# Arquitectura hardware para la aceleración de ensamble de secuencias de ADN

por

**Roberto Hernández Munive**

Tesis sometida como requerimiento parcial para obtener el grado de

Maestro en Ciencias, en el área de Ciencias Computacionales

por el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Febrero, 2018

Tonantzintla, Puebla

Supervisor:

**Dra. Claudia Feregrino Uribe**

Coordinación de Ciencias Computacionales

INAOE



*A mi familia y amigos.*

# Índice general

# Índice de figuras

# Índice de tablas

# Agradecimientos

A mi familia y amigos por su apoyo y compañía.

Al Instituto Nacional de Astrofísica, Óptica y Electrónica, por la oportunidad de haber sido uno de sus estudiantes.

A los profesores que aportaron en mi formación académica y personal.

Al Consejo Nacional de Ciencia y Tecnología, por el apoyo económico para estudiar la maestría con la beca número 422282.

# Resumen

En años recientes se ha incrementado significativamente la digitalización de genomas gracias a las nuevas tecnologías de secuenciación; sin embargo, los datos generados por estas herramientas han superado la capacidad de las tecnologías de procesamiento y el almacenamiento, por estos motivos, se han desarrollado nuevos métodos y herramientas de análisis. La importancia del análisis de los genomas abarca desde el estudio de organismos de los cuáles no se ha obtenido su genoma completo debido a su gran tamaño, hasta el desarrollo de medicamentos personalizados.

En esta investigación, se exploran las metodologías utilizadas para realizar la tarea de ensamble de secuencias de ADN. Con base en la gran cantidad de datos disponible se seleccionan las metodologías y algoritmos que permitan obtener resultados en un tiempo aceptable, de acuerdo a los requerimientos de los datos. En la tarea de ensamble de secuencias de ADN hay tres enfoques principales: probabilista, procesamiento digital de señales y la transformación Burrows-Wheeler (BWT). Este último enfoque es el más utilizado debido a que permite el ordenamiento reversible de los datos, facilitando la búsqueda de patrones y la mejora de la compresión.

El objetivo principal de esta investigación es el diseño de una arquitectura hardware para acelerar el ensamble de secuencias de ADN. Se propone una estrategia de preprocesamiento para reducir el costo computacional del cálculo de una única cadena BWT mediante múltiples cadenas BWT. El cálculo de esta cadena BWT, a partir de secuencias de ADN sin procesamiento o transformadas en cadenas BWT,

ayuda a la creación del índice-FM, que permite agilizar la búsqueda de patrones mediante los cuales se forman los grafos de ensamble. Para validar la arquitectura propuesta se realiza su implementación y comparación de resultados contra las implementaciones en software, demostrando un incremento significativo en la velocidad de procesamiento.

# Capítulo 1

# Introducción

## 1.1. Bioinformática molecular

La bioinformática es un campo revolucionario en biología molecular y ciencias computacionales. Una definición de bioinformática molecular es el uso de bases de datos y algoritmos para analizar proteínas, genes y la colección completa de ácido desoxirribonucléico (ADN) que define un organismo (un genoma). Un desafío mayor para la biología es hallar el sentido de enormes cantidades de secuencias y datos estructurales que es generado por secuenciadores de genomas, proteómica y otros esfuerzos de gran escala en la biología molecular.

De acuerdo a la definición del grupo de Institutos Nacionales de la Salud (NIH por sus siglas en inglés; National Institutes of Health), bioinformática es *la organización y análisis de información biológica o relacionada con la biología, normalmente involucrando el uso de computadoras para desarrollar bases de datos, mecanismos de recuperación y herramientas de análisis de datos, especialmente en los campos de biología molecular, biología estructural, y genética.* La disciplina relacionada de biología computacional es *el desarrollo y aplicación de métodos de análisis de da-*

*tos y teoría, modelado matemático y técnicas de simulación computacional para el estudio de sistemas biológicos, de comportamiento y sociales.* Otra definición del Instituto Nacional de Investigación del Genoma Humano (NHGRI, por sus siglas en inglés *National Human Genome Research Institute*) es *la rama de la biología que se ocupa de la adquisición, almacenamiento, visualización y análisis de la información encontrada en los datos de secuencias de ácidos nucleicos y proteínas.*

Russ Altman (1998), Altman y Dugan (2003) proveen dos definiciones de bioinformática. La primera implica el flujo de información siguiendo el dogma central de la biología molecular. La segunda definición implica el flujo de información que se transfiere basándose en métodos científicos. En esta definición se incluyen problemas tales como diseño, validación y distribución de software; almacenamiento y distribución de datos; implementación de trabajos de investigación reproducibles; así como la interpretación de experimentos.

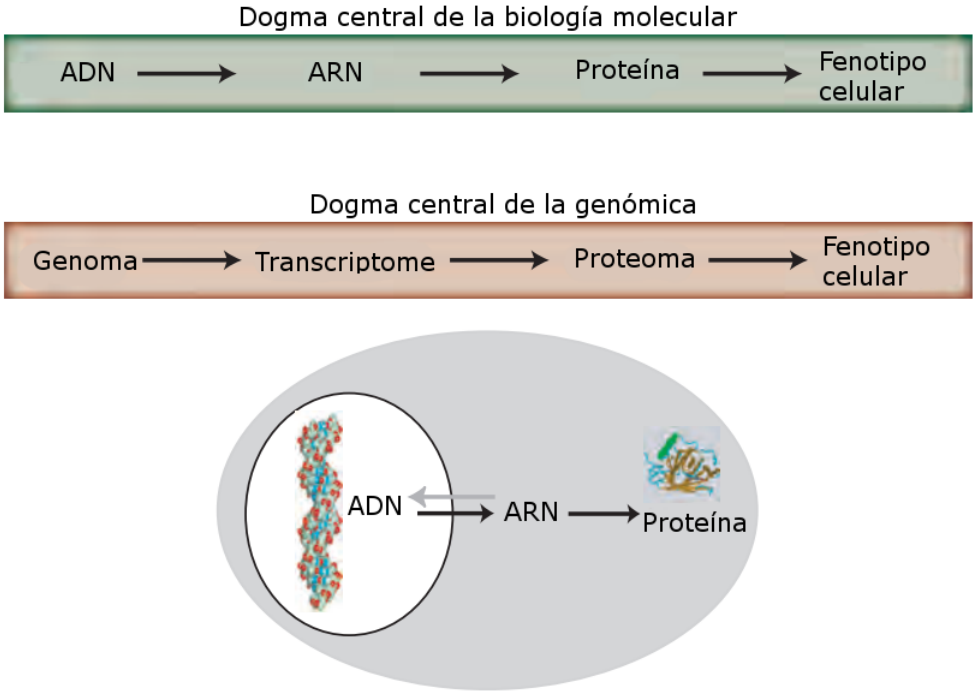



Figura 1.1: Dogmas de la bioinformática [Pevsner, 2015]

En la figura 1.1 se resumen las áreas de bioinformática y genómica desde tres perspectivas. La primera perspectiva de la bioinformática es la célula. Este se enfoca en la colección de ADN (genoma), el ácido ribonucleico (ARN, transcriptoma) y la secuencia de proteínas (proteoma). El enfoque bioinformático a los datos de secuencias moleculares implica la aplicación de algoritmos informáticos y bases de datos a la biología molecular y celular, a esto se le conoce frecuentemente como genómica funcional. Este enfoque puede abordarse desde niveles que van desde genes y proteínas individuales hasta rutas y redes celulares e incluso respuestas genómicas completas.

La segunda perspectiva de la bioinformática se aborda desde el enfoque celular, gracias a lo cual es posible centrarse en organismos individuales. Cada organismo cambia a través de diferentes etapas de desarrollo y a través de diferentes regiones del cuerpo (en caso de organismos multicelulares). Ampliando nuestra visión desde el nivel de la célula al organismo, podemos considerar el genoma del individuo (colección completa de genes), incluyendo los genes que se expresan como transcripciones de ácido ribonucleico (ARN) y los productos de proteína.

Hoy en día existen millones de especies vivas, se han clasificado en tres ramas principales; bacterias, arqueas y eukaryotas. Las bases de datos de secuencias moleculares contienen actualmente una secuencia de ADN de 300,000 especies diferentes. Miles de genomas de diferentes organismos se encuentran completos. Una de las principales lecciones que estamos aprendiendo es sobre la unidad fundamental de la vida a nivel molecular. También estamos llegando a apreciar el poder de la genómica comparativa, en la que se comparan los genomas. A través del análisis de secuencias de ADN estamos aprendiendo cómo evolucionan los cromosomas y son esculpidos a través de procesos como duplicaciones, deleciones y reordenamientos cromosómicos, y a través de duplicaciones de todo el genoma.

## 1.2. Plataformas de secuenciación

El incremento de potencia y la reducción de costos en las tecnologías de secuenciación de próxima generación (NGS, por sus siglas en inglés *Next Generation Sequencing*) han generado una creciente gama de aplicaciones. Este tipo de tecnología es relativamente reciente, por lo que sus costos originalmente mucho más elevados se han reducido, permitiendo utilizarse como el método predilecto para realizar estudios de asociación del genoma completo (en inglés, GWAS, Genome-Wide Association Study) utilizando arreglos de polimorfismo de nucleótido único (método utilizado para detectar polimorfismos dentro de una población, dentro de las aplicaciones más importantes son para determinar la susceptibilidad a enfermedades y para medir la eficacia de fármacos diseñados específicamente para un individuo), dado que no podemos permitirnos realizar la secuenciación de todo el genoma en miles de individuos. Este hecho está cambiando rápidamente y la secuenciación se convierte en nuestro microscopio molecular, una herramienta para obtener una primera mirada. Aunque la replicación, la transcripción, la traducción, la metilación y el plegamiento del ADN nuclear son procesos completamente diferentes, todos ellos pueden ser estudiados mediante secuenciación.

Una ventaja importante de los datos de secuencias es su calidad, robustez y bajo nivel de ruido. Cabe señalar que un proyecto exitoso de NGS requiere experiencia tanto en el laboratorio húmedo como en el de bioinformática para garantizar datos de alta calidad y su interpretación. La secuencia en sí misma es una dura prueba de su corrección. Un sistema de secuenciación no producirá secuencias *aleatorias* y cuando esto se hace evidente inmediatamente a partir de las llamadas de control de calidad obtenidas. Además, las secuencias aleatorias no tendrán coincidencia y pueden descartarse fácilmente durante el análisis de los datos y cuando su número excede un determinado umbral, se vuelve evidente que existe un problema serio en

algún punto del estudio.

### 1.2.1. Tecnologías de secuenciación actuales

Las diferentes compañías de las plataformas de secuenciación han ideado diferentes estrategias para preparar las bibliotecas de secuencias en plantillas adecuadas, así como para detectar la señal y finalmente leer la secuencia de ADN. Para los sistemas Illumina, Solid, PGM y 454 se requiere una amplificación para aumentar la relación señal a ruido porque los sistemas no son lo suficientemente sensibles como para detectar la extensión de una base a nivel de la molécula individual de la plantilla de ADN. Por otro lado, los sistemas Heliscope y PacBio SMRT no necesitan etapas de preamplificación ya que estos sistemas son lo suficientemente sensibles como para detectar ampliaciones de plantilla individuales para una sola molécula. Las diferentes estrategias para generar las lecturas de secuencias también conducen a diferencias en la capacidad de salida para las diferentes plataformas, eso se muestra en la tabla 1.1.

| Fabricante | Secuenciador | Secuencia por | Detección | Tipos de ejecución | Tiempo de ejecución | Longitud de lectura (bp) | #reads por ejecución | Salida por ejecución | Observaciones |
|---|---|---|---|---|---|---|---|---|---|
| Roche | GS FLX Titanium XL + | Detección de síntesis y pirofosfato | Extremo único | 23 h | 700 | 1 milliones | 700 Mb | | |
| | GS Junior System | Síntesis | Detección de Pirofosfato | Extremo único | 10 h | 400 | 0.1 millones | 40 Mb | |
| LifeTechnologies | Ion torrent | Síntesis | Proton release | Extremo único | 4 h | 200-400 | 4 millones | 1.5-2 Gb | Ion318 Chip |
| | Proton | Síntesis | Liberación de protones | Extremo único | 4 h | 125 | 60-80 millones | 8-10 Gb | IonP1 chip |
| | Abi / solid | Ligation | Detección de fluorescencia de sondas de di-base | Extremo único y emparejado | 10 días | $75 + 35$ | 2.7 billones | 300 Gb | |
| Illumina / solexa | HiSeq2000 / 2500 | Síntesis | Fluorescencia; Terminadores reversibles | Extremo único y emparejado | 12 days | $2 \times 100$ | 3 billones | 600 Gb | modo de alto rendimiento |
| | MiSeq | Síntesis | Fluorescencia; Terminadores reversibles | Extremo único y emparejado | 65 h | $2 \times 300$ | 25 millones | 15 Gb | |
| Pacific biosciences | RSII | Single molecule synthesis | Fluorescence; Terminalmente fosfolinados | Extremo único | 2 días | 50 % de lecturas $>$ 10 kb | 0,8 millones | 5 Gb | 16 células SMRT |
| Helicos | Heliscope | Síntesis de moléculas individuales | Fluorescencia; Terminador virtual | Extremo único | 10 días | $\sim 30$ | 500 millones | 15 Gb | Dos flowcells en paralelo |

Tabla 1.1: Salida por plataforma tecnológica de secuenciación

### 1.2.2. Tecnologías de secuenciación futuras

Los avances realizados en la tecnología de secuenciación en los últimos años han sido impresionantes. Sin embargo, la última plataforma de secuenciación funcionaría en moléculas de ADN o ARN sin ninguna (pre) amplificación, sin uso de pasos ópticos, lecturas de Mb a Gb de longitud, sin sesgo de Guanina-Citosina (CG), alta precisión de lectura y sería lo suficientemente flexible como para generar tantas lecturas de secuencias como sean necesarias para determinada investigación. Además, debe ser no costoso tanto para adquirir y ejecutar, fácil de operar, contar con tiem-

pos cortos de ejecución y con sencillos pasos de biblioteca de preparación o ninguna. Sobra decir que una plataforma de secuenciación con estas características no existe todavía.

Secuenciación de nanopart de Oxford: Las características esperadas para los secuenciadores de nanopore son: molécula única, sin amplificación, detección de bases sin etiquetas, lecturas largas, bajo sesgo de GC y escalabilidad en los datos de salida. El principio básico detrás de la tecnología es el túnel de moléculas (polímero) a través de un poro que separa dos compartimientos. La presencia física de la molécula que pasa a través del poro provoca un cambio temporal característico en el potencial entre los dos compartimentos que permite la identificación de la molécula específica. Dos versiones de la secuenciación de nanopore ADN se encuentran en desarrollo, es decir, utilizando el poro natural forma la proteína alfa-hemolisina o poros de estado sólido fabricados. Oxford nanopore technologies (ONT) es una de las empresas que trabajan en la construcción de dispositivos de secuenciación de nano poros. Aunque se ha enfocado ONT en la secuenciación de ácidos nucleicos, esta tecnología en principio podría aplicarse a cualquier (bio) polímero, siempre que las moléculas produzcan cambios distinguibles en la corriente entre los compartimentos.

### 1.2.3. Aplicaciones de las tecnologías de secuenciación NGS

Cada aplicación requiere una cantidad mínima de lecturas de una longitud predefinida para obtener un conjunto de datos que permita obtener conclusiones confiables. Las especificaciones difieren para cada aplicación. Sin embargo, dependiendo de la pregunta de investigación específica, el número de lecturas y la longitud de lectura pueden ser diferentes.

**Secuenciación del genoma De novo**

Una aplicación principal de la tecnología NGS es la caracterización completa del genoma completo de una especie en particular. Después de abordar primero los organismos modelo principales (levadura, Escherichia coli, Drosophila, Arabidopsis, ratón) y el genoma humano, se está realizando un número cada vez mayor de secuencias del genoma. También se han invertido considerables esfuerzos en el análisis de genomas de especies amenazadas o incluso extintas. Juntos estos proyectos producen una enorme cantidad de datos que, además de algunos estudios evolutivos, siguen teniendo gran parte sin analizar.

## 1.3. Ensambladores de secuencias de ADN

El método tradicional para secuenciar genomas consistió en:

1. Tomar pequeñas muestras de sangre que contiene millones de células con ADN idéntico.
2. Separar el ADN en fragmentos basándose en métodos bioquímicos.
3. Por último, se secuenciaron los fragmentos para producir lecturas.

La dificultad de este método se incrementa debido a que los investigadores no conocen la procedencia de las lecturas, por lo que se deben hallar los empalmes en las lecturas para reconstruir el genoma.

Aunque se han secuenciado muchos genomas, un genoma gigantesco como el de Amoeba dubia (se trata de una ameba, caracterizada por tener el genoma más grande conocido) se mantiene más allá de las capacidades de los secuenciadores modernos. De forma estricta, los biólogos pueden generar fácilmente suficientes lecturas

para analizar un genoma grande, sin embargo, el ensamble de estas lecturas sigue presentando el mayor reto computacional.

Algunas complicaciones prácticas que incrementan la dificultad de ensamblar un genoma se encuentran directamente relacionadas con la adquisición de las lecturas, el ADN es una doble hebra, de modo que no tenemos manera de identificar a priori el origen de la hebra dada una lectura, esto implica que no podemos saber cuando utilizar una lectura o su complemento inverso en la tarea de ensamble. De igual manera, los secuenciadores modernos pueden producir errores, de tal modo que las lecturas generadas regularmente contendrán errores. Estos detalles complican el ensamble de los genomas porque dificultan la correcta identificación de los empalmes. Otro factor que debe ser tomado en cuenta es que algunas regiones del genoma pueden no estar cubiertas por ninguna lectura, imposibilitando la reconstrucción completa del genoma.

Podemos resumir el proceso de ensamble en los siguientes pasos:

- Obtener secuencias cortas leídas a partir de clones de ADN.
- Para realizar el ensamble los fragmentos leídos se deben unir entre sí eliminando las superposiciones, de modo de obtener fragmentos de mayor longitud.
- Las secuencias más largas se conocen como contigs.
- Los contigs pueden ser superpuestos entre sí para formar supercontigs, estos son orientados a lo largo de un mapa físico de un cromosoma en una única dirección.
- Los supercontigs también se pueden superponer para crear el mapa final de alta resolución del genoma.

## 1.4. Field Programmable Gate Array's

Field Programmable Gate Arrays (FPGAs) son circuitos integrados con capacidad para programar lógica digital en el campo, estos fueron creados en la década de 1980. Originalmente se crearon para permitir el diseño de lógica personalizada. Los FPGAs han evolucionado de un dispositivo útil con una interfaz humilde a un sistema de circuito integrado con su propio microprocesador, con bloques de memoria e interfaces.
A diferencia de otras formas utilizadas en la construcción de hardware. Los FPGAs tienen características únicas. Ya que permiten la construcción exacta del hardware requerido en lugar de utilizar el mismo producto estándar para una aplicación específica o *ASSP* por sus siglas en inglés *Application Specific Standard Product*, al igual que asumir el riesgo, tiempo y costo de utilizar un Circuito Integrado para Aplicaciones Específicas (o ASIC, por sus siglas en inglés) para realizar un diseño.

La capacidad de personalizar el FPGA significa que, de manera frecuente, en un FPGA, es posible realizar operaciones de manera simple, rápida y con mejor eficiencia energética que un microprocesador o una ASSP.
Un FPGA permite programar características y funciones para adaptar estándares y reconfigurar hardware a aplicaciones específicas incluso después de haber realizado una implementación.

