UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

Algoritmos y arquitectura para cálculo de similitudes genómicas en aceleradores en hardware

Tesis para optar al grado de Doctor en Ciencias de la Ingeniería
con mención en Ingeniería Eléctrica

Javier Esteban Soto Salcedo
Concepción - Chile
2023

Profesor Guía:
Dr. Miguel Figueroa T.
Dra. Cecilia Hernández R.

Universidad de Concepción
Facultad de Ingeniería
Departamento de Ingeniería Eléctrica

Profesor Guía:
Dr. Miguel Figueroa T.
Dra. Cecilia Hernández R.

Algoritmos y arquitectura para cálculo de similitudes genómicas en aceleradores en hardware

Javier Esteban Soto Salcedo

Informe de tesis
Doctorado en Ciencias de la Ingeniería c/m en Ingeniería Eléctrica

24 de agosto de 2023

Resumen

La similitud entre genomas es un concepto fundamental en bioinformática. Esta se utiliza para determinar relaciones evolutivas o filogenéticas entre diferentes especies, para identificar regiones similares en genomas, o para organizar genomas con características genéticas similares en grupos o clústeres. Existen diversas métricas de comparación utilizadas en la literatura, las cuales utilizan diferentes principios para medir la similitud. Una de las más utilizadas en genómica es la similitud de Jaccard, la que se basa en teoría de conjuntos y evalúa la presencia o ausencia de elementos en los dos conjuntos a comparar para indicar qué tan similares son. Otra métrica utilizada es la divergencia de Jensen-Shannon, que usa información estadística asociada a la distribución de probabilidad de un genoma, que puede estar representado por más de una secuencia. En general, el procesamiento de genomas se realiza describiéndolos como un conjunto de *substrings* de largo k, llamados k-mers. Tanto el proceso de extracción de k-mers como el cálculo de similitud, son tareas computacionalmente desafiantes, debido al alto uso de memoria requerida por el volumen de datos asociado a datos genómicos y a la complejidad cuadrática del cálculo de similitud entre pares.

Este trabajo presenta el diseño de algoritmos y aceleradores usando el paradigma de *FPGA-as-a-service* para calcular la similitud entre genomas utilizando estructuras de datos probabilísticas o *sketches*. El primer acelerador realiza el cálculo de similitud Jaccard entre genomas, pudiendo calcular todas las similitudes entre los elementos de una base de datos o solo aquellos que tengan la posibilidad de superar cierto umbral, permitiendo así reducir significativamente la cantidad de operaciones a realizar. Después de la construcción de sketches, el acelerador puede calcular más de 96 millones de coeficientes Jaccard por segundo en una instancia f1.2xlarge de Amazon Web Services (AWS) con un FPGA XCVU9P, lo que presenta una aceleración de 58 veces sobre el estado del arte en software corriendo en una instancia c5.9xlarge. El acelerador es 27 veces más rápido que una implementación directa en GPU y 4 veces más rápido que una implementación optimizada en GPU, correspondiente a una adaptación del algoritmo diseñado para el FPGA. Ambas implementaciones en GPU fueron evaluadas en una instancia g5.4xlarge que cuenta con una GPU NVIDIA A10G. El segundo acelerador implementa el cálculo de similitud usando divergencia de Jensen-Shannon. El acelerador utiliza una estructura llamada arreglo de colas de prioridad (PQA), la cual almacena los elementos más frecuentes de un conjunto para el cálculo de entropía. Esta implementación alcanza una aceleración de $1,9$ veces comparada con la implementación del mismo algoritmo en GPU, en ambos casos usando las instancias f1.2xlarge y g5.4xlarge respectivamente.

"Los mejores libros son los que nos dicen lo que ya sabemos."

- Winston.

Durante las diferentes etapas de este trabajo he recibido el apoyo financiero del Programa de Becas para Estudios de Doctorado Nacional de la Agencia Nacional de Investigación y Desarrollo de Chile (ANID) del Gobierno de Chile.

Tabla de contenidos

Lista de figuras

Lista de tablas

Lista de algoritmos

Abreviaciones

ADN	Ácido desoxirribonucleico.
NGS	Next-Generation Sequencing.
NCBI	National Center for Biotechnology Information.
OLC	Overlap-Layout-Consensus.
SIMD	Single Instruction, Multiple Data.
CPU	Central Processing Unit.
GPU	Graphics Processing Unit.
FPGA	Field-Programmable Gate Array.
RAM	Random Access Memory.
BRAM	Block RAM.
URAM	Ultra RAM.
DDR	Double Data Rate.
FIFO	First In, First Out.
FaaS	FPGA-as-a-service.
JSD	Jensen-Shannon Divergence.
PQA	Priority Queues Array.
AWS	Amazon Web Services.
HLS	High-Level Synthesis.
RTL	Register-Transfer Level.

Capítulo 1. Introducción

1.1. Introducción general

La genómica es el estudio y análisis de genomas, que corresponden al conjunto de bases de ADN que representan a un organismo. La genómica comprende el estudio de función, organización y evolución de genes, entre otros. Usualmente en bioinformática, los genomas son representados como un conjunto de *k-mers*, que corresponden a subcadenas de k bases de ADN, donde cada base es un nucleótido descrito por un carácter 'A', 'C', 'G' o 'T'. La cantidad máxima de k-mers que se pueden obtener de una secuencia son $L - k + 1$, donde L es el largo de la secuencia. Los k-mers pueden ser utilizados en algoritmos para obtener cuentas, realizar comparaciones, o agruparlos bajo algún criterio. En secuencias de ADN, el valor de k tiene relación con el nivel filogenético que representa dicha secuencia. De acuerdo con Koslicki y Falush [1] con $k = 21$ es posible encontrar similitudes a nivel género, con $k = 31$ al nivel de especies y con $k = 51$ en cepas. Esto se basa en que k-mer más largos describen características más específicas de un organismo.

