles, fetching 32-bit words (8 bases) from memory to compute 64 cells at once. It stores only an 8-element column and a full row of intermediate values, with the small column fitting into GPU's register file.

For the traceback step, which generates the actual alignment in CIGAR (Compact Idiosyncratic Gapped Alignment Report, detailing matches, mismatches, insertions, and deletions), GASAL2 stores a 4-bit direction matrix in global memory. It can also compute the alignment's start position without full traceback by re-aligning backward from the end position until the score matches the previously found maximum.

GASAL2 employs CUDA streams to enable asynchronous execution, meaning CPU tasks can overlap with GPU kernel launches and data transfers, significantly reducing idle CPU cycles and maximizing utilization. The library uses C++ templates to generate specialized kernels at compile time, avoiding costly runtime branching that could slow down GPU execution. Furthermore, GASAL2 offers flexible handling of ambiguous 'N' bases, allowing configurable match/mismatch scores, a critical feature often missing or poorly implemented in other libraries. It also improves memory management by allocating and de-allocating GPU memory only once at program start and end, unlike older approaches that did so repeatedly, thus reducing performance overhead.

Evaluated against a dual-socket hyper-threaded Intel Xeon system (28 cores), GASAL2 consistently outperforms existing solutions. For local alignment calculating only the score and end-position, it is up to 5.35x faster than 56 Intel Xeon threads and up to 10x faster than NVBIO for 100bp reads. When computing the start-position without traceback, GASAL2 achieves up to 6x speedup over CPU implementations and up to 13x over NVBIO for 100bp reads. For full traceback computation, GASAL2's GPU kernel is 4x faster than NVBIO's, leading to overall speedups of 9x, 7x, and 5x for 100, 150, and 300bp reads respectively. It is also 13x and 20x faster than SeqAn and Parasail for traceback alignments up to 300 bases.

In a case study, GASAL2 accelerated the seed-extension stage of BWA-MEM (a DNA read mapper) by over 20x compared to the CPU version with 12 threads. This led to an overall BWA-MEM application speedup of 1.3x.

1.2.2 AGAThA

AGATHA is a novel GPU-accelerated solution specifically designed to address the computational burden of aligning increasingly long DNA sequencing reads, a crucial step in bioinformatics pipelines.Unlike previous GPU acceleration attempts that often compromise the algorithm's exact structure due to GPU-unfriendly computational patterns, AGATHA provides an accurate and efficient GPU-based acceleration of guided sequence alignment. It specifically tackles the challenges of strided/redundant memory accesses and unpredictable workload imbalances inherent in these algorithms [13].

The core of sequence alignment for long reads, as used by de facto standards like Minimap2, is the guided dynamic programming approach. This involves filling a 2D score table based on a reference (R) and query (Q) string, with a computational and space complexity of $O(\hat{N}^2)$. Each cell $H(i, j)$ represents the best alignment score up to the i-th reference character and j-th query character, derived recursively from top, left, and top-left cells.

A key aspect of guided alignment is filtering false positives through heuristics like banding and termination conditions. Banding (e.g., k-banding) limits calculations to a diagonal band, assuming large insertions or deletions indicate false positives.The termination condition (often called Z-drop) stops computation if the difference between a global maximum score and the current score exceeds a threshold, indicating too many mismatches. While these guiding strategies improve CPU throughput, they introduce significant performance overheads on GPUs due to random memory accesses and dynamic workload imbalance, leading to the absence of exact GPU implementations. AGATHA addresses GPU challenges in guided sequence alignment through four key contributions: a rolling window approach for local maximum tracking that uses shared memory and periodic spills to global memory, reducing redundant memory accesses and utilizing coalescing ; a sliced diagonal tiling strategy that partitions alignment bands into anti-diagonal slices, allowing subwarp threads to process chunks and keep intermediate values in registers, significantly reducing run-ahead execution and shared memory requirements for local maximum tracking; subwarp rejoining, a fine-grained work-stealing mechanism to mitigate intra-warp workload imbalance and warp divergence by having idling subwarps rejoin active ones to form larger subwarps, ensuring higher GPU thread utilization; and uneven bucketing, which addresses inter-warp workload imbalance by sorting and redistributing sequences based on length, ensuring a more balanced workload per warp and using subwarp rejoining for dynamic adaptation if termination conditions occur. AGATHA builds upon existing GPU methods like input packing (encoding DNA literals into 4-bit words) and intra-query parallelism (assigning threads to alignment tasks), while specifically tackling warp divergence from guiding heuristics. Although subwarps reduce external fragmentation, they can increase internal fragmentation. Performance evaluations with human genome datasets show AGATHA achieves an 18.8x geometric mean speedup over CPU-based Minimap2.

1.3 FPGA Aligners

There are many FPGA accelerators to perform both the pre-alignment filtering and alignment steps, utilizing different approaches and algorithms. Here, two of them GANDALF and ASAP are briefly presented.

1.3.1 GANDAFL

GANDAFL is a FPGA-based hardware accelerator designed to accelerate short-read alignment, specifically focusing on optimizing the Smith-Waterman (SmW) algorithm's matrix-fill and traceback stages.Short-read alignment is the initial, computationally demanding step in genomic workflows, mapping short DNA fragments to a reference genome. Most state-of-the-art aligners„ use a "seed-and-extend" model: short "seeds" from the read are perfectly matched to the reference, and then extended into full alignments using dynamic programming algorithms like SmW. The SmW algorithm consists of two phases: matrix-fill (computing a similarity matrix) and traceback (reconstructing the optimal alignment path) [14]. GANDAFL implements a linear systolic array scheme for both the matrix-fill and traceback stages, which are typically major bottlenecks.The architecture features an array of Processing Elements (PEs), one for each base of the read sequence.These PEs compute the similarity matrices (H, E, F) in parallel, exploiting the anti-diagonal dependencies of the SmW algorithm.

Key optimizations in GANDAFL include: on-chip traceback, which crucially moves both the matrix-fill and traceback computations onto the FPGA, eliminating significant data transfer overhead of sending large matrices back to the CPU for traceback; an interleaving data scheme, which minimizes stalls due to intra-data dependencies by interleaving the computation of multiple read-reference pairs, allowing L anti-diagonals to be computed per L cycles, significantly improving throughput; and double buffering, which is employed for the E, F, and H matrices to avoid halting the matrix-fill operation while the traceback module reads data. To minimize accelerator invocation overheads, which can lead to days of execution time for frequent calls, the Bowtie2 aligner's software is radically restructured into three phases: a data-gathering phase, a hardware execution phase, and a data-distribution phase. This batching strategy significantly reduces the number of accelerator calls. GANDAFL as a standalone accelerator, GANDAFL delivers up to 116x speedup over state-of-the-art software (Bowtie2's SSE-vectorized SmW implementation) and 2.13x speedup over other state-of-the-art FPGA accelerators like Darwin's GACT.

1.3.2 ASAP

ASAP is a hardware accelerator designed to significantly improve the runtime of short-read alignment, specifically by accelerating the Levenshtein distance (LD) computation, which often accounts for 50-70% of the runtime in tools like SNAP [15]. ASAP's core innovation is its RaceLogic approach, which utilizes the intrinsic propagation delay of circuits as a computational proxy instead of explicitly calculating LD values. It maps mathematical operations vital for LD computation, such as addition and minimization, directly onto circuit topologies: addition is represented by serially connected circuit elements where delays sum up, and minimization is achieved with an OR gate where the earliest signal dictates the output. The accelerator is structured as a lattice of Delay Elements (DEs), each corresponding to a cell in a dynamic programming matrix. Each DE contains three delay blocks (D_M for match/mismatch, D_I for insertion, D_D for deletion) whose delays are based on user-defined "delta-parameters" or gap penalties. A key approximation in ASAP involves setting the match penalty to zero cycles, allowing large portions of the search space corresponding to matches to be explored in almost zero time. This approximation crucially preserves the total ordering of LDs, maintaining alignment accuracy even without computing the exact LD value. Energy is conserved by clock-gating unused DEs, ensuring power consumption only for elements contributing to the final

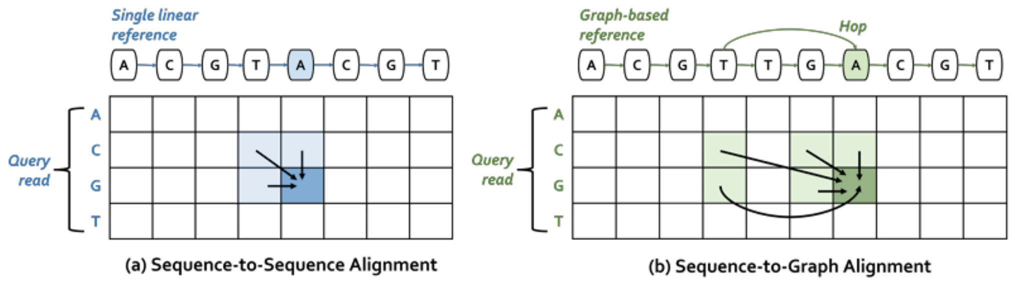


Figure 2.1: The cases of Linear and Graph References

result. Evaluated on a Xilinx FPGA using simulated human genome reads, ASAP demonstrates significant performance gains: LD Computation Speedup: ASAP is approximately 200x faster than an equivalent C implementation of the Smith-Waterman algorithm running on a single CPU core. It is also about 5x faster than other FPGA-based solutions.

2 Sequence-to-Graph Alignment Tools

Sequence-to-Graph alignment compared to traditional Sequence-to-Sequence tools has many challenges including alignment algorithm complexity, constraints due to the topology of different reference graphs and lack of scalability. In contrast to sequence-to-sequence alignment for which there have been developed many aligners of high accuracy both in a software and a software/hardware co-design level, sequence-to-graph is a newly introduced approach that has not been yet explored as much, having a small number of software sequence-to-graph aligners, including the most broadly used GraphAligner and vg, and only two software/hardware co-designed aligners developed: SeGraM and Harp. This work is based on the SeGraM seeding and alignment approaching while comparing alignment accuracy with GraphAligner, therefore these two tools will be presented in more detail. Most of the alignment algorithms have a DP-based approach which mainly includes operation on a table where each column corresponds to a reference character and each row to a query read character. Each cell of the matrix is interpreted as the cost of a partial alignment between the reference and query read subsequences that have been traversed so far. The difference with sequence-to-sequence alignment and the challenge emerging in the sequence-to-graph case is that a new cell in the table is determined by not only its 3 neighboring cells but it must also incorporate non-neighboring cells where there is an edge (i.e. hop) from the non-neighboring character to the current one. Schematically this difference is demonstrated in Figure 2.1:

Briefly, the vg toolbox is able to construct and manipulate genome graph apart from mapping sequences to them and is used to build and transform graphs in the desired form for each other tool. The Harp accelerator [16] is mostly used for pan-genomic analysis and takes advantage of the structural similarities between genome graphs and sequences, effectively reducing graph processing overhead by leveraging their quasi-sequential nature. The HARP accelerator is co-designed with two key algorithmic components: (1) HarpTree, a compact data structure that explicitly captures the quasi-sequential property, simplifying graph processing algorithms in sequence-to-graph (S2G) mapping, and (2) HarpExt, a multi-stage seed-extension algorithm that minimizes graph operation overhead while preserving the accuracy of S2G mapping.

2.1 The vg toolkit

The vg toolkit represents a comprehensive software framework for variation graph genomics, implementing a fundamental shift from linear reference genomes to graph-based structures that capture population-level genetic diversity [6]. The core data model defines a variation graph as G = (N, E, P), where nodes represent DNA sequences, edges define valid sequence transitions, and paths embed known genome sequences within the graph structure. This bidirectional graph architecture supports complex genomic features including inversions, duplications, and structural rearrangements through forward and reverse traversal capabilities with automatic sequence complementation. The read mapping algorithm employs a sophisticated seed-extend-align strategy centered on the GCSA2 (Generalized Compressed Suffix Array) indexing system. This approach identifies Super-Maximal Exact Matches (SMEMs) in linear time independent of graph complexity, followed by clustering using global distance metrics and chaining through Markov models that optimize for match length while minimizing gap penalties. Alignment proceeds through partial order alignment using enhanced Smith-Waterman algorithms with SIMD acceleration and banded dynamic programming. The implementation leverages multiple advanced data structures for scalability and efficiency. The core representation uses Protocol Buffers for serialization, while the succinct xg format provides memory-efficient storage and random access capabilities for gigabase-scale operations. Graph construction supports multiple methodologies including VCF-based variant incorporation, progressive sequence alignment and integration, and direct import from standard formats like GFA. Dynamic editing operations enable real-time graph modification through node splitting and edge insertion while maintaining coordinate translation for version control. Performance characteristics demonstrate the system's practical utility at genomic scales, handling human genome variation graphs containing 80 million variants across 3.2 gigabases with total storage requirements of 25-63 GB including indices. The multithreaded implementation provides linear scaling for sequence lengths extending to multiple megabases. Accuracy improvements are particularly notable for reads containing non-reference alleles, where vg eliminates mapping bias and provides balanced coverage independent of variant size or complexity. The toolkit integrates seamlessly with existing genomic workflows through GAM (Graph Alignment Map) format support and BAM export capabilities. Quality score integration employs probabilistic alignment models for enhanced accuracy, while mapping quality computation compares optimal versus suboptimal alignments. Advanced applications extend beyond variant calling to functional genomics, including ChIP-seq analysis with reduced reference bias and splice-aware RNA-seq mapping through graph-encoded transcript relationships.

2.2 GraphAligner

GraphAligner was introduced as a software tool to contribute to the efficient fast alignment of longs reads to graphs, which vg was unable to do as it is finetuned in small reads. This tool uses banded sequence-to-graph alignments able to align noisy longs reads to bidirectional de Bruijn graphs of the whole genome and is able to perform error correction of the long reads. A de Bruijn graph is a directed graph used to represent overlapping sequences of symbols, particularly useful in genome assembly and sequence analysis. Its structure consists of nodes representing unique (k-1)-mers and edges connecting nodes where their corresponding sequences overlap by k-2 bases. Each edge represents a full k-mer, effectively encoding the continuity of a sequence [10]. After having transformed the bidirectional graph into the alignment graph, which is a directed node-labeled graph, a minimizer index is built. The minimizers are extracted from the sequences of the nodes and then are stored together with the nodeID and position they are found within the node. Then the k-mers are organized in buckets based on the k-mer's modulo and the buckets are indexed so as to be able to query a k-mer based on its modulo. After a k-mer is being searched in the buckets of the indexed reference chaining is performed though superbubbles, which are induced acyclic subgraphs with one unique entrance node, one unique exiting node and some internal nodes. Two superbubbles belong to the same chain if the end node of a superbubble is the start node of another. After having found the superbubbles all seed hits in the chain can be assigned a linear position and traditional sequence-to-sequence chaining approaches can be applied (such as the one from minimap). GraphAligner then uses a DP algorithm to extend the seeds. This algorithm calculates a DP matrix whose scores describe the edit distance of an alignment ending at a specific position in the read and a specific position in the graph, while using bit-parallel operations to decrease the alignment algorithm's computational complexity. The main idea utilized by GraphAligner is banded alignment which has also been used in a previously developed format in sequence-to-sequence aligners. This approach

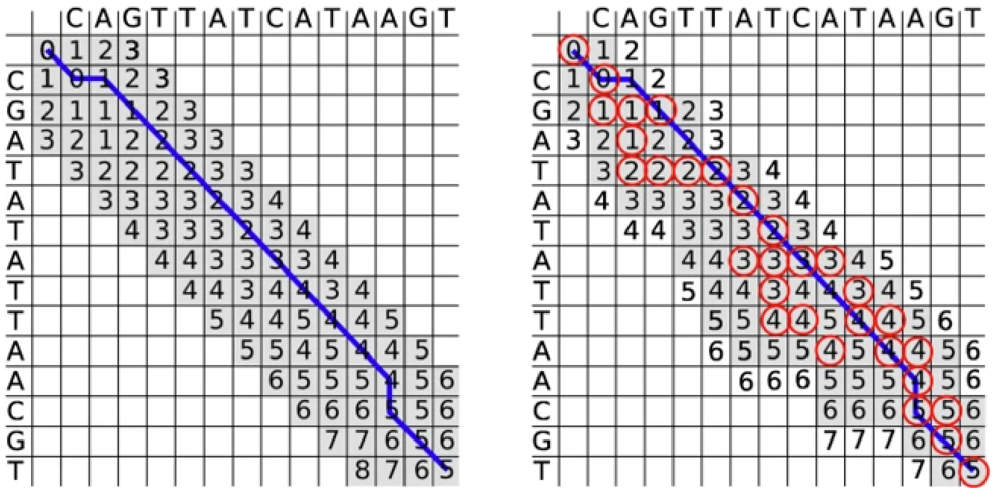


Figure 2.2: GraphAligner Alignment Approach

finds the minimum score m and defines a cell to be inside the band if its score is at most m+b, where b is a banding parameter, enabling the handling of arbitrary topologies such as that of a graph. For example, in Figure 2.2 the left case is a standard banded alignment with a band width of b=3. The reference sequence is displayed at the top, and the query sequence on the left. The gray cells fall within the band and are included in the calculations. The blue line indicates the traceback of the optimal alignment. On the right example there is score-based banding with a band width of b=1. The reference is on top, and the query is on the left. The gray cells are inside the band, and the blue line represents the traceback. The red-circled cells highlight the minimum values for each row, which are found during the matrix calculation and determine whether a cell is inside the band. A cell is inside the band if its score is within b of the minimum score in the same row. Cells with a number on a white background are used to identify the band's boundary but are not part of the band and are excluded from the next row's calculation.

To get a more holistic idea of how the abound banded alignment looks in a subgraph of Figure 2.3 shows the DP matrix for aligning a read to the graph. Each node has a corresponding column in the matrix shown with arrows. The gray area represents parts of the matrix which are calculated in contrast to the non calculated white parts. This score-based banding implicitly limits the exploration of the alternative paths whose scores become worse, promoting the choice of the optimal path which ends up being explored completely. The traceback process involves following the optimal alignment path or sequence to determine the sequence of operations (substitutions, insertions or deletions) that produce the best alignment between the read query and the reference sequence.

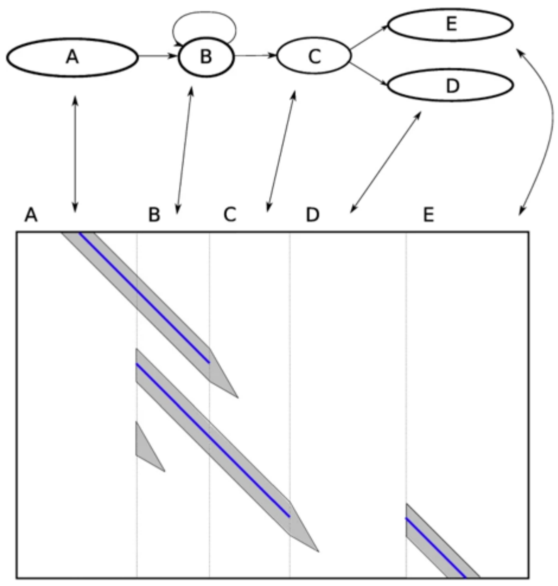


Figure 2.3: DP matrix for aligning a read to the Graph

2.3 SeGraM

In contrast to this dynamic programming-based approach, which is mostly used in software-based approaches, SeGraM, which is the first of the only two sequence-to-graph hardware accelerators, uses bitwise operations for the alignment and traceback process [8]. The overview of the SeGraM accelerator architecture is seen in Figure 2.4. The accelerator is made up of two distinct blocks, the MinSeed accelerator which is designated to finding candidate alignment subgraphs for an input read, and the BitAlign Accelerator which performs the actual alignment, finding both the edit distance between the read and the subgraph and performing traceback to find the optimal alignment and output a CIGAR coded string. The MinSeed algorithm is an altered version of the minimap2 algorithm used for seeding in sequence-to-sequence accelerators. It extracts the lexicographically smaller k-mer (minimizer) from a window and searches it in the indexed reference found in the main memory. If the minimizer is found in many locations of the reference (has high occurrence frequency) it is eliminated. For this design 0.02% of the most frequent minimizers are discarded, while the locations of the ones remaining are fetched to the seed scratchpad. Then the candidate subgraph by finding the leftmost and rightmost positions of the seed within the graph to extract the candidate alignment subgraphs.

The linearized sequences of the candidate subgraphs are then passed to the BitAlign block which uses an altered version of the Bitap algorithm used in GenASM, to incorporate non-neighboring nodes in the calculation of the intermediate bitvectors later used for traceback. An overview of the BitAlign algorithm is shown in Algorithm 6. This altered version of the Bitap algorithm which utilizes bitwise operations to find the number and type of edits between

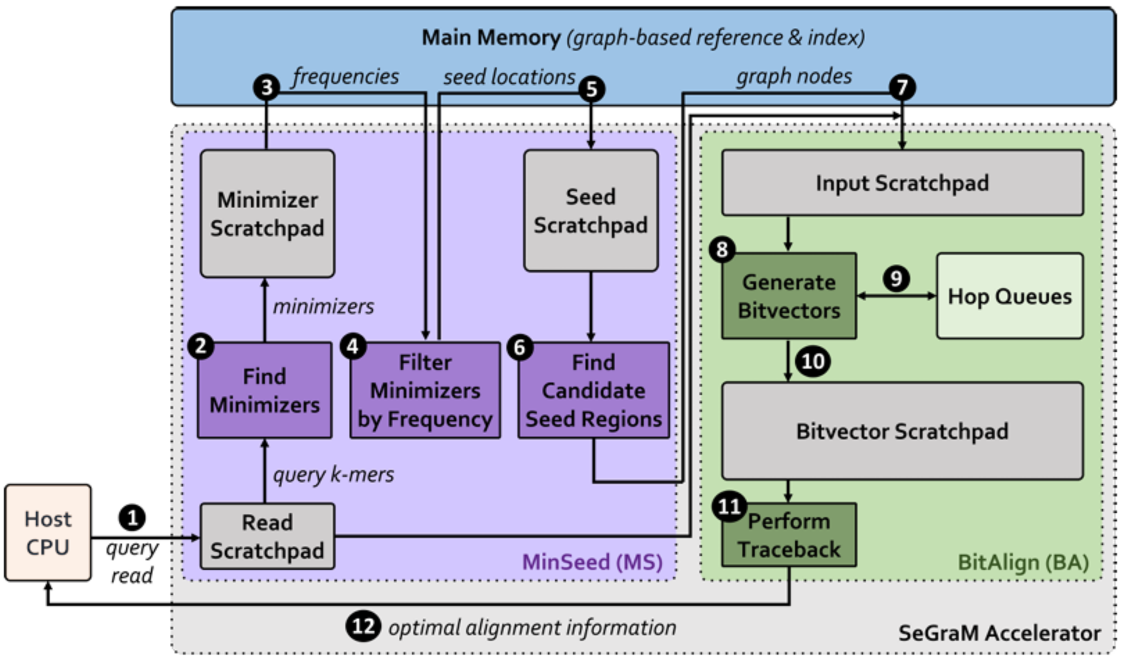
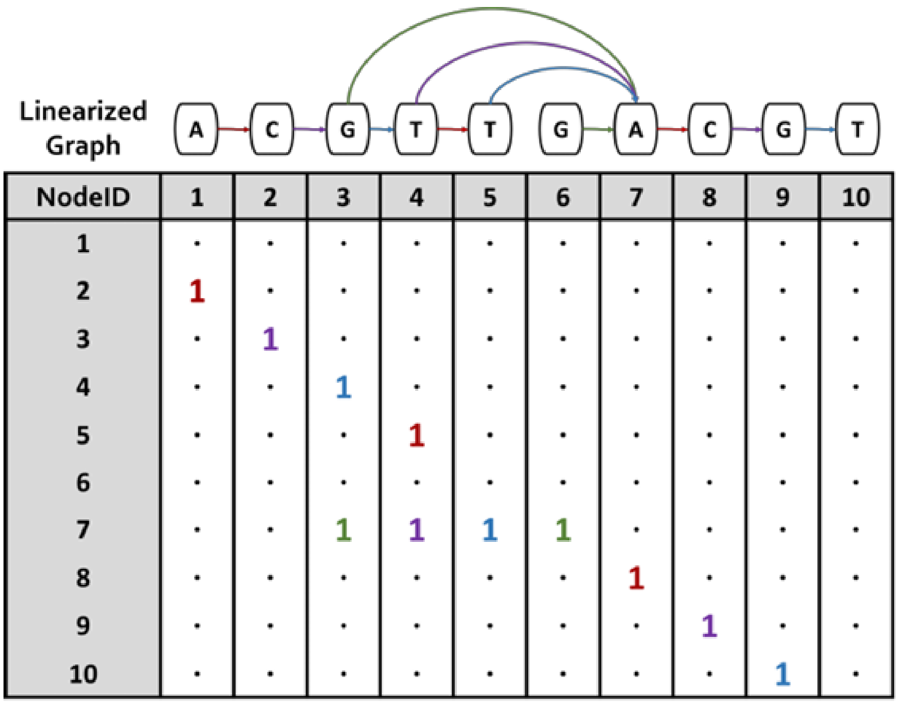


Figure 2.4: High Level Architecture of the SeGraM Accelerator [8]

the read sequence and the candidate reference subgraph that has been found during the seeding step, needs to incorporate non neighboring characters as well when the reference is found in a graph structure.

To store these bitvectors corresponding to the non-neighbor nodes, the design introduces HopQueue Registers to pass these bitvectors from one processing element to the other. Furthermore, the hop information between nodes of the graph is stored in an adjacency matrix called HopBits that has the format shown in Figure 2.5: To decrease and bound the sizes of the HopQueue Registers and the HopBits matrix the design introduces the concept of hop limit which is basically the distance between the farthest node to consider and the current node. After having explored the characteristics (number of hops and their values) of 24 genome graphs one for each human chromosome, SeGraM chooses a hop limit equal to 12 to cover more than 99% of all hops in the graph-based reference. The results of this profiling are shown in Figure 2.6 where the x-axis has the hop limits and the y-axis the fraction of total hops covered when this respective hop limit is set. This design choice introduces a tradeoff between power/area overhead and accuracy which leaves room for improvement. To alleviate this accuracy issue and remove the hop limit it is essential to first understand the reference graph structures and their characteristics in terms of number and size. However, it is additionally important to see whether such great hops end up giving valid alignment with reads or there might be some hops that are rarely incorporated within alignments, meaning that they represent a very rare mutation. In that case it would be a fair tradeoff of latency, power, area and accuracy since the elimination of hops rarely found in align-



NodeID	1	2	3	4	5	6	7	8	9	10
1	·	·	·	·	·	·	·	·	·	·
2	1	·	·	·	·	·	·	·	·	·
3	·	1	·	·	·	·	·	·	·	·
4	·	·	1	·	·	·	·	·	·	·
5	·	·	·	1	·	·	·	·	·	·
6	·	·	·	·	·	·	·	·	·	·
7	·	·	1	1	1	1	·	·	·	·
8	·	·	·	·	·	·	1	·	·	·
9	·	·	·	·	·	·	·	1	·	·
10	·	·	·	·	·	·	·	·	1	·

Figure 2.5: Neighbor Information as Represented within SeGraM [8]

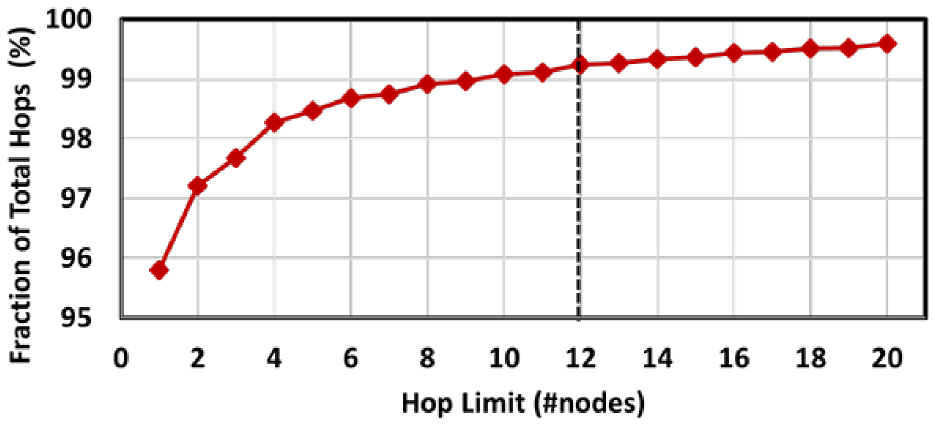


Figure 2.6: Effect of the hop limit on the fraction of hops included when performing sequence-to-graph alignment [8]

ment subgraphs for a high percentage of reads would not affect the accuracy for most of the time. Overall, SeGraM manages to increase throughput compared to GraphAligner while decreasing power and area, there is a trade-off of accuracy since this accelerator is designed to have a hop limit meaning that a very small (close to 1%) but yet existing percentage of hops are not taken into account when aligning sequences.

Chapter 3

The Genome Analysis Pipeline

This chapter presents the theoretical and technical foundations required to understand the contributions of this thesis. Starting with an overview of the concept of genomics and the different stages of the genome analysis pipeline, which are explained in detail, this part of the chapter aims at introducing the reader to the different steps and processes used to sequence the entire DNA of an organism.

This section presents the concept of Genomics and describes in detail the Genome Analysis Pipeline used to sequence the entire DNA (or RNA in some cases) of an organism starting from a biological sample and ending up with a FASTA file with the genome sequence.

1 The Genome and Genomics

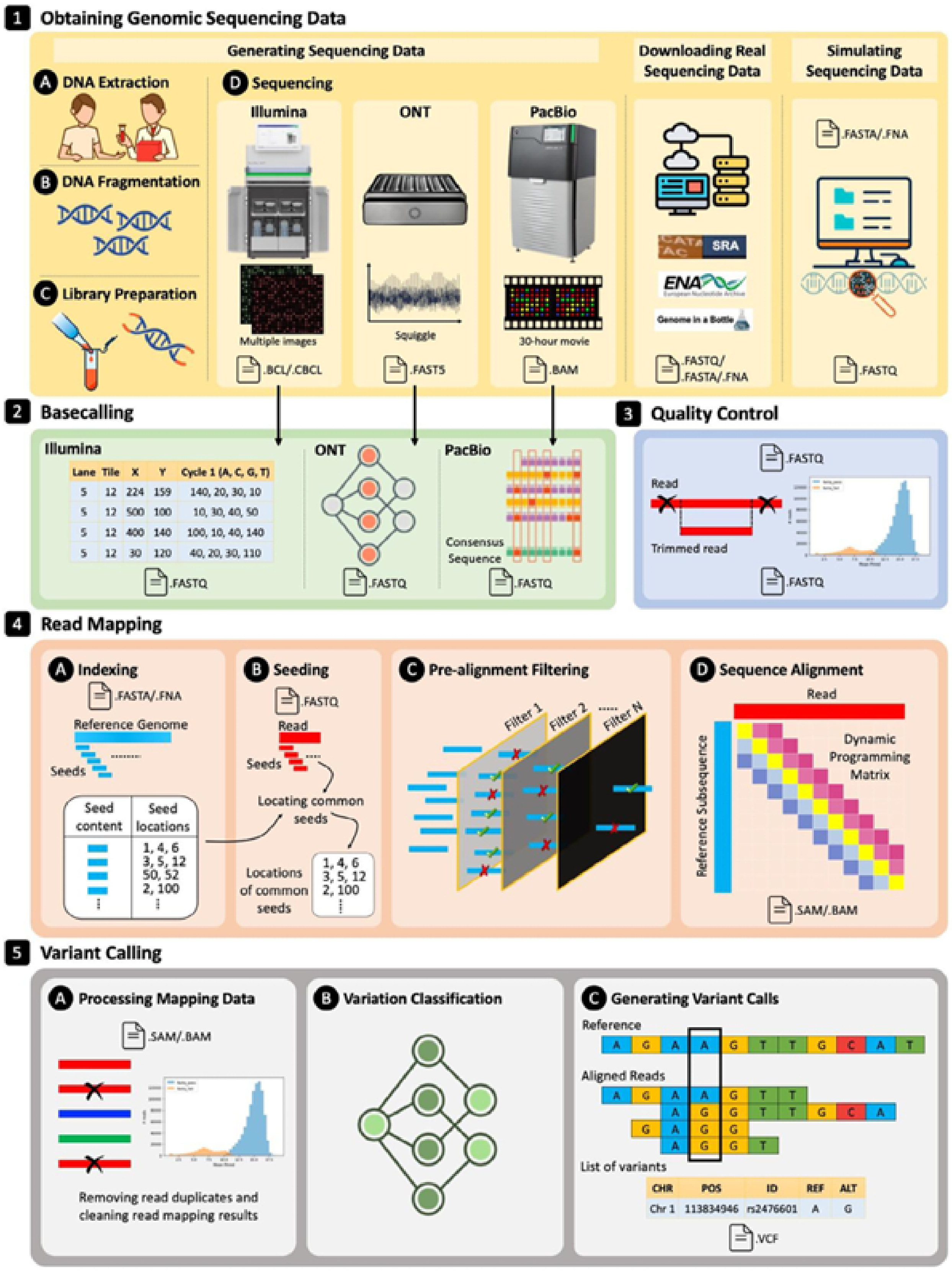
The term omics refers to the study of the collective characterization and quantification of biological molecules that determine the structure, function and dynamics of an organism or a group of organisms. The omics group of disciplines includes genomics, proteomics, metabolomics, metagenomics, phenomics and transcriptomics, each of which focuses on the study of their respective biological macromolecules or biological processes. The related suffix -ome is used to refer to the objects of study of the above omics fields therefore the genome is study through genomics, the proteome through proteomics etc. There are numerous omics fields with the most explored ones being genomics, which study the genomes of organisms, proteomics, which focuses on the exploration of the structure and function of proteins, and the transcriptomics which involve the study of all types of RNA molecules [17].