Actualmente, los FPGAs contienen mezclas configurables de memorias estáticas de acceso aleatorio (SRAM o Flash), pines de alta velocidad de entrada y salida, bloques lógicos y enrutamiento. De forma más específica, un FPGA contiene elementos lógicos programables llamados elementos lógicos (LEs), ordenados de forma jerárquica en interconexiones reconfigurables que permitan a los LEs estar físicamente conectados entre sí. De esta manera se puede configurar los LEs para realizar funciones complejas o simplemente formar compuertas lógicas básicas, como AND y

OR. La mayoría de los FPGAs también contienen bloques de memoria.

## 1.5. Descripción

Un genoma es la colección completa de ácido desoxirribonucleico (ADN) de un organismo, el cual describe sus características y lo hace único. El ADN se mantiene unido mediante un puente de hidrógeno; esto forma una relación complementaria (conocida como ley de Chargaff) entre las bases nitrogenada; adenina ($A$)-timina($T$), citosina ($C$)-guanina ($G$) (también conocidos como nucleótidos). En otras palabras, la cantidad de adenina es igual a la de timina, de igual forma la cantidad de guanina es igual a la de citosina, $A + G = C + T$.

Con base en la ley de Chargaff, el genoma de un organismo puede ser expresado utilizando una sola hebra de la doble hélice y describirlo computacionalmente como una cadena de caracteres formada por un alfabeto de cuatro símbolos $\Sigma = \{A, C, G, T\}$ representando los cuatro nucleótidos. La secuenciación del genoma de un organismo es el proceso seguido para obtener su representación digital. Durante este proceso se debe aislar el ADN y leerlo mediante una máquina de secuenciación para obtener cadenas de texto y posteriormente procesarlo computacionalmente. Las máquinas de secuenciación están limitadas a leer una pequeña cantidad de pares de bases (comprendiendo un rango de cientos de bases nitrogenadas); estas lecturas son conocidas como *short-reads*, las cuales son leídas de forma aleatoria, por tal motivo deberán ser ensambladas posteriormente. A este método se le conoce como $shotgun-sequencing$.

## 1.6. Problemática

Dentro del área de bioinformática, el ensamble de secuencias se ha resuelto durante mucho tiempo. Al inicio, el estudio de ADN requiere una cantidad conside-

rablemente alta de recursos humanos y tecnológicos para llevarse acabo. Un ejemplo de esto es el *Human Genome Project* donde se halló gran parte del genoma humano. El tiempo requerido por este proyecto fue de 13 años con un costo de tres mil millones de dólares. Gran parte del tiempo se utilizó para ensamblar las secuencias de ADN, extraídas de los voluntarios. De esta forma, tecnologías y algoritmos de secuenciación se han utilizado para realizar ensambles en menos tiempo y costo, de cualquier manera, queda mucho trabajo pendiente en ambas vertientes. Es posible dividir la tarea de ensamble de secuencias de ADN en tres subproblemas de acuerdo al enfoque *Overlap, Layout y Consensus* (OLC)

Las etapas de *overlap* y *layout* realizan varias operaciones que crecen exponencialmente dada la entrada, por lo cual, la tarea del diseño de una arquitectura hardware que sea capaz de llevar acabo técnicas nuevas y apropiadas para ser implementadas de manera física tiene una marcada importancia.

## 1.7. Justificación

La creciente capacidad computacional y el diseño de nuevos algoritmos permiten crear soluciones que agilizan la obtención de resultados en diversas ramas de la bioinformática. En particular, la tarea de ensambles *de novo* es necesaria como una primera aproximación para el análisis de los genomas, sin embargo, sigue siendo costosa en recursos humanos y tecnológicos. Nuevos algoritmos y tecnologías hardware se desarrollan día a día para disminuir el tiempo requerido en diversos problemas de la bioinformática. De esta manera es posible implementar vía hardware las diversas soluciones tecnológicas reportadas hasta la fecha para crear una arquitectura que nos permita realizar la tarea de ensambles utilizando menos tiempo del requerido por las soluciones actuales.

## 1.8. Objetivo general

Diseñar e implementar una arquitectura hardware para acelerar el ensamble de secuencias de ADN con el objetivo de reducir el tiempo de procesamiento requerido en comparación con las implementaciones software.

## 1.9. Objetivos específicos

- Identificar y seleccionar algoritmos de emparejamiento adecuados para su implementación hardware en las etapas de ensamble.
- Explorar el espacio de diseño arquitectural para cada algoritmo de emparejamiento.
- Evaluar la arquitectura propuesta en términos de recursos de procesamiento en hardware y software.

Para alcanzar el objetivo general, los algoritmos seleccionados deben ser adaptados para cada etapa de ensamble al igual que para el diseño de la arquitectura hardware, así como la verificación de los resultados finales.

## 1.10. Metodología

Dentro de la metodología se ha escogido utilizar el enfoque utilizado por los trabajos [Simpson, 2010, Simpson, 2011]. Esto nos permite generar un grafo de ensamblaje, que aunque no sean los más rápidos para encontrar los primeros resultados, esta metodología puede crear un índice que acelere la referencia futura. Utiliza el mismo grafo para realizar consultas con k-mers (todas las posibles subcadenas de

longitud $k$ de una cadena) de diferentes longitudes sin tener que realizar una reconstrucción completa del grafo, en contraste con técnicas similares. Este método también tiene la ventaja de usar menos memoria que otras herramientas como SOAP2 o VELVET. El uso de datos procesados como *Burrows-Wheeler Transform*[Simpson, 2010] no requiere tiempo extra para revertir la transformación, esto es útil cuando se analiza un genoma previamente comprimido.

Además, utilizamos un algoritmo para fusionar los conjuntos de datos (cadenas de lectura corta en la transformación de Burrows-Wheeler BWT), Holt, McMillan y otros, [Holt, 2014] tratan el problema de combinar múltiples cadenas BWT. Desde su enfoque, es posible identificar el origen de cuerdas individuales. El objetivo de su trabajo es agregar nueva información a un conjunto de datos existente combinando diferentes conjuntos de datos y mejorando los coeficientes de compresión [Holt, 2014]; esto reduce el tiempo requerido para consultar conjuntos de datos múltiples. Esta técnica mejorará la construcción del grafo que se utilizará para ensamblar el genoma. Este trabajo utiliza el índice FM para construir el grafo de cadena, y esto puede ser más rápido, utilizando el trabajo propuesto por Chacón, Moure y otros, [Chacon et al., 2013] es posible reducir el número de operaciones necesarias para coincidir con un patrón en las lecturas.

## 1.11. Estructura del documento

En el siguiente capítulo se describen brevemente los fundamentos teóricos para entender el contenido de este trabajo, iniciando con la transformación Burrows-Wheeler y la estructura utilizada para realizar búsquedas; conocida como índice FM. El capítulo concluye con la descripción de los tipos grafos utilizados para realizar los ensambles y el funcionamiento de las arquitecturas hardware implementadas en el campo de la bioinformática molecular.

En el capítulo 3 se realiza una proyección del estado del arte y los métodos empleados para tratar el problema de ensamble de secuencias de ADN. Se revisan los enfoques de hardware y algorítmicos en el área. En este apartado se revisan trabajos que incluyen la compresión de datos y la codificación de nucleótidos y secuencias de control. Este finaliza con un resumen y ubica el trabajo en el área.

Dentro el capítulo 4 se describe el ensamblador de manera general y especifíca de forma algorítmica para cada una de sus etapas así como su análisis de complejidad, también se describe la forma en la que se adaptó la arquitectura en cada una de las etapas.

El capítulo 5 presenta el diseño experimental utilizado y los resultados de la implementación de la arquitectura hardware en cada una de las tres etapas en las que se divide el ensamblador. Finaliza con los resultados obtenidos de la salida del ensamblador en comparación con los resultados esperados, así como la comparación con resultados de otras arquitecturas.

Por último, el capítulo 6 presenta un resumen del trabajo resaltando las contribuciones realizadas en cada etapa del ensamblador. Posteriormente se revisan los resultados y se comparan con los objetivos planteados. El trabajo concluye ubicando este trabajo en el área de investigación así como el planteamiento del trabajo futuro.

# Capítulo 2

# Marco teórico / Fundamentos

## 2.1. Transformación Burrows-Wheeler

La transformación Burrows-Wheeler (BWT por sus siglas en inglés) es uno de los mejores métodos para compresión sin pérdida en texto, que ha sido utilizado más allá de su propósito original. También conocida como *ordenamiento por bloque*, debido a que toma un bloque del texto y lo permuta. Una de las desventajas principales del enfoque es que no puede procesar el texto un símbolo a la vez, este debe ser leído en bloques y procesado. En muchas aplicaciones, esta no es una limitante, pero existen aplicaciones que requieren el procesar datos al vuelo, como son recibidos. Otro punto a su favor es debido a que el texto puede ser ordenado, muchas implementaciones de este algoritmo utilizan la codificación ASCII, por lo que las comparaciones son triviales.

Esta transformación ha sido utilizada para muchas más aplicaciones que la compresión, dado que implícitamente contiene un índice del texto de entrada. Para este trabajo en particular se retoma el uso para tareas de búsqueda de patrones e índices de texto completo, que conduce a aplicaciones que van desde la bioinformática hasta la traducción automática. Se presenta la transformación Burrows-Wheeler para

codificar una cadena $T$ de $n$ caracteres de longitud $T[1...n]$, sobre un alfabeto $\Sigma$ de $|\Sigma|$ caracteres.

| # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | m | i | s | s | i | s | s | i | p | p | i | $ |
| 2 | i | s | s | i | s | s | i | p | p | i | $ | m |
| 3 | s | s | i | s | s | i | p | p | i | $ | m | i |
| 4 | s | i | s | s | i | p | p | i | $ | m | i | s |
| 5 | i | s | s | i | p | p | i | $ | m | i | s | s |
| 6 | s | s | i | p | p | i | $ | m | i | s | s | i |
| 7 | s | i | p | p | i | $ | m | i | s | s | i | s |
| 8 | i | p | p | i | $ | m | i | s | s | i | s | s |
| 9 | p | p | i | $ | m | i | s | s | i | s | s | i |
| 10 | p | i | $ | m | i | s | s | i | s | s | i | p |
| 11 | i | $ | m | i | s | s | i | s | s | i | p | p |
| 12 | $ | m | i | s | s | i | s | s | i | p | p | i |

Rotaciones

| # | F | | L |
|---|---|---|---|
| 12 | $ | mississipp | i |
| 11 | i | $mississip | p |
| 8 | i | ppi$missis | s |
| 5 | i | ssippi$mis | s |
| 2 | i | ssissippi$ | m |
| 1 | m | ississippi | $ |
| 10 | p | i$mississi | p |
| 9 | p | pi$mississ | i |
| 7 | s | ippi$missi | s |
| 4 | s | issippi$mi | s |
| 6 | s | sippi$miss | i |
| 3 | s | sissippi$m | i |

Rotaciones ordenadas

Tabla 2.1: Ejemplo de la transformada Burrows-Wheeler para la cadena de texto *mississippi*

Podemos crear un arreglo $R[1...n]$ de referencias a las cadenas rotadas del texto de entrada $T$. Inicialmente $R[i]$ enumera a cada $i$ desde 1 hasta $n$, para representar la lista desordenada. Para ordenar se utiliza la subcadena iniciando en $T[R[i]]$ como clave de comparación. El ejemplo en la tabla 2.1 es el resultado del ordenamiento. La posición 2 es la primera cadena rotada en orden lexicográfico ($ississ...$), seguido por la posición 3 ($ssiss...$), la posición 6 ($ssippi...$). De este modo el arreglo de referencias es $R = [2, 3, 6, 8, 12, 1, 4, 5, 7, 10, 8, 11]$.

El arreglo $R$ es un índice de carácteres en T, correspondiente a la primera columna del arreglo de rotaciones ordenado, comúnmente conocido como $F$ en la

literatura de BWT. La última columna del arreglo de rotaciones ordenado, se conoce como $L$, es la salida de la BWT y se lee como $T[R[i] - 1]$, donde $i$ varía de 1 a $n$. De este modo, para el ejemplo presentado, la salida de la transformación es $L = miisi\$ssspsp$. De este modo, se necesita almacenar un índice $a$ para poder recuperar la cadena original, este índice corresponde a la posición en $L$ del último carácter de la cadena de texto original. En este caso $a = 1$, en este sentido, podemos agregar un símbolo a la cadena de entrada para marcar la cadena original respecto a las demás cadenas rotadas, comúnmente se utiliza el símbolo \$ como terminal.

En la descripción, la transformación se calcula utilizando el espacio $O(n)$ para $R$. El tiempo utilizado es $O(n)$ para la creación del arreglo $R$, más el tiempo necesario para ordenar. Tomando en cuenta el tiempo medio $O(n \log_n)$ como el que utiliza el algoritmo *quicksort*.

La transformación es simple, las subcadenas nunca son creadas, solo se almacenan referencias a las posiciones en la cadena original. El tamaño del texto transformado es idéntico al original y contiene exactamente los mismos caracteres pero en otro orden. A simple vista pareciera que no se ha logrado nada, sin embargo, se han agrupado caracteres similares, esto es, caracteres que pertenecen a la misma subcadena. Pareciera increíble, que, la transformación inversa existiese, a pesar de eso, existe y es posible implementarla de forma eficiente.

Distinguimos entre una transformación hacia adelante, la cual produce la cadena para ser comprimida, y la transformación hacía atrás retorna la cadena de texto original desde la transformada. La BWT hacia adelante consiste de tres pasos básicos:

- Agregar al final de $T$ un símbolo especial, menor que cualquier otro en el alfabeto de $T$.

- Desde una matriz conceptual $M$, con filas que representan rotaciones cíclicas de la cadena $T$ ordenadas lexicográficamente.

- Construir la transformada $L$ tomando la última columna de $M$.

Nótese en la tabla 2.1, que cada columna de $M$ es una permutan de la última columna L, y en particular la primera columna de $M$, llamada $F$, se obtiene de ordenar lexicográficamente los caracteres en $L$. Existe una fuerte relación entre la matriz $M$ y el arreglo de sufijos $SA$ de la cadena $T$. Cuándo se ordena las filas de la matriz $M$, esencialmente se ordenan los sufijos de $T$. Por consiguiente, la entrada $SA[i]$ apunta al sufijo de $T$ ocupando (un prefijo de) la i-ésima fila de $M$.

La rotación cíclica de las filas de $M$ es crucial para definir el BWT hacía atrás, la cual se basa en dos observaciones fáciles de demostrar.

- Dada la i-ésima fila de $M$, su último carácter $L[i]$ precede a su primer carácter $F[i]$ en el texto original $T$, es decir, $T = ...L[i]F[i]....$

- Sea $L[i] = c$ y sea $r_i$ el rango de la fila $M[i]$ entre todas las filas terminando con el carácter $c$. Toma la fila $M[j]$ como la $r[i]$ fila de $M$ iniciando con $c$. Entonces, el carácter correspondiente a $L[i]$ en la primera columna $F$ está ubicada en $F[j]$ (a esto se le conoce como mapeo $LF$, donde $LF[i] = j$).

La transformación de Burrows-Wheeler fue un gran avance dado que proporciona una transformación reversible para el texto que lo hace mucho más fácil de comprimir. Hay muchas otras transformaciones reversibles que se podrían aplicar a un texto, por ejemplo, podrían almacenar los caracteres al revés, o las dos primeras letras después de que cada espacio puede ser transpuesto, pero estos no nos ayudan a comprimir el texto. El poder de la BWT es que reúne caracteres relacionados, de la misma manera que un transformación Fourier separa hacia fuera componentes de alta frecuencia y de baja frecuencia.

La compresión y la búsqueda de patrones se encuentra relacionada. Una manera de percibir un método de compresión es que simplemente busca patrones y se aprovecha de ellos para quitar la repetición. En los métodos de compresión, el patrón es el contexto utilizado a buscar, para hacer predicciones basadas en lo que ha sucedido en ocurrencias anteriores del contexto. Para la BWT el proceso de búsqueda es mucho más simple, porque la codificación está basada en el ordenamiento lexicográfico de cada subcadena del texto, de las cuales tenemos una lista ordenada, disponible como producto de la transformación. Todas estas subcadenas no son generadas, simplemente son una lista de referencias a posiciones en el texto original. La columna $L$ (la cual se muestra como salida de la BWT) es realmente sólo el carácter de la cadena original que precede al existente en la columna $F$.
De esta forma, la búsqueda basada en la transformada Burrows-Wheeler es fácil, ya que utiliza una arreglo auxiliar que es necesario para la decodificación, las filas en la lista pueden ser accedidas de manera aleatoria y los caracteres en cada fila son leídos de forma simple en tiempo lineal. Todo esto es posible sin necesidad de decodificar el texto.

## 2.2. FM-Index

El índice-FM es una estructura de datos comprimidos diseñado para la búsqueda de patrones en textos completos. El componente clave de prácticamente cualquiera de las múltiples variantes de este índice es la operación *rank*, que generalmente se realiza en un vector de bits $B$, que, para un número entero dado $j$, devuelve el número de bits determinados en prefijo de $B$ de longitud $j$. Se construye el índice de un texto $T[1...n]$ sobre un alfabeto $\Sigma = 1, ..., \sigma$. El índice se consulta con los patrones de la forma $P[1..m]$. La operación de fila se calculará para el vector de bits $B[1...n]$.

El índice de FM es básicamente el resultado de la transformación de Burrows–Wheeler de texto $T$, denotado como $T_{BWT}$, con dos estructuras de ayudante:

un arreglo de cuenta $C[1...\sigma]$ tal que $C[i] = |T[j] : T[j] \leq i \ y \ 1 \leq j \leq n|$, y un dato de estructura contestador $Occ(T_{BWT}, c, pos) = |T_{BWT}[j] : T_{BWT}[j] = c \ y \ 1 \leq j \leq pos|$ consultas. Esto permite contar las ocurrencias de un patrón P, encontrar los rangos del arreglo de sufijos (SA) de $T$ iniciando con sufijos sucesivos de $P$ en las iteraciones sucesivas del bucle. Si la función $Occ$ se realiza con un árbol wavelet, el conteo de consultas se ejecuta en $O(mlog_\sigma)$ en el peor de los casos. Existen muchas variantes de índice FM, con diferentes relaciones de complejidad espacio-temporales y diferentes desempeños prácticos.
En el año 2000, Paolo Ferragina y Giovanni Manzini publicaron un artículo describiendo como la BWT con algunas estructuras de datos pequeñas pueden ser utilizadas como un índice de un texto eficiente en espacio. Se le nombró índice-FM, porque al igual que el mapeo LF fue la llave para entender cómo la BWT es reversible, también es la llave para entender cómo este puede ser utilizado como índice.

Sea $T[1, u]$ un texto sobre un alfabeto $\Sigma$, y $Z = BW(T)$ la transformación BWT. Se desea encontrar un patrón $P[1, p]$ en $Z$, donde se reporten todas las ocurrencias de $P$ en el texto sin comprimir y sin tener que revertir la transformación BWT en $Z$. El algoritmo hace uso de la relación entre arreglo de sufijos $SA$ y la matriz $M$. Recordando que el arreglo $SA$ posee dos grandes propiedades estructurales que son explotadas para soportar búsquedas rápidas de patrones.