Las tecnologías de secuenciación modernas, como secuenciación de próxima generación (Next Generation Sequencing, NGS), han habilitado la producción de una gran cantidad de datos genómicos accesibles de forma pública. Por ejemplo, la base de datos Reference Sequence (RefSeq) del Centro Nacional para la Información Biotecnológica (National Center for Biotechnology Information, NCBI), el cual provee de un "conjunto de secuencias bien anotadas, completo, integrado y no redundante", ha tenido un crecimiento lineal en los últimos años. En junio del 2022 esta base de datos contenía secuencias de 119 373 organismos diferentes, con más de 6 TiB de datos descomprimidos [2]. Otras bases de datos disponen de aún más datos, como los datos unificados de proteínas gastrointestinales humanas (Unified Human Gastrointestinal Protein, UHGP) [3], que tiene un catálogo de 204 938 genomas de referencia. El mapa de variación genómica (Genome Variation Map, GVM) [4] creció desde 8884 muestras en septiembre de 2017 a 69,004 muestras en enero de 2022, siendo compuesto por 977 482 968 variantes de 43 especies. Las estimaciones muestran que solo los datos genómicos de humanos van a requerir entre 2 y 40 exabytes de almacenamiento para el 2025 [5].

Con el objetivo de reducir el tiempo de ejecución, en la literatura se han propuesto arquitecturas que explotan el paralelismo de los datos a nivel de hebras y procesos, usando Unidades de Procesamiento Gráfico (Graphic Processing Unit, GPU), Arreglos de Compuertas Lógicas Pro-

gramables (Field Programmable Gate Arrays, FPGA) y procesamiento en la nube [6]. Incluso las implementaciones en software [7, 8, 9] utilizan paralelismo a nivel hebras e instrucciones SIMD (Single Instruction, Multiple Data) para reducir el tiempo de ejecución. Las implementaciones que utilizan el paradigma SIMD en aceleradores GPU se han vuelto comunes en el último tiempo. En la literatura es posible encontrar implementaciones de alineadores de secuencias [10, 11], conteo de k-mers [12] y clasificación de datos metagenómicos [13] con aceleraciones mayores a dos órdenes de magnitud [13] comparados con arquitecturas tradicionales. Los aceleradores basados en FPGA han sido utilizados para acelerar procesos como generación de semillas en alineadores de secuencias [14, 15, 16], conteo de k-mers [6, 17] y superposición de pares [18] en el proceso de ensamblaje. Recientemente, la disponibilidad de FPGA en las plataformas de cómputo en la nube como AWS [19] ha facilitado el uso de estos aceleradores en aplicaciones de genómica [14, 18, 20, 21]. Estas plataformas han introducido un nuevo paradigma conocido como FPGA como servicio o *FPGA-as-a-service*, el que ha sido utilizado en genómica y áreas relacionadas [22, 23]. Este paradigma hace referencia al acceso a aceleradores bajo un esquema de servicio, lo que elimina el costo inicial de adquisición de los dispositivos físicos para el desarrollo en estas plataformas.

El cálculo de la similitud entre genomas es una métrica de comparación que indica el grado de semejanza entre dos cadenas de ADN. Este es un proceso esencial en diferentes aplicaciones en biología y biomedicina, como *clustering* [24, 25], ensamblaje de genomas [26, 27, 28] o taxonomía metagenómica [29, 30]. Existen muchas métricas para el cálculo de similitud [31], una de las más populares es el coeficiente de Jaccard, también conocido como similitud Jaccard. Esta métrica se utiliza para comparar dos conjuntos usando la relación entre la cardinalidad de la intersección y la unión de los dos conjuntos. En los últimos años, se han publicado herramientas como Mash [24] y Dashing [7], que utilizan algoritmos eficientes y técnicas de paralelismo para estimar la similitud de Jaccard. Otras métricas, como la divergencia de Jensen-Shannon que utiliza información estadística de los elementos, se han utilizado para discriminar entre muestras de ADN [32] de distintos conjuntos y como medida de similitud en procesos de comparación de genomas genomas [33] y estudios de asociación de genomas completos (Genome-Wide Association Study, GWAS) [34].

El crecimiento continuo de los datos genómicos presenta un desafío para los sistemas informáticos de cómputo general. Para afrontar este problema, algunas soluciones de procesamiento de datos genómicos utilizan algoritmos de uso eficiente de espacios basados en sketches, los que son estructuras de datos probabilísticas que pueden estimar características de un conjunto de datos, tales como cardinalidad, frecuencia o pertenencia, con un uso de memoria sublineal al

tamaño de los datos de entrada [24, 7]. Además, sketches del mismo tamaño pueden representar conjuntos de diferente tamaño, y los algoritmos pueden extraer propiedades de los conjuntos directamente desde los sketches [24, 35, 36, 37, 7].

En la literatura se presentan diversas implementaciones para el cálculo de similitud. Sin embargo no se han explorado implementaciones en hardware de ellas, lo que puede ocurrir por la complejidad y tiempos de desarrollo que requieren. La similitud de Jaccard posee implementaciones eficientes en software, utilizando paralelismo a nivel de hebras e instrucciones vectoriales [7]. Esto no se ve en el caso de como la divergencia de Jensen-Shannon, que no tiene trabajos dedicados enteramente a su cálculo. En ninguno de estos casos existen implementaciones en GPU o FPGA para acelerar estos algoritmos, cuya disponibilidad ha aumentado gracias a las plataformas de cómputo en la nube, siendo una de sus aplicaciones la aceleración de algoritmos para problemas de genómica.

