UAB
Universitat Autònoma de Barcelona

# Accelerating Sequence Alignment on GPUs using the Wavefront Algorithm

**Quim Aguado Puig**

**ADVERTIMENT.** L'accés als continguts d'aquesta tesi queda condicionat a l'acceptació de les condicions d'ús establertes per la següent llicència Creative Commons:  https://creativecommons.org/licenses/?lang=ca

**ADVERTENCIA.** El acceso a los contenidos de esta tesis queda condicionado a la aceptación de las condiciones de uso establecidas por la siguiente licencia Creative Commons:  https://creativecommons.org/licenses/?lang=es

**WARNING.** The access to the contents of this doctoral thesis it is limited to the acceptance of the use conditions set by the following Creative Commons license:  https://creativecommons.org/licenses/?lang=en

UAB
Universitat Autònoma
de Barcelona

Department of Computer Architecture & Operating Systems
Escola d'Enginyeria

# Accelerating Sequence Alignment on GPUs using the Wavefront Algorithm

*A thesis submitted by **Quim Aguado Puig** in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science (Research line High Performance Computing) from Universitat Autònoma de Barcelona, advised by Dr. Santiago Marco-Sola and Dr. Juan Carlos Moure.*

Barcelona, September 9, 2026

The thesis titled **"Accelerating Sequence Alignment on GPUs using the Wavefront Algorithm"** submitted by **Quim Aguado Puig** in fulfillment of the requirement for the PhD degree from Universitat Autònoma de Barcelona. This doctoral thesis belongs to the Computer Science doctoral program (Research line High Performance Computing) and was presented to the Department of Computer Architecture & Operating Systems at the Escola d'Enginyeria of Universitat Autònoma de Barcelona, under the supervision of Dr. Santiago Marco-Sola and Dr. Juan Carlos Moure.

This research has been supported by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of the total cost eligible under the DRAC project [001-P-001723], the Spanish Ministerio de Ciencia e Innovación under grant [PRE2021-101059], the Spanish Ministry, Ministerio para la Transformación Digital y de la Función Pública, the European Union – NextGenerationEU through the Càtedra Chip UPC project (grant number TSI-069100-2023-0015), and by the Ministerio de Ciencia, Innovación y Universidades y de la Agencia Estatal de Investigación (CNS2025-166442 funded by MICIU/AEI/10.13039/501100011033).

Advisor

SANTIAGO MARCO-SOLA

Advisor

JUAN CARLOS MOURE

Author

QUIM AGUADO PUIG

# *Abstract*

In the last few decades, the volume of genomic data generated by modern DNA sequencing technologies has grown at an unprecedented pace. Large-scale population studies, clinical diagnostics, personalized medicine, and epidemiological surveillance now rely on processing massive collections of DNA sequences. As sequencing throughput continues to increase, the computational cost of downstream analysis has become a primary bottleneck in genomics pipelines and clinical diagnosis.

Sequence alignment is a fundamental and computationally intensive component of current genome data analysis pipelines. It is the cornerstone of many critical methods, such as read mapping, variant detection, genome assembly, and comparative genomics. Depending on the application, alignment can be formulated in different modes, including global and semi-global, and under diverse scoring models, such as edit and affine-gap. Classical sequence alignment algorithms are based on dynamic programming (DP) and require $O(n^2)$ time and space to compute optimal alignments, where $n$ is the sequence length. Although these methods provide optimal solutions, their quadratic cost strongly limits scalability for aligning long sequences and large genomic datasets. To mitigate this cost, many practical heuristic approaches have been adopted to improve performance at the expense of optimality. However, the absence of optimality guarantees may lead to missed or suboptimal alignments, potentially affecting the accuracy of downstream genomic analyses.

In response to these limitations, hardware acceleration has been extensively explored to alleviate the computational burden of DP-based sequence alignment. Accelerated implementations based on GPUs, FPGAs, and ASICs exploit the regular structure of the DP matrix and its inherent parallelism to improve performance and energy consumption. However, these accelerators still inherit the quadratic complexity of the underlying algorithm. Consequently, despite substantial constant-factor speedups, scalability remains fundamentally constrained by the $O(n^2)$ complexity. Moreover, many of these accelerator designs are often tailored to specific sequence alignment variants, limiting their flexibility across the diverse formulations of sequence alignment.

The recent introduction of the Wavefront Alignment algorithm (WFA) marks a significant shift in the algorithmic approach to solving the sequence alignment problem. WFA is an exact, output-sensitive method whose complexity depends on the optimal alignment score $s$ rather than solely on sequence length, achieving $O(ns)$ time and $O(s^2)$ memory complexity. Moreover, WFA provides a general formulation that can be instantiated for different alignment modes, including global and semi-global alignment, and supports multiple scoring schemes such as linear and affine gap penalties. In practice, the WFA formulation enables substantial reductions in alignment computation for relatively similar sequences, which are common in many genomics applications. However, despite its favorable asymptotic behavior, practical implementations of WFA on general-purpose CPUs are constrained by irregular control flow and non-uniform memory access patterns. These characteristics limit efficient vectorization, reduce memory locality, and limit the ability to exploit the inherent parallelism exposed by the WFA

algorithm. As a result, there remains a significant gap between the algorithm's theoretical parallel potential and the practical performance of a high-performance implementation on modern computing architectures.

This thesis addresses this gap through an algorithm-architecture co-design approach based on the WFA sequence alignment framework. It presents novel adaptations of the WFA framework and high-performance GPU implementations that redesign critical aspects of the algorithm, introducing new parallelization strategies and execution schemes to align the WFA computation with the architectural constraints of highly parallel accelerators such as GPUs. Moreover, this thesis explores trade-offs among parallel scalability, memory footprint, synchronization overhead, and optimality, aiming to achieve both performance and flexibility towards different use cases in genome data analysis. The proposed techniques improve parallel scalability, reduce control-flow divergence, and enhance memory efficiency while preserving optimality under the selected scoring function and alignment mode. Overall, the contributions of this thesis demonstrate that WFA-based output-sensitive alignment algorithms can be effectively mapped to heterogeneous architectures, enabling scalable and accurate sequence alignment for emerging large-scale genomics workloads. This thesis aims to bridge the gap between the theoretical parallelism exposed by WFA and its practical realization on massively parallel accelerator architectures through its work and contributions.

First, this thesis presents the design and implementation of eWFA-GPU, an optimal WFA-based edit-distance aligner that addresses the memory limitations of other methods through novel algorithmic and architectural optimization. By introducing a succinct piggybacked backtrace strategy and a bit-parallel packed comparison kernel, it reduces memory usage by up to $4\times$ and achieves speedups of up to $265\times$ over optimized CPU baselines. This contribution demonstrates that careful redesign of memory representation and backtrace handling is paramount to enabling scalable, optimal WFA-based alignment on GPU architectures.

Second, this thesis extends previous work by supporting the biologically relevant gap-affine alignment model, presenting WFA-GPU. This contribution introduces a robust alignment scheduling strategy and a GPU-aware approximation mechanism that enable efficient processing of long and noisy reads from third-generation sequencing technologies. For full alignment-path computation on real long-read datasets, exact WFA-GPU achieves speedups of up to $4.3\times$ over multithreaded CPU WFA. The GPU-aware approximate mode reaches up to $18.2\times$ over that baseline on the Nanopore dataset, with 98.6% optimal-score agreement. This contribution demonstrates that efficient and scalable gap-affine alignment on GPUs requires heterogeneous CPU-GPU co-design and dynamic scheduling to achieve both performance scalability and robustness, even when aligning long and noisy sequences.

Finally, to address the challenge of aligning ultra-long sequences, this thesis presents BiGWAVE, a GPU-accelerated implementation of the Bidirectional WFA (biWFA) algorithm. By introducing a hyperplane tiling strategy that regularizes parallelism and improves utilization of the GPU memory hierarchy, BiGWAVE enables exact alignment of megabase-scale sequences on commodity accelerators. The implementation achieves speedups of up to $24.6\times$ over optimized multithreaded CPU implementations of biWFA. Notably, this contribution demonstrates that optimal wavefront alignment can scale to megabase-length sequences on GPUs, provided that parallelism and memory usage are jointly co-designed.

Altogether, the contributions of this thesis demonstrate that scalable and exact sequence alignment on GPUs requires more than kernel-level acceleration. Through memory-aware backtrace design, heterogeneous CPU–GPU scheduling, and algorithmic restructuring of wavefront computation, this work shows that output-sensitive alignment can be effectively mapped to modern accelerator architectures. The resulting systems enable exact edit-distance, gap-affine, and megabase-scale alignments to run efficiently on commodity GPUs, providing a scalable foundation for large-scale genomic analyses in the era of long-read and population-scale sequencing.

**Keywords:** Sequence Alignment, GPU Acceleration, Wavefront Alignment algorithm (WFA), Algorithm-Architecture Co-design, Parallel Computing, Heterogeneous Computing, Bioinformatics, Genome Data Analysis

# *Resum*

En les últimes dècades, el volum de dades genòmiques generades per les tecnologies modernes de seqüenciació d'ADN ha crescut a un ritme sense precedents. Els estudis poblacionals a gran escala, el diagnòstic clínic, la medicina personalitzada i la vigilància epidemiològica depenen actualment del processament de col·leccions massives de seqüències d'ADN. A mesura que la capacitat de seqüenciació continua augmentant, el cost computacional de l'anàlisi posterior s'ha convertit en el coll d'ampolla principal en els fluxos de treball genòmics i en el diagnòstic clínic.

L'alineament de seqüències és un component fonamental i computacionalment intensiu dels fluxos de treball actuals d'anàlisi de dades genòmiques. És la pedra angular de molts mètodes crítics, com ara el *read-mapping*, la detecció de variants, l'assemblatge de genomes i la genòmica comparativa. Depenent de l'aplicació, l'alineament es pot formular en diferents modes, incloent-hi el global i el semiglobal, i sota diversos models de puntuació, com el de distància d'edició i el de *gap-affine*. Els algorismes clàssics d'alineament de seqüències es basen en la programació dinàmica (DP) i requereixen $O(n^2)$ en temps i espai per calcular alineaments òptims, on $n$ és la longitud de la seqüència. Tot i que aquests mètodes proporcionen solucions òptimes, el seu cost quadràtic limita fortament l'escalabilitat a l'hora d'alinear seqüències llargues i grans conjunts de dades genòmiques. Per mitigar aquest cost, s'han adoptat molts enfocaments heurístics per millorar el rendiment a costa de l'optimalitat. Malgrat això, l'absència de garanties d'optimalitat pot donar lloc a alineaments perduts o subòptims, la qual cosa pot afectar la precisió de les anàlisis genòmiques posteriors.

Com a resposta a aquestes limitacions, l'acceleració per *hardware* s'ha explorat àmpliament per alleujar la càrrega computacional de l'alineament de seqüències basat en DP. Les implementacions accelerades basades en GPUs, FPGAs i ASICs aprofiten l'estructura regular de la matriu de DP i el seu paral·lelisme inherent per millorar el rendiment i el consum energètic. No obstant això, aquests acceleradors continuen heretant la complexitat quadràtica de l'algorisme subjacent. En conseqüència, malgrat les acceleracions substancials d'un factor constant, l'escalabilitat continua fonamentalment limitada per la complexitat $O(n^2)$. A més, molts dissenys d'aquests acceleradors solen estar adaptats a variants específiques d'alineament de seqüències, fet que en limita la flexibilitat davant les diverses formulacions del problema.

La recent introducció de l'algorisme d'alineament de wavefront (WFA) marca un canvi significatiu en l'enfocament algorísmic per resoldre el problema de l'alineament de seqüències. El WFA és un mètode exacte i *output-sensitive*, la complexitat del qual depèn de la puntuació òptima de l'alineament ($s$) en comptes de dependre únicament de la longitud de la seqüència, assolint una complexitat temporal de $O(ns)$ i una complexitat de memòria de $O(s^2)$. A més a més, el WFA proporciona una formulació general que es pot instanciar per a diferents modes d'alineament, incloent-hi el global i el semiglobal, i suporta múltiples esquemes de puntuació com les penalitzacions *gap-lineal* o *gap-affine*. A la pràctica, la formulació del WFA permet reduccions substancials en el càlcul de l'alineament per a seqüències relativament similars, que

són comunes en moltes aplicacions genòmiques. Malgrat el seu comportament asimptòtic favorable, les implementacions pràctiques del WFA en CPUs de propòsit general estan limitades per un flux de control irregular i patrons d'accés a memòria no uniformes. Aquestes característiques limiten la vectorització eficient, redueixen la localitat de la memòria i restringeixen la capacitat d'aprofitar el paral·lelisme inherent que exposa l'algorisme WFA. Com a resultat, continua havent-hi una escletxa important entre el potencial paral·lel teòric de l'algorisme i el rendiment pràctic d'una implementació d'alt rendiment en arquitectures de computació modernes.

Aquesta tesi aborda aquesta problemàtica mitjançant un enfocament de codisseny d'algorisme i arquitectura basat en el marc d'alineament de seqüències WFA. Presenta noves adaptacions del marc WFA i implementacions d'alt rendiment per a GPU que redissenyen aspectes crítics de l'algorisme, introduint noves estratègies de paral·lelització i esquemes d'execució per alinear el càlcul del WFA amb les restriccions arquitectòniques d'acceleradors altament paral·lels com les GPUs. D'altra banda, aquesta tesi explora els equilibris entre l'escalabilitat paral·lela, el consum de memòria, la sobrecàrrega de sincronització i l'optimalitat, amb l'objectiu d'aconseguir tant rendiment com flexibilitat per a diferents casos d'ús en l'anàlisi de dades genòmiques. Les tècniques proposades milloren l'escalabilitat paral·lela, redueixen la divergència en el flux de control i incrementen l'eficiència de la memòria tot preservant l'optimalitat. En general, les contribucions d'aquesta tesi demostren que els algorismes d'alineament *output-sensitive* basats en WFA es poden portar eficaçment en arquitectures heterogènies, permetent un alineament de seqüències escalable i precís per a les noves càrregues de treball genòmiques a gran escala. Aquesta tesi pretén tancar la bretxa entre el paral·lelisme teòric exposat pel WFA i la seva realització pràctica en arquitectures d'acceleradors massivament paral·lels.

En primer lloc, aquesta tesi presenta el disseny i la implementació de eWFA-GPU, un alineador òptim de distància d'edició basat en WFA que resol les limitacions de memòria d'altres mètodes mitjançant una nova optimització algorísmica i arquitectònica. En introduir una estratègia de *piggyback* i un *kernel* de comparació *bit-parallel*, redueix l'ús de memòria fins a $4\times$ i aconsegueix increments de velocitat de fins a $265\times$ respecte a les implementacions optimitzades per a CPU. Aquesta contribució demostra que el redisseny de la representació de la memòria i la gestió del *backtrce* és primordial per permetre un alineament escalable i òptim basat en WFA en arquitectures GPU.

En segon lloc, aquesta tesi amplia el treball previ en donar suport al model d'alineament *gap-affine*, biològicament més rellevant, presentant WFA-GPU. Aquesta contribució introdueix una estratègia de planificació d'alineament i un mecanisme heurístic adaptat a GPUs que permeten el processament eficient de *reads* llargs procedents de tecnologies de seqüenciació de tercera generació. WFA-GPU aconsegueix increments de velocitat de fins a $18{,}2\times$ en comparació amb les implementacions d'última generació en alinear conjunts de dades biològiques reals, preservant al mateix temps una alta precisió d'alineament. Aquesta contribució demostra que l'alineament eficient i escalable amb *gap-affine* en GPUs requereix un codisseny heterogeni CPU-GPU i una planificació dinàmica per assolir tant l'escalabilitat del rendiment com la robustesa, fins i tot quan s'alineen seqüències llargues i sorolloses.

Finalment, per abordar el repte d'alinear seqüències ultrallargues, aquesta tesi presenta BiGWAVE, una implementació accelerada per GPU de l'algorisme WFA bidireccional (biWFA). Mitjançant la introducció d'una estratègia de partició del treball que regularitza el

paral·lelisme i millora la utilització de la jerarquia de memòria de la GPU, BiGWAVE permet l'alineament exacte de seqüències a escala de megabases en acceleradors comercials. La implementació aconsegueix increments de velocitat de fins a 24.6× respecte a implementacions multithread optimitzades per a CPU de biWFA. Destacablement, aquesta contribució demostra que l'alineament òptim utilitzant el WFA pot escalar fins a seqüències amb longituds de megabases en GPUs, sempre que el paral·lelisme i l'ús de la memòria es codissenyin de manera conjunta.

En conjunt, les contribucions d'aquesta tesi demostren que un alineament de seqüències escalable i exacte en GPUs requereix més que una acceleració a nivell de *kernel*. Mitjançant un disseny de *backtrace* conscient de la memòria, la planificació heterogènia CPU-GPU i la reestructuració algorísmica del càlcul del wavefront, aquest treball mostra que l'alineament sensible a la sortida es pot implementar de manera eficaç en les arquitectures d'acceleradors modernes. Els sistemes resultants permeten que els alineaments exactes de distància d'edició, de *gap-affine* i a escala de megabases s'executin de manera eficient en GPUs comercials, proporcionant una base escalable per a anàlisis genòmiques a gran escala en l'era de les seqüències genòmiques llargues i la seqüenciació a escala poblacional.

**Paraules Clau:** Sequence Alignment, GPU Acceleration, Wavefront Alignment algorithm (WFA), Algorithm-Architecture Co-design, Parallel Computing, Heterogeneous Computing, Bioinformatics, Genome Data Analysis

# *Resumen*

En las últimas décadas, el volumen de datos genómicos generados por las tecnologías modernas de secuenciación de ADN ha crecido a un ritmo sin precedentes. Los estudios poblacionales a gran escala, el diagnóstico clínico, la medicina personalizada y la vigilancia epidemiológica dependen ahora del procesamiento de colecciones masivas de secuencias de ADN. A medida que el rendimiento de la secuenciación continúa aumentando, el coste computacional del análisis posterior se ha convertido en un cuello de botella principal en los flujos de trabajo genómicos y el diagnóstico clínico.

El alineamiento de secuencias es un componente fundamental y computacionalmente intensivo de los flujos de trabajo actuales de análisis de datos genómicos. Es la piedra angular de muchos métodos críticos, como el *read-mapping*, la detección de variantes, el ensamblaje de genomas y la metagenómica. Dependiendo de la aplicación, el alineamiento se puede formular en diferentes modos, incluyendo el global y el semiglobal, y bajo diversos modelos de puntuación, como el de distancia de edición y el de *gap-affine*. Los algoritmos clásicos de alineamiento de secuencias se basan en la programación dinámica (DP) y requieren $O(n^2)$ en tiempo y espacio para calcular alineamientos óptimos, donde $n$ es la longitud de la secuencia. Aunque estos métodos proporcionan soluciones óptimas, su coste cuadrático limita fuertemente la escalabilidad para alinear secuencias largas y grandes conjuntos de datos genómicos. Para mitigar este coste, se han adoptado muchos enfoques heurísticos prácticos para mejorar el rendimiento a expensas de la optimalidad. Sin embargo, la ausencia de garantías de optimalidad puede dar lugar a alineamientos perdidos o subóptimos, lo que podría afectar a la precisión de los análisis genómicos posteriores.

En respuesta a estas limitaciones, la aceleración por hardware se ha explorado ampliamente para aliviar la carga computacional del alineamiento de secuencias basado en DP. Las implementaciones aceleradas basadas en GPUs, FPGAs y ASICs explotan la estructura regular de la matriz de DP y su paralelismo inherente para mejorar el rendimiento y el consumo de energía. Sin embargo, estos aceleradores todavía heredan la complejidad cuadrática del algoritmo subyacente. En consecuencia, a pesar de las aceleraciones sustanciales de un factor constante, la escalabilidad sigue estando fundamentalmente limitada por la complejidad $O(n^2)$. Además, muchos de estos diseños de aceleradores suelen estar adaptados a variantes específicas de alineamiento de secuencias, lo que limita su flexibilidad frente a las diversas formulaciones del problema.

La reciente introducción del algoritmo de alineamiento de wavefront (WFA) marca un cambio significativo en el enfoque algorítmico para resolver el problema del alineamiento de secuencias. El WFA es un método exacto y *output-sensitive* cuya complejidad depende del coste óptimo del alineamiento ($s$) en lugar de depender únicamente de la longitud de la secuencia, logrando una complejidad temporal de $O(ns)$ y una complejidad de memoria de $O(s^2)$. Además, el WFA proporciona una formulación general que puede instanciarse para diferentes modos de alineamiento, incluyendo el global y el semiglobal, y soporta múltiples esquemas de puntuación como las variantes de *gap-affine*. En la práctica, la formulación del WFA permite reducciones

sustanciales en el cálculo del alineamiento para secuencias relativamente similares, que son comunes en muchas aplicaciones genómicas. Sin embargo, a pesar de su comportamiento asintótico favorable, las implementaciones prácticas del WFA en CPUs de propósito general están limitadas por un flujo de control irregular y patrones de acceso a memoria no uniformes. Estas características limitan la vectorización eficiente, reducen la localidad de la memoria y restringen la capacidad de aprovechar el paralelismo inherente expuesto por el algoritmo WFA. Como resultado, sigue habiendo una brecha importante entre el potencial paralelo teórico del algoritmo y el rendimiento práctico de una implementación de alto rendimiento en arquitecturas de computación modernas.

Esta tesis aborda esta brecha mediante un enfoque de codiseño de algoritmo y arquitectura basado en el marco de alineamiento de secuencias WFA. Presenta nuevas adaptaciones del marco WFA e implementaciones de alto rendimiento para GPU que rediseñan aspectos críticos del algoritmo, introduciendo nuevas estrategias de paralelización y esquemas de ejecución para alinear el cálculo del WFA con las restricciones arquitectónicas de aceleradores altamente paralelos como las GPUs. Por otra parte, esta tesis explora los equilibrios entre la escalabilidad paralela, la memoria, la sobrecarga por sincronización y la optimalidad, con el objetivo de lograr tanto rendimiento como flexibilidad para diferentes casos de uso en el análisis de datos genómicos. Las técnicas propuestas mejoran la escalabilidad paralela, reducen la divergencia en el flujo de control y aumentan la eficiencia de la memoria, al tiempo que preservan la optimalidad. En general, las contribuciones de esta tesis demuestran que los algoritmos de alineamiento basados en WFA se pueden mapear eficazmente en arquitecturas heterogéneas, permitiendo un alineamiento de secuencias escalable y preciso para las nuevas cargas de trabajo genómicas a gran escala. Esta tesis pretende cerrar la brecha entre el paralelismo teórico expuesto por el WFA y su realización práctica en arquitecturas de aceleradores masivamente paralelos a través de su trabajo y contribuciones.

En primer lugar, esta tesis presenta el diseño y la implementación de eWFA-GPU, un alineador óptimo de distancia de edición basado en WFA que resuelve las limitaciones de memoria de otros métodos a través de una novedosa optimización algorítmica y arquitectónica. Al introducir una estrategia de *piggyback* y un kernel de comparación paralelo a nivel de bits, reduce el uso de memoria hasta en $4\times$ y consigue incrementos de velocidad de hasta $265\times$ con respecto al programa optimizado para CPU. Esta contribución demuestra que el rediseño de la representación de la memoria y la gestión del *backtrace* es fundamental para permitir un alineamiento escalable y óptimo basado en el algoritmo WFA en arquitecturas GPU.

En segundo lugar, esta tesis amplía el trabajo previo al dar soporte al modelo de alineamiento *gap-affine*, biológicamente relevante, presentando WFA-GPU. Esta contribución introduce una estrategia robusta de planificación de alineamientos y un mecanismo de aproximación adaptado a GPU que permiten el procesamiento eficiente de *reads* largos y con errores, procedentes de tecnologías de secuenciación de tercera generación. WFA-GPU consigue incrementos de velocidad de hasta $18{,}2\times$ en comparación con implementaciones actuales al alinear conjuntos de datos biológicos reales, preservando al mismo tiempo una alta precisión de alineamiento. Esta contribución demuestra que un alineamiento eficiente y escalable usando *gap-affine* en GPUs requiere un codiseño heterogéneo CPU-GPU y una planificación dinámica para lograr tanto la escalabilidad del rendimiento como la robustez, incluso al alinear secuencias largas y ruidosas.

Finalmente, para abordar el reto de alinear secuencias ultralargas, esta tesis presenta BiGWAVE, una implementación acelerada por GPU del algoritmo WFA bidireccional (biWFA). Mediante la introducción de una estrategia de particionamiento del trabajo que regulariza el paralelismo y mejora la utilización de la jerarquía de memoria de la GPU, BiGWAVE permite el alineamiento exacto de secuencias a escala de megabases en aceleradores comerciales. La implementación consigue incrementos de velocidad de hasta 24.6× con respecto a implementaciones multihilo optimizadas para CPU de biWFA. En particular, esta contribución demuestra que el alineamiento óptimo puede escalar hasta secuencias con longitudes de megabases en GPUs, siempre que el paralelismo y el uso de la memoria se codiseñen de manera conjunta.

En conjunto, las contribuciones de esta tesis demuestran que el alineamiento escalable y exacto de secuencias en GPUs requiere algo más que la aceleración a nivel de *kernel*. A través de un diseño de *piggyback*, la planificación heterogénea CPU-GPU y la reestructuración algorítmica del cálculo de los *wavefronts*, este trabajo muestra que el alineamiento se puede mapear eficazmente en las arquitecturas de aceleradores modernas. Los sistemas resultantes permiten que los alineamientos exactos de *edit-distance*, de *gap-affine* y a escala de megabases se ejecuten de manera eficiente en GPUs comerciales, proporcionando una base escalable para análisis genómicos a gran escala en la era de los *long-reads* y la secuenciación a escala poblacional.

**Palabras Clave:** Sequence Alignment, GPU Acceleration, Wavefront Alignment algorithm (WFA), Algorithm-Architecture Co-design, Parallel Computing, Heterogeneous Computing, Bioinformatics, Genome Data Analysis

# *Acknowledgements*

Dono les gràcies a totes les persones que m'han ajudat durant aquests cinc anys, que m'han donat suport i m'han fet créixer com a persona i com a investigador.

En primer lloc, al Santiago per l'oportunitat que em va donar d'explorar el món de la recerca, al qual mai m'hauria imaginat que m'acabaria dedicant. Més enllà de l'ajuda tècnica, es deixa la pell perquè tots els seus estudiants tinguin bons temes de recerca i condicions laborals dignes, i es preocupa perque estiguem animats i bé emocionalment. També estic molt agraït al Juan Carlos pels seus consells, les llargues discussions tècniques, i especialment per l'oportunitat d'iniciar-me en la docència, una experiència amb la qual he gaudit i après molt gràcies a tenir referents com ell. Amb tots dos he gaudit al màxim d'aquesta etapa.

Gràcies als meus pares, Jordi i Cristina, a les meves germanes Èlia i Neus, i a tota la meva família. Si he arribat fins aquí ha estat pel seu suport, i em sento molt afortunat de tenir-los. Als meus avis, el Guillermo i la Maria, per acompanyar-me en aquest camí. També vull recordar la Pilarín, a qui estic segur que li hauria fet molta il·lusió veure com tanco aquesta etapa. Tot i que durant aquest temps hi ha hagut moments tristos i durs, gràcies a vosaltres els hem anat superant.

A l'Anna, gràcies per tota la teva energia, pel teu suport i per saber treure sempre la millor versió de mi. Els dos tanquem una etapa ara, i estic segur que les següents seran iguals o millors.

Vull agraïr al Miquel Moretó el seu suport, i a tots els companys i companyes del BSC que han estat al meu costat. En especial, al Christos, la Judith, el Max, el Cristian, l'Òscar, el Víctor, la Noelia, el Guillem, el Narcís, el Xavi, el Quim Elegant, l'Alejandro, el Lorién, el Ruben i tots aquells amb qui he compartit moments. També a tot el departament d'Arquitectura de Computadors i Sistemes Operatius (DACSO) de la UAB, especialment a la Irene, la Paula, la Cristina, el Suren, el Marc, l'Icíar, la Haoyuan, el Joan i la resta de doctorands: sense vosaltres, els dies haurien estat molt més avorrits.

I also want to thank Gagan and Kristof for the opportunity to work in a top tech company. Thank you for believing in me and for always being supportive. A big thanks as well to all my coworkers at AMD for their help during my time there.

No em puc oblidar dels meus amics: per les tardes al gimnàs o al rocòdrom, els partits de futbol, els dies a la platja, els sopars, les calçotades, les escapades al Pirineu... Gràcies també al Marc i a la Clàudia per aguantar-me tots aquests anys al pis.

I per acabar, a tota la gent amb qui he coincidit i que de ben segur em deixo de mencionar: gràcies a vosaltres també.

# Contents

**List of Figures** **xv**

**List of Tables** **xvi**

**1 Introduction** **1**
1.1 Context and Motivation . . . 1
1.2 Goal of this Thesis . . . 4
1.3 Thesis Statement . . . 4
1.4 Contributions . . . 5
1.5 List of Publications . . . 6
1.6 Thesis Organization . . . 7

**2 Background** **9**
2.1 Genome Biology and Computational Genomics . . . 9
2.2 Genome Sequencing Technologies . . . 10
2.3 Genome Data Analyses . . . 11
2.4 Pairwise Sequence Alignment . . . 13
2.4.1 Historical context . . . 13
2.4.2 Alignment Distance Functions . . . 14
2.4.3 Sequence Alignment Modalities . . . 15
2.4.4 Edit Distance . . . 16
2.4.5 Gap-affine Distance . . . 17
2.4.6 Two-piece Gap-Affine Distance . . . 17
2.5 Sequence Alignment Algorithms . . . 18
2.5.1 The Smith-Waterman-Gotoh Algorithm . . . 18
2.5.2 The Wavefront Alignment Algorithm . . . 19
2.5.3 The Bidirectional Wavefront Alignment Algorithm . . . 21
2.6 Graphical Processing Units . . . 26
2.7 HPC Tools and Libraries for Sequence Alignment . . . 27

**3 Methodology** **29**
3.1 Experimental Platform . . . 29
3.2 Datasets . . . 31
3.2.1 Synthetic Datasets . . . 31

3.2.2 Real Biological Datasets . . . 31
3.3 Evaluation Metrics . . . 32
3.4 Evaluation Methodology . . . 32

**4 GPU Accelerated Edit-Distance Sequence Alignment 34**
4.1 Introduction and Motivation . . . 34
4.2 Method . . . 35
4.2.1 Piggybacked Alignment Operations . . . 37
4.2.2 Kernel Specialization . . . 38
4.2.3 Bit-parallel Packed Sequence Comparison . . . 39
4.2.4 Overlapping kernel computation with data transfers . . . 40
4.3 Experimental Results . . . 41
4.3.1 Performance evaluation . . . 41
4.3.2 Profiling, Scalability, and Limitations . . . 42
4.3.2.1 Overall System Profiling . . . 43
4.3.2.2 Alignment Kernel Performance Profiling . . . 44
4.3.2.3 Alignment Kernel Selection . . . 45
4.3.3 Parameter Tuning . . . 45
4.3.4 Evaluation on Other Devices . . . 47
4.4 Discussion . . . 48

**5 Gap-Affine Sequence Alignment on GPU 49**
5.1 Introduction and Motivation . . . 49
5.2 Method . . . 50
5.2.1 WFA-GPU parallel algorithm . . . 50
5.2.2 Alignment Scheduling and GPU Memory Management . . . 51
5.2.3 Piggybacked Backtrace Operations . . . 52
5.2.4 Bit-Parallel Sequence Comparison using Packed DNA Sequences . . . 54
5.2.5 GPU-aware Approximate Wavefront Alignment . . . 55
5.2.6 CPU-GPU Co-design System . . . 55
5.3 Experimental Results . . . 56
5.3.1 Evaluation on Simulated Data . . . 56
5.3.2 Evaluation on Real Data . . . 59
5.3.3 Power consumption analysis . . . 61
5.3.4 Performance scalability . . . 62
5.3.5 Approximated WFA-GPU parameter exploration . . . 63
5.4 Discussion . . . 64

**6 Ultra-Long Sequence Alignment on GPUs 66**
6.1 Introduction and Motivation . . . 66
6.2 Method . . . 67
6.2.1 Hyperplane Tiling Strategy . . . 69
6.2.2 Accelerating the *Extend* Operation . . . 70
6.3 Experimental Results . . . 72

6.3.1 Evaluation on Simulated Data . . . . . 72
6.3.2 Evaluation on Real Biological Data . . . . . 74
6.3.3 Performance Characterization . . . . . 75
6.3.3.1 Overall System Profiling . . . . . 75
6.3.3.2 GPU Kernel Profiling . . . . . 76
6.3.3.3 Parameter Tuning . . . . . 77
6.4 Discussion . . . . . 83

**7 Conclusions** **84**
7.1 Summary of Contributions . . . . . 84
7.2 Concluding Discussion and Perspectives . . . . . 85
7.3 Future Work . . . . . 86
7.4 Financial Support . . . . . 87

# List of Figures


2.1 Edit and affine distance functions . . . . . 14
2.2 Global, semi-global, and local alignment modes . . . . . 15
2.3 Diagonal extension . . . . . 20
2.4 Breakpoints in biWFA recursion . . . . . 22
2.5 biWFA overlap detection . . . . . 23

4.1 Mapping of CUDA resources into WFA work. . . . . 36
4.2 Wavefront data layout aligning two sequences. . . . . 38
4.3 Computation and memory overlap of the eWFA-GPU proposal . . . . . 41
4.4 Overall profiling of eWFA-GPU . . . . . 43

5.1 Detail of the parallel computation of a given wavefront in WFA-GPU . . . . . 51
5.2 Illustration of the piggybacked backtrace strategy and data-layout organization in WFA-GPU . . . . . 53
5.3 WFA-GPU performance results . . . . . 59
5.4 WFA-GPU scalability of different GPU devices . . . . . 62
5.5 Approximate WFA-GPU parameter exploration . . . . . 63

6.1 Parallel elements of the BiGWAVE algorithm . . . . . 68
6.2 Hyperplane tiling strategy for the WFA and biWFA algorithms. . . . . 69
6.3 Implementation detail of a BiGWAVE tile . . . . . 71
6.4 Distribution of the *extend* length on different datasets. . . . . 72
6.5 Score distribution of Scrooge compared with the optimal alignment . . . . . 76
6.6 BiGWAVE performance results . . . . . 77
6.7 Overall system profiling of BiGWAVE . . . . . 78
6.8 SOL profiling of BiGWAVE . . . . . 79
6.9 Memory profiling of BiGWAVE . . . . . 79
6.10 Work imbalance analysis of BiGWAVE . . . . . 80
6.11 Optimal number of threads per block exploration . . . . . 81
6.12 Optimal number of tile steps exploration . . . . . 82

# List of Tables


3.1 Properties of hosts used for evaluation. . . . . 30
3.2 GPU specifications. The RTX 4090 column gives model specifications. Listed clocks are nominal values rather than measured sustained benchmark frequencies. 30

4.1 eWFA-GPU performance results . . . . . 41
4.2 Performance metrics of each eWFA-GPU specialised alignment kernel on datasets of 1000bp . . . . . 44
4.3 Performance metrics of each eWFA-GPU specialized alignment kernel on datasets of 150bp . . . . . 46
4.4 eWFA-GPU alignment time with different number of threads per block . . . . . 46
4.5 Alignment time (in milliseconds) of 10 million alignments using eWFA-GPU on different devices. . . . . 47

5.1 Time and optimal-score agreement of WFA-GPU on simulated datasets . . . . 58
5.2 Time and optimal-score agreement of WFA-GPU on real datasets . . . . . 60
5.3 Average power usage of WFA-GPU . . . . . 61
5.4 Energy consumption of WFA-GPU . . . . . 61

6.1 Performance results of BiGWAVE on a simulated dataset of 10K basepairs . . . 73
6.2 Performance results of BiGWAVE on a simulated dataset of 100K basepairs . . 73
6.3 Performance results of BiGWAVE on a simulated dataset of 1M basepairs . . . 74
6.4 Performance results of BiGWAVE using real datasets . . . . . 75

# Chapter 1

# Introduction

## 1.1 Context and Motivation

Genome biology studies the structure, function, and variation of an organism's complete genetic material. The genome encodes the molecular instructions that govern cellular processes, development, and adaptation. Advances in molecular biology and sequencing technologies have transformed genomics into a data-driven discipline, enabling large-scale analysis of DNA sequences to systematically study genetic variation across individuals, populations, and species. Furthermore, in healthcare, genome analysis supports disease diagnosis, variant interpretation, and precision medicine, enabling treatments tailored to individual genetic profiles. As a result, genome analysis has proven critical in neonatal intensive care, oncology, and outbreak surveillance. Beyond clinical applications, genomics plays a central role in agriculture, guiding crop improvement, livestock breeding, and resilience to environmental stress.