- Todos los sufijos del texto T[1,u] que sean prefijos de un patrón $P[1, p]$ utilizan una porción contigua (subarreglo) de $SA$.

- El subarreglo tiene como posición inicial $sp$ y posición terminal $ep$, donde $sp$ es la posición lexicográfica de la cadena $P$ entre la secuencia ordenada de los sufijos del texto.

### 2.2.1. Paso I: Conteo de ocurrencias

El paso inicial del algoritmo es identificar la posición de *sp* y *ep*, mediante el acceso a la cadena $Z$ y a estructuras de datos auxiliares.

**Algorithm 1** Búsqueda BWT

```
1:  procedure BUSQUEDABWT(P[1,p])
2:      c ← P[p]
3:      i ← p
4:      sp ← C[c]+1
5:      ep ← C[c]+1
6:  top:
7:      while sp <= ep AND i ≥ 2 do
8:          c ← P[i − 1]
9:          sp ← C[c] + Occc, 1, sp − 1 + 1
10:         ep ← C[c] + Occ(c, 1, ep)
11:         i ← i-1
12:     if i ≥ stringlen then return false
```

El algoritmo 1 consiste en $p$ fases, cada una preserva la invariante: En la i-ésima fase, el parámetro *sp* apunta a la primera fila de $M$, que es prefijo de $P[i, p]$ y del parámetro *ep* apunta a la última fila de $M$, como prefijo de P[i,p]. El pseudocódigo 1. Se inicia con $i = p$, para que *sp* y *ep* sean determinados por el arreglo $C$, definido como el histograma acumulado de los símbolos del texto $T$. Si $ep \leq sp$ , entonces se puede implementar el mapeo LF. Después de la fase final, *sp* y *ep* son delimitadores de la porción de $M$ (y por consiguiente, del arreglo de sufijos $SA$) que contienen todos los textos de sufijos prefijo de P. El entero $(ep - sp + 1)$ cuenta el número de ocurrencias de $P$ en $T$.

El tiempo de ejecución del algoritmo 1 depende de el costo de la función *Occ*, el número de ocurrencias de un patrón $P[1, p]$ en $T[1, u]$ puede ser calculado en $O(p)$.

### 2.2.2. Paso II: Ubicación de ocurrencias

Ahora se considera el problema de determinar la posición exacta en el texto $T[1,u]$ de todas las ocurrencias de un patrón $P[1,p]$. Esto quiere decir que para $s = sp, sp+1, ..., ep$, se necesita hallar la posición $pos(s)$ en $T$ del sufijo cuyo prefijo es la s-ésima fila de $M$, es decir $M[s]$. Se utiliza el mapeo LF para hallar la fila $s$ correspondiente al sufijo $T[pos(s)-1,u]$. Se itera en esta función $v$ veces, hasta que $s$ apunte a una fila marcada, entonces se calcula la $pos(s) = pos(s') + v$. El punto crucial del algoritmo es el marcado lógico de las filas de $M$ correspondiente a los sufijos del texto, iniciando en posiciones $1 + in, i = 0, ..., u/n$. Esta solución consiste en almacenar el número de filas en un esquema de filas de dos niveles. Se particionan las filas de M en filas de tamaño $\Theta(\log^2{}_u)$ cada una. Para cada fila, se toman las filas marcadas y se almacenan en un árbol B utilizando como llave la distancia desde el inicio de la fila. Ya que la fila contiene a lo más $O(\log^2{}_u)$ llaves, cada consulta de $O(\log\log_u)$ bits de longitud, requiere $O(1)$ tiempo en la RAM.

El espacio total requerido por el marcado lógico es $(O(u/n)\log\log_u)$ bits. Adicionalmente, para cada fila marcada se almacena la posición inicial del sufijo correspondiente en $T$, el cual requiere $O(\log_u)$ bits de espacio adicional por fila marcada. Consecutivamente, el espacio total ocupado es $O((u/n)\log_u) = O(u/\log_u)$ bits. Respecto a la complejidad temporal, el algoritmo calcula $pos(s)$ en máximo $n = \Theta(\log^2{}_u)$ pasos. Por lo tanto, las ocurrencias de un patrón $P$ en $T$ retornan un tiempo acotado por $O(Occ\ \log^2{}_p)$, con un espacio por encima de $O(u/\log_u)$ bits.

## 2.3. Grafos de ensamble de secuencias de ADN

Para solucionar el problema de la reconstrucción de cadenas tenemos que buscar por un camino en el grafo de empalmes que visite cada nodo exactamente una

vez. Un camino en un grafo que visita cada nodo una vez es conocido como camino Hamiltoniano, un grafo puede tener más de un camino Hamiltoniano.

Para diseñar un algoritmo eficiente se necesita conocer el trabajo de Nicolas de Bruijn. Quien se interesó en resolver un problema puramente teórico, descrito de la siguiente manera.

- Una cadena binaria es una cadena compuesta únicamente de 0's y 1's; una cadena binaria es $k$-universal si contiene cada $k-mer$ binario exactamente una vez. Por ejemplo, 0001110100 es una cadena $3-universal$, ya que contiene cada uno de los ocho $3-mers$ binarios $(000, 001, 011, 111, 110, 101, 010$ y $100)$ exactamente una vez, ver figura 2.1.

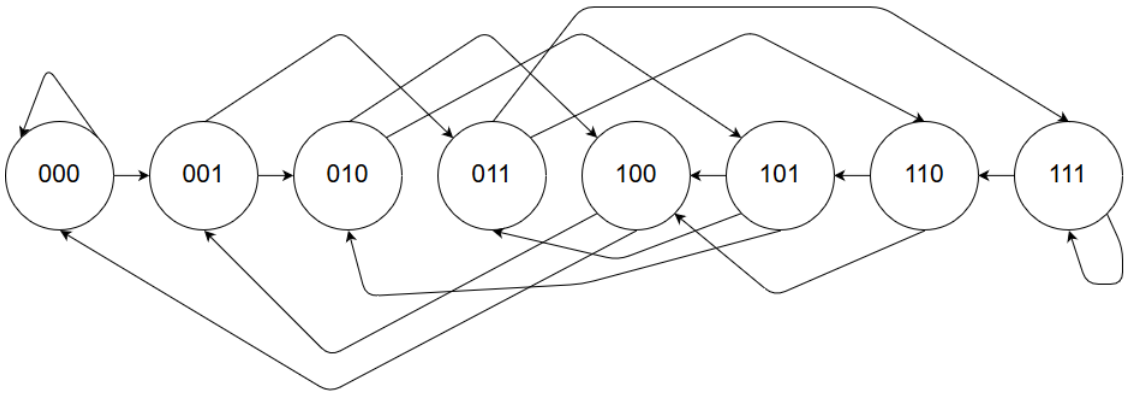



Figura 2.1: Grafo de la cadena $3-universal$: 0001110100.

Encontrar una cadena $k$ universal es equivalente a solucionar el problema de la reconstrucción de cadenas, como la composición $k$-mer es a la colección de todos los $k$-mers binarios. Por lo tanto, encontrar una cadena $k$ universal es equivalente a encontrar un camino Hamiltoniano en el grafo de empalmes formado en todos los $k$-mers binarios. Nicolas de Bruijn estaba interesado en construir cadenas $k$ universales para valores arbitrarios de $k$.

En lugar de buscar por ciclos Hamiltonianos en grandes grafos, de Bruijn desarrollo una nueva manera de representar una composición $k$-mer utilizando un grafo.

Por ejemplo, la representación del genoma $TAATGCCATGGGATGTT$ como una secuencia de 3-mers:

TAA AAT ATG TGC GCC CCA CAT ATG TGG GGG GGA GAT ATG TGT GTT

En este caso, en vez de asignar estos 3-mers a los nodos, se asignan a los vértices, posibilitando reconstruir de nuevo el genoma siguiendo este camino de izquierda a derecha, añadiendo un nuevo nucleótido en cada paso. Dado que cada par de vértices consecutivos representan 3-mers consecutivos que se superponen en dos nucleótidos, etiquetaremos cada nodo de este grafo con un 2-mer representando los nucleótidos superpuestos compartidos por los bordes a ambos lados del nodo.

Nada parece nuevo aquí hasta que se empieza a pegar idénticamente los nodos etiquetados, se lleva los tres nodos AT cada vez más cerca uno del otro hasta que han sido pegados en un solo nodo. Se toma en cuenta que también hay tres nodos etiquetados por $TG$ que se pegan. Finalmente, pegamos juntos los dos nodos etiquetados $GG$ ($GG$ y $GG$), que produce un tipo especial de bordes llamado ciclo que conecta $GG$ a sí mismo.
El número de nodos en el grafo resultante se ha reducido de 16 a 11, mientras que el número de vértices permaneció igual. Este grafo se conoce como el grafo de *de Bruijn* de TAATGCCAATGGGATGTT, denotado $DEBRUIJN_3(TAATGCCAATGGGATGTT)$. Obsérvese que este grafo de Bruijn tiene tres vértices diferentes que conectan $AT$ a $TG$ representando tres copias de la repetición $ATG$.

En general, dado un genoma $Text$, $PATHGRAPH_k(Text)$ es la ruta que consiste en $|Text| - k + 1$ vértices, donde el i-ésimo borde de esta ruta está marcado por el i-ésimo k-mer en $Text$ y el i-ésimo nodo de la ruta está marcado por el i-ésimo $(k-1)$-mer en $Text$. El gráfico de Bruijn $DEBRUIJN_k(Text)$ se forma pegando nodos identificados de forma idéntica en $PATHGRAAPH_k(Text)$

## 2.4. Arquitecturas hardware para el procesamiento de secuencias de ADN

El advenimiento de las últimas generaciones de tecnologías de secuenciación ha abierto muchas nuevas oportunidades de investigación en los campos de la biología y la medicina, incluyendo la secuencia celular de ácido desoxirribonucleico (ADN), el descubrimiento de genes y las relaciones evolutivas. Estas tecnologías han contribuido al crecimiento exponencial de los datos biológicos que están disponibles para los investigadores. Por ejemplo, el GenBank ha duplicado su tamaño de datos aproximadamente cada 18 meses y en su versión de diciembre de 2010 incluyó más de $(122 \times 10^9)$ pares de bases (pbs) de varias especies diferentes. Para ayudar a los biólogos en la extracción de información útil y en la interpretación de las bases de datos de secuencias de gran tamaño, se ha desarrollado un conjunto de algoritmos de alineación (por ejemplo, Smith-Waterman ampliamente utilizado) para resolver muchos problemas abiertos en la campo de la bioinformática, tales como:

- Re-secuenciación del ADN, donde el ensamblaje del genoma se hace frente a un genoma de referencia.
- Alineación de Secuencia Múltiple (MSA), donde múltiples genomas están alineados para realizar la anotación del genoma.
- Hallazgo de genes, donde las secuencias de ácido ribonucleico (ARN) (el transcriptoma) están alineadas contra el genoma del organismo para identificar nuevos genes.

En la actualidad, un enfoque de secuenciación común se basa en la aplicación de las tecnologías High Throughput Short Read (HTSR), para reducir el costo del proceso de secuenciación. Esta técnica consiste en cortar los fragmentos de ADN bajo análisis

en fragmentos más cortos (lecturas), los cuales se secuencian individualmente y se alinean contra una secuencia de referencia. Por lo tanto, las demandas computacionales para el análisis de los datos biológicos producidos por las diversas tecnologías de secuenciación han llevado al desarrollo de varias estrategias de aceleración que apuntan a paralelizar la ejecución de los algoritmos de alineación. Algunas de estas estrategias están basadas en software, mientras que otras utilizan implementaciones de hardware dedicadas.

Entre las primeras, una implementación optimizada que utiliza instrucciones de Single-Instruction Multiple-Data (SIMD) para CPU actuales se adopta comúnmente en programas de alineación de secuencias, como SSEARCH35. Otras implementaciones de software hacen uso de las capacidades de ejecución altamente paralelas presentadas por Graphics Processing Units (GPUs) para lograr un alto rendimiento de alineación. Con respecto a las implementaciones de hardware, éstas incluyen implementaciones de Circuitos Integrados Aplicados Específicos (ASIC) y de Arreglos de Compuertas Programables en Campo (FPGA). Independientemente de la implementación considerada, las arquitecturas de hardware más comunes y eficientes asignan el algoritmo de alineación a una matriz sistólica de Elemento de Procesamiento (PE). Además, aunque se han presentado algunas matrices bidimensionales, las implementaciones más comunes adoptan matrices unidimensionales (lineales). De hecho, las principales diferencias entre las diversas implementaciones se relacionan con el diseño del PE individual. Sin embargo, algunos de estos diseños simplifican excesivamente el algoritmo implementado, calculando únicamente la distancia de edición entre un par de secuencias, por lo que no se adaptan para acelerar el algoritmo S-W más genérico. También se puso a disposición una solución comercial, desarrollada por *CLC bio* e implementada en FPGA, pero con poca información sobre su arquitectura.

Sin embargo, todas las soluciones de hardware presentadas anteriormente sólo se centran en acelerar la primera fase del algoritmo S-W (relleno matricial DP), sin

tener en cuenta la segunda fase (rastreo), que normalmente se realiza utilizando un procesador de propósito general (GPP) Post etapa de procesamiento. Se propuso una arquitectura de hardware que también acelera la fase de rastreo. Sin embargo, sólo se aborda el problema de alineación global. Además, las arquitecturas de hardware propuestas anteriormente no se optimizan fácilmente para tratar secuencias de lecturas cortas, obtenidas a partir de plataformas de secuenciación HTSR actuales.

En otra perspectiva, ha habido un creciente interés en el desarrollo de soluciones de procesamiento que combinan, en un solo paquete, las capacidades de reconfiguración ofrecidas por FPGAs con las ventajas de una CPU cableada. Estas soluciones, como el procesador Intel Atom E645C, permiten implementar aceleradores de hardware altamente especializados acoplados con una CPU de uso general, con el fin de mejorar significativamente el rendimiento general del sistema. Además, haciendo uso de las capacidades de reconfiguración ofrecidas, es posible implementar una amplia gama de aceleradores de acuerdo con la tarea específica que se está ejecutando actualmente. Esta capacidad de multiplexación de tareas a lo largo del tiempo reduce el costo total de propiedad de dicho sistema debido a su adaptabilidad, bajo coste inicial y alto rendimiento.

Para superar las limitaciones de las arquitecturas de aceleradores anteriores, para mejorar el rendimiento general de secuenciación y aprovechar las ventajas proporcionadas por los FPGA actuales, se retoma una arquitectura de hardware junto a una técnica para acelerar la alineación de secuencias. Este acelerador, dirigido a una plataforma integrada, se basa en la explotación de las siguientes dos importantes contribuciones:

1. Una técnica innovadora y bastante eficiente que hace uso de la información recogida durante el cálculo de las puntuaciones de alineación en la fase de llenado de matriz (en hardware), con el fin de reducir significativamente los requisitos de tiempo y memoria de la fase de rastreo (posteriormente implementado en

software). Para soportar tal técnica, la arquitectura de acelerador de hardware desarrollada se integró estrechamente con un GPP, para formar un sistema de alineación local completo y bastante eficiente implementado en un FPGA.

2. También se aprovecha un nivel adicional de paralelismo en la estructura de aceleración propuesta para aumentar aún más su rendimiento. Con la estructura presentada, varias secuencias de consulta pueden alinearse simultáneamente con la misma secuencia de referencia, permitiendo así una aceleración significativa de la tarea de alineación de las lecturas cortas contra el genoma de referencia, tal como se utiliza en las técnicas HTSR. Esto se logra configurando el acelerador desarrollado en un flujo múltiple. La estructura incluye múltiples arreglos lineales que trabajan en paralelo. La aceleración que consigue con dicha mejora es proporcional al número de matrices lineales que implementa (definidas a través de la parametrización de la plataforma).

# Capítulo 3

# Trabajo relacionado

Dentro del estado del arte se han identificado tres diferentes enfoques que abordan el problema de ensamble de secuencias de ADN.

- **Procesamiento Digital de Señales:** Este ha sido el enfoque menos trabajado, de cualquier forma, es importante realizar la revisión para analizar las técnicas utilizadas de emparejamiento de señales, donde se podrían utilizar técnicas bien conocidas para hallar secuencias candidatas a ensamblarse y construir el grafo de ensamble, esto sería posible utilizando el enfoque OLC, reemplazando el enfoque de emparejamiento de cadenas por el de señales.

- **Probabilista:** En este enfoque se ha probado que el problema de ensamble de genomas puede ser resuelto de manera similar a los rompecabezas de 1 - dimensión para lecturas independientes y las lecturas en pares son análogas a resolver un rompecabezas de 2 - dimensiones.

- **Transformación Burrows-Wheeler:** Este es uno de los enfoques más utilizados, y se han propuesto diversas arquitecturas para distintas tareas en bioinformática molecular. Nuevos algoritmos permiten acelerar el tiempo requerido por las consultas, lo cual representa una gran ventaja sobre los otros métodos,

además requiere una fracción del espacio de memoria.

Desde sus inicios, la bioinformática molecular ha trabajado con enormes cantidades de datos, por tal motivo, en muchas ocasiones los datos se encuentran comprimidos utilizando técnicas relacionadas con la transformación Burrows-Wheeler (BWT). En los casos donde no se utiliza la BWT se debe realizar un pre-procesamiento para revertir la compresión utilizando la estructura de datos conocida como FM-index, la cuál es un vector de sufijos ordenados, calculados durante la BWT. El FM-index ha sido utilizado para realizar consultas aprovechando las estadísticas previamente obtenidas, en lugar de utilizar la búsqueda exhaustiva como otros métodos. De este modo, la importancia de esta técnica se ha identificado para tareas de ensamble de genomas, mapeo y alineación de secuencias, donde las consultas de emparejamiento son altamente repetitivas.

## 3.1. Algoritmos de ensamble de cadenas de ADN

En el trabajo de [Simpson, 2010, Simpson, 2011] se resalta la importancia de estas técnicas. Al utilizar una combinación de la BWT con el índice-FM construyeron un ensamblador de secuencias de ADN utilizando la metodología *OLC*. Esta técnica utiliza tres etapas descritas a continuación:

1. *Creación del índice*: Utiliza la BWT para formar un índice. Al ordenar por prefijos los símbolos de las cadenas, facilitará las búsquedas en la siguiente etapa. En el trabajo [Simpson, 2010] se describe un algoritmo para la creación del índice, sin embargo, este se puede realizar de otra forma, en [Holt, 2014] se propone la creación del índice utilizando un algoritmo de mezcla de cadenas BWT, lo que facilita la inserción de más cadenas una vez construido, sin la necesidad de reconstruirlo desde el inicio.

Estos dos trabajos pueden ser modificados para disminuir temporalmente la construcción del índice, tal como lo propone en [Chacon et al., 2013]. Es posible utilizar prefijos de distintos tamaños para acelerar la construcción, dividiendo la complejidad algorítmica en $n$ (sea $n$ la longitud del prefijo), como desventaja tiene un incremento exponencial en el tamaño del alfabeto del índice. Esto no representa un problema, dado que el tamaño original del alfabeto es de 5 elementos se puede tomar un $n < 6$ sin notar un incremento significativo en el uso de memoria RAM.