Esta tesis presenta el diseño de algoritmos y arquitecturas para el cálculo de similitud entre genomas, usando dos métodos populares, como lo son la similitud de Jaccard y la divergencia de Jensen–Shannon. Los algoritmos y arquitecturas propuestas tienen un enfoque en aceleradores hardware, reutilizando la mayor cantidad de datos para explotar el paralelismo de ellos, reduciendo al mismo tiempo la cantidad de transacciones con memoria externa. Para ambos métodos se propuso una arquitectura en GPU y FPGA, evaluando el desempeño en cada una de ellas en contraste con implementaciones en software. Las arquitecturas propuestas en esta tesis hacen uso de estructuras probabilísticas de datos o sketches, con el fin de cumplir con las restricciones de memoria presente en los aceleradores utilizados. Todo el desarrollo de esta tesis se hizo usando los servicios en la nube de Amazon (Amazon Web Services, AWS).

1.2. Hipótesis

Los algoritmos de streaming y las estructuras de datos probabilísticas habilitan el uso de aceleradores hardware en genómica, ya que permiten desarrollar aplicaciones en plataformas atractivas por sus altas capacidades de paralelismo, pero con fuertes restricciones de memoria.

1.3. Objetivo general

El objetivo principal de esta tesis es el diseño de algoritmos y arquitecturas para aceleradores hardware, usando estructuras de datos probabilísticas y algoritmos de streaming, para acelerar el cálculo de similitud entre secuencias genómicas.

1.4. Objetivos específicos

1. Identificación de métricas de similitud entre genomas con potencial de aceleración en hardware.
2. Diseño de algoritmos de streaming para el manejo eficiente de grandes volúmenes de datos.
3. Diseño de arquitecturas para implementar estructuras probabilísticas en hardware.
4. Implementación de arquitecturas y algoritmos diseñados en aceleradores GPU y FPGA.
5. Diseño de software para manejo de lectura de genomas, escritura de resultados y control del acelerador.
6. Evaluación y comparación del desempeño de diferentes soluciones en infraestructura en la nube.

1.5. Temario

El temario de este trabajo se describe a continuación:

1. El Capítulo 2 presenta la revisión del estado del arte correspondiente a desarrollos en hardware para procesamiento de datos genómicos y cálculo de similitud entre genomas.
2. El Capítulo 3 describe las estructuras de datos sobre las que se basó el trabajo desarrollado.
3. El Capítulo 4 presenta el desarrollo de un acelerador en hardware para el cálculo de similitud Jaccard entre un conjunto de genomas.

4. El Capítulo 5 presenta el desarrollo de un acelerador en hardware para el cálculo de la divergencia de Jensen-Shannon como indicador de similitud entre genomas.

5. El Capítulo 6 presenta los resultados de ambos trabajos y el marco sobre el cual fueron validados.

6. El Capítulo 7 recopila las conclusiones alcanzadas durante el desarrollo de esta tesis y delinea el trabajo a futuro en este tema.

Capítulo 2. Estado del arte

2.1. Similitudes genómicas

El cálculo de similitud es un proceso con muchas aplicaciones en genómica y con requerimientos de cómputo altos, ya que implica mediciones de métricas entre pares, lo que tiene una complejidad $O(n^2)$ cuando se requiere calcular todas las posibles similitudes. La similitud de Jaccard se usa de forma recurrente en aplicaciones con genomas. Ésta está relacionada con la identidad promedio de nucleótidos (Average Nucleotide Identity, ANI), donde un valor de ANI de 97 % es considerado como un umbral de similitud válido para una correcta clasificación de procariontes [38].

Mash, presentado por Ondov et al. [24], es un software para el cálculo de similitud, que utiliza el índice de Jaccard como métrica. Mash extiende el sketch MinHash [39] para estimar la similitud Jaccard entre pares de conjuntos. Zhao [35] presentó BinDash, un sketch MinHash que usa una función hash de una permutación o *rolling hash*. BinDash solo almacena 14 bits por hash, por lo que usa la mitad de la memoria que Mash y es un 10 % más rápido. Recientemente, Baker y Langmead [7] presentaron Dashing, una alternativa a Mash que utiliza sketches HyperLogLog para estimar la similitud de Jaccard usando las cardinalidades de los conjuntos y el principio de inclusión-exclusión. Dashing es una implementación que explota el paralelismo presente en procesadores actuales, hace un uso intensivo de múltiples hebras e instrucciones SIMD, alcanzando un alto rendimiento comparado con otras alternativas como Mash, sobre el cual obtiene aceleraciones entre $2,3$ y $9,4$ veces, dependiendo del tamaño del sketch. En un servidor con cuatro procesadores Intel E7-4830 y 112 hebras físicas, Dashing puede procesar 87 000 genomas en 6 minutos [7].

Dashing y Mash son herramientas importantes en la comparación de genomas y predicción funcional [40]. Algunas aplicaciones que usan Dashing incluyen la investigación de enfermedades [40, 41], computo de distancias evolutivas [42], y similitud de genomas [43]. Moustafa et al. [40] incluyó Dashing y Mash como herramientas para el cálculo de distancias en comparativas genómicas para descifrar la propagación y patogenia de enfermedades infecciosas. Lipworth et al. [41] usa Dashing para identificar potenciales factores genéticos patógenos involucrados en el incremento de infecciones relacionadas con las bacterias Escherichia coli y *Klebsiella*. Criscuolo [42] usa la similitud entre pares para estimar la inferencia filogenética entre genomas, sugiriendo que herramientas como Dashing y Mash son fundamentales para definir una métrica

de distancia evolutiva.