The rapid evolution of DNA sequencing technologies has dramatically reduced the cost and time required to generate genomic data. Over the past decade, the cost per genome has declined at a rate that outpaces Moore's Law. Modern sequencing technologies can generate terabytes of data in a single run. Large-scale initiatives, including the 1000 Genomes Project and the UK Biobank sequencing effort, routinely analyze thousands to hundreds of thousands of genomes to study population-scale genetic variation. As a result, genomic data production has shifted from a technological limitation to a data-analysis challenge. That is, while acquiring genomic data has become increasingly affordable, the computational resources required to store, process, and analyze these datasets have emerged as a dominant cost factor. In modern bioinformatics pipelines, computation is often the primary bottleneck, both in execution time and in infrastructure costs. Consequently, improving the efficiency of core computational tasks is essential to ensure that genomic analysis can scale with sequencing capacity.

Over the past two decades, sequencing technologies have evolved through multiple generations. Second-generation platforms, most notably Illumina, produce short reads of approximately 100-200 base pairs with high accuracy and massive parallel throughput. These technologies have enabled population-scale projects and large clinical studies. More recently, third-generation sequencing technologies, such as those developed by Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT), have shifted the landscape by producing long and ultra-long reads, ranging from several kilobases to megabases in length. These longer reads

simplify tasks such as genome assembly and structural variant detection, but they introduce new computational challenges due to their size and, in some cases, higher raw error rates. As a result, modern genomics pipelines must handle not only increasing data volumes but also substantially longer and noisier sequences, further increasing the computational needs of downstream analysis.

As sequencing projects expand from thousands to hundreds of thousands of genomes, the ability to analyze data rapidly becomes critical. In healthcare, reducing the turnaround time of genome analysis can directly impact diagnosis, clinical decision-making, and treatment selection for critically ill patients [28, 84]. Although sequencing throughput has increased substantially, downstream computational analysis remains a major bottleneck. For instance, whole-genome analysis can take days to weeks, largely due to the computational costs of alignment, variant detection, and large-scale data processing. Recent demonstrations of ultra-rapid genome sequencing have reduced end-to-end analysis time to under eight hours [38]. However, these results rely on large-scale cloud infrastructures and specialized high-performance systems, with reported costs exceeding 500 USD per patient. Such configurations are not representative of routine clinical practice. Consequently, accelerating and optimizing the core computational stages of genomic pipelines is essential for reducing execution time, infrastructure requirements, and overall costs, enabling scalable and economically viable genome analysis.

Notably, sequence alignment is one of the most computationally intensive steps in the genomics pipeline. Sequence alignment involves comparing two biological sequences under a given scoring model, typically accounting for substitutions, insertions, and deletions. In computational biology, it is a critical component of read-mapping [60, 57, 72], multiple sequence alignment [90], de-novo assembly [87, 53], variant detection [82], error correction [89], pangenome construction [36, 61], and others. Sequence alignment is also fundamental in other fields, such as network security [55], pattern matching, software engineering, or artificial intelligence.

Traditionally, the sequence alignment problem is solved using dynamic programming (DP) algorithms. For sequences of length $n$, classical DP formulations require $O(n^2)$ time and space. While these algorithms guarantee optimal solutions and remain the foundation of most alignment libraries and tools, their quadratic time and space complexity severely limit scalability for long sequences and large datasets. As sequencing technologies shift toward long and ultra-long reads, the cost of quadratic alignment becomes increasingly prohibitive, making classical DP approaches a fundamental computational bottleneck in modern genomics pipelines.

To reduce the computational cost of quadratic DP, many practical tools rely on aggressive heuristics to approximate sequence alignment. These approaches reduce the search space by applying techniques such as banded alignment, diagonal pruning, and early-termination criteria (a.k.a. drops). By avoiding exhaustive exploration of the full DP matrix, heuristic methods can significantly improve performance, particularly for large-scale read-mapping and database-search applications. However, this acceleration comes at the cost of losing optimality guarantees. Depending on the characteristics of the input data, heuristic strategies may miss valid alignments or produce suboptimal results, propagating errors to downstream analyses and affecting tasks such as variant calling, structural variant detection, or genome assembly. These limitations become particularly pronounced in the presence of high sequencing error rates,

structural variation, or complex genomic regions. As modern sequencing technologies increasingly produce long and noisy reads, maintaining a balance between computational efficiency and alignment accuracy remains a central challenge for scalable genomic analysis.

In parallel with these heuristic strategies, substantial research efforts have focused on accelerating exact DP algorithms through specialized hardware. Due to its high computational requirements, DP-based sequence alignment algorithms have been extensively accelerated using general-purpose SIMD instructions [67, 102, 34], custom CPU instructions [29, 42], GPUs [22, 3, 81], FPGAs [17, 13], and even Processing-In-Memory (PIM) [27, 5]. Even though some of these proposals achieve significant speedups compared to classical CPU computing, their quadratic complexity still limits scalability to sequences of moderate length. Furthermore, due to the rise of Artificial Intelligence (AI), GPUs have become widely used and are the most common accelerators in data centers and sequencing machines. GPUs are often used in parts of bioinformatics pipelines that leverage AI-based algorithms (e.g., base calling or variant calling). However, GPUs are scarcely used for other computationally intensive steps involving sequence alignment.

While hardware acceleration improves the performance of classical DP algorithms, it does not address their fundamental quadratic complexity. Recently, a new family of alignment algorithms has emerged that tackles this limitation from an algorithmic perspective: the Wavefront Alignment Algorithm (WFA) and its more recent bidirectional variant (biWFA). Unlike classical DP approaches, the computational and memory requirements of these algorithms depend on the optimal alignment score rather than directly on sequence length. These output-sensitive formulations substantially reduce both memory usage and computational cost when aligning similar sequences, which are common in many genomics applications. As a result, WFA-based methods significantly accelerate long-read alignment in practical scenarios, enabling analyses that would otherwise be computationally prohibitive.

Despite its algorithmic advantages, practical implementations of WFA on conventional general-purpose CPU architectures remain limited in exploiting available parallelism. While WFA significantly reduces the computational cost compared to classical DP formulations, its execution still involves exploring multiple wavefront diagonals and alignment states that can be processed in parallel. However, on multicore CPUs, the available degree of parallelism is relatively limited, and scaling performance typically relies on coarse-grained parallelism across independent alignments. In contrast, Graphics Processing Units (GPUs) provide a promising alternative. With thousands of lightweight threads and high memory bandwidth, GPUs offer the potential to exploit the fine-grained parallelism inherent in WFA-based alignment more effectively, opening the door to scalable, high-throughput sequence alignment.

At the same time, GPUs have become a central component of modern computing infrastructures. They are widely deployed in data centers and sequencing platforms, where they routinely accelerate computationally intensive tasks, particularly AI-based ones, such as base calling and variant detection. However, despite the substantial computational demands of sequence alignment and the parallelism exposed by the WFA formulation, no GPU implementations of WFA-based alignment algorithms were available at the time this research began. This situation revealed a clear gap between the algorithmic parallelism offered by WFA and the capabilities of modern accelerator architectures. Bridging this gap requires rethinking the design

and implementation of wavefront-based alignment algorithms to efficiently map their computation and memory behavior onto massively parallel hardware. In this context, GPUs emerge as a promising platform for exploiting the parallelism inherent in WFA and enabling scalable sequence alignment on large genomic datasets.

## 1.2 Goal of this Thesis

The main goal of this thesis is to design efficient, scalable, optimal sequence alignment algorithms based on the WFA strategy, tailored to exploit the capabilities of modern GPU architectures. This work investigates how the principles of WFA-based alignment can be reformulated to better expose parallelism, reduce synchronization overhead, and improve memory efficiency on massively parallel hardware. Ultimately, this thesis aims to deliver high-performance alignment algorithms that combine efficiency, scalability, and flexibility, enabling their practical use in modern genomic pipelines.

To that end, this thesis explores algorithmic designs that leverage the output-sensitive nature of WFA while restructuring key aspects of the computation to better match the execution model of GPUs. This includes revisiting how alignment states are generated and processed in parallel, as well as how intermediate data structures are represented and accessed. The objective is to develop alignment algorithms that preserve the optimality and flexibility of WFA-based methods while enabling efficient large-scale execution on modern GPU-based accelerator architectures. Moreover, by exploiting the massive parallelism and memory bandwidth available on GPUs, this thesis aims to enable scalable, high-throughput sequence alignment for emerging genomics workloads involving long and ultra-long reads.

## 1.3 Thesis Statement

Sequence alignment is a fundamental computational problem in genomics and bioinformatics. As sequencing technologies continue to generate ever-larger volumes of genomic data, efficient and scalable alignment algorithms have become essential to enabling cost-effective genomic analysis. While classical DP formulations guarantee optimal alignments, their quadratic computational cost limits their scalability for long sequences and large datasets.

This thesis argues that the algorithmic principles of WFA-based alignment can be reformulated to efficiently exploit GPU architectures. By redesigning key components of the WFA computation (including work computation, synchronization, and memory organization), it is possible to expose substantial fine-grained parallelism while preserving the exactness and flexibility of the original formulation. The central thesis of this work is that output-sensitive alignment algorithms derived from the WFA framework can be redesigned to map efficiently onto GPU architectures, enabling scalable and high-performance optimal sequence alignment for modern genomics workloads. Through the algorithmic designs and implementations presented in this thesis, we demonstrate that WFA-based alignment can fully exploit the computational capabilities of modern GPUs, significantly accelerating sequence alignment while maintaining the accuracy required for real-world genomic analysis pipelines.

## 1.4 Contributions

In this thesis, we propose a set of high-performance algorithms and implementations that enable the acceleration of sequence alignment on modern GPU architectures. We address the scalability limitations of traditional DP methods by adapting the WFA and biWFA algorithms for heterogeneous computing systems. The main contributions of this thesis are the following.

**Contribution 1: eWFA-GPU - A Fast, Edit-distance GPU Algorithm Based on the WFA.** The first contribution of this thesis is the design and implementation of eWFA-GPU, the first GPU-accelerated aligner based on the WFA algorithm tailored for the edit-distance metric. To overcome the quadratic memory growth associated with wavefront alignment, we introduce a novel *piggybacked backtrace strategy*. This encoding scheme enables the reconstruction of the alignment path by storing partial backtrace information within the wavefront data structures, reducing memory requirements by up to 4× and allowing the algorithm to reside entirely in the GPU's fast on-chip shared memory. Furthermore, we propose a bit-parallel packed sequence comparison kernel that performs block-wise comparisons of DNA sequences, significantly reducing thread divergence and instruction count. The resulting implementation achieves speedups of up to 265× over state-of-the-art CPU libraries and up to 56× over previous GPU tools.

**Contribution 2: WFA-GPU - Gap-affine Sequence Alignment on GPUs.** Building upon the previous contribution, we extend the wavefront acceleration to support the gap-affine distance function, which models the different costs of opening and extending a gap. This contribution addresses the challenges of aligning long and noisy reads, such as those from PacBio and Nanopore technologies, on commodity GPUs. We present a robust CPU-GPU co-design that employs an alignment scheduling system to minimize memory consumption by creating a pool of aligners that maximizes the GPU occupancy. Additionally, we introduce a GPU-aware approximated wavefront alignment heuristic, which utilizes a banded strategy to reduce the computational volume for high-error alignments with high observed optimal-score agreement. For full alignment-path computation on real long-read datasets, the exact mode delivers speedups of up to 4.3× over multithreaded CPU WFA. The GPU-aware approximate mode reaches up to 18.2× over the same baseline on the Nanopore dataset, with 98.6% optimal-score agreement.

**Contribution 3: BiGWAVE - GPU-aware BiWFA Algorithm for Ultra-long Sequence Alignment.** To tackle the memory complexity limitations of aligning ultra-long sequences (up to megabases in length), we present BiGWAVE, a GPU-accelerated implementation of the Bidirectional WFA (biWFA) algorithm. This contribution exploits three levels of parallelism: inter-sequence, intra-sequence, and recursion-level parallelism. We propose a novel *hyperplane tiling strategy* that partitions the WFA alignment space into tiles with regular parallelism, enabling the efficient use of the GPU memory hierarchy even for massive alignment problems. Furthermore, we characterize and optimize the extending phase of the algorithm using bitwise operations to minimize memory latency. BiGWAVE is the first GPU tool capable of producing exact alignments for megabase-long sequences, achieving speedups of up to 24.6× on simulated datasets compared to CPU-based biWFA implementations.

## 1.5 List of Publications

The research conducted during this thesis has resulted in several scientific publications, which are categorized according to the contribution and role of the author. The work presented in this thesis resulted in two central publications, in which the author led the research, design, implementation, and manuscript writing.

1. **Quim Aguado-Puig**, Max Doblas, Christos Matzoros, Antonio Espinosa, Juan Carlos Moure, Santiago Marco-Sola, Miquel Moreto. "WFA-GPU: Gap-affine pairwise read-alignment using GPUs". [1]

    - **Journal**: Bioinformatics (Oxford University Press)

2. **Quim Aguado-Puig**, Santiago Marco-Sola, Juan Carlos Moure, David Castells-Rufas, Lluc Alvarez, Antonio Espinosa, and Miquel Moreto. "Accelerating Edit-Distance Sequence Alignment on GPU Using the Wavefront Algorithm." [2]

    - **Journal**: IEEE ACCESS

Additionally, the following publications were produced during the development of this thesis, in which the author contributed to the research and development, and are aligned with the research topics and problems presented in this thesis.

1. Lorién López-Villellas, Cristian Iñiguez, Albert Jiménez-Blanco, **Quim Aguado-Puig**, Miquel Moretó, Jesús Alastruey-Benedé, Pablo Ibáñez, Santiago Marco-Sola "Singletrack: An Algorithm for Improving Memory Consumption and Performance of Gap-Affine Sequence Alignment". [66]

    - **Journal**: Bioinformatics (Oxford University Press) *(Under Review)*

2. Max Doblas, Oscar Lostes-Cazorla, **Quim Aguado-Puig**, Cristian Iñiguez, Miquel Moreto, Santiago Marco-Sola "QuickEd: High-performance exact sequence alignment based on bound-and-align". [30]

    - **Journal**: Bioinformatics (Oxford University Press)

3. Alejandro Alonso-Marín, Ivan Fernandez, **Quim Aguado-Puig**, Juan Gómez-Luna, Santiago Marco-Sola, Onur Mutlu, and Miquel Moreto "BIMSA: accelerating long sequence alignment using processing-in-memory". [5]

    - **Journal**: Bioinformatics (Oxford University Press)

4. Max Doblas, Oscar Lostes, **Quim Aguado-Puig**, Nicholas Cebry, Pau Fontova-Musté, Christopher Batten, Santiago Marco-Sola, Miquel Moretó. "GMX: Instruction Set Extensions for Fast, Scalable, and Efficient Genome Sequence Alignment" [29]

    - **Conference**: IEEE/ACM International Symposium on Microarchitecture (MICRO 2023)

5. Lorién López-Villellas, Rubén Langarita-Benítez, Asaf Badouh, Víctor Soria-Pardos, **Quim Aguado-Puig**, Guillem López-Paradís, Max Doblas, Javier Setoain, Chulho Kim, Makoto Ono, Adrià Armejach, Santiago Marco-Sola, Jesús Alastruey-Benedé, Pablo Ibáñez, Miquel Moretó "GenArchBench: A genomics benchmark suite for arm HPC processors". [67]

    - **Journal**: Future Generation Computer Systems

6. David Castells-Rufas, Santiago Marco-Sola, Juan Carlos Moure, **Quim Aguado-Puig**, and Antonio Espinosa. "FPGA Acceleration of Pre-Alignment Filters for Short Read Mapping With HLS." [20]

    - **Journal**: IEEE ACCESS

7. David Castells-Rufas, Santiago Marco-Sola, **Quim Aguado-Puig**, Antonio Espinosa-Morales, Juan Carlos Moure, Lluc Alvarez, and Miquel Moretó. "OpenCL-based FPGA accelerator for semi-global approximate string matching using diagonal bit-vectors." [19]

    - **Conference**: International Conference on Field-Programmable Logic and Applications (FPL)

## 1.6 Thesis Organization

The contents of this thesis are organized as follows.

- **Chapter 2** provides the necessary background to contextualize this work's contributions. It begins with an overview of genome sequencing technologies and the standard genomics analysis pipeline. Subsequently, it details the theoretical foundations of pairwise sequence alignment, focusing on DP approaches, the Wavefront Alignment algorithm (WFA), and its bidirectional variant (biWFA). The chapter concludes with a review of the GPU architecture and programming model, as well as an analysis of existing high-performance tools for sequence alignment.

- **Chapter 3** outlines the methodology employed throughout this research. It describes the hardware and software environments used for the experimental evaluation, including the specifications of the GPU architectures and the datasets utilized to benchmark the proposed solutions.

- **Chapter 4** presents **eWFA-GPU**, a GPU-accelerated algorithm for computing the exact edit-distance alignment. This chapter details the adaptation of the WFA algorithm to the GPU execution model, introducing the piggybacked backtrace strategy to minimize memory consumption and a bit-parallel kernel for efficient sequence comparison. The performance of the proposed implementation is evaluated against state-of-the-art CPU and GPU aligners.

- **Chapter 5** extends the previous work to the gap-affine distance metric, introducing **WFA-GPU**. This chapter addresses the challenges of aligning long, noisy reads by implementing an efficient memory management system and a GPU-aware approximate alignment heuristic. It also describes a CPU-GPU co-design strategy to maximize resource utilization.

- **Chapter 6** focuses on the alignment of ultra-long sequences, presenting **BiGWAVE**. This chapter describes the parallelization of the biWFA algorithm on GPUs, introducing a hyperplane tiling strategy to use the memory hierarchy effectively. The implementation is characterized and evaluated using both simulated and real biological datasets containing megabase-scale sequences.

- **Chapter 7** concludes the thesis by summarizing the main contributions and findings. It revisits the central hypothesis that WFA-based alignment can be effectively mapped onto GPU architectures and discusses how the proposed techniques enable scalable, efficient sequence alignment across diverse problem settings. The chapter also highlights the key design insights derived from this work, including the importance of memory-efficient representations, hardware-aware scheduling, and algorithm-architecture co-design. Finally, it outlines potential future research directions, such as extending these approaches to more complex scoring models and further improving scalability for emerging large-scale genomics workloads.

# Chapter 2

# Background

## 2.1 Genome Biology and Computational Genomics

The genome constitutes the complete set of genetic material of an organism, encoded as long sequences of deoxyribonucleic acid (DNA). At its core, DNA can be abstracted as a string over a finite alphabet $\{A, C, G, T\}$, where each symbol represents a nucleotide base. DNA is packaged into chromosomes, which contain all or part of an organism's genetic information. For example, each human cell has 23 pairs of chromosomes. DNA sequences exhibit a highly structured composition, including coding regions, non-coding regions, and repetitive elements. While the biological interpretation of these regions is complex, from a computational perspective, the genome provides a natural representation as a discrete sequence, enabling the application of algorithmic and data-structural techniques for its analysis.

The central dogma of molecular biology describes the flow of genetic information within a biological system, from DNA to ribonucleic acid (RNA) and ultimately to proteins. In this process, segments of DNA are transcribed into RNA, which is then translated into proteins that perform most cellular functions. From a computational perspective, this establishes a functional interpretation of genomic sequences, where specific substrings (genes) encode instructions that are executed through transcription and translation. This mapping between sequence and function provides a key motivation for sequence analysis, as variations at the DNA level may propagate to changes in RNA and protein products, potentially affecting phenotype, disease, or organismal traits.

Within the genome, genes represent functional units that encode proteins or functional RNA molecules, interleaved with regulatory elements that control gene expression. In addition to coding regions, large portions of the genome consist of non-coding DNA, including introns, intergenic regions, and repetitive sequences such as transposable elements. This heterogeneous structure introduces variability in sequence composition and complexity, with direct implications for computational analysis, as different regions may exhibit distinct patterns, levels of conservation, and varying degrees of susceptibility to variation.

Genomic sequences are not static but evolve over time through mutation and recombination. These changes manifest as substitutions (single-nucleotide differences), insertions and deletions (indels), and larger structural variations such as duplications, inversions, or translocations. From a computational standpoint, these variations define the types of differences that

must be modeled when comparing sequences. In particular, substitutions and indels correspond directly to the basic edit operations used in sequence alignment formulations, establishing a clear connection between biological processes and algorithmic models.

The abstraction of biological sequences as strings enables the formulation of genomic problems within the framework of string processing and combinatorial algorithms. A sequence can be characterized by its length, alphabet, and error model, while comparisons between sequences can be framed in terms of similarity or distance measures. This abstraction allows the use of well-established algorithmic techniques to identify correspondences, detect variations, and quantify similarity, forming the basis of many core problems in computational genomics.

Comparing genomic sequences is fundamental to understanding biological function and variation. Sequence comparison enables the identification of conserved regions, the detection of mutations, and the inference of functional or evolutionary relationships. These tasks require determining how two sequences relate to each other in the presence of differences, which naturally leads to the problem of establishing correspondences between their elements.

The analysis of genomic sequences poses several computational challenges. Genomes can span millions to billions of nucleotides, and modern studies often involve comparing large collections of sequences. Furthermore, biological data is inherently noisy, particularly in the presence of sequencing errors or complex structural variation. These factors require algorithms that are both efficient and robust, capable of handling large-scale datasets while accurately modeling biological variability. As a result, the need for scalable and effective computational methods is central to modern genomics.

## 2.2 Genome Sequencing Technologies

Genome sequencing technologies comprise the experimental methods used to determine the nucleotide sequence of DNA molecules, that is, the exact order of bases in a genome. These technologies operate by reading fragments of DNA and reconstructing the original sequence through computational analysis. Each DNA chunk is called a *read*. Depending on the underlying methodology, sequencing platforms differ in read length, throughput, accuracy, and cost, producing data with distinct characteristics that directly influence downstream computational processing. As a result, genome sequencing is not only a data generation process but also a key driver of the algorithmic and computational challenges in modern genomics.

Genome sequencing technologies have undergone a rapid evolution over the past decades, dramatically increasing throughput while reducing cost. Early sequencing methods, such as Sanger sequencing, provided highly accurate reads but were limited in scalability and throughput. The advent of second-generation sequencing (also known as next-generation sequencing, NGS) introduced massively parallel sequencing platforms capable of generating millions to billions of short reads in a single run. This technological shift enabled large-scale genomic studies and significantly accelerated the adoption of sequencing in both research and clinical settings.

Second-generation sequencing technologies, such as those developed by Illumina, produce short reads, typically ranging from tens to a few hundred base pairs. These platforms achieve high throughput and low cost per base by sequencing many fragments in parallel. However, the short read length introduces challenges for downstream analysis, particularly in repetitive

or complex genomic regions, where reconstructing the original sequence requires resolving ambiguities across many overlapping fragments.

More recent third-generation sequencing technologies, including Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT), have shifted the paradigm by producing much longer reads, often spanning thousands to millions of base pairs. These long reads enable more contiguous genome assemblies, improved detection of structural variants, and better resolution of complex genomic regions. As a result, they open new opportunities for applications such as de novo assembly, pangenome construction, and the analysis of highly repetitive sequences.

Despite their advantages, long-read sequencing technologies typically exhibit higher error rates than second-generation platforms, particularly with Oxford Nanopore sequencing. This introduces additional challenges for computational analysis, as algorithms must account for both biological variation and sequencing noise. Consequently, there is a fundamental trade-off among read length, accuracy, and throughput across sequencing technologies, which directly affects the design and performance requirements of downstream computational methods.

Notwithstanding the myriad sequencing technologies available today, no single sequencing technology can produce a complete genome from end to end. Current technological approaches involve slicing the DNA molecule into smaller fragments that can be read by modern sequencing machines.

In summary, the most widely spread sequencing technologies are the following.

- **Short reads**: Illumina is the main company producing short-read sequencing machines. Illumina machines deliver high-throughput and high-quality short reads (i.e., circa 100-200bp long). This technology is also called *second-generation sequencing*.
- **Long reads**: Also called *third-generation technologies*, the main ones are from ONT and PacBio.
  - **Ultra-long reads**: Oxford Nanopore Technologies (ONT) produces very long reads by passing the DNA molecule through a nano-scale pore and measuring the changes in the electrical field around the pore. This technology produces very long reads (up to millions of nucleotides per read) and has very high throughput, but the generated data is noisy (with 2%-10% error introduced by the sequencing machine).
  - **Accurate long reads**: Sequencing machines from the company Pacific Biosciences (PacBio) are able to produce HiFi reads (i.e., High Fidelity). HiFi sequences are usually relatively long (of the order of tens of thousands of nucleotides) and depict high accuracy. Nevertheless, this technology has lower throughput than other technologies, as its high accuracy is achieved by sequencing DNA fragments multiple times and computing a consensus across all iterations.

## 2.3 Genome Data Analyses

Modern genomics pipelines consist of a sequence of computational stages that transform raw sequencing output into biologically meaningful insights. Although specific implementations

vary depending on the application and sequencing technology, most pipelines follow a common structure that includes data acquisition, signal processing, read preprocessing, sequence comparison, and variant interpretation.

The sequencing process begins with the acquisition of genomic data, which may originate from biological samples, publicly available datasets, or simulated data used for benchmarking and method development.

Sequencing machines produce raw signals that depend on the underlying technology. These signals must be converted into sequences over the DNA alphabet $\{A, C, G, T\}$ to enable downstream analysis. This process, known as basecalling, constitutes the first major computational step in the pipeline. The nature of the raw data varies across platforms, for example, electrical current signals in Oxford Nanopore sequencing or imaging data in Illumina and PacBio systems. In recent years, there has been increasing interest in bypassing basecalling by performing downstream analysis directly on raw signals, potentially reducing computational overhead and preserving signal-level information [54, 64, 35].

Once reads are represented in nucleotide form, a quality control stage is applied to assess and improve data reliability. This includes trimming low-quality regions, filtering unreliable reads, and correcting sequencing errors introduced during library preparation or sequencing. The specific procedures depend on the characteristics of each technology, as different platforms exhibit distinct error profiles (e.g., systematic errors or read-end quality degradation). The outcome of this stage is a set of high-confidence reads suitable for downstream analysis.

The next step is read mapping, which involves identifying regions of a reference genome that match each sequenced read. Since individual genomes differ from the reference and sequencing errors are present, exact matching is not sufficient. Instead, approximate matching techniques are employed, typically based on edit distance or gap-affine scoring models, which allow substitutions, insertions, and deletions. While more general scoring schemes, such as concave gap penalties, have been proposed [41], they are rarely used in practice due to their higher computational complexity and limited adoption [18]. Read mapping is typically decomposed into four stages: indexing, seeding, chaining, and sequence alignment. The first three stages act as filtering mechanisms that reduce the search space by discarding unlikely candidate regions. As a result, the final sequence alignment stage, which is computationally the most expensive, is applied only to a reduced set of promising candidates.

The final stage of the pipeline is variant calling, which aims to identify genetic differences between the sequenced sample and the reference genome. This includes single-nucleotide variants, insertions and deletions, and larger structural variations. A key challenge in this stage is distinguishing true biological variation from sequencing errors. Classical approaches, such as the GATK HaplotypeCaller [83], rely on probabilistic models such as Pair Hidden Markov Models, whereas more recent methods, such as DeepVariant [82], leverage deep learning techniques to improve accuracy.

Although computational bottlenecks vary across sequencing technologies and applications, read mapping consistently ranks among the most demanding stages in the pipeline. In practice, its throughput can be several orders of magnitude lower than that of the sequencing process itself, with reported slowdowns ranging from $2.4\times$ to $341\times$ [8]. This imbalance highlights the need for more efficient algorithms and implementations, particularly for the sequence alignment component that dominates the overall computational cost.

## 2.4 Pairwise Sequence Alignment

Among the different stages and algorithms of the genomics pipeline, sequence alignment plays a central role, as it provides the fundamental mechanism for comparing biological sequences. At a high level, sequence alignment consists of identifying correspondences between two sequences to reveal regions of similarity and difference, allowing for mismatches and gaps. This operation underlies key tasks such as read mapping, variant detection, and genome assembly, and directly impacts the accuracy of downstream analyses. As a result, the design of efficient and accurate sequence alignment algorithms is a core problem in computational genomics.

### 2.4.1 Historical context

Saul Needleman and Christian Wunsch initially proposed the DP algorithm to find similarities between protein amino acid sequences [79]. They use a similarity matrix that determines the matching score and the mismatch penalty for each pair of elements in the sequence alphabet. They also use a gap penalty, which is added to insertions and deletions. The algorithm proposed has $O(n^2m)$ time complexity, where $n$ and $m$ are the lengths of the sequences being aligned.

Several algorithms based on the Needleman-Wunsch were proposed [85, 86, 96], reducing the time and space complexity of the algorithm to $O(nm)$, and adapting it to the edit-distance (introduced by Levenshtein in 1966 [58]). These algorithms with quadratic complexity are usually called Needleman-Wunsch (NW) as well, even though the original formulation had cubic time complexity.

In 1975, Hirschberg introduced a divide-and-conquer algorithm to compute the Longest Common Subsequence (LCS) derived from the NW algorithm [47]. It maintains the same $O(nm)$ time complexity, but shows linear space complexity (from $O(nm)$ to $O(min(n,m))$). Future contributions have been inspired by this method to achieve linear space in other distance functions or algorithm types [76, 69].

An algorithm to align two sequences using the gap-affine distance function in $O(nm)$ time and space was introduced by Gotoh [39], building on the initial proposal by Waterman, Smith, and Beyer [97]. This algorithm uses three matrices and is usually referred to as the Smith-Waterman-Gotoh (SWG) algorithm.

Another type of alignment algorithm was introduced by Ukkonen [94] and Myers [75], based on diagonal transitions. These methods exploit similarities between sequences to speed up the alignment process, obtaining an $O(ns)$ time complexity, where $s$ is the optimal alignment score. They only formulate these kinds of algorithms for the edit distance and the LCS problem.

In 1988, Myers and Miller used Hirschberg's ideas to propose an algorithm that computes the gap-affine distance with linear memory space [76] while maintaining the quadratic time complexity on the length of the sequences.

In 2021, Marco-Sola presented the Wavefront Alignment algorithm (WFA) [71]. This proposal generalizes the diagonal transition idea to the gap-affine distance function, obtaining an $O(ns)$ time algorithm in $O(s^2)$ space complexity. Additionally, its formulation makes it well-suited for exploitation of parallel or SIMD architectures. Furthermore, in 2022, the same

authors combined ideas from the WFA and Hirschberg algorithm to decrease the memory complexity to $O(s)$ while retaining the same $O(ns)$ time complexity [69]. This improved algorithm is known as the Bidirectional Wavefront Algorithm (BiWFA).

### 2.4.2 Alignment Distance Functions

A distance function is a metric used to model the differences between two sequences. They define which transformations are valid (e.g., mismatch, insertion, deletion, transposition) and which costs are associated with each. Examples of commonly used distance functions are linear and affine functions.

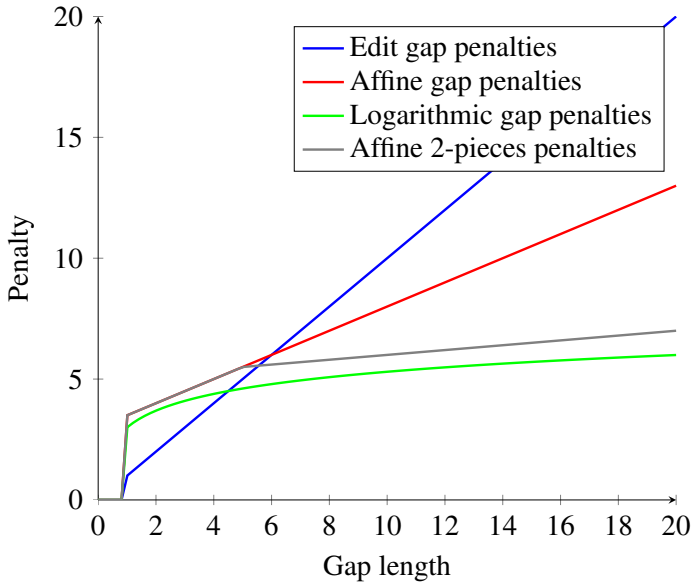



Figure 2.1: Penalties cost with gap length for different distance functions.

A widely used linear distance function is the edit-distance [58]. Described in detail in Section 2.4.4, this distance function accepts three transformations: mismatch, insertion, and deletion. Each of those has a penalty of 1, while the match has a penalty of 0. A variation of edit-distance is the Damerau-Levenshtein distance [24], which adds the transposition of two adjacent characters as a valid operation.

Another linear function is the gap-linear function, which assigns a fixed value to each operation. This is similar to edit distance, but insertions, deletions, and mismatches have independent costs that may differ from 1.

A more biologically relevant distance function is gap-affine (explained in detail in Section 2.4.5). In this case, starting an insertion or deletion incurs an initial penalty (opening a gap), and extending an open gap incurs a different penalty (extending a gap). This way, longer contiguous gaps are preferred over smaller scattered gaps.

Another proposal for biological applications is the gap-log (convex) distance. In this case, as in gap-affine, there is a gap-open cost when opening an insertion or deletion, but the extension of an opened gap is now the logarithm of a fixed value. This way, the penalty of a gap is

$O + \ell \times log(E)$, where $O$ is the gap-open penalty, $E$ is the gap-extend penalty, and $\ell$ is the gap length.

General convex distance functions are computationally expensive. For that, a 2-piece gap-affine function was proposed [40]. It is presented as a lightweight compromise between gap-affine and gap-log distances. It proposes having two different opening and extension penalties. Using the right combination (i.e., one piece with a smaller opening and bigger extension, and the other piece with a bigger opening and smaller extension), a 2-piece gap-affine function can produce a good approximation to convex distance functions. In Figure 2.1, the penalty increase produced by a gap depending on its length is graphically shown. As explained before, the 2-piece gap-affine and convex distance functions have similar behaviors.

### 2.4.3 Sequence Alignment Modalities

An alignment mode determines which parts of the sequences are taken into account when computing the score. The main ones are the following (Figure 2.2 illustrates these concepts).

- *Global alignment*: The goal is to align the sequences in their entire length; that is, end-to-end.
- *Semi-global alignment*: Also called *glocal alignment* and *ends-free alignment*. The goal is to allow gaps at the beginning and end of a sequence without incurring any penalty. It allows finding overlaps between sequences.
- *Local alignment*: The goal is to align two local sections of sequences, maximizing a similarity score that rewards matches and penalizes mismatches and gaps. The final alignment may not align the ends of both sequences, but a small inner fragment with high similarity.

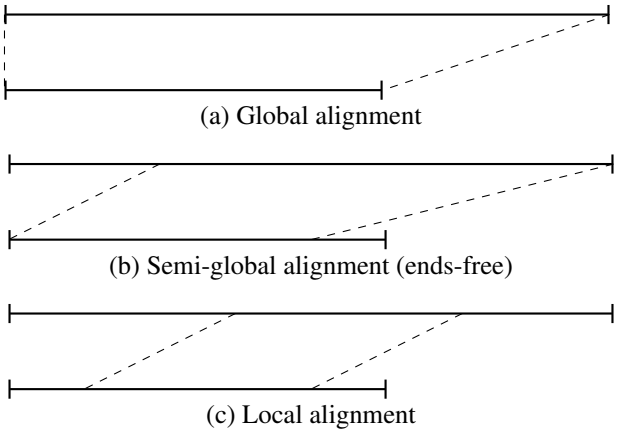



Figure 2.2: The dotted lines indicate the sections in the sequences taken into account to compute the alignment score. **(a)** Global Alignment: The full sequences are considered when aligning. **(b)** Semi-Global Alignment: The alignment may not take into account the ends of one of the sequences. **(c)** Local Alignment: Align the highest-scoring local sections within the sequences; the alignment may not take into account the end of both sequences.