Genomics, which is the most explored omics field, focuses on studying the structure, function, evolution, mapping and editing of genomes, while in contrast to genetics, aims at the holistic characterization and quantification of all of an organism's genes, their interrelations and influence on the organism, rather than just understanding individual genes and their roles in inheritance.

A genome is composed of the genetic information of an organism including the nucleotide sequences of DNA (or RNA in RNA viruses). Most eukaryotic organisms have a main large nuclear genome and a smaller mitochondrial genome, which includes genes encoding mainly mitochondrial proteins involved in cellular respiration. The nuclear genome includes both protein-coding genes and non-coding genes, functional regions such as regulatory sequences and junk DNA which has no evident function. Protein-coding genes are transcribed into mRNA which is then translated into polypeptide chains in the cytoplasmic ribosomes that after post translational alterations becomes functional proteins. Non-coding sequences on the other hand encode for other types of RNA (tRNA, microRNA, rRNA, piRNA etc.) and regulatory sequences that control gene expression or have no apparent function. The nuclear genome is organized into chromosomes which are made up of DNA molecules and packaging proteins, while in diploid organisms' chromosomes are found in pairs. The genome however refers to only one copy of each chromosome. For example, the standards reference genome of humans consists of a copy of each of the 22 autosomal plus one X and one Y chromosome. A DNA genome sequence is the complete list of nucleotides (Adenine/A, Thymine/T, Cytosine/C, Guanine/G) that make up all the chromosomes of an individual or a species, while sequencing is the process of determining this sequence. The sequencing of genomic molecules has greatly propelled research and innovation in biomedicine and other life sciences, with applications spanning clinical medicine, outbreak monitoring, and the investigation of pathogens and urban microbial ecosystems. These advancements have been primarily fueled by the successful mapping of the human genome and the emergence of high-throughput sequencing technologies, which have significantly lowered the cost of DNA sequencing. To manage the increasing volume and complexity of sequencing data, the bioinformatics community has created a wide range of software tools. These tools have transformed modern biology, becoming essential to life sciences and emphasizing the urgent demand for faster and more efficient computational solutions. The following figure includes all the intermediate steps of the genome analysis pipeline which namely include obtaining genomic sequencing data, basecalling, read mapping and variant calling [1].

2 DNA Sequencing Technologies

The first attempts to determine DNA sequences was proposed in 1970 by Ray Wu, who established a location-specific primer extension strategy that involved DNA polymerase catalysis, synthetic location-specific primers and specific nucleotide labeling. The technique evolved by Frederick Sanger, who had already successfully performed protein sequencing in the 1950s, by using a



Lane	Tile	X	Y	Cycle 1 (A, C, G, T)
5	12	224	159	140, 20, 30, 10
5	12	500	100	10, 30, 40, 50
5	12	400	140	100, 10, 40, 140
5	12	30	120	40, 20, 30, 110

CHR	POS	ID	REF	ALT
Chr 1	113834946	rs2476601	A	G

Figure 3.1: Overall Genome Analysis Pipeline [2]

primer-extension strategy to accelerate DNA sequencing. Briefly, this method is based on the selective incorporation of chain-terminating dideoxynucleosides (ddNTPs) by DNA polymerase during in vitro DNA replication. The first DNA sequencing of a bacteriophage was performed successfully in 1977 using two techniques, that of Sanger's and that of Maxam's, while it was not until the 2000s that collective scientific efforts and research made the sequencing of human DNA possible with a cost of approximately 3 billion USD. This sequencing era was referred to as first generation sequencing, since which many advancements have been made in terms of increasing sequencing throughput while reducing the cost of sequencing per nucleotide. To generate genomic sequencing data fragments, also known as **reads**, three basic steps are needed, including sample collection, library preparation and sequencing. The first two steps are performed within the context of the wet laboratory prior to performing the actual sequencing process with the use of special sequencing machines that vary in terms of many parameters. More specifically, each sequencing machine has different properties including sequencing throughput, which is defined as the number of bases generated by the sequencing machine per second, read length, sequencing error rate, type of raw sequencing data, sequencing machine size and cost. Depending on the number of bases existed in a single read, those are categorized into three categories: short reads (a few hundred bases), ultra-long reads (from hundreds to millions of bases) and accurate long reads (up to some thousands of bases). As further explained below there seems to be a tradeoff between the read length and accuracy with sequencing machines producing shorter reads having naturally higher accuracy [9]. Even though there seem to be many differences between the existing sequencing technologies, most of them follow the same basic principles including DNA fragmentation as part of the library preparation protocol. This method includes intentionally breaking the DNA strands into fragments through i.e. resonance vibration, enabling exploitation of a large number of DNA fragments for higher sequencing yield, as in the case of sequencing really long DNA fragments its quality decreases due to the limited lifetime of polymerase enzymes used for the process. Apart from generating sequencing data, there is also the option to use publicly available sequences found in online databases such as the Sequence Read Archive (SRA), the European Nucleotide Archive (ENA) or the NCBI Reference Sequence Database (RefSeq) databases. The read files found in these databases are usually in one of the following formats which include the read sequences together with relevant information such as a unique identifier for each read, in the case of the FASTA file format, or more information such as read quality scores in FASTQ files. It is also possible to generate sequencing data using computer simulations when lacked other resources, through special read simulators that consider the different mechanisms and chemistry of various sequencing machines to be able to produce reads like those naturally extracted by each one of these technologies. These simulators use a reference genome to extract read sequences of given length incorporating types of genetic variations and a sequencing error. The output of these simulators is in FASTA, FASTQ or BAM format which can be further used by the mapping pipeline.

Sequencing machines, as already stated, are divided into three basic categories depending on the length of reads they are able to produce. Short read sequencing (second generation sequencing) technologies are able to generate subsequences of length 100-300 bases. Illumina machines are the ones dominating this technology, providing short reads with low sequencing error rate (0.1% of the read length) and allowing the generation of over 6 terabases in a single sequencing run. Third generation sequencing, also known as ultra-long read (or nanopore) sequencing, was first introduced in 2014 and is now able to sequence over 14 terabases in a single sequencing run in about 3 days with a read accuracy reaching up to 98% in some cases. This technology is implemented into very small sequencing machines that measure electrical current changes as nucleic acids are passed through the nanopore, which are then during the base-calling step decoded into an actual DNA sequence. Finally, third and fourth sequencing technologies introduced by Pacific Biosciences (PacBio) are able to generate long (up to 10-30kbases) and accurate (99.9%) reads. These machines can generate over 35 gigabases in a single sequencing run in about 30 hours. In order to better understand the advantages and disadvantages of the above types of sequencing technologies it is important to understand the two basic processes for which these reads may be used. One is read mapping, which as will be better demonstrated later, basically compares the read sequence to a reference sequence and tries to find candidate region of origin to be able to compose the entire genome of a sample, and the other one is de novo gene assembly which does not require a reference genome but rather constructs the genome sequence from overlapping read sequences without the need for comparison with a reference genome. Therefore, even though short reads introduce low sequencing error and high sequencing throughput, they are not the best candidate when it comes to de novo genome assembly since repetitive sequences found in short reads pose challenges. However, their equivalent length is really useful in performing read mapping in parallel, by load balancing between several CPU threads. Ultra long reads on the other hand require more complex computational analysis with higher executing time and memory footprint, even though they provide more contiguous assembly than short reads. They also have a higher error rate which introduces the need of incorporating new computational steps to polish errors in the assembly. Finally, high accurate long reads may have been a key enabler of the improvements in human genome assembly, however their high cost of sequencing and computationally expensive basecalling step imposes some challenges in their widespread usage [2].

3 Basecalling and Quality Control

Most of the sequencing technologies described above do not provide read sequences in the DNA alphabet, instead they output a native format that needs to be converted to a standard character format via basecalling algorithms, which are the first computational step of the genome sequencing pipeline. Basecalling basically translates the noisy electrical signals generated to strings of DNA nucleotide bases (A, T, G, C) and its accuracy and speed directly affect the respective parameters of the overall pipeline. The basecalling mechanisms of the three major types of technologies differ from one another. For instance, Illumina technology outputs multiple images of fluorescence intensities for each sequencing cycle which are stored as binary data in a CBCL file. Similarly, the PacBio technology generates high accuracy long reads input for the basecalling algorithm, a 30-hour movie of continuously captured fluorescence traces. Nanopore sequencing involves the conversion of raw electrical signals which can be very challenging due to the signals' stochastic behavior, low SNR, long signal dependencies of event data on neighboring nucleotides and the inability of the sensors to measure the change of electrical signal due to one single nucleotide but rather a group of nearby nucleotides. Due to the special nature of the basecaller needed for such peculiar signals there are many software-based and hardware-based methods proposed to increase both their accuracy and performance [2].More specifically, most software approaches involve the use of Deep Neural Networks, Hidden Markov Models, LSTMs and RNNs while hardware-based implementations tend to accelerate the above approaches by designing custom architectures for software/hardware co-design or introducing novel process in memory (PIM) techniques (Swordfish [18],GenPIP [19]). There are also novel approaches that can accurately perform real-time mapping of raw nanopore signals, without the need of translating them, by using a hash-based seed-and-extend mechanism (RawHash2 [20], Rawsamble [21]). After basecalling, along with the translated base a read quality score is reported in the quality control (QC) step of the pipeline. This step is responsible for evaluating the quality of the entire read sequence through examining the intrinsic quality of the read sequence before and after basecalling, assessing the length of the read and counting the number of ambiguous bases in the read. Quality evaluation is also done on individual bases apart from the QC of the entire read sequence. Depending on the quality of the bases in different regions in the read the QC algorithm is able to mask out an untrustworthy region in the read or to trim beginning or/and ending trailing regions that are more likely to be of lower quality.

4 Indexing and Seeding

Indexing is the part of the pipeline that deals with creating a large index database using subsequences extracted from a reference genome to enable its quick and efficient querying. These subsequences are called **seeds**. Then, the mapper utilizes the prepared index database to identify one or more potential regions of the reference genome that are likely to resemble each read sequence. This is achieved by matching subsequences extracted from each read with those stored in the index database. It is important to note that the extraction of these subsequences of the reference and the reads is performed using the same algorithm with the main difference being that the indexing step stores the seeds of the reference genome in an indexing database while the seeding step uses the read extracted seeds to query the indexing database [2]. Indexing is performed only once in order for a look up structure to be generated for a reference genome, therefore it is not on the critical path of the entire genome analysis pipeline nor constitutes a bottleneck. The index structure that represents the reference genomes includes the seeds or a correspondent hash value as well as information on the position of the seed in the reference, therefore it is evident that the number of extracted seeds, their length and their frequency can greatly affect the overall memory footprint, performance and accuracy of read mapping. To alleviate these issues three major strategies are explored to improve indexing and seeding techniques, including better sampling, more efficient indexing data structures and minimizing data movement through specialized hardware. The purpose of sampling seeds is to eliminate redundant information that can be derived from the extracted seeds. For instance, selecting all possible overlapping subsequences of length k (known as **k-mers**) as seeds results in each base appearing in k seeds.This leads to an unnecessarily large memory footprint and inefficient querying due to the high number of seed hits. To address this, state-of-the-art read mapping algorithms (e.g. minimap2) aim to reduce the number of seeds included in the index structure by sampling a smaller subset of k-mers from the complete set. A common approach for this reduction involves imposing an order (e.g. lexicographical or based on hash values) on groups of w overlapping k-mers and selecting only the k-mer with the smallest order from each group. These selected k-mers are referred to as minimizer k-mers. The minimizer-based method ensures that at least one seed is retained from each group of k-mers, a property known as the windowing guarantee. This approach minimizes information loss, which depends on the chosen values of k and w. It is also possible to filter the minimizer seeds based on their frequence of occurrence and filter out the most common ones. An alternative approach to that of the minimizers is that of the syncmers that has shown to provide more uniform distribution of seeds by selecting a seed as a minimizer as the one whose substring, located at a fixed position, has the smallest order compared to the substrings of other seeds [3]. This method ensures a consistent gap between consecutive minimizers, allowing read mappers to identify mapping locations for reads that might remain unmapped when using minimizer-based read mapping methods. Other approaches include minimizer sampling without windowing, by extracting different minimizers from the entire read with the use of different hash functions (MinHash). After selecting an appropriate method for extracting seeds, the next step is to store or query them efficiently using an index. A straightforward data structure for finding seed matches is a hash table, which uses the hash value of each seed as a key and the corresponding location lists as values. Hash tables continue to be utilized by state-of-the-art read mappers due to their practical performance. The choice of hash function is a critical design decision for hash tables. Ideally, the hash function should exhibit low collision rates to ensure that distinct seeds are assigned unique hash values. Conversely, increasing collision rates for highly similar seeds can enhance sensitivity by grouping similar seeds under the same hash value. In some state-of-the-art indexing algorithms other data indexing structures have been proposed such as the FM-index which provides a compressed representation of the full-text index enabling the querying of seeds of arbitrary length with a lower memory footprint. Finally, since both indexing and seeding are memory-intensive tasks they require alternative architectures and approaches to reduce data movement during their runtime. Such approaches include processing in memory designs, the incorporation of hardware accelerators on DRAM chips (MEDAL) or including accelerators in SSDs so as for the index to remain in storage (GenStore) [4].

5 Pre-alignment Filtering

After identifying one or more potential mapping locations for the read in the reference genome, the read mapper evaluates the similarity between the read and the corresponding segments extracted from these locations. Although these segments share common seeds with the read, they may still be either similar or dissimilar. To avoid performing computationally expensive sequence alignment algorithms on dissimilar sequences, read mappers often employ filtering heuristics known as pre-alignment filters. The core concept of pre-alignment filtering is to rapidly estimate the number of edits between two sequences and use this estimation to determine whether a detailed, computationally intensive alignment based on dynamic programming (DP) is necessary. To characterize two sequences as similar or dissimilar the metric of the edit distance is introduced and defined as the minimum number of single character changes needed to convert one sequence into the other. By skipping the DP-based alignment for cases where the sequences differ by more than the edit distance threshold, significant computational time is saved. If the estimated edit distance exceeds the threshold, the sequences are deemed dissimilar, and alignment calculations are bypassed, since only genomic sequences pairs with an edit distance less or equal to the edit distance threshold may provide useful data for the genomic analysis. Therefore, pre-alignment filters play a crucial role in quickly eliminating some of the dissimilar sequences before passing them to the computationally expensive optimal alignment algorithm [5]. Most state-of-the-art pre-alignment filters algorithmic principles lie on either the pigeonhole principle, base counting, q-gram filtering or sparse DP. The pigeonhole principle when applied to the pre-alignment filtering problem is translated as if two sequences differ by E edits, they should share a common subsequence among any set of E+1 non overlapping subsequences. One example of a pre-alignment filter that efficiently works on CPU, GPU and FPGA and uses the pigeonhole principle is the SneakySnake filter which maps the pre-alignment filtering problem to the single net routing problem (SNR problem) in VLSI chip layout. The SNR problem provides a straightforward example to help illustrate and visualize the pre-alignment filtering problem. There are notable similarities between the two problems [5]. Both aim to determine an optimal solution that minimizes propagation delay, which corresponds to minimizing the number of edits in the context of pre-alignment filtering. They also share a characteristic flexibility in assigning connections; in the SNR problem, the source and destination nodes can be freely chosen from the I/O pads around the chip, just as matching regions can be selected in pre-alignment filtering. Additionally, both problems involve handling obstacles, such as edits in the case of pre-alignment filtering, while adhering to specific constraints in their respective contexts. The base counting is the simplest pre-alignment filtering idea that compares the number of bases of each type (A, T, C, G) in the read with the corresponding base numbers in the reference sequence segment. For example, if the read has two more As than the reference, then the minimum edit distance between the two sequences would be greater or equal to two. A criterion that many of these filters use is to add the absolute differences of these numbers between the two sequences and after dividing it in half see if the resulting number is greater than the edit threshold and if so, consider the two sequences dissimilar and reject the alignment. Using this method a very high percentage of dissimilar sequences are rejected [2]. The q-gram filtering approach evaluates all possible overlapping substrings of length q (referred to as q-grams) within a sequence. For a sequence of length m, there are $mq+1$ overlapping q-grams, obtained by sliding a window of length q across the sequence. A single difference in the sequence can affect up to q overlapping q-grams. Thus, the minimum number of shared q-grams between two similar sequences can be calculated as $(mq+1)q \cdot E$. This approach relies on straightforward operations such as summations and comparisons, making it highly suitable for hardware acceleration implementations, as demonstrated in tools like the GRIM-Filter [22]. Finally, sparse DP algorithms leverage exact matches (seeds) between a read and a reference segment to optimize execution time. This exact match regions are also known as anchors in the literature. By excluding the regions corresponding to these seeds from the edit distance estimation, these algorithms focus only on the remaining unmatched areas, as the seeds have already been identified during the indexing phase. To enhance efficiency, sparse DP filtering methods connect overlapping seeds to form extended chains, using the cumulative length of these chains as a criterion for filtering sequence pairs. This technique, commonly referred to as chaining, is implemented in tools such as minimap2. Chaining in many genome analysis pipelines may stand as an individual step in between the pre-alignment filter and the aligner, as proposed in minimap2. After filtering, overlapping **seed hits**, meaning matches between the seeds of the indexed reference and the read's extracted seeds, are identified. Chaining then links these overlapping seeds together into longer chains, where each seed in the chain is connected by a small gap. This process helps group together regions of the read and reference genome that are likely to align well, reducing the search space for the subsequent alignment step.

6 Alignment

After filtering out the majority of mapping locations that result in dissimilar sequence pairs, the read mapping process computes the sequence alignment information for each read and reference segment extracted from the identified mapping locations. Sequence alignment calculations are often expedited using one of two methods including enhancing the performance of optimal affine gap scoring dynamic programming (DP)-based algorithms through hardware accelerators, or employing heuristics that trade off alignment score optimality to reduce alignment time. Affine gap scores are typically computed using the Smith-Waterman-Gotoh algorithm, which assigns linear integer scores for matches and substitutions, and affine integer scores for gaps. While affine gap scores offer more flexibility than linear or unit (edit distance) costs, they come with a higher computational cost due to the additional complexity. Despite over three decades of efforts to accelerate sequence alignment, the fastest known edit distance algorithm still has a nearly quadratic time complexity, $O(\frac{m^2}{logm})$ for a sequence of length m, which is a proven lower bound under the strong exponential time hypothesis. A common strategy to reduce computational work without compromising optimality is to introduce an edit distance threshold, limiting the maximum number of allowed single-character edits in the alignment. This leads to the computation of only a subset of the dynamic programming (DP) table entries, specifically diagonal vectors, as originally proposed in Ukkonen's banded algorithm. The number of diagonal vectors required for calculating the DP matrix is 2E+1, where E represents the edit distance threshold, thus reducing the time complexity to $O(m \cdot E)$. This approach is effective for short reads, where sequencing error rates are typically low, allowing for a lower edit distance threshold. However, for long reads with high sequencing error rates (up to 20% of the read length), a higher edit distance threshold is required, resulting in the need to compute more entries in the DP matrix compared to short reads. Many hardware accelerated aligners have been proposed for CPUs, GPUs, FPGAs and ASICs enabling the parallel computation of alignments of many independent sequence pairs or provide application-specific, power and area efficient solutions to accelerate sequence analysis. Other aligners limit their functionality or sacrifice the optimality of the alignment solution in order to reduce alignment execution time, such as the Myers' bit-vector algorithm or the Darwin alignment accelerator. Another such example is the GenASM accelerator [7] which utilizes the Bitap algorithm to perform approximate string matching using bitvectors. There are alternative methods that restrict the number of entries calculated in the DP matrix through one of two strategies: (1) utilizing sparse DP, or (2) applying a greedy approach to maintain a high alignment score. However, both methods have the potential to yield suboptimal alignments. The first method employs the same sparse DP technique used in pre-alignment filtering, but applies it during the alignment step, as seen in the Exonerate tool. This approach involves performing DP-based alignments only between non-overlapping chains to quickly estimate the total number of edits. The second method, implemented in X-drop , works by (1) avoiding the calculation of entries (and their neighbors) whose alignment scores are significantly lower than the highest score encountered so far (with the threshold being a user-defined parameter), and (2) terminating early when it is unlikely that a high alignment score can be achieved. The X-drop algorithm guarantees the optimal alignment between relatively similar sequences only for certain scoring functions. At the end of the alignment step the alignment algorithm has determined which of the candidate mapping locations in the reference matches best with the input read. Traceback is also performed between the reference and the input read to find the optimal alignment, otherwise the most likely, based on the scoring function (e.g. least number of edits). Then a CIGAR string is used to defined this optimal alignment, which shows the sequence and position of each match, substitution, insertion and deletion for the read with respect to the reference mapping location (e.g. a CIGAR string looks like this 32M1S1D1S65M, means that two sequences have the same 32 first and 65 last characters and the query sequence has two substitutions and one deletion in between). Generally, the alignment step is very computationally costly and constitutes the major bottleneck in the entire pipeline and thus its acceleration is of great importance.

7 Variant Calling

After having successfully completed the alignment task, the genome sequencing pipeline is concluded since the reads have been aligned to the reference genome and therefore their position of origin is now known. The additional step of variant calling aims at finding the differences between an individual or a group of individuals, such as a population, and a reference genome of a species. Detecting the variants is a very important step in genome analysis as the phenotypes of an individual or population differentiate from another due to these variations. This can also be very helpful when examining the genomics of a specific disease found in certain populations or specific individuals, to enhance the development of diagnostic and therapeutic tools. How most of the variant calling tools work involve the detection of the locations in the genome where the reference genome and the sequencing reads have different bases. Most variant callers initiate by processing the read mapping data, organizing them and sorting them to facilitate processing overlapping alignments. After having done so, the callers detect the actual genomic variations in the processed read mapping data and classify the variations into SNPs, short indels, and SVs. Using these data, they construct a local graph assembly that shows both homozygous and heterozygous alternate loci and classifies the probability of each allele using Hidden Markov Models. The output of this step is a VCF file which includes the interpretation of the classification output without the low-quality variations which are filtered out. The state-of-the-art most frequently used variant caller is the DeepVariant [23] caller which performs all the above steps. Figure 3.2 summarizes the basic steps of the genome analysis pipeline:

8 Genome Graphs

The current human reference genome is organized as a linear haploid DNA sequence. However, this design has practical constraints due to the high genetic diversity observed among human populations. On average, an individual genome contains 3.5–4.0 million single-nucleotide polymorphisms (SNPs)

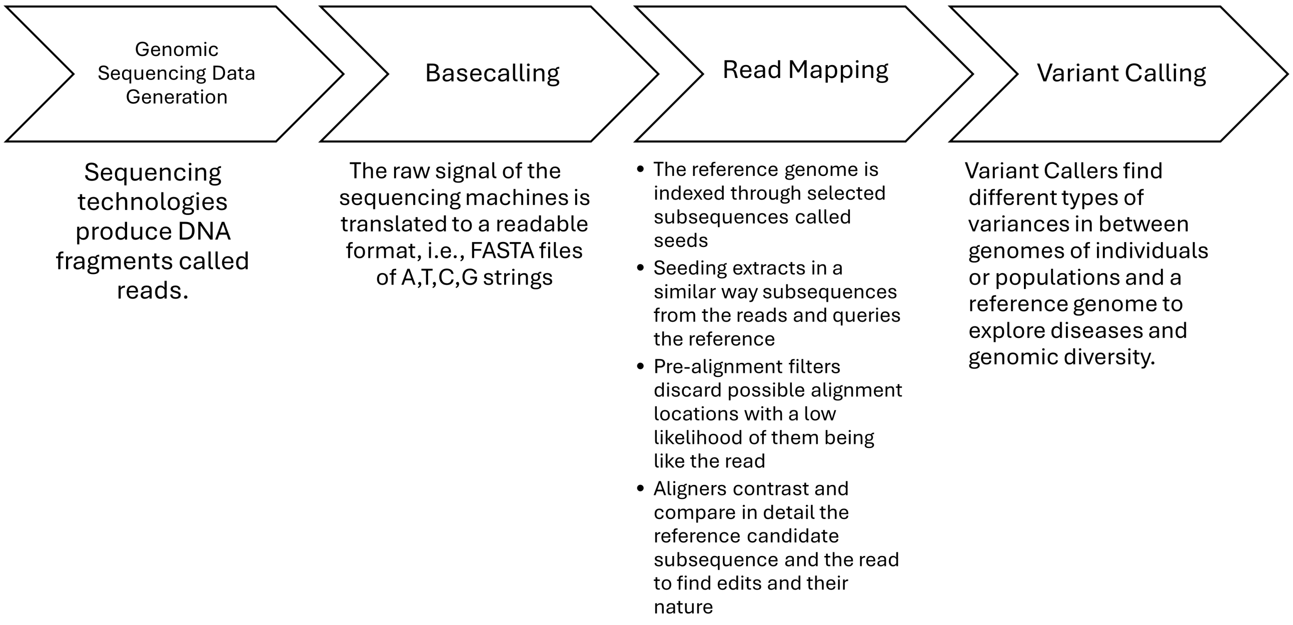

Genomic Sequencing Data Generation
Basecalling
Read Mapping
Variant Calling
Sequencing technologies produce DNA fragments called reads.
The raw signal of the sequencing machines is translated to a readable format, i.e., FASTA files of A,T,C,G strings
• The reference genome is indexed through selected subsequences called seeds
• Seeding extracts in a similar way subsequences from the reads and queries the reference
• Pre-alignment filters discard possible alignment locations with a low likelihood of them being like the read
• Aligners contrast and compare in detail the reference candidate subsequence and the read to find edits and their nature
Variant Callers find different types of variances in between genomes of individuals or populations and a reference genome to explore diseases and genomic diversity.

Figure 3.2: Summary of the Genome Analysis Pipeline

or small insertions and deletions (indels), as well as approximately 2,500 large structural variations (SVs) when compared to the reference genome. These genetic differences can lead to errors in mapping sequencing reads, particularly when reads overlap SV breakpoints, causing them to either align incorrectly or fail to align entirely. As a result, mapping accuracy varies considerably depending on the genomic region and the genetic divergence of the sample being analyzed. Such inaccuracies can lead to undetected true variants (false negatives), incorrectly identified variants (false positives), and complications in other downstream applications that depend on precise read mapping. In order to incorporate all these genetic variations and make the alignment (and perhaps the variant calling step) more efficient, genome graphs have been introduced as a reference structure that includes all these types all genetic variation present within populations. In general, genome graphs are used in applications including variant calling, genome assembly, error correction and multiple sequence alignment [24]. Genome graphs integrate the reference genome with genetic variations and polymorphic haplotypes, offering a shift away from aligning sequences to a single reference genome. Instead, they utilize the genetic diversity observed in human populations. Recent studies have demonstrated the advantages of graph-genome representations for aligning human whole-genome sequencing data. These graph-based approaches not only improve the alignment of sequencing reads and haplotype resolution but also provide a more comprehensive representation of genetic diversity through a human pan-reference genome.

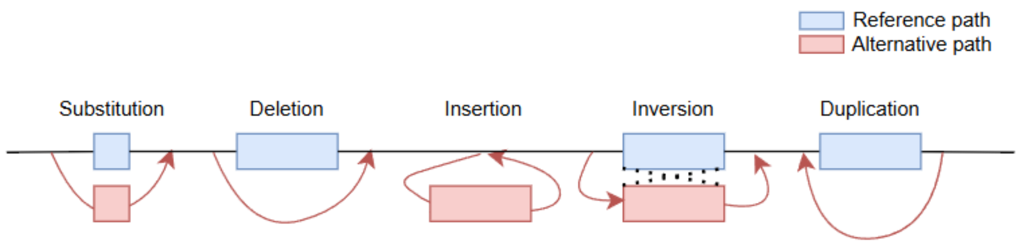


Figure 3.3: Schematic Representation of Different Types of Variances in a Reference Graph

This approach is particularly valuable for studying African genomes, which exhibit substantial sequence variation. Many genomic analysis pipelines nowadays incorporate algorithms that use a graph structure as the reference genome instead of a linear sequence. The first computational step in graph-based sequence analysis involves constructing the graph structure. This is achieved by combining the reference genome with variants from variation databases to create alternative paths within the graph. Each genetic variant introduces an edge, expanding the range of potential sequences represented in the graph. There are different tools that can be used to construct a genome graph with the VG toolkit and minigraph. The graphical representation of Figure 3.3 shows the different possible variants existing in a DNA molecule or in our case within the reference graph. In this work and in the case of aligners in general we neglect the inversion and duplication cases since these alterations cannot be described separately by the CIGAR representation.

9 Sequence-to-Graph Alignment

After building the graph, it is indexed using a specialized algorithm. Sequencing reads are then aligned to the graph (sequence-to -graph alignment), which provides greater accuracy compared to alignment against a single reference genome. For instance, a genetic locus with two distinct haplotypes—one containing four SNVs and the other two SNVs—can be accurately resolved using graph alignment. Unlike linear alignment, which is biased toward the reference and fails to align reads with multiple SNV mismatches, graph alignment correctly places all reads. Following alignment, additional tools are used to extract relevant information from the graph and present it in a format suitable for interpretation by biomedical researchers. Alternatively, the alignment results can be output in the standard SAM format. A simplified schematic of genome graph alignments is shown in Figure 3.4 for a variable region with six SNVs (left). The SNV patterns in the sequencing reads indicate that the sample is heterozygous at this locus. One allele contains two SNVs, represented by blue and yellow rectangles above the graph, while the other allele contains four SNVs, depicted

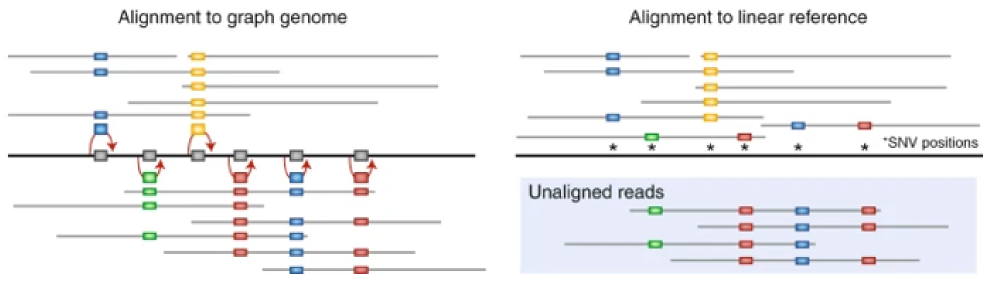


Figure 3.4: Sequence-to-Sequence vs Sequence-to-Graph Alignment [24]

as a sequence of green, red, blue, and red rectangles below the graph. All reads are successfully aligned to the graph reference without mismatches. On the right, the same reads are aligned using a linear approach that permits up to two mismatches. This linear alignment is biased toward the reference sequence, resulting in the failure of aligning four reads from the divergent allele.

Chapter 4

Experimental Setup and Development Tools

The first part of the chapter focuses on Field-Programmable Gate Arrays (FPGAs) and more specifically their architecture and the tools used to create designs eligible for them, focusing on the use of High-Level Synthesis (HLS). Additionally, a brief comparison of FPGAs with other computational platforms (CPUs, GPUs and ASICs) is presented to demonstrate the advantages of FPGA-based acceleration.

The second part of this chapter outlines the experimental setup (computational infrastructure and FPGA hardware platform) and some features of the Vitis HLS tool which was primarily used to develop the proposed design.

1 Field-Programmable Gate Arrays (FPGAs) and High Level Synthesis (HLS)

High-level synthesis (HLS), also known as C synthesis, algorithmic or behavioral synthesis is an automated design process of transforming an abstract behavioral specification of a digital system into a register level structure that realizes the given behavior. It is a design tool that allows the designer to work at a higher level of abstraction to create high-performance hardware while having better control over optimization of their design [25]. HLS can be divided into distinct steps to convert a behavioral description into circuit modules by placing cells of physical elements, such as transistors, into several rows and connecting I/O pins through routing in the channels found in between the cells. Figure 4.1 shows the different HLS stages, namely: Allocation, Binding and Scheduling.