2. *Búsqueda de superposiciones*: Se utiliza una modificación algoritmo de búsqueda de hacia atrás con el índice-FM para encontrar las superposiciones aproximadas para el índice generado, el tamaño del alfabeto vuelve a ser de 5 símbolos. En el trabajo de [Kucherov et al., 2014] se propone una búsqueda hacia adelante y atrás para disminuir la cantidad de iteraciones requeridas para hallar las superposiciones aproximadas. Dentro de esta etapa se genera un grafo con las superposiciones halladas, posteriormente se eliminan los vértices irreducibles utilizando un pequeño algoritmo agregado a la búsqueda, aprovechando nuevamente el índice-FM.

3. *Recorrido en el grafo de secuencias*: En la última etapa se recorre el grafo y se retorna el conjunto de *contigs* ensamblados.

Los resultados temporales de este trabajo se presentan en [Simpson, 2011], a pesar de tener un tiempo mayor que los trabajos comparados, tiene dos grandes ventajas; el uso de Memoria de Acceso Aleatorio (RAM) es muy inferior y la característica más importante, es que, tiene la habilidad de crear un índice para volver a ejecutar el ensamble con distintos parámetros sin la necesidad de ejecutarse completamente, esto le permite generar un nuevo ensamble en menor tiempo que las herramientas más destacadas utilizando una fracción de la memoria requerida por los otros.

## 3.2. Métodos de ensamble utilizando la transformada Burrows-Wheeler

En capítulos anteriores se ha mencionado la importancia y la dificultad del problema de ensamble dentro de la bioinformática. La mayoría de genomas, particularmente genomas de eukaryota, son altamente repetitivos, hecho que dificulta el ensamble al obstaculizar hallar las relaciones verdaderas entre las lecturas con muchas opciones falsas.

Con intención de ayudar a revelar las relaciones verdaderas entre las lecturas y aquellas introducidas por diferentes copias de las repeticiones, resulta útil construir un grafo donde todas las copias de una repetición se encuentren colapsadas dentro de un solo segmento. Este grafo se encuentra definido como un grafo de repeticiones.

Esta estructura es una consecuencia natural del método de grafo de de Bruijn de ensamble de secuencias como la descomposición de las secuencias leídas en k-mers colapsando las repeticiones que comparten el mismo k-mer dentro de un único vértice. Se conocen otras alternativas, por ejemplo, el grafo de cadena, este es construido partiendo de un grafo de los pares emparejados entre las secuencias leídas y transformándolo en un grado de cadena, removiendo los vértices de transición.

Al igual que el grafo de de Bruijn, tiene la propiedad de que las repeticiones son colapsadas dentro de una única unidad sin la necesidad de descomponer las lecturas en $k$-mers, dado que se encuentra basado en máximos empalmes, los cuáles son típicamente de mayor longitud que los $k$-mers de de Bruijn, al igual que este último, ayuda a eliminar la ambigüedad en lecturas más largas que la de de Bruijn en sus pasos posteriores. Debido al costo que representa construir este tipo de grafos, la mayoría de los ensambladores han sido basados en el enfoque de de Bruijn.

Sin embargo, también se ha utilizado la transformada Burrows-Wheeler para

construir el grafo de cadena utilizando el indexado del conjunto de secuencias, en la tabla 3.1 muestra trabajos relacionados que han utilizado de forma eficiente esta metodología para tareas de ensamble o de preprocesamiento, además, [Simpson, 2010, Simpson, 2011] muestran que es posible su construcción utilizando directamente el índice-FM sin la necesidad de hallar todos los empalmes además de ser eficiente en tiempo y espacio. Si bien no todos los trabajos presentados en esta tabla se encuentran directamente relacionados con tareas de ensamble, sí realizan tareas que emplean tareas de búsquedas repetitivas, como por ejemplo, las alineaciones, mapeos y compresión, al igual que estas, la corrección de errores es un preprocesamiento común a la tarea de ensamble.

| Estado del Arte | | | | |
|---|---|---|---|---|
| Artículo | Área | Metodología | Contribución | Compresión |
| [Greenstein et al., 2015] | Corrección de errores | FM-index | Uso de K-mer para corrección de errores en short-reads | Sí |
| [Wienbrandt, 2013] | Ensamble | Usa la computadora RIVYE-RA y BWT | Ensamble con BWTA | Sí |
| [Danek et al., 2012] | Ensamble | BWT para hallar repeticiones | Nueva herramienta: BWatrs | Sí |
| [Rodland, 2013] | Ensamble | Nuevo índice para los k-mer | kFM-index | Sí |
| [Chikhi et al., 2014] | Ensamble | De Bruijn graphs | Usa FM-index y ABySS | No |
| [Zhang et al., 2015] | Ensamble | FM-index | Ensamble FM-index paralelo | Sí |
| [Holt, 2014] | Compresión | Mezcla multiples-cadenas BWT | Mejora la razón de compresión | Sí |
| [Simpson, 2010] | Ensamble | FM-index | Utiliza empalmes para ensam-blar | Sí |
| [Simpson, 2011] | Ensamble | FM-index | Ensambla genomas usando FM-index | Sí |
| [Mantaci et al., 2005] | Compresión | BWT para genomas | Transformación extendida para genomas | Sí |

Tabla 3.1: Estado del arte: Transformada Burrows-Wheeler

## 3.3. Métodos de ensamble en arquitecturas hardware

Las máquinas de secuenciación de próxima generación ofrecen millones de fragmentos cortos de ADN llamados *short-reads*. La asignación de un volumen tan grande de *short-reads* es un aspecto fundamental en la biología computacional moderna. La secuenciación completa de un organismo da la posibilidad de entender las enfermedades genéticas, identifica genomas de cáncer y explora las posibilidades de nuevos medicamentos y medicina personalizada. El mapeo de lecturas cortas se ha realizado ya utilizando herramientas de software bien conocidas, al igual que usando un clúster de procesadores de propósito general (CPU). El mapeo de *short-reads* en un genoma tan grande como un ser humano sigue siendo un gran desafío para las aplicaciones de software. Recientes avances de hardware programable como arreglos de compuertas programables en campo (FPGAs) proporcionan una solución eficaz a este reto. Los FPGAs contienen millones de puertas lógicas programables y están conectados a grandes recursos de almacenamiento como memoria de doble tasa de transferencia (DDR) tipo 2 y 3 así como el cubo de memoria híbrido (HMC).

En la tabla 3.2 se resume el estado del arte que sirve de guía para este trabajo, en esta se presentan trabajos que reducen el costo temporal en las diferentes tareas para el ensamble de genomas. Esto se logra al acelerar la comunicación con las memorias al igual que al limitar el uso del procesador de propósito general para transportar los datos, mientras que las operaciones de mayor frecuencia son realizadas por una arquitectura sintetizada dentro de una FPGA. En los trabajos presentados por [Kuo et al., 2013, Varma et al., 2014, Hu et al., 2012, Pfeiffer et al., 2009] se crean arquitecturas a partir de algoritmos y herramientas software bien conocidas en la tarea de ensamble. Por otro lado los trabajos [Arram et al., 2015, Arram et al., 2013a] basan sus arquitecturas el la búsqueda a través del índice-FM para tareas de alineación y compresión.

| Estado del Arte | | | | |
|---|---|---|---|---|
| Artículo | Área | Metodología | Contribución | Compresión |
| [Kuo et al., 2013] | Ensamble | FPGA | pre-procesador y post-procesador | No |
| [Varma et al., 2014] | Ensamble | FPGAs con HEBs | Arquitectura hardware de Velvet | No |
| [Hu et al., 2012] | Ensamble | Implementación CMOS para comparación paralela | arquitectura CMOS para ensamble de genomas | No |
| [Waidyasooriya, 2016] | Alineación | alineación por FPGA de short-reads | Aceleración por hardware para short-reads | Sí |
| [Waidyasooriya et al., 2013] | Mapeo | Arquitectura hardware | Algoritmo e implementación BWA | Sí |
| [Arram et al., 2013b] | Mapeo | Arquitectura hardware | Emparejamiento de cadena aproximado | Sí |
| [Arram et al., 2015] | Compresión | Arquitectura hardware | Búsqueda en FM-index | Sí |
| [Arram et al., 2013a] | Alineación | Arquitectura hardware | FM-index backtacking | Sí |
| [Pfeiffer et al., 2009] | Ensamble | implementación paralela en FPGA | paralelismo en FPGAs de bajo costo | No |

Tabla 3.2: Estado del arte: Arquitectuas hardware

## 3.4. Métodos de ensamble y subtareas importantes

En la tabla 3.3 se resumen los métodos del estado del arte que han sido abordados para tratar las tareas de ensamble y mapeo. Dentro de esta sección se abarcan cuatro subtareas importantes:

1. Grafos de de Bruijn: En este apartado se trata de resolver subproblemas relacionados con este tipo de grafos, por ejemplo, grafos que no son conexos y grafos con ciclos.

2. Caminos de Euler: En este trabajo se realiza un análisis y se resuelve el problema de hallar el camino de Euler para obtener el ensamble, esto resulta útil con grandes cantidades de datos

3. Modelo probabilista: En este enfoque se ha probado que el problema de ensamble de genomas puede ser resuelto de manera similar a los rompecabezas.

4. Modelo de Procesamiento Digital de Señales: Este enfoque es el menos utilizado, realizando un submuestreo y filtrado de la señal de un *short-read* busca señales de otros *short-reads* que se asemejen para poder empalmarlos y de este modo formar el mapeo.

| Estado del Arte | | | | |
|---|---|---|---|---|
| Artículo | Área | Metodología | Contribución | Compresión |
| [Huang et al., 2013] | Ensamble | Grafos de De Bruijn | No requiere almacenar los grafos completos | No |
| [Poirier et al., 2015a] | Ensamble | Ray comparación de algoritmos | Demuestra la eficiencia de las FPGA | No |
| [Goodarzi et al., 2014] | Ensamble | Mezcla contigs | Crea contigs usando short-reads | No |
| [Sasoglu and Tse, 2014] | Ensamble | Modelo probabilista | Demostración del ensamble de genomas | No |
| [Sović et al., 2013] | Ensamble | Survey | Algoritmos e implementaciones | - |
| [Sedlar et al., 2014] | Compara ADN | DSP | Sub-muestreo de la señal de ADN | Sí |
| [Buermans, 2014] | Tecnologías de secuenciación | Survey | Análisis de las tecnologías de | - |
| [Andrews, 2016] | Controla calidad en las Short-reads | Revisa los datos brutos | Conjunto de análisis modular | Sí |
| [Wang et al., 2015] | Ensamble | Búsqueda de caminos de Euler | Eficiente con grandes cantidades de datos | No |

Tabla 3.3: Estado del arte: Ensamble y corrección de errores

## 3.5. Resumen

En las tablas presentadas anteriormente se muestran los artículos revisados y el enfoque que ellos utilizaron. Es importante señalar algunos trabajos que operan sobre BWT, los cuales son interesantes por sus resultados. En este trabajo, se retoman los algoritmos y métodos para mejorar la velocidad de procesamiento, encontrar superposiciones y construir el gráfo usando el índice FM. Esto también se puede usar para ensamblar lecturas cortas previamente comprimidas. En la tabla 3.4 se muestran los artículos base de este trabajo.

| Artículo | Metodología | Propuesta |
|---|---|---|
| [Simpson, 2010] | Grafo de cadenas utilizando el índice-FM | Arquitecturas hardware para la aceleración de la creación el índice-FM |
| [Simpson, 2011] | Ensamblador basado en [Simpson, 2010] | Arquitectura hardware y modificación algorítmica para la creación del índice-FM |
| [Holt, 2014] | Creación del índice-FM a partir cadenas BWT | Modificación algorítmica |
| [Li, 2014] | Creación del índice-FM a partir de lecturas | Arquitectura hardware basada en el algoritmo |
| [Chacon et al., 2013] | Búsqueda de patrones acelerada para el índice-FM | Adaptar la metodología para la formación del índice-FM |

Tabla 3.4: Trabajos base

# Capítulo 4

# Arquitectura hardware propuesta para la creación del índice-FM

Este trabajo se basa en el trabajo de [Simpson, 2011], en el cuál se ha creado una metodología que utiliza diversos algoritmos relacionados con la transformación Burrows-Wheeler, incluyendo el índice-FM, ampliamente utilizado para la búsqueda de patrones.

El algoritmo *String Graph Assembler* (SGA) realiza consultas sobre un índice-FM construído a partir de un conjunto de lecturas de secuencias. El pipeline del SGA comienza preprocesando las lecturas de la secuencia para filtrar o recortar lecturas con varias llamadas base ambiguas o de baja calidad. El índice de FM se construye con el conjunto filtrado de las lecturas y llamadas base de errores que se detectan y corrigen usando frecuencias k-mer. Las lecturas corregidas son reindexadas, entonces las secuencias duplicadas se eliminan, se filtran secuencias restantes de baja calidad, y se construye un grafo de la cadenas. Los *contigs* se ensamblan en el grafo de cadena y se construyen los *scaffolds* si hay datos de extremos emparejados o *mate-par*. La figura 4.1 muestra el flujo de datos a través del pipeline.

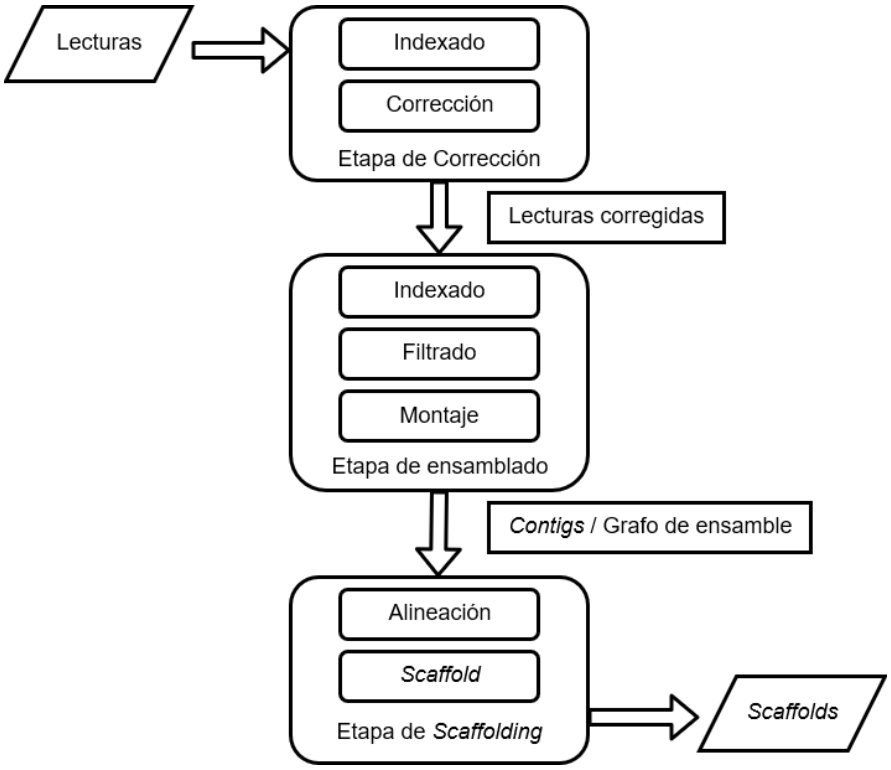



Figura 4.1: Diagrama de alto nivel del pipeline del ensamblador SGA.

La etapa de corrección de error inicia con la construcción de un índice de FM para las lecturas, luego realiza la corrección de errores. La etapa de ensamble toma las lecturas corregidas como entrada, vuelve a construir los índices, elimina los duplicados y lecturas de baja calidad, por último construye los contigs. La etapa de scaffolding realinea las lecturas originales a los *contigs* usando BWA, construye un grafo de *scaffold* con las alineaciones y salida un conjunto final de *scaffolds*.

Para demostrar el funcionamiento del SGA y su capacidad para escalar a grandes genomas, los autores han realizado corrección de errores y ensamble en una amplia gama de tamaños de genomas, desde bacterias a mamíferos. Por estas razones se ha seleccionado este ensamblador y nuestro trabajo se encuentra centrado en el diseño de una arquitectura hardware para la construcción del índice. En las siguientes secciones se describe el funcionamiento general del algoritmo SGA y la metodología

propuesta para acelerar su ejecución mediante una arquitectura hardware.

El índice FM desempeña un papel importante en la alineación de secuencias de ADN, el ensamblaje de novo y la compresión. La construcción rápida y ligera del índice FM para un gran conjunto de datos es la clave para estas aplicaciones. En este contexto, se han desarrollado algunos algoritmos que superan sustancialmente los algoritmos anteriores. Sin embargo, sólo son eficientes para lecturas cortas. No existía un algoritmo rápido y práctico para las lecturas de secuencias largas. Gracias al algoritmo presentado en [Li, 2014] se pudo crear la herramienta RopeBWT, que utiliza el ensamblador SGA para la creación de sus índices.

## 4.1. Arquitectura basada en el algoritmo ropeBWT

### 4.1.1. Creación del índice a partir de lecturas

RopeBWT es una herramienta para construir el índice FM para una colección de secuencias de ADN. Funciona insertando incrementalmente una o varias secuencias en una posición de pseudo-BWT existente por posición, comenzando desde el final de las secuencias. Este algoritmo puede considerarse en gran medida como una mezcla del algoritmo BCR [Bauer et al., 2011] y el índice FM dinámico. No obstante, el algoritmo RopeBWT es único en cuanto a que puede implícitamente clasificar la entrada en orden lexicográfico inverso (RLO) o en orden lexicográfico de complemento inverso (RCLO) mientras se construye el índice.

### 4.1.2. Procedimiento

Sea $\Sigma = A, C, G, T$ el alfabeto del ADN con un orden lexicográfico $A < C < G < T < N$. Cada elemento en $\Sigma$ se llama símbolo y una secuencia de símbolos

llamados cadena sobre el alfabeto $\Sigma$. Dada una cadena $P$, $|P|$ es su longitud y $P[i]$ es el símbolo en la posición $i$. Un centinela \$ es lexicográficamente menor que todos los otros símbolos. Por simplicidad, tenemos $P[-1] = P[|P|] = \$$. También se denomina $P^{\sim}$ como el inverso de $P$ y $P$ como el complemento inverso de $P^{-1}$.

Dada una lista de cadenas sobre $\Sigma$, $(P_i)_{0 \leq i < m}$, vamos $T = P_0\$_0...P_{m-1}\$_{m-1}$ con $\$_0 < ... < \$_{m-1} < A < C < G < T < N$. La matriz de sufijo de $T$ es una matriz de números enteros $S$ tal que $S(i)$, $0 \leq |T|$, es la posición inicial del i-ésimo sufijo más pequeño en la colección $T$. La Transformada Burrows-Wheeler, o BWT, de $T$ puede calcularse como $B[i] = T[S(i) - 1]$. Para la descripción del algoritmo, segmentamos $B$ en $B = B_\$, B_A, B_C, B_G, B_T, B_N$, en donde $B_a[i] = B[i + C(a)]$ con $C(a) = |j : T[j] < a|$ siendo la matriz de recuentos acumulativos. Por la definición de matriz de sufijo y BWT, $B_a$ consiste en todos los símbolos con su siguiente símbolo en $T$ siendo $a$. Esta es la definición de la BWT para una lista ordenada de cadenas.

A continuación se define la BWT para un conjunto no ordenado de cadenas $C$ imponiendo una orden de clasificación arbitraria en $C$. Decimos que la lista $P(i)_i$ está en el orden lexicográfico inverso (RLO), si $\widetilde{P}_i \leq \widetilde{P}_j$ para cualquier $i \leq j$; y que está en el orden lexicográfico de complemento inverso (RCLO), si $\bar{P}_i \leq \bar{P}_j$ para cualquier $i < j$. El RLO-BWT de $C$, designado por $B^{RLO}(C)$, se construye ordenando las cadenas $C$ en RLO; aplicando el procedimiento del párrafo anterior en la lista ordenada. RCLO-BWT $B^{RCLO}(C)$ puede construirse de forma similar.