### 2.4.4 Edit Distance

The edit distance (also known as the *Levenshtein* distance or unitary distance) is a metric used to quantify the dissimilarity between two sequences by counting the minimum number of edit operations required to transform one sequence into the other. The allowed operations are insertion, deletion, and substitution, each incurring a unit cost, while matching characters have zero cost. The edit distance can be efficiently computed using DP techniques [79, 95, 43].

Let $P$ and $Q$ be two sequences of length $n$ and $m$, respectively. The edit-distance $e$ between them can be computed using Eq. 2.1. In practice, the edit distance $e$ between them is obtained by filling a DP matrix $M \in \mathbb{N}^{(n+1)\times(m+1)}$, where each entry $M_{i,j}$ represents the minimum edit cost to transform the prefix $P[1..i]$ into $Q[1..j]$. The final solution is given by the value in $M_{n,m}$.

$$M_{i,j} = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} M_{i-1,j-1} + \delta(P_i, Q_j) \\ M_{i,j-1} + 1 \\ M_{i-1,j} + 1 \end{cases} & \text{Otherwise} \end{cases} \tag{2.1a}$$

$$\delta(P_i, Q_j) = \begin{cases} 0 & \text{if } P_i = Q_j \\ 1 & \text{if } P_i \neq Q_j \end{cases} \tag{2.1b}$$

In addition to computing the edit distance value, the DP matrix encodes the structure of an optimal alignment between the two sequences. This alignment can be recovered through a traceback procedure that starts at cell $M_{n,m}$ and iteratively follows the predecessor cell that realizes the minimum in the recurrence. In particular, at each position $(i, j)$, the traceback selects one of the following transitions: (i) a diagonal move to $(i-1, j-1)$ corresponding to a match or substitution, (ii) a horizontal move to $(i, j-1)$ corresponding to an insertion in $P$, or (iii) a vertical move to $(i-1, j)$ corresponding to a deletion in $P$. This process is repeated until reaching the origin $M_{0,0}$, reconstructing an optimal sequence of edit operations (i.e., the optimal alignment).

It is important to note that multiple optimal tracebacks may exist when different predecessor cells yield the same minimum cost. In such cases, any of the corresponding paths defines a valid optimal alignment. However, recovering the alignment requires either storing additional traceback information during DP computation or recomputing decisions during backtracking, which has direct implications for memory usage in practical implementations.

The DP formulation computes each entry $M_{i,j}$ in constant time by evaluating a fixed number of candidate transitions. Since the matrix contains $(n+1)(m+1)$ entries, the total time complexity is $\Theta(nm)$. Similarly, storing the full DP matrix requires $\Theta(nm)$ space, which can become prohibitive for long sequences. If only the edit distance value is required, the space complexity can be reduced to $\Theta(\min(n,m))$ by retaining only two rows (or columns) at a time. However, this optimization prevents direct traceback, as the full matrix is no longer available. Recovering the alignment in linear space requires more sophisticated strategies, such as divide-and-conquer techniques (e.g., Hirschberg's algorithm), which trade additional recomputation for reduced memory usage.

From a practical standpoint, this quadratic time and space complexity constitutes the main scalability bottleneck of classical alignment algorithms, particularly for long-read sequencing data. Moreover, the regular structure of the DP matrix exposes a high degree of parallelism, but also imposes strict data dependencies along its anti-diagonals, which constrain efficient parallel execution. These characteristics, together with the need to balance memory usage and traceback capability, are central considerations in the design of modern sequence alignment algorithms and directly motivate the development of alternative formulations.

### 2.4.5 Gap-affine Distance

The gap-affine distance is a more biologically realistic scoring model for sequence alignment, as it captures the tendency of biological sequences to exhibit contiguous insertions or deletions rather than multiple isolated gaps. Unlike the edit distance, which assigns a uniform cost to each insertion or deletion, the gap-affine model distinguishes between opening a gap and extending it, assigning different penalties to each operation.

Formally, the alignment is computed using the same basic edit operations as in edit distance: substitutions (mismatches), insertions, and deletions. However, the cost model is defined by three parameters: mismatch penalty ($x$), gap-open penalty ($o$), and gap-extend penalty ($e$). Under this model, a gap of length $\ell$ incurs a total cost of $o + e \cdot \ell$ where the initial cost $o$ penalizes the introduction of a new gap, and the extension cost $e$ penalizes each additional symbol in the gap. Typically, $o \gg e$, reflecting the biological preference for fewer, longer gaps over many short ones.

Compared to the edit distance, this formulation introduces a dependency between consecutive gap operations, since the cost of extending a gap differs from that of opening a new one. As a result, the alignment problem can no longer be modeled with a single DP matrix, since the recurrence must distinguish whether the current state corresponds to a match/mismatch or to an ongoing gap.

To account for this, classical DP formulations for gap-affine alignment introduce multiple matrices to encode different alignment states, typically separating matches/mismatches and gaps in each sequence. This leads to a more complex recurrence, but allows the model to accurately represent biologically meaningful gap structures. This behavior can be graphically illustrated in Figure 2.1.

### 2.4.6 Two-piece Gap-Affine Distance

While the gap-affine model provides a more realistic representation of biological insertions and deletions than the edit distance, it still assumes a single linear cost regime for all gap lengths. In practice, biological sequences often exhibit heterogeneous gap behavior, in which short indels and long structural variations follow different statistical patterns. As a result, a single affine function may not adequately capture the diversity of gap structures observed in real genomic data.

To address this limitation, the two-piece affine gap model (also referred to as dual-affine or piecewise-affine gap model) extends the classical affine formulation by introducing two distinct affine cost regimes. Two sets of gap penalties are defined: $(o_1, e_1)$ for short gaps and $(o_2, e_2)$

for long gaps. The cost of a gap of length $\ell$ is then given by $\min\{o_1 + e_1 \cdot \ell,\ o_2 + e_2 \cdot \ell\}$. Typically, the first regime is configured with a smaller gap-open penalty and larger extension cost, favoring short gaps, while the second regime uses a larger gap-open penalty and smaller extension cost, favoring long gaps. This piecewise definition allows the scoring model to adapt to different gap lengths.

This formulation provides greater flexibility in modeling biological variation, as it can simultaneously capture small indels and large structural variations within a unified framework. In particular, short sequencing errors and small mutations are better represented by the first regime, while longer insertions or deletions, such as those arising from recombination or structural variation, are better captured by the second regime.

From an algorithmic perspective, introducing multiple gap regimes increases the complexity of the DP formulation. In addition to distinguishing between matches/mismatches and gaps, the DP must now track which affine regime is active for each gap. This typically requires extending the standard three-matrix formulation (match, insertion, deletion) to multiple states per gap type, or equivalently, maintaining parallel gap matrices for each regime. As a consequence, both the computational cost and memory requirements increase compared to the standard gap-affine model, due to the additional states and transitions that must be evaluated at each DP cell.

## 2.5 Sequence Alignment Algorithms

Sequence alignment algorithms provide the computational foundation for comparing biological sequences under a given scoring model. Classical DP algorithms, such as Needleman-Wunsch and Smith-Waterman, establish the standard framework for optimal alignment by exhaustively exploring the solution space. Extensions such as the Gotoh algorithm incorporate affine gap penalties while preserving optimality. However, their quadratic time and space complexity limit scalability for long sequences. To address these limitations, more recent approaches have focused on reducing the search space explored or adapting computation to the structure of the alignment problem. In particular, output-sensitive algorithms, such as the Wavefront Alignment (WFA) algorithm and its bidirectional variant (biWFA), exploit the relationship between sequence similarity and computational cost to achieve improved performance in practical scenarios.

This section reviews the main algorithmic developments in sequence alignment, starting with classical DP formulations with affine gap penalties and progressing to modern output-sensitive methods. The discussion focuses on the Smith-Waterman-Gotoh algorithm as the canonical DP-based approach, followed by the modern WFA and biWFA algorithms, which form the basis for the contributions presented in this thesis.

### 2.5.1 The Smith-Waterman-Gotoh Algorithm

The Smith-Waterman-Gotoh (SWG) algorithm is the classical DP method for computing sequence alignments under a gap-affine scoring model. It extends the original Smith-Waterman formulation by incorporating distinct penalties for gap opening and gap extension, enabling a

more biologically realistic representation of insertions and deletions. To model these dependencies, the algorithm maintains three DP matrices: $M$ for matches/mismatches, $I$ for insertions (gaps in $P$), and $D$ for deletions (gaps in $Q$). These matrices are filled according to the recurrence shown in Eq. 2.2.

$$
\begin{aligned}
I_{i,j} &= \min \begin{cases} M_{i,j-1} + o + e & \text{Open insertion} \\ I_{i,j-1} + e & \text{Extend insertion} \end{cases} \\
D_{i,j} &= \min \begin{cases} M_{i-1,j} + o + e & \text{Open deletion} \\ D_{i-1,j} + e & \text{Extend deletion} \end{cases} \\
M_{i,j} &= \min \begin{cases} I_{i,j} & \text{Insertion} \\ D_{i,j} & \text{Deletion} \\ M_{i-1,j-1} + \delta(P_i, Q_j) & \text{Match/Mismatch} \end{cases} \\
\delta(P_i, Q_j) &= \begin{cases} 0 & \text{if } P_i = Q_j \\ \mathrm{x} & \text{if } P_i \neq Q_j \end{cases}
\end{aligned}
\tag{2.2}
$$

From an algorithmic perspective, SWG preserves the $\Theta(nm)$ time and space complexity of classical DP formulations, while increasing the constant factors due to the additional matrices and transitions. Nevertheless, it remains the standard optimal method for gap-affine alignment and serves as the baseline for most practical implementations.

A key observation of this formulation arises when match operations have zero cost. In this setting, extending along a diagonal through matching characters does not increase the alignment score. Is easy to see that values along a diagonal evolve monotonically with respect to the edit score, a property already observed in earlier works [94, 75, 56]. In particular, if $P_i = Q_j$, then $\delta(P_i, Q_j) = 0$, and the recurrence ensures that the optimal value at $M_{i,j}$ can be obtained directly from $M_{i-1,j-1}$ without increasing the score. This observation shows the idea of extending matches along diagonals at zero cost, which is later exploited to reduce the search space explored in output-sensitive alignment algorithms.

### 2.5.2 The Wavefront Alignment Algorithm

The Wavefront Alignment algorithm (WFA) is an output-sensitive alignment algorithm (also classified as a diagonal transition algorithm) originally formulated for computing the global gap-affine distance [71]. Unlike classical DP approaches that fill the entire DP matrix, WFA exploits the structure of the alignment problem to explore only the relevant portion of the search space. WFA formulation can be adapted to many alignment modes and scoring schemes,.

The core idea behind WFA is to leverage the monotonicity of alignment scores along diagonals, as discussed in the previous section. Instead of storing the full DP matrix, the algorithm keeps track of the farthest-reaching (f.r.) point on each diagonal for a given alignment score. This compact representation allows the algorithm to operate on a sparse set of states, significantly reducing both memory usage and computational cost when sequences are similar.

Formally, the algorithm groups all diagonals that reach a given score $s$ into a structure called a *wavefront*, denoted as $\widetilde{W}_s$. Each wavefront is represented as a vector of offsets indexed by

diagonal $k$, where each entry $\widetilde{W}_{s,k}$ corresponds to the farthest-reaching point along that diagonal. Specifically, $\widetilde{W}_{s,k}$ encodes the coordinate $(h,v) = (\widetilde{W}_{s,k}, \widetilde{W}_{s,k} - k)$ in the DP matrix. The objective of the algorithm is to compute the smallest score $s$ such that the target cell $(n,m)$ is reached by some diagonal in $\widetilde{W}_s$.

To support gap-affine penalties, WFA maintains multiple wavefront components that correspond to different alignment states. Let $\widetilde{M}_s$, $\widetilde{X}_s$, $\widetilde{I}_s$, and $\widetilde{D}_s$ denote the wavefronts corresponding to match, mismatch, insertion, and deletion states, respectively. We denote $\widetilde{W} = \{\widetilde{M}, \widetilde{X}, \widetilde{I}, \widetilde{D}\}$ as the set of all wavefront components. This decomposition mirrors the structure of the SWG formulation, but replaces full matrices with sparse diagonal representations.

The algorithm iteratively constructs wavefronts for increasing scores $s = 0, 1, \ldots$ until the target cell is reached. The propagation of wavefronts is governed by the recurrences in Eq. 2.3, which define how each component is computed from previously computed wavefronts.

$$\begin{aligned}
\widetilde{I}_{s,k} &= \max\{\widetilde{M}_{s-o-e,k-1} + 1, \widetilde{I}_{s-e,k-1} + 1\} \\
\widetilde{D}_{s,k} &= \max\{\widetilde{M}_{s-o-e,k+1}, \widetilde{D}_{s-e,k+1}\} \\
\widetilde{X}_{s,k} &= \max\{\widetilde{M}_{s-x,k} + 1, \widetilde{I}_{s,k}, \widetilde{D}_{s,k}\} \\
\widetilde{M}_{s,k} &= \widetilde{X}_{s,k} + LCP(q_{\widetilde{X}_{s,k}-k \ldots n-1}, t_{\widetilde{X}_{s,k} \ldots m-1})
\end{aligned} \tag{2.3}$$

From an operational perspective, these recurrences are typically decomposed into two main operators executed at each iteration. The *next* operator computes $\widetilde{X}_s$ by combining mismatch and gap transitions, while the *extend* operator computes $\widetilde{M}_s$ by extending matches along diagonals using the longest common prefix (LCP). This separation is key to efficiency, as the extend phase can advance multiple DP cells at once.

This behavior is illustrated in Fig. 2.3, where diagonal extensions do not increase the alignment score until a mismatch is encountered. This property enables WFA to skip large regions of the DP matrix corresponding to matching substrings, which is the main source of its efficiency.

| | | G | A | A | T | A |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| G | 1 | 0 | 1 | 2 | 3 | 4 |
| A | 2 | 1 | 0 | 1 | 2 | 3 |
| T | 3 | 2 | 1 | 1 | 1 | 2 |
| T | 4 | | | | | |
| A | 5 | | | | | |

Figure 2.3: Alignment score is not increased until a mismatch is found.

The overall structure of the WFA algorithm is shown in Algorithm 1. The algorithm initializes the wavefront at score $s = 0$ by extending matches from the origin, then iteratively

increases the score and generates new wavefronts until the target diagonal reaches the end of the sequences.

**Algorithm 1:** WFA algorithm.

**Input:** $q,t$ strings, $\{x,o,e\}$ gap-affine penalties

**Function** WFA_ALIGN($q,t,\{x,o,e\}$) **begin**
  $\widetilde{M}_{0,0} \leftarrow LCP(q_{0\ldots n-1}, t_{0\ldots m-1})$
  $s \leftarrow 0$
  **while** $\widetilde{M}_{s,m-n} \neq m$ **do**
    $s \leftarrow s+1$
    **for** $k$ ***in*** *diagonals*($\widetilde{W}_s$) **do**
      $\widetilde{I}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k-1}+1, \widetilde{I}_{s-e,k-1}+1\}$
      $\widetilde{D}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k+1}, \widetilde{D}_{s-e,k+1}\}$
      $\widetilde{X}_{s,k} \leftarrow \max\{\widetilde{M}_{s-x,k}+1, \widetilde{I}_{s,k}, \widetilde{D}_{s,k}\}$
      $\widetilde{M}_{s,k} \leftarrow$ WFA_EXTEND($\widetilde{X}_{s,k}, k, q, t$)

The extend operator, shown in Algorithm 2, computes the longest common prefix starting from a given offset and diagonal, effectively collapsing consecutive matches into a single step.

**Algorithm 2:** WFA *extend* operator.

**Input:** $o$, $k$ integers, $p$, $t$ strings

**Function** EXTEND($o$, $k$, $p$, $t$) **begin**
  $n,m \leftarrow$ **length**(p), **length**(t)
  $v,h \leftarrow o, o-k$
  **while** $v < n$ ***and*** $h < m$ ***and*** $p[v] = t[h]$ **do**
    $o,v,h \leftarrow o+1, v+1, h+1$
  **return** o

The WFA algorithm achieves $O(ns)$ time complexity and $O(s^2)$ memory complexity, where $s$ is the optimal alignment score. This output-sensitive behavior makes WFA particularly efficient for aligning similar sequences, where $s \ll n$. By avoiding the full DP matrix and focusing computation along relevant diagonals, WFA significantly reduces both computational work and memory footprint compared to classical approaches.

### 2.5.3 The Bidirectional Wavefront Alignment Algorithm

Based on the WFA [71], Marco-Sola et al. introduced a memory-efficient variant called Bidirectional WFA (biWFA) [70]. While WFA achieves $O(ns)$ time and $O(s^2)$ memory complexity, biWFA reduces the memory footprint to $O(s)$ by combining forward and reverse wavefront computations with a divide-and-conquer strategy, while preserving the same $O(ns)$ time complexity.

The key idea behind biWFA is to decompose the optimal alignment into smaller sub-alignments by identifying intermediate points, called *breakpoints*, along the optimal alignment

path. Given two sequences $p$ and $t$ of lengths $n$ and $m$, and an optimal alignment score $s$, the algorithm recursively splits the alignment into subproblems of smaller score. At recursion level $r$, the alignment is divided into $2^r$ sub-alignments, each with score approximately $s/2^r$. This process continues until trivial subproblems are reached, after which the full alignment is reconstructed by concatenating the partial solutions.

This recursive decomposition is illustrated in Fig. 2.4, where the optimal alignment path is progressively divided into smaller segments. Each breakpoint (black dots) defines a split of the alignment into two sub-alignments, which are processed independently in subsequent recursion levels.

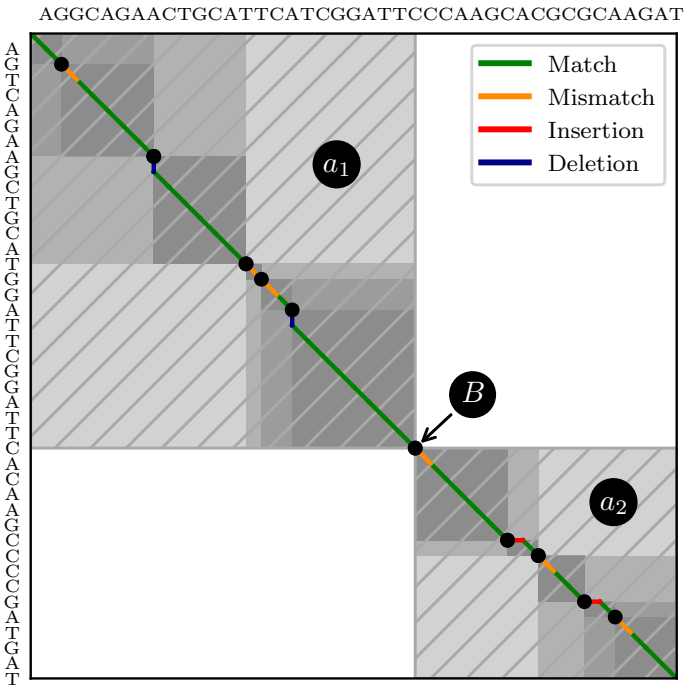



Figure 2.4: Recursive decomposition of the optimal alignment path. Each breakpoint splits the problem into two sub-alignments of a smaller score.

To identify breakpoints, biWFA performs simultaneous forward ($\overrightarrow{\mathscr{A}}$) and reverse ($\overleftarrow{\mathscr{A}}$) alignments. The forward alignment processes sequences from $(0,0)$ to $(n,m)$, while the reverse alignment processes the reversed sequences, effectively computing the DP matrix from $(n,m)$ backwards. Each direction computes wavefronts independently using the basic WFA formulation. A breakpoint is detected when a forward wavefront $\overrightarrow{W}_{e_{fw}}$ and a reverse wavefront $\overleftarrow{W}_{e_{rev}}$ overlap on a common diagonal. Intuitively, this indicates that the two partial alignments meet at a consistent point in the DP matrix. The condition for overlap is that the forward and reverse offsets on the same diagonal cover the full length of the target sequence. This mechanism is depicted in Fig. 2.5, where the intersection between forward (black) and reverse (blue) wavefronts determines the breakpoint (highlighted in red).

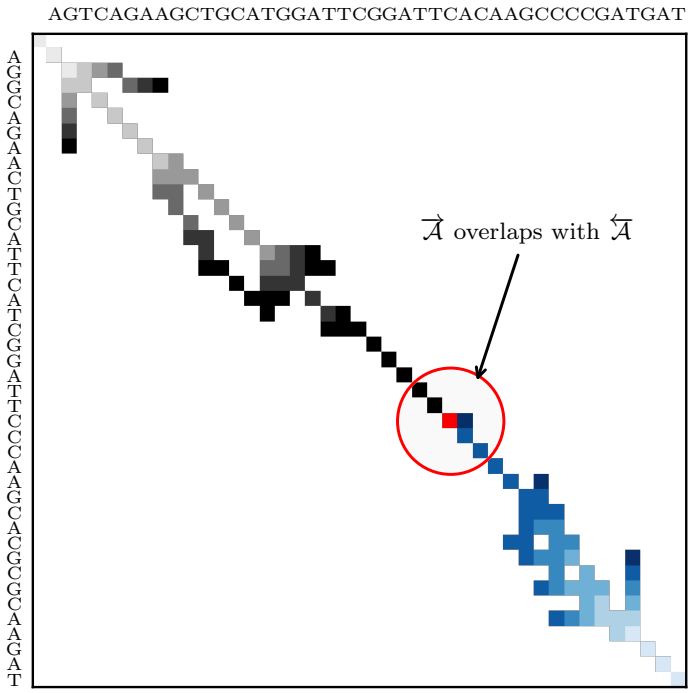



Figure 2.5: A breakpoint is detected when forward and reverse wavefronts intersect on a common diagonal.

In the edit-distance setting, biWFA uses a simplified wavefront recurrence compared to the gap-affine case. For each score $e$, a wavefront $\widetilde{W}_e$ of width $2e+1$ is computed from the previous wavefront $\widetilde{W}_{e-1}$ using Eq. 2.4, followed by diagonal extension using Eq. 2.5.

$$\widetilde{W}_{e+1,k} = \max \begin{cases} \widetilde{W}_{e,k+1} & \text{(deletion)} \\ \widetilde{W}_{e,k} + 1 & \text{(mismatch)} \\ \widetilde{W}_{e,k-1} + 1 & \text{(insertion)} \end{cases} \tag{2.4}$$

$$\widetilde{W}_{e,k} = \widetilde{W}_{e,k} + LCP(p_{\widetilde{W}_{e,k}-k \ldots n-1}, t_{\widetilde{W}_{e,k} \ldots m-1}) \tag{2.5}$$

The algorithm alternates between forward and reverse wavefront expansions, checking for overlap after each step. Importantly, only the most recent wavefronts in each direction need to be stored, which enables the reduction from $O(s^2)$ to $O(s)$ memory.

The overall structure of the biWFA algorithm for edit distance is shown in Algorithm 3. The algorithm initializes forward and reverse wavefronts, iteratively expands them, and stops when an overlap (breakpoint) is found. The total alignment score is then obtained as the sum of forward and reverse scores. The overlap condition is evaluated as shown in Algorithm 4, where common diagonals between forward and reverse wavefronts are checked. The generation of new wavefronts is shown in Algorithm 5.

**Algorithm 3:** High-level bidirectional WFA procedure for computing edit distance.

**Input:** $p,t$ strings
**Output:** Optimal edit distance between $p$ and $t$

**Function** BIWFA_ALIGN($p,t$) **begin**
  // Reverse sequences to enable backward alignment
  $\overleftarrow{p}, \overleftarrow{t} \leftarrow p_{n...0}, t_{m...0}$
  // Initialize forward and reverse wavefronts at score 0
  $\overrightarrow{W}in \leftarrow$ WFA_EXTEND$(0,0,p,t)$
  $\overleftarrow{W}in \leftarrow$ WFA_EXTEND$(0,0,\overleftarrow{p},\overleftarrow{t})$
  // Initialize edit-distance levels explored in both directions
  $e_{fw}, e_{rev} \leftarrow 0,0$
  // Expand both searches until their wavefronts overlap on a common diagonal
  **while** *not* BIWFA_OVERLAP*($\overrightarrow{W}in$, $\overleftarrow{W}in$, $t$)* **do**
    // Compute next forward wavefront
    $\overrightarrow{W}out \leftarrow$ NEXT_WF($\overrightarrow{W}in$, p, t, $e_{fw}$)
    // Increase forward score
    $e_{fw} \leftarrow e_{fw} + 1$
    // Check whether forward and reverse wavefronts meet
    **if** BIWFA_OVERLAP*($\overrightarrow{W}out$, $\overleftarrow{W}in$, $t$)* **then**
      **break**
    // Compute next reverse wavefront
    $\overleftarrow{W}out \leftarrow$ NEXT_WF($\overleftarrow{W}in$, $\overleftarrow{p}$, $\overleftarrow{t}$, $e_{rev}$)
    // Increase reverse score
    $e_{rev} \leftarrow e_{rev} + 1$
    // Keep only the most recent wavefronts in each direction
    SWAP($\overrightarrow{W}in$, $\overrightarrow{W}out$)
    SWAP($\overleftarrow{W}in$, $\overleftarrow{W}out$)
  // Return the sum of forward and reverse scores
  **return** $e_{fw} + e_{rev}$

Algorithm 3 computes the edit distance by simultaneously exploring the alignment space from both ends. Starting from the origin and the destination of the DP matrix, the algorithm alternates forward and reverse wavefront expansions until both searches overlap on a common diagonal. This overlap determines a breakpoint and implies that the optimal score is the sum of the forward and reverse scores accumulated up to that point. Operationally, the algorithm avoids storing the full set of wavefronts, retaining only the most recent forward and reverse frontiers, which is the key to achieving linear memory complexity in the alignment score.

Algorithm 4 determines whether the forward and reverse searches have met, which is the condition used by biWFA to identify a breakpoint. Operationally, the procedure examines the diagonals present in both wavefronts and checks whether the sum of their farthest offsets is sufficient to cover the full length of the target sequence. When this condition holds, the forward and reverse partial alignments intersect in the DP space, and the current recursion level can stop and split the problem into two smaller sub-alignments.

Algorithm 5 computes the next wavefront by extending the current alignment frontier from score $e$ to $e+1$. For each diagonal $k$, the algorithm first performs a transition step that considers

**Algorithm 4:** Overlap test between forward and reverse biWFA wavefronts.

**Input:** $\overrightarrow{W}_{e_{fw}}$ and $\overleftarrow{W}_{e_{rev}}$ wavefronts, $t$ string
**Output:** **True** if both wavefronts overlap, **False** otherwise

**Function** BIWFA_OVERLAP($\overrightarrow{W}_{e_{fw}}$, $\overleftarrow{W}_{e_{rev}}$, $t$) **begin**
    `// Assume no breakpoint has been found yet`
    breakpoint_found $\leftarrow$ **False**
    `// Check all diagonals shared by the forward and reverse wavefronts`
    **for** $k_{fw}, k_{rev}$ *in* COMMON_DIAGONALS($\overrightarrow{W}_{e_{fw}}$, $\overleftarrow{W}_{e_{rev}}$) **do**
        `// If both offsets cover the full length of t, the searches overlap`
        **if** $\overrightarrow{W}_{e_{fw},k_{fw}} + \overleftarrow{W}_{e_{rev},k_{rev}} \geq$ ***length***$(t)$ **then**
            breakpoint_found $\leftarrow$ **True**
            **break**
    `// Return whether an overlap was detected`
    **return** breakpoint_found

**Algorithm 5:** Computation of the next wavefront in biWFA (edit-distance case).

**Input:** $\widetilde{W}_{in}$ wavefront, $p$, $t$ strings, $e$ integer
**Output:** Next wavefront $\widetilde{W}_{out}$ at score $e+1$

**Function** NEXT_WF($\widetilde{W}_{in}$, $p$, $t$, $e$) **begin**
    $k_{lo} \leftarrow -e$
    $k_{hi} \leftarrow e$
    $\widetilde{W}_{out} \leftarrow \varnothing$
    `// Compute next offsets for all diagonals`
    **for** $k \leftarrow k_{lo}$ ***to*** $k_{hi}$ **do**
        `// Transition step: insertion, mismatch, deletion`
        $\widetilde{W}_{out,k} \leftarrow \max\{\widetilde{W}_{in,k-1}+1,\ \widetilde{W}_{in,k}+1,\ \widetilde{W}_{in,k+1}\}$
        `// Extension step: advance along matches (longest common prefix)`
        $\widetilde{W}_{out,k} \leftarrow$ WFA_EXTEND($\widetilde{W}_{out,k}, k, p, t$)
    **return** $\widetilde{W}_{out}$

the three possible edit operations (insertion, deletion, and mismatch), selecting the farthest-reaching offset. Then, it applies an extension step that advances along matching characters using the longest common prefix between the corresponding suffixes of $p$ and $t$. This separation between transition and extension is the key mechanism that allows WFA-based methods to efficiently skip contiguous matches and achieve output-sensitive performance.

The optimal alignment (CIGAR) can be reconstructed using two strategies. One approach recursively decomposes the alignment until trivial subproblems are reached, storing breakpoint operations and concatenating them. Alternatively, recursion can stop at a given threshold, and a classical algorithm can be used to complete the alignment in each subproblem. Overall, biWFA preserves the output-sensitive time complexity of WFA while drastically reducing memory usage. This makes it particularly suitable for aligning long sequences, where the quadratic memory requirements of classical methods become prohibitive.

## 2.6 Graphical Processing Units

GPUs are massively parallel, throughput-oriented devices. They have multiple processing units called stream multiprocessors (SMs) capable of executing hundreds of instructions in parallel by using deep pipelines and aggressive fine-grained multithreading.

The SM is the main processing unit of the GPU. It implements the Single Instruction Multiple Threads (SIMT) execution model, in which an instruction is simultaneously issued to groups of 32 threads, called *warps*. Even though all the elements of a *warp* execute the same instructions, each thread can maintain an independent control flow, having its own Program Counter (PC) and call stack. Due to conditional logic, all 32 threads in a *warp* may not need to execute the issued instruction. In this case, some threads are predicated and do not perform any useful work. This problem is called *thread divergence*, and has a negative performance impact on GPU programs. Each SM contains four SIMD in-order cores, called *SM subpartitions*, which manage a fixed-size pool of warps. Each warp is allocated to an SM *sub partition* and resides there from launch to completion.

Modern GPUs expose a hierarchical memory system designed to balance capacity, latency, and bandwidth. This hierarchy includes global memory, L2 and L1 caches, shared memory, and registers, each with different performance characteristics and usage models. Global memory is a 49-bit virtual address space mapped into the GPU's physical DRAM or HBM memory. It is the largest, but slowest, memory space. Global memory is visible to all threads running on the GPU. Global memory is accessible by all threads and is commonly used for bulk data storage and communication between kernels.

The GPU has a single large L2 cache shared across all the SMs. In addition to caching data, it also performs compression and global atomics. Each SM has an L1 data cache that receives the memory requests that the SM issues. In modern architectures, when an SM executes a global memory instruction for a warp, a *request* containing the information of all participant threads of the warp is sent into the L1. Depending on the access pattern, a different number of *sectors* (aligned 32-byte chunk of memory) are requested in one or multiple 128-byte cache lines. The L1 cache has multiple processing stages that operate in a pipeline manner. The maximum unit that can pass through the L1 pipeline every cycle is called a *wavefront* (not to be confused with a wavefront from the biWFA algorithm). If all cache lines or sectors can not be accessed using a single *wavefront*, multiple *wavefronts* are created, serializing the memory accesses. The number of *wavefronts* required for a memory instruction depends on the data access pattern; an instruction that requires 4 sectors may require 1, 2, or more *wavefronts* to get the data from the memory subsystem.

In addition to caches, GPUs provide a programmer-managed on-chip memory known as *shared memory*, which shares physical resources with the L1 cache. Shared memory is much faster than global memory but has a limited capacity (typically tens of kilobytes per SM). Shared memory is organized into 32 banks, such that consecutive 32-bit words map to consecutive banks. When threads in a warp access addresses that map to different banks, all accesses can proceed in parallel. However, if multiple threads access the same bank, a *bank conflict* occurs, and the accesses are serialized. Proper data layout and access patterns are therefore critical to fully exploit shared memory performance.

Registers form the fastest level of the memory hierarchy and are used to store per-thread variables. Each SM subpartition contains a large register file (e.g., 64 KiB), resulting in a total of approximately 256 KiB of registers per SM. The availability of registers directly impacts occupancy, as the number of threads that can be scheduled concurrently is limited by total register usage. It is important to emphasize that access to the memory subsystem has extremely long latencies compared to arithmetic latencies. For this reason, it is paramount to perform most computations on registers or fast on-chip memories. It is also important to maintain a high amount of parallelism, as having available vast amounts of parallelism is the main latency-hiding method of GPUs.

Since its release in 2006, CUDA has become the most popular programming model for general-purpose NVIDIA GPU computing. CUDA comes with a software environment that allows using a superset of C/C++, together with API calls, to program one or multiple GPU devices. The CUDA programming model provides a heterogeneous environment where the host code runs on the CPU, and the device code runs on a physically separate GPU. Both the host and device can maintain their own separate memory spaces; meanwhile, CUDA supports data transfer between host and device memory. The CUDA programming model defines a computation hierarchy formed by kernels, thread blocks, warps, and threads.

- **Kernel:** Minimum unit of work sent from the CPU to the GPU. In short, a kernel is a function executed in parallel on a GPU by a large number of different CUDA threads.
- **Thread block:** A group of threads that are executed by one SM and cannot migrate to other SMs (except during preemption or dynamic parallelism). Threads within a block can cooperate via synchronization primitives, using registers, or shared memory. Thread blocks are scheduled non-deterministically for independent MIMD execution into SMs.
- **Warp:** A thread block is divided into batches of 32 threads, called warps, which are the smallest scheduling unit.
- **Thread:** Minimum execution unit of programmed instructions in CUDA.

## 2.7 HPC Tools and Libraries for Sequence Alignment

Over the years, many efforts have been devoted to finding new algorithms and more efficient implementations for computing pairwise edit-distance alignments. In [78], Navarro provides a comprehensive review of the most relevant algorithms and a performance evaluation for different datasets and configurations. Most alignment algorithms can be classified into four categories: DP-based, automaton, filters, and bit-parallel algorithms. In practice, bit-parallel algorithms outperform other approaches. Most notably, these include the BPM [77], the O(ND) [75], and the Wu-Manber (WM) [98] algorithms.

Based on the most successful algorithmic approaches, many high-performance CPU libraries have been presented. Some of them have become extensively used due to their efficiency or versatility, most notably, Edlib [88], BGSA [101], and SeqAn [31] for the edit distance. Edlib is an efficient CPU implementation of the BPM algorithm used within many Bioinformatics tools. BGSA is also a highly efficient implementation of the BPM algorithm,

optimized to exploit vectorization on multi- and many-core CPUs. SeqAn is a sequence analysis library that implements a hybrid algorithm that combines the memory-efficient Hirschberg's algorithm [31] with the BPM algorithm.

Regarding CPU libraries capable of computing the gap-affine and dual gap-affine distance functions, the most relevant ones are WFA2-lib, implementing the WFA and biWFA algorithm [71, 70], and KSW2, integrated into the read-mapper minimap2 [60].

Additionally, there have been many efforts to adapt and optimize these algorithms on GPU devices. Most relevant proposals are based on DP, computing cells antidiagonal-wise in parallel [25, 92, 11, 63, 51]. Meanwhile, some research efforts have been focused on producing efficient CUDA implementations of the classical Needleman-Wunsch [59] algorithm; other proposals have focused on novel organisations of the DP-matrix to exploit the GPU resources [33] efficiently. In particular, in [80] and [48], the authors propose an algorithm to reduce memory operations when computing the DP-matrix, by using *warp-shuffle* instructions of current Nvidia GPU architectures.