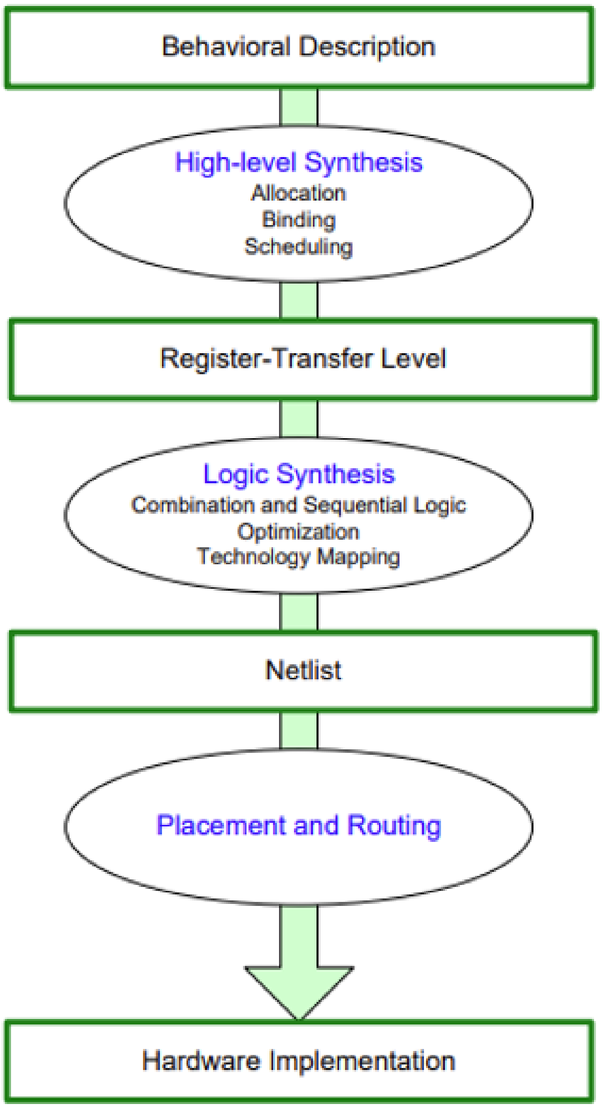


Figure 4.1: From Behavioral Description to Hardware Implementation

The first step of HLS includes allocating necessary resources for the computations needed in the provided behavioral description which will then be bound to the corresponding operations. Then these operations are scheduled, meaning deciding which operations will occur in which operation cycle, and the output of the high-level synthesizer is an RT-level description which is then logically synthesized to produce an optimal gate netlist. Binding then determines which hardware resource implements each scheduled operation.HLS, therefore, provides an easy and efficient way to accelerate parts of their algorithms by routing their hardware implementations on compilation reprogrammable target platforms such as a Field Programmable Gate Array (FPGA). A characteristic of these platforms is that they are significantly faster for some applications due to their parallel nature and optimality in terms of the number of gates used for certain processes.That is why a big trend is the use of FPGAs as hardware accelerators utilized to accelerate certain parts of an algorithm and share that

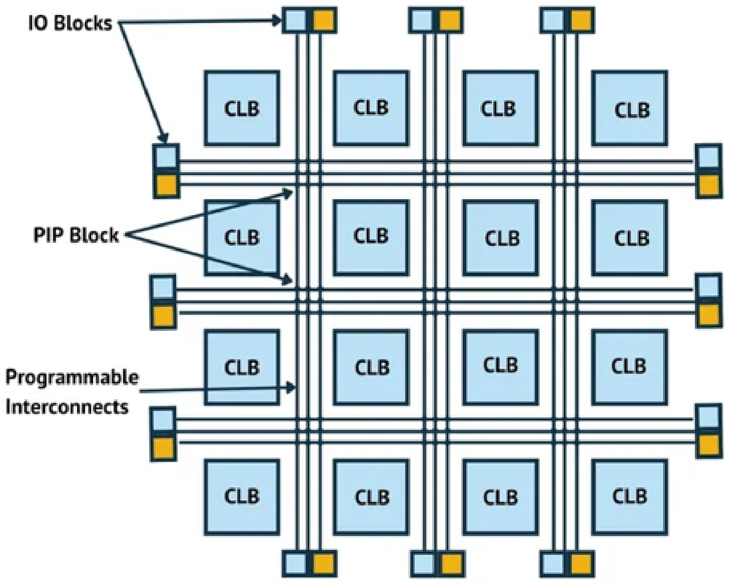


Figure 4.2: Simple FPGA Diagram

part of computation between the FPGA and a general-purpose processor. A typical FPGA architecture consists of an array of logic blocks (configurable logic blocks-CLBs), I/O pads and routing channels between the blocks. A logic block consists of a few blocks including a 4-input LUT, a full adder and a D-type FF (Figure 4.2) [26].

Their configuration is generally through hardware description languages such as Verilog and VHDL, which are used to produce the RTL output of various high-level synthesizers such as the Vivado HLS. The Vivado HLS is a broadly used high level commercial synthesizer developed by Xilinx whose flow is presented in Figure 4.3.

After the synthesis step the primary output is the implementation in RTL format, provided in Verilog or VHDL formats, which is further synthesized to a gate-level implementation and an FPGA bitstream file by logic synthesis [25].

1.1 FPGA comparison with CPUs, GPUs and ASICs

FPGAs have been used to accelerate various applications by accommodating tailored computationally intensive applications overpassing Central Processing Units (CPUs) and Graphics Processing Units (GPUs) in Neural Networks', Networking, Graph Processing, Genomics and other applications. CPUs and GPUs are fundamental computing components that have been widely used for decades. CPUs were the original computing processors, built to execute specific instruction sets for versatile, general-purpose computing. While this design provides great flexibility across various applications, CPUs struggle with parallel processing as computational requirements have increased. Furthermore, GPUs were developed to overcome these parallel processing limitations by sac-

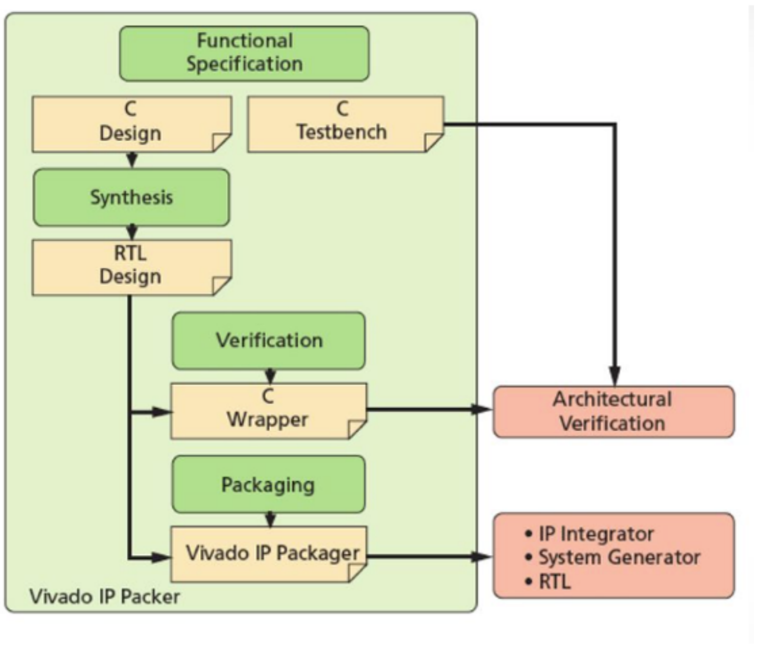


Figure 4.3: Vivado HLS Flow

rificing some CPU flexibility in exchange for massive simultaneous computation capabilities. With thousands of small processing cores working together, GPUs excel at handling data-intensive tasks that require concurrent processing.

However, certain applications benefit most from purpose-built hardware, leading to the development of Application-Specific Integrated Circuits (ASICs). Despite their optimization advantages, ASICs have major disadvantages: they lack adaptability, are expensive to produce, require lengthy development cycles, and take considerable time to reach market.

FPGAs strike a balance between custom hardware performance and practical development constraints. FPGAs can be reconfigured for specific application requirements while maintaining flexibility. Their design features pre-built logic connections that simply need activation, enabling rapid hardware prototyping without computationally expensive simulations. Modern FPGA development tools such as the Vivado Suite presented above, have become highly advanced, automatically managing logic element connections and allowing engineers to concentrate on application-level behavioral and logical simulation. This streamlined process makes FPGAs an attractive option for projects needing customized performance with reasonable development timelines.

1.2 HLS Pragmas

The main advantage of the HLS tools is maintaining an efficient resource usage while keeping a high performance, by enabling the designer to add HLS-defined directives (pragmas) which result in the synthesis of an optimized IP block. Depending on the performance and area specifications, the directives can create

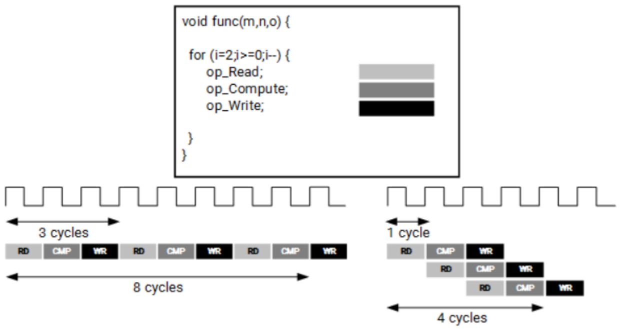


Figure 4.4: Example with and without Loop Pipelining

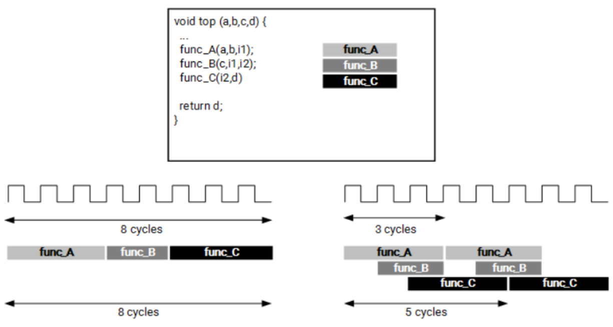


Figure 4.5: Example with and without Dataflow Pipelining

optimized results taking those into account. Supervision of the performance is achieved by inspecting the synthesis report produced by the tool, which includes metrics such as area, latency, initiation interval and resources allocation. The incorporation of directives alters those parameters, and they can be appropriately applied to functions, loops, arrays and in arrays including combinations of those elements. More specifically, function related directives aim at enabling functions to execute concurrently (dataflow, pipeline directives) or removing function hierarchy to reduce function call overhead and examine logic optimization (inline directive). Loop related pragmas may reduce the cost of transition between different loops and within the same loop to allow the parallel execution of multiple loops (unroll, pipeline directives). More specifically, the pipeline pragma reduces the initiation interval (II) for a function or loop allowing the concurrent execution of operations, since a pipelined function or loop can process new inputs every N clock cycles where N is the II of the loop function, as since in the example of Figure 4.4 [27].

The dataflow pragma enables task-level pipelining allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and the overall throughput of the design. It allows for the operations in a function or loop to start operating before the previous function or loop completes all its operations, as seen in the example of Figure 4.5 [27].

The unrolling directive is used to create multiple independent operations rather than a single collection of operations by transforming loops through creating multiple copies of the loop body in the RTL design, which allows some (partial unroll) or all (complete unroll) loop iterations to occur in parallel. Finally, array related pragmas (array partition, array reshape and array map directives) change accordingly the array layout by reshaping or partitioning to remove bottlenecks introduced by the fact that block-RAMs (which are used to implement arrays) can be at most dual-port (maximum two elements of the same array can be accessed at the same time).

1.3 Interface Synthesis and the M_AXI Interfaces

The arguments of the top-level function in a Vitis HLS design are synthesized into interfaces and ports which group multiple signals to define the communication protocol between the HLS design and components external to the design [27]. The type of interfaces created automatically by Vitis HLS depend on the data type and direction of the parameters of the top-level function, the default interface configuration settings, the target flow (including the Vivado IP flow and the Vitis Kernel flow) for the active solution and any specified interface directives. The interface defines three elements of the kernel, including the channels for data to flow into or out of the HLS design, the port protocol to control the flow of the data through the data channel and the execution control scheme for the HLS design. Vitis HLS supports memory, stream and register interface paradigms:

- Memory (m_axi): the data is accessed by the kernel through memory such as DDR, HBM,PLRAM/BRAM/URAM. Used to implement arrays and pointers to arrays.

- Stream (axis): the data is streamed into the kernel from another streaming source (e.g. another kernel) and can also be streamed out of the kernel. Used to implement hls::stream

- Register (s_axilite):The data is accessed by the kernel through register interfaces and performed by software register reads/writes.Used to implement scalar, pointer to scalars and reference.

AXI4 memory-mapped (m_axi) interfaces are a convenient way of sharing data across different elements of the accelerated application, such as between the host and kernel having independent read and write channels, supporting burst-based accesses and providing a queue for outstanding transactions. By default, Vitis HLS groups arguments with compatible options into a single m_axi interface adapter as described in m_axi bundles. Bundling ports into a single interface helps save device resources by eliminating AXI4 logic. However, a single bundle can limit the performance of the IP because all the memory transfers have to go through a single interface which can read and write simultaneously, though only at one location. Therefore, using multiple bundles lets you increase performance by creating multiple interfaces to connect to memory banks.

2 Experimental Setup

In this section, we will be referring to the FPGA platform used to build and test the proposed accelerator design on. Additionally, a short reference to the development tools utilized during the process, including the Vivado Design Suite 2021.1, the Vitis Unified Software Platform 2021.1 and OpenCL, will be made.

2.1 The AMD Alveo U200 Data Center Accelerator Card

The AMD Alveo U200 Data Center Accelerator Card [28] was employed to validate the results and assess the application's performance. Designed by Xilinx/AMD, this card targets high-performance computing (HPC), machine learning (ML), and data-intensive processing tasks. It utilizes FPGA (field-programmable gate array) technology, enabling developers to tailor hardware functionality for specific operations. The Alveo U200 interfaces with a CPU through a PCIe Gen3 x16 connection, offering flexibility across multiple use cases. It excels in scenarios such as low-latency inference for deep neural networks (DNNs), genomic sequence alignment, and advanced image and video processing. With its ability to support extensive parallel processing, the Alveo U200 provides a customizable and efficient solution for demanding computational workloads.

The device includes three Super Logic Regions (SLRs). According to [29], an SLR is a single die segment within an FPGA that uses SSI (Stacked Silicon Interconnect) technology. Each SLR contains a dedicated set of resources—such as Configurable Logic Blocks (CLBs), Block RAM, and Digital Signal Processing (DSP) units—as summarized in Table 4.1, and follows a layout similar to that of traditional, non-SSI devices.

Developed by Xilinx, SSI technology is a sophisticated packaging method that combines multiple silicon dies into a single FPGA chip [30]. This modular architecture addresses the scalability challenges of conventional monolithic FPGAs by linking smaller die segments (SLRs) through high-speed interconnects.

Communication between SLRs in SSI-based FPGAs is facilitated by Super Long Lines (SLLs), which span the silicon interposer to provide high-bandwidth, low-latency links across the different regions. These SLLs significantly improve data flow between dies, boosting the scalability and performance of SSI-integrated FPGAs. However, because SLLs must cross the interposer and connect multiple SLRs, they introduce longer signal paths, which can increase latency compared to within-die routing. This extended communication, along with added routing complexity, may also result in higher power usage—potentially impacting the efficiency of designs that rely heavily on inter-SLR data exchange.

U200 Resources	**SLR 0**	**SLR 1**	**SLR 2**
LUTs	388K	205K	385K
Registers	776K	410K	770K
Block RAM tiles	720	420	720
UltraRAMs	320	160	320
DSPs	2280	1320	2280

Table 4.1: Alveo U200 Resources per SLR

2.2 Vivado Design Suite 2021.1 and Vitis Unified Software Platform 2021.1

Vivado is Xilinx's comprehensive design and verification platform tailored for developing hardware on Xilinx FPGAs and SoCs (System-on-Chip). It provides a rich set of tools for designing, simulating, and verifying digital circuits, supporting both hardware description languages like Verilog and VHDL and schematic-based approaches through an intuitive graphical interface.

Key features of Vivado include advanced logic synthesis—translating high-level code into gate-level structures—and implementation, which maps the logic onto the FPGA's physical resources such as lookup tables (LUTs) and flip-flops. The tool also features a powerful place-and-route engine that arranges these elements within the FPGA to optimize performance and minimize latency.

Additionally, Vivado generates detailed reports on resource usage and performance, helping designers identify inefficiencies and optimize their systems prior to deployment. Overall, it streamlines the FPGA development workflow by combining design, analysis, and optimization in a single environment.

2.3 Vitis Unified Software Platform 2021.1 and Vitis HLS 2021.1

Vitis is Xilinx's unified development platform designed to support both hardware and software engineers in building applications for Xilinx devices, including FPGAs, SoCs, and Versal ACAPs. It caters to a wide range of use cases, from embedded systems to high-performance computing and hardware acceleration, and is compatible with both edge devices and data center hardware. By abstracting complex hardware details, Vitis simplifies application development on FPGAs and adaptive platforms, promoting a streamlined hardware-software co-design process that integrates smoothly with Vivado.

A central feature of Vitis is Vitis HLS, which enables developers—particularly those with a software background—to create FPGA accelerators using familiar programming languages like C, C++, or OpenCL. This eliminates the need to write low-level hardware description languages (HDLs) such as Verilog or VHDL. Vitis HLS converts high-level code into RTL (Register-Transfer Level) that can be further processed using Vivado, significantly lowering the entry barrier for FPGA development.

In version 2021.1, Vitis HLS includes an enhanced compiler capable of translating C/C++ code into Verilog or VHDL while supporting directive-based optimizations. Through the use of pragmas and directives, developers can fine-tune performance aspects like pipelining, loop unrolling, and memory access, all without manual HDL coding. This offers a flexible and accessible approach to optimizing FPGA implementations, balancing ease of use with control over hardware performance. The steps performed by Vitis HLS to convert a high level specification into a fully timed implementation by the open-source HLS compiler [31] agree with the workflow of Figure 4.1.

2.4 OpenCL via Xilinx Runtime (XRT)

Effective communication between a host system and an FPGA through PCIe requires a specialized software framework. OpenCL fulfills this role by enabling program development for diverse computing platforms, including CPUs, GPUs, FPGAs, and other processors. Its hardware-agnostic design allows developers to write code once and deploy it across different device types with little modification. In OpenCL terminology, a kernel represents the computational code that runs on the target device, such as an FPGA. Although OpenCL offers a conceptual framework for communicating with mixed computing environments—specifically between the host CPU and FPGA—actual hardware communication requires specialized driver software. Xilinx addressed this requirement by creating the Xilinx Runtime (XRT), a middleware solution that handles the interface between host applications and FPGA devices. XRT functions as an intermediary layer that bridges hardware and software components, streamlining the development process and enabling effective FPGA-based acceleration across diverse applications, including OpenCL-based projects. Through XRT, developers can deploy, configure, and run OpenCL kernels on FPGAs while the runtime handles resource management tasks like memory buffer allocation and kernel instantiation. Since XRT encompasses broader functionality beyond standard OpenCL capabilities, it also supports direct OpenCL function calls—a feature leveraged in the current system design. This architectural decision promotes portability, allowing the implementation to be adapted for different FPGA platforms and facilitating testing and verification across various FPGA solutions. This cross-platform compatibility makes XRT a valuable development tool for optimizing applications across multiple FPGA architectures.

Chapter 5

Understanding and Examining the Characteristics of Genome Graphs and Read Datasets

In order to better understand the nature, structure, and size of the reference genome graph data as well as the read datasets quantitative and qualitative characteristics a profiling analysis needs to be performed. This process is also essential for choosing the values of certain parameters and design choices that play a crucial role in the proposed micro architecture explained in detail in the following chapters. Furthermore, it is important to check how final alignments of the read dataset, as derived from the state-of-the-art software sequence-to-graph alignment tool, are distributed in the graph.

1 Reference Genome Graphs' Characteristics

1.1 Profiling of the Entire Reference Graph Dataset

In this section, a detailed analysis of the different characteristics of the reference genome graphs will be presented. The reference genome graphs used for the scope of this work are produced using the latest major release of the human genome assembly, GRCh38, as a starting reference genome, and 7 additional Variant Calling Files (VCFs) for HG001-007 from the GIABproject (v3.3.2). Using those data, 24 reference graph are produced, one for each of the 22 autosomal chromosome and two for sex chromosomes X and Y. Firstly, each node can be seen as a struct with certain characterizing elements including, its sequence, the nodeIDs of its parental nodes (inNodes) the number of parental nodes (num_inNodes), the nodeIDs of its children nodes (outNodes) and the number of those (num_outNodes). An example of how values are attributed to these characteristics is visually shown in Figure 5.1.

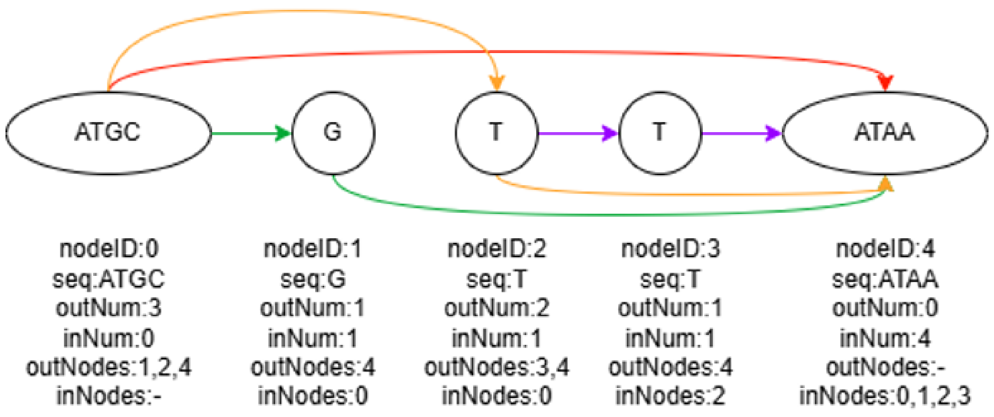


Figure 5.1: Reference Graph Representation

In general, since the size of chromosomes decrease as their numbering increases (chromosome X is an exception), the respective reference graphs' number of nodes decreases accordingly as seen in Figure 5.2. In some cases though the decreasing tendency of the number of nodes is not followed which can be interpreted by the fact that some chromosome might have less variants than other, therefore less divergence from the original linear reference and thus less nodes.

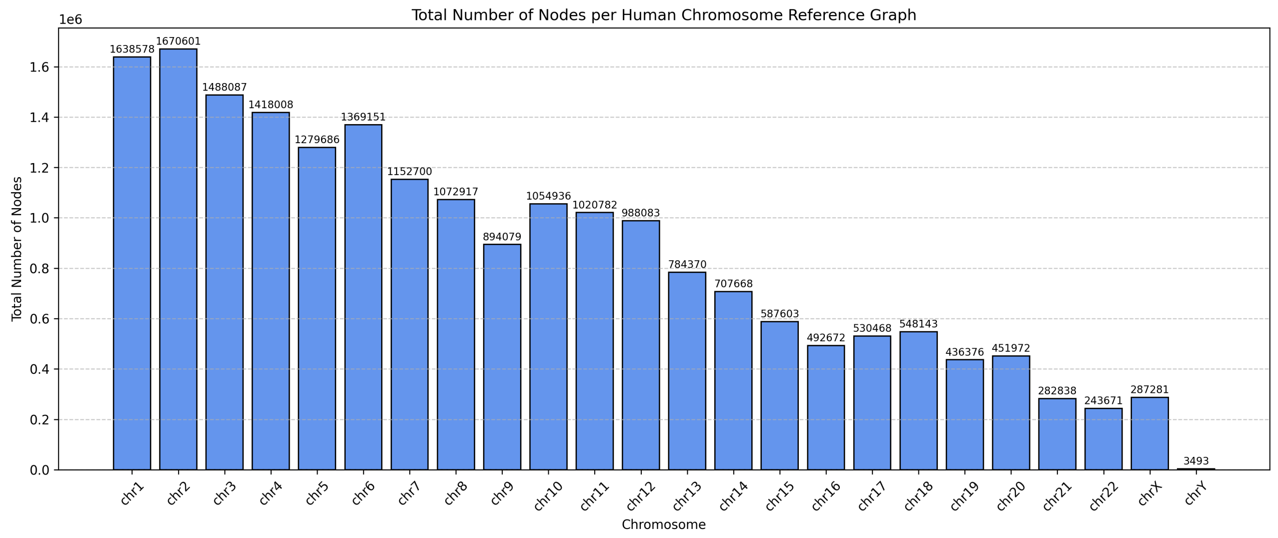


Figure 5.2: Number of Nodes of the Human Genome Reference Graphs

A second characteristic that is examined is the total sequence length of the graph which is defined as the length of the concatenated sequence of the sorted nodes of the graph. In Figure 5.3, one can see better the decreasing sizes of the chromosomes as reflected by the decreasing total sequence length, since the incorporated variations are not expected to alter much the relationship in between them as most of the introduced variants are single nucleotide variants (SNVs) or small deletions/insertions.

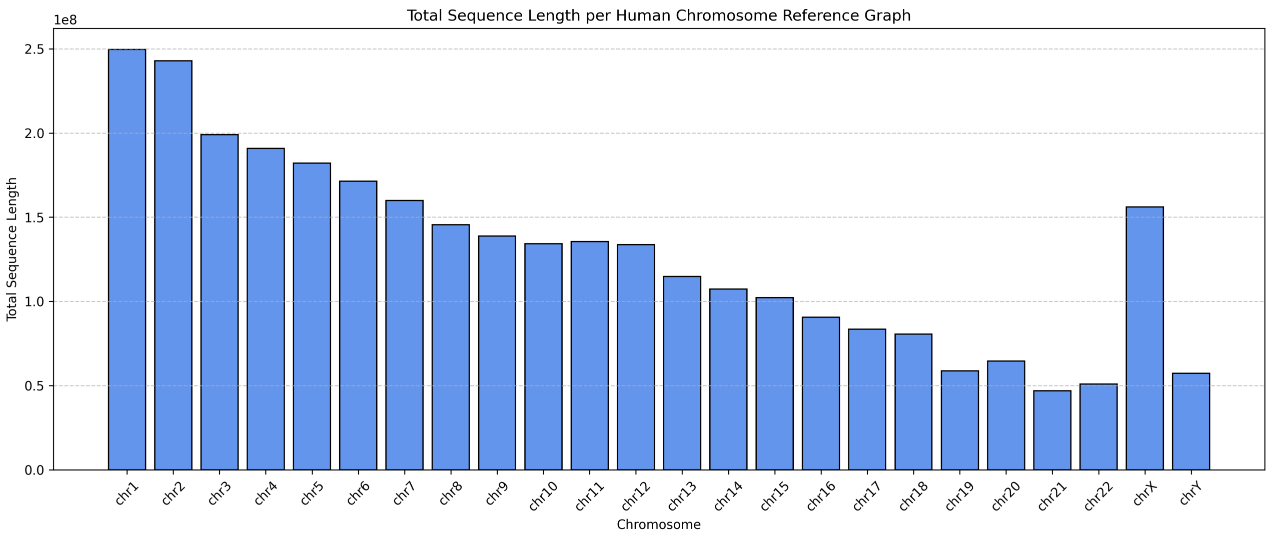


Figure 5.3: Total Sequence Length of the Human Genome Reference Graphs

Additionally to examine the complexity of the graphs, the total number of edges within the graphs is measured and presented in the histograms of Figure 5.4. It is also interesting to notice that chromosome Y is a linear graph, meaning that all nodes are just connected to their immediate neighbor and no other.

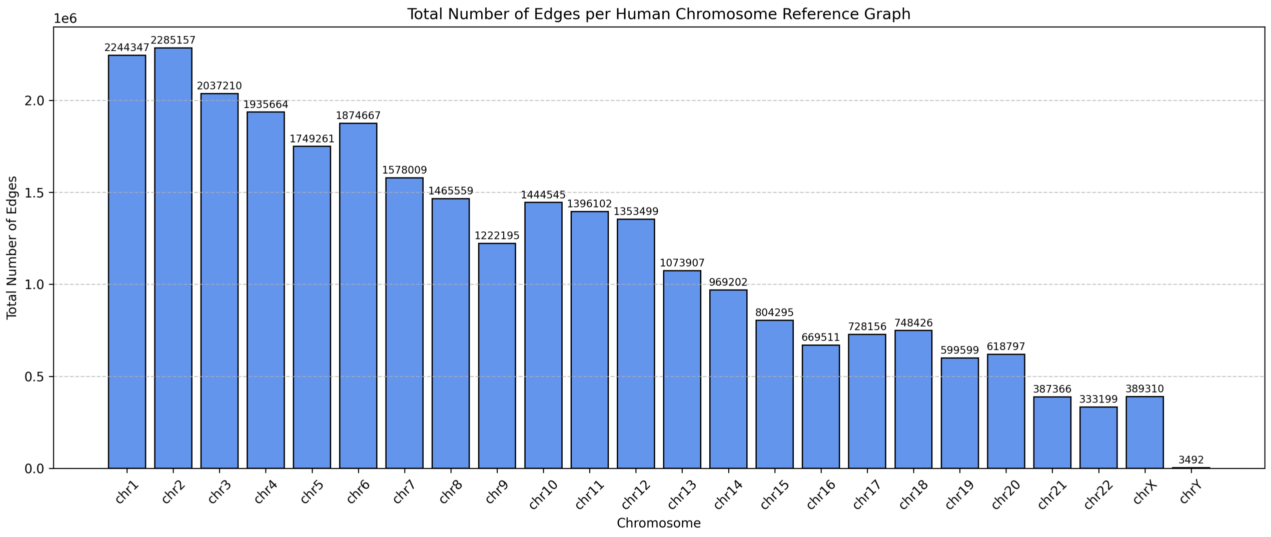


Figure 5.4: Total Number of Edges of the Human Genome Reference Graphs

1.2 Further Examination of a Sample Graph (Human Chromosome 22 Graph)

To simplify the next crucial steps of the pre-processing profiling analysis, only its results concerning chromosome 22 will be presented. To better examine the sequence lengths of the individual nodes, the following diagram of Figure 5.5 shows their distribution. As expected, since most variants are single nucleotide ones, most nodes include only one character. However there are still many nodes with long sequences of length higher than 1000 or even 16000 bases in unknown locations, meaning that for example not all 16384 bases long sequences are found in the first or last nodes grouped together. Therefore, during designing one has to be careful when retrieving sequence data concerning an entire subgraph since most likely some of this data is not useful to perform the alignment step. For example, when wanting to align a short read of 100 bases to a subgraph that includes a 1000 base long sequence, it is certain that the entire sequence of the node will not be used and thus not needed to be retrieved in its entirety, or perhaps not at all.

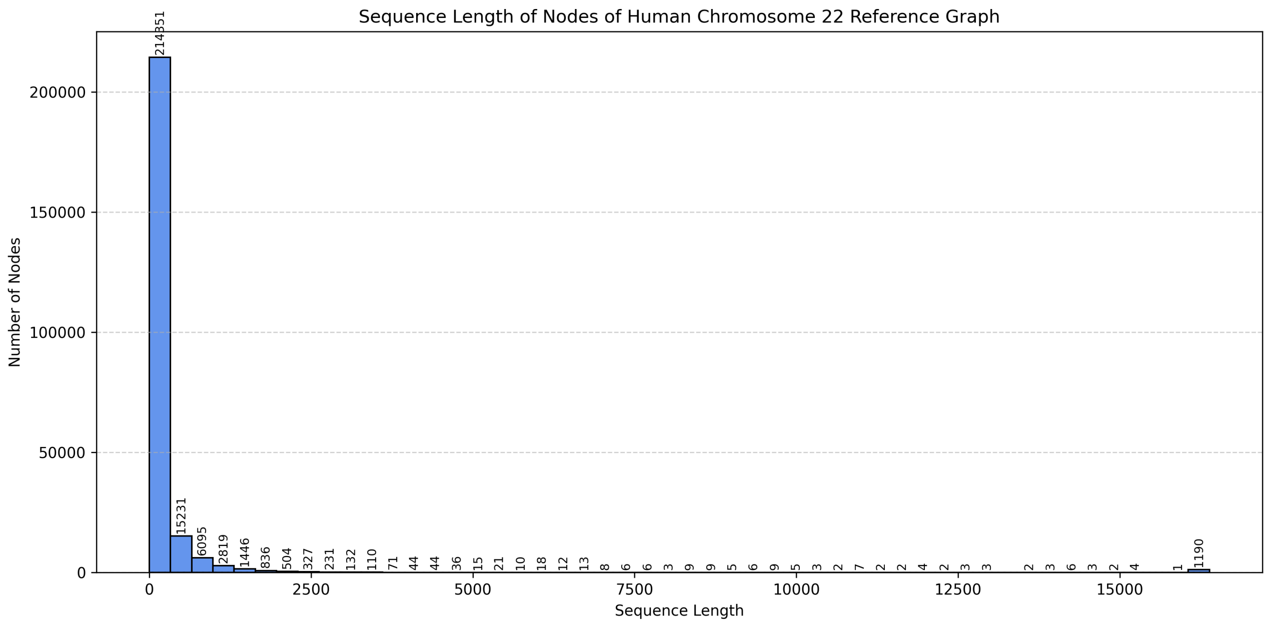


Figure 5.5: Nodes' Sequence Lengths' Distribution for Chromosome 22 Reference Graph

The chosen representation of the graph as presented in detail in Section **??** requires previous measurement of the number of parents and children to determine the size of the corresponding tables. These numbers can be found through num_inNodes and num_outNodes profiling respectively as shown in Figures 5.6 and 5.7. To achieve higher accuracy, trading off utilization, the arrays' size will be determined by the maximum values of those parameters. Since the maximum values characterize only one node (as it is the case in most chromosomal graphs), one could decide to decrease the size of the arrays

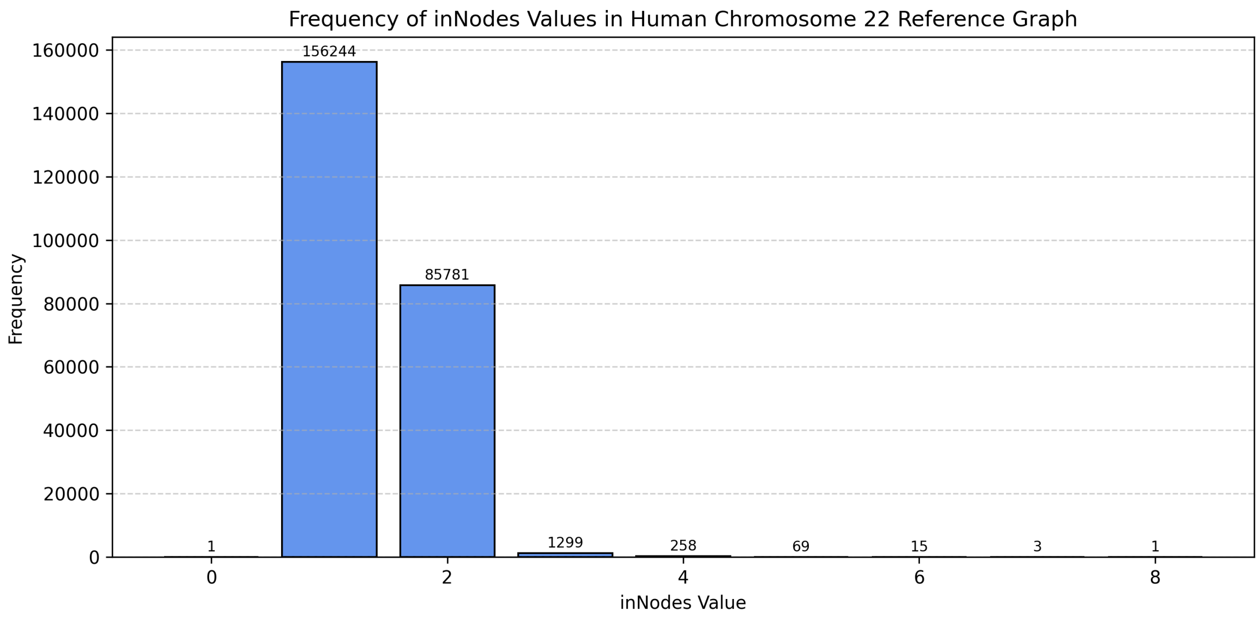


Figure 5.6: Number of in_Nodes Distribution for Chromosome 22 Reference Graph

accordingly decreasing accuracy by neglecting possible alignment subgraphs. Additionally, due to the SNVs most of the nodes only have one inNode or outNode.