En $B^{RCLO}(\{P_i\}_i \cup \{\bar{P}_j\}_j)$, la secuencia $k-th$ menor es el complemento inverso de la secuencia $k - th$ en el índice FM. Esta propiedad elimina la necesidad de mantener una matriz extra para vincular el rango y la posición de una secuencia en el índice FM, y así ayuda a reducir la memoria de algunos algoritmos basados en el índice FM. Para lecturas cortas, RLO / RCLO-BWT también se incrementa la razón de compresión.

Inicialmente se definen dos operaciones de cadena: $rank(c, k; B)$ e $insert(c; k; B)$,

donde $rank(c; k; B) = |i < k : B[i] = c|$ indica el número de símbolos $c$ antes de la posición $k$ en $B$ e $insert(c; k; B)$ inserta el símbolo $c$ después de $k$ símbolos en $B$ con todos los símbolos después de la posición $K$ recorriéndose para hacer espacio para $c$. Implementamos las dos operaciones representando cada $B_c$ en un $B + -tree$ en memoria, donde una hoja guarda una codificación con una longitud $n$ de caracteres repetidos ($RLE$) de una cadena y un nodo interno mantiene el recuento de cada símbolo en las hojas descendientes del nodo.

**Algorithm 2**

**Recibe:** Una cadena $P$ a una cadena $BWT$ $B$ para $T$

**Retorna:** $BWT$ de $TP\$$

```
1: function AÑADIRUNACADENA(B, P)
2:     c ← $
3:     k ← |i : B[i] = $|
4:     for i ← |P| − 1 hasta −1 do
5:         insert(P[i], k; B_c)
6:         k ← rank(P[i], k; B_c) + Σ_{a<c}|j : B_a[j] = P[i]|
7:         c ← P[i]
8:     return B
```

El algoritmo 2 agrega una cadena a un índice existente, insertando cada uno de su símbolo al final de $P$. El algoritmo 3 construye RLO / RCLO-BWT de una manera similar al algoritmo 2 excepto que inserta $P[i]$ a $[l, u)$, el intervalo de sufijo del sufijo $P's$ que comienza en $i + 1$, y que los símbolos BWT en este intervalo ya están ordenados. Este proceso aplica implícitamente un tipo de raíz desde el final de $P$, clasificándolo en las cadenas existentes en el BWT en RLO / RCLO. Nótese que si se cambia la línea 1 a $l \Leftarrow u \Leftarrow |i : B[i] = \$|$, el algoritmo 3 se convertirá en el algoritmo 2. Recordando que el algoritmo de BCR [Bauer et al., 2013] es, hasta cierto punto,la versión Multi-string del algoritmo 2.

**Algorithm 3** Inserta una cadena en RLO/RCLO-BWT.

**Recibe:** $B^{RLO}(C)$ o $B^{RCLO}(C)$ y una cadena $P$

**Retorna:** $B^{RLO}(C \cup P)$ o $B^{RCLO}(C \cup P)$

1: **function** AÑADIRRLO/RCLO-BWT(B, P, comp)
2: $[l, u) \leftarrow [0, |i : B[i] = \$)$
3: **for** $i \leftarrow |P| - 1$ hasta $-1$ **do**
4: $[l, u) \leftarrow INSERTAUX(B, P[i], l, u, P[i + 1], comp)$
5: **return** B
6: **function** INSERTAUX(B,c',l,u,c,comp)
7: $k \leftarrow l$
8: **if** $comp = TRUE \wedge c' \neq "N"$ **then**
9: **for** $a \leftarrow \$ - 1 \vee < a$ hasta $"N"$ **do**
10: $k \leftarrow k + [rank(a, u; B_c) + rank(a, l; B_c)]$
11: **else**
12: **for** $\$ \leq a < c'$ **do**
13: $k \leftarrow k + [rank(a, u; B_c) + rank(a, l; B_c)]$
14: $l' \leftarrow [rank(c', l; B_c)$
15: $u' \leftarrow rank(c', u; B_c)]$
16: $insert(c', k; B_c)$
17: $m \leftarrow \Sigma_{a<c} | j : B_a[j] = c'$
18: **return** $[l' + m, u' + m]$

**Algorithm 4** Inserta múltiples cadenas

**Recibe:** Una cadena $BWT\ B$ y una lista de cadenas $P_{kk}$

**Retorna:** La cadena $BWT\ B$ actualizada con las cadenas en el orden especificado

1: **function** AÑADIRMULTIPLESCADENAS(B, $P_{kk}$,sort,comp)
2: **for** $i \leq j < |P_{kk}|$ **do**
3: $A(j).c \leftarrow \$$
4: $A(j).i \leftarrow j$
5: **if** $sort = TRUE$ **then**
6: $[A(j).l, A(j).u) \leftarrow [0, |i : B[i] = \$|)$
7: **else**
8: $[A(j).l, A(j).u) \leftarrow |i : B[i] = \$| + j$
9: $d \leftarrow 0$
10: **while** $|A| \neq 0$ **do** ▷ Matriz de clasificación estable A por A(.).c
11: **for** $i \leq j < |P_{kk}|$ **do**
12: $c \leftarrow A(j).c$
13: $A(j).c \leftarrow P_{A(j).i}[|P_{A(j).i}| - 1 - d]$
14: $[A(j).l, A(j).u) \leftarrow INSERTAUX(B, A(j).c, A(j).l, A(j).u, c, comp)$
15: **if** $A(j).c = \$$ **then**
16: Eliminar A(j)
17: $d \leftarrow d + 1$
18: **return** B

Siguiendo un razonamiento similar, podemos extender el algoritmo 3 para insertar múltiples cadenas al mismo tiempo, lo que da como resultado el algoritmo 4. Usamos un arreglo $A(j)$ para mantener el estado de la $j$-ésima secuencia después de insertar su sufijo *d-long*. En la línea 2, $A(j).c$ es el símbolo previamente insertado y $[A(j).l, A(j).u]$ es el intervalo al que se inserta el nuevo símbolo. En la implementación, podemos acelerar el modo de clasificación insertando varios símbolos en la línea 3.

Cuando B está representada por una estructura de árbol balanceada, la complejidad temporal de los tres algoritmos es $\mathcal{O}(n \log n)$, donde $n$ es el número total de símbolos en la entrada. Sin embargo, veremos más adelante que para las secuencias cortas, el Algoritmo 3 es más rápido que los otros dos algoritmos, debido a la localidad de los accesos a la memoria, a la posibilidad de actualizar en caché $B+-tree$ y a la paralelización del ciclo $for$ en la línea 1. Estas técnicas son más eficaces para un lote más grande de cadenas más cortas. Sin tener en cuenta RLO / RCLO, el algoritmo 4 es similar a BCR excepto que BCR mantiene $B$ en arreglos monolíticos. Como resultado, la complejidad de tiempo de BCR es $\mathcal{O}(nl)$, en donde $l$ es la longitud máxima de lecturas, no escalando bien a $l$.

### 4.1.3. Funcionamiento de la arquitectura

La arquitectura hardware propuesta que genera la BWT de las cadenas con las características mencionadas divide a la arquitectura en tres tareas; la primera consiste en la inserción de símbolos y la posición de estos, en la segunda el cálculo de las posiciones donde se realizarán las inserciones. La tercera consiste en la actualización de las posiciones de los símbolos previamente insertados.

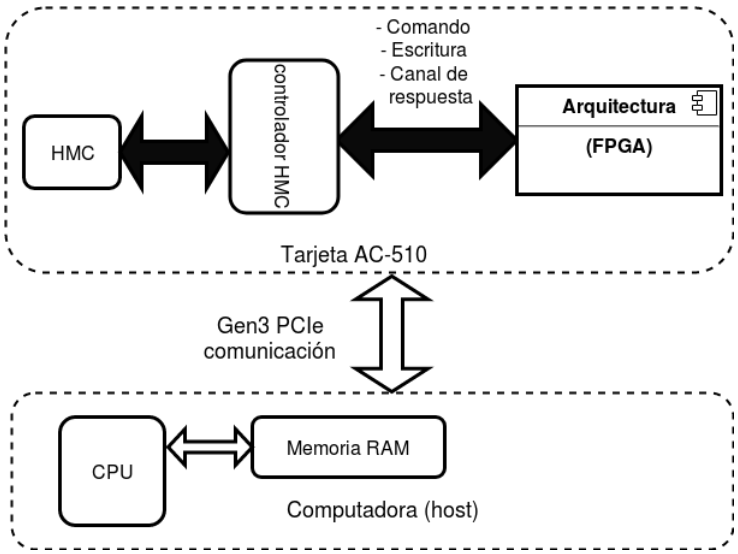



Figura 4.2: Comunicación de la arquitectura compatible con memorias del tipo HMC.

La figura 4.2 muestra un diagrama de alto nivel de la comunicación en la arquitectura propuesta. Esta arquitectura se compone de un procesador de propósito general, requiere de comunicación mediante una interfaz conocida como interconexión de componentes periféricos *express* (PCIe, por sus siglas en inglés), el sistema de memoria HMC y la arquitectura hardware. El procesador de propósito general y la interfaz PCIe son necesarios para recibir el conjunto de datos y enviar la BWT de salida. El controlador de memoria utilizado es HMC de la plataforma *pico computing*. El controlador de memoria crea la interfaz necesaria entre la memoria HMC y la arquitectura hardware.

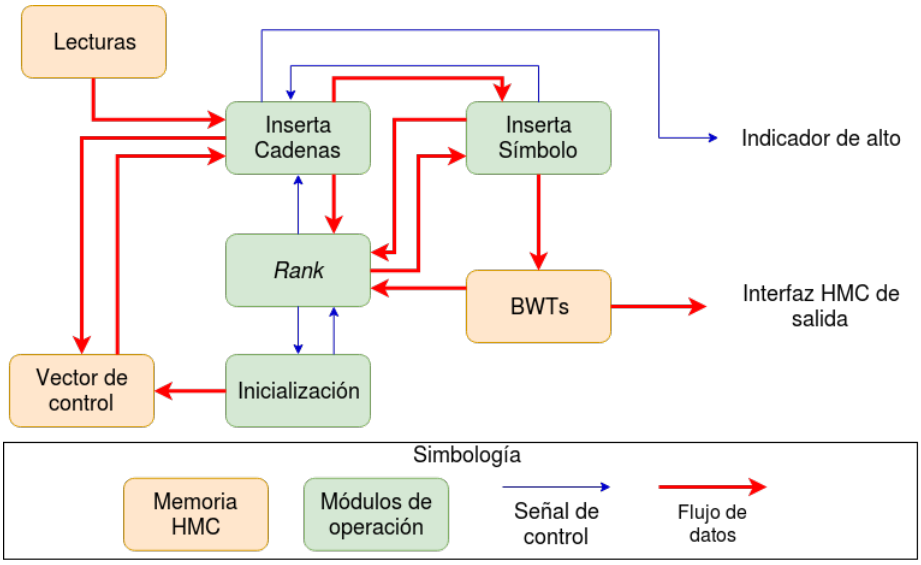



Figura 4.3: Diagrama de alto nivel del diseño de la arquitectura propuesta para el algoritmo ropeBWT.

La figura 4.3 describe el diagrama a bloques de la arquitectura hardware. Requiere almacenar una gran cantidad de datos, por lo que se necesita utilizar una memoria con una interfaz que permita acceder a los datos a gran velocidad, leer las lecturas y escribir en el vector de control que alimenta a los módulos de *Inicialización* e *Inserta Cadenas*, el primero toma palabras de 8-bits, mientras que el segundo maneja palabras de 32-bits. El módulo *Rank* maneja palabras de 32-bits, envía y recibe señales al término de su operación de los módulos *Inserta Cadenas* e *Inserta símbolo*, este último módulo escribe palabras de 32-bits en la memoria en donde se almacena la cadena BWT. El módulo *Inserta Cadenas* determina el término del cálculo de la BWT y envía una señal de alto, en caso contrario se envía otra señal para continuar con el cálculo.

La figura 4.4 muestra la máquina de estados que describe el comportamiento de la arquitectura propuesta. En el estado E0, la arquitectura recibe las cadenas, toma los caracteres a insertar en la BWT, mezcla e inicializa los apuntadores y calcula la posición en donde se debe insertar el siguiente símbolo. En el estado E1, se realiza

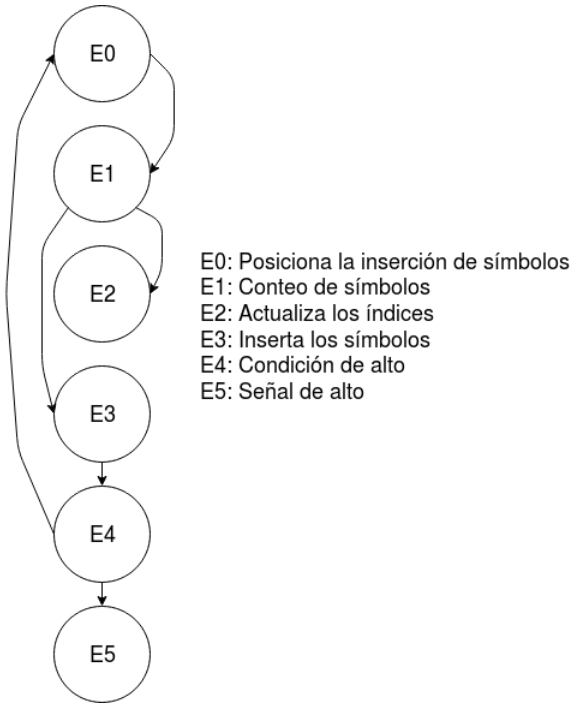



Figura 4.4: Máquina de estados de la arquitectura hardware del algoritmo $ropeBWT$.

el conteo de símbolos para su posicionamiento dependiendo del conteo requerido por el algoritmo. En el estado E2 actualizan los índices de los símbolos insertados previamente. En el último estado E3, inserta cada símbolo en la posición indicada. El estado E4 comprueba que se hayan recorrido todos los símbolos de todas las cadenas para terminar la inserción de los elementos. El último estado, E5, se indica que ha terminado la BWT.

## 4.2. Arquitectura basada en el algoritmo mergeBWT

### 4.2.1. Creación del índice a partir de cadenas BWT

Un método para crear una BWT a partir de múltiples cadenas BWT se presenta en el trabajo de [Holt, 2014]. Se define una cadena como una serie de $k$ símbolos

del alfabeto $\Sigma$, terminando con un símbolo especial \$. Sea $S$ una colección de cadenas, $S = S_1, S_2, ..., S_m$. Se construye la BWT con múltiples cadenas de modo que una cadena se puede reconstituir anteponiendo los predecesores hasta que se alcanza el índice inicial (cada cadena forma un bucle en el BWT). En una sola pasada a través de todas las BWT de entrada, contamos el número de ocurrencias para cada símbolo y determinamos una lista de compensaciones en la BWT final para el primer sufijo que comienza con cada símbolo. Estos recuentos y desplazamientos son un pequeño subconjunto del índice FM, para la BWT de salida, en lugar de los BWT de entrada.

El objetivo del algoritmo es construir el intercalado de dos o más cadenas BWT tal que sus sufijos implícitos se encuentran ordenados. Se inicia construyendo un intercalado inicial de las cadenas BWT de entrada, esto es una concatenación de las entradas. Luego de una serie de iteraciones en este intercalado, en la que cada iteración actúa como un paso del ordenamiento radix sobre los sufijos representados por el intercalado BWT. Luego de una iteración, el intercalado será ordenado lexicográficamente para todos los sufijos de longitud 1. En la segunda iteración, los sufijos de longitud 2 serán ordenados. Estas iteraciones continuarán hasta que no haya cambios en el intercalado, indicando que los sufijos se encuentran completamente ordenados.

Dadas dos cadenas BWT $B_0 = msbwt(S_0)$ y $B_1 = msbwt(S_1)$, de longitud $m$ y $n$, respectivamente, el resultado, $B_2 = msbwt(S_0, S_1)$, puede ser construido si el intercalado de $B_0$ y $B_1$ es conocido. El algoritmo utiliza un arreglo auxiliar $I$ para el intercalado, el cual es una serie de ceros y unos de longitud $(m + n)$, de tal forma que cada cero corresponde a un símbolo originado en $B_0$ y cada uno a un símbolo originado en $B_1$. Tal que hay exactamente $m$ ceros y $n$ unos en $I$. Como el algoritmo de intercalado es progresivo, este intercalado $I$ será corregido hasta la convergencia de un intercalado final.

El algoritmo también se apoya de un arreglo que almacena el histograma acumulado, un equivalente al componente $offsets$ del índice FM.

**Algorithm 5** Actualiza

**Recibe:** El intercalado $I$ de $B_0$ y $B_1$, las cadenas BWT $B_0$ y $B_1$ y el arreglo de $offsets$

**Retorna:** El intercalado $I$ actualizado

```
1:  function MERGEITER(I, B_0,B_1,offsets)
2:      ISig ← [null] * len(I)$
3:      actualPos0 ← 0
4:      actualPos1 ← 0
5:      tempIndex ← offsets
6:      for todo b en I do
7:          if b == 0 then
8:              c ← B_0[actualPos0]
9:              actualPos0 ← actualPos0 + 1
10:         else
11:             c ← B_1[actualPos1]
12:             actualPos1 ← actualPos1 + 1
13:         ISigPos ← tempIndex[c]
14:         ISig[ISigPos] ← b
15:         tempIndex[c] ← tempIndex[c] + 1
16:     return ISig
```

El algoritmo 5 realiza una iteración del intercalado para dos cadenas BWT. Para aplicar esta función a una mezcla completa se requiere comprobar la convergencia, el algoritmo 6 presenta dicha comprobación.

La BWT es la representación de un ordenamiento de sufijos. Dada una cadena BWT esta puede ser reordenada, utilizando un ordenamiento radix para recuperar los sufijos de longitud 1 en orden lexicográfico. Si la BWT se antepone a los sufijos ordenados de longitud 1 y se ordena de nuevo utilizando solo los caracteres BWT

**Algorithm 6** Intercalado

**Recibe:** Las cadenas BWT $B_0$ y $B_1$ y el alfabeto $\Sigma$

**Retorna:** La convergencia del intercalado $I$

1: **function** BWTMERGE($B_0$,$B_1$,$\Sigma$)
2: $off \leftarrow 0\$$
3: **for** todo $c$ en $\Sigma$ **do**
4: $totals[c] \leftarrow cuenta(c, B_0) + cuenta(c, B_1)$
5: $offsets[c] \leftarrow off$
6: $off \leftarrow off + totals[c]$
7: $I \leftarrow null\$$
8: $ret \leftarrow [0] * len(B_0) + [1] * len(B_1)\$$
9: **while** $ret \neq I$ **do**
10: $I \leftarrow ret\$$
11: $ret \leftarrow mergeIter(I, B_0, B_1, offsets)\$$
12: **return** ret

antepuestos, todos los sufijos de longitud 2 en el BWT se recuperan en orden lexicográfico. Si este proceso se repite para $i$ iteraciones, los sufijos de longitud $i$ en la BWT pueden ser recuperados. Nótese que este procedimiento realiza un ordenamiento radix en los símbolos menos significantes de todos los sufijos de longitud $i$. Como el algoritmo realiza el ordenamiento de los prefijos de los sufijos al incrementar la longitud del prefijo, en el fondo, se realiza un ordenamiento radix del símbolo más significativo en los sufijos completos.