Many other GPU-based methods have opted for accelerating bit-parallel algorithms. In [93], the authors propose using warp-shuffle operations to simulate a 1024-bit machine word, allowing to perform approximate string matching on long patterns. Also, in [52], the authors exploit the Crochemore algorithm based on a suffix automaton for bit-parallel alignment. As in [74], other proposals revisit the Shift-Or and Wu-Manber algorithms, implementing them as inclusive-scan operations to enable multiple parallel computations. Similarly, in [22], the authors propose a thread-cooperative version of the BPM algorithm, achieving very high performance results on an Nvidia GTX 680 GPU. Other implementations, like GASAL2 [3] and ADEPT [9], Logan [100], and Darwin [4] opted for accelerating the more traditional Smith-Waterman algorithm. There are also proposals, like Scrooge [65], based on heuristic algorithms.

Furthermore, there have been many proposals to optimize sequence alignment on field-programmable gate array (FPGA) devices [19, 17, 49, 45]. Most notable FPGA implementations exploit bit-parallel techniques and custom processing designs to accelerate the computation of multiple alignments in parallel.

# Chapter 3

# Methodology

This chapter presents the methodological principles common to the contributions in this thesis. Its purpose is to provide a unified and consistent framework for algorithm design, implementation, and evaluation, avoiding repetition across individual chapters. While each contribution introduces specific methodological details tailored to its context, all of them adhere to the common criteria and practices described here. Rather than describing individual experimental setups in isolation, this chapter abstracts the shared decisions that control the evaluation process. This includes the definition of input models, the selection of datasets, the execution environment, and the metrics used to assess performance.

The evaluation aims to make algorithmic, implementation, and hardware effects distinguishable. Experimental settings are reported per contribution: the studies were performed on different platforms, so results from different chapters should not be treated as controlled cross-platform comparisons.

The remainder of this chapter describes the common methodology adopted throughout the thesis, the hardware and software environments used for benchmarking, the datasets selected to represent various genomic analysis use-cases, and the metrics used to assess the performance and accuracy of the proposed algorithms.

## 3.1 Experimental Platform

We conducted our experiments on a set of high-performance computing nodes equipped with different CPU and GPU architectures. The selected platforms span from consumer-grade GPUs to data center accelerators, enabling evaluation of the proposed methods across a wide range of computational and memory configurations. This diversity is essential to assess the scalability and robustness of the algorithms across different hardware characteristics, including memory capacity, compute throughput, and memory bandwidth. The detailed specifications of the host systems are reported in Table 3.1, while the properties of the GPU accelerators are summarized in Table 3.2.

The GPU devices considered in this study differ significantly in terms of compute capability, number of streaming multiprocessors (SMs), available parallelism, and memory hierarchy. These architectural differences directly affect the behavior of alignment algorithms, particularly with respect to occupancy, memory pressure, and latency hiding. The platforms allow

Table 3.1: Properties of hosts used for evaluation.

| | Node 1 | Node 2 | Node 3 | Node 4 |
|---|---|---|---|---|
| CPU | IBM Power9 | Intel i9-9900K | Intel Xeon-W2155 | i9-13900KF |
| Number of Cores | 20 | 8 | 10 | 24 |
| Threads per Core | 4 | 2 | 2 | 2 P / 1 E |
| Main Memory (GB) | — | 32 | 128 | 64 |
| GPU | V100 | RTX 2080 Ti | RTX 3080 | RTX 4090 |

us to examine architecture-dependent behavior; differences in host systems and experimental settings limit the extent to which trends can be attributed to the GPU alone.

Table 3.2: GPU specifications. The RTX 4090 column gives model specifications. Listed clocks are nominal values rather than measured sustained benchmark frequencies.

| | V100 | RTX 2080 Ti | RTX 3080 | RTX 4090 |
|---|---|---|---|---|
| Compute capability | 7.0 | 7.5 | 8.6 | 8.9 |
| Listed clock frequency (MHz) | 877 | 1605 | 1710 | 2235 |
| SMs | 80 | 68 | 68 | 128 |
| FP32 CUDA cores | 5120 | 4352 | 8704 | 16384 |
| Maximum warps per SM | 64 | 32 | 48 | 48 |
| Register space per SM (KiB) | 256 | 256 | 256 | 256 |
| Shared memory per SM (KiB) | 96 | 64 | 100 | 100 |
| Global memory size (GiB) | 16 | 11 | 10 | 24 |
| L2 cache size (KiB) | 6144 | 5632 | 5120 | 73728 |
| Host to Device bandwidth (GB/s) | 67.1 | 13.2 | 12.3 | — |
| Device to Host bandwidth (GB/s) | 65.8 | 13.2 | 13.1 | — |

In addition to GPU characteristics, host systems also play a significant role in overall execution. Differences in CPU architecture, core count, and main memory capacity may affect data preparation, scheduling, and host-device communication. For this reason, both host and device specifications are explicitly reported and taken into account when interpreting the experimental results. While the tables report the main architectural characteristics relevant to this study, some parameters, such as effective memory bandwidth and interconnect performance, may vary depending on system-level factors. When necessary, these aspects are discussed in the corresponding experimental sections to provide additional context for the observed results.

## 3.2 Datasets

The performance of sequence alignment algorithms is strongly influenced by sequence length and similarity, typically measured by the error rate. These dimensions determine both the computational complexity and the memory requirements of DP and WFA-based methods. To capture this variability in a controlled and representative manner, we consider a combination of synthetic and real biological datasets.

### 3.2.1 Synthetic Datasets

Synthetic datasets are used to systematically explore the behavior of the algorithms under controlled conditions. This allows the isolation of the effect of individual parameters, such as sequence length and error rate, which is not possible with real data alone. For that, we generate pairs of sequences with lengths ranging from short reads (150 bp) to ultra-long reads (1 Mbp). This range covers typical genomics workloads, from Illumina sequencing to long-read technologies, and stresses both compute and memory scalability.

Synthetic pairs contain substitutions, insertions, and deletions at nominal target error rates between 1% and 20%, with the actual length/error combinations specified in each results table. In Chapter 5, the published setup uses nominal lengths of 150, 1000, and 10 000 bases and average edit-error rates of 2%, 5%, and 10% [1]. A nominal mutation rate is not necessarily the optimal edit distance divided by sequence length: repeated edits, indels, and alternative alignment paths can change that relationship. All the synthetic datasets are generated using the tool available in WFA2-lib[1].

### 3.2.2 Real Biological Datasets

To complement the controlled evaluation, we include real biological datasets that reflect standard genomics workloads. These datasets capture sequencing-specific error profiles, length distributions, and structural variations that are not fully represented by synthetic data.

- **Illumina (Short Reads):** Real human reads ($\sim$ 150-250 bp) with low error rates ($< 1\%$), obtained from the Genome in a Bottle (GIAB) project. The dataset is available at GIAB data repository.
- **PacBio CCS (circular consensus sequencing):** Long reads of up to 18 Kbp from the GIAB project, representing moderate-length sequences with intermediate error rates.
- **PacBio HiFi:** High-fidelity long reads (up to approximately 25 Kbp in the Chapter 5 dataset), obtained from the PrecisionFDA Truth Challenge V2. The dataset is available at PrecisionFDA challenge portal.
- **ONT UL:** Ultra-long reads generated using the Oxford Nanopore Technologies (ONT) PromethION platform, provided by the Human Pangenome Reference Consortium [73]. These reads stress extreme-length alignment scenarios.

[1] `https://github.com/smarco/WFA2-lib`

- **ONT MinION:** Long ONT reads from [15], restricted to sequences of at least 500 Kbp. As the original dataset contains a limited number of sequences, it is replicated multiple times to construct sufficiently large batches for GPU execution. Replication preserves the individual alignment problems but does not increase the number of independent biological observations. Repeated pairs may affect cache reuse and batch behavior.

The Chapter 5 study additionally uses the Nanopore accessions ERR3279200 and ERR3279201. Its supplementary dataset table reports 100 million pairs for each Illumina dataset and 10 million pairs for each PacBio and Nanopore dataset [1]. These batch counts describe the benchmark workload rather than the number of independent biological observations.

## 3.3 Evaluation Metrics

Exactness means optimality under a specified scoring function and alignment mode. A score-optimal alignment need not be unique and does not establish the true biological correspondence or improved mapping or variant-calling accuracy.

1. **Alignments per second:** The number of input pairs divided by elapsed benchmark time. The timing boundary must distinguish kernel-only timing from full pairwise alignment, including transfers, CPU rescue, traceback, and output generation where applicable. Neither includes a complete read mapper unless explicitly stated.

2. **Equivalent GCUPS:** For a pair of lengths $n$ and $m$ taking time $T$, this is $nm/(10^9 T)$. For a batch, the numerator is $\sum_i n_i m_i$ and $T$ is the batch time. WFA does not update every DP cell, so this is an equivalent DP-work normalization, not a measured rate of actual cell updates or a hardware-independent efficiency measure.

3. **Optimal-score agreement (exact-score rate):** For $N$ evaluated pairs, $R = 100N^{-1}\sum_{i=1}^{N} \mathbf{1}[\hat{s}_i = s_i^\star]$, where $s_i^\star$ is the exact optimum and $\hat{s}_i$ is the reported score under the same objective. This metric is called *recall* in the original Chapter 5 publication and some embedded figures. Matching the score does not require an identical CIGAR when multiple optimal paths exist; path validity is a separate correctness condition. Incompatible alignment modes are marked not comparable, and execution failures are reported separately. For an approximate method, 100% observed agreement is not a general exactness guarantee.

## 3.4 Evaluation Methodology

The evaluation follows a common workflow comprising dataset preparation, execution on the target platform, measurement of performance and optimal-score agreement, and interpretation of the results. Synthetic datasets allow sequence length and nominal error rate to be varied systematically, while biological datasets expose the implementations to less uniform alignment workloads. Together, these experiments examine both the throughput achievable for a given workload and the resource limitations that arise as alignment difficulty increases.

To support fair comparisons, implementations are evaluated on the same input pairs and, where supported, the same scoring function and alignment mode. Implementation-specific optimizations are compatible with this comparison provided that they preserve the problem being solved. Exact and approximate configurations are distinguished explicitly, and performance gains from approximation are interpreted alongside optimal-score agreement. Likewise, a distance-only execution is compared separately from an execution that reconstructs the alignment path, because the latter includes additional computation and memory requirements. These distinctions prevent differences in the requested output or objective from being interpreted as implementation speedups.

The published Chapter 5 comparison uses WFA2-lib commit `931181d`, ten physical CPU cores, and an RTX 3080 on the Xeon-W2155 host. Its GPU timings include host–device transfers, alignment, traceback, and CIGAR generation. Distance-only executions omit path reconstruction and must be distinguished from full-path executions, as must exact and approximate modes. CPU and GPU results are compared within each experiment using the same input pairs and stated scoring objective; edit-distance tools re-scored under gap-affine penalties and local-only tools are explicitly qualified in Chapter 5.

For a fixed workload, speedup is the ratio of the baseline execution time to the evaluated implementation's execution time, using matching timing boundaries. Full pairwise-alignment throughput captures the combined effect of computation, data movement, and host-side work. Kernel-level measurements and profiling, where presented, help explain these results by identifying memory, synchronization, or utilization bottlenecks; they do not replace the full timing measurement. Because transfers and computation can overlap, the sum of individual component times need not equal elapsed execution time. Results are interpreted within their experimental platform, and comparisons across chapters account for differences in hardware and configuration. Timeouts, allocation failures, and unsupported configurations are reported separately rather than treated as successful fast executions.

The combined analysis of execution time, throughput, and optimal-score agreement distinguishes improvements in computational efficiency from changes in the alignment objective. In particular, performance on short, low-error inputs is interpreted separately from performance on long or noisy inputs, since the latter expose more wavefront parallelism while increasing memory requirements. This workload-dependent analysis provides the basis for assessing the benefits and limitations of each proposed implementation.

# Chapter 4

# GPU Accelerated Edit-Distance Sequence Alignment

## 4.1 Introduction and Motivation

Nowadays, analyzing large-scale genomics workloads requires aligning millions of sequences against a reference genome under strict time constraints. This demand is further intensified by the increasing length of sequencing reads, particularly in long-read technologies, which significantly raise both computational and memory requirements. As a result, efficient and scalable alignment algorithms are essential to process these datasets within practical time frames.

Previous research has demonstrated the effectiveness of modern GPUs in accelerating high-performance computing applications, including sequence alignment. GPUs provide massive parallelism and high memory bandwidth, which can be exploited to accelerate DP and string-matching algorithms. In particular, CUDA-based implementations have shown substantial performance gains for alignment-related workloads [100, 22, 21, 23, 99, 10]. However, many existing GPU solutions are tailored to specific scoring schemes or rely on heuristic approximations, limiting their applicability in scenarios requiring exact alignment.

At the same time, recent algorithmic advances such as the Wavefront Alignment (WFA) algorithm have shown that exact sequence alignment can be computed more efficiently by exploiting the structure of the edit distance problem. WFA reduces the computational complexity by focusing on relevant diagonals of the DP matrix, making it particularly suitable for aligning similar sequences. Despite its advantages, WFA is not directly amenable to GPU acceleration. Its execution is inherently irregular, with dynamically growing wavefronts, non-uniform memory access patterns, and fine-grained dependencies across diagonals. These characteristics conflict with the execution model of GPUs, which favors regular computation, predictable memory access, and high occupancy. Consequently, existing GPU alignment methods rely on simpler DP formulations or heuristic approximations, leaving a gap between algorithmic efficiency and hardware acceleration.

This chapter is motivated by the need to bridge this gap by bringing the algorithmic advantages of WFA to GPU architectures without sacrificing accuracy. Achieving this requires rethinking both the algorithmic structure and its mapping to hardware, with particular attention

to memory footprint, parallelization strategy, and latency hiding. The goal is not only to accelerate alignment, but to do so in a way that preserves the theoretical benefits of WFA while fully exploiting GPU resources.

In this chapter, we present the first GPU implementation of the WFA algorithm for edit-distance alignment. We adapt the original formulation to better match GPU execution constraints and introduce optimizations that enable efficient parallel execution. The resulting implementation, eWFA-GPU, achieves high throughput while preserving exactness, providing a practical solution for large-scale sequence alignment.

## 4.2 Method

This section presents our proposed method to accelerate edit-distance sequence alignment using the WFA algorithm on a GPU. In the following, we present the main challenges to adapt the WFA algorithm to the CUDA programming model and the contributions and trade-offs of the proposed implementation.

There are two main strategies for parallelizing computations on GPU devices: coarse- and fine-grained. In the case of the WFA algorithm, a coarse-grained parallelization strategy assigns each CUDA thread to compute a single alignment, whereas a fine-grained strategy has multiple CUDA threads collaborate to align a single pair of sequences.

In a coarse-grained approach, each thread within the block requires its own pair of sequences and wavefront data structures to perform the alignment. Due to the limited size of the shared memory, using this approach forces storing data in global memory space, resulting in long-latency memory accesses. Moreover, a coarse-grained strategy is bound to generate divergence across threads' execution within a block, as each alignment requires a different amount of computation. Ultimately, a coarse-grain approach faces significant performance limitations that can largely reduce the overall execution speed of the algorithm on a GPU.

In contrast, a fine-grained strategy computes each alignment using a thread block. This way, all threads within the block work cooperatively to compute a single alignment problem. This approach heavily reduces the consumption of shared memory and registers, allowing the storage of the wavefront structures in shared memory for several thread blocks, which can operate concurrently in the same SM (increasing the occupancy). Furthermore, the computational pattern depicted by the WFA algorithm allows us to efficiently map the computations across the threads of a block (Figure 4.1). We exploit the fact that computations on each diagonal are independent, allowing us to compute every element in each wavefront $\widetilde{W}_e$ in parallel for both operations *extend()* and *computeNext()*. Our solution exploits this parallelism approach, where each thread block computes a single alignment problem, and each thread within the block is assigned a different diagonal offset to compute. This way, we implement the WFA to be computed using a thread block. For each wavefront $\widetilde{W}_e$ (containing $2e+1$ diagonals), threads within the block extend independently each diagonal $k$ offset (i.e., apply operator *extend()*); and then, compute the corresponding $k$ offset of the next wavefront $\widetilde{W}_{e+1}$ (i.e., apply operator *computeNext()*).

Nevertheless, this approach faces some performance challenges of its own. Concerning the memory utilization, wavefronts naturally become larger as the alignment error $e$ considered

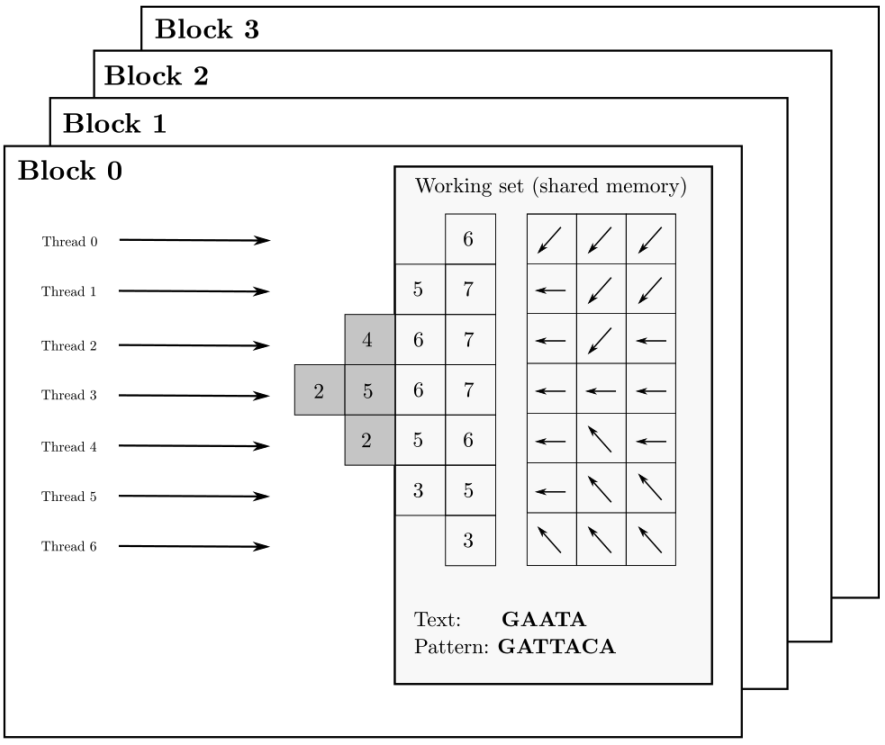



Figure 4.1: Mapping of CUDA resources into WFA work.

grows during the alignment computation (i.e., $|\widetilde{W}_e| = 1+2e$). It follows that the overall number of wavefront elements required to align a pair of sequences with alignment error $e$ is given by $\sum_{n=0}^{e} 1+2n = (e+1)^2$. Note that all the wavefronts need to be stored to retrieve the edit operations that originated the minimum edit-distance alignment. Consequently, the memory requirements grow quadratically with the alignment error, posing a scalability limitation when storing the data on shared memory. To palliate this limitation and exploit the benefits of using the fast shared memory, we propose a succinct encoding scheme where the wavefronts store partial backtraces as the alignment is computed (Section 4.2.1).

Depending on the alignment error between the input sequences, some alignments may require more shared memory than others. Requesting memory for the worst-case alignments will limit the number of concurrent thread blocks running on an SM and, ultimately, the performance of the whole execution. For that reason, we implement three different kernel specializations, each one supporting a different maximum alignment error. This way, our implementation can optimize the resource usage for each scenario and achieve higher performance for cases where the alignment error is bounded (Section 4.2.2).

Moreover, the computation performed by the *extend()* operator can be largely irregular as it depends on the number of matching characters on each diagonal. To minimize thread divergence, we use a packed sequence encoding that allows performing bit-parallel sequence comparisons (i.e., block-wise comparisons), reducing the chances of divergence, and saving memory at the same time (see Section 4.2.3).

Additionally, modern GPUs allow simultaneous data transfers and kernel execution to exploit parallelism further. In this way, the system minimizes the impact of data offloading from the host and overlaps transfer with computation on the device. Our solution implements an alignment batch system that allows multiple alignment problems to be solved in parallel while performing data transfers HtoD and DtoH (see Section 4.2.4).

### 4.2.1 Piggybacked Alignment Operations

As stated before, the WFA algorithm requires storing all the intermediate wavefront vectors $\widetilde{W}_e$ to be able to trace back the optimum alignment. As a result, the memory consumption of the algorithm grows quadratically with the alignment error, posing a severe constraint on the shared memory for scalability. Here, we propose a succinct encoding of the wavefronts based on storing partial backtraces as the alignment is computed.

For an alignment of distance $e$, the WFA backtrace algorithm computes the optimum alignment path from $(n,m)$ to $(0,0)$, traversing all the wavefront vectors from $\widetilde{W}_e$ to $\widetilde{W}_0$. In particular, each step of the backtrace checks the adjacent offsets (e.g., from $\widetilde{W}_{e,k}$ to $\widetilde{W}_{e-1,k+1}$, $\widetilde{W}_{e-1,k}$, or $\widetilde{W}_{e-1,k-1}$) for the one that originated the minimum cost alignment according to Eq. 2.4. In essence, each iteration in the backtrace process computes a step in the alignment path. To avoid storing all the wavefront offsets explicitly, we propose to explicitly compute each backtrace step (i.e., $\leftarrow, \nwarrow, \swarrow$) and store it together with the previous steps in a backtrace vector. In this way, our implementation piggybacks the partial backtraces $\widetilde{B}_{e,k}$ from every offset $\widetilde{W}_{e,k}$ to the beginning of the alignment $\widetilde{W}_{0,0}$. As a result, our solution only needs to store two wavefronts (i.e., $\widetilde{W}_e$ and $\widetilde{W}_{e+1}$) and their partial backtraces $\widetilde{B}_e$ and $\widetilde{B}_{e+1}$ for each step of the algorithm.

Figure 4.2 illustrates our proposal aligning the sequences $T =$ "*GAATA*" and $P =$ "*GATTACA*". The example shows that the alignment process ends at $\widetilde{W}_{3,-2}$ (i.e., the minimum edit-distance between $P$ and $T$ is $e = 3$). The alignment path from $\widetilde{W}_{3,-2}$ to $\widetilde{W}_{0,0}$ is explicitly stored in the backtrace vector at $\widetilde{B}_{e,k} =$ " $\leftarrow\nwarrow\nwarrow$ ".

However, the backtrace vector does not contain the full alignment path but just the edit operations (i.e., mismatches, insertions, and deletions) within the alignment. To recover the full alignment path, we need to recover the matches between backtrace steps. To that purpose, we use the WFA's *extend()* operator to compute stretches of matches between successive backtrace steps. This strategy is shown in Algorithm 6. Note that this algorithm only needs to operate once on the backtrace vector of the optimum alignment, and its time complexity is proportional to the length of the alignment path.

In practice, each backtrace step can be efficiently computed in the *computeNext()* operation and encoded in two bits (i.e., 32 backtrace steps per 64-bit word). For that, we piggyback each offset in Eq. 2.4 with its corresponding backtrace step on its two less significant bits. After the maximum calculation, the resulting backtrace step is appended to the backtrace vector at the end.

The succinct encoding of the backtrace steps leads to a significant reduction in memory consumption. Using 32-bit offsets, the straightforward implementation of the WFA algorithm requires $(e+1)^2 \times 4$ bytes to align a pair of sequences of error $e$. Using the proposed scheme, we reduce the required memory structures to the last two computed wavefronts and their corresponding backtrace vectors (i.e., $4e \times (4 + 2e/8)$ bytes). For any sufficiently large $e$, this

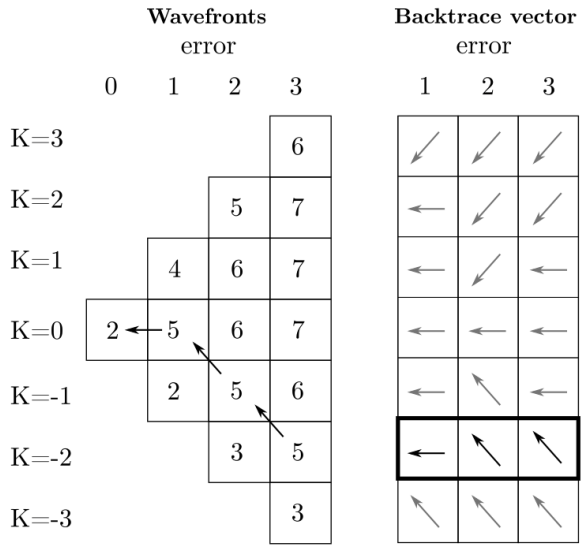



Figure 4.2: Wavefront data layout for aligning the sequences $T =$ "*GAATA*" and $P =$ "*GATTACA*".

represents up to a 4x reduction in memory usage. In practice, for moderately large $e$ values, all the backtrace vectors can be fitted in shared memory. Furthermore, to enable coalesced memory accesses and avoid bank conflicts, we implement a *struct-of-arrays* approach, separating the wavefront offsets from the backtrace vectors. As a result, subsequent backtrace vectors are stored contiguously, enabling fast accesses when all threads in a warp access the backtrace vectors.

### 4.2.2 Kernel Specialization

Although the introduction of the backtrace vectors (Section 4.2.1) reduces memory requirements, shared memory usage remains a major performance bottleneck when scaling to larger alignment errors (see Section 4.3.2). In practice, our implementation uses bit vectors to store the backtrace vectors. For instance, using 64-bit words, we could store up to 32 edit operations (i.e., each edit operation encoded using 2 bits). As the maximum alignment error increases, this approach requires longer bit-vectors. In turn, large bit-vectors put additional pressure on the shared memory usage and hinder performance. Therefore, it is important to bound the maximum alignment error for each batch of sequences and use the most suitable configuration that minimizes the memory used by the backtrace vectors.

On that account, we implemented three different kernels, each one supporting a different maximum alignment error: 32, 64, and 128 errors. For convenience, we call these kernels E32, E64, and E128, respectively. Each kernel requires storing 64-bit, 128-bit, and 256-bit words per diagonal of the wavefront and therefore requires more shared memory as the alignment error supported increases. The execution of these kernels displays different performance tradeoffs discussed in Section 4.3.2.

**Algorithm 6:** Algorithm to retrieve the alignment from the backtrace vector

```
1  Function retrieveAlignment(P,T,W̃_e,W̃_{e+1}):
2      offset ← 0
3      k ← 0
4      A ← ∅
5      for i ← 0 to e do
6          m ← extend(P,T,k,offset)
7          A ← A+'M', (m)⋯, 'M'
8          op ← B̃_k[i]
9          switch op do
10             case ↖ do // Deletion
11                 k ← k − 1
12                 A ← A+'D'
13             case ← do // Mismatch
14                 offset ← offset + 1
15                 A ← A+'X'
16             case ↙ do // Insertion
17                 offset ← offset + 1
18                 k ← k + 1
19                 A ← A+'I'
20     m ← extend(P,T,k,offset)
21     A ← A+'M', (m)⋯, 'M'
```

It is important to note that the length of the backtrace vector imposes a limit on the maximum alignment error, but not on the maximum sequence supported. For instance, the E128 implementation could be used to align sequences of 1000 nucleotides up to a 12.8% error rate or 10K long sequences up to a 1.28% error rate. For moderately long sequences (i.e., between 100 and 1000 nucleotides), our implementation supports alignments with error rates up to 10%. Nevertheless, it is possible to extend this approach to higher error rates, using longer bit-vectors, at the cost of using more memory and potential performance slowdowns (see Section 4.3.2).

### 4.2.3 Bit-parallel Packed Sequence Comparison

As opposed to the *computeNext()* operation, the *extend()* operation can require performing a different number of computations per diagonal. Specifically, the inner loop of *extend()* iterates as many times as the total number of characters that match along each diagonal. Thus, threads within a block executing this operation are bound to diverge, which can diminish the overall performance.

To mitigate this problem, we use a packed sequence encoding that allows performing bit-parallel sequence comparisons; that is, comparing blocks of characters, anticipating comparisons, and reducing the variability between diagonals. Taking advantage of the reduced DNA alphabet (i.e., nucleotides A, C, G, and T), we propose to use a 2-bit-packed encoding scheme to increase the number of nucleotides compared per block (i.e., 16 nucleotides per 32-bit word). Furthermore, packing and reducing the size of the input sequences lowers the memory requirements in shared memory and, in turn, allows fitting more CUDA blocks on the same SM.

Nonetheless, this approach introduces the need to pack the input sequences beforehand. Sequence packing can be performed on the host CPU, or it can be offloaded to the GPU. Although packing sequences on the CPU would help to reduce the amount of data that has to be transferred to the GPU, packing computations and memory transfers can be overlapped with the alignment kernels (see Section 4.2.4). Not to mention that current high-speed transfer technologies, such as NVLink, allow even faster transfers from the host to the device. For instance, using an Nvidia V100, the offloading of raw sequences and packing on the GPU turns out to be faster than packing the sequences on the CPU and transferring the packed sequences.

Furthermore, sequence packing turns out to be a straightforward operation. Due to the ASCII representation of the DNA letters (i.e., A=1000**00**1, C=1000**01**1, G=1000**11**1, T=1010**10**0), it only requires extracting the bits on positions 1 and 2 (unique bits in every DNA letter encoded in the ASCII). This encoding has been extensively used in multiple bioinformatics and genomics applications for packing DNA sequence databases and genome references. However, our implementation does not assume the preprocessing of the input sequences and allows using ASCII-encoded DNA sequences, packing their content on the GPU.

Altogether, this approach accelerates the computations performed within the *extend()* kernel, decreasing the execution divergence between threads, and reducing the number of instructions executed as well as the overall shared memory used. Compared to the vanilla implementation, our experiments show that this strategy accelerates the kernel execution time from 1.6× to 1.9× and reduces the number of executed instructions by a factor of 1.7× to 2.1×. Most importantly, it reduces between 1.2× and 1.7× the number of predicated-off threads in a warp (i.e., inactive threads when divergent branches occur and threads take separate paths).

### 4.2.4 Overlapping kernel computation with data transfers

At the system level, memory transfers from host to device take a significant percentage of the total execution time since all the sequences have to be stored in the device to perform the alignment. Hiding transfer latencies with computation is key to avoiding performance slowdowns due to the offloading of computation to the GPU. The CUDA programming model allows the creation of various streams to overlap computing kernels with memory transfers. All operations within a CUDA stream are synchronous; however, they can operate asynchronously between other running streams. As a result, launching independent kernels and memory transfers to different CUDA streams can effectively overlap computation with memory transfers.

To effectively implement this strategy, we created batches of sequences to be transferred and aligned in parallel. This way, the computation kernels of a given batch can be overlapped with computations and memory transfers from other batches. This concept is illustrated in Figure 4.3.

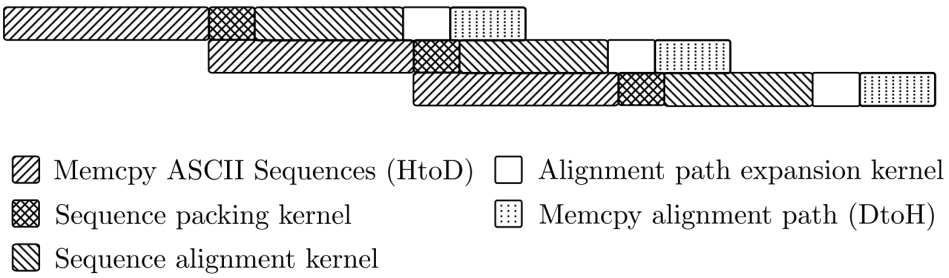



Figure 4.3: Compute kernels of multiple batches are overlapped with data transfers (DthH and HtoD).

## 4.3 Experimental Results

### 4.3.1 Performance evaluation

Table 4.1: Alignment time (in milliseconds) for an input of 10 million alignments using different alignment implementations on CPU and GPU. Note that CPU implementations were executed using 80 threads. The first half of the table presents alignment times of implementations that only compute the edit-distance (not the alignment). The second half shows alignment times of implementations that compute the edit-distance and the full alignment path. Best execution times are marked in bold.

| | | | length=150 | | | length=300 | | | length=1000 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | e=2% | e=5% | e=10% | e=2% | e=5% | e=10% | e=2% | e=5% | e=10% |
| Distance only | GPU | NVBIO.DP | 622 | 656 | 642 | 1013 | 1020 | 1050 | 2534 | 2506 | 2446 |
| | | NVBio.BPM | 608 | 630 | 658 | 856 | 833 | 894 | 2474 | n/a | n/a |
| | | eWFA-GPU | **89** | **92** | **100** | **143** | **140** | **197** | **399** | **487** | **994** |
| Full alignment | CPU | Edlib | 8928 | 9001 | 9057 | 16853 | 16338 | 16591 | 120248 | 50988 | 59799 |
| | | O(ND) | 7872 | 7798 | 7444 | 14456 | 14350 | 13873 | 21217 | 38553 | 36110 |
| | | BPM | 6773 | 7204 | 6985 | 12484 | 12340 | 12526 | 46062 | 47043 | 45610 |
| | | eWFA | 577 | 758 | 1110 | 672 | 1150 | 2090 | 1310 | 3900 | 11500 |
| | GPU | GASAL2 | 1206 | 1228 | 1239 | 4365 | 4366 | 4394 | n/a | n/a | n/a |
| | | wmCudaTile | 734 | 1074 | 1473 | 2142 | 2836 | 5038 | 13706 | 30571 | 108447 |
| | | eWFA-GPU | **91** | **95** | **116** | **144** | **160** | **252** | **453** | **689** | **1928** |

To present a comprehensive evaluation of the performance of different methods, Table 4.1 shows the alignment time for each implementation when aligning 10 million sequences of varying lengths and error rates. We report the total execution time, including transfer times (i.e., host-to-device and device-to-host), for GPU executions. All CPU implementations were executed using 80 threads. Overall, results show that eWFA-GPU executes 2.9-265× faster than the CPU-based methods and 8-56× faster than other GPU implementations.

Compared to established CPU alignment algorithms, eWFA-GPU performs 24-102× faster than the BPM algorithm and 19-100× faster than the O(ND) implementation. Similarly, we obtain speedups of 31-265× compared to Edlib. Compared to the CPU implementation of the eWFA, our GPU implementation delivers 3-9× times more performance. In particular, the

speedups obtained by eWFA-GPU increase with higher alignment error rates as the wavefronts increase in size and more wavefront computation can be done in parallel (see Section 4.3.2).

Regarding the GPU implementations, eWFA-GPU outperforms the widely-used NVBio library, achieving speedups of 2.5-7.4× compared to NVBio's classical DP-based implementation and speedups of 4.5-7.2× compared to NVBio's BPM. Compared to wmCudaTile, eWFA-GPU achieves up to 12× speedup for short sequences (i.e., 150 nucleotides) and up to 56× speedup for longer sequences. Compared to GASAL2, eWFA-GPU is 10-30× faster. In general, eWFA-GPU execution time scales better with the sequence length, compared to the other GPU implementations. In particular, the performance of DP-based methods, like GASAL2, is strongly limited by the sequence length. Ultimately, aligning long sequences with GASAL2 becomes impractical (e.g., 1000 nucleotides or more). For a fair comparison, it is important to acknowledge that GASAL2 implements the gap-affine distance, which is more complex and costly than computing the edit-distance alignment.

Unsurprisingly, DP-based implementations (i.e., BPM, Edlib, NVBio, and GASAL2) are insensitive to the alignment error, performing the same amount of computations to align similar sequences as to align very divergent ones. As a result, the performance of classical DP-based algorithms is heavily constrained by the sequence length and not by the sequence's homology. For that reason, some tools, like Edlib, implement heuristics that prune the DP computations at the expense of potentially missing the optimal alignment (note the reduction in the execution time when aligning sequences of 1000 nucleotides with $e >= 5\%$). In contrast, error-sensitive methods, like the eWFA-GPU, perform faster when aligning highly similar sequences by exploiting their similarities to accelerate the alignment process. These methods are only constrained by the nominal amount of differences between the sequences.