The final reference graph characteristic needed to be examined is the maximum hop value. A hop equal to one is defined in this work as an edge starting from an node and ending in an intermediate neighbor. Figure 5.8 demonstrates a small example of how a hope in measured. Figure 5.9 shows the distribution of hop values for human chromosome 22.

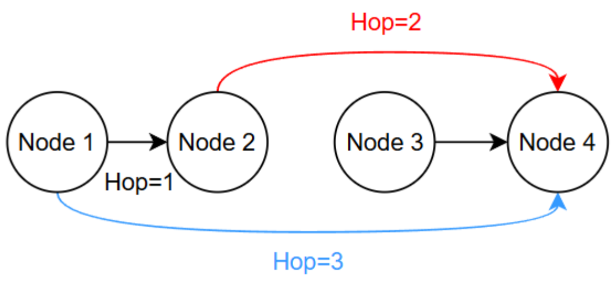


Figure 5.8: Hop Definition and Measurement Example

As expected most of the hops have small values due to the nature of the variants existing in the graph (SNVs and small InDels). To increase accuracy in this design a hop limit is not introduced however as SeGraM has proposed, the incorporation of one does not affect accuracy greatly, however it could if the accelerator was to be used in other types of alignment tasks such as aligning reads to a multi-species reference graph were larger insertions and deletions are for sure present.

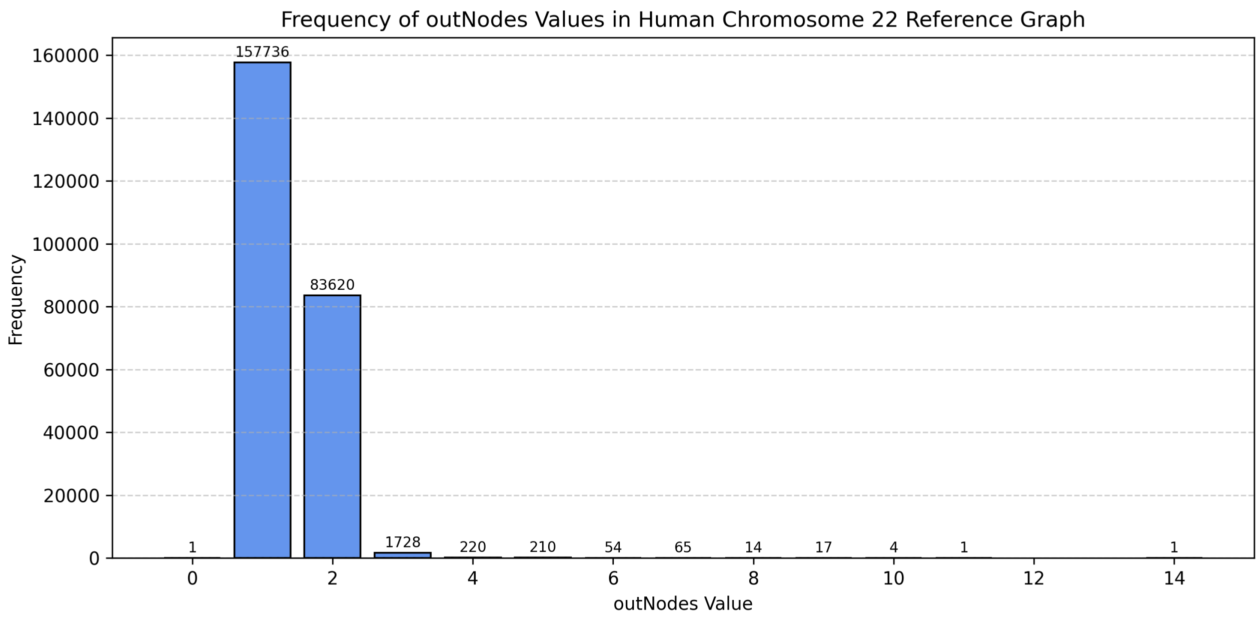


Figure 5.7: Number of out_Nodes Distribution for Chromosome 22 Reference Graph

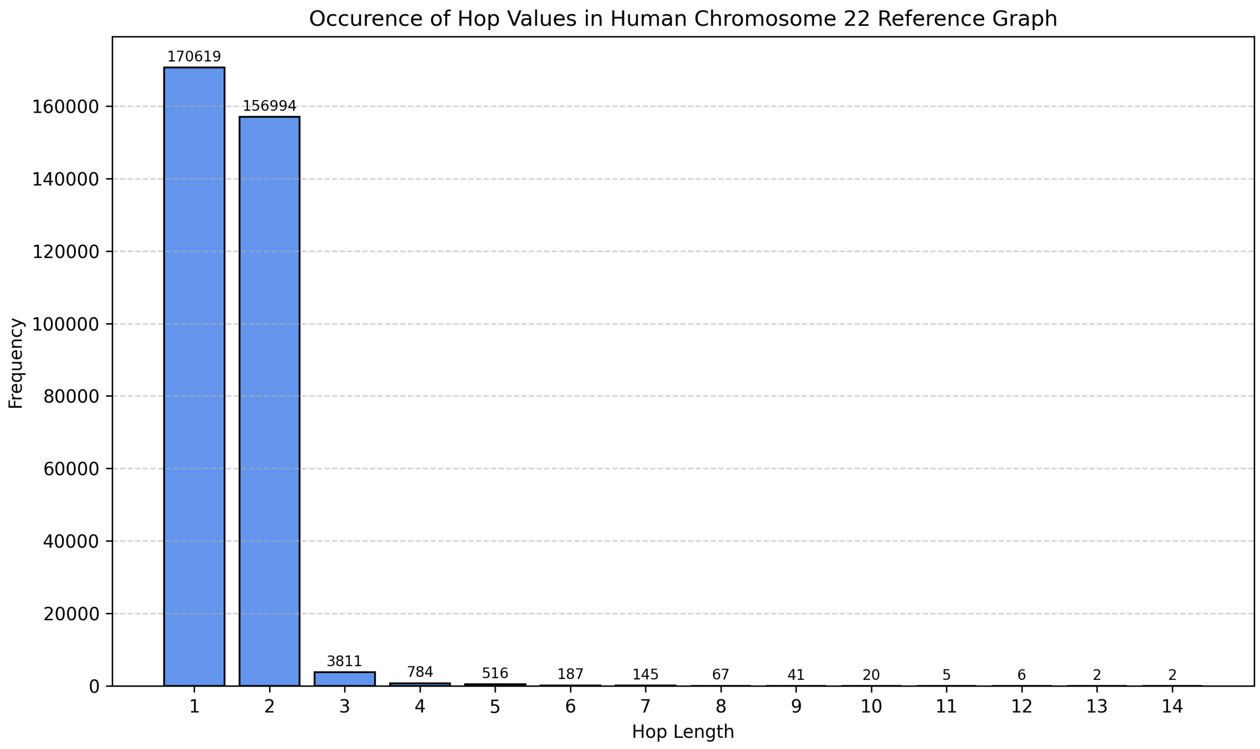


Figure 5.9: Distribution of Hop Values in Chromosome 22

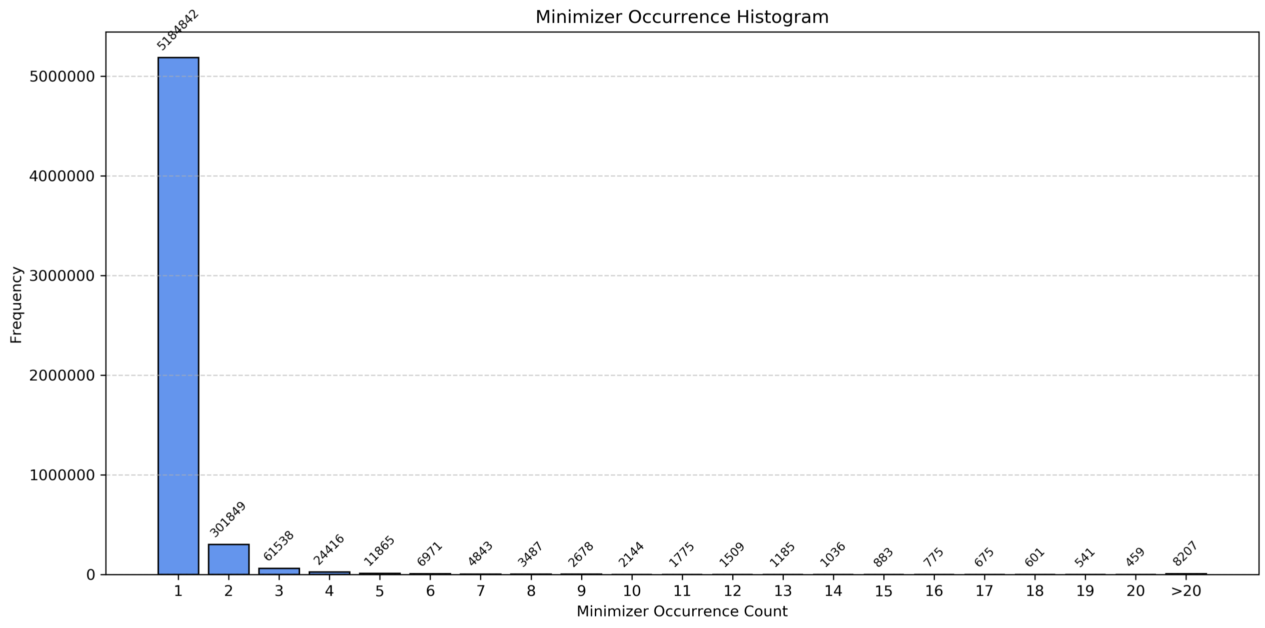


Figure 5.10: Distribution of Minimizers' with Low Occurrences

1.3 Reference Graph Minimizers' Characteristics

Additional to the graph characteristics' information, it is equally important to count the number of extracted minimizers. Figure 5.10 shows the distribution of minimizers occuring less than 20 times in the reference. Most of them are unique, meaning that they occur only one time, however there are some minimizers that are exist in more than 5000 positions in the reference graph as shown in Figure **??**.

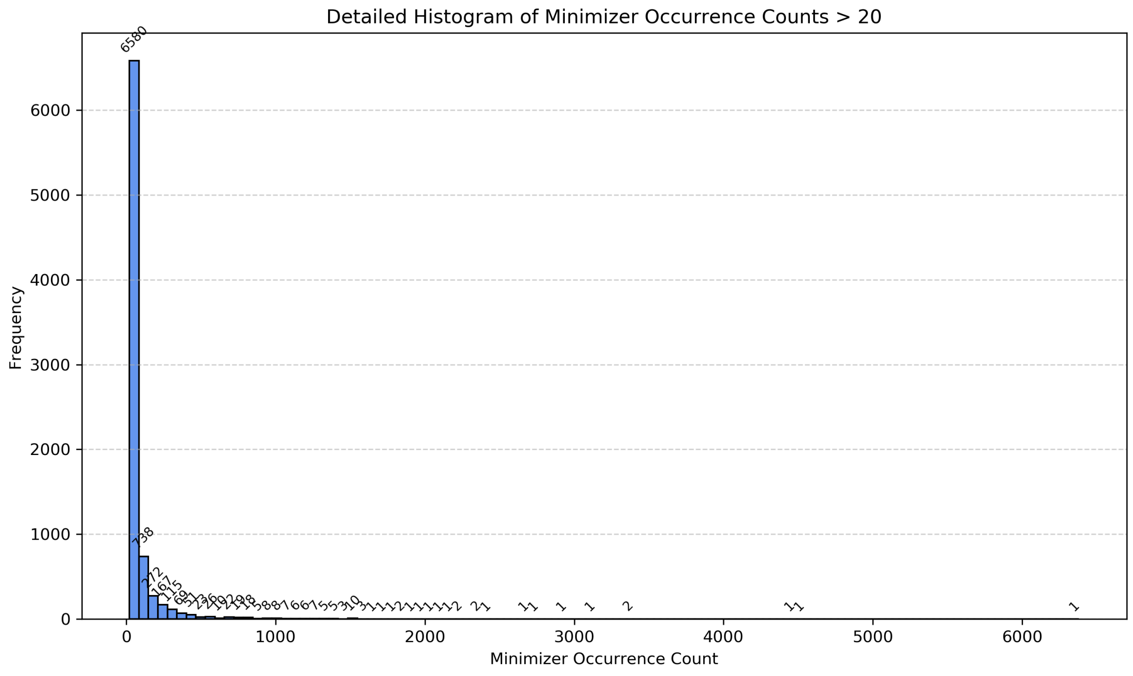


Figure 5.11: Distribution of Minimizers' with High Occurrences

The existence of minimizers found in so many locations introduces the need to filter out most common reads as done in [8] or introduce a seed limit that can be adjusted accordingly, as done in this design. Decreasing the seed limit inevitably will directly lower accuracy, since many possible alignment locations will be discarded during the seeding process.

Finally, Figure 5.12 shows the occurrences of minimizers starting with each letter of the DNA alphabet. As expected due to the way minimizers are extracted, always picking the lexicographically smaller k-mer within a window, most minimizers start with the letter A. The lack of a uniform distribution rises issues in handling the minimizers during the search step of seeding introducing the need to break the reference minimizers into more groups to perform more balanced parallel search.

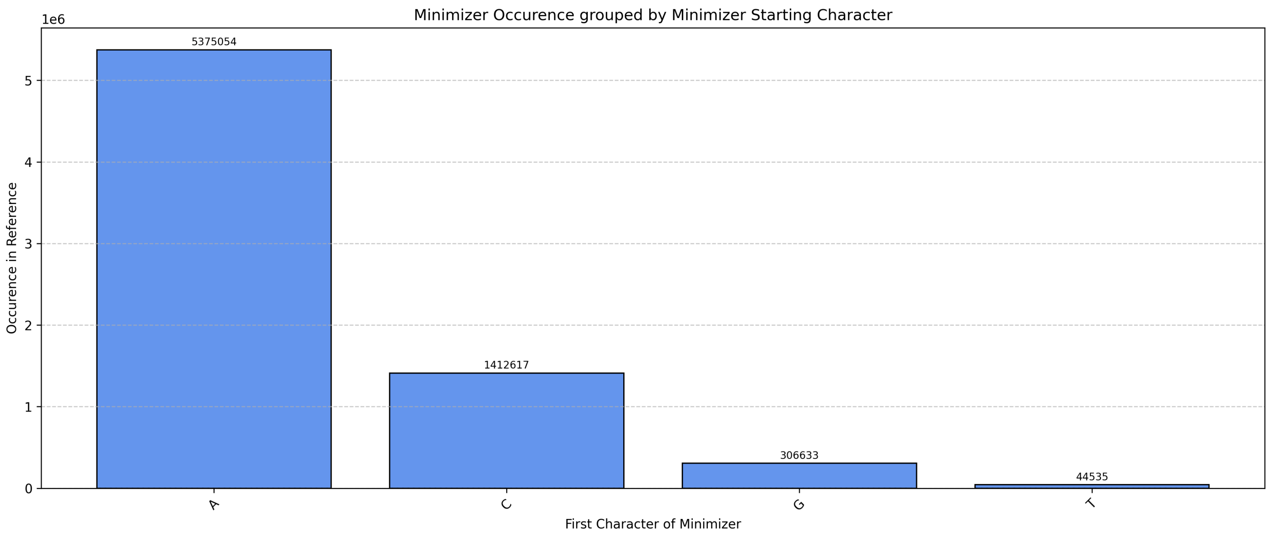


Figure 5.12: Distribution of Minimizers' Occurrences grouped by First Character

2 Read Dataset Profiling

Hop limit imposition is a possible and logical design choice that will improve utilization and efficiency decreasing the system's accuracy if it is set to a very low value. To examine the actual effect on accuracy the incorporation of the read dataset profiling is needed to see the values of hops actually existing in the final alignments. To do so, GraphAligner was ran to align to the above reference graphs two sets of reads: PacBio 10kbp reads with 10% error rate and Illumina short read dataset including 100bp, 150bp and 250bp reads. Each of the five read groups has 10000 reads.

The existing hop values in the final alignments increase with the length of the read as expected since longer reads naturally require bigger subgraphs to be aligned to. However, even in those cases it is evident that only a small number of reads is aligned to such big subgraphs. In the case of short reads, one can

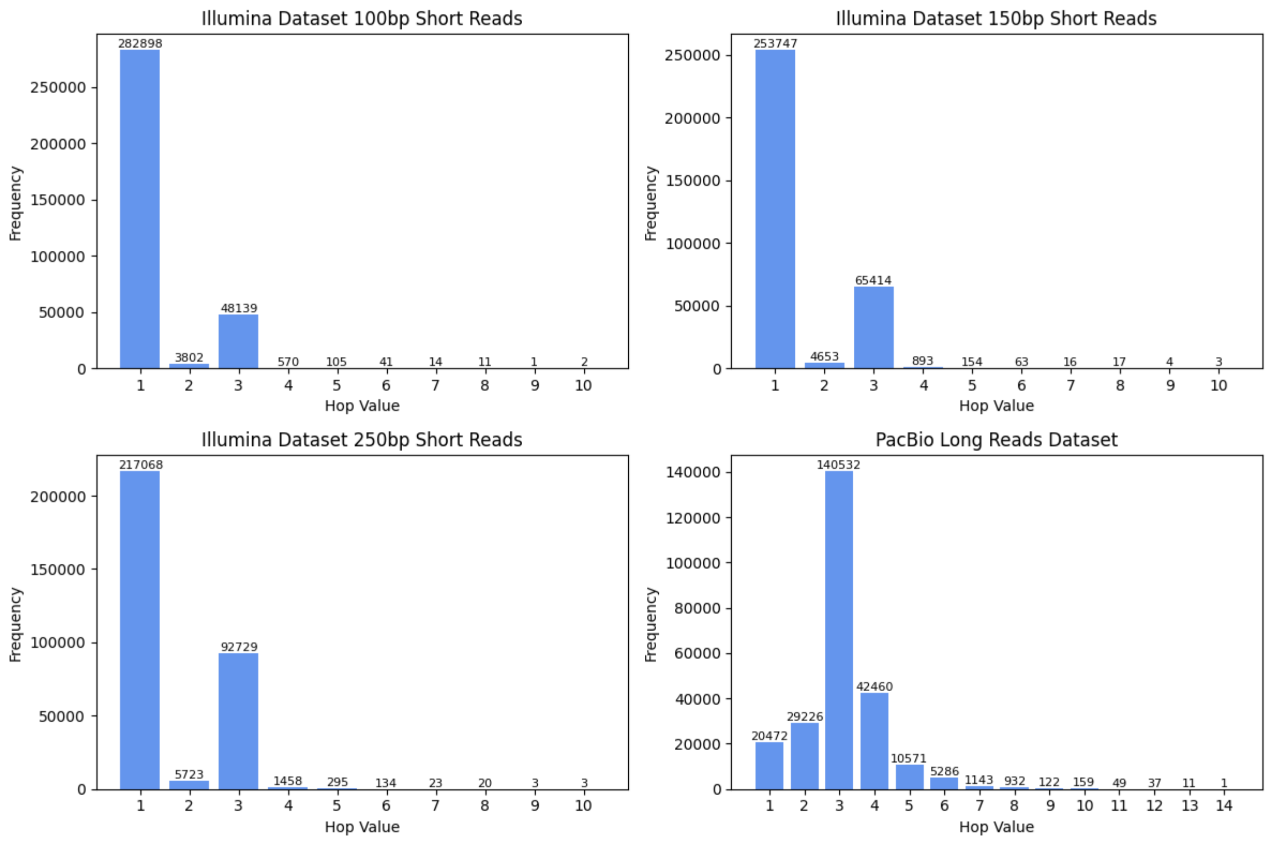


Figure 5.13: Hop Values in Final Alignments of Different Short and Long Read Dataset

see that the introduction of a hop limit will not affect accuracy, since the actual existing hop within the alignment is significantly lower than the maximum hop existing in the graph and will most likely be never found within an alignment.

Chapter 6

Proposed Workflow, Algorithm and Accelerator Design

1 Workflow and Algorithms

1.1 Reference Graph Building and Topological Sorting

To start with, it is essential to understand how the reference graph is constructed using multiple linear genomes. Figure 6.1 shows an example of genome graph construction using four small sequences.

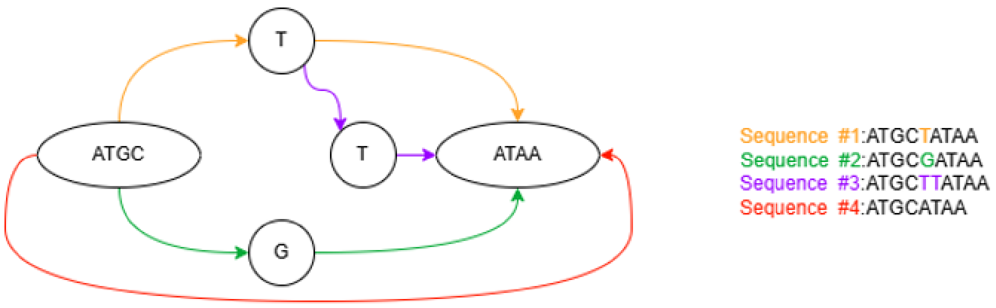


Figure 6.1: Building a Reference Graph out of Linear Reference Genomes

In order to proceed with the mapping and alignment steps it is essential to ensure that the graph is topologically sorted meaning that the directed genome graph has its ventricles linearly ordered such as for every directed edge (u,v) starting from vertex u to vertex v, u comes before in ordering. Without topological sorting, algorithms might attempt to process a node before processing its dependencies, leading to incorrect results. Basically, topological sorting works for directed acyclic graphs using linear time algorithms, such as Introsort which is implemented by the vg toolbox to perform graph topological sorting. The vg ids –s command was used to create the graph from linear genomes. This algorithm having a worst-case runtime of O(nlogn) basically traverses the graph in depth and sorts the nodes so as for each node to have all of the nodes with smaller nodeID values before and all with greater after.

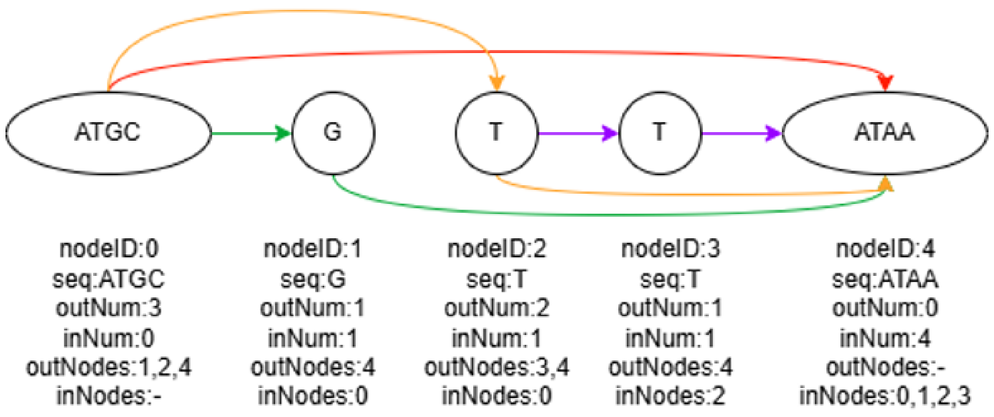
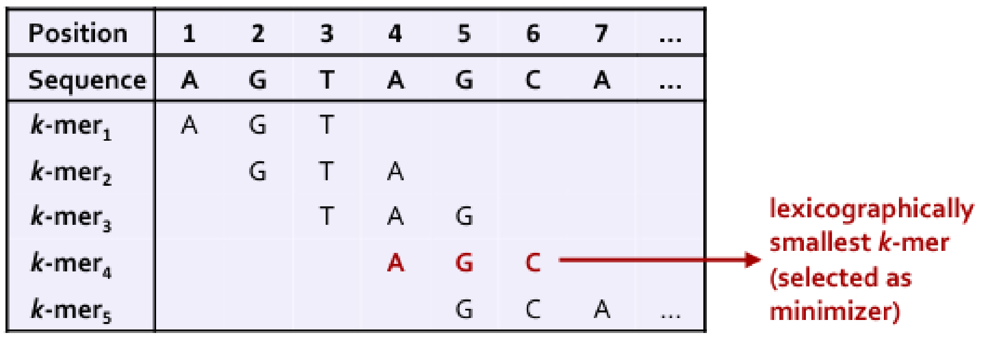


Figure 6.2: Reference Graph Representation

1.2 Minimizer Extraction and Reference Indexing

After having constructed the graph the next step that is performed offline once for each reference graph is indexing. The reference graph is organized as a structure of nodes. Each node is characterized by an integer nodeID, the number of out_Nodes, which basically are the nodes connected to the node via an edge starting from the node, a list of the out_Nodes' nodeIDs, the number of in_Nodes, which basically are the nodes connected to the node via an edge ending at the node, a list of the in_Nodes' nodeIDs and the string of the DNA sequence. The graph of Figure 6.1 will look like the graph in Figure 6.2, sorting and structure creation:

Several different arrays of information are used to adequately and efficiently store the information that describes the graph and is needed for the seeding and alignment steps. Initially, it is evident that for the alignment step an array including the nodeIDs and their respective sequences is needed for the alignment step. Additionally, the information regarding neighboring nodes and connections in between them is also needed for both the seeding and alignment steps. For that reason, four types of arrays are introduced, two of them concerning the out_Nodes information and two respectively for the in_Nodes information. One array includes in its i-th row the nodeIDs of the greater neighbors (out_Nodes) of the node with nodeID equal to i and the other array the nodeIDs of the smaller neighbors of that node (in_Nodes). The two other arrays include the sequence lengths of the in_Nodes and out_Nodes of the i-th node respectively. Additionally, other characteristics of the graph have been profiled during this pre-processing step, later used in the implementation part, such as the maximum hop, the maximum number of inNodes and outNodes, sequences' lengths etc. Furthermore, it is essential to index the reference so that during the seeding step the heuristic algorithm is able to locate possible candidate alignment regions without searching the entire reference genome. For that



Position	1	2	3	4	5	6	7	...
Sequence	A	G	T	A	G	C	A	...
k-mer$_1$	A	G	T					
k-mer$_2$		G	T	A				
k-mer$_3$			T	A	G			
k-mer$_4$				A	G	C		
k-mer$_5$					G	C	A	...

lexicographically smallest *k*-mer (selected as minimizer)

Figure 6.3: Minimizer Extraction Example

purpose, minimizers are extracted from the reference genome graph sequences using the lexicographically smaller k-mer as the minimizer of a window. The example of Figure 6.3 shows how the minimizer is extracted from a window of size 5 and k-mer length equal to 3. After having extracted the minimizer from the current window the window slides so as to not have overlapping windows.

For each minimizer extracted from the nodes that have a sequence length greater than the length of the window its position needs to be stored, meaning that both the nodeID and positional offset in the node are kept. Instead of filtering the most common minimizers of the reference out as work [8] suggests, we later set a seed limit that can be adjusted by the user. Also, in order to have the minimizer index better organized and later reduce the seed searching latency and accesses to memory, the structure is broken down into batches, so as to have all minimizers starting with the same two characters in the same minimizer's reference batch. Table 6.1 summarizes the arrays produced for the reference graph once during the pre-processing step. Additionally, to perform the alignment task after the seeding procedure, the sequences found within the nodes of the graph are needed. Since nodes have variable sequence lengths, it is best to store the concatenated sequence of the entire graph within a character table. To be able to navigate within this array, the cumulative sequence lengths of the nodes are needed so as find the starting and ending positions of a nodes sequence within the concatenated string. The cumulative_lengths array includes in its i-th position the sum of all the sequences' lengths of the nodes with nodeIDs smaller than the nodeID of the i-th node. Thus, if the sequence of the i-th node needs to be retrieved from memory, the starting and ending position of the sequences array will be the cumulative lengths (i-1)-th position value and the i-th value respectively.

All the above steps are referred to as the pre-processing steps which happen only once for each reference graph and are loaded on the main memory.

Table 6.1: Arrays used to depict Reference Graph Related Information

Data Structure	**Description**
`sequences[sum_of_lengths_of_all_nodes]`	Includes the concatenated sequence of all nodes after sorting
`cumulative_lengths[i]`	Includes the cumulative sum of lengths up till the i-th node
`array_inNodes_seq_len[i][j]`	Includes the sequence lengths of the j-th inNode of the i-th node.
`array_inNodes_ids[i][j]`	Includes the nodeID of the j-th inNode of the i-th node.
`array_outNodes_seq_len[i][j]`	Includes the sequence lengths of the j-th outNode of the i-th node.
`array_outNodes_ids[i][j]`	Includes the nodeID of the j-th outNode of the i-th node.
`minimizers[i]`	Includes the minimizers' sequences of the i-th minimizer.
`minimizers_positions[i][2]`	Includes the nodeID and positional offset of the i-th minimizer.

1.3 Seeding

After having processed the reference genome, the seeding process is initialized by extracting the minimizers from the read sequence using the same algorithm as the one used to extract the minimizers from the reference genome. Therefore, the lexicographically smaller k-mer within a window will be a minimizer. The number of minimizers extracted from the entirety of the read when the sliding windows are not overlapping can be priorly calculated. To find the formula that provides us with the number of minimizers of a read sequence with known length we will use a recursive approach. Suppose the read length is equal to N while the k-mer length m and the window size w. The window each time moves with a step size of w-m which enables us to pick a smaller number of minimizers rather than having a step size of one that would increase redundancy. As one can see from the example of Figure 6.3 each window includes w k-mers, while the total number of k-mers in the read is evidently equal to $K = N - m + 1$, since each character initializes only one k-mer apart from the m-1 last characters which are left out since there are not enough characters to have k-mers starting from their position. Since the window slides with a step of w-k each time and because each window produces one minimizer, the final formula that provides the total number of minimizers is the following: $K = \frac{N-m+1}{w-m}$

After the minimizers are extracted, they are kept together with their starting offset meaning their position in the read. The next step is to search for them in the reference index. To reduce the search time and memory accesses the minimizer search is performed within the relative bucket depending on the first two characters of the minimizer. The minimizer is searched within the

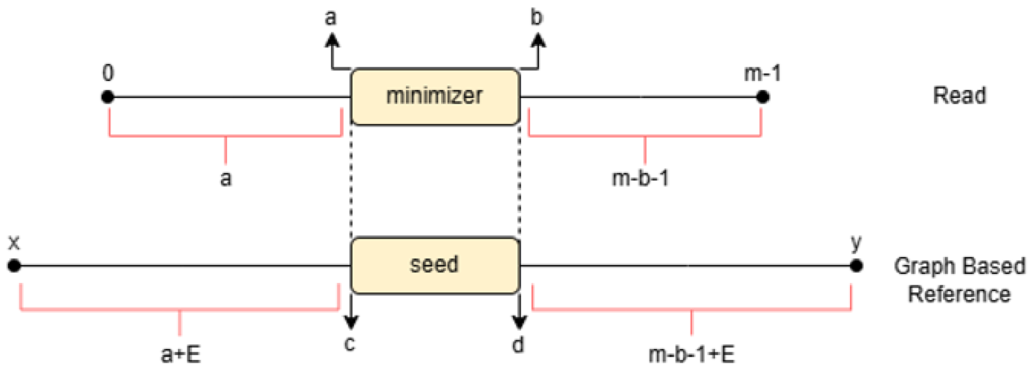


Figure 6.4: Relative Offset Calculation

respective bucket and if found its position within the reference graph (nodeID and starting offset) is retrieved and the seed hit is successful. Therefore, each seed (minimizer that has successfully been located in the index) is characterized by three location integers, its starting position in the read, its starting position within the node sequence and the nodeID of the node within which it is found. Since in most cases there are more than one seed hits for a minimizer (a seed is found in many positions in the graph) we set a seed hit limit adjustable by the user to determine the maximum number of locations kept for further investigation. After some seed hits are successfully found, the next step is to calculate the relative starting and ending offsets of the read in the node where the seed hit was located. As shown in Figure 6.4, the minimizer found in the read query of length m, has a seed hit in the reference which is characterized by the starting offset within the node c and its ending offset d, which is basically equal to $c + minimizer\_length - 1$. The next step is to find the region of the node (or graph) that will be used in the alignment step and will include characters sufficient to align the entire read. In order to do that the x and y positions must be calculated, which as it can be derived from Figure 6.4, are equal to $c - a - E$ and $d + m - b - 1 + E$, where E is the edit threshold. From now on, the x value will be referred to as the relative starting offset and y as the relative ending offset.