Las iteraciones de el ciclo *while* en el algoritmo 6 es equivalente a realizar este ordenamiento radix. El arreglo $I$ indica el intercalado actual de los símbolos al inicio de una iteración. Los bits son ubicados en la siguiente posición disponible para su correspondiente símbolo de acuerdo al arreglo $tempIndex$. Para cada iteración en el ciclo, el ordenamiento de los sufijos es extendido en un símbolo hasta que $I$ converge a un intercalado correcto. Los sufijos actuales nunca son explícitamente reconstruidos o almacenados. La tabla 4.1 muestra los estados para cada iteración $i$, para mezclar dos cadenas; $ACCA\$$ y $CAAA\$$. Las respectivas cadenas BWT son $AC\$CA$ y $AAAC\$$, el alfabeto de entrada es $\Sigma = \{\$, A, C, G, T\}$ y la salida es el vector de intercalado $I$ que representa la cadena $AACAAC\$C\$A$.

Considerando la condición inicial $k = 0$. Todos los sufijos de longitud 0 son idénticos y el ordenamiento es de ceros seguidos de unos. Considerando la iteración $k = 1$ donde el intercalado $I$, es un ordenamiento estable de los primeros $i$ símbolos de los sufijos de los correspondientes BWT. En la siguiente iteración, el algoritmo realiza un recorrido de $I$ retornando el símbolo predecesor correspondiente para cada bit en $I$. Si dos sufijos tienen diferentes símbolos iniciales, entonces los sufijos son automáticamente ordenados correctamente porque cada uno será colocado en la posición correcta. Todos los sufijos $(i + 1)$ iniciando con el mismo símbolo, $c$, son colocados secuencialmente en la salida en orden lexicográfico. Dado que los $i$ sufijos

Tabla 4.1: Mezcla de dos cadenas BWT

| Iteración0 | | | Iteración1 | | | Iteración2 | | | Iteración3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | S | B | I | S | B | I | S | B | I | S | B |
| 0 | | A | 0 | $ | A | 0 | $A | A | 0 | $AC | A |
| 0 | | C | 1 | $ | A | 1 | $C | A | 1 | $CA | A |
| 0 | | $ | 0 | A | C | 0 | A$ | C | 0 | A$A | C |
| 0 | | C | 0 | A | $ | 1 | A$ | A | 1 | A$C | A |
| 0 | | A | 1 | A | A | 1 | AA | A | 1 | AA$ | A |
| 1 | | A | 1 | A | A | 1 | AA | C | 1 | AAA | C |
| 1 | | A | 1 | A | C | 0 | AC | $ | 0 | ACC | $ |
| 1 | | A | 0 | C | C | 0 | CA | C | 0 | CA$ | C |
| 1 | | C | 0 | C | A | 1 | CA | $ | 1 | CAA | $ |
| 1 | | $ | 1 | C | $ | 0 | CC | A | 0 | CCA | A |

se encuentran ordenados, si el símbolo $c$ se encuentra en dos índices, $x$ y $y$ donde $x < y$, entonces los sufijos de longitud $(i+1)$ correspondientes deben ser de la forma $cX$ y $cY$ donde $X$ es un sufijo de longitud $i$ que lexicográficamente preceden a los otros sufijos de longitud $i$ de $Y$. Lo que implica que los sufijos de longitud $(i+1)$ correspondientes también se encuentren ordenados.

Se sabe que luego de $(k+1)$ iteraciones, todos los sufijos de longitud $k+1$ serán lexicográficamente ordenados. Si las subcadenas comunes más largas $(LCS)$ son de longitud $k$, entonces esto indica que luego de $k+1$ iteraciones los sufijos serán ordenados hasta los primeros $(k+1)$ símbolos. En las siguientes iteraciones no habrá cambios en el ordenamiento de los sufijos de tal modo que se detecta la convergencia del intercalado. El algoritmo detecta la convergencia luego de $LCS+1$ iteraciones máximo.

La complejidad espacial de este algoritmo es $O(N)$, ya que utiliza $O(N)$ bits de

memoria pertenecientes a la creación de $I$ e $ISig$. Para un alfabeto de tamaño fijo se utilizan 5 variables de tamaño constante para almacenar el histograma acumulado. Por motivos prácticos, las cadenas $(B_0, B_1)$ son almacenados en disco. El algoritmo puede ser extendido para mezclar más de dos cadenas utilizando un vector de bits para distinguir el origen de cada cadena BWT para cada posición en la BWT intercalada. La forma más simple de lograr esto es extender el arreglo $I$ para utilizar múltiples bits, permitiendo mezclar múltiples cadenas BWT al mismo tiempo. Esta extensión tiene un costo $O(N * \log(F))$ bits para almacenar el arreglo $I$.

El rendimiento de este algoritmo no es afectado directamente por la longitud de la cadena o el número de cadenas. Para una constante $N$, incrementar la longitud de las cadenas o disminuir el número de cadenas no afectará el tiempo de ejecución a menos que también se incremente la $LCS$ entre dos cadenas BWT. El algoritmo requiere solo $O(N)$bits para el intercalado. Dado lo anterior el tiempo de ejecución para este algoritmo es $O(LCS * N)$

En este trabajo se propone modificar este algoritmo para dividir el número de iteraciones requeridas para la convergencia entre $n$, a cambio de incrementar el total de memoria requerida para almacenar el índice. Cada paso es computacionalmente más complejo a medida que se incrementa $n$ y la mejora temporal es apenas notoria. Esta mejora se basa en el trabajo de [Chacon et al., 2013], donde se propone una modificación similar al algoritmo utilizado para las búsquedas utilizando el índice FM.

Se propone modificar el alfabeto utilizado como entrada de este algoritmo, reemplazando los símbolos por los sufijos de longitud $n$. Con este objetivo se debe reemplazar el alfabeto $\Sigma$ por la permutación los símbolos en cadenas de longitud $n$. Por ejemplo, para un alfabeto $\Sigma' = \{$ \$\$, \$$A$, \$$C$, \$$G$, \$$T$, $A$\$, $AA$, $AC$, $AG$, $AT$, $C$\$, $CA$, $CC$, $CG$, $CT$, $G$\$, $GA$, $GC$, $GG$, $GT$, $T$\$, $TA$, $TC$, $TG$, $TT$ $\}$.

La BWT de una cadena R, es una permutación de símbolos en R que permite

realizar operaciones de búsqueda como en un árbol de sufijos, utilizando una operación llamada mapeo LF. Esta misma operación nos permitirá calcular los sufijos necesarios. El mapeo LF es una operación definida como la suma de dos funciones de conteo $C() + Occ()$. La función $C(B, s)$ cuenta el número de ocurrencias en $B$ de los símbolos que son lexicográficamente menores que el símbolo $s$. La segunda función, $Occ(B, s, p)$ cuenta el número de veces que un símbolo $s$ aparece en $B$ antes de la posición $p$.

**Algorithm 7** Nuevas cadenas pseudo BWT

**Recibe:** Las cadenas BWT y el número $n$ de pasos por iteración

**Retorna:** Las nuevas cadenas pseudo BWT

```
1:  function NuevasBWTs(BWTs,pasos)
2:      nuevasBWT[long(BWTs)]
3:      for i = 0 hasta long(BWTs) do
4:          offsets ← CalculaOffsets(BWTs[i])
5:          for j = 0 hasta long(BWTs[i]) do
6:              c ← BWTs[i][j]
7:              nchar ← BWTs[i][j]
8:              for s = pasos hasta 0 do
9:                  nchar[s] ← c
10:                 c ← off[c] + Occ(BWTs[i], j)
                nchar.inserta(nchar)
11:     return nuevasBWT
```

El algoritmo 7 calcula las cadenas pseudo BWT que se basan en un alfabeto de tamaño $5^n$, siendo 5 la longitud base de nuestro alfabeto. Cada símbolo de nuestras nuevas cadenas será compuesto por los sufijos de longitud $n$ pertenecientes a nuestra cadena original, lo que nos permite calcular de forma simultánea cada símbolo de las nuevas pseudo BWT. La salida de este algoritmo será la entrada del algoritmo original y nos permite disminuir el número de iteraciones requeridas para alcanzar

la convergencia del vector $I$.

Tabla 4.2: Mezcla de dos cadenas BWT con dos pasos por iteración

| Iteración0 | | | Iteración1 | | | Iteración2 | | |
|---|---|---|---|---|---|---|---|---|
| I | S | B | I | S | B | I | S | B |
| 0 | | CA | 0 | $A | CA | 0 | $ACC | CA |
| 0 | | CC | 1 | $C | AA | 1 | $CAA | AA |
| 0 | | A$ | 0 | A$ | CC | 0 | A$AC | CC |
| 0 | | AC | 1 | A$ | AA | 1 | A$CA | AA |
| 0 | | $A | 1 | AA | CA | 1 | AA$A | CA |
| 1 | | AA | 1 | AA | $C | 1 | AAA$ | $C |
| 1 | | AA | 0 | AC | A$ | 0 | ACCA | A$ |
| 1 | | CA | 0 | CA | AC | 0 | CA$C | AC |
| 1 | | $C | 1 | CA | A$ | 1 | CAAA | A$ |
| 1 | | A$ | 0 | CC | $A | 0 | CCA$ | $A |

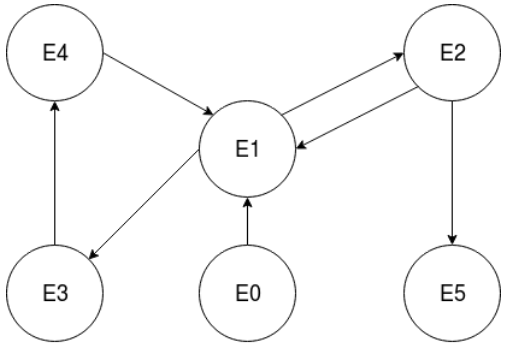
La tabla 4.2 muestra un ejemplo de la modificación algorítmica utilizando dos pasos por iteración para mezclar las cadenas BWT; $AC\$CA$ y $AAAC\$$. En esta tabla podemos apreciar que se requiere un mínimo de dos iteraciones para detectar la convergencia. El resultado es el mismo vector obtenido en la tabla 4.1

### 4.2.2. Funcionamiento de la arquitectura

La arquitectura hardware propuesta para la implementación que genera la mezcla de cadenas BWT con las modificaciones mencionadas al algoritmo requiere tres etapas; la primera consiste en la creación de nuevas cadenas utilizando los nuevos símbolos compuestos de prefijos de longitud $n$, en la segunda la actualización de los índices y su comprobación de convergencia. La tercera consiste en la actualización del índice dados los símbolos en las nuevas cadenas y de las posiciones de éstos dadas

por el vector de $offsets$.

Esta arquitectura fue pensada para utilizar el mismo tipo de memoria que la presentada en la figura 4.2. Esta arquitectura utiliza el procesador de propósito general para copiar los datos de entrada y resultados a la memoria HMC mediante la interfaz PCIe. El controlador de memoria HMC administra los datos.




E0: Calcula nuevos símbolos
E1: Actualiza índice
E2: Verifica convergencia
E3: Lee símbolos en I
E4: Actualiza *offsets*
E5: Señal de alto

Figura 4.5: Máquina de estados de la arquitectura hardware para mezclar cadenas BWT.

La figura 4.5 muestra la máquina de estados finitos que describe el comportamiento de la arquitectura hardware propuesta para el algoritmo de mezcla de cadenas BWT. El estado E0, calcula los nuevos símbolos que utiliza la arquitectura basándose en los sufijos de longitud $n$ y con ellos crea nuevas cadenas pseudo BWT. En el estado E1, la arquitectura actualiza los índices que posteriormente serán comprobados en el estado E2 de la arquitectura para verificar la convergencia. Dentro del estado E3 se toma cada símbolo de las cadenas BWT en el orden indicado por el índice $I$. El estado E4 inserta los símbolos leídos por el estado E3 en las posiciones indicadas por el vector de $offsets$ y actualiza las posiciones. El último estado, E5, indica que la arquitectura ha finalizado el cálculo correctamente.

La figura 4.6 describe el diagrama a bloques de la arquitectura hardware. Requiere de BRAMs para actualizar datos en el vector de $offsets$. Se alimenta a los

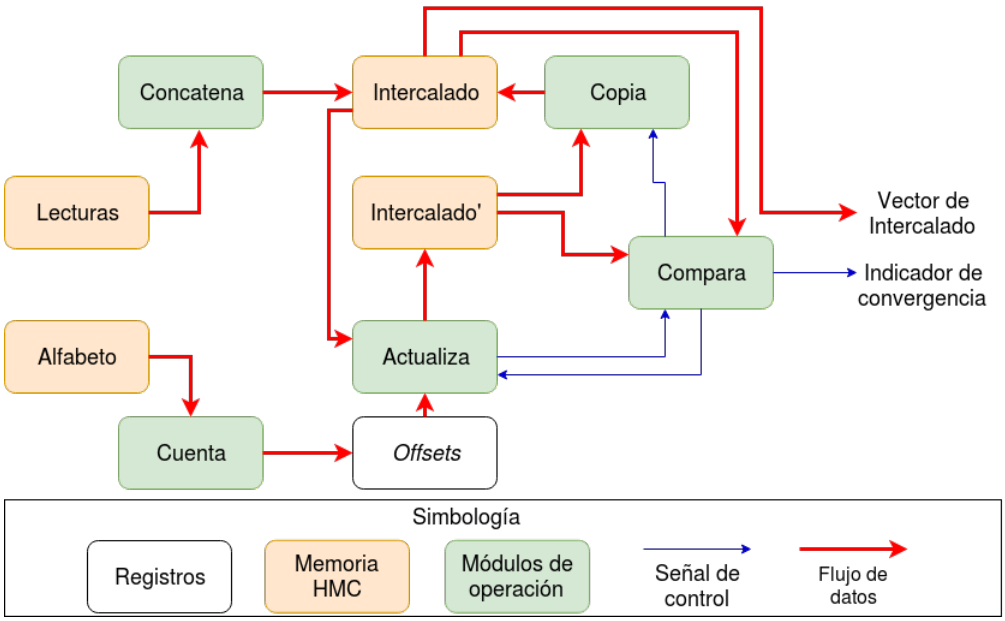



Figura 4.6: Diagrama de alto nivel del diseño de la arquitectura propuesta para la mezcla de cadenas BWT.

módulos *Concatena* y *Cuenta*, que realizan el primer intercalado y el conteo del histograma acumulado respectivamente, retornan palabras de 32-bits. El módulo *Actualiza* escribe palabras de 32-bits en el vector $I'$ y lee palabras de 32-bits del vector de intercalado $I$, ambos almacenados en la memoria HMC, envía una señal al término de su operación y recibe una señal de restablecimiento del módulo *Compara*. El módulo de comparación recibe una señal del módulo de actualización para poder leer palabras de 32-bits de los vectores de intercalado. En caso de no detectar la convergencia envía una señal al módulo *Copia* que inicia un proceso de copiado de palabras de 32-bits del vector $I'$ al vector $I$, en caso contrario se envía la señal de alto y se recupera el vector $I$ de la memoria HMC.

La figura 4.7 describe el diagrama a bloques de la arquitectura hardware. Esta arquitectura toma datos de memoria para realizar las dos operaciones básicas. El módulo *Occ*, toma datos en palabras de 8-bits y comunica palabras de 32-bits, de forma similar el segundo módulo *Cuenta*, escribe sus resultados en una BRAM para

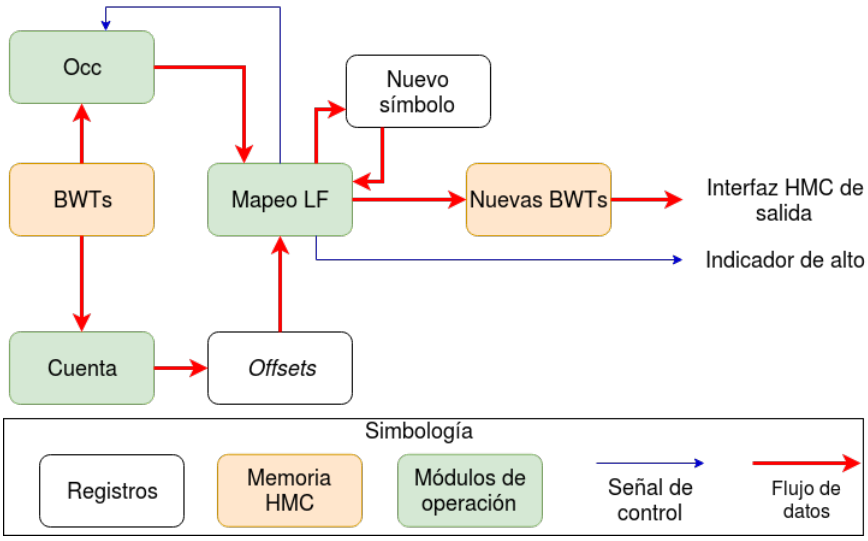



Figura 4.7: Diagrama de alto nivel del diseño de la arquitectura propuesta para la modificación al algoritmo de mezcla.

que el módulo *Mapeo LF* pueda determinar los nuevos símbolos de las cadenas pseudo BWT, mismo que se escribe en un BRAM en palabras de 32-bits, este mismo escribe los nuevos símbolos en las nuevas cadenas en palabras de 32-bits y envía la señal de alto.

## 4.3. Resumen

En esta sección se presentaron los algoritmos que fueron utilizados como base para el diseño de la arquitectura hardware que acelera la etapa de construcción del índice que utiliza el ensamblador SGA. También se propone una mejora para el algoritmo presentado en [Holt, 2014] para disminuir el número de iteraciones requeridas para la convergencia a cambio de incrementar el uso de memoria y requerir un preprocesamiento que es fácilmente paralelizable. En la siguiente sección se describen los resultados obtenidos de la síntesis de los diseños propuestos en este capítulo para ambos algoritmos así como el desempeño de la modificación algorítmica.

# Capítulo 5

# Experimentos y resultados

Dentro de este capítulo se detallan los experimentos, resultados y evaluaciones realizadas. La primera sección describe el diseño experimental utilizado con base en las métricas de evaluación, la plataforma de experimentación, así como los datos que se utilizan en los experimentos. Dentro de la segunda sección se incluye una revisión de las métricas utilizadas, en las que se incluyen el tiempo de ejecución y el área utilizadas, en la misma sección. Por último se analizan los resultados obtenidos de los tres métodos implementados como arquitectura hardware, estos se comparan con otras arquitecturas que tratan el problema de de ensambles de secuencias de ADN, de igual forma se compara con el software en el que se encuentra basada la arquitectura.

## 5.1. Métricas de evaluación

Las métricas comúnmente utilizadas para evaluar arquitecturas hardware son el rendimiento y el área. En casos más particulares se utilizan métricas adicionales. Para arquitecturas de lógica secuencial, la complejidad temporal se calcula a partir de los ciclos de reloj que necesita la arquitectura completa. En arquitecturas de

lógica combinacional, la complejidad temporal se puede determinar con base en la frecuencia máxima de operación que se determina por el máximo retraso en la ruta combinacional, como es el caso de estas arquitecturas.