### 4.3.2 Profiling, Scalability, and Limitations

Our solution relies on exploiting the fast on-chip memory of the GPU to improve the execution time. As explained in Section 4.2, our implementation stores the algorithm's working set (i.e., sequences, offsets, and backtraces) in shared memory, enabling fast accesses at the expense of limiting the maximum amount of memory that each alignment can use. As the algorithm's shared memory requirement grows quadratically with the alignment error, the memory consumed by the offsets and backtraces becomes the most limiting factor. In turn, increasing the shared memory consumed per alignment limits the number of thread blocks that can be executed concurrently on each SM. Therefore, the maximum supported alignment error strongly constrains the number of alignments that can be processed on each SM, thus limiting the performance and scalability of the solution to high error rates. Due to these limitations, our solution implements three specialized alignment kernels, each supporting a different maximum number of errors per alignment (i.e., 32, 64, and 128 errors; see Section 4.2.2). In this section, we show that selecting the appropriate kernel, tuned to the maximum expected alignment error, is crucial for achieving optimal performance.

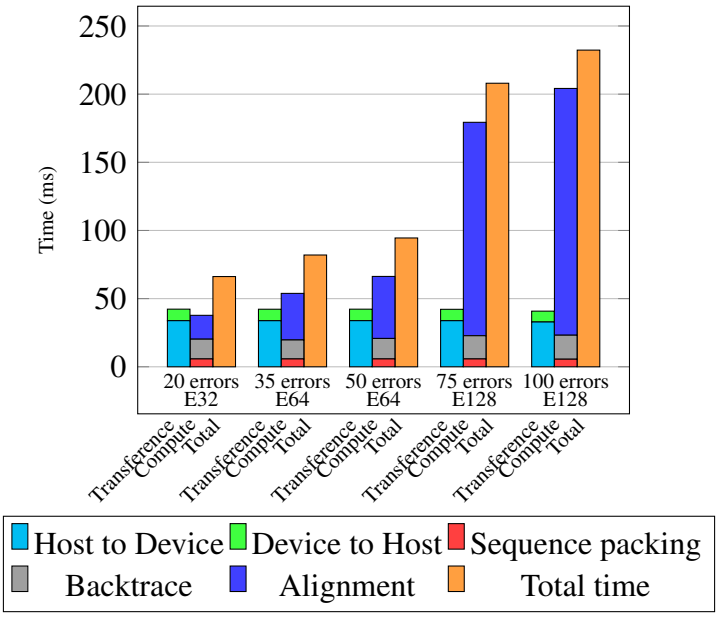



Figure 4.4: Application execution time is broken down into transfer time (i.e., host to device and device to host), kernel execution time, and total execution time. Each execution was performed using a dataset containing 1 million sequences of 1000 nucleotides with different error rates (i.e., 20, 35, 50, 75, and 100 nominal errors).

#### 4.3.2.1 Overall System Profiling

Having three different specialized kernels, the performance of the executions changes depending on the alignment error between the sequences. Fig. 4.4 shows the application execution times aligning datasets with different error rates, broken down into transfer time (i.e., HtoD and DtoH), kernel execution time, and total execution time. In the figure, each execution is represented using three columns: the first one showing the aggregated time of the memory copies between CPU and GPU, the second one showing the GPU kernels computation times, and the third one showing the overall execution time. Note how transfer times are being effectively overlapped with kernel computations.

In particular, when aligning homologous sequences (e.g., 20 differences between the sequences) with the E32 kernel, we observe that data transfers become the main performance bottleneck. In this case, the kernels' computation can be effectively overlapped with transfers (disregarding initialisation times), resulting in the fastest execution times. As the number of differences increases, our implementation requires using kernels that support higher error rates. In these scenarios (E64 and E128), the kernel's computing time overtakes the transfer time and becomes the main bottleneck. Most notably, the alignment kernel time increases with higher error rates (especially due to increments in the size of the backtrace vectors from E64 to E128). As opposed to transfer, packing, and backtrace times remain constant across all executions for all datasets used (i.e., sequences of 1000 nucleotides).

Table 4.2: Performance metrics of each specialised alignment kernel on the Nvidia V100 GPU. Executions were performed using datasets of 1M sequences of 1000 nucleotides. Each dataset contains sequences that align with an average error rate of 2%, 5%, and 10% (i.e., 20, 50, and 100 nominal differences). Each execution was performed using the minimum alignment error supporting kernel (i.e., E32, E64, E128)

| **Dataset:** | 1000nt 2% error | 1000nt 5% error | 1000nt 10% error |
|---|---|---|---|
| Alignment kernel executed | E32 | E64 | E128 |
| Maximum error supported | 32 | 64 | 128 |
| Threads per block. | 32 | 64 | 128 |
| Shared memory per block (KiB) | 2.14 | 5.74 | 19.08 |
| Occupancy (active warps per SM) | 31.86 | 31.87 | 19.94 |
| Kernel time (ms) | 17.27 | 45.19 | 190.36 |
| SM busy (%) | 87.74 | 82.50 | 61.96 |
| Global memory throughput (GiB/s) | 35.86 | 14.05 | 3.46 |
| Executed warp instructions (x$10^9$) | 6.31 | 15.47 | 48.82 |
| Avg. active threads per warp | 10.10 | 21.36 | 27.40 |
| Warp cycles per issued instruction | 9.08 | 9.66 | 8.04 |

#### 4.3.2.2 Alignment Kernel Performance Profiling

Due to its significance, we focus on the alignment kernel to characterize its performance and understand the GPU resource utilization. Table 4.2 reports a summary of the most relevant performance metrics of the execution of the three alignment kernel specializations.

Concerning memory utilization, the alignment kernel only accesses global memory at the beginning of the execution to copy the input sequences into shared memory. Due to the limited usage of global memory, the effective throughput reached is very low and rapidly decreases as the compute time grows for executions using higher alignment error rates. For the rest of the execution, the alignment kernel only accesses the fast on-chip shared memory.

Regarding computation on the GPU, in Table 4.2 we observe that all the alignment kernel specializations are consistently between 60% and 87.74% of the maximum SM core instruction throughput (i.e., SM busy). Furthermore, a more detailed profiling reveals that none of the SM computing pipelines is fully saturated. In particular, the most used computing pipeline, the ALU pipeline, reaches a 87% utilization on the E32 kernel, 80% on the E64 kernel, and 30% utilization on the E128 kernel. Additionally, note that the warp stall time (i.e., warp cycles per issued instruction) remains similar across all executions.

These results reveal that the real limiting factor of these executions is not the lack of computing resources on the GPU but the lack of computing parallelism. When aligning up to higher alignment error rates, the wavefronts become larger, and thus, an SM can exploit more threads to perform parallel computations. Accordingly, Table 4.2 shows that the average active threads

per warp increases from 10.1 to 27.4 (out of a maximum of 32 threads per warp) when executing kernels with higher alignment error support. In turn, this increase in parallelism is reflected in the total warp instructions executed. As the alignment error increases, we would expect an $O(e^2)$ increase in the number of warp instructions. However, we observe a much gentler growth alleviated by the utilization of more threads per warp.

Nevertheless, this increase in the number of active threads per warp does not immediately translate into higher SM utilization (i.e., SM busy). Note that higher-alignment-error supporting kernels require more shared memory per block (Table 4.2). Therefore, the maximum number of active warps per SM is bounded by the total shared memory available and the shared memory required per block. Table 4.2 shows that the occupancy drops from 31.86 to 19.94 when aligning sequences up to 100 nominal differences using the E128 kernel. As a result, the SM is busy, and the computing pipelines usage is reduced from 87.74% to 61.96%. Ultimately, as the profiling results show, the performance of the alignment kernel execution attends to a trade-off between the shared memory required by each thread block and the maximum active threads per warp that can be exploited to perform the alignment computations.

#### 4.3.2.3 Alignment Kernel Selection

In order to maximize performance, it is crucial to select the alignment kernel that minimizes the shared memory consumption while being capable of aligning up to the maximum error required by the input dataset. Table 4.3 presents the performance results from using the three different kernel specializations to align the same dataset. First, we can observe that each alignment kernel gradually requires more shared memory (from 1.65KiB to 18.61KiB per thread block), reducing occupancy (from 31.50 to 4.88) and ultimately leading to longer kernel execution times (i.e., a slowdown of 11× from E32 to E128). When using the same dataset, all three executions compute the same alignments and process wavefronts of the same length. Consequently, the effective parallelism attained is the same for all the kernels (i.e., average active threads per warp), and the executed warp instructions remain constant for all the executions (ignoring overheads associated with operating with longer backtrace vectors). Hence, the maximum amount of parallel computations depends on the maximum alignment error, not on the alignment kernel specialization. Considering that the three kernels are capable of supporting the maximum alignment error of the dataset, selecting an oversized kernel can lead to a slowdown of up to 3.8×.

In conclusion, utilizing the best-fitted kernel (in terms of maximum alignment error supported and shared memory consumed) is key for performance. Specifically, for alignments with a small alignment error, where the parallelism is rather limited and only a few threads per block can effectively compute useful work in parallel. Balancing the number of alignments per SM and the maximum number of active threads per block is crucial for an efficient exploitation of the GPU computing resources.

### 4.3.3 Parameter Tuning

Most often, GPU-based implementations depict specific parameters that can strongly impact performance and have to be configured with caution. For the eWFA-GPU, the number of

Table 4.3: Performance metrics of each specialized alignment kernel on the Nvidia V100 GPU. All executions were performed using 32 threads per block, aligning a dataset of 1M sequences of 150 nucleotides with an average error rate of 5% (i.e., average of 7.5 nominal differences). Each execution was performed using a different alignment kernel; that is, E32, E64, and E128.

| **Dataset: 150 nucleotides (5% alignment error)** | Kernel E32 | Kernel E64 | Kernel E128 |
|---|---|---|---|
| Maximum error supported | 32 | 64 | 128 |
| Shared memory per block (KiB) | 1.65 | 5.27 | 18.61 |
| Occupancy (active warps per SM) | 31.50 | 17.63 | 4.88 |
| Kernel time (ms) | 3.93 | 4.91 | 14.96 |
| SM busy (%) | 92.95 | 79.37 | 28.43 |
| Global memory throughput (GiB/s) | 37.15 | 33.00 | 12.43 |
| Executed warp instructions (x$10^9$) | 1.51 | 1.61 | 1.76 |
| Avg. active threads per warp | 10.48 | 11.34 | 13.03 |
| Warp cycles per issued instruction | 8.46 | 5.54 | 4.28 |

threads per block (and, therefore, the number of threads per alignment) determines the maximum work that can be done in parallel computing an alignment. If there are more threads than wavefront elements, some threads never perform useful work, and GPU resources are not efficiently used. Conversely, if a wavefront is larger than the number of threads in a block, the implementation requires multiple iterations; hence, losing parallelism.

Table 4.4 lists the performance trade-offs using a different number of threads per block. For short sequences and small error rates, using small blocks (e.g., one warp) reduces the number of idle threads per block. For short and medium sequences (i.e., 150-300 nucleotides), using 32 or 64 threads per block yields very similar performance. However, using more threads per block leads to a performance drop caused by idle threads consuming GPU resources. Similarly, for aligning long sequences (i.e., 1000 nucleotides), the best performance is achieved by using 128 threads per block. Note that using fewer threads per block leads to an underutilization of GPU resources and, using more threads, to a waste of GPU resources by idle threads.

Table 4.4: Alignment time (in milliseconds) of 10 million alignments using a different number of threads per block. All the datasets used for this comparison have a 10% error rate.

| Nucleotides | Threads per block | | | |
|---|---|---|---|---|
| | 32 | 64 | 128 | 256 |
| 150 | **116.2** | 121.3 | 149.5 | 228.9 |
| 300 | **252.0** | 255.8 | 327.9 | 436.6 |
| 1000 | 2795.2 | 2091.0 | **1928.1** | 2092.4 |

### 4.3.4 Evaluation on Other Devices

To provide a detailed analysis of the proposed solution's performance, we also evaluated our implementation on two additional GPU models: an Nvidia GeForce RTX 2080 Ti and an Nvidia GeForce RTX 3080. The computing capabilities of each device used are listed in Table 3.2.

The results of the execution on other GPU devices are shown in Table 4.5. On the GeForce RTX 2080 Ti, our implementation is limited by the CPU-GPU bandwidth. The device is connected through PCI Express, achieving a bandwidth of 13GB/s on average. For instance, a batch of 10 million sequences of 1000 nucleotides represents 21GB of input data. Transferring all this data to the GPU using the available peak bandwidth of 13GiB/s would take 1615 milliseconds. That is about 87% of the total execution time. Even with the proposed strategy to overlap computation with data transfers, the overall execution time is still bounded by the time required for data transfers to the device.

In the case of the RTX 3080, most execution times are similar to the RTX 2080 results, as they have similar PCI Express bandwidth. Overall, computation kernels are faster than memory transfers and can be effectively overlapped. However, when aligning 1000-nucleotide-long sequences with 10% of error, computation kernels take more time than memory transfers, mainly due to the intensive usage of shared memory. As shown in Table 3.2, the RTX 3080 has more shared memory available per SM than other devices, allowing it to have more alignments per SM, and therefore, achieving better performance than the RTX 2080.

Table 4.5: Alignment time (in milliseconds) of 10 million alignments using eWFA-GPU on different devices.

| Average nucleotides | Error | V100 | RTX 2080 Ti | RTX 3080 |
|---|---|---|---|---|
| 150 | 2% | 91 | 299 | 333 |
| | 5% | 95 | 298 | 332 |
| | 10% | 116 | 301 | 335 |
| 300 | 2% | 144 | 555 | 585 |
| | 5% | 160 | 563 | 585 |
| | 10% | 252 | 565 | 590 |
| 1000 | 2% | 453 | 1864 | 1921 |
| | 5% | 689 | 1878 | 1989 |
| | 10% | 1928 | 2900 | 2103 |

## 4.4 Discussion

This chapter presents eWFA-GPU, a GPU-accelerated algorithm based on the WFA formulation for exact edit-distance alignment. The implementation preserves optimality while achieving substantial performance improvements over existing approaches. Moreover, this work introduces several algorithmic and architectural optimizations that adapt wavefront-based alignment to the constraints and opportunities of GPU architectures.

A central contribution is the piggybacked backtrace strategy, which significantly reduces memory requirements during alignment. By encoding traceback information within the wavefront computation, the algorithm avoids storing the full DP state, requiring only a small number of active wavefronts. This design allows the working set to fit within fast on-chip memory, improving both memory efficiency and execution speed. In addition, we introduce a high-performance sequence packing kernel that enables efficient block-wise comparisons, accelerating the *extend* operation, a dominant component of the WFA algorithm. These optimizations are complemented by a fully asynchronous execution model that overlaps kernel execution with memory transfers, effectively hiding data movement latency and improving overall throughput.

Experimental results demonstrate that eWFA-GPU consistently outperforms state-of-the-art CPU and GPU alignment libraries. On the NVIDIA V100 GPU, our implementation achieves speedups of up to 265× over Edlib and 9.2× over the CPU version of WFA. Compared to other CPU baselines, we obtain speedups of up to 101.7× over BPM and 100.4× over the O($ND$) algorithm. Compared with GPU-based aligners, eWFA-GPU achieves up to 56.2× speedup over wmCudaTile, 30.3× over GASAL2, and 7.4× over NVBio. GASAL2 computes gap-affine alignment, which entails a more complex scoring model and higher computational cost, making direct comparisons conservative.

The results highlight several important observations. First, adapting the algorithm to the memory hierarchy is critical, as reducing memory footprint enables higher occupancy and better utilization of GPU resources. Second, the *extend* operation remains a key performance bottleneck, and optimizing it directly impacts overall throughput. Third, overlapping computation and data transfers is essential to fully exploit the GPU execution model, particularly for large-scale batch processing. These insights inform the design of subsequent contributions in this thesis, where both algorithmic structure and hardware-aware optimizations are co-designed.

Overall, eWFA-GPU provides an efficient and scalable solution for applications requiring exact edit-distance alignment over large genomic datasets. The implementation is released as open-source software under the MIT license and is publicly available at eWFA-GPU repository, facilitating reproducibility and adoption in practical bioinformatics pipelines.

# Chapter 5

# Gap-Affine Sequence Alignment on GPU

## 5.1 Introduction and Motivation

In Chapter 4, we demonstrated that the WFA algorithm can be effectively adapted to GPU architectures, achieving high-performance exact edit-distance alignment. However, that design presents two important limitations. First, it relies on the edit-distance scoring model, which assigns unit costs to individual edits rather than distinguishing gap opening from gap extension. Second, the design is constrained by on-chip memory, which limits the maximum number of errors that can be processed and scalability for more challenging alignment problems.

In practical bioinformatics applications, gap-affine scoring is the standard model, as it provides a more realistic representation of sequence evolution. Supporting this model introduces additional complexity, as it requires handling multiple dependent wavefronts and more intricate recurrence relations. At the same time, real-world workloads often involve long and noisy sequences, in which the number of errors can exceed the capacity of fast on-chip memory, necessitating designs that scale beyond these constraints.

Extending WFA to gap-affine scoring while preserving GPU efficiency presents some challenges. The gap-affine formulation increases both the number of states and the dependencies among them, complicating parallelization and memory management. Moreover, removing the shared-memory limitation requires redesigning the data structures and execution strategy to operate efficiently on global memory, without degrading performance due to increased latency.

In this chapter, we address these challenges by introducing a GPU-accelerated design and implementation of WFA that supports gap-affine alignment and removes the limitations on the maximum number of errors. We extend the piggyback backtrace strategy to operate under this more general scoring model and redesign the memory layout to scale with the available GPU DRAM. The resulting approach enables exact gap-affine alignment on GPUs while maintaining high throughput, bridging the gap between biologically-accurate results and hardware-efficient execution.

## 5.2 Method

In this section, we present the parallel WFA-GPU algorithm, the main performance challenges, and effective solutions to mitigate these problems. This way, Section 5.2.1 presents an overview of the parallel algorithm and its mapping to the GPU computing architecture. Afterwards, we discuss the principal performance limitations of its GPU implementation.

First, we show that the memory requirements grow quadratically as the alignment error increases, limiting the scalability of the implementation when aligning tens of thousands of sequences in parallel. To alleviate this problem, in Sections 5.2.2 and 5.2.3, we present an efficient GPU memory management strategy and an algorithmic technique to reduce the overall memory used by the WFA on a GPU.

Second, we argue that the number of operations to compute successive wavefronts becomes a limiting factor when aligning large and noisy sequences. In Section 5.2.4, we present a strategy to accelerate the *LCP()* computation. In Section 5.2.5, we propose a GPU-aware heuristic extension to reduce the volume of computations performed on the GPU.

Finally, in Section 5.2.6, we present a CPU-GPU co-design, which allows overlapping the execution of GPU kernels, CPU tasks, and data transfers (between the memories of the two devices) to improve the overall performance of the implementation.

### 5.2.1 WFA-GPU parallel algorithm

Some parallel solutions to the sequence alignment problem are based on exploiting inter-sequence parallelism (i.e., letting each GPU thread compute a different alignment). However, as parallelism increases, aligning a large number of sequences simultaneously requires unfeasible amounts of memory. Moreover, differences in the workload (e.g., sequence length and alignment error) cause variations in the execution flow of each alignment, generating thread divergence. Alternatively, it is feasible to exploit parallelism within a single alignment task (i.e., intra-sequence parallelism). However, a single alignment rarely allows for exploiting the massive amounts of parallel resources available on modern GPUs. Our proposal is to combine both parallel strategies (inter- and intra-parallelism) to compute multiple alignments on concurrent thread blocks, each using several threads cooperatively to calculate a single alignment.

The WFA algorithm depicts a simple computational pattern to calculate each sequence alignment. Eq. 2.3 shows that a wavefront $\widetilde{W}_s$ is computed using only wavefronts $\widetilde{W}_{s-o-e}$, $\widetilde{W}_{s-e}$, and $\widetilde{W}_{s-x}$. More importantly, each diagonal offset $\widetilde{W}_{s,k}$ can be computed independently. Fig. 5.1 illustrates in detail the parallel computation of a given wavefront $s$ using multiple GPU threads, exploiting intra-sequence parallelism (ⓑ). At the same time, other alignments can be computed by different thread blocks on the GPU using inter-sequence parallelism (ⓐ).

Algorithm 7 presents the high-level pseudocode of the WFA-GPU. For each alignment, multiple threads cooperate to compute consecutive wavefronts. In particular, for every score $s$ and diagonal $k$, each GPU thread in the block computes $\widetilde{W}_{s,k}$ independently (lines 7-10). After every diagonal of wavefronts $\widetilde{W}_s$ is computed, GPU threads synchronize (line 11) and proceed to compute the following wavefronts $\widetilde{W}_{s+1}$. This process is repeated until the optimal alignment is found (line 4).

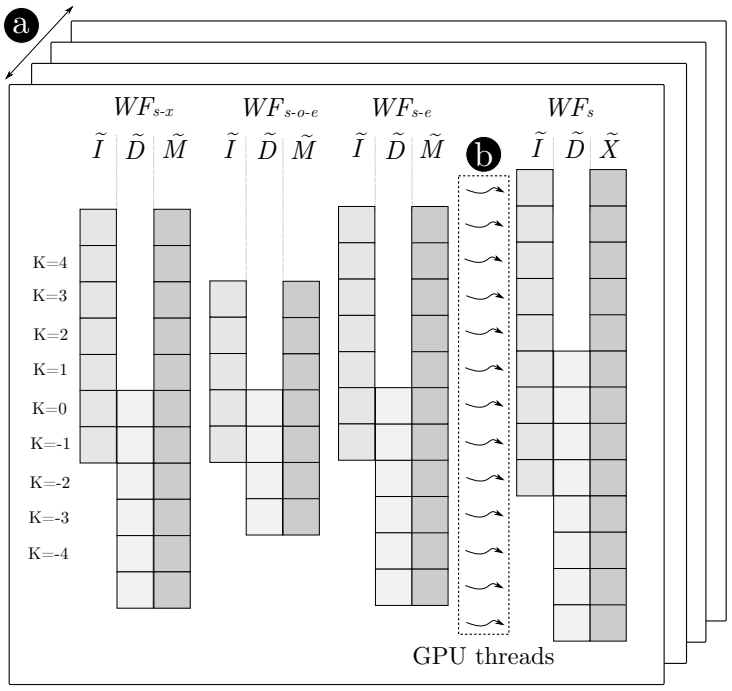



Figure 5.1: Multiple alignments are computed concurrently on the GPU, exploiting inter-sequence parallelism (a). At the same time, multiple GPU threads compute different diagonals of a given wavefront ($\widetilde{W}_s$) in parallel, exploiting intra-sequence parallelism (b).

Each GPU thread concurrently undertakes the computation of the data item of a different diagonal offset, applying Eq. 2.3 and using the previously computed wavefronts and the *LCP()* function. Notably, as the algorithm progresses, wavefronts become increasingly larger, and the potential parallelism of the problem grows. For large and noisy sequences, the problem becomes embarrassingly parallel, allowing us to perform up to $2s+1$ parallel computations for each $\widetilde{W}_s$. Such a highly parallel algorithm is suitable for modern GPUs.

### 5.2.2 Alignment Scheduling and GPU Memory Management

A simple and naive implementation would spawn a thread block for each WFA alignment offloaded to the GPU. However, each WFA alignment kernel requires GPU memory to store all the intermediate wavefronts. It is not feasible to reserve GPU memory for every alignment in advance when processing tens of thousands of sequence alignments. However, it is possible to request an upper bound of the total WFA memory required for a number of alignments that can be processed in parallel on the GPU at the same time.

Thus, our implementation creates a pool of outstanding WFA aligners and allocates memory for as many alignment blocks as can be processed simultaneously on the GPU. Then, an alignment scheduler assigns WFA alignments to thread blocks. Whenever a thread block finishes an alignment, it requests another from the alignment pool until all alignments offloaded to the GPU have been completed. We guarantee the correct synchronization of the different aligners by using atomic operations and a global counter.

**Algorithm 7:** WFA-GPU parallel algorithm.

**Input:** $q,t$ strings, $\{x,o,e\}$ gap-affine penalties

**Function** WFAGPU_ALIGN_KERNEL($q,t,\{x,o,e\}$) **begin**
  `// Initial conditions`
  $\widetilde{M}_{0,0} \leftarrow LCP(q_{0\ldots n-1}, t_{0\ldots m-1})$;
  $s \leftarrow 0$;
  **while** $\widetilde{M}_{s,m-n} \neq m$ **do**
    $s \leftarrow s+1$;
    **parallel foreach thread** $k$ ***in*** *diagonals*($\widetilde{W}_s$) **do**
      `// Compute wavefronts with score` $s$
      $\widetilde{I}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k-1}+1, \widetilde{I}_{s-e,k-1}+1\}$;
      $\widetilde{D}_{s,k} \leftarrow \max\{\widetilde{M}_{s-o-e,k+1}, \widetilde{D}_{s-e,k+1}\}$;
      $\widetilde{X}_{s,k} \leftarrow \max\{\widetilde{M}_{s-x,k}+1, \widetilde{I}_{s,k}, \widetilde{D}_{s,k}\}$;
      `// Compute LCP()`
      $\widetilde{M}_{s,k} \leftarrow \widetilde{X}_{s,k} + LCP(q_{\widetilde{X}_{s,k}-k\ldots n-1}, t_{\widetilde{X}_{s,k}\ldots m-1})$;
    **Synchronize_threads();**

Nevertheless, having to allocate GPU memory beforehand forces us to estimate the maximum memory required by each WFA alignment in advance. For that, our method establishes a configurable upper bound on the required memory based on a conservative estimation of the maximum error rate (i.e., 10% of sequence length by default on our example tool). Nonetheless, some alignments may override initial estimations and require more memory. For those cases, the WFA-GPU implements a rescue mechanism that returns the alignment to the CPU to be computed using the original WFA algorithm. In practice, when aligning long and noisy sequences (like those produced by PacBio or Nanopore Technologies), the amount of rescued alignments is below 0.2%. Furthermore, the computation of the rescued alignments can be performed in the host CPU; meanwhile, the GPU is computing other alignments, as described in Section 5.2.6.

Although modern GPUs are equipped with large DRAM memories, access to global memory is relatively slow and can potentially reduce the performance of GPU applications. To take advantage of fast on-chip memories and minimize the latency of global memory accesses in the GPU, our implementation allocates the most frequently accessed wavefront diagonals (i.e. the central diagonals) of wavefronts in the shared memory. This way, the WFA-GPU benefits from fast on-chip memory access to the elements of the central diagonals.

### 5.2.3 Piggybacked Backtrace Operations

For an alignment with optimal score $s$, the WFA algorithm requires storing all the intermediate wavefronts up to $\widetilde{W}_s$ to be able to retrieve the alignment path (a.k.a. CIGAR) during the final backtrace step. However, alignments with a large nominal number of errors require a non-negligible amount of memory. That is, an upper-bound of $3\sum_{i=0}^{s} 1+2i = 3(s+1)^2$ wavefronts offsets, consuming up to $12(s+1)^2$ bytes per alignment. These memory requirements

become impractical when aligning multiple noisy sequences in parallel, even for modern GPUs equipped with large amounts of global memory.

To reduce memory consumption, our method piggybacks the backtrace operations (i.e., **X**, **I**, and **D**) onto the wavefronts as they are computed. Using only two bits, each backtrace operation is encoded in a bitmap stored for every diagonal of the wavefront. Therefore, for a given score $s$ and diagonal $k$, our method stores a bitmap with the alignment operations required to reach $\widetilde{W}_{s,k}$. It follows that the bitmap associated with $\widetilde{M}_{s,m-n}$ contains the backtrace of the optimal alignment. Note that these bitmaps do not grow regularly, as a different number of backtrace operations may have the same score (e.g., multiple insertions may be equivalent to a mismatch).

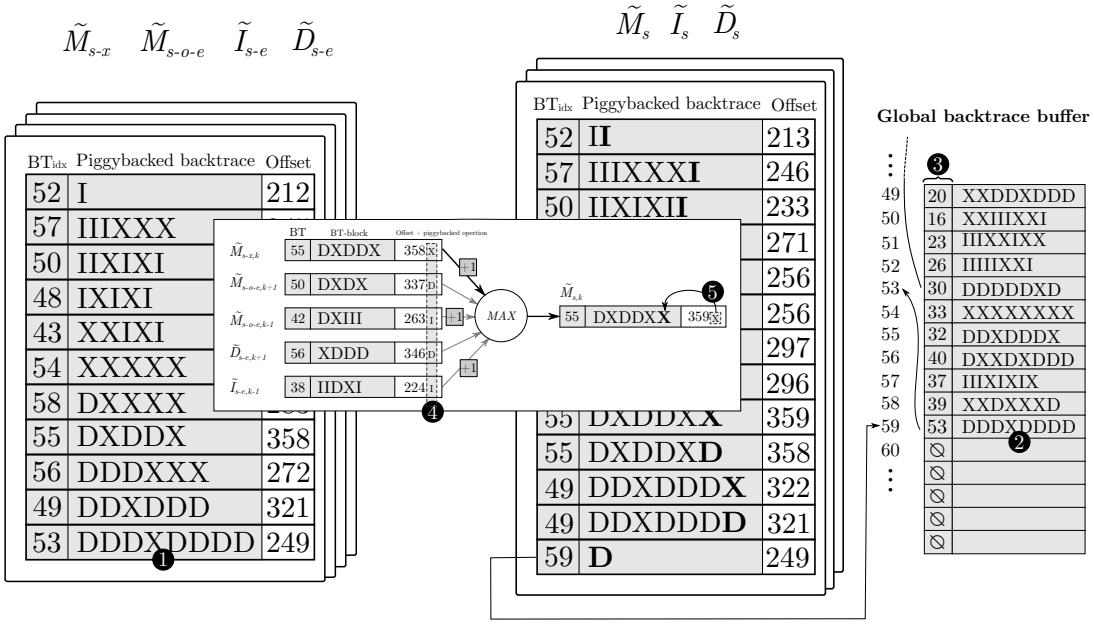



Figure 5.2: Illustration of the piggybacked backtrace strategy and data-layout organization. From left to right of the figure, we show the source wavefronts ($\widetilde{M}_{s-x}$, $\widetilde{M}_{s-o-e}$, $\widetilde{I}_{s-e}$, and $\widetilde{D}_{s-e}$) and how they are combined to generate wavefronts at score $s$. Each diagonal has a BT-block (shaded in grey) and an offset. At the center, we detail the process of computing Eq. 2.3 for a single diagonal and the piggyback of the corresponding backtrace operation. At the bottom, the global BT-buffer is depicted, where each slot represents a BT-block (displayed vertically for better readability of the figure).

In practice, our implementation uses 32-bit bitmap words to store backtrace operations (i.e., BT-block). Once a BT-block of a diagonal is full and cannot encode more backtrace operations (❶ on Fig. 5.2), it is offloaded to a global backtrace buffer (❷) (i.e., BT-buffer). Each BT-block stores an index to the previous BT-block (❸) in the chain that allows retrieving the complete alignment backtrace associated with any $\widetilde{W}_{s,k}$ offset. A complete alignment backtrace is recovered by traversing the linked BT-blocks starting from the last one.

The computation of each backtrace operation is coupled with the computation performed in Algorithm 7 to generate each diagonal offset. For that, the corresponding backtrace operation (i.e., **X**, **I**, and **D**) is piggybacked to each source wavefront offset ($\widetilde{M}_{s-o-e,k-1}$, $\widetilde{M}_{s-o-e,k+1}$, $\widetilde{I}_{s-e,k-1}$, $\widetilde{D}_{s-e,k+1}$, and $\widetilde{M}_{s-x,k}$) in Eq. 2.3, using the two least significant bits (❹). Then, as a

byproduct of the computation of the maximum offset, the corresponding backtrace operation is found piggybacked, and is added to the current BT-block (❺). In practice, this strategy turns out to be computationally lightweight.

Note that BT-blocks only contain edit operations (i.e. **X**, **I**, **D**) and not the matches in between. To retrieve the complete alignment CIGAR, the algorithm needs to compute any missing matches between backtrace operations. Nonetheless, this is a remarkably simple operation. Using the same *LCP()* function presented earlier, the algorithm computes matches until a mismatch is found. Then, it adds the following backtrace operation and proceeds again to compute the *LCP()*. This process halts when all the backtrace operations from the chain of BT-blocks have been processed.

However, the piggyback strategy introduces some extra operations when computing Eq. 2.3 to add the backtrace operation to the lower bits of the offset. Nevertheless, this operation can be implemented using fast bitwise instructions, adding a low overhead. Additionally, an atomic operation is needed when offloading a BT-block into the global backtrace buffer.

Overall, the piggyback strategy effectively reduces the memory consumed by the wavefronts to 4 bits per entry (accounting for the BT-block indices). Compared to storing the raw wavefront offsets as the original WFA does (i.e., 4 bytes per entry), this strategy represents an $8\times$ reduction.

### 5.2.4 Bit-Parallel Sequence Comparison using Packed DNA Sequences

WFA's execution time is dominated by the computation of the *LCP()* function [32]. A naive implementation would compare sequences character by character until a non-matching character is found. This approach not only executes a non-negligible amount of instructions per *LCP()* call but also creates divergence across threads computing the same alignment. That is, each GPU thread within a block performs a different number of comparisons depending on the characters being compared. Because GPU threads execute in groups of 32 threads in lock-step mode, divergent execution (i.e., variable-iteration loops) forces idle threads to wait until all threads have finished iterating.

To alleviate this problem, we propose a bit-packed encoding of DNA sequences using 2 bits per base. The encoding turns out to be remarkably simple, as the ASCII representation of each base has two unique bits on positions 1 and 2 (i.e., A=1000**00**1, C=1000**01**1, G=1000**11**1, T=1010**10**0). This process is also well-suited for GPU execution, as each character can be computed in parallel. Sequence packing is done at runtime, just before the WFA-GPU alignment kernel starts. Using a bit-packed representation, our implementation compares blocks of 16 bases at once using 32-bit operations. This strategy reduces execution divergence and, most importantly, the total number of instructions executed, which translates into faster execution times.

### 5.2.5 GPU-aware Approximate Wavefront Alignment

For some applications, the exact computation of the alignment may be unnecessary, and a reasonable approximation of the optimal solution may suffice (e.g., pre-filtering [6, 7] and clustering [103, 104] applications). For that, heuristic strategies can reduce the number of computations required, improving performance at the expense of a potential loss of score optimality. In addition to the exact WFA-GPU algorithm, we propose a heuristic extension that improves the performance of the exact algorithm, with high observed optimal-score agreement. Our heuristic strategy draws inspiration from the original WFA-Adapt [71] heuristic and other adaptive techniques [91]. Unlike previous approaches, our strategy is tailored to the architecture and resources available in the GPU and exploits the GPU parallelism to minimize the overhead of executing the heuristic.

Our heuristic strategy employs a fixed-size wavefront of length $\beta$. Initially, wavefronts are centered around the main diagonal. Every $\lambda$ steps, the algorithm computes the most promising diagonal. That is, the diagonal closest to the target cell (the bottom-right corner of the DP table). Afterwards, newly computed wavefronts are centered around the most promising diagonal, focusing the wavefront computation towards the most promising partial alignment. As a result, the heuristic avoids the computation of wavefront diagonals that are left behind (unlikely to lead to the optimal solution).

Selecting adequate values of $\beta$ and $\lambda$ can have an impact on performance and optimal-score agreement. Note that a small $\beta$ can lead to a significant reduction in optimal-score agreement. Similarly, a small $\lambda$ could lead to overheads in computing the heuristic. In contrast, large values of $\beta$ and $\lambda$ could render the heuristic ineffective. Our implementation selects a value of $\beta$ such that all the wavefronts can fit into the fast on-chip shared memory of the GPU. Regarding $\lambda$, we observe that values $\lambda \leq 100$ perform similarly in terms of optimal-score agreement. Moreover, to alleviate the computational burden when $\lambda$ is small, our implementation uses a thread-cooperative strategy. For that, threads within a warp cooperate to find the most promising diagonal and center the wavefront around it. Section 5.3.5 presents a more in-depth parameter exploration of the impact of $(\beta, \lambda)$ on time and optimal-score agreement.