1.4 Subgraph Extraction and Extention

After having found the relative starting and ending offsets of the seed hits the next step is to find the candidate alignment subgraphs each one of which includes the node of the seed hit. This step is crucial since it provides a way of finding the actual rightmost and leftmost limits of the subgraph candidate for alignment with the entire read. To understand the four different cases that this so-called subgraph extension algorithm deals with the following cases are presented. The first case, which is the simplest one, includes the cases where the entire read fits to be aligned within the node of the seed hit. In this case the already found offset and characterizing nodeIDs can be used as the limits of the candidate alignment subgraph itself. Assume E=0, for all cases to make things more straightforward. As shown in the example of Figure 6.5, the relative starting offsets of the seed hits are positive meaning that to the left of the starting position of the minimizer in the starting node there are enough characters within the same node to align the remaining sequence to the left of the starting position of the respective minimizer in the read. In fact the offset from which the alignment step will start looking for alignments is equal to the relative starting offset of the group, which as one can see is indeed the position that the reference sequence to be aligned to read starts. The same goes for the ending characters with the node having enough characters the left of the seed to align the entire read, since the relative ending offset is smaller than the sequence length of the node.

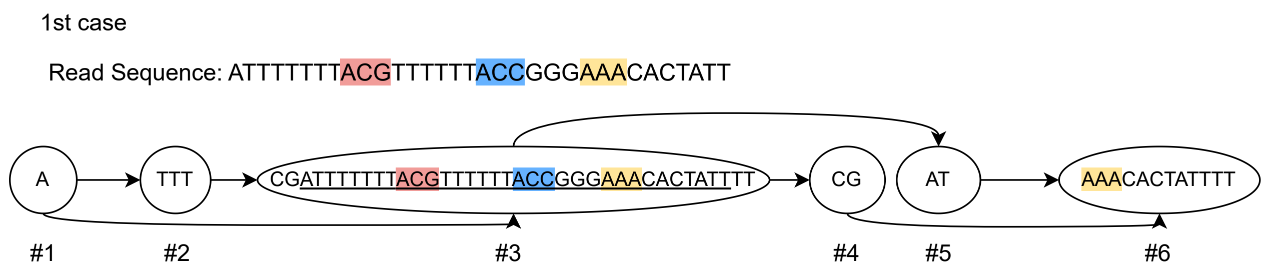

1st case
Read Sequence: ATTTTTTTACGTTTTTTACCGGGAAACACTATT
A
TTT
CGATTTTTTTACGTTTTTTACCGGGAAACACTATTTT
CG
AT
AAACACTATTTT
#1
#2
#3
#4
#5
#6

Figure 6.5: The node of the seed hit has enough characters to align the entire read

Seed hit	Start offset in read	NodeID	Start offset in ref	Relative start offset	Relative end offset	Sequence length in end node
ACG	8	3	10	2	37	39
ACC	17	3	19	2	37	39
AAA	23	3	25	2	37	39

Table 6.2: Example array of seed hits and their alignment metrics

More specifically, the relative ending offset of the group is lower than the end node sequence length, meaning that there are enough characters in the ending node to align the characters remaining on the right side of the last minimizer in the read. Specifically, to align the read, there are more than end_node_sequence_length-relative_ending_offset characters needed, in the case below three more characters are needed. Overall, this part of the algorithm ensures that the candidate alignment subgraph has more or equal number of characters to align the given read, since if this is not the case as the next examples will illustrate, there is a need to extend the subgraph to the left or to the right or to both directions to have enough characters to align the read.

The second case is that of subgraph extension needed to the left of the starting node of the subgraph, when the relative starting offset of the seed hit is negative, meaning that there are not enough characters to the left of the seed to align the remaining characters to the left of the respective minimizer. The absolute value of the relative starting offset of the groups is equal to the number of extra characters needed to the left of the first seed to have equal number of characters with the remaining characters of the respective minimizer in the read. As one can see from the example below, the starting subsequence of the read before the first minimizer is found in node #3 in position $|(relative\_starting\_offset)| - 1$, since the absolute value actually represents a sequence length and not the offset which is smaller by one.

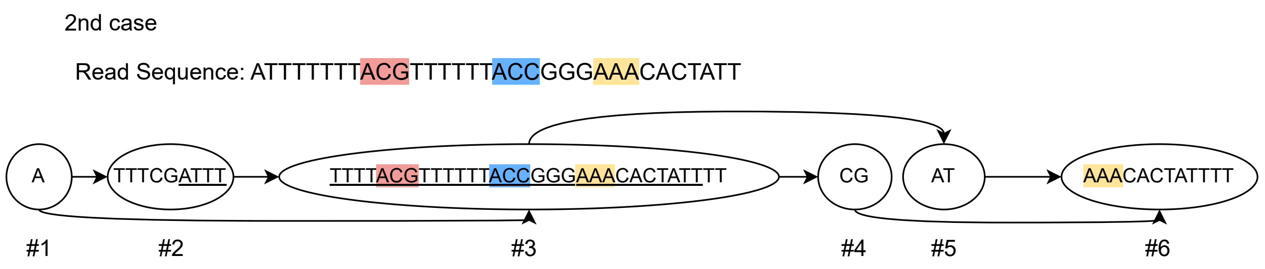

2nd case
Read Sequence: ATTTTTTTACGTTTTTTACCGGGAAACACTATT
A
TTTCGATTT
TTTTACGTTTTTTACCGGGAAACACTATTTT
CG
AT
AAACACTATTTT
#1
#2
#3
#4
#5
#6

Figure 6.6: The node of the seed hit does not have enough characters to the left to align the entire read

Seed hit	Start offset in read	NodeID	Start offset in ref	Relative start offset	Relative end offset	Sequence length in end node
ACG	8	3	4	-4	31	33
ACC	17	3	13	-4	31	33
AAA	23	3	19	-4	31	33

Table 6.3: Example array of seed hits and their alignment metrics

The complexity of finding the real starting offset of the candidate subgraph in this case increases when the starting node has many inNodes, which means that a possible alignment of the starting subsequence of the read might be located in any of those inNodes neighbors, therefore it is essential to find all possible real starting positions, the left limit of the candidate subgraphs. One could claim that by just tracking the biggest hop of inNodes would include all the other possible paths, however this is inefficient for the alignment step since a larger subgraph which is not necessary. Also, as seen when testing the algorithm in various graphs, the same real starting position might be found many times, since there might be more than one paths within the graph to get there. If this is the case, then the subgraph is passed only once to the aligner and there is no need to run the alignment step many times for the same subgraph since when passing the subgraph all the paths are incorporated and examined there in terms of finding the best alignment. In order to find the real starting offset, the negative offset is added to the sequence lengths of the inNodes of the starting node, which are already located in the already calculated array_in during the preprocessing step. When the sum is greater or equal to zero, it means that this inNode has sufficient number of characters to accommodate the starting subsequence of the read till the first character of the minimizer. If the sum remains negative it means that more characters are need and that the inNodes of that inNode need to be recursively checked, till there is no negative sum. This is done in a software environment using a variation of BFS that incorporates path lengths to find subgraphs with sufficient characters to align a read.

However, the challenge is to not use recursion since aiming for a hardware friendly implementation. Therefore, the above idea will have to be altered and adjusted. The proposed hardware friendly implementation of the subgraph extension algorithm in the case of negative starting offset, when more characters are needed to the left of the starting point of the first seed is described in Algorithm 1. The recursion is substituted by the use of two one dimensional arrays, in which the sequence lengths of the inNodes and their respective nodeIDs are loaded respectively. Through the algorithm, the negative offset (whose absolute is the number of remaining needed characters) is added to all elements of the temp table that includes the sequences' lengths. If the sum is positive, it means that the respective inNode, whose ID is found in the respective position in the temp table with the nodeIDs, has sufficient characters to align the rest of the read to the left. In fact, the result of this addition gives the real starting offset of the subgraph. In case the result of the addition is negative, the specific inNode does not have enough characters and therefore its inNodes should be further examined. The nodeID together with the resulting negative value of the addition are stored respectively in two separate one dimensional arrays, where the nodes whose inNodes should be examined are put. The whole process ends, meaning that all the possible starting positions of the candidate subgraphs have been found when the elements of these two arrays are equal to –1. To bound these two arrays that include the nodeIDs and relative offsets, the graph is profiled so as to find the maximum number of paths of length equal to read_length+E that pass from a specific node.

The third case presented in the example below is that of extension subgraph needed to the right of the end node so that all the characters of the read found to the right of the seed, have a sufficient number of characters to be aligned to in the reference graph. Such a case is characterized by the fact that the relative ending offset is greater than the number of characters in the end node. In fact, the number of characters needed is equal to the difference

Algorithm 1 Only Left Extension Hardware Friendly Approach

Inputs : `seed_hit_info[nodeID][start_offset_in_node][start_offset_in_read]`, `relative_start_offset`,`relative_end_offset`

Outputs: `Candidate_Alignment_Subgraph_Coordinates[start_nodeID][end_nodeID][final_relative_start_offset][final_relative_end_offset]`

```
if $relative\_start\_offset < 0$ and $relative\_end\_offset <= sequence\_length[nodeID]$ then
    $remaining\_characters[subgraph\_number\_thres] \leftarrow -1$
    $temp\_subgraph\_nodeIDs[subgraph\_number\_thres] \leftarrow -1$
    $remaining\_characters[0] \leftarrow -relative\_start\_offset$
    $temp\_subgraph\_nodeIDs[0] \leftarrow seed\_hit\_info[nodeID]$
    $offset \leftarrow 1$
    for $i \leftarrow 0$ to $subgraph\_number\_thres - 1$ do
        $cur\_Node \leftarrow temp\_subgraph\_nodeIDs[i]$
        $cur\_offset \leftarrow remaining\_characters[i]$
        $inNodes\_nodeIDs \leftarrow array\_inNodes\_ids[cur\_Node]$
        $inNodes\_seq\_len \leftarrow array\_inNodes\_seq\_len[cur\_Node]$
        for $j \leftarrow 0$ to $max\_inNodes - 1$ do
            $cur\_offset \leftarrow inNodes\_seq\_len[j] + cur\_offset$
            if $cur\_offset > 0$ then
                $start\_nodeID \leftarrow inNodes\_nodeIDs[j]$
                $end\_nodeID \leftarrow seed\_hit\_info[nodeID]$
                $final\_relative\_start\_offset \leftarrow cur\_offset$
                $final\_relative\_end\_offset \leftarrow relative\_end\_offset$
            end
            else
                $temp\_subgraph\_nodeIDs[offset] \leftarrow inNodes\_nodeIDs[j]$
                $remaining\_characters[offset] \leftarrow cur\_offset$  $offset \leftarrow offset + 1$
            end
        end
    end
end
```

relative_ending_offset-end_node_seq_len, which is greater than the real ending offset by one.

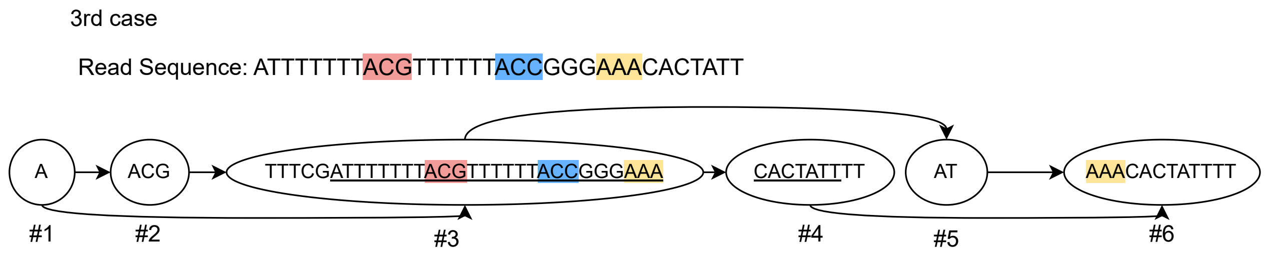

3rd case
Read Sequence: ATTTTTTTACGTTTTTTACCGGGAAACACTATT
A
ACG
TTTCGATTTTTTTACGTTTTTTACCGGGAAA
CACTATTTT
AT
AAACACTATTTT
#1
#2
#3
#4
#5
#6

Figure 6.7: The node of the seed hit does not have enough characters to the right to align the entire read

Seed hit	Start offset in read	NodeID	Start offset in ref	Relative start offset	Relative end offset	Sequence length in end node
ACG	8	3	13	5	40	33
ACC	17	3	22	5	40	33
AAA	23	3	28	5	40	33

Table 6.4: Example array of seed hits and their alignment metrics for the right extension case

An example of this case is illustrated in the small subgraph shown of Figure 6.7. As one can intuitively see, in order for the read to be aligned in the subgraph all the characters in node #3 will be used and additionally seven extra characters are needed to the right. These characters will be taken by node #4 in this case till the letter in position six, or generally from the outNodes of the node. Since node #3 has a second outNode, a second subgraph extracted will include nodes #3,#5 and #6, as two of the needed characters can be retrieved from node #5 and the rest four from node #6. A recursive algorithmic approach that finds all the possible real ending positions of the candidate alignment subgraph would again involve a BFS variation to find all paths with adequate number of characters to align the read. Since recursive function are not synthesized in hardware, as with the previous case, two arrays are utilized to store the nodeIDs of the nodes whose outNodes need to be visited to get enough characters to align the end of the read to the right of the node. By subtracting the sequence length of each outNode from the number of remaining characters the end_offset is being calculated, while extra outNodes are being processed till all possible positions are found, meaning till the real_ending_offset becomes positive (Algorithm 2). The final case combines the two cases presented above, meaning that the candidate alignment subgraph needs to be extended both ways to ensure that there are enough alignment characters both in the beginning and the ending of the read. As it can be seen from the example of Figure 6.8, the relative starting offset is negative while the

Algorithm 2 Only Right Extension Hardware Friendly Approach

```
Inputs :   seed_hit_info[nodeID][start_offset_in_node][start_offset_in_read],
           relative_start_offset,relative_end_offset
Outputs:   Candidate_Alignment_Subgraph_Coordinates[start_nodeID][end_nodeID]
           [final_relative_start_offset][final_relative_end_offset]
$remaining\_ending\_chars \leftarrow sequence\_length[nodeID] - relative\_end\_offset$ if
 $relative\_start\_offset > 0$ and $remaining\_ending\_chars < 0$ then
    $remaining\_characters[subgraph\_number\_thres] \leftarrow -1$
    $temp\_subgraph\_nodeIDs[subgraph\_number\_thres] \leftarrow -1$
    $remaining\_characters[0] \leftarrow remaining\_ending\_chars$
    $temp\_subgraph\_nodeIDs[0] \leftarrow seed\_hit\_info[nodeID]$
    $offset \leftarrow 1$
    for $i \leftarrow 0$ to $subgraph\_number\_thres - 1$ do
        $cur\_Node \leftarrow temp\_subgraph\_nodeIDs[i]$
        $cur\_offset \leftarrow remaining\_characters[i]$
        $outNodes\_nodeIDs \leftarrow array\_outNodes\_ids[cur\_Node]$
        $outNodes\_seq\_len \leftarrow array\_inNodes\_seq\_len[cur\_Node]$
        for $j \leftarrow 0$ to $max\_outNodes - 1$ do
            $cur\_offset \leftarrow outNodes\_seq\_len[j] + cur\_offset$
            if $cur\_offset >= 0$ then
                $start\_nodeID \leftarrow seed\_hit\_info[nodeID]$
                $end\_nodeID \leftarrow outNodes\_nodeIDs[j]$
                $final\_relative\_start\_offset \leftarrow relative\_start\_offset$
                $final\_relative\_end\_offset \leftarrow cur\_offset$
            end
            else
                $temp\_subgraph\_nodeIDs[offset] \leftarrow outNodes\_nodeIDs[j]$
                $remaining\_characters[offset] \leftarrow cur\_offset$  $offset \leftarrow offset + 1$
            end
        end
    end
end
```

relative ending offset is higher than the node sequence length, meaning that the subgraph needs to be extended to both directions.

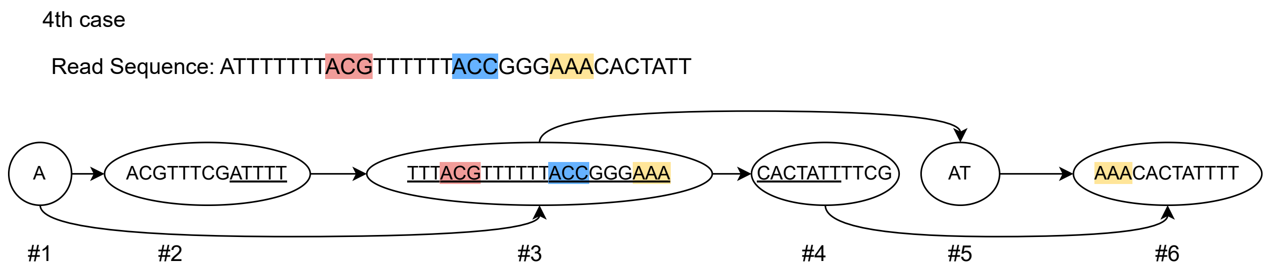


Figure 6.8: The node of the seed hit does not have enough characters to align the entire read in both directions

Seed hit	Start offset in read	NodeID	Start offset in ref	Relative start offset	Relative end offset	Sequence length in end node
ACG	8	3	3	-5	30	23
ACC	17	3	12	-5	30	23
AAA	23	3	18	-5	30	23

Table 6.5: Example array of seed hits and their alignment metrics for the right extension case

Therefore, both algorithms above are performed for subgraph extension in both sides. Additionally, this time all the combinations of real starting and ending positions should be considered as individual candidate subgraphs, so the found starting and ending positions are stored in two arrays which are used to create different combinations of candidate subgraphs.

Algorithm 3 Both Sides Extension Hardware Friendly Approach

```
Inputs  : seed_hit_info[nodeID][start_offset_in_node][start_offset_in_read],
          relative_start_offset,relative_end_offset
Outputs: Candidate_Alignment_Subgraph_Coordinates[start_nodeID][end_nodeID]
          [final_relative_start_offset][final_relative_end_offset]
if $relative\_start\_offset \geq 0$ and $relative\_end\_offset > end\_node\_seq\_len$ then
    possible_starting_offsets ← only_left_extension($seed\_hit\_info$,$relative\_start\_offset$)
    possible_ending_offsets ← only_right_extension($seed\_hit\_info$,$relative\_end\_offset$)
    for $i \leftarrow 0$ to $limit$ do
        for $j \leftarrow 0$ to $limit$ do
            $real\_starting\_position \leftarrow possible\_starting\_offsets[i]$
            $real\_ending\_position \leftarrow possible\_ending\_offsets[j]$
        end
    end
end
```

1.5 Subgraph Linearization

The seeding and subgraph extraction step has provided the coordinates of the candidate alignment subgraph: start and end nodes and start and end offsets. In order to proceed with the alignment step an intermediate step is needed to find the different paths of the subgraph and concatenate the respective sequences to be input to the sequence-to-sequence aligner used in the proposed architecture. In order to reuse the arrays brought from memory for each of the four functions the path extraction functions use the neighbor's nodeIDs array to find the all paths of the subgraph. The paths are then stored as series of nodeIDs and passed to another function that concatenates the respective nodes' sequences. The subgraph extension algorithm ensures that at least one path within the candidate subgraph has enough length to align the entire read and filters out the paths that do not. The example subgraph of Figure 6.9 shows the outputs of the different modules used up until now.

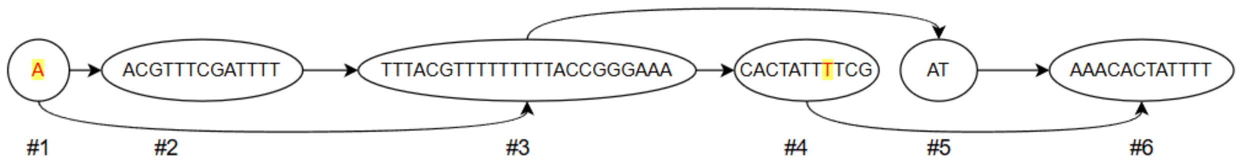


Figure 6.9: Linearization Example

Seeder Output	Start Node = 1, End Node = 4, Start Offset = 0, End Offset = 7
Path Finding Function	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ $1 \rightarrow 3 \rightarrow 4$
Sequence Linearization	AACGTTTCGATTTTTTTACGTTTTTTTTTACCGGGAAAC ACTATTTATTTACGTTTTTTTTTACCGGGAAACACTATTT

Table 6.6: Example outputs for Seeder and Path Finding

1.6 Alignment

1.6.1 Sequence-to-Sequence Alignment

In the proposed design a sequence-to-sequence aligner based on the Bitap algorithm is used. This algorithm is an Approximate String Matching (ASM) one that aims at replacing dynamic programming based alignment algorithms with bitwise operations' based ones. This algorithm efficiently calculates the minimum edit distance between a reference sequence (such as a reference genome) and a query pattern (such as a sequencing read), allowing up to k errors. When k equals zero, the algorithm performs exact pattern matching. The Bitap algorithm begins with a preprocessing phase that transforms the query pattern into m-sized pattern bitmasks, denoted as PM. The algorithm creates one pattern bitmask for each character in the alphabet. Using the convention that 0 represents a match in Bitap, we set PM[a][i] = 0 when pattern[i] equals character a from the alphabet (e.g., A, C, G, T). These pattern bitmasks enable binary representation of the query pattern. Following bitmask preparation, all bits in the status bit vectors R[d] (where d ranges from 0 to k) are initialized to 1. Each R[d] bit vector at text position i maintains partial matching information between text[i:(n-1)] and the query pattern with at most d errors. Since no matches exist at execution start, all status bit vectors are initialized with 1s. The algorithm preserves status bit vectors from the previous iteration with edit distance d in oldR[d] to incorporate partial matches in subsequent iterations. The algorithm processes each text character sequentially, one per iteration. During each text iteration, the algorithm retrieves the pattern bitmask for the current text character (PM) . After computing the status bit vector for exact matches, it calculates status bit vectors for each distance level (R[d] where d = 1...k).For each distance d, the algorithm computes three intermediate bit vectors representing different error types using oldR[d-1] or R[d-1], since adding a new error increases the distance by 1. The intermediate bit vector for the match case (M) uses oldR[d]. The algorithm searches for string matches when the current pattern character is missing. It copies partial match information from the previous character (oldR[d-1]; consuming a text character) without shifting (not consuming a pattern character) to create the deletion bit vector. When the current pattern character and text character don't match, the algorithm takes partial match information from the previous character (oldR[d-1]; consuming a text character) and shifts it left by one position (consuming a pattern character) before storing it as the substitution bit vector . The algorithm searches for string matches when a pattern character is missing from the text. It copies partial match information from the current character (R[d-1]; not consuming a text character) and shifts it left by one position (consuming a pattern character) before storing it as the insertion bit vector. The algorithm identifies string matches only when the current pattern character matches the current text character. It takes partial match information from the previous character (oldR[d]; consuming a text character without increasing edit distance), shifts it left by one position (consuming a pattern character), and performs an OR operation with the pattern bitmask of the current text character (curPM; comparing text and pattern characters) before storing the result as the match bit vector. After computing all four intermediate bit vectors, the algorithm performs an AND operation to consider all possible partial matches. This AND operation preserves all 0s that exist in any of the four bit vectors, representing all potential locations for string matches with edit distance d up to the current position. The result of this AND operation becomes the R[d] status bit vector for the current iteration. This process repeats for each potential edit distance value from 0 to k. When the most significant bit of the R[d] bit vector becomes

0, it indicates a match starting at position i in the text with edit distance d. The algorithm continues traversing the text until all possible text positions have been examined. The Bitap Algorithm [7] explained above in detail can be seen in Algorithm 4.

Algorithm 4 Bitap Algorithm

```
1: Inputs: text (reference), pattern (query), k (edit distance threshold)
2: Outputs: startLoc (matching location), editDist (minimum edit distance)
3: $n \leftarrow$ length of reference text
4: $m \leftarrow$ length of query pattern
5: procedure PRE-PROCESSING
6:    PM ← generatePatternBitmaskACGT(pattern)              ▷ pre-process the pattern
7:    for d in 0:k do
8:       R[d] ← 111..111                                    ▷ initialize R bitvectors to 1s
9: procedure EDIT DISTANCE CALCULATION
10:   for i in (n-1):-1:0 do                                ▷ iterate over each text character
11:      curChar ← text[i]
12:      for d in 0:k do
13:         oldR[d] ← R[d]                  ▷ copy previous iterations' bitvectors as oldR
14:      curPM ← PM[curChar]                                ▷ retrieve the pattern bitmask
15:      R[0] ← (oldR[0]«1) | curPM                         ▷ status bitvector for exact match
16:      for d in 1:k do                                    ▷ iterate over each edit distance
17:         deletion (D) ← oldR[d-1]
18:         substitution (S) ← (oldR[d-1]«1)
19:         insertion (I) ← (R[d-1]«1)
20:         match (M) ← (oldR[d]«1) | curPM
21:         R[d] ← D & S & I & M                            ▷ status bitvector for d errors
22:      if MSB of R[d] == 0, where $0 \leq d \leq k$   ▷ check if MSB is 0
23:         startLoc ← i                                    ▷ matching location
24:         editDist ← d                                    ▷ found minimum edit distance
```

After the GenASM-DC algorithm pinpoints a text's matching location and its edit distance, the new GenASM-TB algorithm takes over. Its role is to find the precise sequence of matches, substitutions, insertions, and deletions, along with their positions (known as a CIGAR string), for the matched region. This process starts from the first character of the matched region, examines each character to determine the optimal operation (match, substitution, insertion, or deletion) in each iteration, and concludes at the last character of the matched region. GenASM-TB leverages the intermediate bitvectors (match, substitution, deletion, and insertion) generated by GenASM-DC. Once a '0' is found at the Most Significant Bit (MSB) of one of the R[d] bitvectors, indicating a string match with d errors, GenASM-TB traces backward through these bitvectors to the Least Significant Bit (LSB), following a chain of '0's (which signify matches at each location) and reversing the bitwise operations. At each position, based on which of the four bitvectors holds a '0', the sequence of operations (the traceback output) is determined for the corresponding alignment found by GenASM-DC. Unlike GenASM-DC, GenASM-TB has an irregular control flow within the stored intermediate bitvectors, which varies depending on the specific text and pattern.

Algorithm 5 GenASM-TB Algorithm

Inputs: text (reference), n (pattern (query)), m, W (window size), O (overlap size)
Output: CIGAR (complete traceback output)

```
1: while $curPattern < m$ && ($curText < n$) do        ▷ start positions of sub-pattern and sub-text
2:     $sub\text{-}pattern \leftarrow pattern[curPattern : curPattern + W]$
3:     $sub\text{-}text \leftarrow text[curText : curText + W]$
4:     intermediate bitvectors ← GenASM-DC(sub-pattern, sub-text, W)
5:     patternI ← W-1                  ▷ pattern index (position of 0 being processed)
6:     textI ← 0                                                        ▷ text index
7:     curError ← editDist from GenASM-DC              ▷ number of remaining errors
8:     <patternConsumed, textConsumed> ←$< 0, 0 >$
9:     prev ← ""                             ▷ output of previous TB iteration
10:    while $textConsumed < (W\text{-}O)$ && $patternConsumed < (W\text{-}O)$ do
11:        status ← 0
12:        if $ins[textI][curError][patternI] = 0$ && $prev ==' I'$
13:            status ← 3; add "I" to CIGAR;                     ▷ insertion-extend
14:        else if $del[textI][curError][patternI] = 0$ && $prev ==' D'$
15:            status ← 4; add "D" to CIGAR;                      ▷ deletion-extend
16:        else if $match[textI][curError][patternI] = 0$
17:            status ← 1; add "M" to CIGAR; prev ← "M"                   ▷ match
18:        else if $subs[textI][curError][patternI] = 0$
19:            status ← 2; add "S" to CIGAR; prev ← "S"            ▷ substitution
20:        else if $ins[textI][curError][patternI] = 0$
21:            status ← 3; add "I" to CIGAR; prev ← "I"          ▷ insertion-open
22:        else if $del[textI][curError][patternI] = 0$
23:            status ← 4; add "D" to CIGAR; prev ← "D"           ▷ deletion-open
24:        if ($status > 1$)
25:            curError–
26:        if ($status > 0$) && ($status \neq 3$)
27:            textI++; textConsumed++                             ▷ M, S, or D
28:        if ($status > 0$) && ($status \neq 4$)
29:            patternI–; patternConsumed++                        ▷ M, S, or I
30:    curPattern ← curPattern+patternConsumed
31:    curText ← curText+textConsumed
```

The overall workflow returns the CIGAR string together with the coordinates of the alignment subgraph, start node, end node, starting offset and ending offset.

1.6.2 Sequence-to-Graph Alignment

An alternative to using the sequence-to-sequence aligner together with the linearization step would be to use a sequence-to-graph aligner straight after the seeding procedure. The sequence-to-graph aligner takes as input the candidate alignment subgraph extracted coordinates:start node, end node, start offset and end offset, and proceeds to perform alignment without the need to extract paths from the subgraph by just linearizing the entire subgraph sequence and applying a variation of the Bitap algorithm to align the read with the subgraph sequence. Before the actual alignment step, we propose the addition of a subgraph filter to filter out nodes whose information will not be used, in order to reduce the sizes of linearized sequences and reduce latency.

The subgraph filtering step takes advantage of the sorted nature of the graphs and involves removing nodes that do not have at least one edge, either incoming or out coming, that originates or terminates respectively in a node of the extracted subgraph. Figure 6.10 shows these two cases.The first case shown in the green subgraph example, is filtering out the red node which is unreachable by any node with a smaller nodeID in the subgraph, therefore the node does not take part in the alignment, since if it could have another smaller subgraph starting from the red node would emerge. Respectively the red node of the blue subgraph will be filtered out since the node does not have an outNode to a node with nodeID smaller than the end Node ID, thus it will not play any part in the alignment step, as if it had another smaller subgraph ending in the red node would have been extracted.

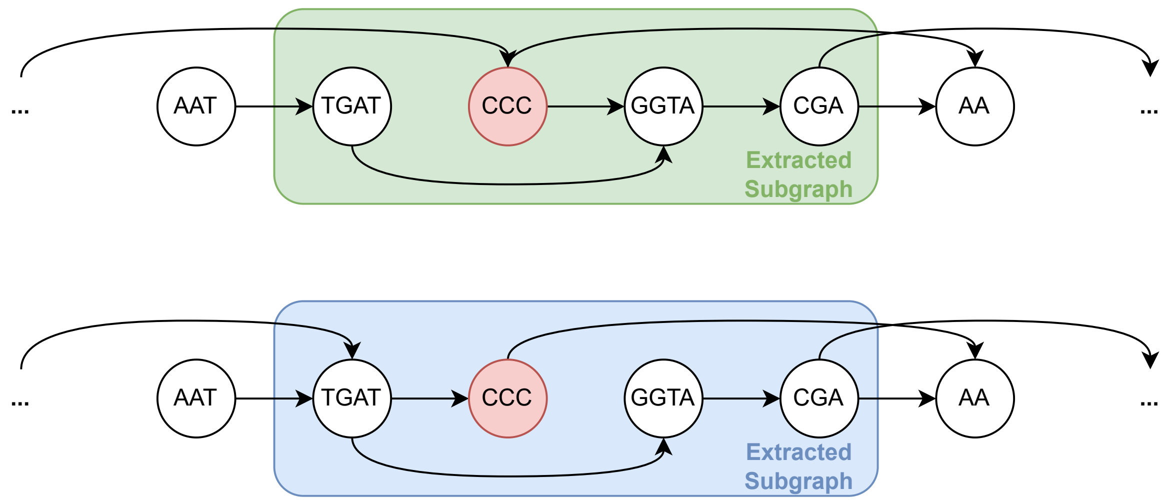


Figure 6.10: First Subgraph Filtering Step

The second part of the filtering involves removing nodes with sequence lengths longer than the characters remaining to be aligned with the intermediate nodes. The number of these characters, also referred to as character filtering threshold can be calculated as:

$character\ filtering\ threshold\ =$
$read\ length - end\ offset - (start\ node\ sequence\ length - start\ offset)$

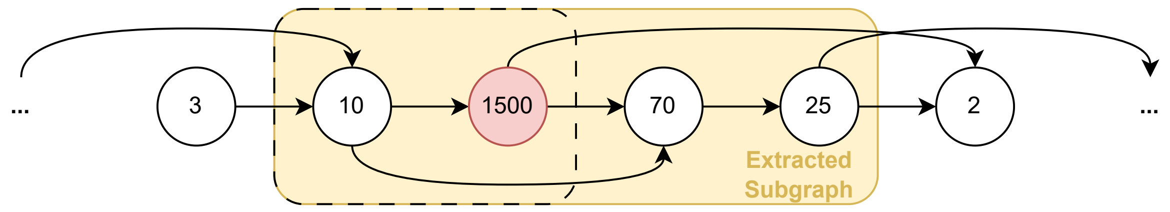


Figure 6.11: Second Subgraph Filtering Step

Figure 6.11 shows an example of a node filtered out in the case of a short read's alignment. The eliminated node might be connected with at least one other node within the subgraph passing initial filtering, but has a sequence length far greater than the character filtering threshold when the read length is for example equal to 100. Another smaller subgraph (noted within the dotted border) will be extracted to include candidate alignment characters from the red node taking into account again only the needed characters and not having to retrieve the entire sequence which will not be needed in the case of the yellow subgraph.