El área utilizada indica los recursos de hardware que utiliza el diseño. Debido a la complejidad y a las necesidades de cada arquitectura, no hay una métrica estándar para medir la cantidad de recursos que una arquitectura requiere de un FPGA. En el diseño e implementación de una arquitectura en un dispositivo FPGA, la síntesis, espacio y reportes de ruta sobre el uso de recursos utilizados indican sus requerimientos, en particular, algunos de los datos más importantes son: el número de bloques lógicos configurables *slices*, número de $flipflops$, número de LUT's, número de relojes, bloques de RAM, multiplicadores, entre muchos otros. Dependiendo de la aplicación, algunas arquitecturas solo reportan resultados temporales de la velocidad de ejecución.

Como se mencionó anteriormente, arquitecturas hardware se pueden reportar con respecto al número de $LUT's$ o bien al número de *slices*. Esto no es del todo correcto para realizar comparaciones, debido a que algunas arquitecturas utilizan bloques del FPGA dedicados a realizar ciertas operaciones. Lo anterior conlleva a utilizar una menor área de hardware que otras arquitecturas que realizan operaciones distintas, que serán implementadas en la sección lógica programable y no utilizaría los bloques dedicados, por tal motivo se consume mayor área.

Es conocido que el mismo código HDL sintetizado para diferentes modelos de FPGA's del mismo fabricante logran distintos resultados de área. Esta diferencia se acentúa cuando se sintetiza para FPGA's de diferentes fabricantes. Por estos motivos estas características no son relevantes como métricas de evaluación. Por tal, las comparaciones de área solo deberían realizarse si se comparan arquitecturas implementadas en FPGA's de las mismas características.

### 5.1.1. Plataformas de experimentación

Las arquitecturas hardware presentadas son diseñadas para hacer uso intensivo de memoria dada la gran cantidad de datos de entrada en el tratamiento del problema. Con estos requisitos la arquitectura fue diseñana para funcionar en un prototipo de FPGA unida a una memoria HMC mediante un controlador. Desafortunadamente este prototipo se encuentra en desarrollo y actualmente no fue posible completar la implementación en dicha plataforma, esto se discute con más detalle en el siguiente capítulo. Por tales motivos se seleccionó un FPGA con menor capacidad pero con mayor documentación, la realización de los experimentos se utilizó la tarjeta PYNQ, esta cuenta con un procesador ARM Cortex A-9 que se utiliza para copiar los datos almacenados desde la tarjeta Micro SD hacia la memoria RAM DDR3, la cual cuenta con una capacidad de 512MB. El FPGA que integra esta tarjeta es el XC7Z020CLG400-1 [pyn, 2017], el cual cuenta con 1.3 millones de compuertas configurables. Las implementaciones en software han sido probadas en una computadora que cuenta con un procesador Intel i7-3770K a una frecuencia máxima 3.90 GHz y 32 GB de memoria RAM DDR3 en un sistema Fedora 25 con el kernel de linux 4.13.13-100.

### 5.1.2. Bases de datos utilizadas para la evaluación

En la literatura se han utilizado diferentes bases de datos para probar la funcionalidad de las arquitecturas hardware y software. Estas bases cambian un poco dependiendo del problema específico que ataquen, por ejemplo, se han utilizado bases de datos con lecturas de diferentes longitudes, siendo conjuntos de lecturas con una longitud única, ya sea de pares de bases cortas o largas, o bien utilizar conjuntos de lecturas con longitudes variables. Estas bases de datos pueden contener lecturas procesadas con la BWT como es en el caso del algoritmo presentado en [Holt, 2014]

o lecturas sin procesar para generar la BWT de todas las lecturas como el algoritmo presentado en [Li, 2014].

| Datos | Símbolos | Utilizado en |
|---|---|---|
| E. Coli 5m emparejado | 25518957 | [ANGUS, 2017] |
| Venter | 420603376 | [Li, 2014] |
| E. Coli MG1655 | 870876856 | [Simpson, 2011] |
| E. Coli 5m recortado | 319458930 | [ANGUS, 2017] |

Tabla 5.1: Bases de datos utilizadas en los experimentos.

En la tabla 5.1 se presentan las bases de datos utilizadas para cada experimento las cuales fueron tomadas de los respectivos trabajos donde fueron utilizadas, esto se realiza para tener una forma de comparar los resultados con los trabajos relacionados de manera justa. Estas bases fueron seleccionadas por ser pequeñas en relación a las otras presentadas en esos artículos, este hecho no afecta los resultados presentados como se puede ver en los resultados del trabajo [Holt, 2014]. Las lecturas que contienen bases ambiguas son retiradas del conjunto de pruebas tal como lo hace el trabajo [Li, 2014].

## 5.2. Experimento: Arquitectura para el algoritmo ropeBWT para la generación de cadenas BWT

### Descripción

La arquitectura hardware se evalúa utilizando el *throughput* y el área utilizada. La evaluación del área se realiza utilizando el reporte de uso de hardware que proporciona el sintetizador de la herramienta Vivado HLS. El *throughput* de la arquitectura se compara con el *throughput* de la implementación en software de la herramienta publicada en [Li, 2014].

### Metodología

El *throughput* considera el tiempo de uso del procesador de la PC y de la implementación de la arquitectura hardware. En las pruebas de rendiemiento, el conjunto de datos descrito en la sección anterior es utilizado para evaluar la arquitectura hardware y compararla con el algoritmo en software. Dado que la memoria RAM en la tarjeta de prueba es limitada, solo se realizan las pruebas con un número limitado de lecturas de modo que puedan ser transferidas a la memoria de la tarjeta y quede espacio suficiente para almacenar el resultado. En este mismo experimento se forman los índices RLO y RCLO, dado que estas operaciones no tienen mayor complejidad que leer las cadenas en orden inverso, leerlas en orden inverso y cambiar el símbolo leído por su complemento. Los resultados de la arquitectura se validan con la herramienta publicada en [Li, 2014] para verificar que las salidas sean idénticas.

### Resultados

En la tabla 5.2 se presentan los resultados de uso de recursos obtenidos al momento de sintetizar la arquitectura, esta debe utilizar también la comunicación de *Interfaz Avanzada eXtensible AXI* para poder utilizar su interfaz de Acceso Directo a Memoria *DMA* y realizar la transferencia de datos de entrada y salida, por tal motivo se consideran también estos recursos como parte de la arquitectura.

En la tabla 5.3 se reporta el *throughput* de la arquitectura diseñada contra el *throughput* de la implementación en software presentada por el trabajo original.

### Conclusiones

En la tabla 5.2 se obtiene un bajo uso de recursos respecto a los disponibles por la tarjeta PYNQ por lo cual se pueden crear múltiples instancias de la arquitectura

| Nombre | BRAM | DSP48E | FF | LUT |
|---|---|---|---|---|
| *DSP* | - | - | - | - |
| Expresión | - | - | 0 | 1180 |
| *FIFO* | - | - | - | - |
| Instancia | 2 | - | 1869 | 3715 |
| Memoria | - | - | - | - |
| Multiplexor | - | - | - | 1230 |
| Registro | - | - | 1515 | - |
| Total | 2 | - | 3384 | 6125 |
| Disponible | 280 | 220 | 106400 | 53200 |
| Uso ( %) | 0 | 0 | 3 | 11 |

Tabla 5.2: Recursos hardware utilizados por la arquitectura para el algoritmo ropeBWT.

para tratar el problema de forma paralela. Aunque esto se ve limitado por el uso de la interfaz *AXI DMA*, la cual genera un cuello de botella que se puede evitar implementando la arquitectura en otra tecnología que cuente con un mayor ancho de banda en la interfaz de memoria.

En la tabla 5.3 se observa una mejora en el *throughput* con respecto a la implementación software, la cual es 12 veces más rápida, ya que puede procesar más pares de bases con respecto al software. Para realizar los experimentos se tomó una cantidad fija de 185961 lecturas para que pueda ser cargada a la memoria de la tarjeta de pruebas, estas lecturas tienen diferentes longitudes dependiendo de la base de datos.

| Datos | *throughput* Software | *throughput* Hardware |
|---|---|---|
| E. Coli 5m emparejado | 2.56 MB/s | 31.48 MB/s |
| E. Coli 5m emparejado (RLO) | 1.98 MB/s | 24.37 MB/s |
| E. Coli 5m emparejado (RCLO) | 0.89 MB/s | 10.94 MB/s |
| Venter | 1.79 MB/s | 22.05 MB/s |
| Venter (RLO) | 1.79 MB/s | 22.03 MB/s |
| Venter (RCLO) | 0.80 MB/s | 9.84 MB/s |
| E. Coli MG1655 | 4.02 MB/s | 49.53 MB/s |
| E. Coli MG1655 (RLO) | 3.58 MB/s | 44.08 MB/s |
| E. Coli MG1655 (RCLO) | 1.69 MB/s | 20.78 MB/s |
| E. Coli 5m recortado | 4.63 MB/s | 57.00 MB/s |
| E. Coli 5m recortado (RLO) | 3.77 MB/s | 46.41 MB/s |
| E. Coli 5m recortado (RCLO) | 1.72 MB/s | 21.17 MB/s |

Tabla 5.3: Resultados de ejecución de la arquitectura hardware ropeBWT.

### 5.2.1. Experimento: Mezcla de cadenas BWT en *n* pasos

#### Descripción

Para validar la medida de éxito de este experimento se utiliza el tiempo de ejecución del algoritmo original contra el tiempo de ejecución del algoritmo modificado, incluyendo el tiempo que toma crear la pseudo BWT con el nuevo alfabeto. También se toma en cuenta el espacio requerido en memoria para almacenar el histograma acumulado del nuevo alfabeto y de la pseudo BWT respecto al algoritmo original.

#### Metodología

En este experimento se toma la misma base de datos 5.1 para comparar la ejecución. En lugar de comparar las longitudes promedio de los símbolos codificados, se compara el tiempo de ejecución y el número de iteraciones requeridas para alcanzar

la convergencia.

La tabla 5.1 muestra la base de datos de las muestras de ADN utilizadas y el número de símbolos que contiene cada muestra. En este caso se utiliza la base de datos completa en cada ejecución debido a que los datos completos pueden ser cargados a memoria RAM.

### Resultados

| Datos ($n$) | Algoritmo original | | | Algoritmo modificado | | |
|---|---|---|---|---|---|---|
| | Tiempo | RAM | Iteraciones | Tiempo | RAM | Iteraciones |
| E. Coli 5m emparejado (1) | 192.69s | 80B | 75 | 192.169s | 80B | 75 |
| E. Coli 5m emparejado (2) | - | - | - | 98.45s | 240B | 38 |
| E. Coli 5m emparejado (3) | - | - | - | 70.18s | 1040B | 26 |
| E. Coli 5m emparejado (4) | - | - | - | 57.68s | 5040B | 20 |
| E. Coli 5m emparejado (5) | - | - | - | 48.91s | 25.04KB | 16 |
| E. Coli 5m emparejado (6) | - | - | - | 45.92s | 125.04KB | 14 |
| Venter (1) | 145.75m | 80B | 83 | 145.75m | 80B | 83 |
| Venter (2) | - | - | - | 65.56m | 240B | 42 |
| Venter (3) | - | - | - | 45.41m | 1040B | 29 |
| Venter (4) | - | - | - | 35.44m | 5040B | 22 |
| Venter (5) | - | - | - | 19.63m | 25.04KB | 18 |
| Venter (6) | - | - | - | 19.5m | 125.04KB | 15 |

Tabla 5.4: Resultados de ejecución de la implementación en software de la mezcla de cadenas BWT en *n* pasos, parte 1 de 2.

| Datos ($n$) | Algoritmo original | | | Algoritmo modificado | | |
|---|---|---|---|---|---|---|
| | Tiempo | RAM | Iteraciones | Tiempo | RAM | Iteraciones |
| E. Coli MG1655 (1) | 608.03m | 80B | 153 | 608.03m | 80B | 153 |
| E. Coli MG1655 (2) | - | - | - | 277.79m | 240B | 77 |
| E. Coli MG1655 (3) | - | - | - | 192.33m | 1040B | 52 |
| E. Coli MG1655 (4) | - | - | - | 142.93m | 5040B | 39 |
| E. Coli MG1655 (5) | - | - | - | 118.63m | 25.04KB | 32 |
| E. Coli MG1655 (6) | - | - | - | 102.53m | 125.04KB | 27 |
| E. Coli 5m recortado (1) | 78.85m | 80B | 72 | 78.85m | 80B | 72 |
| E. Coli 5m recortado (2) | - | - | - | 40.88m | 240B | 37 |
| E. Coli 5m recortado (3) | - | - | - | 27.89m | 1040B | 25 |
| E. Coli 5m recortado (4) | - | - | - | 21.64m | 5040B | 19 |
| E. Coli 5m recortado (5) | - | - | - | 18.59 | 25.04KB | 16 |
| E. Coli 5m recortado (6) | - | - | - | 18.26m | 125.04KB | 13 |

Tabla 5.5: Resultados de ejecución de la implementación en software de la mezcla de cadenas BWT en *n* pasos, parte 2 de 2.

#### Conclusiones

En las tablas 5.4 y 5.5 se muestra la ventaja que alcanza la modificación propuesta sobre el algoritmo original, a pesar del costo adicional que existe al calcular el mapeo-LF para cada lectura y el incremento de consumo de memoria RAM, no presenta una desventaja ante los tiempos y las iteraciones requeridas por el algoritmo original. Es importante notar que para $n$ mayor a 5 la mejora temporal no presenta una ventaja significativa, aunque el consumo de memoria seguirá incrementándose exponencialmente. Se puede afirmar que para $n$ en el rango de 2 a 4 se presenta la mejor relación costo-beneficio respecto al consumo de memoria y la disminución de iteraciones.

### 5.2.2. Experimento: Arquitectura para la mezcla de cadenas BWT

#### Descripción

En esta arquitectura hardware la medida de éxito se valida de igual manera con el *throughput* y el área utilizada. El reporte de uso del número de compuertas configurables es proporcionado por el sintetizador de la herramienta Vivado HLS. El *throughput* de la arquitectura se compara con el *throughput* de la implementación en software realizada a partir del algoritmo del trabajo publicado en [Holt, 2014]. Esta implementación fue verificada a partir de los resultados obtenidos de la herramienta publicada en [Li, 2014], los resultados de la arquitectura son verificados con la misma herramienta a modo de validación.

#### Metodología

Al igual que en la implementación de la arquitectura para el algoritmo ropeBWT, este experimento se ejecutó con un conjunto limitado de las bases de datos presentadas en la tabla 5.1. En las pruebas de rendimiento, el conjunto de datos

descrito en la sección anterior es utilizado para evaluar la arquitectura hardware y comparar el *throughput* con el algoritmo implementado en software.

**Resultados**

| Nombre | BRAM | DSP48E | FF | LUT |
|---|---|---|---|---|
| *DSP* | - | - | - | - |
| Expresión | - | - | 0 | 1928 |
| *FIFO* | - | - | - | - |
| Instancia | 2 | - | 776 | 1004 |
| Memoria | 6 | - | 0 | 0 |
| Multiplexor | - | - | - | 996 |
| Registro | 0 | - | 2151 | 160 |
| Total | 8 | 0 | 2927 | 4088 |
| Disponible | 280 | 280 | 106400 | 53200 |
| Uso ( %) | 2 | 2 | 2 | 7 |

Tabla 5.6: Recursos de hardware utilizados por la arquitectura hardware para el algoritmo de mezcla de cadenas BWT.

En la tabla 5.6 se presenta el uso de recursos utilizados por la arquitectura hardware al momento de sintetizarla, esta también debe utilizar la *Interfaz Avanzada eXtensible AXI* para poder leer y escribir los datos mediante el Acceso Directo a Memoria *DMA*, por tal motivo también se consideran estos recursos como parte de la arquitectura.

En la tabla 5.7 se reporta el *throughput* de la arquitectura diseñada contra el *throughput* de la implementación en software realizada con base en el algoritmo presentado en el trabajo de [Holt, 2014].

| Datos | *throughput* del Software | *throughput* del Hardware |
|---|---|---|
| E. Coli 5m emparejado | 0.13 MB /s | 2.05 MB/s |
| Venter | 0.04 MB/s | 0.63 MB/s |
| E. Coli MG1655 | 0.02 MB/s | 0.31 MB/s |
| E. Coli 5m recortado | 0.06 MB /s | 0.32 MB/s |

Tabla 5.7: Resultados de ejecución de la arquitectura hardware de la mezcla de cadenas BWT.

#### Conclusiones

Los resultados obtenidos por esta aquitectura obtienen un *throughput* inferior a la arquitectura propuesta para el algoritmo ropeBWT, sin embargo, estos algoritmos requieren diferentes datos de entrada. La arquitectura propuesta obtiene un *throughput* 15 veces más rápido a la implementación en software del algoritmo original.

### 5.2.3. Experimento: Arquitectura para la mezcla de cadenas BWT en *n* pasos

#### Descripción

Esta arquitectura se basa en realizar un preprocesamiento de las cadenas BWT de entrada, por lo que se agrega un módulo que realiza el mapeo-LF de cada símbolo de las cadenas, las cuales son independientes y se calculan en paralelo. El número de iteraciones requeridas para alcanzar la convergencia se divide entre $n$ y este a su vez incrementa exponencialmente el número de símbolos en el alfabeto. Por tal motivo el uso de memoria también crece como se muestra en las tablas 5.4 y 5.5. Como se puede apreciar en la tabla, el uso de recursos se incrementa y se encuentra un punto donde el número $n$ ya no presenta una mejora significativa. Con esta premisa

se limita el número $n$ en la arquitectura a 4 pasos por iteración.

**Metodología**

La arquitectura se prueba de igual manera que la arquitectura basada en el algoritmo de mezcla original. En esta prueba se compara con el algoritmo modificado. Los datos se cargan por partes para que puedan ser procesados en la tarjeta de pruebas.

**Resultados**

| Nombre | BRAM | DSP48E | FF | LUT |
|---|---|---|---|---|
| *DSP* | - | - | - | - |
| Expresión | - | 15 | 0 | 2608 |
| *FIFO* | - | - | - | - |
| Instancia | 2 | - | 922 | 1260 |
| Memoria | 96 | - | 128 | 6 |
| Multiplexor | - | - | - | 1475 |
| Registro | - | - | 3181 | - |
| Total | 98 | 15 | 4221 | 5349 |
| Disponible | 280 | 220 | 106400 | 53200 |
| Uso ( %) | 35 | 6 | 3 | 10 |

Tabla 5.8: Recursos de hardware utilizados por la arquitectura hardware para el algoritmo de mezcla de cadenas BWT en *n* pasos.

La tabla 5.8 presenta el uso de recursos requeridos por la arquitectura al momento de realizar su síntetisis. Ésta al igual que las arquirecturas anteriores requiere de la *Interfaz Avanzada eXtensible AXI* y la interfaz de Acceso Directo a Memoria

*DMA*. Se muestra que el uso de recursos es mayor que el obtenido en la arquitectura basada en el objetivo original, sin embargo, los recursos disponibles son mayores que los requeridos por la arquitectura. Esto hace posible implementar más instancias de la arquitectura y realizar esta tarea en paralelo.

| Datos $n$ | *throughput* del Software | *throughput* del Hardware |
|---|---|---|
| E. Coli 5m emparejado (2) | 0.26 MB/s | 4.23 MB/s |
| E. Coli 5m emparejado (3) | 0.37 MB/s | 6.06 MB/s |
| E. Coli 5m emparejado (4) | 0.45 MB/s | 7.40 MB/s |
| E. Coli 5m emparejado (5) | 0.53 MB/s | 8.75 MB/s |
| E. Coli 5m emparejado (6) | 0.57 MB/s | 9.44 MB/s |
| Venter (2) | 0.11 MB/s | 1.79 MB/s |
| Venter (3) | 0.15 MB/s | 2.46 MB/s |
| Venter (4) | 0.20 MB/s | 3.29 MB/s |
| Venter (5) | 0.24 MB/s | 3.96 MB/s |
| Venter (6) | 0.24 MB/s | 3.97 MB/s |

Tabla 5.9: Resultados de ejecución de la arquitectura hardware de la mezcla de cadenas BWT en *n* pasos, parte 1 de 2.