### 5.2.6 CPU-GPU Co-design System

The WFA-GPU implements a CPU-GPU co-design that allows the simultaneous execution of GPU computations overlapped with data transfers and CPU alignment rescue. To maximize performance, our implementation offloads batches containing multiple alignments to the GPU. For that, input sequences from a batch have to be transferred to the device. To minimize GPU idle times, our implementation makes asynchronous kernel launches, allowing overlapping data transfers with GPU computations. That is, while the GPU is computing the alignments for a given batch, the sequences of the following batch are being copied to the device. As a result, latencies due to transfer times are effectively hidden and overlap with useful GPU computations.

Furthermore, the asynchronous implementation of WFA-GPU allows employing idle CPU time to rescue alignments returned by the GPU. As explained before, a small percentage of alignments may not be aligned in the GPU due to exceeding memory requirements. For those

few cases, the implementation overlaps the CPU WFA execution with GPU computations and data transfers.

## 5.3 Experimental Results

Throughout this chapter, optimal-score agreement (also called exact-score rate, denoted $R$ in the tables) is the fraction of benchmark pairs whose reported score equals the exact optimum for the same scoring function and alignment mode. It is termed "recall" in the figures. This metric should not be confused with read-mapping recall, variant-calling sensitivity, or biological truth. For Edlib and CudaAligner, the returned edit-distance CIGAR is re-scored with gap-affine penalties; disagreement therefore concerns the selected gap-affine objective. ADEPT performs local alignment and is marked not comparable (n/c) against the global objective.

### 5.3.1 Evaluation on Simulated Data

Table 5.1 shows time (in seconds) and optimal-score agreement (percentage of sequences for which the optimal alignment was correctly reported) for the alignment executions using simulated datasets.

Considering the alignment of short sequences (i.e., $\sim$150bps), NVBio outperforms all other tools at the expense of a significant reduction in optimal-score agreement as the alignment error increases. WFA-GPU is between 1.2 and 2.9 times faster than the best CPU time obtained. GASAL2 shows good performance (as it is tailored to short sequence alignment). Because DP approaches are error-independent, GASAL2 outperforms our implementation when the error is high. The other GPU aligners, ADEPT and CudaAligner, are one order of magnitude slower than WFA-GPU and GASAL2. For these short-sequence datasets, using the WFA-GPU approximate alignment does not provide any significant benefit because the error is very small.

For medium-length sequences ($\sim$1Kbps), GPU implementations either fail due to execution errors (like NVBio and ADEPT) or obtain an optimal-score agreement below 10%. Only GASAL2 remains competitive when the error increases, but at a significantly low optimal-score agreement (52.4%). Compared to CPU implementations, WFA-GPU executes 2.5-3.3$\times$ faster than the original WFA and up to 5500$\times$ faster than other libraries. Using WFA-GPU approximate alignment, we obtain an additional speedup of up to 3$\times$ when computing the alignment path (backtrace).

Experiments aligning long simulated sequences (i.e., $\sim$10Kbps) turn out to be the most challenging for most GPU tools. All other GPU implementations either fail (i.e., ADEPT and NVBio), disagree with the gap-affine optimum (i.e., CudaAligner), or have significantly low optimal-score agreement (i.e., GASAL2 with less than 50% optimal-score agreement). WFA-GPU is the only GPU implementation that can scale to long sequences, reporting the optimal alignment result. When compared to the original WFA, WFA-GPU executes 3.1-3.8$\times$ faster. Using the approximate WFA-GPU alignment, the speed increase is even greater, at 14.2-61.9$\times$, while achieving 100% optimal-score agreement on these datasets.

When executing WFA-GPU without calculating the alignment path (only calculating the alignment distance), the process is 1.8-4.7$\times$ faster than the baseline WFA-GPU. This results

in a speed increase of up to 16.8× compared to the CPU WFA implementation. In addition, we implement an approximated distance-only kernel, which gives a speedup of up to 133.0× compared to the multithreaded CPU WFA implementation.

Table 5.1: Time (T, in seconds) and optimal-score agreement (R, the percentage of scores matching the exact optimum) for simulated datasets with different error rates (e). All CPU executions use 10 threads. †Implementations can only produce edit-distance alignments.

| | | **100 Million alignments × 150bps** | | | | | | **10 Million alignments × 1Kbps** | | | | | | **100K alignments × 10Kbps** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **e=2%** | | **e=5%** | | **e=10%** | | **e=2%** | | **e=5%** | | **e=10%** | | **e=2%** | | **e=5%** | | **e=10%** | |
| | | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) |
| **GPU** | GASAL2 | 13 | 98.3 | 14 | 95.8 | 14 | 93.0 | 44 | 84.6 | 44 | 64.7 | 45 | 52.4 | 3219 | 47.0 | 3224 | 36.8 | 3229 | 34.1 |
| | ADEPT | 161 | n/c | 161 | n/c | 161 | n/c | error | n/a | error | n/a | error | n/a | error | n/a | error | n/a | error | n/a |
| | NVBio | 6 | 58.2 | 6 | 6.1 | 6 | 0.0 | error | n/a | error | n/a | error | n/a | error | n/a | error | n/a | error | n/a |
| | CudaAligner† | 113 | 95.3 | 113 | 66.1 | 117 | 25.7 | 86 | 9.1 | 86 | 1.6 | 86 | 0.3 | 20 | 0.9 | 20 | 0.0 | 20 | 0.0 |
| | WFA-GPU | 11 | 100 | 18 | 100 | 27 | 100 | 6 | 100 | 20 | 100 | 78 | 100 | 4 | 100 | 21 | 100 | 71 | 100 |
| | ├approx | 11 | 100 | 18 | 100 | 27 | 100 | 6 | 100 | 11 | 100 | 21 | 100 | 1 | 100 | 2 | 100 | 4 | 100 |
| | └distance | 6 | 100 | 8 | 100 | 13 | 100 | 3 | 100 | 9 | 100 | 32 | 100 | 2 | 100 | 6 | 100 | 16 | 100 |
| | └approx | 6 | 100 | 7 | 100 | 10 | 100 | 3 | 100 | 4 | 100 | 7 | 100 | 1 | 100 | 1 | 100 | 2 | 100 |
| **CPU** | Seqan | 2028 | 100 | 2028 | 100 | 2190 | 100 | 9066 | 100 | 9288 | 100 | 9774 | 100 | 9912 | 100 | 10098 | 100 | 10440 | 100 |
| | Parasail(strip) | 8436 | 100 | 8358 | 100 | 8238 | 100 | 32274 | 100 | 32178 | 100 | 31842 | 100 | 33408 | 100 | 33306 | 100 | 32760 | 100 |
| | Parasail(scan) | 1197 | 100 | 1183 | 100 | 1197 | 100 | 3882 | 100 | 3902 | 100 | 3911 | 100 | 4869 | 100 | 4857 | 100 | 4825 | 100 |
| | Parasail(diag) | 1908 | 100 | 1914 | 100 | 1916 | 100 | 7482 | 100 | 7488 | 100 | 7488 | 100 | 7686 | 100 | 7692 | 100 | 7698 | 100 |
| | Edlib† | 210 | 95.9 | 202 | 69.6 | 200 | 30.0 | 118 | 67.6 | 118 | 9.5 | 135 | 0.0 | 36 | 1.8 | 38 | 0.0 | 43 | 0.0 |
| | KSW2 | 298 | 100 | 306 | 100 | 317 | 100 | 876 | 100 | 882 | 100 | 882 | 100 | 2028 | 100 | 2034 | 100 | 2040 | 100 |
| | WFA | 12 | 100 | 37 | 100 | 78 | 100 | 15 | 100 | 61 | 100 | 209 | 100 | 14 | 100 | 83 | 100 | 272 | 100 |

### 5.3.2 Evaluation on Real Data

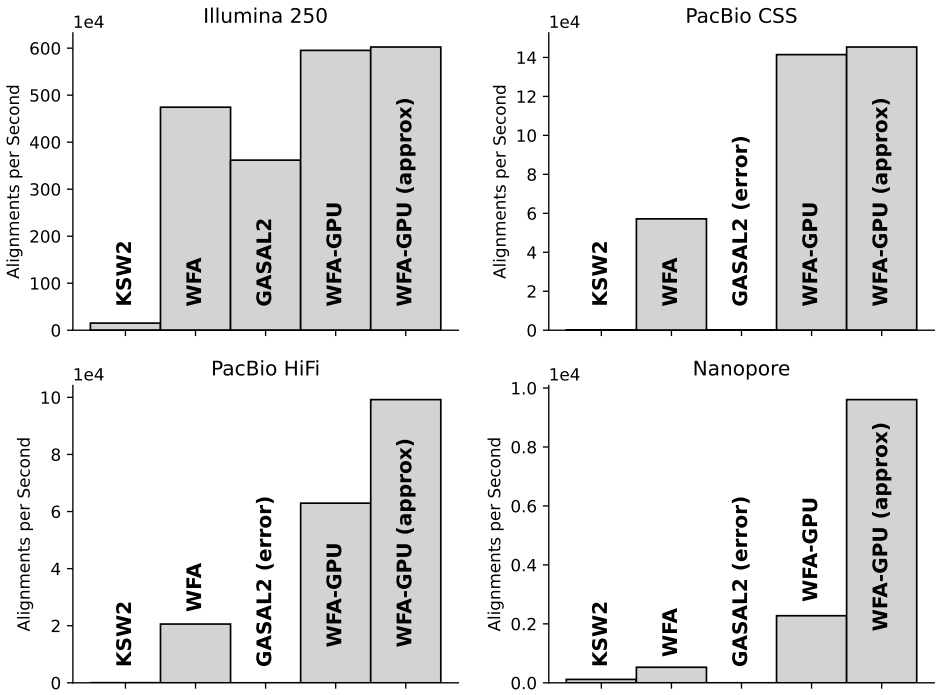



Figure 5.3: Alignments per second obtained with the most performant CPU and GPU gap-affine implementations compared with WFA-GPU.

Table 5.2 compares the performance of WFA-GPU to other state-of-the-art libraries and tools when aligning real datasets. Figure 5.3 illustrates the results of the most relevant datasets and implementations.

For the case of aligning high-quality short sequences, like those produced by Illumina sequencers, NVBio delivers the fastest results at the expense of scoring low in optimal-score agreement (only 49.2% and 27.2% of the reported scores match the gap-affine optimum). GASAL2 delivers similar performance to WFA-GPU when aligning the Illumina 150 dataset and is 1.7× slower when aligning Illumina 250 (which has slightly longer sequences). Compared to the original WFA, which achieves the best execution time among all CPU libraries, WFA-GPU is 1.2× slower when aligning the Illumina 150 dataset, and 1.3× faster when aligning the Illumina 250 dataset. Compared to other CPU libraries, WFA-GPU obtains remarkable speedups (up to 1264× compared with Parasail). On these datasets (that have a small nominal error), using WFA-GPU approximate alignment has little effect, providing an additional speed increase of only 1.3×. Computing only the alignment distance (distance-only kernel) is 1.4-2× faster than computing the whole alignment, giving an overall speedup of 1.7× compared with the multithreaded CPU WFA implementation.

Using PacBio sequences, WFA-GPU achieves a speedup of 2.5× (on PacBio CCS) and 3× (on PacBio HiFi) compared to the multithreaded CPU version of the WFA. The speedup rises up to 4.8× if we don't compute the alignment path (distance-only version). Our implementation

Table 5.2: Time (T, in seconds) and optimal-score agreement (R, the percentage of scores matching the exact optimum) for real datasets. All CPU executions use 10 threads. †Implementations that can only produce edit-distance alignments. Executions taking more than 48 hours are marked as timeout.

| | | **Illumina 150 100M alignments** | | **Illumina 250 100M alignments** | | **PacBio CCS 10M alignments** | | **PacBio HiFi 10M alignments** | | **Nanopore 10M alignments** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) |
| **GPU** | GASAL2 | 12 | 99.9 | 28 | 94.7 | error | n/a | error | n/a | error | n/a |
| | ADEPT | 158 | n/c | 308 | n/c | error | n/a | error | n/a | error | n/a |
| | NVBio | 4 | 49.2 | 6 | 27.2 | error | n/a | error | n/a | error | n/a |
| | CudaAligner† | 102 | 97.5 | 169 | 91.6 | 2036 | 42.6 | 3105 | 32.5 | 1350 | 0.0 |
| | WFA-GPU | 13 | 100 | 17 | 100 | 71 | 100 | 159 | 100 | 4395 | 100 |
| | ├approx | 10 | 100 | 17 | 100 | 66 | 98.6 | 101 | 98.4 | 1041 | 98.7 |
| | └distance | 7 | 100 | 13 | 100 | 48 | 100 | 100 | 100 | 1083 | 100 |
| | └approx | 5 | 100 | 13 | 100 | 55 | 98.6 | 69 | 98.4 | 569 | 98.6 |
| **CPU** | Seqan | 1810 | 100 | 5076 | 100 | timeout | n/a | timeout | n/a | timeout | n/a |
| | Parasail(strip) | 8424 | 99.9 | 21240 | 99.9 | timeout | n/a | timeout | n/a | timeout | n/a |
| | Parasail(scan) | 1160 | 99.9 | 2814 | 99.9 | timeout | n/a | timeout | n/a | timeout | n/a |
| | Parasail(diag) | 1878 | 99.9 | 4912 | 99.9 | timeout | n/a | timeout | n/a | timeout | n/a |
| | Edlib† | 155 | 97.5 | 246 | 91.8 | 2127 | 47.7 | 7998 | 35.7 | 22620 | 0.0 |
| | KSW2 | 251 | 100 | 661 | 100 | 163860 | 100 | timeout | n/a | 86580 | 100 |
| | WFA | 11 | 100 | 21 | 100 | 175 | 100 | 486 | 100 | 18960 | 100 |

outperforms by up to four orders of magnitude other CPU tools and libraries. The only GPU implementation able to finish is CudaAligner, even though it obtains a significantly low optimal-score agreement (less than 50%) while being between 19.5-26.7× slower than our solution. When using WFA-GPU approximate alignment, we obtain an overall speedup of 2.7× and 4.8× for the CCS and HiFi datasets, respectively. Combining the approximated approach and the distance-only kernel, the speedup goes up to 7.1×.

Regarding the Nanopore dataset, which consists of large sequences with a high error rate, WFA-GPU is 4.3× faster than the CPU implementation of WFA and 17.5× faster when using the distance-only version. In comparison, CudaAligner is 3× faster for this execution, but its re-scored edit-distance paths have 0% agreement with the gap-affine optimum. When compared to other CPU libraries, WFA-GPU is up to 19.7× faster. However, this speedup is not as high as on previous datasets because alignments with a high nominal error represent the worst-case scenario for the WFA algorithm. Despite this, a high error rate presents a good opportunity for approximate methods to improve execution times. In particular, the WFA-GPU approximate approach obtains a speed increase of 18.2× compared with the multithreaded WFA CPU version. Only computing the alignment distance while using approximate WFA-GPU results in a speedup of 33.3×.

In general, other GPU implementations either fail to scale with increasing sequence lengths, drop enormously in the optimal-score agreement, or simply fail to execute. Although CudaAligner delivers good performance results, it optimizes edit distance rather than the gap-affine objective used here. While traditional DP-based algorithms fail to scale with sequence length, WFA-based implementations demonstrate scaling with increasing error rates and lengths. In practice, DP-based implementations require impractical execution times. In contrast, our WFA-GPU implementation delivers good performance results, even when aligning long and noisy

sequences. Additionally, our approximate alignment method for WFA can provide additional speedups with less than 2% suboptimal scores on the reported real-data comparison.

### 5.3.3 Power consumption analysis

In this section, we analyze the power-consumption improvements achieved by using WFA-GPU over WFA (CPU). It's worth noting that making an entirely equitable comparison is a challenging endeavor, as our analysis is tailored to our specific CPU-GPU configuration. We have tried to make the comparison as fair as possible, using a balanced CPU-GPU combination. However, we acknowledge that comparing diverse hardware devices introduces complexities in the evaluation process.

The thermal design power (TDP) of the NVIDIA GeForce 3080 is 320W, while the TDP of the Intel Xeon-W2155 is 140W. However, many hardware components, like the CPU, memory, and the motherboard, are not accounted for in the CPU TDP. Moreover, hardware devices do not always work at their maximum TDP. As a reference, Table 5.3 presents the average power usage by the CPU and GPU when executing WFA (CPU) and WFA-GPU, respectively.

Table 5.3: Average power usage (in Watts) to compute alignments using the CPU and GPU aligners. CPU consumption is obtained using Linux perf, and GPU consumption is obtained using nvidia-smi.

| Sequence length | 150 | 1000 | 1000 | 10000 | 10000 |
|---|---|---|---|---|---|
| Error rate (%) | 5 | 5 | 10 | 5 | 10 |
| WFA (CPU) | 134 | 149 | 140 | 129 | 129 |
| WFA-GPU | 130 | 185 | 290 | 298 | 312 |

Table 5.3 shows that the GPU has a higher power draw during the alignment process. However, it is important to emphasize that WFA-GPU significantly reduces execution time, which makes WFA-GPU experiments more power-efficient in terms of energy consumption (measured in Watt-hours, or *Wh*) compared with CPU executions. Table 5.4 provides a comparison of power consumption (*Wh*) between CPU and GPU executions. Remarkably, our results demonstrate that WFA-GPU experiments consume less overall energy, even though GPU devices require more power than typical CPUs.

Table 5.4: Power consumption (in Watt-hours) to compute alignments using the CPU and GPU aligners.

| Sequence length | 150 | 1000 | 1000 | 10000 | 10000 |
|---|---|---|---|---|---|
| Error rate (%) | 5 | 5 | 10 | 5 | 10 |
| WFA (CPU) | 1.4 | 2.5 | 8.1 | 3.0 | 9.8 |
| WFA-GPU | 0.6 | 1.0 | 5.1 | 1.9 | 6.1 |

### 5.3.4 Performance scalability

Our method is not constrained by the length of the input sequences, but rather by the sequence error rate (i.e., maximum alignment score). According to the maximum alignment score, WFA-GPU allocates the necessary memory, maximizing the total number of GPU workers spawned on the device. Ultimately, this allocation depends on the available computational resources of the device (i.e., Memory and Streaming Multiprocessors, SMs). Note that the memory taken by the input sequences is negligible compared to the memory required by the internal WFA structures. In Figure 5.4, we show the maximum number of GPU workers that can be accommodated on various GPU models, based on the maximum alignment score allowed.

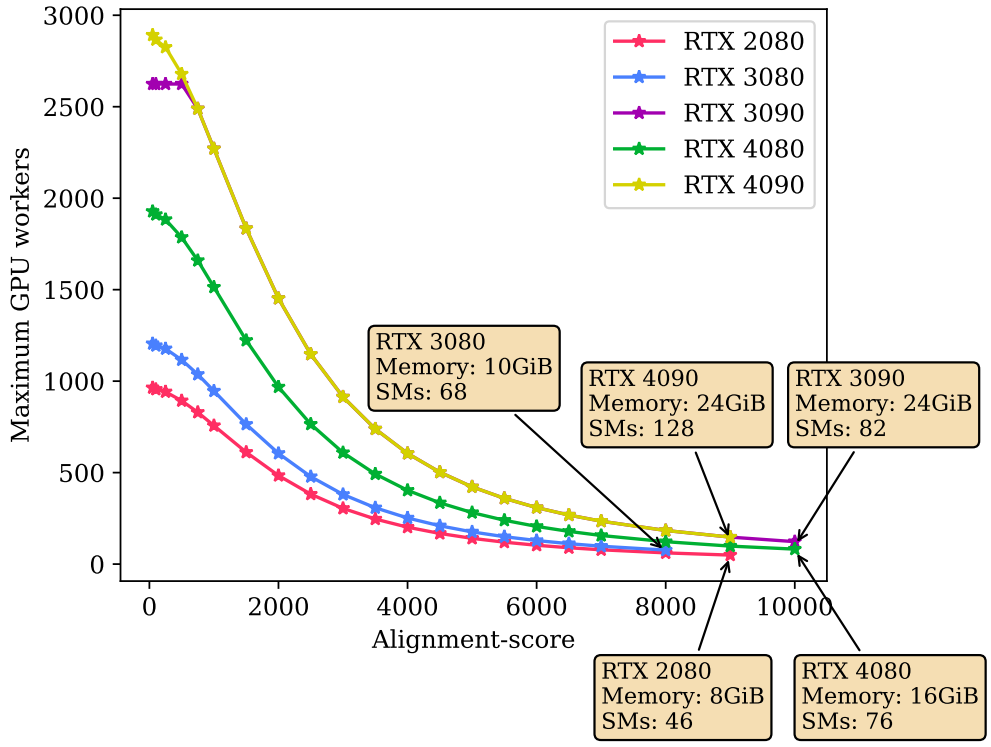



Figure 5.4: Maximum number of GPU workers that can be executed in parallel depending on the maximum alignment score for different GPU device models (i.e., different numbers of SMs and Memory).

Figure 5.4 shows how the maximum number of parallel GPU workers decreases as the maximum alignment-score increases. For large alignment-score values, only one GPU worker can be allocated per SM. Labeled points at the right of Figure 5.4 show the maximum alignment score for different GPU models. For instance, an RTX 4080 GPU, equipped with 16GiB, could align a pair of sequences with a maximum alignment score of 10,000.

A GPU with more SMs does not necessarily support a higher maximum score per alignment when the scheduler allocates workers across all SMs. For example, the RTX 4090 and RTX 3090 each have 24 GiB of device memory, but distributing that memory among more concurrent

workers reduces the budget available per worker. This is a scheduling-dependent limit, rather than an intrinsic limit on the length of a single input sequence.

### 5.3.5 Approximated WFA-GPU parameter exploration

The heuristic parameters affect execution time and optimal-score agreement (percentage of scores matching the exact optimum). By changing width of the band ($\beta$) and the number of steps between the band re-centering ($\lambda$) we can find the balance point between time and optimal-score agreement.

It is essential to emphasize that utilizing a narrower bandwidth ($\beta$) can significantly influence memory usage. Since employing more threads than the band size is impractical, choosing a smaller $\beta$ results in GPU workers with fewer threads. Consequently, this allows for accommodating a higher number of workers within each Streaming Multiprocessor (SM). This increased parallelism, however, comes at the cost of utilizing more GPU memory, as each worker necessitates its own dedicated memory space. It may be the case that using small bands for alignments with high errors generates a number of GPU workers that exceeds the available memory capacity.

We present the Nanopore dataset as a case study. This dataset is the one that benefits the most from the adaptive strategy due to its long sequences and high error rates. Figure 5.5 shows how time and optimal-score agreement are affected by each $(\beta, \lambda)$ combination. It can be seen that there is nearly no performance benefit in incrementing $\lambda$ beyond 100. When using a large width for the band ($\beta$), a $\lambda = 100$ is sufficient and, when using a smaller $\beta$, setting $\lambda$ around 50 is recommended. In any case, the reduction in optimal-score agreement is usually less than three percentage points for the configurations discussed. Therefore, unless an optimality guarantee is required, default $(\beta, \lambda)$ parameters will yield a good balance between execution time and minimal loss in optimal-score agreement.

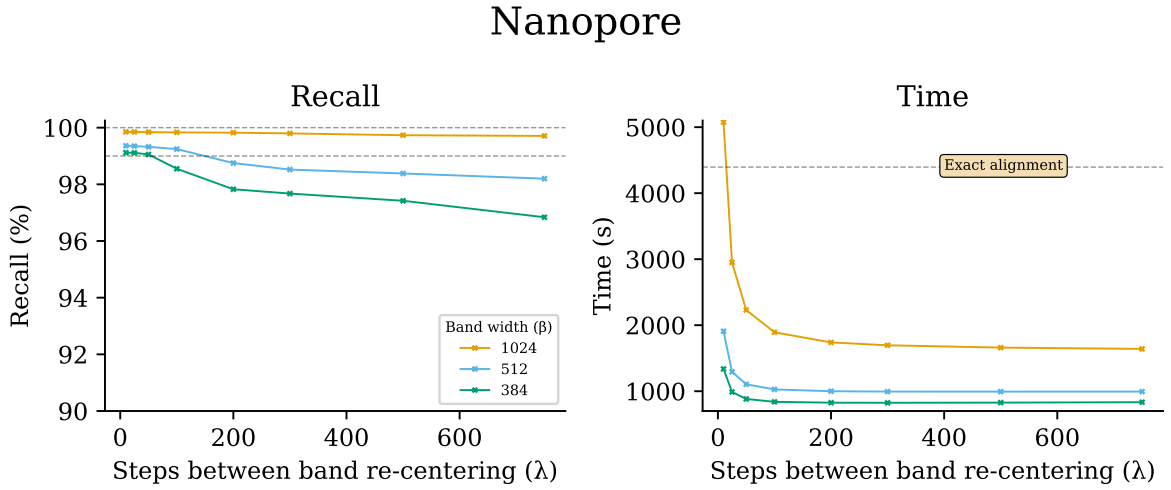



Figure 5.5: Optimal-score agreement (labeled "Recall" in the original artwork) and time (in seconds) when aligning the Nanopore dataset using different $\beta$ and $\lambda$ parameters.

## 5.4 Discussion

Future advances in sequencing technologies and genomics present critical challenges to current bioinformatics methods and tools. This situation calls for improved methods and tools that can scale with increasing production yields and sequence lengths. Current HPC computing relies on GPUs as successful hardware accelerators for computing-intensive applications in many areas of research.

This chapter addresses the limitations identified in Chapter 4 by extending the GPU acceleration of WFA to the gap-affine scoring model and removing constraints on the maximum number of errors. The resulting WFA-GPU implementation enables exact gap-affine alignment on GPUs, combining algorithmic efficiency with hardware-aware design. This represents a significant step towards the practical, scalable alignment of long and noisy sequences, which are increasingly common in modern genomics workflows.

The main contribution of this work lies in adapting the WFA formulation to support gap-affine scoring while preserving efficient execution on GPU architectures. This requires managing multiple dependent wavefronts and redesigning data structures to operate beyond shared-memory limits. By extending the piggyback backtrace strategy and carefully organizing memory accesses, the implementation scales with the available GPU DRAM while maintaining high throughput. In addition, the design incorporates both exact and approximate alignment modes, allowing a flexible trade-off between performance and optimal-score agreement for challenging alignment scenarios.

Experimental results show that WFA-GPU outperforms the evaluated CPU and GPU alignment tools on many workloads, with the largest gains on long-read datasets. The implementation achieves speedups of up to 29× over existing GPU aligners and up to four orders of magnitude over classical DP-based CPU libraries. Compared to the multithreaded CPU implementation of WFA, we obtain speedups of up to 4.3× on real datasets in exact alignment-path mode. When computing only the alignment distance, performance improves further, reaching up to 4× additional speedup. For long and noisy sequences, the GPU-aware approximate alignment strategy achieves speedups of up to 18.2× over WFA CPU while retaining 98.6% optimal-score agreement on the Nanopore dataset. These figures refer to alignment-path computation; distance-only results are reported separately. To the best of our knowledge, this is the first GPU-based pairwise aligner capable of computing exact gap-affine alignments for long-read datasets, such as PacBio HiFi and Oxford Nanopore, within practical execution times on commodity GPUs.

The results also highlight the scalability and cost-effectiveness of GPU-based alignment. Compared to CPU-only configurations, GPU-accelerated systems provide a higher performance-to-cost ratio, even under optimistic assumptions of CPU scalability. This makes GPUs an attractive platform for large-scale sequence alignment workloads, where both throughput and cost are critical factors. In addition to performance improvements, WFA-GPU reduces energy consumption by a factor of 1.6–2.5× compared to CPU-based alignment. This reduction is particularly relevant for large-scale deployments, where energy efficiency directly impacts operational costs. Together with its scalability and support for exact alignment, this motivates future integration of WFA-GPU into bioinformatics pipelines.

Overall, WFA-GPU provides an accurate, scalable, and high-performance solution for gap-affine sequence alignment on GPUs. By combining algorithmic advances with hardware-aware optimizations, it bridges the gap between biological relevance and efficient computation. The implementation is released as open-source software and is publicly available at WFA-GPU repository, facilitating reproducibility and adoption in real-world applications.

# Chapter 6

# Ultra-Long Sequence Alignment on GPUs

## 6.1 Introduction and Motivation

Chapters 4 and 5 showed that WFA-based alignment can be effectively accelerated on GPU architectures for edit-distance and gap-affine scoring. These contributions demonstrated that combining algorithmic improvements with hardware-aware design yields substantial performance gains over classical DP approaches. However, they also exposed an important remaining limitation: when sequences become significantly long, the memory requirements of the underlying wavefront formulation become the dominant bottleneck.

This limitation is increasingly relevant in modern genomics. Third-generation sequencing technologies, particularly Oxford Nanopore and PacBio, now produce reads ranging from tens of kilobases to megabases in length. Such ultra-long reads are essential for applications including structural variant detection, telomere-to-telomere assembly, repeat resolution, and pangenome analysis, where long-range sequence context is critical. At the same time, these applications require exact or highly accurate alignments over very large sequence spans, making them particularly demanding from both computational and memory perspectives.

Although the WFA algorithm substantially reduces the computational complexity of exact alignment, its original formulation still requires $O(s^2)$ memory, where $s$ is the optimal alignment score. For long and noisy reads, this quadratic dependence remains prohibitive. This problem is especially severe on GPUs. High performance on these devices depends on exploiting fast on-chip memories such as shared memory and registers, yet these resources are severely capacity-limited. As a result, a straightforward GPU implementation of WFA cannot scale to megabase-long alignments without either exhausting memory or falling back to inefficient execution strategies.

The Bidirectional Wavefront Alignment algorithm (biWFA) provides the algorithmic basis to overcome this limitation. By combining forward and reverse wavefront computations with recursive breakpoint decomposition, biWFA reduces the memory complexity from $O(s^2)$ to $O(s)$ while preserving the output-sensitive $O(ns)$ time complexity. This makes biWFA a natural candidate for ultra-long exact alignment. However, efficiently mapping biWFA to GPU architectures presents some challenges. In addition to the irregularity already present in WFA, biWFA introduces recursive sub-alignments, breakpoint detection, and fine-grained synchronization challenges, complicating efficient parallel execution.

The main challenge addressed in this chapter is therefore not only to reduce memory consumption but also to do so in a way that preserves GPU efficiency. This requires exposing enough parallelism to keep the device busy, organizing the computation so that large wavefronts can be processed with limited on-chip memory, and restructuring the execution to support the recursive nature of biWFA. In particular, ultra-long sequence alignment demands a co-design of algorithm and architecture in which memory hierarchy, tiling strategy, and parallel decomposition are treated as primary design constraints.

In this chapter, we present BiGWAVE, a GPU-accelerated design and implementation of the biWFA algorithm for exact alignment of ultra-long sequences. BiGWAVE exploits three complementary levels of parallelism: inter-sequence parallelism across independent alignments, intra-sequence parallelism within each wavefront, and recursion-level parallelism across independent sub-alignments generated by breakpoint decomposition. To make this practical on GPUs, we introduce a hyperplane tiling strategy that regularizes wavefront computation and enables large alignments to progress with bounded on-chip memory. We further optimize the *extend* operation, which remains a key performance bottleneck, through GPU-friendly bit-parallel sequence comparison.

Overall, this chapter extends the scope of wavefront-based GPU alignment from short and long reads to ultra-long reads. It shows that exact megabase-scale alignment can be performed efficiently on commodity GPUs when output-sensitive algorithms are combined with a memory-aware and parallel GPU design. In this way, BiGWAVE closes an important gap in the thesis: enabling scalable, exact, and practical sequence alignment across the full spectrum of read lengths encountered in modern sequencing technologies and genomics analyses.

## 6.2 Method

Several methods have emerged to accelerate the WFA algorithm on GPUs [2, 1, 37], Processing-in-Memory (PIM) [27, 5], FPGAs [46, 16], and ASICS [44], exploiting the two levels of parallelism of the WFA: inter-sequence and intra-sequence. The biWFA algorithm adds a third level: the recursion level parallelism.

Fig. 6.1 shows the different levels of parallelism available in the biWFA algorithm. Intra-sequence parallelism exploits the independent work available inside a single alignment. Due to the computational patterns of Eq. 2.4 and 2.5, each element of a wavefront, both in the forward ($\overrightarrow{\mathscr{A}}$) and reverse ($\overleftarrow{\mathscr{A}}$) alignments, can be computed independently from each other (❶). We exploit this parallelism by assigning a thread to each element of a wavefront. If the wavefront exceeds the number of threads available, threads iterate over it until all the elements are computed.

Inter-sequence parallelism can be used to align different sequences in parallel (❷). Even though this is the most evident and straightforward parallelism available, it is crucial to exploit the vast amount of resources of modern parallel devices (like GPUs), as a single alignment does not generate enough work to keep the device busy. We exploit this parallelism by spawning a fixed number of thread-blocks that dynamically get alignments from a pool of pending sequences. This way, we can keep the GPU busy by having multiple alignments being computed simultaneously.

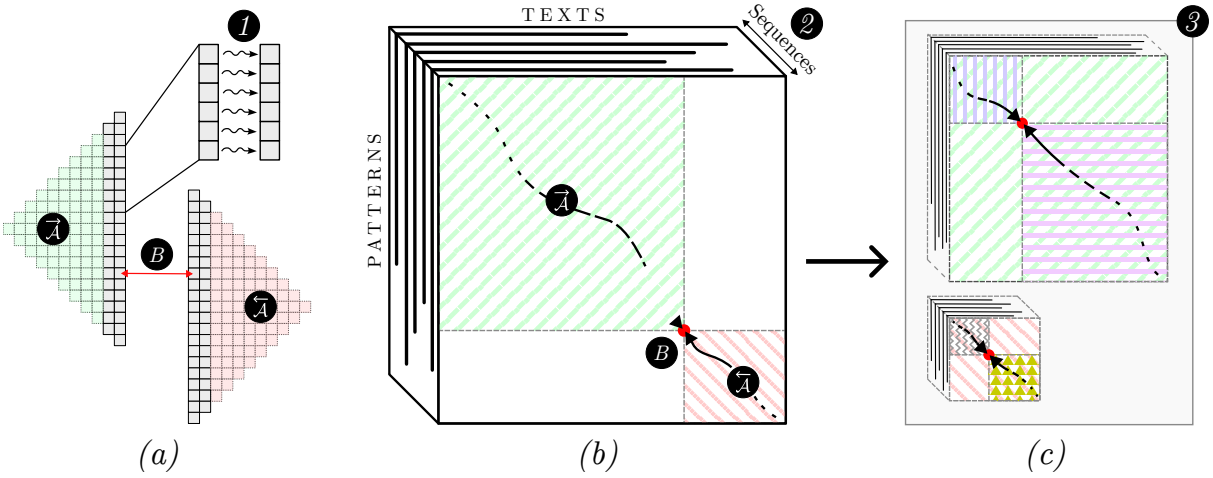


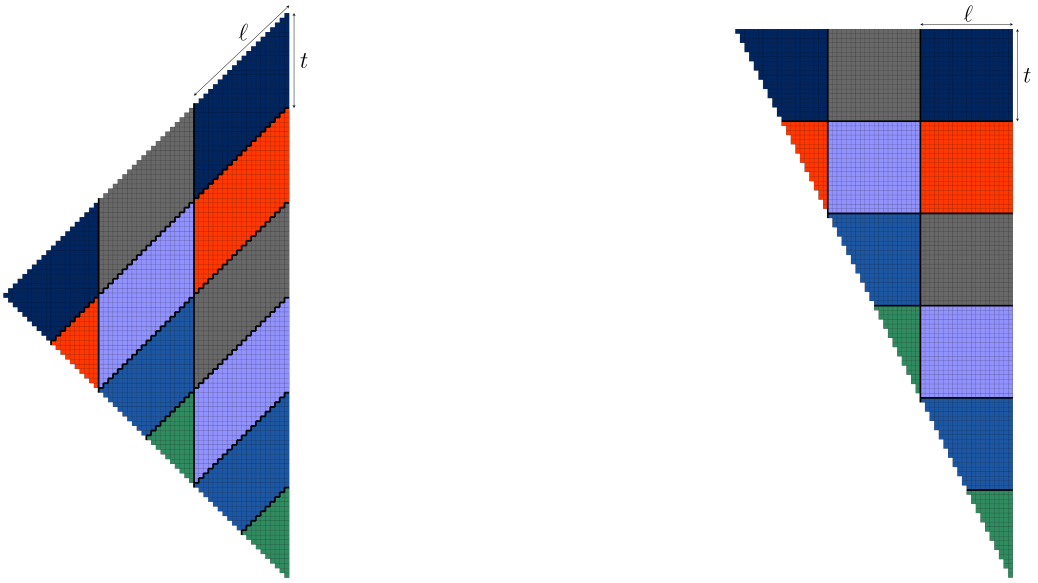

Figure 6.1: **(a)** Wavefront representation of the initial alignment, with a wavefront representing the forward alignment ($\overrightarrow{\mathscr{A}}$) and another representing the reverse alignment ($\overleftarrow{\mathscr{A}}$). Multiple threads can compute different elements of a wavefront in parallel. **(b)** Representation of the biWFA on the DP table, note that the breakpoint divides the alignment into parts of equal score. Therefore, the size of the two sub-alignments can be different (depending on the error distribution). Multiple alignments are performed simultaneously to exploit inter-sequence parallelism. **(c)** The two sub-alignments generated by the first breakpoint, this is the first recursion level of the algorithm.