After filtering subgraph linearization is performed, which is basically the process of concatenated the sequences of all nodes of the subgraph from start to end node, taking again advantage of the sorting we performed during the preprocessing step. For example, the linearized subgraph of the green case of Figure 6.10 will be the sequence: TGATGGTACGA. Additionally, before proceeding with the alignment step we create two bitvectors with the length of the concatenated sequence to keep track of the starting and ending characters of hops. Each bit of the bitvector found in the position of a node starting character is set to one and the same respectively for each bit of the end positions bitvector that corresponds to a node's ending character. Especially ending characters of nodes are very important as they show the existence of a hop at that place, which will be taken into account by the sequence-to-graph alignment algorithm. As previously referred to, the sequence-to-graph alignment algorithm implemented in the proposed design is a variation of the Bitap algorithm, called BitAlign [8] also used by SeGraM. The algorithm utilizes the linearized subgraph sequence, positional bitvectors, the edit distance threshold and the read query sequence, to calculate minimum edit distance (BitAlign algorithm) and output the CIGAR string (extracted during the traceback step). The algorithm performs bitwise operations to align the sequences just like Bitap however incorporates extra bitvectors to store hop related alignment information. The algorithm starts by generating four bitmasks (PM) from the read sequence (one for each letter of the DNA alphabet). Those bitvectors are created by setting PM[a][i]=0, when read[i]=a and $a \in$ A,C,G,T, since 0 means match in the Bitap algorithm. In contrast with the Bitap algorithm, BitAlign needs to store all of the status bitvectors for all of the text interations (thus allR[n][d]), where n the length of the linearized subgraph, in contrast with the case of sequence-to-sequence alignment which stores the most recent status bitvectors. This difference is again due to the existence of hops. As in the classical Bitap algorithm, the reference text is traversed in reverse order and the corresponding pattern bitmask is retrieved for each character of the linearized subgraph. Additionally to computing three of the intermediate bitvectors (corresponding to match, substitution and deletion), all successor nodes that the current node has are incorporated in the status bitvectors in the places referring to ending characters. Insertion is the only intermediate bitvector not calculated separately for the hops since insertion consumes a character from the read rather than the reference. The bitwise operations used to calculate these bitvectors are the same as in Bitap, while the intermediate bitvectors are ANDed and stored into the R[d] bitvector which will then be used to calculate the minimum edit distance. Traceback takes the intermediate bitvectors as input and outputs the CIGAR string using Algorithm 5.

Algorithm 6 BitAlign Algorithm

```
 1: $n \leftarrow$ length of linearized reference subgraph
 2: $m \leftarrow$ length of query read
 3: $PM \leftarrow$ generatePatternBitmasks(query-read)
 4: $allR[n][d] \leftarrow 111\ldots111$                ▷ init $R[d]$ bitvectors for all characters with 1s
 5: for $i$ in $(n-1) : -1 : 0$ do
 6:     $curChar \leftarrow$ subgraph-nodes[$i$].char
 7:     $curPM \leftarrow PM[curChar]$                       ▷ retrieve the pattern bitmask
 8:     $R[0] \leftarrow 111\ldots111$
 9:     for $j$ in subgraph-nodes[$i$].successors do
10:         $R[0] \leftarrow ((R[j][0] \ll 1) \mid curPM) \& R[0]$      ▷ exact match calculation
11:     $allR[i][0] \leftarrow R[0]$
12:     for $d$ in $1 : k$ do
13:         $I \leftarrow (allR[i][d-1] \ll 1)$                          ▷ insertion
14:         $R[d] \leftarrow I$                               ▷ status bitvector for $d$ errors
15:         for $j$ in subgraph-nodes[$i$].successors do
16:             $D \leftarrow allR[j][d-1]$                              ▷ deletion
17:             $S \leftarrow allR[j][d-1] \ll 1$                        ▷ substitution
18:             $M \leftarrow (allR[j][d] \ll 1) \mid curPM$             ▷ match
19:             $R[d] \leftarrow D \land S \land M \land R[d]$
20:         $allR[i][d] \leftarrow R_d$
21: return traceback($allR$, subgraph, query-read)
```

2 Hardware Architecture

In this section, the proposed hardware architecture is presented in detail. Additionally, we show the mapping of the algorithms and steps described above to the different hardware modules and blocks. We start by presenting a high level overview of the design including all designs and proceed with analyzing their micro architecture. We will also be referring to the HLS directives used to achieve parallelism and improvement of the overall design.

2.1 Overview of the Integrated Design with S2S Aligner

The integrated design can be broken down into three separate blocks each one of which performs one specific step of the proposed workflow: seeding, linearization and alignment. As shown in Figure 6.12 the three respective blocks are in series since the output of one works as an input to the next one. The host CPU passes a batch of N reads of m length to the seeder who is assigned to find all candidate alignment subgraphs of the reference graph for the reads of the batch. Within the seeder there are three discrete sub steps to do so, including minimizer extraction from all the reads of the batch, minimizer searching in the reference minimizers to get seed hits and then find all subgraphs sufficient to align each read of the batch. The subgraphs' coordinates, including start and end node as well as start and end offsets within those nodes sequences, are then passed to the linearization block which basically needs to produce the concatenated sequences of the paths of the subgraph as the reference sequence used in the alignment step. Within the linearization block, there are two different cases executed, the one, which is actually the most common for short reads, works one single node subgraphs, meaning cases where the entire candidate alignment reference sequence is found within one node, and the other involving also path finding before producing the actual concatenated sequence since the subgraphs include more than one nodes.

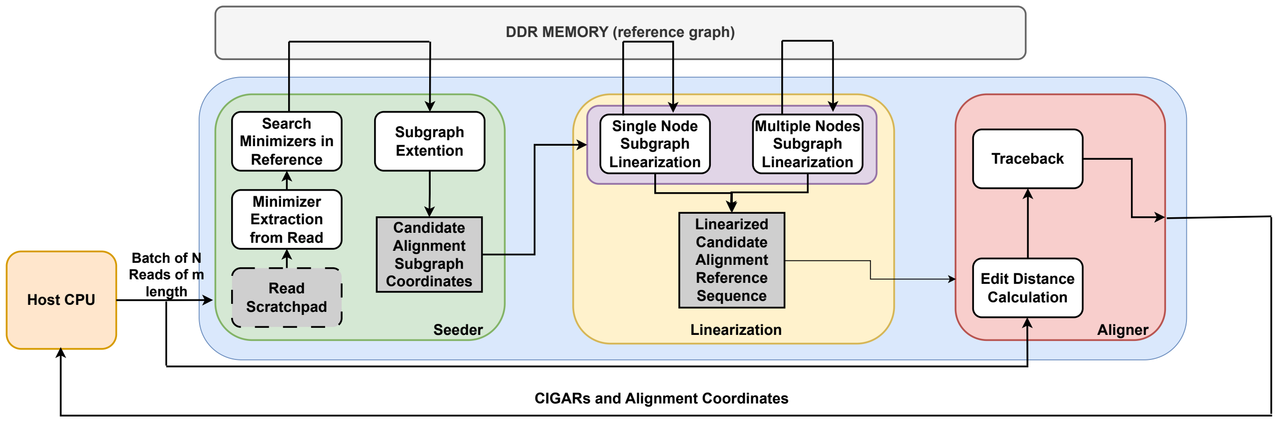

DDR MEMORY (reference graph)
Search Minimizers in Reference
Subgraph Extention
Minimizer Extraction from Read
Candidate Alignment Subgraph Coordinates
Read Scratchpad
Seeder
Single Node Subgraph Linearization
Multiple Nodes Subgraph Linearization
Linearized Candidate Alignment Reference Sequence
Linearization
Traceback
Edit Distance Calculation
Aligner
Host CPU
Batch of N Reads of m length
CIGARs and Alignment Coordinates

Figure 6.12: High Level Architecture of Design with Sequence-to-Sequence Aligner

The final alignment block uses the concatenated reference sequence to perform sequence-to-sequence alignment between it and the respective read sequence, producing an opened CIGAR string that is returned together with the alignment coordinates (start/end nodes and start/end offsets) to the host. The host then produces the final CIGAR format by counting the characters of the open CIGAR string. As depicted in the diagram, only the seeding and linearization block retrieve data from the main memory while the aligner uses data found in kernel arrays. The three blocks are integrated within one kernel that has as input the read sequences organized in batches and outputs their alignment locations in the reference graph together with a CIGAR string that shows the type and number of differences in between two sequences (reference and read). All blocks can also be used as separate kernels with the host giving the corresponding inputs in each case.

2.1.1 Seeder Block Architecture

The Seeder architecture is shown in more detail in Figure 6.13. The three orange blocks represent the three major blocks also seen in Figure 6.12 namely: minimizer extraction, search and subgraph extension blocks. There are two different versions of the seeder module evaluated in this work, one with and one without read scratchpad. By incorporating a read scratchpad to get the reads in batches instead of inputting a single one each time the entire kernel is run, we increase resources utilization however improve performance as it will be shown in the next chapters. The first part main module of the seeder executes the minimizer extraction from the reads. Within this module the reads are broken done into windows in parallel and then the minimizers are extracted for these windows in parallel. By previously isolating windows the complexity of the loop is reduced to O(n) instead of O(n^2) which would have been the case if the window was rolled over the read. Since the length of the reads, as well as the minimizer length k and window size w are known one can calculate before hand the number of minimizers expected to be extracted from a batch of N reads. The window is slide with a step w-k, therefore the number of windows can be found and is equal to the number of the minimizers. With those known values it is possible to create static arrays to store the windows' sequences and the minimizers' sequences. The minimizers' sequences are then stored in a scratchpad together with their positional offsets within the read and the read identifier of the query sequence they were extracted from. To achieve parallelism in this module we use array partition for the tables that include the windows and minimizers' information and unrolling for the loop that calls the extract minimizers from window sub module.

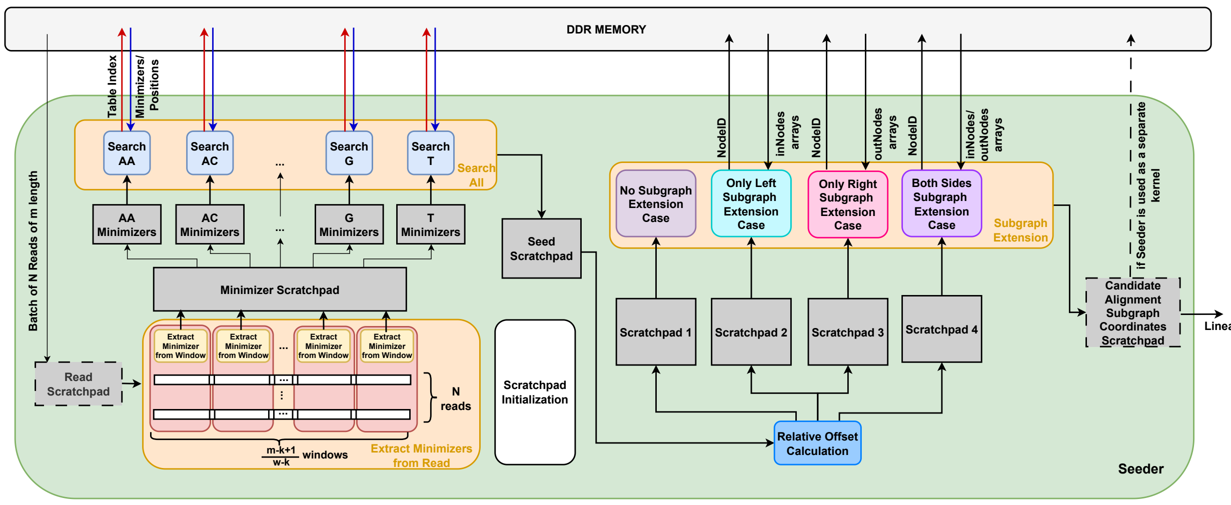


Figure 6.13: Seeder Block Architecture

The scratchpad initialization block is a function that initializes all the values of the scratchpads of the kernel when a new read or group of reads is input in the kernel. This block could be eliminated however in that case in the output of the seeder we would have results referring to previously processed reads or groups in which case we would need to add an extra filtering step to discard them before streaming them to the aligner. Additionally if remains of the previously processed reads were in left in the scratchpad in most cases additional unnecessary calculations would need to be performed within the different blocks or add additional filtering steps in between them. Scratchpad initialization is run in parallel with the extract minimizers block using a dataflow directive. After having written the minimizer information in the minimizer scratchpad, we divide the minimizers into smaller scratchpads based on their first two starting characters. The notion behind this grouping choice relies on the profiling data we produced during the pre-processing step and the limitation of the available memory channels of one SLR. As we saw during profiling, minimizers starting with A and C are much more numerous than the ones starting with G and T, therefore an extracted minimizer starting with A will need significantly higher time to be found in the reference. This kind of bucketing not only allows us to not search for a minimizer through the entire reference minimizer table but also introduce parallelism in the search process. In order for multiple functions to access main memory in parallel each needs to communicate with the DDR memory through a different m_axi channel. As explained in Chapter 4, all arguments of a function are grouped together in one bundle As shown in Figure 5.12, minimizers starting with A and C are much more abundant therefore since we aim breakdown the searching process into parallel sub processes we established a more balanced grouping of the reference minimizers. Different groupings have also been tried, however it is possible to have up to sixteen different groups due to the number m_axi channels available in one SLR. Since the minimizers starting with G and T are already much less than the ones starting with A and C, we choose not to divide them even further. For the cases of minimizers starting with A and C, in order to keep symmetry and not create further inbalance by dividing one into more groups than the other and after having seen that indeed withit the reads they are the most common minizers, we divide each category of them into four subcategories, taking into account also their second starting character, meaning that we have now ten groups of minimizers starting with:AA,AC,AG,AT,CA,CC,CG,CT,G and T. The existence of the minimizer scratchpad that includes all minimizers enables fast changes in the architecture of the kernel, to have more or less subgroups or even skip the entire optimized searching block and perform search of the minimizers in the reference without any prior subgrouping, which would be (as we have seen) much worse in terms of performance but would have lower utilization percentages. The scratchpads of the groups of the minimizers are divided into two parts, one having the actual minimizer sequences and the other the positional offset of the minimizer in the read and the read id of the read they have come from, since after distributing them in the different groups there is no other way to keep track of which read of the batch they came from. Each one of the groups' scratchpads is input to one of the search submodules that run in parallel to each other thanks to the attribution of different memory interfaces to each one of them. Search modules communicate with the DDR memory to retrieve the respective reference minimizers located in different arrays within the memory. More specifically, the reference minimizers are also divided into buckets as previously said, corresponding to the minimizers as grouped after extraction in the kernel. The respective table index is provided to the each search function and the correct minimizers are retrieved from the main memory. Figure 6.14 shows the architecture of the search module which is placed ten times in this version of the seeder architecture. The sub module takes as input the reads' minimizers' scratchpad as well as the index of the table each function refers to. There are two smaller blocks that are executed in parallel within the search module, one used to prefetch batches of reference minimizers from the main memory and the second one to search for the read minimizers in the already prefetched batch. It works in a double buffer way technique, while the $i+1$-th batch is fetched, searched is performed on the i-th batch. If the read minimizer is found in one batch, then the actual search block retrieves from memory the positions of that specific minimizer in the reference. Additionally, we use a limit in the number of seeds instead of filtering out the most frequent minimizers, so that the user is able to determine the maximum number of candidate alignment subgraphs. Positions of the found minimizer in the reference are retrieved from main memory till until they reach the number of seeds limit or their occurrence in the read (for example most minimizers are found only once in the reference therefore only one position will be retrieved). The position information includes the nodeID of the node where the reference minimizer is extracted from and its starting offset in that node's sequence.

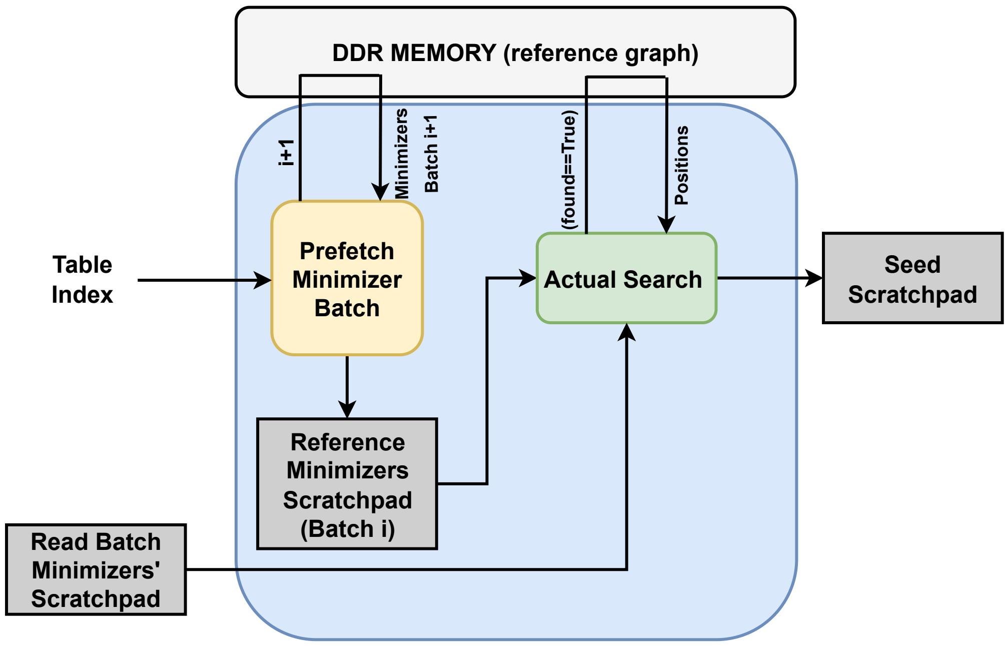


Figure 6.14: Search Block Architecture

As it will be better shown in the evaluation part, the introduction of the read scratchpad increases the performance of the kernel basically due to the nature of the implementation of the search step. More specifically, as the scratchpad increases more minimizers are put into the minimizers groups' scratchpads meaning that by accessing the main memory once to bring in a batch of the reference minimizers, more minimizers can be searched therefore the performance of the system for the entire volume of the dataset increases, while in the case of a single read batch, the same batches need to be retrieved for each of the minimizers of the dataset again and again increasing memory accesses and thus system latency.

The read minimizers found in the reference are now called seed hits. The seed hits position is now characterized by four values: the starting offset in the read sequence (calculated during the minimizer extraction step), the starting offset in the reference sequence of the node it was extracted from, the nodeID of that node and the read ID of the sequence the minimizer came from. These four characterizing numbers are stored in the seed scratchpad. The next step is to calculate the relative offsets of each seed in order to perform subgraph extension if needed. As also previously explained the two relative offsets: starting and ending are calculated as shown below:

$$\begin{aligned}\text{relative\_start\_offset\_of\_the\_seed\_hit} &= \text{seed\_start\_pos\_in\_ref\_graph\_node}\\ &\quad - \text{seed\_start\_pos\_in\_the\_read}\end{aligned}$$

$$\begin{aligned}\text{relative\_end\_offset\_of\_the\_seed\_hit} &= \text{read\_length}\\ &\quad - \text{seed\_start\_pos\_in\_the\_read}\\ &\quad + \text{seed\_start\_pos\_in\_ref\_graph\_node}\end{aligned}$$

In order to categorize a seed in one of the four subgraph extension categories the above offsets are compared to zero and the sequence length of the node of the seed hit respectively. The combination of the outcomes of these two comparisons determines in which of the designated scratchpads the seed hit will be written to. More specifically as explained analytically in Section 1.4, if the relative start offset is greater or equal to zero and the relative start offset smaller or equal to the seed hit's nodeID, there are enough characters to align the read within the node of the seed hit and therefore there is no need for extension. The seed hits that fall into that category are written into Scratchpad 1 as seen in figure 6.13 and then passed to the no subgraph extension case which basically writes the relative offsets results to the candidate alignment subgraph coordinates scratchpad if the seeder kernel is used alone, or call the linearization module to proceed with the alignment if the seeder is part of the integrated alignment design. The only left extension case scratchpad includes the seed hits whose start relative offset is negative and ending relative offset smaller or equal to the nodeID sequence length (Scratchpad 2). The only left subgraph extension which is called for those seed hits retrieves from memory part of the two inNodes arrays (one with parental nodeIDs and one with parental nodes' sequences' length). the size of the subarrays retrieved is determined by the profiling shown in Figure **??** that determines the maximum alignment hop existing in the read dataset final alignments. It is also possible to introduce another value that basically is equal to the maximum hop existing in a path of length equal to the read sequence length in the reference graph. By imposing this limit, one is sure that all possible reads could be aligned. The same is done for the only right subgraph extension case sub module which is input the cases of seed hits where the relative start offset is positive and the relative end offset is greater than the seed hit's sequence length. This function retrieves from memory and stores in local arrays, just like the left extension case, a part of the outNodes arrays with the children node's nodeIDs and sequence lengths. Again only the part of the array that includes the information of the nodeID of the seed hit is retrieved. Finally, the fourth case of both sides subgraph extension acts upon the seed hits of Scratchpad 4, which have both a negative start offset and an end offset greater than the node's sequence length. This block retrieves from memory sub arrays of all four of the inNodes and outNodes arrays to find final starting and ending subgraph coordinates. All final coordinates from the four functions are written into the scratchpad which is then sent but to the host if the seeder is used as an individual kernel or each function calls the respective block of the linearization module if the seeder is in the integrated design. All four extension functions run in parallel to one another. To be able to run the fourth function concurrently with the second and third which need to access the same arrays in the main memory, again we assign different memory interfaces and create two different pointers to show in the same place in memory so that one of them can be used by the fourth function.

2.1.2 Linearization Block

The second block of the integrated design is the linearization block that is responsible for finding the paths within multiple nodes candidate alignment subgraphs and concatenating their nodes' sequences into a single linear reference sequence. It also accommodates the case of a single node alignment subgraph by just retrieving from memory the part of the sequences array that corresponds to the seed hit node and respective starting and ending positions.

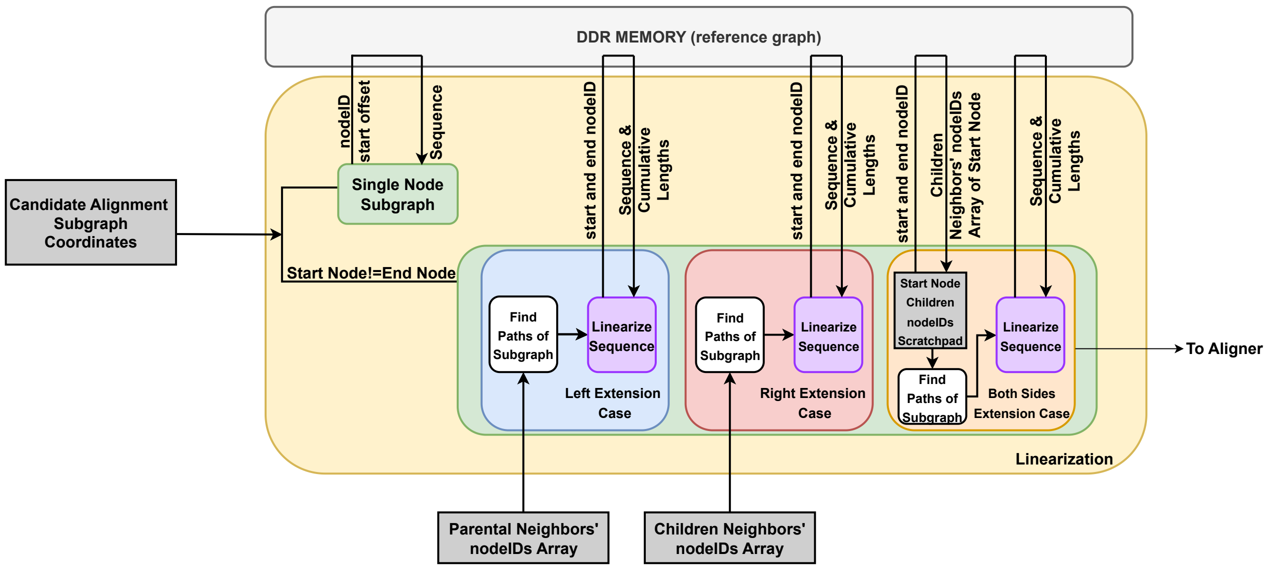

DDR MEMORY (reference graph)
nodeID
start offset
Sequence
start and end nodeID
Sequence & Cumulative Lengths
start and end nodeID
Sequence & Cumulative Lengths
start and end nodeID
Children
Neighbors' nodeIDs Array of Start Node
Sequence & Cumulative Lengths
Candidate Alignment Subgraph Coordinates
Single Node Subgraph
Start Node!=End Node
Find Paths of Subgraph
Linearize Sequence
Left Extension Case
Find Paths of Subgraph
Linearize Sequence
Right Extension Case
Start Node Children nodeIDs Scratchpad
Find Paths of Subgraph
Linearize Sequence
Both Sides Extension Case
To Aligner
Linearization
Parental Neighbors' nodeIDs Array
Children Neighbors' nodeIDs Array

Figure 6.15: Linearization Block Architecture

Which sub block of the linearization module is used for each candidate subgraph is determined by two conditions. The first is determining whether the subgraph is a single node one (start nodeID is the same with the end nodeID) or a multi-node one (start nodeID is different from the end nodeID). Starting off with the simplest case of the single node subgraph, the corresponding sub module using the nodeID accesses the cumulative lengths table to find the positions of the sequence array that it should retrieve. The starting retrieving position is calculated by adding the start offset to the cumulative length of the node of the single node subgraph. The sequence is brought from the main memory to a local table and is ready to be passed to the sequence-to-sequence aligner as the reference sequence.

The second case is chosen when the subgraph has been extracted through only left side extension. The second and third cases could be merged with the fourth, however having them separated reduces main memory accesses as each variation uses the already brought inNodes and outNodes arrays information respectively without the need of reloading them. The second case uses the inNodes nodeIDs array to find the different paths present in the subgraph. The find paths of subgraph module basically goes over the inNodes array using a for loop and appends the paths will finding parental nodeIDs that are lower than the starting nodeID, meaning that they are outside the borders of the subgraph. To have fixed bounds in the path finding loop, two parameters need to be determined, the maximum depth of a path and the maximum number of paths. This design sets the maximum depth as the difference between the starting and ending node and to set the size of the array that keeps the paths, as the maximum hop existing in the final alignments as calculated by profiling of Figure **??**. To calculate the maximum number of paths we use calculate the maximum number of paths in the final alignment subgraphs as extracted by GraphAligner. Each found path is stored as an array of nodeIDs and passed to the sequence linearization block, which uses the start and end nodeIDs to retrieve the subgraph's cumulative lengths array region in a local table. Then it obtains the sequences using the cumulative lengths values as indexes, manipulating start, intermediate and end nodes accordingly (gets the full sequence of the intermediate nodes and the sequence lengths starting from the start offset and ending with the end offset for the start and end nodes respectively). The third case utilizes the already loaded outNodes array with the nodeIDs of the children of the nodes to find the paths of the subgraph in a similar way the second case does. As soon as the paths are found the respective sequences are retrieved from the main memory and concatenated into a single reference sequence string. The two cases described above have the characteristic that the neighbor information exist already in the prefetched arrays used for the subgraph extension steps. This is not the case when it comes to both sides extension since the information of the intermediate nodes of the subgraph is not yet obtained. We have already information on the parents of the start node and the children of the end node, however not necessarily information on the children of the start and parents of the end node as we would need to find the paths of the subgraph. Therefore in this case, there is an additional memory access to retrieve part of the outNodes array that corresponds to the candidate subgraph (only the rows starting from the start node's and ending with the ending node's since the array is sorted). Using the information from this array, the paths are extracted the same way as described for the previous cases and the sequence is linearized into the reference sequence string.

2.1.3 Aligner Block

After the reference sequences have been concatenated in linear arrays, we use them as the linear reference sequence to perform sequence-to-sequence alignment with the read sequence. The aligner block as the alignment algorithm is divided into to main sub parts: the edit distance calculation part, that calculates the number of edits needed for the reference and the read query to become the same, and the traceback step that produces the CIGAR string which determines the types of edits and the positions they have been made.

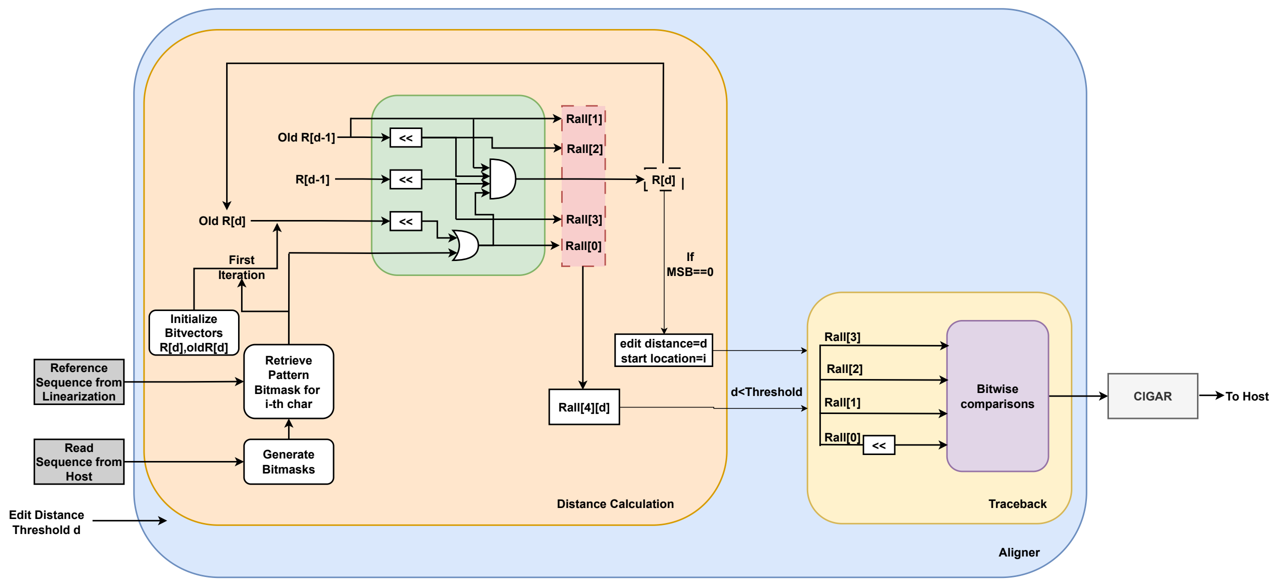
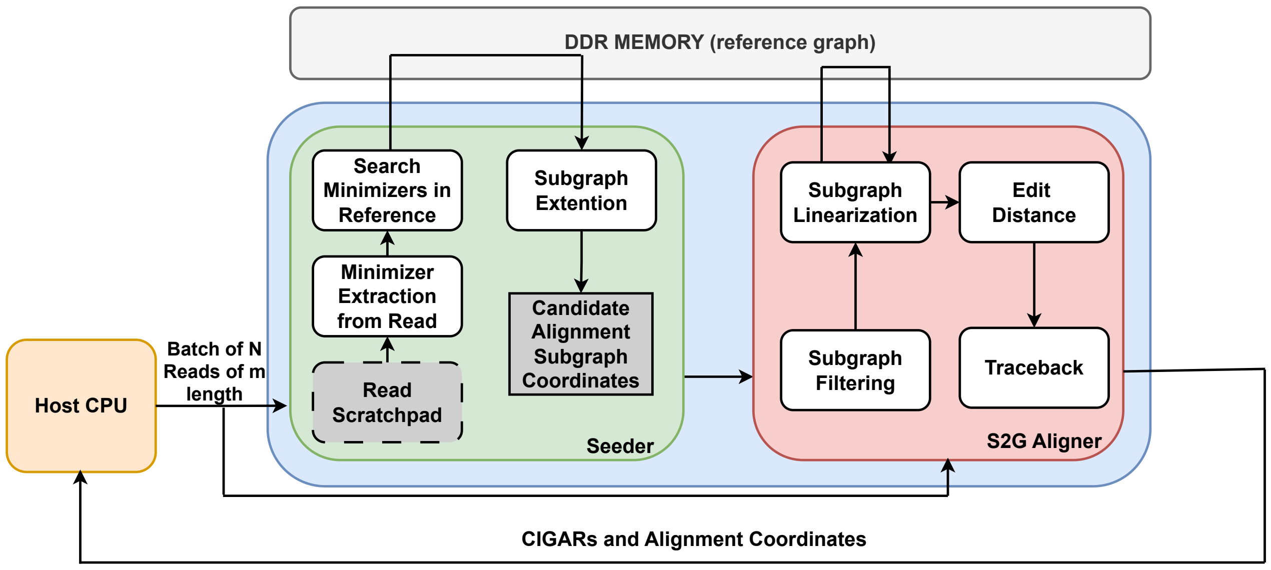


Figure 6.16: Aligner Block Architecture

The aligner block can exist as a standalone aligner block as well as as part of the integrated design to perform sequence-to- sequence alignment. It is the only block the design of Figure 6.12 that does not communicate with the main memory and takes as input the reference sequence produced and stored in a scratchpad during the linearization step, together the read sequence that has already been retrieved from memory during the seeding step. To use a lower number of bits to represent the sequences we transform the character sequence with a bitvector of length double of that of the bitvector (we are taking advantage of the ap_uint data types in HLS) and represent each character with two bits: $A \leftarrow 00$,$C \leftarrow 01$, $G \leftarrow 10$, $T \leftarrow 11$. The same is done for the reference sequence which has the same length with the read sequence increased by the edit threshold (in case of insertions), noted as d. The first step is to generate the bitmasks for the read bitvector, producing four bitvectors one for each character of the DNA alphabet. Initialization of the R bitvectors to 1 is also performed in parallel batches (through HLS unroll and array partition pragmas). Then a for loop starts iterating over the reference bitvector in reverse order, two bits are retrieved during each iteration and depending on their value the corresponding element of the bitmask bitvector array is retrieved. Then the retrieved bitmask is used to calculate exact match bitvector (Rall[0]) and then the other bitvector are calculated using bitwise operations as seen in the green block of Figure 6.16 and in Algorithm 4. We choose to store the match, deletion, substitution and insertion intermediate bitvectors for each edit distance d (ranging from 0 to the edit distance threshold k) in the Rall array, which is a 3-dimentional array of bitvectors (ap_uint) of length equal to the read, with dimensions [4][k+1][n], so as to store for each edit distance and reference character, the four intermediate bitvectors ($Rall[0] \leftarrow Match$,$Rall[1] \leftarrow Deletion$, $Rall[2] \leftarrow Substitution$, $Rall[3] \leftarrow Insertion$). These intermediate bitvectors are calculated in parallel using unroll and partition directives. The intermediate bitvectors are ANDed to produce the status bitvector for each edit distance d. Then we check whether the MSB of the R[d]s and if it is zero, edit distance is set equal to d and alignment starting location at the i-th reference character. After having iterated over the entire reference sequence if the minimum edit distance is less than the edit distance threshold, the Rall array is passed to the Traceback block, otherwise the candidate alignment pair (reference and read) is discarded. The traceback block gets the intermediate bitvectors and checks the different conditions shown in Algorithm 5 to characterize the type of edit. Conditions are represented as bool variables whose values are calculated in parallel. Then depending on the type of edit detected, there is either pattern consumption (in cases of deletion, match or substitution) or text advancement (in cases of insertion, match or substitution) while all operations except match consume edits. The CIGAR, which in our case is represented as a bitvector (ap_uint) of size equal to two times the read length, is then updated depending on the type of edit that was detected. Each one of the four different types of edits is encoded in two bits ($Match \leftarrow 00$, $Insertion \leftarrow 01$, $Deletion \leftarrow 10$ and $Substitution \leftarrow 11$). The CIGAR bitvector representation is then returned to the top algner function to be translated into a character string of length equal to the read length. This string is then returned to the host where it is compressed in conventional CIGAR format.