En las tablas 5.9 y 5.10 se reporta el *throughput* de la arquitectura diseñada contra el *throughput* de la implementación en software realizada con base en la modificación del algoritmo presentado en el trabajo de [Holt, 2014].

**Conclusiones**

Como se muestra en las tablas 5.9 y 5.10 podemos apreciar que existe una ventaja por parte de la arquitectura, la cual es capaz de procesar 16 veces más datos que la implementación en software. El uso de recursos presentado en la tabla 5.8 indica que es factible agregar módulos dedicados al cálculo de los nuevos símbolos para realizar esta tarea en paralelo.

| Datos $n$ | *throughput* del Software | *throughput* del Hardware |
|---|---|---|
| E. Coli MG1655 (2) | 0.05 MB/s | 0.81 MB/s |
| E. Coli MG1655 (3) | 0.07 MB/s | 1.148 MB/s |
| E. Coli MG1655 (4) | 0.10 MB/s | 1.64 MB/s |
| E. Coli MG1655 (5) | 0.12 MB/s | 1.98 MB/s |
| E. Coli MG1655 (6) | 0.14 MB/s | 2.31 MB/s |
| E. Coli 5m recortado (2) | 0.13 MB/s | 2.11 MB/s |
| E. Coli 5m recortado (3) | 0.19 MB/s | 3.11 MB/s |
| E. Coli 5m recortado (4) | 0.24 MB/s | 3.94 MB/s |
| E. Coli 5m recortado (5) | 0.28 MB/s | 4.62 MB/s |
| E. Coli 5m recortado (6) | 0.29 MB/s | 4.80 MB/s |

Tabla 5.10: Resultados de ejecución de la arquitectura hardware de la mezcla de cadenas BWT en *n* pasos, parte 2 de 2.

## 5.3. Conclusiones

En este capítulo se presentan los resultados y la evaluación de rendimiento. La evaluación se realizó retomando las bases de datos utilizadas en los trabajos de la literatura. Todas las arquitecturas propuestas fueron probadas bajo las mismas circunstancias al igual que la implementación en software del algoritmo propuesto. Las arquitecturas y el algoritmo se implementan utilizando la lógica secuencial presentada en el capítulo anterior. Debido al tiempo de desarrollo requerido por la plataforma *Pico Computing*, la cual aún se encuentra en desarrollo, se optó por ejecutar los experimentos en una tarjeta de pruebas más conocida y de desarrollo rápido.

A la fecha publicación de este trabajo no hay otras arquitecturas basadas en los algoritmos de [Holt, 2014, Li, 2014], por tal motivo no es posible realizar una com-

paración directa de las arquitecturas propuestas. Aunque existen otros trabajos que utilizan el índice-FM como [Arram et al., 2015, Arram et al., 2013a], una comparación específica en determinados módulos no sería justa para ninguna de las partes, ya que las arquitecturas se enfocan a tareas diferentes. Por estos motivos únicamente se realiza la comparación con las respectivas versiones de software.

# Capítulo 6

# Conclusiones y trabajo futuro

## 6.1. Antecedentes

En este trabajo se propone una estrategia para acelerar el ensamble de secuencias de ADN mediante las arquitecturas hardware presentadas para calcular la BWT de las lecturas. Estas abarcan dos enfoques retomados de los trabajos de [Holt, 2014] y [Li, 2014], el primero parte de lecturas que han sido previamente convertidas a cadenas BWT y genera una única cadena BWT ordenada lexicográficamente. El segundo toma lecturas sin procesar y genera las cadenas BWT, BWT RLO y BWT RCLO en el orden proporcionado. Ambos trabajos tienen la finalidad de construir un índice-FM para llevar tareas de emparejamiento tal como lo hace la metodología propuesta en [Simpson, 2010, Simpson, 2011], en la cual se propone un ensamblador que genera un grafo de cadenas utilizando los empalmes hallados mediante el índice-FM. En los resultados presentados en [Simpson, 2011], se demuestra que una de las tareas que requieren mayor tiempo de ejecución es la construcción de dicho índice. Con estas premisas se retomaron los trabajos mencionados y se realizaron las adaptaciones algorítmicas para el diseño de las arquitecturas hardware propuestas.

## 6.2. Revisión de objetivos

Los objetivos presentados en el primer capítulo de este trabajo han sido completados al haber realizado el diseño de una arquitectura hardware capaz de reducir el tiempo de la construcción del índice-FM en metodología SGA para el ensamble de secuencias de ADN, la cual utiliza el algoritmo de emparejamiento basado en el índice-FM que también fue tratado aunque no se completó en este trabajo.

## 6.3. Resumen y contribuciones

Se han propuesto tres arquitecturas hardware para acelerar la construcción del índice-FM en la tarea de ensamble de secuencias de ADN. Estas arquitecturas se basan principalmente en los trabajos presentados en la tabla 6.1. Las tres arquitecturas hardware propuestas aceleran la creación del índice-FM bajo dos esquemas diferentes, el primero toma cadenas BWT, cuyas lecturas han sido previamente convertidas con diferentes fines, por ejemplo, la compresión de bases de datos, genera una única BWT ordenada lexicográficamente con la finalidad de poder realizar la búsqueda de patrones y de este modo encontrar los empalmes. El segundo esquema genera la cadena BWT de las lecturas sin procesamiento, en un orden dado, por lo cual requiere un ordenamiento previo para obtener una cadena BWT de lecturas en orden lexicográfico. De este modo se logra abarcar la creación del índice-FM para ensamblar genomas que han sido procesados mediante la BWT sin necesidad de revertir dicha transformación, así como de genomas cuyas lecturas se encuentran sin ningún tipo de procesamiento. La mejora algorítmica propuesta fue diseñada para adaptarse a la capacidad de paralelismo que se puede alcanzar en una FPGA, sin embargo, este puede extenderse para realizar otro tipo de tareas como el reconocimiento de patrones, realizado de forma similar al trabajo presentado en [Chacon et al., 2013].

| Trabajo | Descripción el algoritmo | Contribución |
|---|---|---|
| [Holt, 2014] | Cálculo de una cadena BWT a partir de múltiples cadenas BWT. | Diseño de una arquitectura hardware para acelerar el algoritmo. |
| [Li, 2014] | Cálculo de cadena BWT a partir de lecturas. | Diseño de una arquitectura hardware para acelerar el cálculo de las cadenas BWT, BWT RLO y BWT RCLO. |
| [Simpson, 2011] | Ensamble de lecturas mediante la generación del índice-FM y grafos de cadenas. | Acelerar la construcción del índice-FM mediante las arquitecturas propuestas. |
| [Chacon et al., 2013] | Aceleración del reconocimiento de patrones mediante el uso del índice-FM y el mapeo-LF. | Adaptación del algoritmo presentado para acelerar el algoritmo propuesto en [Holt, 2014] mediante el uso del mapeo-LF. |
| Aceleración de [Holt, 2014] | Propuesta de modificación del algoritmo propuesto en [Holt, 2014]. | Diseño de una arquitectura hardware para acelerar la modificación propuesta. |

Tabla 6.1: Contribuciones

## 6.4. Trabajo futuro

Tomando en cuenta los resultados obtenidos en este trabajo, se identificó que existen diferentes aplicaciones algorítmicas del mapeo-LF para acelerar la búsqueda de patrones basadas en el índice-FM que no han sido reportadas. Estas se pueden aplicar, por ejemplo, dentro de la herramienta SGA para disminuir el tiempo requerido a la formación del grafo de cadena, entre otras herramientas existentes. Respecto a las arquitecturas hardware, éstas se han implementado realizando las tareas de forma secuencial, siendo posible llevar a cabo la paralelización.

Las arquitecturas hardware propuestas fueron diseñadas para implementarse en la plataforma *Pico Computing* de *Micron*, sin embargo, esta no se encuentra suficientemente madura como para realizar el desarrollo completo en el tiempo requerido para este trabajo. Una vez que la plataforma alcance la madurez necesaria se pueden migrar las arquitecturas propuestas y aprovechar el ancho de banda de la interfaz y la memoria HMC para acelerar las lecturas y escrituras que requieren estas tareas.

# Bibliografía

[NHG, 2016] (2016). National human genome research institute.

[pyn, 2017] (2017). *Zynq-7000 All Programmable SoC Data Sheet*. Xilinx, Inc. Rev. 1.11.

[Ahmed et al., 2014] Ahmed, M., Ahmad, I., and Ahmad, M. S. (2014). A survey of genome sequence assembly techniques and algorithms using high-performance computing. *J Supercomput*, 71(1):293–339.

[Andrews, 2016] Andrews, S. (2016). Fastqc a quality control tool for high throughput sequence data. Technical report, Babraham Bioinformatics Institute.

[ANGUS, 2017] ANGUS (2017). Assembling e. coli sequences with velvet.

[Arram et al., 2015] Arram, J., Pflanzer, M., Kaplan, T., and Luk, W. (2015). Fpga acceleration of reference-based compression for genomic data. *2015 International Conference on Field Programmable Technology (FPT)*.

[Arram et al., 2013a] Arram, J., Tsoi, K. H., Luk, W., and Jiang, P. (2013a). Hardware acceleration of genetic sequence alignment. *Lecture Notes in Computer Science*, pages 13–24.

[Arram et al., 2013b] Arram, J., Tsoi, K. H., Luk, W., and Jiang, P. (2013b). Reconfigurable acceleration of short read mapping. *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*.

[Bauer et al., 2011] Bauer, M. J., Cox, A. J., and Rosone, G. (2011). Lightweight bwt construction for very large string collections. *Combinatorial Pattern Matching*, 6661:219–231.

[Bauer et al., 2013] Bauer, M. J., Cox, A. J., and Rosone, G. (2013). Lightweight algorithms for constructing and inverting the bwt of string collections. *Theoretical Computer Science*, 483:134–148.

[Brankovic et al., 2016] Brankovic, L., Iliopoulos, C. S., Kundu, R., Mohamed, M., Pissis, S. P., and Vayani, F. (2016). Linear-time superbubble identification algorithm for genome assembly. *Theoretical Computer Science*, 609:374–383.

[Buermans, 2014] Buermans, H.P.J.den Dunnen, J. (2014). Next generation sequencing technology: Advances and applications. *Biochimica et Biophysica Acta (BBA) - Molecular Basis of Disease*, 1842(10):1932–1941.

[Burrows and Wheeler, 1994] Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. Technical report.

[Chacon et al., 2013] Chacon, A., Moure, J. C., Espinosa, A., and Hernández, P. (2013). n-step fm-index for faster pattern matching. *Procedia Computer Science*, 18:70 – 79.

[Chikhi et al., 2014] Chikhi, R., Limasset, A., Jackman, S., Simpson, J. T., and Medvedev, P. (2014). On the representation of de bruijn graphs. In *Lecture Notes in Computer Science*, pages 35–55. Springer Science + Business Media.

[Compeau et al., 2011] Compeau, P. E. C., Pevzner, P. A., and Tesler, G. (2011). How to apply de bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991.

[Danek et al., 2012] Danek, A., Pokrzywa, R., Makałowska, I., and Polański, A. (2012). *Application of the Burrows-Wheeler Transform for Searching for Ap-*

*proximate Tandem Repeats*, pages 255–266. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Ferragina, 2001] Ferragina, PaoloManzini, G. (2001). An experimental study of a compressed index. *Information Sciences*, 135(1-2):13–28.

[Ferragina, 2000] Ferragina, P.Manzini, G. (2000). Opportunistic data structures with applications. *Proceedings 41st Annual Symposium on Foundations of Computer Science*.

[Ferragina et al., 2011] Ferragina, P., Gagie, T., and Manzini, G. (2011). Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730.

[Goodarzi et al., 2014] Goodarzi, M., Houghten, S., and Liang, P. (2014). Effect of multi-k contig merging in de novo dna assembly. In *Bioinformatics and Bioengineering (BIBE), 2014 IEEE International Conference on*, pages 355–361.

[Goujon et al., 2010] Goujon, M., McWilliam, H., Li, W., Valentin, F., Squizzato, S., Paern, J., and Lopez, R. (2010). A new bioinformatics analysis tools framework at embl-ebi. *Nucleic Acids Research*, 38(Web Server):W695–W699.

[Greenstein et al., 2015] Greenstein, S., Holt, J., and McMillan, L. (2015). Short read error correction using an fm-index. In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pages 101–104.

[Holt, 2014] Holt, J.McMillan, L. (2014). Merging of multi-string bwts with applications. *Bioinformatics*, 30(24):3524–3531.

[Hu, 2016] Hu, YuanqiGeorgiou, P. (2016). A real-time de novo dna sequencing assembly platform based on an fpga implementation. *IEEE/ACM Trans. Comput. Biol. and Bioinf.*, 13(2):291–300.

[Hu et al., 2012] Hu, Y., Liu, Y., Toumazou, C., and Georgiou, P. (2012). A cmos architecture allowing parallel dna comparison for on-chip assembly. *2012 IEEE International Symposium on Circuits and Systems*.

[Huang et al., 2013] Huang, Y. L., Liu, C. S., Li, Y. C., and Lu, Y. C. (2013). Architecture and circuit design of parallel processing elements for de novo sequence assembly. In *SOC Conference (SOCC), 2013 IEEE 26th International*, pages 50–54.

[Jones, 2004] Jones, Neil CPevzner, P. (2004). *An introduction to bioinformatics algorithms*. MIT Press.

[Kucherov et al., 2014] Kucherov, G., Salikhov, K., and Tsur, D. (2014). Approximate string matching using a bidirectional index. *Combinatorial Pattern Matching*, pages 222–231.

[Kuo et al., 2013] Kuo, Y.-H., Liu, C.-S., Li, Y.-C., and Lu, Y.-C. (2013). Parallel architecture and hardware implementation of pre-processor and post-processor for sequence assembly. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*.

[Lapidus, 2015] Lapidus, A. (2015). Genome sequence databases: Sequencing and assembly? *Reference Module in Biomedical Sciences*.

[Li, 2014] Li, H. (2014). Fast construction of fm-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275.

[Mantaci et al., 2005] Mantaci, S., Restivo, A., Rosone, G., and Sciortino, M. (2005). An extension of the burrows wheeler transform and applications to sequence comparison and data compression. *Combinatorial Pattern Matching*, pages 178–189.

[McWilliam et al., 2013] McWilliam, H., Li, W., Uludag, M., Squizzato, S., Park, Y. M., Buso, N., Cowley, A. P., and Lopez, R. (2013). Analysis tool web services from the embl-ebi. *Nucleic Acids Research*, 41(W1):W597–W600.

[Pevsner, 2015] Pevsner, J. (2015). *Bioinformatics and Functional Genomics 3rd Edition*. John Wiley Sons, 3 edition.

[Pfeiffer et al., 2009] Pfeiffer, G., Baumgart, S., Schröder, J., and Schimmler, M. (2009). A massively parallel architecture for bioinformatics. *Lecture Notes in Computer Science*, pages 994–1003.

[Poirier et al., 2015a] Poirier, C., Gosselin, B., and Fortier, P. (2015a). Dna assembly with de bruijn graphs on fpga. In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. Institute of Electrical and Electronics Engineers (IEEE).

[Poirier et al., 2015b] Poirier, C., Gosselin, B., and Fortier, P. (2015b). Dna assembly with de bruijn graphs on fpga. *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*.

[Rodland, 2013] Rodland, E. (2013). Compact representation of k-mer de bruijn graphs for genome read assembly. *BMC Bioinformatics*, 14(1):313.

[Sasoglu and Tse, 2014] Sasoglu, E. and Tse, D. (2014). Dna assembly from paired reads as 2-d jigsaw puzzles. In *2014 IEEE International Symposium on Information Theory*. Institute of Electrical Electronics Engineers (IEEE).

[Sedlar et al., 2014] Sedlar, K., Skutkova, H., Vitek, M., and Provaznik, I. (2014). Prokaryotic dna signal downsampling for fast whole genome comparison. *Advances in Intelligent Systems and Computing*, pages 373–383.

[Shomorony et al., 2016] Shomorony, I., Kamath, G. M., Xia, F., Courtade, T. A., and Tse, D. N. (2016). Partial dna assembly: A rate-distortion perspective. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 1799–1803.

[Sievers et al., 2014] Sievers, F., Wilm, A., Dineen, D., Gibson, T. J., Karplus, K., Li, W., Lopez, R., McWilliam, H., Remmert, M., and Soding, J. e. a. (2014). Fast,

scalable generation of high-quality protein multiple sequence alignments using clustal omega. *Molecular Systems Biology*, 7(1):539–539.

[Simpson, 2010] Simpson, J. T.Durbin, R. (2010). Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373.

[Simpson, 2011] Simpson, J. T.Durbin, R. (2011). Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556.

[Sović et al., 2013] Sović, I., Skala, K., and Šikić, M. (2013). Approaches to dna de novo assembly. In *Information Communication Technology Electronics Microelectronics (MIPRO), 2013 36th International Convention on*, pages 351–359.

[Varma et al., 2014] Varma, B. S. C., Paul, K., and Balakrishnan, M. (2014). Accelerating genome assembly using hard embedded blocks in fpgas. *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*.

[Varma et al., 2013] Varma, B. S. C., Paul, K., Balakrishnan, M., and Lavenier, D. (2013). Fassem: Fpga based acceleration of de novo genome assembly. *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*.

[Waidyasooriya et al., 2013] Waidyasooriya, H. M., Hariyama, M., and Kameyama, M. (2013). Implementation of a custom hardware-accelerator for short-read mapping using burrows-wheeler alignment. *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*.

[Waidyasooriya, 2016] Waidyasooriya, Hasitha MuthumalaHariyama, M. (2016). Hardware-acceleration of short-read alignment based on the burrows-wheeler transform. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1358–1372.

[Wang et al., 2015] Wang, C., Guo, M., Liu, X., Liu, Y., and Zou, Q. (2015). Seedsgraph: an efficient assembler for next-generation sequencing data. *BMC Med Genomics*, 8(Suppl 2):S13.

[Wienbrandt, 2013] Wienbrandt, L. (2013). *Bioinformatics Applications on the FPGA-Based High-Performance Computer RIVYERA*, pages 81–103. Springer New York, New York, NY.

[Zhang et al., 2015] Zhang, F., Liao, X., Peng, S., Cui, Y., Wang, B., Zhu, X., and Liu, J. (2015). A hybrid parallel strategy based on string graph theory to improve de novo dna assembly on the tianhe-2 supercomputer. *Interdisciplinary Sciences: Computational Life Sciences*, 8(2):169–176.

[Şaşoğlu and Tse, 2014] Şaşoğlu, E. and Tse, D. (2014). Dna assembly from paired reads as 2-d jigsaw puzzles. In *2014 IEEE International Symposium on Information Theory*, pages 1286–1290.