Finally, biWFA introduces a third level of parallelism exploiting the independent sub-alignments generated by the breakpoints: recursion-level parallelism. As explained before, each time a breakpoint is found (B), two sub-alignments are generated, which can be computed in parallel (3). Therefore, in the recursion level $r$ there are $2^r$ independent sub-alignments. We exploit this parallelism by launching a different GPU kernel for each recursion level, treating each sub-alignment as an independent alignment. We create sub-alignments until a certain error threshold is reached. We choose this threshold to be 64 and use the highly-optimized WFA kernel from [2] to compute the last sub-alignments in parallel. The sub-alignments' optimal paths (CIGAR) of each sequence pair are then concatenated to obtain the final alignment path.

As wavefronts grow linearly, at a given alignment step $e$, the wavefront $\widetilde{W}_e$ contains $2e+1$ elements. This growth poses a challenge when utilizing fast on-chip memories, which have limited capacity. Furthermore, assigning all available *shared memory* to a single CUDA block means that only one thread-block can use the shared memory of a Streaming Multiprocessor (SM) at a time, preventing other blocks from being scheduled on the same SM and negatively impacting performance.

To be able to align sequences that have a very high alignment score while doing most of the computations on the fast on-chip memories of the GPUs, a strategy that divides the solution space into multiple tiles is needed. We propose an efficient GPU-aware tiling strategy that allows the advance of only a certain region of a Wavefront using fast on-chip memories while maintaining a regular number of parallel elements each step.

### 6.2.1 Hyperplane Tiling Strategy




(a) biWFA tiling strategy which depicts a regular number of parallel elements on each tile.

(b) Space transformation that makes the tiles regularly aligned with the data structures.

Figure 6.2: Hyperplane tiling strategy for the WFA and biWFA algorithms.

The SIMT programming model inherent to GPUs frequently leads to partitioning problems into tiles that exhibit uniform parallelism (i.e., maintaining the same amount of parallel work at each step). In particular, this tiling strategy has proven successful in addressing problems involving stencil patterns with wavefront parallelism in other algorithms [50, 26, 68, 14, 12, 62].

We introduce an intra-sequence parallel strategy for WFA and biWFA, in which tiles maintain a consistent number of parallel elements at each step. This approach enhances GPU utilization, enabling the advancement of an unlimited number of steps without incurring redundant computations.

Let $t$ be the height of the tile (i.e., the number of wavefront elements that the tile includes), and $\ell$ the number of steps that a tile advances. Given any wavefront $\widetilde{W}_e$ of length $2e+1$, we divide $\widetilde{W}_e$ into $\frac{2e+1}{t}$ number of tiles. Starting from the top of the wavefront, the first $t$ offsets belong to the first tile, the second $t$ offsets to the second tile, and so onThis way, a tile has a constant number of parallel elements across all its steps. The solution space is partitioned as shown in Fig. 6.2a.

To compute a tile at position $(i, j)$, the borders with tiles $(i-1, j)$ and $(i, j-1)$ are needed. Vertical borders are naturally aligned with the WFA data structure, but diagonal borders are not. This introduces control logic into the implementation and worsens memory locality. We propose a space transformation of the original algorithm to solve these issues.

Let $\widetilde{H}_e$ be a wavefront at a distance $e$, starting at $z = 0$ (first element) and with a length of $2e+1$. For every offset in $\widetilde{H}_e$ at position $z$ ($\widetilde{H}_{e,z}$), there exists an equivalent $\widetilde{W}_{e,k}$ in the traditional WFA encoding. The transition between spaces (from $k$ to $z$) is managed through

Eq. 6.2. With this transformation, each position in $\widetilde{H}_e$ is shifted up by one position compared with its equivalent $\widetilde{W}_e$. Therefore, the recurrence of the algorithm needs to be adapted. Eq. 6.1 shows the new recurrences taking into account these changes. The resulting solution space from this strategy is graphically shown in Fig. 6.2b. It can be seen how the tile borders are aligned with the natural borders of the data structure.

$$\widetilde{H}_{e+1,z} = \max \begin{cases} \widetilde{H}_{e,z-2} & \text{(Deletion)} \\ \widetilde{H}_{e,z-1}+1 & \text{(Mismatch)} \\ \widetilde{H}_{e,z}+1 & \text{(Insertion)} \end{cases} \tag{6.1a}$$

$$\widetilde{H}_{e,z} = \widetilde{H}_{e,z} + LCP(p_{(\widetilde{H}_{e,(e-z)})-k\ldots n}, t_{\widetilde{H}_{e,(e-z)}\ldots m}) \tag{6.1b}$$

$$z = e-k \tag{6.2}$$

We implement this strategy by computing tiles with the height ($t$) being the number of threads in a thread-block. This way, each thread is responsible for a diagonal within the tile. We reserve a double buffer in shared memory, where vertical wavefront values for a specific tile are stored and updated. The vertical tile borders are stored in global memory. Thread-blocks compute the tiles column-wise. This way, only the horizontal border of the previously calculated tile is needed to proceed to the current tile, and can reside in a single reusable buffer on *shared memory*. This is shown in Fig. 5, where horizontal borders are highlighted in red.

This strategy presents a problem when computing the first breakpoint. The algorithm needs to check for overlap between $\overrightarrow{\mathscr{A}}$ and $\overleftarrow{\mathscr{A}}$ at every step. This is incompatible with our tiling strategy, as the tiles advance several steps at a time. After computing a tile column, we have advanced from the wavefront $\widetilde{H}_e$ to $\widetilde{H}_{e+\ell}$, both in the forward and reverse alignments. If a breakpoint is present in $\widetilde{H}_{e+\ell}$, but not in $\widetilde{H}_e$, we can guarantee that the breakpoint exists in the range $\{e+1,\ldots,e+\ell\}$. This way, when a breakpoint is found at $\widetilde{H}_{e+\ell}$, we recompute in global memory all the wavefronts from $\widetilde{H}_{e+1}$ to $\widetilde{H}_{e+\ell}$, checking for overlap in every step.

Once the first breakpoint is found, the total edit distance of the alignment is known. Therefore, we know the exact wavefront in which the breakpoint will be found. If an alignment has a distance of $e$, the first recursion kernel needs to stop exactly at $\widetilde{H}_{\frac{e}{2}}$. This way, we do not need to check for overlap on each step.

### 6.2.2 Accelerating the *Extend* Operation

The *extend()* operation (Eq. 2.5, Algorithm 2) is the irregular part of each biWFA step. While the *next()* operation (Eq. 2.4) depicts the same amount of work per diagonal each time it is executed, the *extend()* operation can generate a different amount of work on each diagonal. This is not ideal for GPU computing, as the work difference between threads or warps can lead to sub-optimal performance. Additionally, making character comparisons one by one rapidly becomes the performance bottleneck, issuing many instructions, memory requests, and introducing divergence. We characterize the *extend()* operator and propose a GPU-friendly method to enhance the execution time.

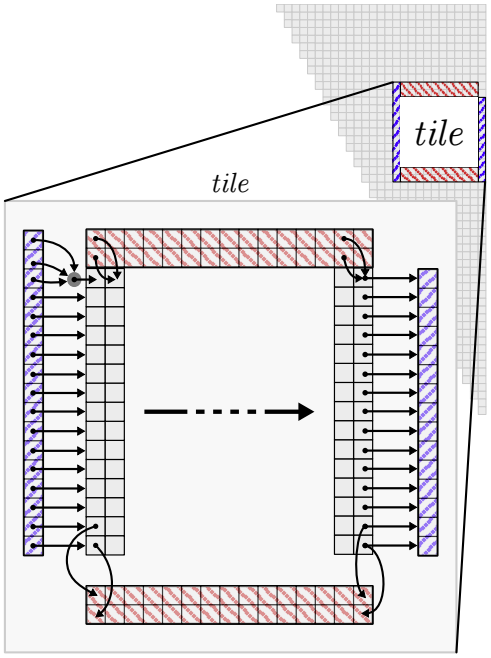



Figure 6.3: Implementation detail of a single tile. Vertical borders (in blue) are copied from global memory to a double buffer in shared memory. When the tile is finished, the result is copied back into global memory. Horizontal (in red) borders reside in shared memory, as tiles are computed column-wise by a single CUDA block.

We analyze the *extend()* operation by counting how many equal characters (i.e., *matches*) are found on each cell. This characterization is inherently dependent on the dataset. We employ simulated and real datasets to discern the patterns, providing a comprehensive understanding of the operation's behavior in datasets of our interest. We present the results in Fig. 6.4.

A consistent pattern emerges across all datasets: the majority of extensions (>99%) involve comparing fewer than 4 characters. This poses a unique challenge for GPU optimization, given that short extends result in sparse memory accesses with minimal computational demands.

Let us assume that most of the time we fetch a character from memory for performing the *extend()* is cached, which is what we observe from our kernel profiling data (see Section 6.3.3.2). In this case, 16 bytes are moved from L1 into registers even if only 1B is needed for the comparison, using less than 10% of the transferred data. Additionally, a load instruction is needed per cell, effectively using a small number of bytes per load instruction, which imposes a high stress on the load-store units (LSU).

We have analyzed and implemented several optimizations to tackle this problem, including diagonal comparison memorization and warp-level cooperative strategies. The problem we faced is that the *extend()* is composed of a (very inefficient) memory load and a single bitwise comparison per diagonal. Therefore, any strategy aimed at accelerating the *extend()* process should have almost no logic involved to have lower latency or higher throughput than the naive strategy.

We found the best-performing strategy to be comparing 4 characters in a bit-parallel manner

using bitwise operations. First, load a 32-bit integer (containing four 8-bit ASCII DNA characters) from the two sequences, and perform a bitwise *xor* and a *count trailing zeroes* instruction on the resulting comparison, to get the number of equal characters. This way, we can advance the matching character comparison in blocks of 4 elements.

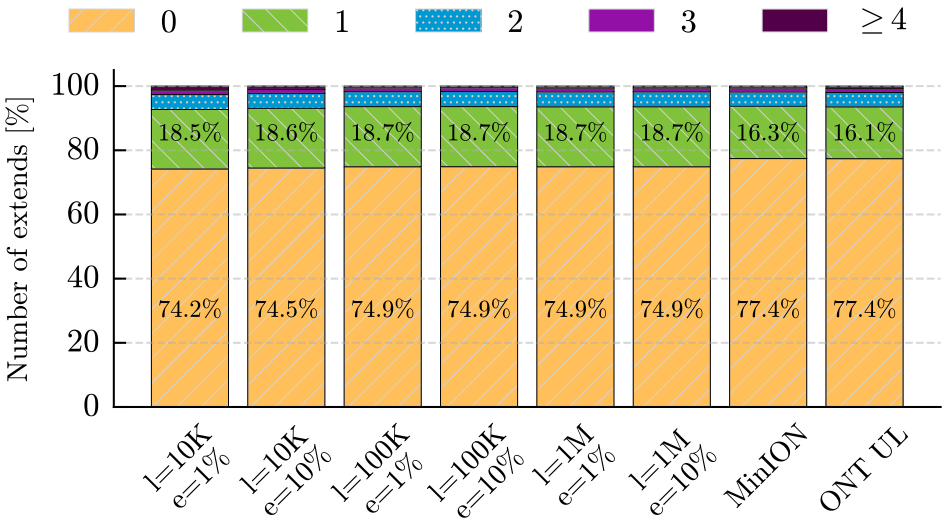



Figure 6.4: Distribution of the *extend* length on different datasets.

## 6.3 Experimental Results

The implementation evaluated here computes global edit-distance alignments of pairs of linear sequences. Optimal-score agreement (exact-score rate, denoted *R* in the tables and "recall" in the figures) compares reported scores with the exact edit-distance optimum. Support for affine and dual-affine scoring in BiGWAVE is future work.

This section evaluates the performance, scalability, and optimal-score agreement of BiGWAVE using both simulated and real biological datasets. We first compare our implementation against state-of-the-art CPU and GPU aligners on controlled synthetic workloads of increasing sequence length and error rate, and then assess its behavior on real ultra-long sequencing datasets. Finally, we characterize the implementation through detailed profiling and parameter exploration to better understand the main performance bottlenecks and the impact of key design choices.

### 6.3.1 Evaluation on Simulated Data

Tables 6.1, 6.2, and 6.3 show time (in seconds) and optimal-score agreement (percentage of alignments in which the optimal value was correctly reported) for executions using simulated datasets. Among all datasets, there is a consistent pattern: BiGWAVE achieves more speedup as the error rate grows. This is because the wavefront structures are larger at higher error rates, providing more parallelism for the GPU.

Table 6.1: Time (in seconds) and optimal-score agreement (as percentage of scores matching the exact optimum) of different aligners on simulated data. The best time for each dataset is highlighted in bold. † We could only execute TSUNAMI with a dataset of 10K alignments; hence, we multiplied the obtained time assuming perfect scalability.

| | | **1M alignments × 10Kbases** | | | | | |
|---|---|---|---|---|---|---|---|
| | | e=1% | | e=5% | | e=10% | |
| | | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) |
| **GPU** | GASAL2 | oom | n/a | oom | n/a | oom | n/a |
| | TSUNAMI | 5.3† | 100 | error | n/a | error | n/a |
| | Scrooge | 129.9 | 99.2 | 144.7 | 95.9 | 157.8 | 91.8 |
| | WFA-GPU | 1.9 | 100 | 7.5 | 100 | 21.5 | 100 |
| | BiGWAVE | 3.0 | 100 | **4.9** | 100 | **7.4** | 100 |
| **CPU** | edlib | 111.5 | 100 | 118.9 | 100 | 136.3 | 100 |
| | biWFA | **1.3** | 100 | 18.2 | 100 | 67.2 | 100 |

Regarding sequences of ≈10K nucleotides on average, when the error rate is only 1% (nominal error of ≈100), the GPU does not have a lot of parallelism available per alignment, and therefore, the CPU version of biWFA is 2.3× faster than our implementation. For the 5% and 10% cases, BiGWAVE achieves 3.7× and 9.1× speedup compared to the biWFA CPU version, respectively. Compared to Scrooge, BiGWAVE is 21.3–43.3× faster on these datasets while remaining exact. As TSUNAMI does not support splitting the input into multiple batches, we had to reduce the dataset size to 10K sequence pairs. We multiplied the obtained time, assuming perfect scalability. This strategy works for the dataset with an error rate of 1%. On datasets with 5% and 10%, it failed with an "illegal memory access" error.

Table 6.2: Time (in seconds) and optimal-score agreement (as the percentage of scores matching the exact optimum) of different aligners on simulated data. The best time for each dataset is highlighted in bold. † We could only execute TSUNAMI with a dataset of 10K alignments; hence, we multiplied the obtained time assuming perfect scalability.

| | | **100K alignments × 100Kbases** | | | | | |
|---|---|---|---|---|---|---|---|
| | | e=1% | | e=5% | | e=10% | |
| | | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) |
| **GPU** | GASAL2 | oom | n/a | oom | n/a | oom | n/a |
| | TSUNAMI | 1.1† | 0.0 | 1.1† | 0.0 | 1.1† | 0.0 |
| | Scrooge | 130.1 | 99.2 | 144.6 | 96.0 | 158.3 | 91.5 |
| | WFA-GPU | n/a | n/a | n/a | n/a | n/a | n/a |
| | BiGWAVE | **3.7** | 100 | **14.8** | 100 | **35.1** | 100 |
| **CPU** | edlib | 181.7 | 100 | 411.5 | 100 | 646.5 | 100 |
| | biWFA | 9.3 | 100 | 188.4 | 100 | 672.0 | 100 |

When aligning sequences of ≈100K bases, BiGWAVE outperforms all other implementations. Compared with the CPU version of biWFA, our implementation is 2.5×, 12.7×, and 19.1× faster for an error rate of 1%, 5%, and 10%. Compared to Scrooge, BiGWAVE is 4.5–35.2× faster. TSUNAMI was able to execute without reporting any errors, but all the alignments were reported to have a score of 0, probably due to a software error.

Table 6.3: Time (in seconds) and optimal-score agreement (as percentage of scores matching the exact optimum) of different aligners on simulated data. The best time for each dataset is highlighted in bold.

| | | **1000 alignments × 1Mbases** | | | | | |
|---|---|---|---|---|---|---|---|
| | | e=1% | | e=5% | | e=10% | |
| | | T(s) | R(%) | T(s) | R(%) | T(s) | R(%) |
| **GPU** | GASAL2 | oom | n/a | oom | n/a | oom | n/a |
| | TSUNAMI | error | n/a | error | n/a | error | n/a |
| | Scrooge | 12.8 | 99.0 | 14.4 | 96.8 | **15.8** | 90.2 |
| | WFA-GPU | n/a | n/a | n/a | n/a | n/a | n/a |
| | BiGWAVE | **1.6** | 100 | **10.3** | 100 | 28.9 | 100 |
| **CPU** | edlib | 66.5 | 100 | 232.2 | 100 | 449.9 | 100 |
| | biWFA | 11.7 | 100 | 200.4 | 100 | 710.4 | 100 |

Experiments aligning ultra-long sequences (i.e., ≈1 million bases) are where the GPU achieves the largest speedup. BiGWAVE is 7.3, 19.5, and 24.6 times faster than its CPU counterpart for sequences of 1 million characters with 1%, 5%, and 10% divergence. When the error between the sequences is about 10%, Scrooge is 1.8× faster than BiGWAVE, but optimal-score agreement drops to 90.2%. For the other cases, our proposal is 1.4–8.0× faster while remaining exact. TSUNAMI failed with an "illegal memory access" error.

### 6.3.2 Evaluation on Real Biological Data

Results in Table 6.4 show the time and optimal-score agreement of BiGWAVE and other state-of-the-art CPU and GPU aligners on real biological datasets. For the case of the ONT MinION datasets, the fastest implementation is Scrooge, with 0% optimal-score agreement. BiGWAVE is 5.3× faster than the CPU version of biWFA, and 3.5× faster than edlib.

Regarding the ONT UL dataset, Scrooge also fails to report correct alignments. BiGWAVE is 7.2× faster than biWFA, and 9.9× faster than edlib, while maintaining 100% optimal-score agreement.

We take a closer look at Scrooge to understand the distribution of the reported alignment scores. Figure 6.5 shows the cumulative score of the alignments, where the x-axis represents the score, and the y-axis represents the number of alignments with a score less than or equal to the x-axis value. The plot shows that Scrooge consistently generates alignments with scores higher than the optimal one.

Table 6.4: Time (in seconds) and optimal-score agreement (as the percentage of scores matching the exact optimum) of different aligners on real data. The best time for each dataset (with an acceptable optimal-score agreement) is highlighted in bold.

| | | **ONT MinION 10x** | | **ONT UL** | |
|---|---|---|---|---|---|
| | | T(s) | R(%) | T(s) | R(%) |
| **GPU** | GASAL2 | oom | n/a | oom | n/a |
| | TSUNAMI | error | n/a | error | n/a |
| | Scrooge | 7.7 | 0.0 | 49.6 | 0.9 |
| | WFA-GPU | n/a | n/a | n/a | n/a |
| | BiGWAVE | **27.1** | 100 | **28.4** | 100 |
| **CPU** | Edlib | 95.7 | 100 | 281.8 | 100 |
| | biWFA | 144.0 | 100 | 205.2 | 100 |

GASAL2 reported an out-of-memory (*oom*) error, as it is designed to align shorter reads. WFA-GPU could not use its accelerated GPU kernel as most alignments had $> 2^{15}$ bases. TSUNAMI reported an "illegal memory access" error.

Finally, Fig. 6.6 summarizes the overall performance of BiGWAVE compared to representative CPU and GPU aligners across both simulated and real datasets. The results highlight a consistent trend: while CPU-based approaches remain competitive for low-error and shorter sequences, GPU-based methods increasingly dominate as sequence length and error rate grow. In particular, BiGWAVE achieves the highest throughput in most scenarios, with especially significant gains for long and noisy sequences, where larger wavefronts expose more parallelism. On real ultra-long datasets, our implementation maintains high throughput and 100% optimal-score agreement on the evaluated data, whereas some competing GPU methods either fail to execute or exhibit substantially lower optimal-score agreement. This figure consolidates the key observation of this chapter: ultra-long sequence alignment benefits substantially from a GPU-aware redesign of wavefront-based algorithms.

### 6.3.3 Performance Characterization

In this section, we analyze the performance of our proposal by profiling the execution of the whole application and the GPU kernels. We evaluate the performance of our implementation. We also analyze the performance of our implementation when using different configurations of the alignment kernel, such as the number of threads per block, tile size, and dataset used.

#### 6.3.3.1 Overall System Profiling

The characteristics of the dataset used severely impact the overall performance of our proposal. Fig. 6.7 shows the application execution time when aligning different datasets, broken down into data transfer times (i.e., from Host to Device, and from Device to Host), time spent by the kernel that computes the first breakpoint, and time spent by the kernels computing the remaining sub-alignments.

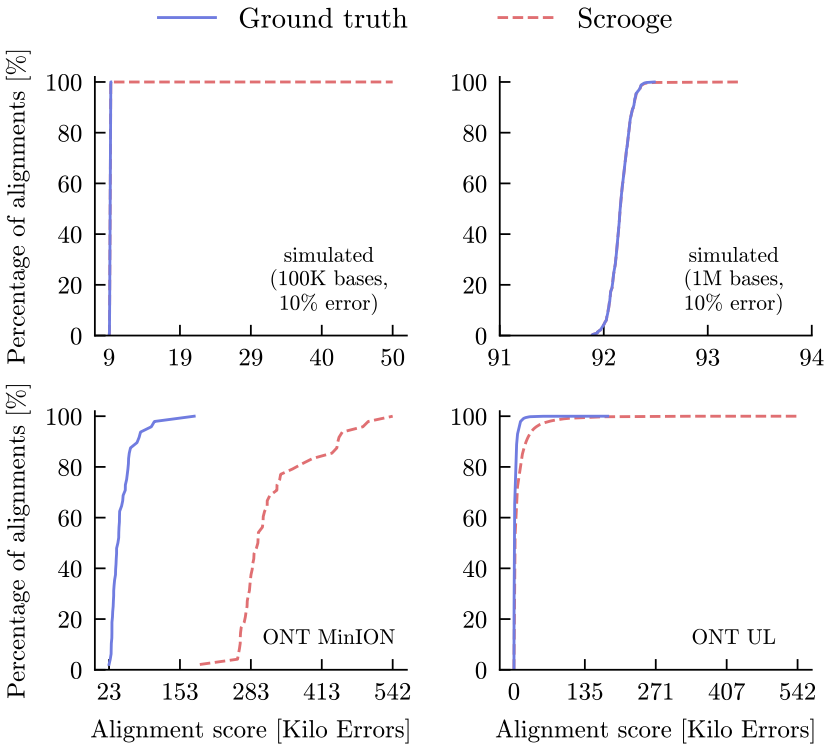



Figure 6.5: Score distribution of Scrooge compared with the optimal alignment. The x-axis represents the score, and the y-axis represents the number of alignments with a score less than or equal to the x-axis value (cumulative sum). The farther to the right the curve is, the worse the alignment.

Regarding the simulated datasets with sequences of 10K characters, a large percentage of the execution time (20%-80%) is spent transferring the sequences from the CPU into the GPU. This is due to the large number of sequences, and the relatively small nominal error, which our implementation can compute at higher throughputs than the transferring speeds. As the nominal error grows (datasets with sequences of 100K basepairs), the memory transfer time becomes a smaller percentage of the execution.

Datasets containing ultra-long reads with high errors (i.e., simulated with sequences of 1M basepairs, and the real datasets) have a few very computationally-intensive alignments. Therefore, the number of sequences that need to be transferred is small. Consequently, most of the time is spent aligning the sequences, and the impact of memory transfers on the total execution time is small (0.2%-12%).

#### 6.3.3.2 GPU Kernel Profiling

We start our analysis with the *Speed of Light* (SOL) metric provided by the NVIDIA Nsight Compute profiler. This metric is a high-level overview of the achieved utilization with respect to the theoretical maximum for the compute and memory resources of the GPU, and indicates the kernel's performance limitations. The results are shown in Fig. 6.8. Kernels with a smaller

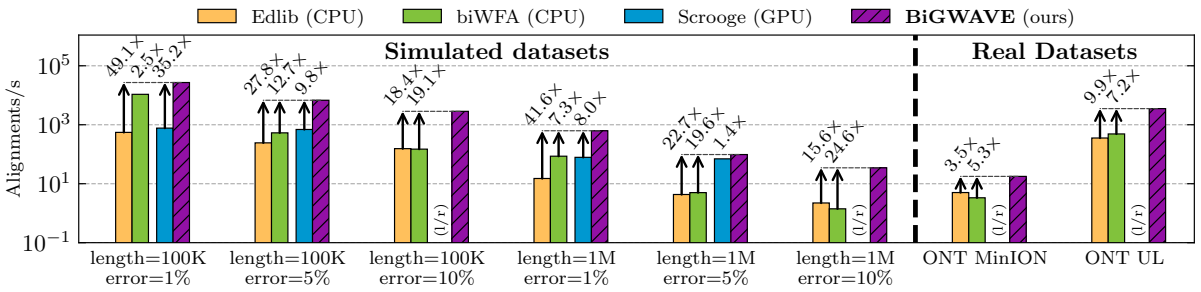



Figure 6.6: Alignments per second (in logarithmic scale) of simulated and real datasets' most relevant CPU and GPU implementations. Executions achieving an optimal-score agreement lower than 95% are marked as *low recall* (l/r)

nominal error (i.e., 1% and 5%) have a higher pressure on the memory subsystem, whereas kernels with larger nominal error (i.e., 10% real datasets) have a higher pressure on the compute units.

The memory metrics are shown in Fig. 6.9. It can be seen how cache throughput (and therefore data locality) increases with error rates. That is, sequences with fewer differences place greater pressure on the memory subsystem. This observation is consistent with the SOL results, which show that *Compute* throughput increases with the error rate.

Due to its nature, some datasets create an unbalanced workload on the GPU. This happens when very large and smaller alignments are on the same dataset. Fig. 6.10 shows the active time (in cycles) for each Streaming Multiprocessor (SM) in the GPU. The central dot indicates the average time that SMs are active, while the error bars represent the minimum and maximum SM active cycles. A large deviation from the average indicates an unbalanced workload, as an SM has been active for a long time while the others have been idle. This happens more pronouncedly in real datasets, especially in the ONT MinION dataset. This is because the ONT MinION dataset has a relatively small number of alignments, and the error rate is not uniform across the dataset, making some SMs idle while others are still processing alignments.

#### 6.3.3.3 Parameter Tuning

In this section, we characterize the effect of changing BiGWAVE parameters on GPU performance. Figure 6.11 shows how the number of threads chosen per CUDA block impacts the performance of our proposal.

For simulated datasets with sequences ranging from 10K to 100K characters, the best performance is achieved with a small to moderate number of threads per block (32-128 threads). Larger blocks reduce the number of GPU workers that can be spawned concurrently (due to the limited number of threads that can be assigned to an SM), and, in the case of these sequences, there is not enough intra-sequence parallelism to fully utilize all the threads in a large block. Regarding simulated sequences of 1M characters, the best performance is achieved with 1024 threads per CUDA block (the maximum supported by the GPU). The extremely large wavefronts generated by these alignments allow all threads in a block to exploit the available

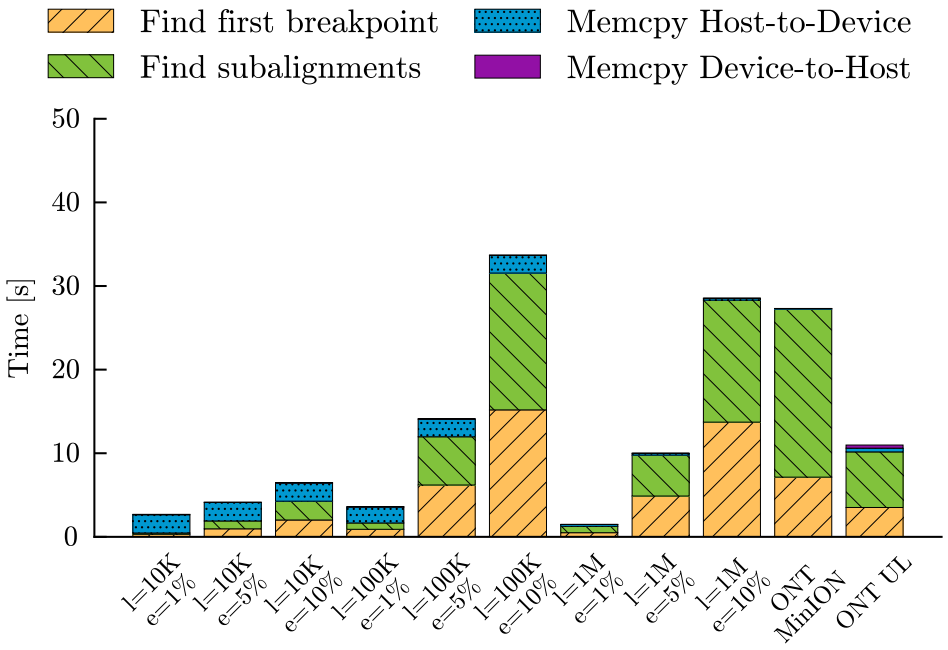



Figure 6.7: Alignment time broken down into data transfer times between CPU and GPU, time spent by the kernel that computes the first breakpoint, and time spent by the kernels computing the remaining sub-alignments.

intra-sequence parallelism. On real biological datasets, ONT MinION shows consistent performance improvements with the number of threads per block, due to the very high number of differences between the sequences. The ONT UL dataset can only be executed using $\geq 512$ threads per block, as smaller blocks cause an *out of memory* error due to the high number of GPU workers created.

The number of steps that a tile advances ($\ell$) also has performance implications, shown in Fig. 6.12. When a breakpoint is found at distance $e$, BiGWAVE has to check each step from $\widetilde{W}_{e-\ell}$ to $\widetilde{W}_e$, introducing redundant work. When the optimal distance ($s$) is not much larger than $\ell$, this overhead is significant. We observe a reduction in overall performance with $\ell > 50$ when aligning simulated datasets of 10K and 100K, and with $\ell > 100$ when aligning sequences of 1M characters. For real datasets, the best performance is achieved at $\ell = 100$.

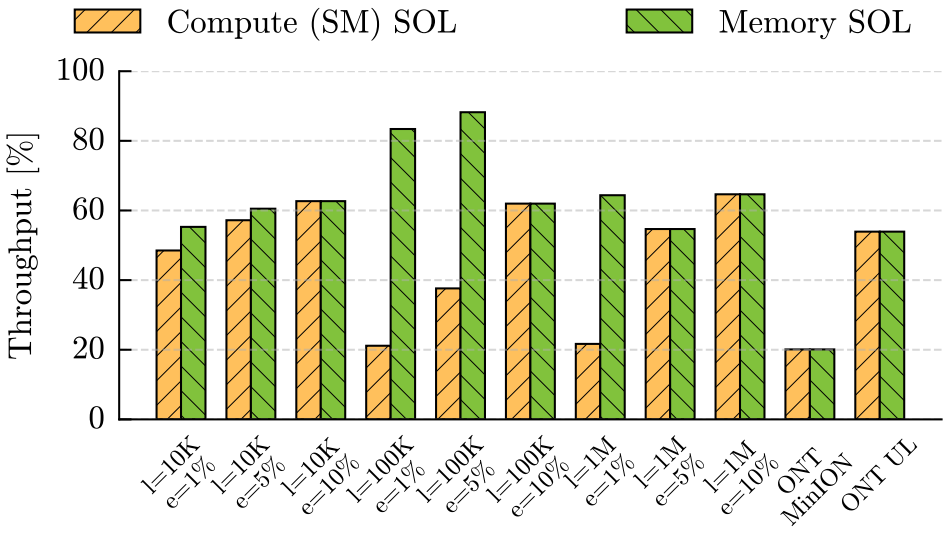



Figure 6.8: Speed of Light metric for the BiGWAVE GPU kernel.

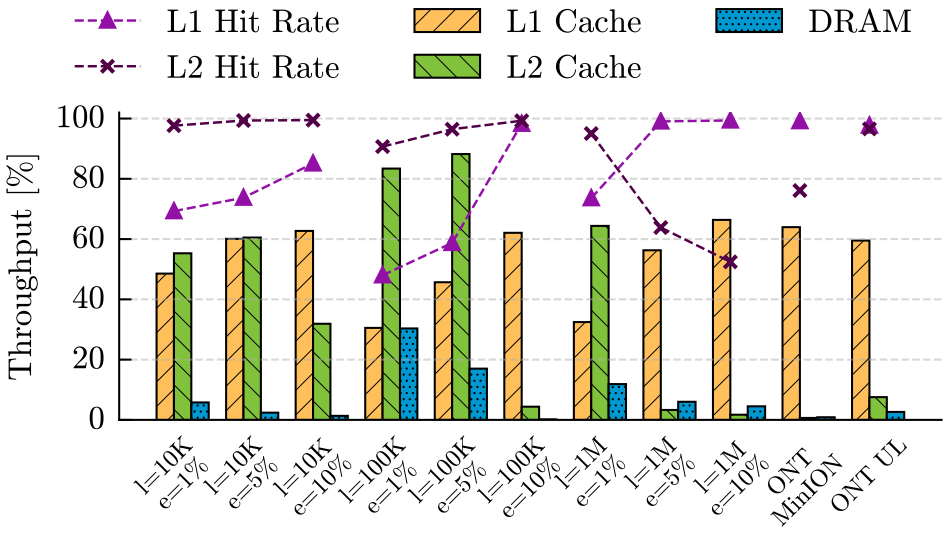



Figure 6.9: Memory metrics for the BiGWAVE GPU kernel.

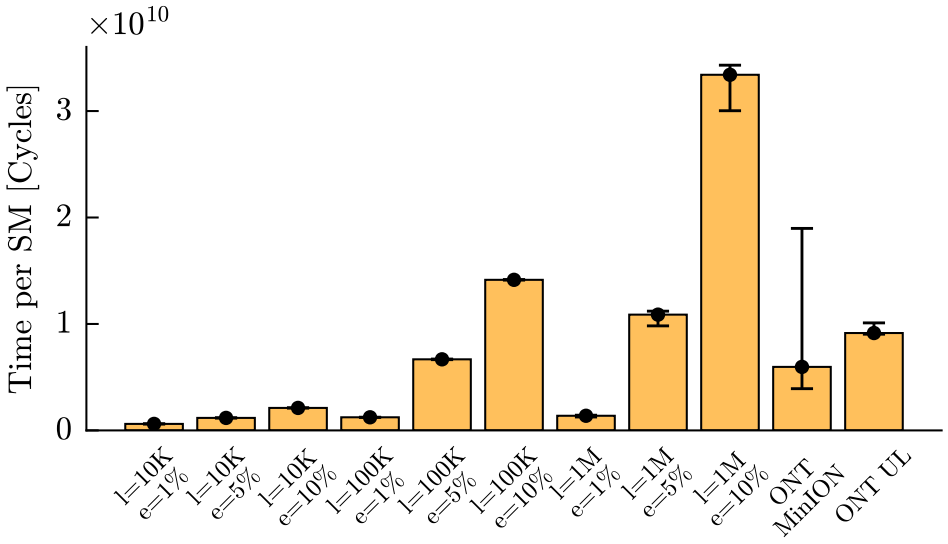



Figure 6.10: Active time (in cycles) for each SM in the GPU. The central dot indicates the average active time, while the error bars represent the minimum and maximum values recorded by the profiler. A large deviation from the average indicates an unbalanced workload.