2.2 Overview of Integrated Design with S2G Aligner

An alternative architecture that is implemented to perform the same task as the architecture presented in Figure 6.12 includes a sequence-to-graph aligner that removes the need for path extraction to perform sequence-to-sequence alignment. This design is shown in Figure 6.17. The linearization block is completely removed, the seeder block remains the same and the sequence-to-sequence aligner block is replaced by a sequence-to-graph aligner module with a slightly alter architecture.



DDR MEMORY (reference graph)
Search Minimizers in Reference
Subgraph Extention
Minimizer Extraction from Read
Candidate Alignment Subgraph Coordinates
Read Scratchpad
Seeder
Subgraph Linearization
Edit Distance
Subgraph Filtering
Traceback
S2G Aligner
Host CPU
Batch of N Reads of m length
CIGARs and Alignment Coordinates

Figure 6.17: High Level Architecture of Design with Sequence-to-Graph Aligner

2.2.1 Sequence-to-Graph Aligner Block

A more detailed schematic of the sequence-to-graph aligner block is given in Figure 6.18. The block includes three sub modules for: subgraph filtering and linearization, alignment and traceback. The entire module takes as input the subgraph coordinates as produced from the Seeder block together with some other data that were retrieved from memory during the seeding on chip. More specifically, the filtering conditions that implement the filtering processes described in Subsection **??** use the already fetched parts of the inNodes and outNodes arrays as well as the respective arrays including the lengths of the nodes. The nodes that make it through the first filtering condition are filtered further by the second condition. The nodes of the subgraph that remain are put in a local scratchpad are then passed to the subgraph linearization sub block (which should not be confused with the path sequence concatenation block found within the linearization block of Design 6.12). This block uses the nodeIDs of the subgraph to retrieve from memory the node sequences along with the cumulative lengths which will be used for the production of the start and end characters bitvectors. The sub module concatenates the sequences into a character array and based on the cumulative lengths, sets ending characters bits of the respective bitvector equal to one. The same goes for the starting characters bitvectors. Since the sizes of the arrays used for the concatenated sequence and bitvectors need to be predetermined, we perform profiling of the alignment subgraphs as extracted by GraphAligner to see the maximum number of an alignment subgraph's concatenated sequence after filtering and use this value as the fixed size of these arrays. Additionally, we introduce another integer array with that size to store the node of origin for each character matching each position of the array with the position of the character in the linearized sequence.

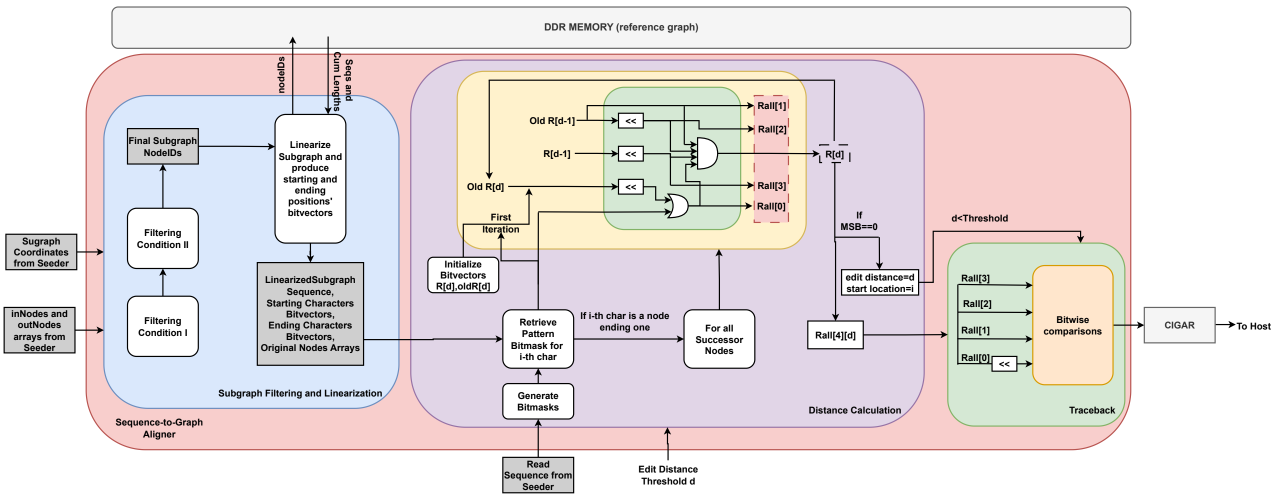

DDR MEMORY (reference graph)
nodeIDs
Seqs and Cum Lengths
Final Subgraph NodeIDs
Linearize Subgraph and produce starting and ending positions' bitvectors
Filtering Condition II
Filtering Condition I
Sugraph Coordinates from Seeder
inNodes and outNodes arrays from Seeder
LinearizedSubgraph Sequence, Starting Characters Bitvectors, Ending Characters Bitvectors, Original Nodes Arrays
Subgraph Filtering and Linearization
Sequence-to-Graph Aligner
Old R[d-1]
R[d-1]
Old R[d]
<<
Rall[1]
Rall[2]
Rall[3]
Rall[0]
R[d]
First Iteration
Initialize Bitvectors R[d],oldR[d]
Retrieve Pattern Bitmask for i-th char
If i-th char is a node ending one
For all Successor Nodes
Generate Bitmasks
Read Sequence from Seeder
Edit Distance Threshold d
If MSB==0
edit distance=d start location=i
Rall[4][d]
Distance Calculation
d<Threshold
Rall[3]
Rall[2]
Rall[1]
Rall[0]
Bitwise comparisons
Traceback
CIGAR
To Host

Figure 6.18: Sequence-to-Graph Aligner Architecture

The linearized graph subsequence, the starting-ending characters' bitvectors and the characters' origin node array are passed to the distance calculation sub block which implements the BitAlign algorithm. In terms of architecture is very similar to the sequence-to-sequence aligner block of Figure 6.16. The only additional sub module is the one treating treating the nodes' ending characters which indicate the existence of a hop. For all successor nodes, which can be found using the already retrieved outNodes table, the Rall bitvectors are calculated using the same bitwise operations as in 6.16. This way it is possible to not calculate multiple times bitvectors of characters found in one node every time a new path is explored. By storing the intermediate bitvectors of the ending characters we don't need to recalculate them when proceeding with the alignment to the successor nodes, in contrast with the case of Design 6.12 where the bitvectors for each node's character are calculated again for each path that node is found in, demanding extra execution time as it will be shown in the next chapter.

Chapter 7

Experimental Evaluation

This chapter presents the performance evaluation of the different architectures of the accelerator and its blocks described in detail in Chapter 6. Execution time and power measurements for all blocks separately and for the different versions of the integrated designs are reported together with their resources utilization. Additionally, an accuracy evaluation is performed for the seeding workflow applied. All software measurements were taken using a Intel XeonSilver 4210 CPU working at 2.20GHz, while as previously referred to in Chapter 4 the FPGA board used to build and run the proposed designs on is an AMD Alveo U200. The reference genome graph is the human chromosome 22 genome graph produced by GRCh38, as a starting reference genome, and 7 additional Variant Calling Files (VCFs) for HG001-007 from the GIABproject (v3.3.2). Reads come from the Illumina short read dataset including 10k 100bp reads. To reference the index and to extract minimizer's from the reads we use 15bp minimizers, window size equal to 20, we set edits distance threshold equal to two and hop limit equal to 15. The size of the indexed reference graph is 224.94MB.

1 Seeder Evaluation

1.1 Seeding Accuracy

Since we propose an altered version of seeding, keeping the main steps of the seeding workflow the same, making design choices on them based on profiling and inserting some additional steps or constrains as described in detail in Chapter 6, it is essential to evaluate the accuracy of our Seeder design using the highly accurate software tool GraphAligner, to perform alignment of the read dataset to the graph and then use its final alignment results to proof check our seeder's accuracy. More specifically, since the seeding step is expected to find more than one candidate alignment positions, we consider an accurate seeding one that includes the coordinates of the final alignment of that specific read as given by GraphAligner. GraphAligner outputs 1.0119 final alignments per alignment while our integrated tool finds more than one accepted final align-

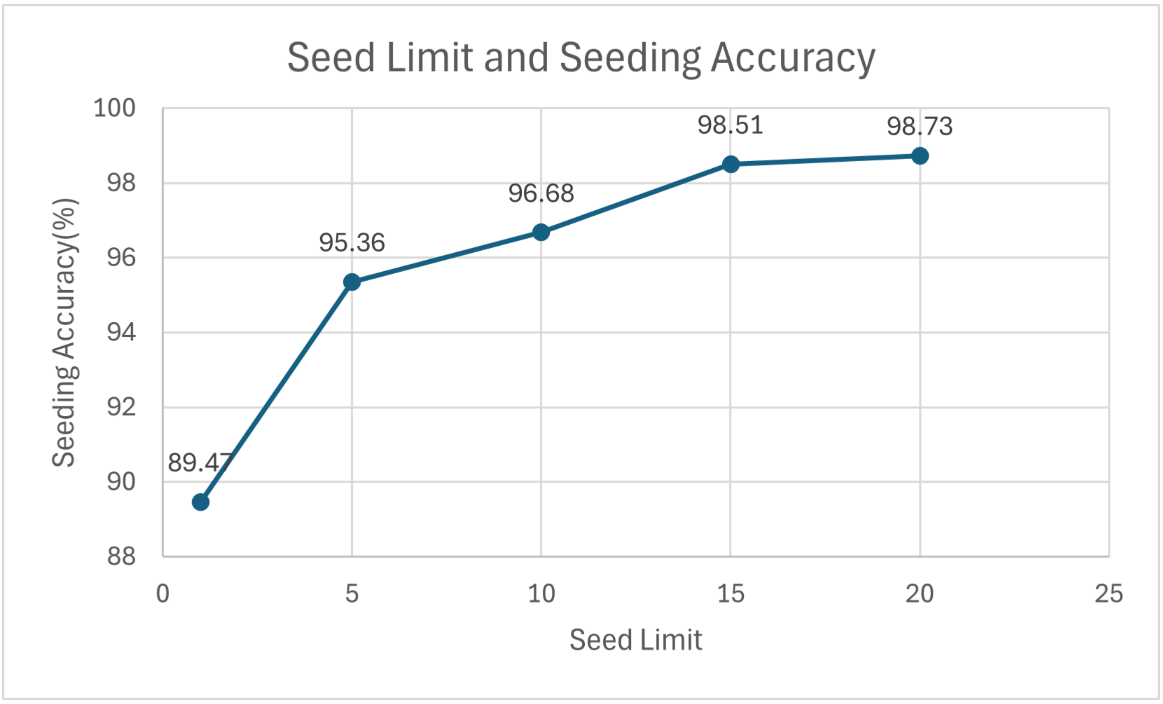


Figure 7.1: Proposed Seeding Method Accuracy Evaluation based on GraphAligner Results

ment positions, since our design does not have a seed scoring and filtering step as GraphAligner does. We examine seeding accuracy changes in regards to the seed limit we impose during seed hit finding to be able to reduce the size of arrays used and thus resource utilization in hardware terms. As seen in Figure 7.1 accuracy is increased with the increase of the seed limit, as naturally more seed hits are found. It is also evident that having a low seed limit does not decrease accuracy critically, even though in these types of applications high accuracy close to 100% is needed, meaning that we could decide to lower the seed limit in case we run short of resources, trading off the system's accuracy. Finally, it is expected to have high seeding accuracies for seed limits lower than five for example, since as seen in Figure 5.10 most minimizers are found in less than five positions in the reference graph, thus most minimizers are expected to have less than five seed hits as also seen during this evaluation step.

1.2 Performance, Utilization and Power

The next step is to evaluate the performance of our Seeder block as a standalone block. The first thing we examine is how execution time per read changes by introducing a read scratchpad, which is expected to reduce execution time since less memory accesses are needed for performing seeding for the overall dataset. Indeed, as seen in Figure 7.2, hardware execution time per read decreases as the size of the read scratchpad increases, speeding up the no scratchpad version performance by a factor of 3.8. It is also shown that there is an infimum to the values of the execution time as each time we increase the scratchpad size the difference in time is smaller than previously (e.g. going from no scratchpad to a 2-read scratchpad reduces time approximately by half, while going from a 4-read scratchpad to a 5-read scratchpad has a factor change equal to 1.2).

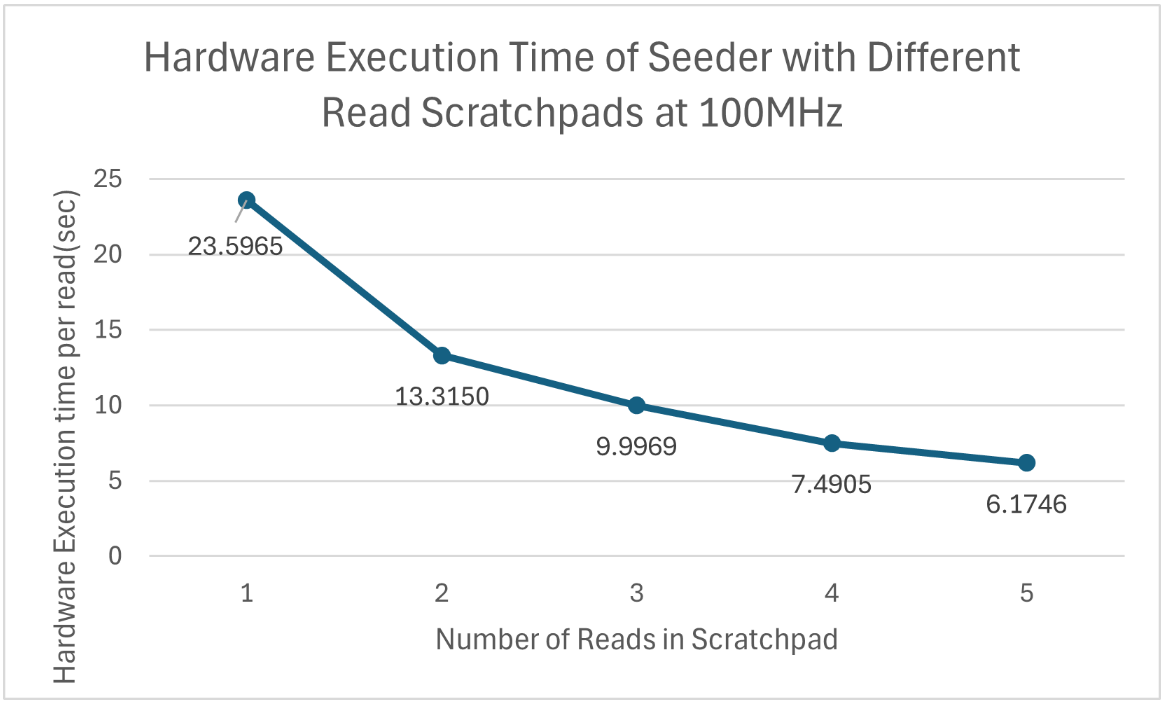


Figure 7.2: Hardware Performance and Read Scratchpad's Size Relationship

In order to examine the tradeoff between resources utilization and performance when increase the read scratchpad size. As seen in Table 7.1 changes in utilization percentages are not that significant (apart from the BRAM utilization that increases by 7% which is due to the increase of the local arrays acting as seed scratchpads).

Utilization (%) / Seeder	BRAM_18K	DSP	FF	LUT	URAM
Seeder without Read Scratchpad	49	3	12	51	7
Seeder with 4-read Scratchpad	56	3	13	52.18	7
Seeder with 5-read Scratchpad	56	3	13	52.71	7

Table 7.1: Seeder Resource-Utilization percentages for Different Read Scratchpad Sizes

Since the version of the Seeder with the 5-read scratchpad gives the best performance without a great increase in the utilization percentages, we choose to examine it in higher frequencies and compare it with the no scratchpad version of the Seeder. As seen in the histogram of Figure 7.3 all hardware execution times are lower than the baseline software performance. Additionally, it is observed that doubling the frequency of operation of the seeder without scratchpad decreases by half initial hardware execution time at 100 MHz giving a hardware speedup of 1.86×. The version of the seeder with the 5-read scratchpad works properly up to 150MHz and results in a speedup of 4.92 × to the seeding software version. The same hardware version of the seeder working at 100 MHz accelerates seeding 3.91×.

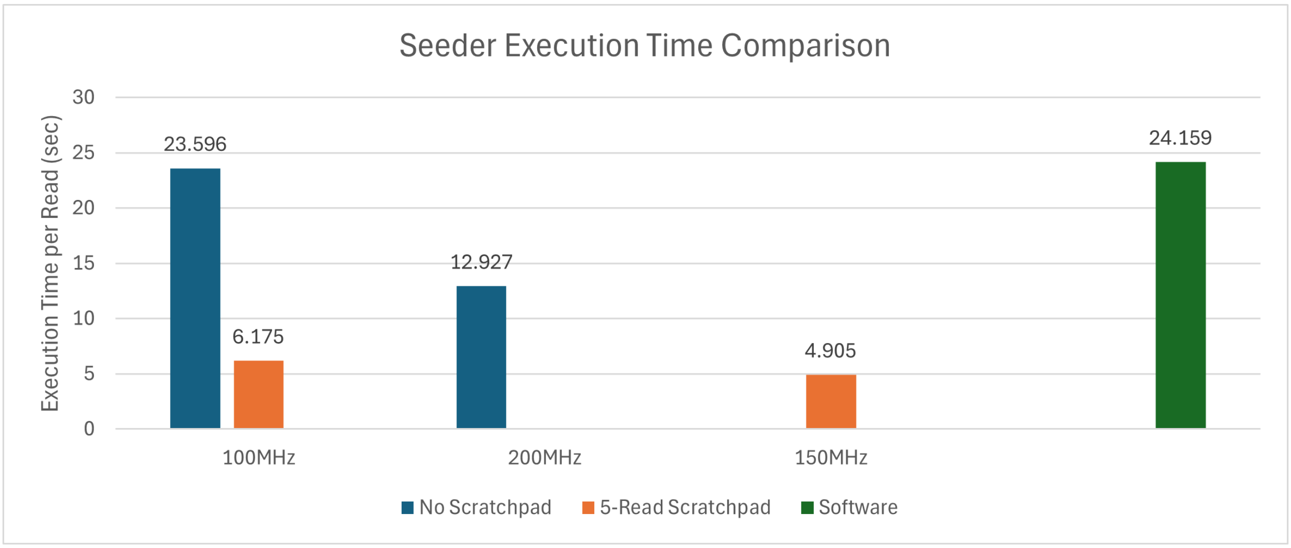


Figure 7.3: Execution Time Comparison of Seeder Block with and without Scratchpad at Different Frequencies

Additionally, we calculate execution cycles by multiplying execution times with the respective operating frequencies (100,150,200 MHz for the FPGA implementations and 2201 MHz for the CPU one). In Figure 7.4 one can see the results of the cycles in a logarithmic scale. It is evident as expected that CPU cycles are much higher than the FPGA ones, due to its significantly higher operating frequency. The lowest number of cycles is achieved by the 5-read scratchpad version operating at 100MHz, being 86.11 times lower than CPU cycles.

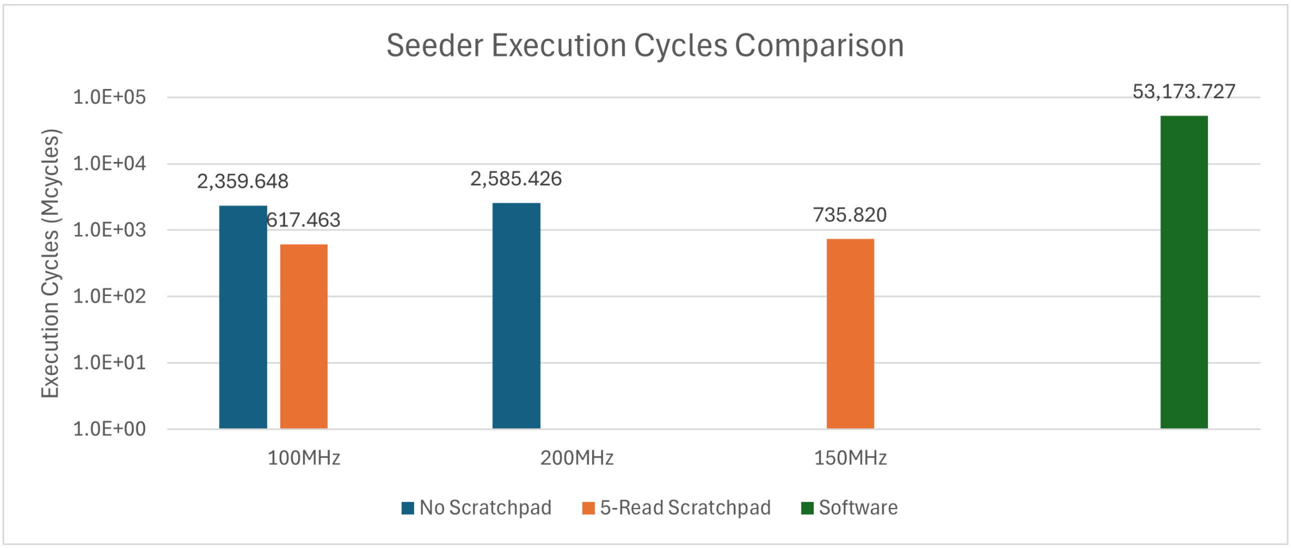


Figure 7.4: Execution Cycles Comparison of Seeder Block with and without Scratchpad at Different Frequencies

In terms of power consumption, the seeder kernel with no read scratchpad consumes 1.319W at 100MHz and 2.654 at 200MHz, approximately two times higher as expected, since P $\propto$ f. It is also interesting to examine the relation in between power consumption and the size of the read scratchpad. Figure 7.5 shows that power consumption slightly increases when the size of the read scratchpad is increased.

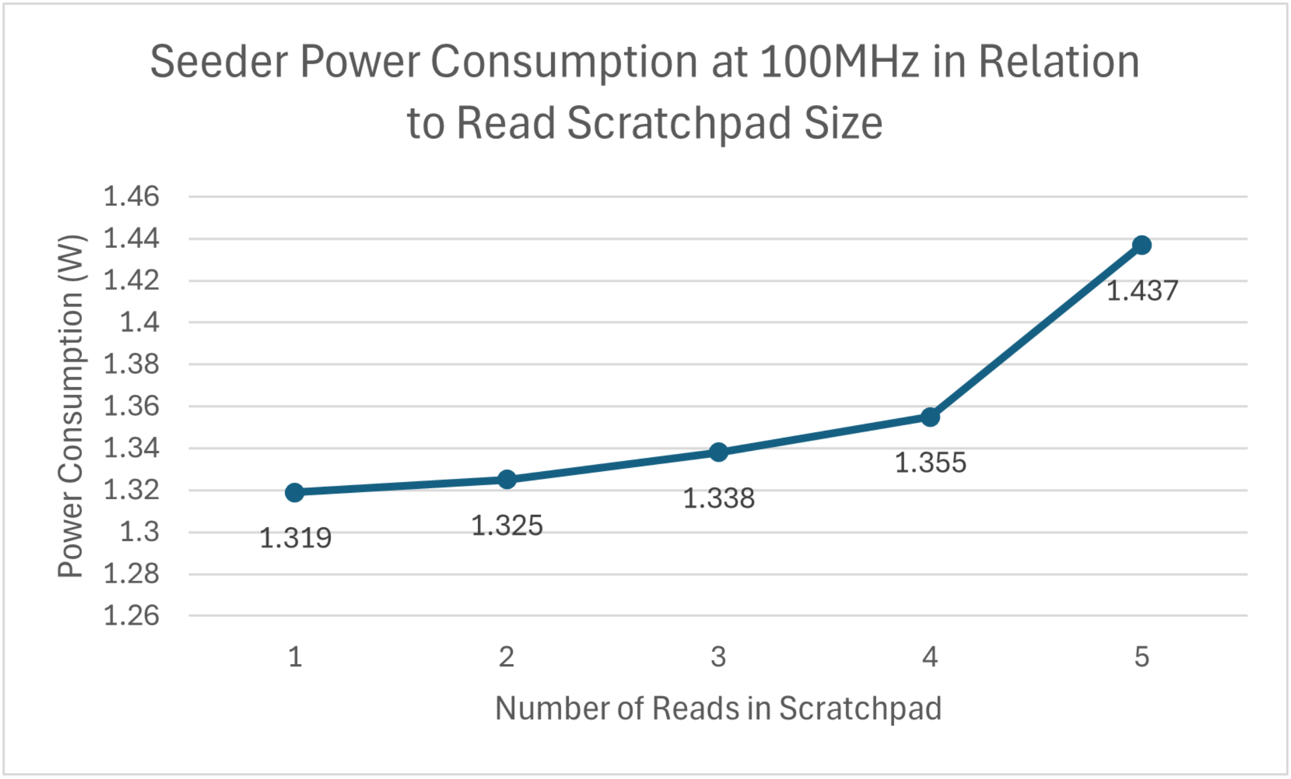


Figure 7.5: Relation of Power Consumption of Seeder Kernel and Read Scratchpad Size at 100MHz

The small changes of power and utilization with the increase of the read scratchpad size together with the increase in performance show that to get our Seeder's block optimal performance we do not in fact make great sacrifices in power and utilization.

2 Linearization and Sequence-to-Sequence Aligner vs Sequence-to-Graph Aligner

This subsection is dedicated to comparing the performances of the two alignment methods used to align a read to a candidate subgraph as extracted by the seeding step. The first method (Design 1), is the one presented in Figure 6.12 where the linearization and alignment blocks are merged into one since linearization is needed to perform the sequence-to-sequence (S2S) alignment, while the second is the stand alone sequence-to-graph block (Design 2) of Figure 6.16. The designs were build for two frequencies, at 100 and 200 MHz and the execution time measurements are given in Figure 7.6. As seen in the diagram, we make a distinction between alignment of a read with a single node candidate alignment subgraph, and a multi node one, since for both cases of aligner, aligning a sequence to a single node subgraph is pure sequence-to-sequence alignment and is implemented the same way. The difference between the two approaches is detected in the way a read is aligned to a subgraph with multiple nodes, since then hops within the subgraph should be considered. The first implementation finds the paths of the subgraph and then performs sequence-to-sequence alignment while the second one bypasses the path finding step and run a sequence-to-graph alignment algorithm that calculates intermediate alignment bitvectors as explained in Chapter 6 for each node only once, reducing overhead. Therefore, we expect, as it also shown by the measurements, that performance differences for the entire dataset lie on the different way the designs handle multiple node alignment subgraphs. As seen in the histograms of Figure 7.6, the software implementation has a great latency in manipulating alignments of reads to subgraphs with multiple nodes while even for the accelerated cases of the FPGA's measurements of the linearization and S2S design, multi-node subgraph alignments (first group) take more time than alignments with single node subgraphs (second group). In fact, in the case of the 100 MHz clock, the first group of alignments is almost 1.8× slower than the second group, while for the 200 MHz clock, approximately 1.3× slower. The decrease in the execution time of aligning the first group lies on the fact that, even though the design is memory bound (as expected and proven by the fact that there is no significant decrease in execution time), increase in the clock frequency increases more the linearization block performance (especially path finding), since we do not see any significant decrease in execution time when comparing second groups alignment time in between the two clock frequencies (thus the aligner block's performance is not really affected). However, it is noticeable that both designs achieve hardware speedup* for the entire dataset equal to 7.67× for the case of the linearization+S2S aligner and 9.9× for the case of the second design, both working at 100 MHz. Respectively, when operating at 200 MHz, the speedups slightly increase as expected:8.02× and 10.48× for Designs 1 and 2 respectively. The hardware speedups when examining only the first group of the dataset are even greater reaching a maximum of 17.08× for Design 1 and 26.28× for Design 2, showing that the proposed designs and especially Design 2 could accelerate alignment of entire datasets of longer reads which mostly are aligned to multiple node subgraphs due to their length. As far as the reads of the second group are concerned, alignment time is more like the same for both designs and examined frequencies (due to the memory bound nature of the application) reaching a speedup of 3.92× in respect to that group's software version.

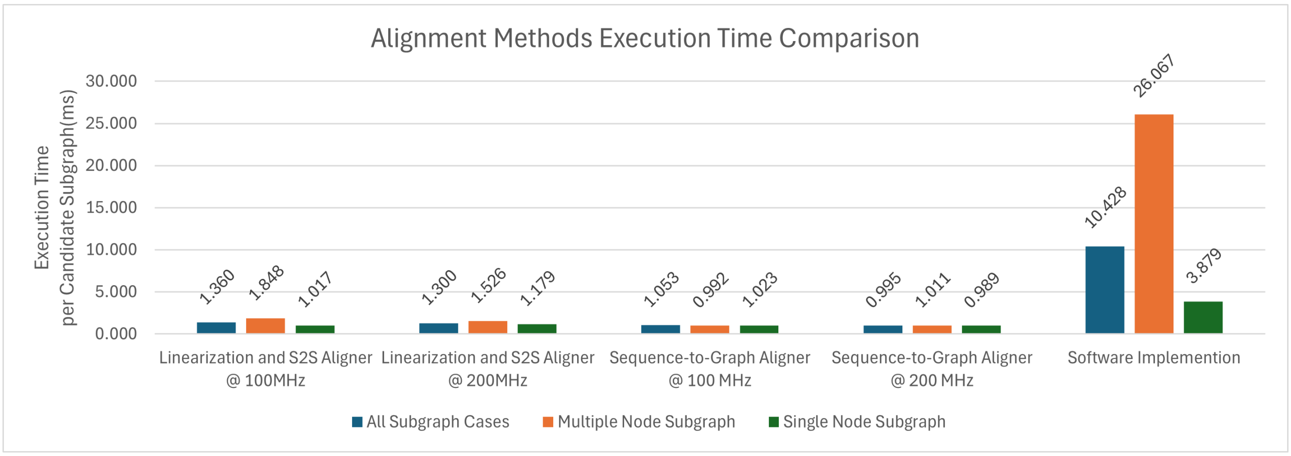


Figure 7.6: Execution Time Comparison of the Two Methods Used for Alignment

Additionally to the pure execution time measured above, we also calculate execution cycles shown in Figure 7.7, by multiplying execution time with the respective frequencies: 100 and 200 MHz for the FPGA implementations respectively and 2201 MHz for the CPU implementation. As expected, software execution cycles are more than 150× the cycles of the slowest design (Design 1) at 100 MHz for the entire dataset. Measurements and observations are analogous to the ones of the pure execution time analysis.