Otra métrica de similitud usada es la divergencia de Jensen-Shannon (JSD), la cual usa la distribución de dos conjuntos para calcular el grado de similitud entre ellos. La divergencia de Jensen-Shannon se utiliza dentro de otros procesos [33, 34], como comparación de genomas usando perfiles de frecuencia para lecturas sin alineamiento [33], o análisis de interacciones entre enfermedades y genes, en estudios de asociación de genomas completos [34]. Ramakrishnan y Bose [32], utilizaron Jensen-Shannon como medida de distancia en el análisis de distribuciones de muestras de ADN sanos y ADN de tumores de carcinoma renal. En este trabajo se utilizó esta métrica como base para entrenar un clasificador para la predicción temprana de cáncer renal. Inward et al. [44] evaluaron múltiples estrategias para estimar los parámetros epidemiológicos del virus SARS-CoV-2, utilizando Jensen-Shannon para comparar los parámetros estimados con los datos genéticos y epidemiológicos.

2.2. Estructuras de datos en genómica

Las estructuras de datos probabilísticas son métodos que permiten representar grandes conjuntos de datos utilizando una cantidad relativamente pequeña de espacio de memoria, proporcionando estimaciones aproximadas de ciertas propiedades o características de los datos. Estas estructuras han sido ampliamente utilizadas en diversas áreas de la informática, como minería de datos, recuperación de información y análisis de grandes conjuntos de datos.

El uso de sketches en genómica es muy extendido [45] y se ha aplicado con éxito para abordar problemas como la búsqueda rápida de similitud entre secuencias genéticas [24, 7], la identificación de variantes genéticas [46] o análisis de abundancia [47]. Bovee and Greenfield [47] propusieron Finch, una herramienta de manipulación de sketches MinHash permitiendo la creación y filtrado de sketches con datos genómicos. Joudaki et al. [48] propusieron un algoritmo para la etapa de *seed-and-extend* en el proceso de alineamiento usando coincidencias inexactas, permitiendo reducir el tiempo de ejecución y manteniendo un acierto dentro de márgenes aceptables. Este algoritmo usa un Tensor sketch para contar k-mers y crear un espacio tensorial. Kockan et al.[49] diseñaron SkSES, un framework para realizar análisis estadísticos de datos genómicos usando sketches para almacenar las estimaciones de frecuencia de los elementos más frecuentes o top-K. Breitwieser et al. [50] presentaron KrakenUniq, un algoritmo de taxonomía para datos metagenómicos. En este trabajo se usa un sketch HyperLogLog para representar los taxones de una base de datos de referencia y luego contabilizar la cantidad de k-mers distintos por taxón

generados por los datos de entrada, con lo cual se realiza la clasificación. Otros trabajos [17, 51] utilizan sketches para reducir el uso de memoria y habilitar el uso de aceleradores hardware en genómica. Mcvicar et al. [17] utilizaron un sketch Bloom filter para el conteo de k-mers en FPGA. Mientras Saavedra et al. [51] usaron un sketch de frecuencia Countmin-CU para la detección de los elementos más frecuentes o Heavy-Hitters, diseñando una arquitectura en FPGA.

Mohamadi et al. [52] presentaron ntCard, un algoritmo de streaming para la estimación de las frecuencias de k-mers en un genoma. Este trabajo presentó una estructura de datos capaz de almacenar información suficiente para estimar el histograma de un genoma. ntCard logra esto haciendo un muestreo de los datos y almacenando información relativa a la cantidad de variaciones por bit vistas en cada k-mer nuevo. La implementación en C++ del algoritmo propuesto alcanzó aceleraciones de 15 veces en tiempo de ejecución sobre el estado del arte.

2.3. Aceleración en hardware

Las aplicaciones de genómica han sido siempre atractivas para los algoritmos paralelos y la aceleración por hardware, debido al rápido crecimiento del volumen de datos. Los aceleradores en plataformas hardware como GPU y FPGA pueden explotar el paralelismo de grano fino a una escala superior a las capacidades de las instrucciones SIMD tradicionales y los múltiples hilos de hardware disponibles en los procesadores de propósito general. Las GPU explotan principalmente el paralelismo SIMD con datos de ancho fijo y aritmética de punto flotante a un nivel masivo utilizando hilos gestionados por hardware. Los FPGA permiten adaptar la arquitectura del acelerador al algoritmo para explotar el paralelismo de grano grueso y fino. El diseño para FPGA consume más tiempo que la programación de GPU, pero puede lograr un rendimiento y una eficiencia energética significativamente mayores [6].

Los algoritmos de genómica acelerados en GPU incluyen: ensamblaje [11], remoción de deleciones [53] y alineadores de lecturas [54, 55]. En algunos ejemplos, Zeni et al. [55] presentaron un alineador de lectura larga de alto rendimiento utilizando un acelerador de seis GPU, que logra un aumento de velocidad de $30,7$ veces sobre una implementación de software de última generación. Goenka et al. [54] presentan un enfoque similar para la alineación del genoma, implementado en una instancia de AWS de 8 GPU, logrando un aumento de velocidad de 14 veces y una reducción de costes de 2,3 veces en comparación con una implementación de software tradicional.

Los FPGA se han utilizado como aceleradores de algoritmos de streaming en aplicaciones que requieren un alto rendimiento y flujo de datos [56, 57, 51, 36, 58]. Debido a su arquitectura configurable de grano fino con restricciones de memoria, los aceleradores basados en FPGA favorecen el uso de algoritmos basados en sketches, que pueden exhibir un gran paralelismo de datos y una pequeña huella de memoria. McVicar et al. [17] diseñaron un acelerador basado en FPGA para contar k-mers utilizando un Bloom fliter, que logra una aceleración de hasta $17,6$ veces sobre una implementación de software multihebra. Asimismo, Saavedra et al. [51] presentaron un acelerador que utiliza un sketch Countmin-CU para detectar los elementos más frecuentes en datos genómicos y de tráfico de red, consiguiendo un aumento de velocidad de más de 700 veces en comparación con un ordenador de escritorio. Otras aplicaciones de aceleradores FPGA basados en sketches son las de Tong y Prasanna [56], que combinan sketches Count-Min y K-ary para detectar *heavy hitters* y *heavy changes* en el tráfico de red, y Soto et al. [59], que estiman la entropía del flujo de red a más de 204 Gbps utilizando Countmin-CU, sketches HLL y un arreglo de colas de prioridad, solución que usa sketches y algoritmos de streaming previamente utilizados en aplicaciones de red y genómica.