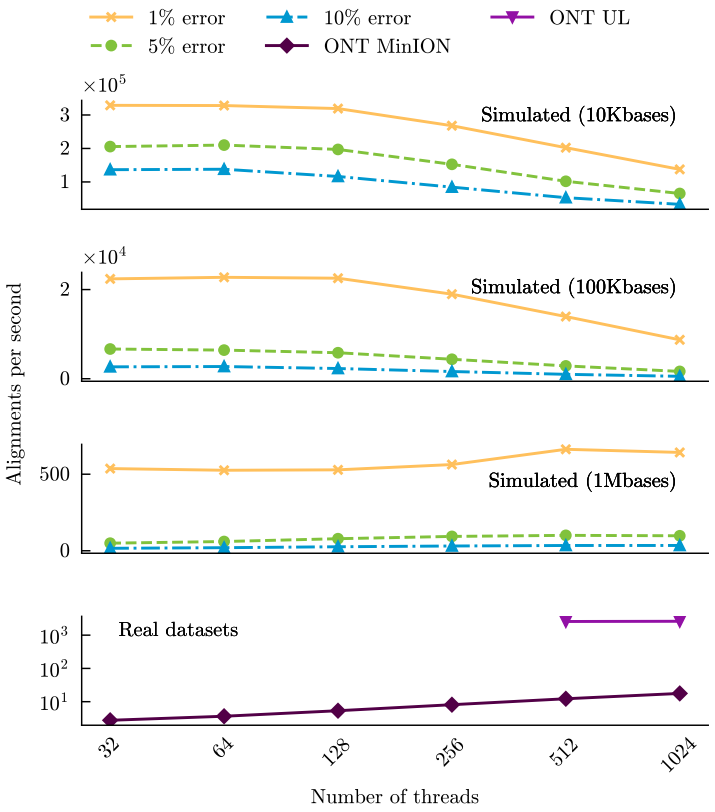



Figure 6.11: Performance (as number of alignments per second) obtained for each dataset using a different number of threads per CUDA block. Executions of ONT UL with less than 512 threads caused an *out of memory* error due to the high number of GPU aligners spawned. Real datasets y-axis is in log scale for better visualization.

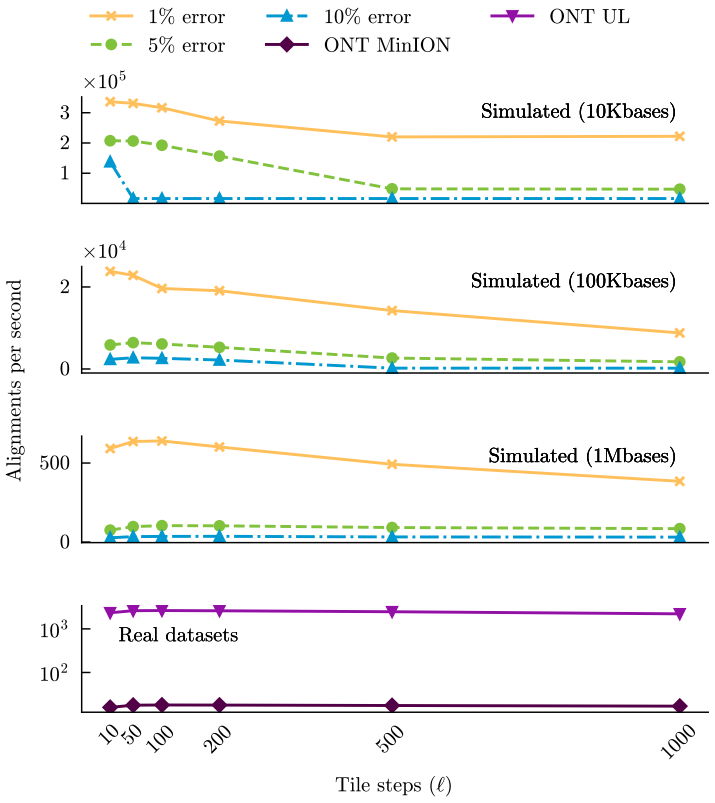



Figure 6.12: Performance (as number of alignments per second) obtained for each dataset using a different number of tile steps ($\ell$). Real datasets y-axis is in log scale for better visualization.

## 6.4 Discussion

Advances in sequencing technologies are fundamentally reshaping the requirements of bioinformatics pipelines. The increasing availability of ultra-long reads, together with the emergence of pangenomic references, demands alignment methods capable of handling significantly longer sequences while maintaining high throughput and accuracy. At the same time, GPUs have become the dominant acceleration platform in modern HPC infrastructures, driven by their widespread adoption in artificial intelligence workloads. Leveraging these architectures effectively for sequence alignment is therefore essential to sustain the scalability of genomic analyses.

In this chapter, we presented BiGWAVE, a GPU-accelerated design and implementation of the biWFA algorithm specifically designed for ultra-long sequence alignment. By combining output-sensitive alignment with a GPU-aware design, our approach overcomes the memory and scalability limitations of previous wavefront-based methods. The proposed solution exploits inter-sequence, intra-sequence, and recursion-level parallelism, together with a hyperplane tiling strategy that enables large alignments to be processed efficiently with limited on-chip memory.

Experimental results demonstrate that BiGWAVE outperforms the evaluated CPU and GPU solutions on many demanding workloads, while CPU biWFA remains faster on the low-error 10 Kbp dataset. The implementation achieves speedups of up to 24.6× on simulated datasets and 5.3–7.2× on real ultra-long sequencing datasets compared to the original multi-threaded biWFA algorithm, while maintaining exact alignment. The results also highlight a key observation of this work: as sequence length and error rate increase, the amount of exploitable parallelism grows, making GPU acceleration increasingly advantageous for ultra-long alignment tasks.

Beyond raw performance, our approach also offers favorable cost-efficiency characteristics. Our solution also provides a cost-effective approach for genome analysis systems. In the evaluated configuration, the processor used has a Recommended Customer Price (RCP) of 564 USD, whereas the GPU has an RCP starting at 1800 USD. Despite the higher device cost, the throughput improvement from BiGWAVE translates into up to 7.7× higher performance per dollar when comparing the listed GPU and CPU device costs. This ratio follows from the maximum speedup, $(710.4/28.9) \times (564/1800) \approx 7.7$, and excludes the cost of the rest of the system. In addition, offloading the alignment workload to the GPU frees the CPU to execute other pipeline stages, improving overall system utilization.

Overall, BiGWAVE provides an accurate, scalable, and efficient solution for ultra-long sequence alignment on GPU hardware. By bridging the gap between algorithmic advances in WFA alignment and the architectural characteristics of modern GPUs, this work enables practical large-scale analysis of long-read sequencing data. The implementation is released as open-source software and is publicly available at BiGWAVE repository, supporting reproducibility and facilitating its adoption in real-world genomic applications.

# Chapter 7

# Conclusions

Sequence alignment remains one of the most critical and computationally demanding stages in modern genomics pipelines, forming the foundation for downstream analyses such as variant calling, genome assembly, and comparative genomics. As sequencing technologies continue to evolve toward longer reads, higher throughput, and increasingly complex pangenomic representations, the scale and complexity of alignment workloads have grown significantly. In this context, the limitations of traditional quadratic DP algorithms and conventional CPU-based architectures have become increasingly evident in execution time, memory requirements, and energy efficiency. Addressing these challenges requires not only more efficient algorithms but also a reconsideration of how these algorithms are mapped onto modern parallel hardware.

This thesis has explored the intersection of high-performance computing and bioinformatics, with a specific focus on accelerating the WFA algorithm on GPUs. By leveraging the output-sensitive properties of WFA and combining them with hardware-aware design and performance optimizations, this work investigates how to bridge the gap between algorithmic efficiency and practical scalability on modern massively parallel computing platforms.

The central hypothesis of this work was that the WFA algorithm, despite its irregular memory access patterns and inherent data dependencies, could be effectively mapped to the massive parallelism of modern GPUs to achieve significant speedups over state-of-the-art CPU baselines. The results presented in Chapters 4, 5, and 6 confirm this hypothesis, demonstrating that, through careful memory management, algorithmic adaptations, and hardware-aware optimizations, GPUs can serve as powerful accelerators for both short-read and ultra-long sequence alignment.

## 7.1 Summary of Contributions

The main contributions and conclusions derived from this thesis can be summarized as follows:

- **Efficient Mapping of WFA to GPU Architectures.** We demonstrated that the naive porting of wavefront algorithms to GPUs is insufficient due to memory bottlenecks. In **eWFA-GPU** (Chapter 4), we introduced the *piggybacked backtrace* strategy, a succinct

encoding scheme that reduces memory requirements by up to 4×. This innovation allowed the alignment context to fit within the GPU's fast on-chip *shared memory*, enabling the first effective GPU implementation of the WFA algorithm for edit distance.

- **Scalable Gap-Affine Alignment for Long Reads.** In **WFA-GPU** (Chapter 5), we addressed the biologically relevant gap-affine model. We concluded that a simple kernel offload is insufficient for real-world scenarios involving long, noisy reads (e.g., Oxford Nanopore). Instead, a **CPU-GPU co-design** is required. The scheduler overlaps GPU alignment with CPU rescue of cases that exceed the allocated GPU resources. For full alignment-path computation, exact WFA-GPU achieves up to 4.3× speedup over multithreaded CPU WFA on real long-read datasets. The GPU-aware approximate mode reaches up to 18.2× on the Nanopore dataset, with 98.6% optimal-score agreement. This highlights the importance of heterogeneous computing strategies where the CPU and GPU work in tandem rather than in isolation.

- **Breaking the Memory Wall for Ultra-Long Sequences.** The alignment of megabase-scale sequences presents a unique challenge due to the sheer size of the DP matrices. In **BiGWAVE** (Chapter 6), we showed that the Bidirectional WFA (biWFA) could be adapted for GPUs using a *hyperplane tiling strategy*. This approach effectively manages intra-sequence parallelism, allowing the GPU to process massive sequences by breaking the problem into manageable tiles that fit in on-chip memory. This contribution establishes the first GPU-based solution for exact alignment of megabase-scale sequences on commodity GPUs, achieving speedups of up to 24.6× on simulated datasets and 5.3–7.2× on real ultra-long datasets over multithreaded CPU biWFA.

## 7.2 Concluding Discussion and Perspectives

The results presented throughout this thesis highlight a key takeaway: the combination of output-sensitive algorithms and hardware-aware design is essential to meet the growing demands of modern genomics. Traditional DP approaches, while robust and widely adopted, are increasingly inadequate for the scale and complexity of current sequencing workloads. In contrast, methods such as WFA and biWFA provide a more suitable algorithmic foundation, as they adapt their computational effort to the actual similarity between sequences. When coupled with the massive parallelism of GPUs, the exact variants improve throughput while preserving optimality under their selected scoring functions. The approximate WFA-GPU mode introduces a separate performance-optimality trade-off.

A central observation of this work is that efficient GPU acceleration of sequence alignment is not merely a matter of parallelizing existing algorithms, but rather requires a co-design approach in which algorithmic structure, memory hierarchy, and execution model are jointly considered. Techniques such as wavefront compression, piggyback backtracing, tiling strategies, and multi-level parallelism are not isolated optimizations but necessary components for bridging the gap between theoretical efficiency and practical performance on modern hardware. This perspective is likely to become increasingly relevant as architectures continue to evolve toward higher degrees of parallelism and deeper memory hierarchies.

The amount of parallelism exposed by WFA is input- and score-dependent. Low-score pairs generate narrow wavefronts, while difficult pairs expose more diagonals but also require more work and memory. Parallelization granularity should consequently change with workload difficulty: short or easy pairs favor concurrency across alignments and small cooperative groups, whereas difficult ultra-long pairs benefit from greater intra-alignment and recursion-level parallelism. BiGWAVE's block-size experiments make this trade-off explicit.

CPU and GPU execution are complementary. CPUs remain competitive for low-score problems and handle cases that exceed GPU allocations; GPUs benefit from sufficiently large batches or wide wavefronts. Effective acceleration therefore depends on scheduling work at a suitable granularity and accounting for transfer and synchronization costs.

The measured gains apply to the evaluated pairwise linear-sequence workloads, hardware, scoring functions, and timing boundaries. They do not demonstrate acceleration of a complete read mapper or genomics pipeline, nor direct graph/pangenome alignment. Approximate WFA-GPU may return suboptimal scores depending on heuristic parameters, and BiGWAVE currently uses edit distance.

Finally, as sequencing technologies continue to evolve, the definition of "challenging" alignment problems will also change. Ultra-long reads, high-error-rate sequences, and increasingly complex genomic references will push both algorithms and hardware to their limits. In this context, the ideas explored in this thesis provide a foundation for the next generation of sequence alignment tools. Continued collaboration between the fields of high-performance computing and bioinformatics will be essential to fully realize this potential and to ensure that computational methods keep pace with advances in sequencing technologies.

## 7.3 Future Work

Three priorities follow from the cross-chapter results: adaptive CPU/GPU dispatch, production-pipeline integration, and richer scoring for BiGWAVE.

**Adaptive CPU/GPU Dispatch.** A next step is to select the execution device and parallelization granularity using sequence length, an estimate of the alignment score, available memory, and batch size. This extends WFA-GPU's existing CPU rescue mechanism into proactive scheduling. The policy should account for estimation, transfer, and scheduling costs and be evaluated on mixed workloads, including cases where CPU execution is faster.

**Integration into Production Mapping Pipelines.** The algorithms presented in this thesis (eWFA-GPU, WFA-GPU, and BiGWAVE) currently operate as standalone libraries or benchmarking tools. To maximize their impact on the bioinformatics community, a critical next step is to integrate these GPU kernels into standard read mapping pipelines such as **Minimap2** or **BWA-MEM2**. This integration would require substantial engineering work, efficiently managing the asynchronous data flow between the host application (performing seeding and chaining) and the GPU accelerator (performing the alignment extension). Evaluation should measure total mapper throughput and latency, including seeding, chaining, transfers, and output, and assess mapping and downstream variant-calling accuracy against appropriate truth sets.

**Advanced Cost Models and Hyperplane Tiling.** The evaluated BiGWAVE implementation uses edit distance. Extending it first to affine and then dual-affine gap penalties is a priority.

WFA-GPU already supports standard gap-affine costs. However, bioinformatics tools often require **dual-affine (convex) gap costs** to model short and long gaps differently. Implementing dual-affine models increases memory pressure due to the requirement of additional matrices (in traditional DP algorithms) or wavefront components (in WFA and biWFA). A promising direction is to apply the **hyperplane tiling strategy** introduced in Chapter 6 to both standard WFA and biWFA under affine and dual-affine models. This would allow the alignment of ultra-long reads with complex gap penalties, leveraging the GPU shared memory.

## 7.4 Financial Support

This research has been supported by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of the total cost eligible under the DRAC project [001-P-001723], the Spanish Ministerio de Ciencia e Innovación under grant [PRE2021-101059], the Spanish Ministry, Ministerio para la Transformación Digital y de la Función Pública, the European Union – NextGenerationEU through the Càtedra Chip UPC project (grant number TSI-069100-2023-0015), and by the Ministerio de Ciencia, Innovación y Universidades y de la Agencia Estatal de Investigación (CNS2025-166442 funded by MICIU/AEI/10.13039/501100011033).

# Bibliography


[1] Quim Aguado-Puig, Max Doblas, Christos Matzoros, Antonio Espinosa, Juan Carlos Moure, Santiago Marco-Sola, and Miquel Moreto. Wfa-gpu: gap-affine pairwise read-alignment using gpus. *Bioinformatics*, 39(12):btad701, 2023.

[2] Quim Aguado-Puig, Santiago Marco-Sola, Juan Carlos Moure, David Castells-Rufas, Lluc Alvarez, Antonio Espinosa, and Miquel Moreto. Accelerating edit-distance sequence alignment on gpu using the wavefront algorithm. *IEEE access*, 10:63782–63796, 2022.

[3] Nauman Ahmed, Jonathan Lévy, Shanshan Ren, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. Gasal2: a gpu accelerated sequence alignment library for high-throughput ngs data. *BMC bioinformatics*, 20:1–20, 2019.

[4] Nauman Ahmed, Tong Dong Qiu, Koen Bertels, and Zaid Al-Ars. Gpu acceleration of darwin read overlapper for de novo assembly of long dna reads. *BMC bioinformatics*, 21(Suppl 13):388, 2020.

[5] Alejandro Alonso-Marín, Ivan Fernandez, Quim Aguado-Puig, Juan Gómez-Luna, Santiago Marco-Sola, Onur Mutlu, and Miquel Moreto. Bimsa: Accelerating long sequence alignment using processing-in-memory. *Bioinformatics*, page btae631, 2024.

[6] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. Shouji: a fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics*, 35(21):4255–4263, 2019.

[7] Mohammed Alser, Hasan Hassan, Hongyi Xin, Oğuz Ergin, Onur Mutlu, and Can Alkan. Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping. *Bioinformatics*, 33(21):3355–3363, 2017.

[8] Mohammed Alser, Joel Lindegger, Can Firtina, Nour Almadhoun, Haiyu Mao, Gagandeep Singh, Juan Gomez-Luna, and Onur Mutlu. Going from molecules to genomic variations to scientific discovery: Intelligent algorithms and architectures for intelligent genome analysis. *arXiv preprint arXiv:2205.07957*, 2022.

[9] Muaaz G Awan, Jack Deslippe, Aydin Buluc, Oguz Selvitopi, Steven Hofmeyr, Leonid Oliker, and Katherine Yelick. Adept: a domain independent sequence alignment strategy for gpu architectures. *BMC bioinformatics*, 21(1):1–29, 2020.

[10] Khaled Balhaf, Mohammad A Alsmirat, Mahmoud Al-Ayyoub, Yaser Jararweh, and Mohammed A Shehab. Accelerating levenshtein and damerau edit distance algorithms using gpu with unified memory. In *2017 8th international conference on information and communication systems (ICICS)*, pages 7–11. IEEE, 2017.

[11] Khaled Balhaf, Mohammed A Shehab, T Wala'a, Mahmoud Al-Ayyoub, Mohammed Al-Saleh, and Yaser Jararweh. Using gpus to speed-up levenshtein edit distance computation. In *2016 7th International Conference on Information and Communication Systems (ICICS)*, pages 80–84. IEEE, 2016.

[12] David Bednárek, Michal Brabec, and Martin Kruliš. Improving matrix-based dynamic programming on massively parallel accelerators. *Information Systems*, 64:175–193, 2017.

[13] Sairam Behera, Severine Catreux, Massimiliano Rossi, Sean Truong, Zhuoyi Huang, Michael Ruehle, Arun Visvanath, Gavin Parnaby, Cooper Roddey, Vitor Onuchic, et al. Comprehensive and accurate genome analysis at scale using dragen accelerated algorithms. *bioRxiv*, 2024.

[14] Mehmet E Belviranli, Peng Deng, Laxmi N Bhuyan, Rajiv Gupta, and Qi Zhu. Peerwave: Exploiting wavefront parallelism on gpus with peer-sm synchronization. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 25–35, 2015.

[15] Rory Bowden, Robert W Davies, Andreas Heger, Alistair T Pagnamenta, Mariateresa de Cesare, Laura E Oikkonen, Duncan Parkes, Colin Freeman, Fatima Dhalla, Smita Y Patel, et al. Sequencing of human genomes with nanopore technology. *Nature communications*, 10(1):1869, 2019.

[16] Beatrice Branchini, Giulia Gerometta, Luisa Cicolini, Alberto Zeni, Emanuele Del Sozzo, and Marco D Santambrogio. Surfing the wavefront of genome alignment. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1754–1758. IEEE, 2022.

[17] Liangwei Cai, Qi Wu, Tongsheng Tang, Zhi Zhou, and Yuan Xu. A design of fpga acceleration system for myers bit-vector based on opencl. In *2019 International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS)*, pages 305–312. IEEE, 2019.

[18] Reed A Cartwright. Logarithmic gap costs decrease alignment accuracy. *BMC bioinformatics*, 7(1):1–12, 2006.

[19] David Castells-Rufas, Santiago Marco-Sola, Quim Aguado-Puig, Antonio Espinosa-Morales, Juan Carlos Moure, Lluc Alvarez, and Miquel Moretó. Opencl-based fpga accelerator for semi-global approximate string matching using diagonal bit-vectors. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 174–178. IEEE, 2021.

[20] David Castells-Rufas, Santiago Marco-Sola, Juan Carlos Moure, Quim Aguado, and Antonio Espinosa. Fpga acceleration of pre-alignment filters for short read mapping with hls. *IEEE Access*, 10:22079–22100, 2022.

[21] Alejandro Chacón, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. Boosting the fm-index on the gpu: Effective techniques to mitigate random memory access. *IEEE/ACM transactions on computational biology and bioinformatics*, 12(5):1048–1059, 2014.

[22] Alejandro Chacón, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. Thread-cooperative, bit-parallel computation of levenshtein distance on gpu. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 103–112, 2014.

[23] Alejandro Chacón, Santiago Marco Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. Fm-index on gpu: A cooperative scheme to reduce memory footprint. In *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 1–9. IEEE, 2014.

[24] Fred J Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[25] Amine Dhraief, Raik Issaoui, and Abdelfettah Belghith. Parallel computing the longest common subsequence (lcs) on gpus: efficiency and language suitability. In *The 1st International Conference on Advanced Communications and Computation (INFOCOMP)*, 2011.

[26] Peng Di, Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In *2012 41st International Conference on Parallel Processing*, pages 350–359. IEEE, 2012.

[27] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj. A framework for high-throughput sequence alignment using real processing-in-memory systems. *Bioinformatics*, 39(5):btad155, 2023.

[28] David Dimmock, Sara Caylor, Bryce Waldman, Wendy Benson, Christina Ashburner, Jason L Carmichael, Jeanne Carroll, Elaine Cham, Shimul Chowdhury, John Cleary, et al. Project baby bear: Rapid precision care incorporating rwgs in 5 california children's hospitals demonstrates improved clinical outcomes and reduced costs of care. *The American Journal of Human Genetics*, 108(7):1231–1238, 2021.

[29] Max Doblas, Oscar Lostes-Cazorla, Quim Aguado-Puig, Nick Cebry, Pau Fontova-Musté, Christopher Frances Batten, Santiago Marco-Sola, and Miquel Moretó. Gmx: Instruction set extensions for fast, scalable, and efficient genome sequence alignment. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1466–1480, 2023.

[30] Max Doblas, Oscar Lostes-Cazorla, Quim Aguado-Puig, Cristian Iñiguez, Miquel Moreto, and Santiago Marco-Sola. Quicked: high-performance exact sequence alignment based on bound-and-align. *Bioinformatics*, 41(3):btaf112, 2025.

[31] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC bioinformatics*, 9(1):1–9, 2008.

[32] Jordan M Eizenga and Benedict Paten. Improving the time and space complexity of the wfa algorithm and generalizing its scoring. *BioRxiv*, pages 2022–01, 2022.

[33] Reza Farivar, Harshit Kharbanda, Shivaram Venkataraman, and Roy H Campbell. An algorithm for fast edit distance computation on gpus. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9. IEEE, 2012.

[34] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

[35] Can Firtina, Maximilian Mordig, Harun Mustafa, Sayan Goswami, Nika Mansouri Ghiasi, Stefano Mercogliano, Furkan Eris, Joël Lindegger, Andre Kahles, and Onur Mutlu. Rawsamble: Overlapping and assembling raw nanopore signals using a hash-based seeding mechanism. *arXiv preprint arXiv:2410.17801*, 2024.

[36] Erik Garrison, Andrea Guarracino, Simon Heumos, Flavia Villani, Zhigui Bao, Lorenzo Tattini, Jörg Hagmann, Sebastian Vorbrugg, Santiago Marco-Sola, Christian Kubica, et al. Building pangenome graphs. *Nature Methods*, pages 1–5, 2024.

[37] Giulia Gerometta, Alberto Zeni, and Marco D Santambrogio. Tsunami: a gpu implementation of the wfa algorithm. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 150–161. IEEE, 2023.

[38] John E Gorzynski, Sneha D Goenka, Kishwar Shafin, Tanner D Jensen, Dianna G Fisk, Megan E Grove, Elizabeth Spiteri, Trevor Pesout, Jean Monlong, Gunjan Baid, et al. Ultrarapid nanopore genome sequencing in a critical care setting. *New England Journal of Medicine*, 386(7):700–702, 2022.

[39] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

[40] Osamu Gotoh. Optimal sequence alignment allowing for long gaps. *Bulletin of mathematical biology*, 52(3):359–373, 1990.

[41] Xun Gu and Wen-Hsiung Li. The size distribution of insertions and deletions in human and rodent pseudogenes suggests the logarithmic gap penalty for sequence alignment. *Journal of molecular evolution*, 40(4):464–473, 1995.

[42] Yufeng Gu, Arun Subramaniyan, Tim Dunn, Alireza Khadem, Kuan-Yu Chen, Somnath Paul, Md Vasimuddin, Sanchit Misra, David Blaauw, Satish Narayanasamy, et al. Gendp: A framework of dynamic programming acceleration for genome sequencing analysis. In

*Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[43] Dan Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News*, 28(4):41–60, 1997.

[44] Abbas Haghi, Lluc Alvarez, Jordi Front, Juan Miguel De Haro Ruiz, Roger Figueras, Max Doblas, Santiago Marco-Sola, and Miquel Moreto. Wfasic: A high-performance asic accelerator for dna sequence alignment on a risc-v soc. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 392–401, 2023.

[45] Abbas Haghi, Santiago Marco-Sola, Lluc Alvarez, Dionysios Diamantopoulos, Christoph Hagleitner, and Miquel Moreto. An fpga accelerator of the wavefront algorithm for genomics pairwise alignment. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 151–159. IEEE, 2021.

[46] Abbas Haghi, Santiago Marco-Sola, Lluc Alvarez, Dionysios Diamantopoulos, Christoph Hagleitner, and Miquel Moreto. Wfa-fpga: An efficient accelerator of the wavefront algorithm for short and long read genomics alignment. *Future Generation Computer Systems*, 149:39–58, 2023.

[47] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[48] ThienLuan Ho, Seung-Rohk Oh, and HyunJin Kim. A parallel approximate string matching under levenshtein distance on graphics processing units using warp-shuffle operations. *PloS one*, 12(10):e0186251, 2017.

[49] Jörn Hoffmann, Dirk Zeckzer, and Martin Bogdan. Using fpgas to accelerate myers bit-vector algorithm. In *XIV Mediterranean Conference on Medical and Biological Engineering and Computing 2016*, pages 535–541. Springer, 2016.

[50] François Irigoin and Rémi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329, 1988.

[51] Kailash W Kalare, Mohammad S Obaidat, Jitendra V Tembhurne, Chandrashekhar Meshram, and Kuei-Fang Hsiao. Parallelization of global sequence alignment on graphics processing unit. In *2020 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*, pages 1–5. IEEE, 2020.

[52] Katsuya Kawanami and Noriyuki Fujimoto. A gpu implementation of a bit-parallel algorithm for computing the longest common subsequence. *Information and Media Technologies*, 10(1):8–16, 2015.

[53] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome research*, 27(5):722–736, 2017.

[54] Sam Kovaka, Yunfan Fan, Bohan Ni, Winston Timp, and Michael C Schatz. Targeted nanopore sequencing by real-time mapping of raw electrical signal with uncalled. *Nature biotechnology*, 39(4):431–441, 2021.

[55] Sandeep Kumar and Eugene H Spafford. A pattern matching model for misuse intrusion detection. 1994.

[56] Gad M Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of algorithms*, 10(2):157–169, 1989.

[57] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[58] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. 10(8):707–710, 1966.

[59] Da Li and Michela Becchi. Multiple pairwise sequence alignments with the needleman-wunsch algorithm on gpu. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1471–1472. IEEE, 2012.

[60] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[61] Heng Li, Xiaowen Feng, and Chong Chu. The design and construction of reference pangenome graphs with minigraph. *Genome biology*, 21(1):265, 2020.

[62] Yuanzhe Li and Loren Schwiebert. Memory-optimized wavefront parallelism on gpus. *International Journal of Parallel Programming*, 48:1008–1031, 2020.

[63] Zuqing Li, Aakashdeep Goyal, and Haklin Kimm. Parallel longest common sequence algorithm on multicore systems using openacc, openmp and openmpi. In *2017 IEEE 11th international symposium on embedded multicore/many-core systems-on-chip (MCSoC)*, pages 158–165. IEEE, 2017.

[64] Joël Lindegger, Can Firtina, Nika Mansouri Ghiasi, Mohammad Sadrosadati, Mohammed Alser, and Onur Mutlu. Rawalign: Accurate, fast, and scalable raw nanopore signal mapping via combining seeding and alignment. *IEEE Access*, 2024.

[65] Joël Lindegger, Damla Senol Cali, Mohammed Alser, Juan Gómez-Luna, Nika Mansouri Ghiasi, and Onur Mutlu. Scrooge: a fast and memory-frugal genomic sequence aligner for cpus, gpus, and asics. *Bioinformatics*, 39(5):btad151, 2023.

[66] Lorién López-Villellas, Cristian Iñiguez, Albert Jiménez-Blanco, Quim Aguado-Puig, Miquel Moretó, Jesús Alastruey-Benedé, Pablo Ibáñez, and Santiago Marco-Sola. Singletrack: An algorithm for improving memory consumption and performance of gap-affine sequence alignment. *bioRxiv*, pages 2025–10, 2025.

[67] Lorién López-Villellas, Rubén Langarita-Benítez, Asaf Badouh, Víctor Soria-Pardos, Quim Aguado-Puig, Guillem López-Paradís, Max Doblas, Javier Setoain, Chulho Kim, Makoto Ono, et al. Genarchbench: A genomics benchmark suite for arm hpc processors. *Future Generation Computer Systems*, 157:313–329, 2024.

[68] Tareq Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and David Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing*, 37(4):C439–C464, 2015.

[69] Santiago Marco-Sola, Jordan M Eizenga, Andrea Guarracino, Benedict Paten, Erik Garrison, and Miquel Moreto. Optimal gap-affine alignment in o (s) space. *bioRxiv*, 2022.

[70] Santiago Marco-Sola, Jordan M Eizenga, Andrea Guarracino, Benedict Paten, Erik Garrison, and Miquel Moreto. Optimal gap-affine alignment in o (s) space. *Bioinformatics*, 39(2):btad074, 2023.

[71] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, 37(4):456–463, 2021.

[72] Santiago Marco-Sola, Michael Sammeth, Roderic Guigó, and Paolo Ribeca. The gem mapper: fast, accurate and versatile alignment by filtration. *Nature methods*, 9(12):1185–1188, 2012.

[73] Karen H Miga and Ting Wang. The need for a human pangenome reference sequence. *Annual Review of Genomics and Human Genetics*, 22:81–102, 2021.

[74] Yasuaki Mitani, Fumihiko Ino, and Kenichi Hagihara. Parallelizing exact and approximate string matching via inclusive scan on a gpu. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1989–2002, 2016.

[75] Eugene W Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[76] Eugene W Myers and Webb Miller. Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17, 1988.

[77] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

[78] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[79] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[80] Lucas SN Nunes, Jacir L Bordim, Koji Nakano, and Yasuaki Ito. A fast approximate string matching algorithm on gpu. In *2015 Third international symposium on computing and networking (CANDAR)*, pages 188–192. IEEE, 2015.

[81] Seongyeon Park, Junguk Hong, Jaeyong Song, Hajin Kim, Youngsok Kim, and Jinho 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*, pages 431–444, 2024.

[82] Ryan Poplin, Pi-Chuan Chang, David Alexander, Scott Schwartz, Thomas Colthurst, Alexander Ku, Dan Newburger, Jojo Dijamco, Nam Nguyen, Pegah T Afshar, et al. A universal snp and small-indel variant caller using deep neural networks. *Nature biotechnology*, 36(10):983–987, 2018.

[83] Ryan Poplin, Valentin Ruano-Rubio, Mark A DePristo, Tim J Fennell, Mauricio O Carneiro, Geraldine A Van der Auwera, David E Kling, Laura D Gauthier, Ami Levy-Moonshine, David Roazen, et al. Scaling accurate genetic variant discovery to tens of thousands of samples. *biorxiv*, page 201178, 2017.

[84] James R Priest, Scott R Ceresnak, Frederick E Dewey, Lindsey E Malloy-Walton, Kyla Dunn, Megan E Grove, Marco V Perez, Katsuhide Maeda, Anne M Dubin, and Euan A Ashley. Molecular diagnosis of long qt syndrome at 10 days of life by rapid whole genome sequencing. *Heart Rhythm*, 11(10):1707–1713, 2014.

[85] David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6, 1972.

[86] Peter H Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

[87] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[88] Martin Šošić and Mile Šikić. Edlib: a c/c++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

[89] Dominik Stanojevic, Dehui Lin, Paola Florez De Sessions, and Mile Sikic. Telomere-to-telomere phased genome assembly using error-corrected simplex nanopore reads. *bioRxiv*, pages 2024–05, 2024.

[90] Martin Steinegger and Johannes Söding. Mmseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. *Nature biotechnology*, 35(11):1026–1028, 2017.

[91] Hajime Suzuki and Masahiro Kasahara. Acceleration of nucleotide semi-global alignment with adaptive banded dynamic programming. *BioRxiv*, page 130633, 2017.

[92] Ayumu Tomiyama and Reiji Suda. Automatic parameter optimization for edit distance algorithm on gpu. In *International Conference on High Performance Computing for Computational Science*, pages 420–434. Springer, 2012.

[93] Tuan Tu Tran, Yongchao Liu, and Bertil Schmidt. Bit-parallel approximate pattern matching: Kepler gpu versus xeon phi. *Parallel Computing*, 54:128–138, 2016.

[94] Esko Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985.

[95] Esko Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1):132–137, 1985.

[96] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[97] Michael S Waterman, Temple F Smith, and William A Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976.

[98] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[99] Keh Kok Yong and Hong Hoe Ong. Accelerating bit-parallel approximate matching on gpu platforms for small patterns. In *2018 Fourth International Conference on Advances in Computing, Communication & Automation (ICACCA)*, pages 1–5. IEEE, 2018.

[100] Alberto Zeni, Giulia Guidi, Marquita Ellis, Nan Ding, Marco D Santambrogio, Steven Hofmeyr, Aydın Buluç, Leonid Oliker, and Katherine Yelick. Logan: High-performance gpu-based x-drop long-read alignment. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 462–471. IEEE, 2020.

[101] Jikai Zhang, Haidong Lan, Yuandong Chan, Yuan Shang, Bertil Schmidt, and Weiguo Liu. Bgsa: A bit-parallel global sequence alignment toolkit for multi-core and many-core architectures. *Bioinformatics*, 35(13):2306–2308, 2019.

[102] Mengyao Zhao, Wan-Ping Lee, Erik P Garrison, and Gabor T Marth. Ssw library: an simd smith-waterman c/c++ library for use in genomic applications. *PloS one*, 8(12):e82138, 2013.

[103] Eduard Zorita, Pol Cusco, and Guillaume J Filion. Starcode: sequence clustering based on all-pairs search. *Bioinformatics*, 31(12):1913–1919, 2015.

[104] Quan Zou, Gang Lin, Xingpeng Jiang, Xiangrong Liu, and Xiangxiang Zeng. Sequence clustering in bioinformatics: an empirical study. *Briefings in bioinformatics*, 21(1):1–10, 2020.