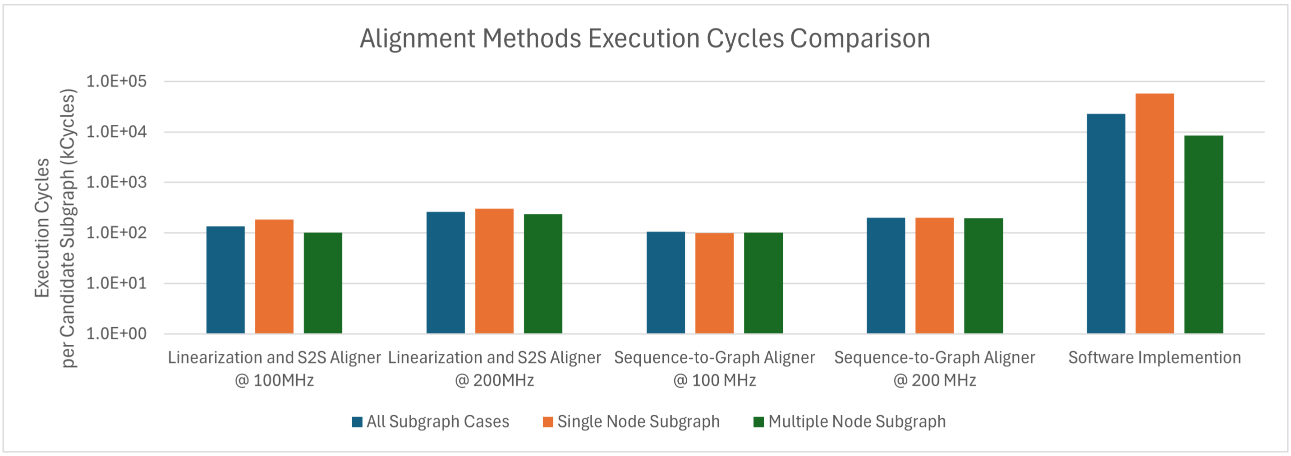


Figure 7.7: Execution Cycles Comparison of the Two Methods Used for Alignment in a Logarithmic Scale

Finally, it is worth noticing that execution time and cycles measurements are closer to the results of second group of the dataset rather than the mean values of the two groups and that is why the short read datasets naturally have more reads aligned to sequences found within a single node than a path within the graph due to their short length. More specifically, the pie chart of Figure

[0]* We calculate Hardware speedup as the quotient of the Software execution time to the Hardware execution time.

7.8 shows the percentages of occurrence of each case of subgraph extension as described in Section 1.4. As shown and expected, most of the seed hits do not require extension since an adequate number of characters is found within their node.

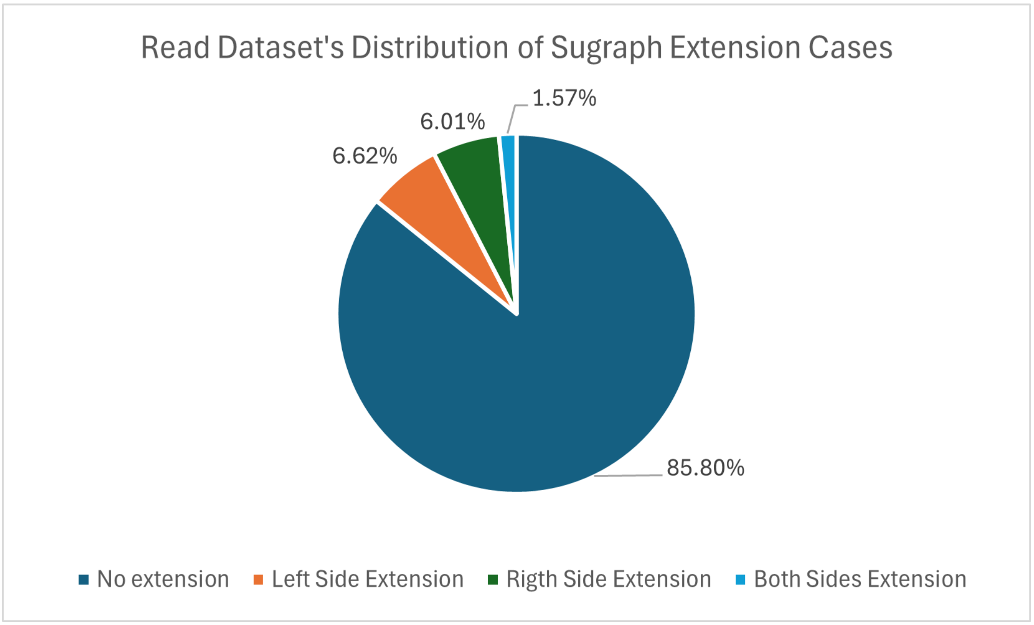


Figure 7.8: Subgraph Extension Cases Profiling in Short Read Dataset

In terms of utilization,both designs have low utilization percentages with a noticeable percentage only in the case of LUTs. Some BRAMs are used as local scratchpads to store the retrieved data from memory (e.g.outNodes tables), while the traceback arrays (Rall) are mapped to URAMs for both designs. There is a slightly higher percentage of LUTs for the Sequence-to-Graph design, which is expected due to the filtering step.

Utilization (%) / Alignment Block Type	BRAM_18K	DSP	FF	LUT	URAM
Linearization with S2S Aligner	~0	~0	1	17	1
Sequence-to-Graph Aligner	~0	~0	1	20	1

Table 7.2: Aligner Designs' Resource-Utilization percentages per SLR

3 Integrated Designs Evaluation

There are two version of the end-to-end system that takes reads as inputs and finds their alignments (CIGARs and alignment positions): Design 1 (Figure 6.12) and Design 2 (Figure 6.17), differing in the method of alignment they use. In this section we evaluate the performance of both these systems and compare them with one another.

3.1 Integrated System with Linearization and Sequence-to-Sequence Aligner

Firstly, we examine performance of Design 1 which has more variations due to the existence of read scratchpads of various sizes. As we are going to see further down, we cannot incorporate a read scratchpad in Design 2 having it in one single SLR, due to resources limitations (this was expected, as Design 1 uses almost all available LUTs to implement the 5-read scratchpad and the S2G Aligner used in Design 2 has slightly more LUT utilization percentage than the Linearization+S2S Aligner block of Design 1). Figure 7.9 shows a comparative diagram of the execution times per read of different hardware variations of Design 1.

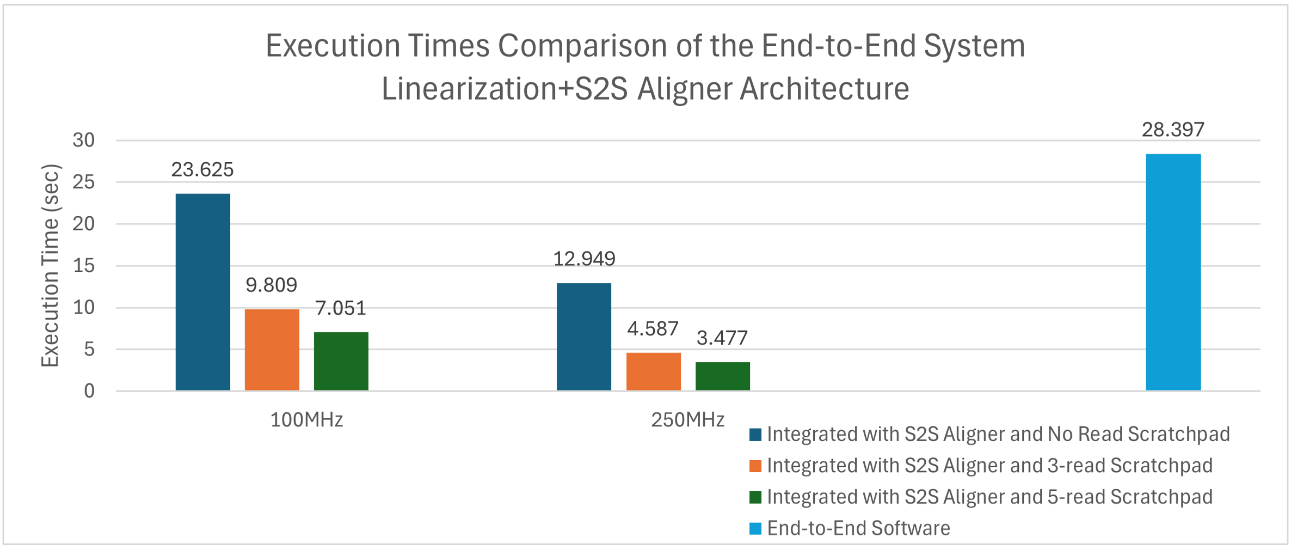


Figure 7.9: End-to-End Design with Linearization and S2S Execution Time

All of them win over the baseline software implementation with hardware speedups ranging from 1.20× (variation without read scratchpad at 100 MHz) to 8.16× (variation with largest possible read scratchpad at 200 MHz). We observe that execution times are closer in values with the Seeder's execution times (Figure 7.3) which is expected since as we saw the linearization and S2S Aligner part has significantly lower execution time than the Seeder which is the most memory bound block that performs most accesses to memory. Thus, it is expected for the performance of the Seeder to be characterizing of the performance of the integrated system regardless of the alignment method which might alter the results slightly but certainly not significantly (as we saw the aligners' performances are also memory bound). We proceed by calculating execution cycles for the different designs and operations frequencies as previously. Again, the software implementation requires 88.64 Mcycles times more than the hardware version with the largest scratchpad working at 100 MHz.

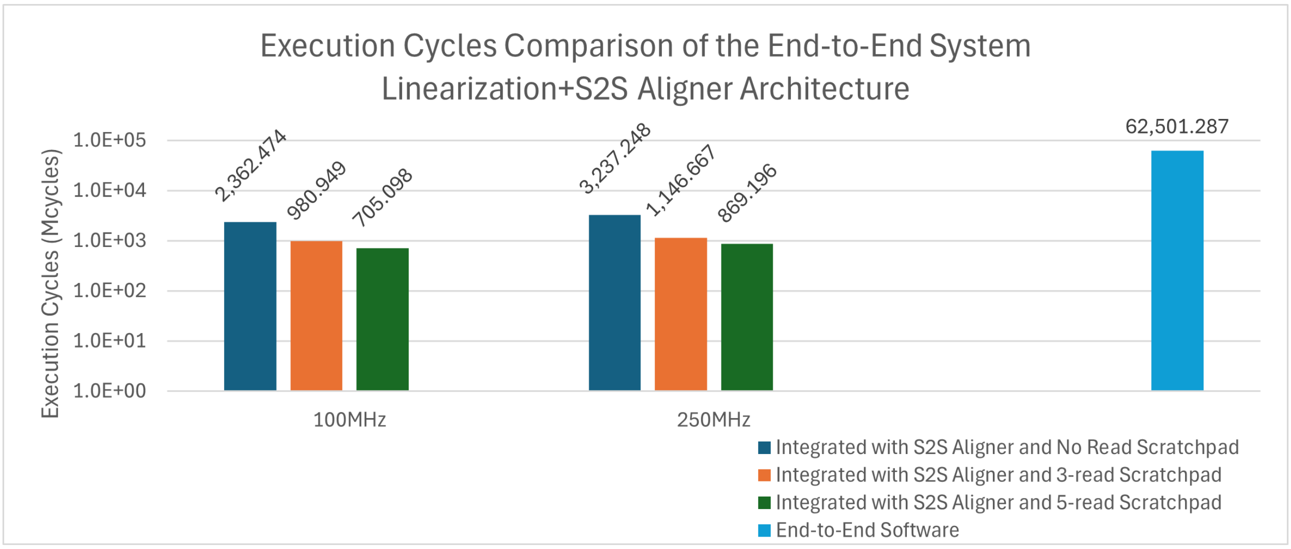


Figure 7.10: End-to-End Design with Linearization and S2S Execution Cycles in Logarithmic Scale

Table 7.3 shows the SLR's resources utilization percentages. It is observed that increase in the size of the read scratchpad increases LUT utilization and redistributes the way other resources are allocated and used (e.g. in the case of the 5-read scratchpad some arrays are mapped to BRAMs instead of URAMs as it was done for the 3-read scratchpad design).

Utilization (%) / Integrated Design with S2S Aligner	BRAM_18K	DSP	FF	LUT	URAM
No Scratchpad	35	3	16	93	9
3-Read Scratchpad	32	3	16	98	13
5-Read Scratchpad	47	0	17	99,69	9

Table 7.3: Aligner Designs' Resource-Utilization percentages per SLR

3.2 Integrated System with Sequence-to-Graph

As expected for both end-to-end designs execution time and cycles are closer to the Seeder's block for reasons explained above. We do see however a slight increase in performance for Design 2 that is due to the Sequence-to-Graph Aligner. After all, alignment times as shown above, are in the millisecond scales while seeding times in seconds, meaning that small differences that seem negligible are not in fact especially when it comes to using the designs in datasets with longer reads or reads with multi node alignments. However, there is still a maximum hardware speedup of 2.19× for Design 2 working at 200 MHz.

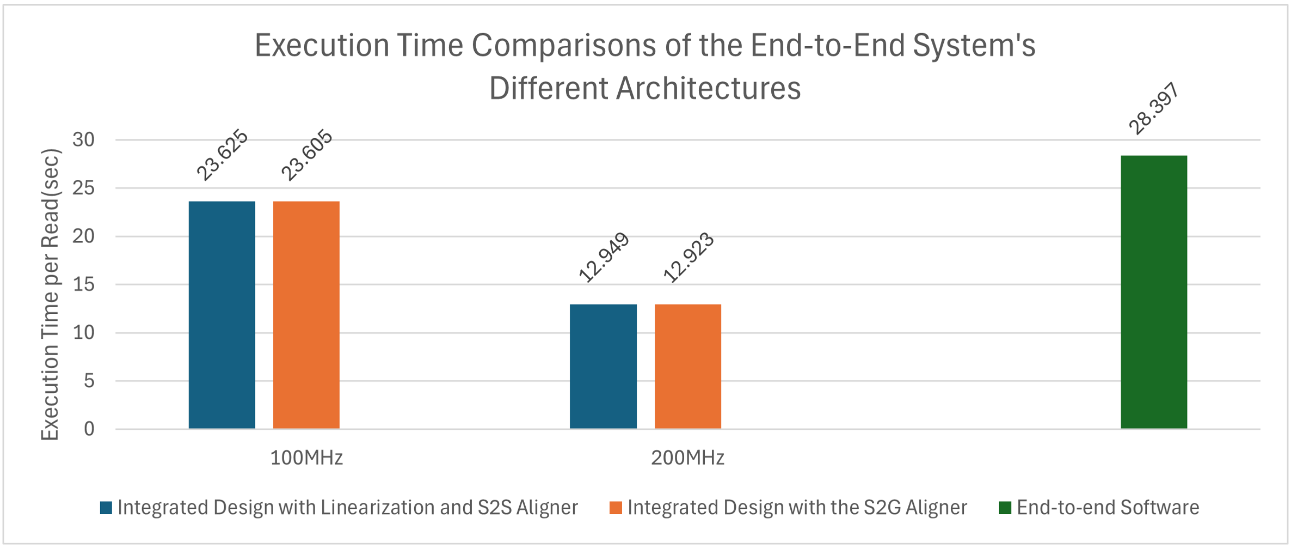


Figure 7.11: Comparison of End-to-End Design 1 and 2 in terms of Execution Time

Additionally for the cycles, which demonstrate the differences in performances due to the different aligner architecture even better, baseline end-to-end software requires 24.181× more cycles than Design 2 implementation running at 200 MHz.

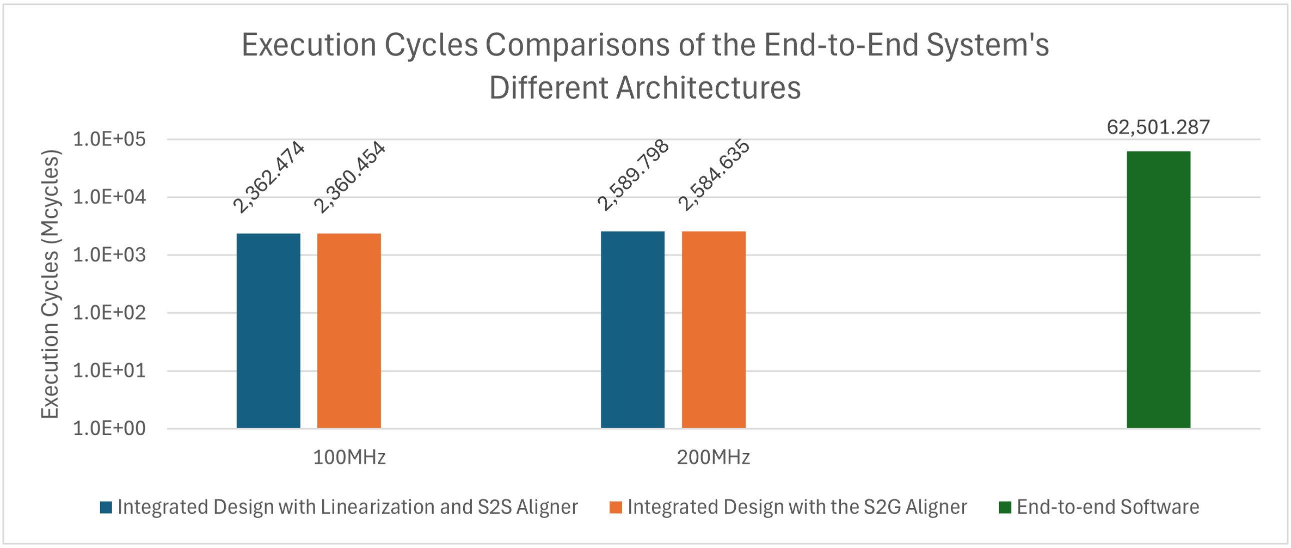


Figure 7.12: Comparison of End-to-End Design 1 and 2 in terms of Execution Cycles

Table 7.4 shows the utilization percentages of Design 2. It would have been ideal to be able to use a read scratchpad for Design 2, since we are expecting to see an even better performance than the respective one of Design 1 with a read scratchpad (since seeding performance would increase). However, due to the increased aligner resource utilization percentages of Design 2 compared to Design 1 (see Table 7.3), we cannot build (in one SLR) and evaluate the end-to-end design with a S2G aligner block and a read scratchpad which we would have expected to have the best performance. Finally, we measured power consumption of the kernels of the two end-to-end designs and show that the S2G

Integrated Design with S2G Aligner \ Utilization (%)	BRAM_18K	DSP	FF	LUT	URAM
No Scratchpad	54	0	15	97	31
2-read Scratchpad	66	0	18	120	31

Table 7.4: Aligner Designs' Resource-Utilization percentages per SLR

Aligner design has slightly higher power consumption, which might also be connected with the increased utilization and the more complicated operations it performs during S2G alignment, especially when hops are involved, while the S2S aligner bypasses them due to the path finding module that is basically implemented as a for loop.

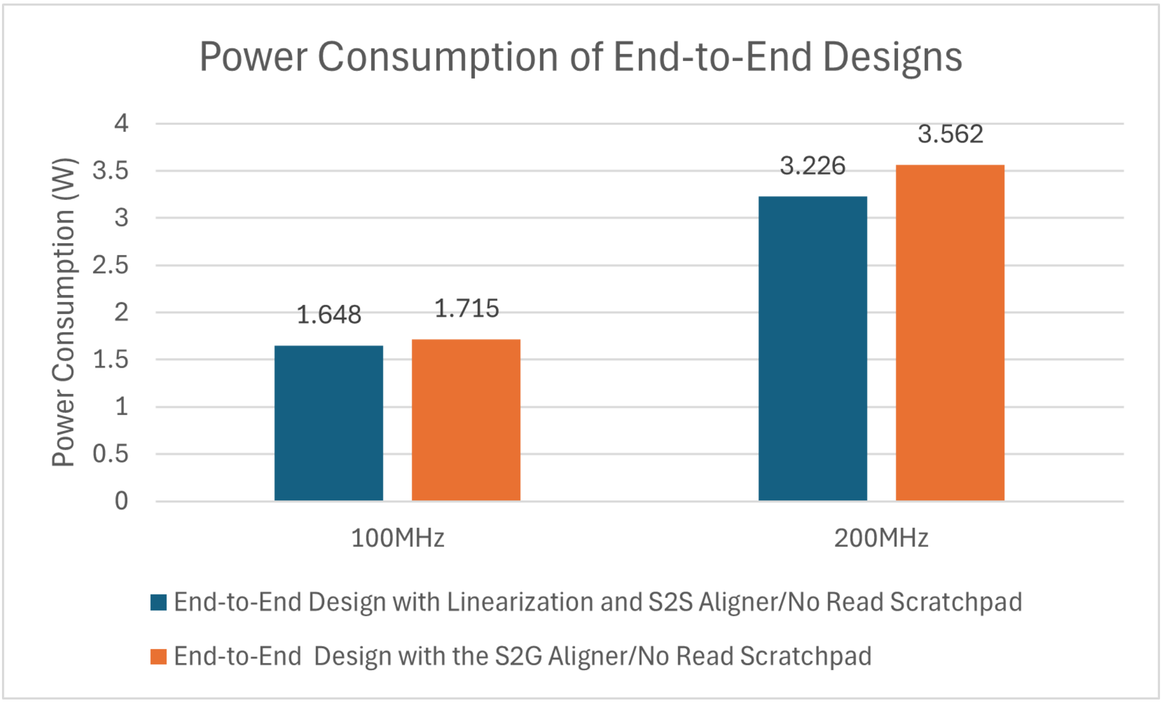


Figure 7.13: Power Consumption of End-to-End Designs

Chapter 8

Conclusion & Future Work

In the era of high throughput sequencing when millions of DNA molecules are being processed and sequenced, the need for accelerating genome analysis algorithms and workflows while maintaining low error rates, is greater than ever. Additionally, the available amount of DNA reference genomes and assemblies works itself as a tool to remove DNA mapping and alignment biases by enriching reference genomes and creating novel ways of representation to enclose all this information. This thesis has focused on examining and finding ways to take advantage of graph representations of genomes, while aiming at proposing a workflow custom architectures to accelerate sequence-to-graph genome mapping and alignment on FPGAs. We have proposed, implemented and evaluated in terms of accuracy and performance a seeding stand alone block which can also be used as part of an integrated end-to-end mapping and alignment accelerator. In this work we have designed and implemented two of such integrated designs : one deducing the sequence-to-graph alignment task to sequence-to-sequence and one performing actual sequence-to-graph alignment, both utilizing the Bitap approximate string matching algorithm. Even though reference graph manipulation can be very challenging and the sequence-to-graph mapping and alignment problem is considered hard to accelerate on hardware due to seeding's memory intense nature, subgraph extraction's recursive approach and alignment's computational demands, we manage to achieve a speedup of X8.16 when comparing to our baseline end-to-end software implementation. Furthermore, we showed that the two approaches when used in an integrated system give similar performances especially for short read datasets.

As part of the future work, we will evaluate the proposed designs in long read datasets for which we expect to better demonstrate the advantages of sequence-to-graph alignment when it comes to the alignment step. We also need to expand our implementation to be able to have more than one reference graphs of various sizes instead of having only one reference graph loaded in the main memory as we have done till now. It would also be a good idea to check our seeder's accuracy for different lengths of reference minimizers and edit distances. Additionally, we will need to optimize the designs even further by using the entire FPGA area and not just one of the three available SLRs, especially to implement the end-to-end design with bigger read scratchpads and further optimizations for both the proposed designs. Finally, as far as the proposed designs as they are now are concerned, the searching process should become more efficient in the future so that the seeding step accesses specific memory addresses of the seeds instead of searching throughout regions of the reference which is inefficient in terms of resources and creates additional latency.

As an extension to the proposed architectures, we could design a pre-alignment subgraph filter to discard candidate alignment subgraphs preventing them from entering the aligner. Such a filter could be implemented based on a base counting logic, meaning to reference the graph so as to keep the number of each character of the DNA alphabet within each node and then during filtering count the occurrence of each characters within the subgraph having also counted their numbers on the read sequence. If the subgraph does not have enough of one of the DNA alphabet characters to align the read, then it will be discarded.

An alternative approach to solving the problem, involving accelerating sequence/subgraph similarity calculation rather than performing alignment which could also be used in other types of applications involving graphs with evolutionary information, multi-species genome graphs etc. Apart from accelerating similarity scoring algorithms, another design could involve machine learning and specifically Graph Neural Networks (GNNs) to extract embeddings from the reference graph and index it in this way. What would also be very interesting would be to try and reduce the bottleneck caused by memory intensity, by building an in-memory processing accelerator to perform some parts or the entire workflow of the application.

Finally, we could also work on accelerating different types of biological applications involving graphs, such as de novo genome assembly through building De Bruijn graph assemblers. Another type of application would be to accelerate to accelerate alignment of different types of sequences (e.g. DNA, RNA, proteins) to a mega graph representing the entire metabolic state of a cell to characterize it.

Bibliography

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

[2] M. Alser, J. Lindegger, C. Firtina, N. Almadhoun, H. Mao, G. Singh, J. Gomez-Luna, and O. Mutlu, "From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures," *Computational and Structural Biotechnology Journal*, vol. 20, pp. 4579–4599, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2001037022003531

[3] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, Sep. 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty191

[4] N. M. Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. S. Cali, C. Firtina, H. Mao, N. A. Alserr, R. Ausavarungnirun, N. Vijaykumar, M. Alser, and O. Mutlu, "Genstore: A high-performance and energy-efficient in-storage computing system for genome sequence analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22).* ACM, 2022. [Online]. Available: https://doi.org/10.1145/3503222.3507702

[5] 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, Apr. 2021. [Online]. Available: https://doi.org/10.1093/bioinformatics/btaa1015

[6] E. Garrison, J. Sirén, A. Novak *et al.*, "Variation graph toolkit improves read mapping by representing genetic variation in the reference," *Nature Biotechnology*, vol. 36, no. 9, pp. 875–879, 2018. [Online]. Available: https://doi.org/10.1038/nbt.4227

[7] D. Senol 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," in *Proceedings of the 53rd International Symposium on Microarchitecture (MICRO)*, Oct. 2020, arXiv preprint arXiv:2009.07692, submitted September16,2020. [Online]. Available: https://arxiv.org/abs/2009.07692

[8] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingol, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. S. Kim, N. M. Ghiasi, G. Singh, J. Gomez-Luna, N. A. Alserr, M. Alser, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "Segram: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*, New York City, NY, USA, Jun. 2022.

[9] M. Alser, J. Lindegger, C. Firtina, N. Almadhoun, H. Mao, G. Singh, J. Gómez-Luna, and O. Mutlu, "From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures," *Computational and Structural Biotechnology Journal*, vol. 20, pp. 4579–4599, Aug. 2022, invited review. [Online]. Available: https://doi.org/10.1016/j.csbj.2022.08.019

[10] M. Rautiainen and T. Marschall, "Graphaligner: rapid and versatile sequence-to-graph alignment," *Genome Biology*, vol. 21, no. 1, p. 253, 2020. [Online]. Available: https://doi.org/10.1186/s13059-020-02157-2

[11] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009. [Online]. Available: https://doi.org/10.1186/gb-2009-10-3-r25

[12] N. Ahmed, J. Lévy, S. Ren, B. Berger, R. Poplin, and S. Batzoglou, "GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data," *BMC Bioinformatics*, vol. 20, no. 1, p. 520, 2019. [Online]. Available: https://doi.org/10.1186/s12859-019-3086-9

[13] S. Park, J. Hong, J. Song, H. Kim, Y. Kim, and J. Lee, "AGAThA: Fast and Efficient GPU Acceleration of Guided Sequence Alignment for Long Read Mapping," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Edinburgh, United Kingdom: Association for Computing Machinery, Mar. 2024, pp. 431–444, originally submitted as a preprint on arXiv:2403.06478 on March11,2024.

[14] K. Koliogeorgi, S. Xydis, G. Gaydadjiev, and D. Soudris, “Gandafl: Dataflow acceleration for short read alignment on ngs data,” *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 3018–3031, 2022.

[15] S. S. Banerjee, M. El-Hadedy, J. B. Lim, Z. T. Kalbarczyk, D. Chen, S. S. Lumetta, and R. K. Iyer, “ASAP: Accelerated Short-Read Alignment on Programmable Hardware,” *IEEE Transactions on Computers*, vol. 68, no. 3, pp. 331–346, Mar. 2019. [Online]. Available: https://doi.org/10.1109/TC.2018.2875733

[16] Y. Zhang, D. Chen, G. Zeng, J. Zhu, Z. Li, L. Chen, S. Wei, and L. Liu, “Harp: Leveraging quasi-sequential characteristics to accelerate sequence-to-graph mapping of long reads,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’24), Volume 3*. ACM, 2024, pp. 512–527. [Online]. Available: https://doi.org/10.1145/3586716.3586731

[17] National Human Genome Research Institute, “A brief guide to genomics,” 2024, accessed: 2025-07-03. [Online]. Available: https://www.genome.gov/about-genomics/fact-sheets/A-Brief-Guide-to-Genomics

[18] T. Shahroodi, G. Singh, M. Zahedi, H. Mao, J. Lindegger, C. Firtina, S. Wong, O. Mutlu, and S. Hamdioui, “Swordfish: A Framework for Evaluating Deep Neural Network-based Basecalling using Computation-In-Memory with Non-Ideal Memristors,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3613424.3614252

[19] H. Mao, M. Alser, M. Sadrosadati, C. Firtina, A. Baranwal, D. Senol Cali, A. Manglik, N. Almadhoun Alserr, and O. Mutlu, “GenPIP: In-Memory Acceleration of Genome Analysis via Tight Integration of Basecalling and Read Mapping,” in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1–13. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00056

[20] C. Firtina, M. Soysal, J. Lindegger, O. Mutlu, M. Alser, M. B. Cavlak, H. Mao, G. Singh, N. M. Ghiasi, F. Eris, A. Kahles, S. Wong, and S. Hamdioui, “RawHash2: Mapping Raw Nanopore Signals Using Hash-Based Seeding and Adaptive Quantization,” *Bioinformatics*, vol. 40, no. 8, p. btae478, 2024. [Online]. Available: https://doi.org/10.1093/bioinformatics/btae478

[21] C. Firtina, M. Mordig, H. Mustafa, S. Goswami, N. M. Ghiasi, S. Mercogliano, F. Eris, J. Lindegger, A. Kahles, and O. Mutlu, “Rawsamble: Overlapping and Assembling Raw Nanopore Signals using a Hash-based Seeding Mechanism,” *arXiv*, vol. abs/2410.17801, Oct. 2024. [Online]. Available: https://arxiv.org/abs/2410.17801

[22] J. S. Kim, D. S. 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. 1, p. 295, 2018. [Online]. Available: https://doi.org/10.1186/s12864-018-4460-0

[23] R. Poplin, P.-C. Chang, D. Alexander, S. Schwartz, T. Colthurst, A. Ku, D. Newburger, J. Dijamco, N. Nguyen, P. T. Afshar, S. S. Gross, L. Dorfman, C. Y. McLean, and M. A. DePristo, “A universal snp and small-indel variant caller using deep neural networks,” *Nature Biotechnology*, vol. 36, no. 10, pp. 983–987, 2018. [Online]. Available: https://doi.org/10.1038/nbt.4235

[24] V. Kamaraj, A. Gupta, M. Narayanan, K. Raman, and H. Sinha, “Unveiling genomic complexity: A framework for genome graph structural analysis and optimised variant calling workflows,” *bioRxiv*, vol. 2024, no. 598220, 2024. [Online]. Available: https://doi.org/10.1101/2024.06.10.598220

[25] P. Coussy and A. Morawiec, *High-Level Synthesis: from Algorithm to Digital Circuit*, ser. The Springer International Series in Engineering and Computer Science. Springer, 2008, vol. 497. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4020-8588-8

[26] I. Damaj, “High-level synthesis,” *Encyclopedia of Computer Science and Engineering*, 2008. [Online]. Available: https://doi.org/10.1002/9780470050118.ecse177

[27] AMD, “Vitis high-level synthesis user guide (ug1399): Additional resources and legal notices,” 2025, accessed: 2025-07-03. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Additional-Resources-and-Legal-Notices

[28] X. Inc., “Alveo u200 and u250 data center accelerator cards data sheet (ds962),” https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf, 2019.

[29] I. Xilinx, *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs*, 2021.

[30] Xilinx, “Stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency,” Xilinx, Tech. Rep. WP380, 2012.

[31] I. Xilinx, “Vitis hls llvm source code and examples,” https://github.com/Xilinx/HLS, 2024.