En los últimos años, se han desarrollado muchas implementaciones de algoritmos relacionados a genómica en FPGA, la mayoría de ellos usando Amazon Web Services. Fujiki et al. [20] diseñaron una arquitectura para acelerar la etapa de extensión de semilla (*seed-extension*) en el proceso de alineamiento. Ellos probaron su implementación en una instancia f1.2xlarge con el algoritmo de alineamiento BWA-MEM2, alcanzando una aceleración de $2,3$ veces sobre la implementación en software. Subramaniyan et al. [14] diseñaron un acelerador hardware para la misma etapa de extensión de semilla. En este trabajo utilizaron una instancia f1.4xlarge, reduciendo el tiempo de la etapa $3,3$ veces y el proceso de completo de alineamiento en $2,1$ veces. Ham et al. [23] usaron una f1.2xlarge para acelerar etapas del flujo de análisis genómico GATK4, alcanzando una aceleración de $19,3$ veces sobre una implementación multihebra en software. Saavedra et al. [36] propusieron un algoritmo y un acelerador en FPGA para obtener k-mers discriminativos, alcanzando una aceleración de 78 veces comparado con una implementación en software que utilizaba instrucciones SIMD y múltiples hebras en un procesador con 12 núcleos. Wu et al. [21] presentaron un acelerador para el proceso de realineamiento, el que representa un tercio del tiempo de ejecución en el diagnóstico genómico de cáncer agudo, alcanzando una aceleración de 81 veces sobre la implementación en software del mismo algoritmo. En otro trabajo, Guo et at. [18] presentaron un detector de superposición entre pares para lecturas largas en FPGA y GPU. Este trabajo aceleró la primera etapa del proceso de ensamblaje, que frecuentemente utiliza un paradigma *Overlap-Layout-Consensus* [60] (OLC). Las implementaciones alcanzan una aceleración de 7 y 28 veces para GPU y FPGA respectivamente, respecto al software de

referencia.

2.4. Discusión

La similitud Jaccard es muy usada para comparar genomas, por esto existen varios trabajos que se han centrado específicamente en esta tarea, buscando distintos algoritmos para estimar esta métrica y usando distintas estrategias de aceleración en software. Por otro lado, la divergencia de Jensen-Shannon se ha utilizado principalmente con distribuciones de datos, por lo que ha sido útil en tareas donde existen varias muestras disponibles de un mismo genoma o gen y se requiere realizar comparaciones entre ellos. Aun así, Jensen-Shannon puede ser utilizado en distancia de genomas completos [31] y su uso presenta una oportunidad, pues no ha sido objeto de implementaciones eficientes en software ni en hardware, ni tampoco de estudios de su cómputo usando sketches.

El uso de sketches en genómica es muy extendido. Las principales aplicaciones de cálculo de distancia y taxonomía utilizan estas estructuras para reducir los requerimientos de memoria asociados al procesamiento. El uso de estructuras de datos probabilísticas está muy relacionado con la aceleración en hardware [61], ya que estas estructuras habilitan el desarrollo de aceleradores para genómica, permitiendo realizar ciertas operaciones en plataformas con memoria interna muy limitada.

Los aceleradores revisados en las secciones anteriores se enfocan en etapas de algoritmos de genómica, presentando aceleraciones de hasta dos órdenes de magnitud [62]. Estos resultados se deben al alto potencial de paralelización que presenta el procesamiento de k-mers, elemento común en muchos de los algoritmos revisados, cuya extracción y uso puede ser paralelizado de forma eficiente. En este punto, un FPGA presentan ventajas sobre una GPU, ya que estos últimos están diseñados para trabajar con operaciones en punto flotante, además solo exponen paralelismo espacial y no es posible definir *pipelines* internos, sin intervención del CPU, como sí lo es un FPGA. Un punto en contra de los FPGA es la escasez de memoria interna, la que llega a cerca de 60 MiB en las gamas altas [63]. Esta limitante se puede sobrellevar de varias formas [64]: utilizando memorias externas, dividiendo el procesamiento en múltiples FPGA o particionando los problemas para procesarlos por parte, en cualquier caso esto implica una complicación adicional en el diseño. Otra posibilidad para reducir el uso de memoria es utilizar estructuras de datos probabilísticas o sketches, los cuales presentan un uso de memoria sublineal respecto a los datos de entrada y pueden ser utilizados para cuentas [65], cardinalidad [66] o

detección de presencia [67]. También es posible utilizar datos comprimidos con la transformada de Burrows-Wheeler [68] o FM-Index [69].

Aunque las estructuras de datos probabilísticas ofrecen ventajas significativas en términos de eficiencia computacional y ahorro de memoria, también presentan limitaciones en la precisión de las estimaciones. Al ser aproximaciones, las respuestas proporcionadas por los sketches pueden no ser exactas, lo que requiere un cuidadoso diseño y calibración según el contexto de aplicación. El uso de estas técnicas en una aplicación como el cálculo de similitud genómica es un buen punto de inicio para el desarrollo de aceleradores hardware en el ámbito de genómica. Al ser un proceso acotado y tener trabajos existentes en el estado del arte, es posible comparar tanto el tiempo de ejecución como los errores asociados a las estimaciones y aproximaciones necesarias para su implementación.

Capítulo 3. Estructuras de datos

3.1. Introducción

Una estructura de datos es una forma de almacenar u organizar datos de forma eficiente en un sistema informático. Estas se caracterizan por tener un tiempo de respuesta rápido y escalar de forma lineal a los datos que almacenan. Un ejemplo común son las tablas hash, que reducen o eliminan (dependiendo de la implementación [70]) el tiempo asociado a la búsqueda de una llave en la tabla. Como el crecimiento de estas estructuras es lineal a los datos, requieren de una gran cantidad de memoria en aplicaciones con grandes volúmenes de datos, resultando prohibitivos o difíciles de utilizar en estas aplicaciones.

Por otro lado, las estructuras probabilísticas o sketches son estructuras de datos que pueden estimar características de un conjunto [71], como cardinalidad [72, 66], frecuencia [73, 65] o pertenencia [74], con una alta precisión y utilizando un espacio sublineal al tamaño del conjunto. Estas estructuras tienen aplicaciones en todas las áreas que manejan grandes volúmenes de datos como: bases de datos [75, 76], redes [77, 78] o genómica [36, 79]. Las estructuras probabilísticas de datos utilizan menos espacio del necesario para almacenar todos los datos de los conjuntos de entrada. Debido a esto es frecuente que ocurran colisiones entre los elementos. Para reducir el efecto de estas colisiones es común utilizar funciones hash universales [80], con el fin de distribuir los datos de manera uniforme en la estructura.

Este capítulo revisa el marco teórico asociado a las estructuras de datos utilizadas en el resto del trabajo que sirven como base de los algoritmos e implementaciones diseñadas.

3.2. Funciones hash

Las funciones hash tienen un rol fundamental en las estructuras de datos probabilísticas, aleatorizando el contenido y representándolo en un tamaño fijo, obteniendo una cantidad determinada de bytes para una entrada de datos de cualquier tamaño. Una buena función hash cumple con tres propiedades: (1) que la salida esté completamente determinada por los datos de entrada, (2) que utilice todos los bits de la entrada para el cálculo de la salida y (3) que la distribución de la salida sea uniforme. Las funciones hash que cumplen estas características se

conocen como funciones hash universales.

Las funciones hash universales fueron propuestas por Carter y Wegman [80], y son aquellas en las que la probabilidad de colisión para un par de elementos es menor a $1/B$ donde B es el tamaño del codominio o la cantidad de hashes posibles. Las funciones hash universales están diseñadas para reducir el número de colisiones generando valores de salida con una distribución uniforme ante datos de entrada aleatorios. Algunas de las funciones hash utilizadas en genómica son: MurmurHash [81], ntHash [82] y WangHash [83].

3.3. Estimación de cardinalidad

El problema de estimación de cardinalidad es la tarea de encontrar la cantidad de elementos distintos presentes en un conjunto de datos. Esta tarea se puede resolver de forma exacta utilizando estructuras como un arreglo de datos o una tabla hash, requiriendo en ambos casos una memoria lineal a la cantidad de elementos distintos. Una alternativa es utilizar estructuras probabilísticas como LogLog o HyperLogLog, las cuales almacenan información en un espacio sublineal con el que es posible estimar la cantidad de elementos distintos con un bajo error.

3.3.1. LogLog

El sketch LogLog fue propuesto por Marianne Durand y Philippe Flajole [72] en el año 2003. Este es una estructura probabilística que almacena información suficiente para estimar la cardinalidad de un conjunto de datos. LogLog se basa en un principio sencillo para estimar la cardinalidad, el cual es que si el conjunto tiene suficientes datos y éstos se distribuyen de forma uniforme, el mayor número observado será una buena estimación de la cantidad de elementos diferentes. Como este enfoque sobreestimará el resultado, LogLog aplica una función hash a los elementos de entrada y utiliza múltiples registros o buckets, distribuyendo los datos de entrada entre $m = 2^p$ buckets, donde p es la cantidad de *bits* del resultado de la función hash utilizados para direccionar la estructura. Cada bucket almacena la cantidad de ceros a la izquierda del sufijo de la función hash como un indicador del número visto, generando una estimación de la magnitud de cada elemento en los diferentes buckets. El proceso de inserción se ilustra en la Figura 3.1. En LogLog la cardinalidad está definida como:

$$C = \alpha_m \; m \; 2^R \tag{3.1}$$

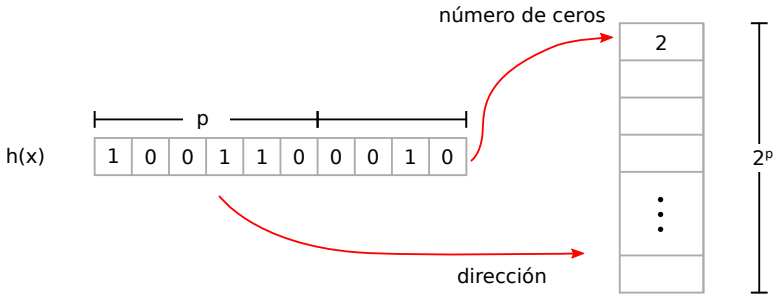


Figura 3.1: Ilustración de inserción a HyperLogLog.

donde C es la cantidad de elementos diferentes, m es la cantidad de buckets, R es la media geométrica de los buckets y $\alpha_m = \left(\Gamma(-1/m)\ \frac{1-2^{1/m}}{log2}\right)^m$. El error asociado a la estimación es

$$\delta \approx \frac{1,3}{\sqrt{m}}. \tag{3.2}$$

3.3.2. HyperLogLog

HyperLogLog presentado por Flajolet et al. [66] presenta una mejora en la estimación respecto a LogLog. HyperLogLog mantiene el proceso de inserción (Algoritmo 1), pero cambia la forma en que se calcula la estimación, utilizando una media armónica normalizada, definida como:

$$\hat{n} = \alpha_m m^2 \left(\sum_{j=1}^{m} 2^{-bucket[j]}\right)^{-1}, \tag{3.3}$$

donde

$$\alpha_m = \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u}\right)\right)^m du\right)^{-1}, \tag{3.4}$$

m es la cantidad de buckets y u son los elementos del conjunto. Para valores de m mayores o iguales a 2^7, α_m puede ser aproximado a $\frac{0,7216\ m}{m+1,079}$. Con esto, HyperLogLog reduce el error de estimación a

$$\delta \approx \frac{1,04}{\sqrt{m}}. \tag{3.5}$$

De acuerdo a lo anterior, el proceso de consulta a la estructura que se describe en el Algo-

Algoritmo 1: Inserción HyperLogLog

Input: Elemento x
Input: Sketch HyperLogLog A
Output: Sketch HyperLogLog A

1 $h \leftarrow hash(x)$
2 $v_1 \leftarrow < h_{63}, h_{64-p} >_2$
3 $v_2 \leftarrow < h_{63-p}, h_0 >_2$
4 $A[v_1] \leftarrow max\{A[v_1], ldz(v_2) + 1\}$

ritmo 2, calcula la media armónica de todos los buckets como:

$$Z = \sum_{j=0}^{m-1} 2^{-A[j]}, \tag{3.6}$$

donde $m = 2^p$ y A es el arreglo de datos donde se almacenan los buckets. Luego la cardinalidad es estimada como:

$$C_{HLL} = \alpha_A \frac{m^2}{Z}, \tag{3.7}$$

donde α_A es una constante asociada al tamaño del sketch. La estimación que obtiene HyperLogLog presenta buenos resultados en los casos donde el conjunto es lo suficientemente grande para que la ocupación de los buckets sea cercana al 100 %. En los casos donde hay un número significativo de buckets no ocupados, el error se incrementa. Heule et al. [84] propusieron una corrección a la estimación para estos casos, recalculando C_{HLL} como:

$$C_{HLL} = m \log\left(\frac{m}{n_z}\right), \tag{3.8}$$

donde n_z es el número de buckets en no utilizados y que almacenan un valor igual a cero.

Otra de las características de HyperLogLog es que define una operación de unión o *merge*, con la que es posible combinar dos estructuras, siempre y cuando tengan las mismas dimensiones, tal como describe el Algoritmo 3. Esta característica es particularmente útil para el procesamiento de streams en paralelo o para el cálculo de ciertos indicadores, como el índice Jaccard.

3.3.3. HyperLogLog++

La estimación de HyperLogLog, incluso con la corrección de la Ecuación (3.8), sigue presentando un error alto en sketches con baja ocupación. Por esto, Heule et al. [84] propusieron una

Algoritmo 2: Consulta HyperLogLog

Input: Sketch HyperLogLog A
Output: Estimación cardinalidad C_{HLL}

1 **Let**
2 $\quad m = 2^p$
3 $\alpha_A = 0{,}7213/(1 + \frac{1{,}079}{m})$ asumiendo $p \geq 7$
4 $Z \leftarrow \sum_{i=0}^{m-1} 2^{-A[i]}$
5 $C_{HLL} \leftarrow \alpha_A \frac{m^2}{Z}$
6 **if** $C_{HLL} \leq 2{,}5m$ **then**
7 $\quad n_z \leftarrow ldz(A)$
8 $\quad C_{HLL} \leftarrow m \log\left(\frac{m}{n_z}\right)$
9 **return** C_{HLL}

Algoritmo 3: Unión HyperLogLog

Input: Sketches HyperLogLog A, B
Output: Sketch HyperLogLog C

1 **Let**
2 $\quad m = 2^p$
3 **for** $i = 0$ **to** $m - 1$ **do**
4 $\quad C[i] \leftarrow \text{máx}(A[i], B[i])$
5 **return** C

variante de HyperLogLog, denominada HyperLogLog++. La cual compensa la estimación con base en datos obtenidos de forma empírica [84]. Para esto se obtiene la cantidad de buckets no utilizados (Paso 4 del Algoritmo 4), y, de ser mayor a cero, se utiliza la misma corrección de la estimación de HyperLogLog (Paso 5). Si esta estimación es menor a un umbral definido por el algoritmo y dependiente del tamaño del sketch, se retorna la estimación con esta corrección. En caso contrario se calcula el estimador original (Pasos 8 a 10 del Algoritmo 4), si esta estimación es menor a cinco veces el tamaño del sketch, ésta es compensada por un *bias* dependiente del valor de la estimación y del tamaño del sketch. HyperLogLog++ presenta una mejora en la estimación respecto al operador original, manteniendo sus características y uso de memoria. Es importante destacar que en casos de ocupación normal HyperLogLog++ y HyperLogLog presentan una estimación idéntica.

Algoritmo 4: Consulta HyperLogLog++

Input: Sketch HyperLogLog++ A
Output: Estimación cardinalidad C_{HLL}

Let
  $m = 2^p$
$n_z \leftarrow zeros\_counter(A)$
if $n_z > 0$ **then**
  $C_{HLL} \leftarrow m \log\left(\frac{m}{n_z}\right)$
  if $C_{HLL} \leq threshold_p$ **then**
    return C_{HLL}
$\alpha_A = 0{,}7213/(1 + \frac{1{,}079}{m})$ asumiendo $p \geq 7$
$Z \leftarrow \sum_{i=0}^{m-1} 2^{-A[i]}$
$C_{HLL} \leftarrow \alpha_A \frac{m^2}{Z}$
if $C_{HLL} \leq 5m$ **then**
  $C_{HLL} \leftarrow C_{HLL} - bias_p(C_{HLL})$
return C_{HLL}

3.4. Estimación de frecuencia

Las estructuras de frecuencia están diseñadas para mantener una tabla con información con la cual es posible estimar la cantidad de veces que un elemento se ha insertado. Esta tabla utiliza una fracción de la memoria necesaria para obtener la frecuencia exacta, y tiene un crecimiento sublineal respecto a la cantidad de elementos insertados. Estas estructuras utilizan funciones hash para distribuir los elementos de entrada de forma uniforme dentro de los límites de memoria utilizados, pudiendo generar colisiones, las que son resueltas de diferente forma según la implementación. Estas estructuras suelen utilizarse en aplicaciones con alto flujo de datos como cálculo de entropía [79] o detección de *heavy hitters* [36].

3.4.1. Count Sketch

Count Sketch [73] es un algoritmo de estimación de frecuencia con un uso eficiente de memoria. Este algoritmo utiliza una matriz de $p \times m$ contadores para almacenar la frecuencia de los elementos de entrada. Para cada elemento a añadir en la estructura se aplican $2p$ funciones hash, dos por cada fila de la matriz de contadores. Como muestra la Figura 3.2, la primera función hash h selecciona el contador o bucket asociado al elemento de entrada en cada una de

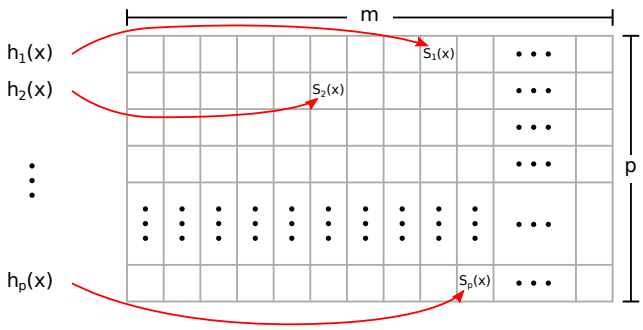


Figura 3.2: Ilustración inserción Count Sketch.

Algoritmo 5: Inserción Count Sketch

Input: Elemento x
Input: Count Sketch CS de dimensiones $p \times m$
Output: Count Sketch CS

```
for i = 0 to p − 1 do
    j ← h_i(x)
    if s_i(x) ≠ 0 then
        CS[i][j] ← CS[i][j] − 1
    else
        CS[i][j] ← CS[i][j] + 1
return CS
```

las filas de la matriz. La segunda función hash s indica si el bucket debe ser incrementado o decrementado. El proceso de inserción, detallado en el Algoritmo 5, puede generar contadores con valores negativos. Es por esto que en el proceso de consulta, definido en el Algoritmo 6, se utiliza la mediana de los p contadores como la estimación de la frecuencia del elemento de entrada.

Este sketch puede generar una sub o sobre estimación de la frecuencia de un elemento debido al algoritmo de inserción que utiliza. La selección de los parámetros está asociada al error de estimación ε y a la probabilidad de mala estimación que tiene un error estándar δ. A mayor número de filas, el estimador tiene más contadores, por lo que δ se reduce, así, la recomendación de número de filas del sketch es:

$$p = \left\lceil ln\frac{1}{\delta} \right\rceil . \tag{3.9}$$

De igual forma, al aumentar el número de contadores se reduce el error de estimación ε por la reducción de colisiones entre los contadores. Luego, la recomendación de número de contadores

Algoritmo 6: Consulta Count Sketch

Input: Elemento x
Input: Count Sketch CS de dimensiones $p \times m$
Output: Estimación de frecuencia $f(x)$

```
Let
    f = {f_i}_0^{p-1}
for i = 0 to p − 1 do
    j ← h_i(x)
    f_i ← CS[i][j]
return mediana (f_0, f_1..., f_{p−1})
```

o buckets es:

$$m \approx \left\lceil \frac{2{,}71828}{\varepsilon^2} \right\rceil . \tag{3.10}$$

3.4.2. Count-Min Sketch

Count–Min Sketch [65] fue propuesto por Graham Cormode y Shan Muthukrishnan. Al igual que el Count Sketch, Count-Min Sketch utiliza una matriz de $p \times m$ contadores para almacenar la frecuencia de los elementos de entrada, pero en lugar de incrementar o decrementar estos contadores dependiendo de una función hash, estos siempre se incrementan en una unidad. El proceso de inserción, detallado en el Algoritmo 7 e ilustrado en la Figura 3.3, solo puede generar una sobreestimación de la frecuencia de los elementos de entrada, dado por las colisiones en los contadores. Producto de lo anterior, la mejor estimación de la frecuencia de un elemento corresponde al mínimo de los contadores asociado a él, como se detalla en el Algoritmo 8. Las dimensiones recomendadas para Count-Min Sketch son:

$$p = \left\lceil ln\frac{1}{\delta} \right\rceil \tag{3.11}$$

y

$$m \approx \left\lceil \frac{2{,}71828}{\varepsilon} \right\rceil . \tag{3.12}$$

De lo que se puede deducir que Count-Min Sketch tiene un uso de memoria menor a Count Sketch para un mismo error.

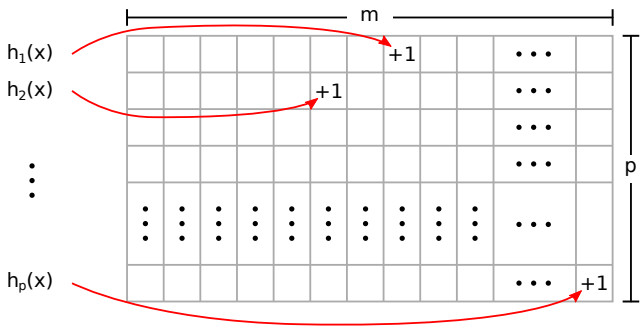


Figura 3.3: Ilustración inserción Count–Min Sketch.

Algoritmo 7: Inserción Count-Min Sketch

Input: Elemento x
Input: Count-Min Sketch CM de dimensiones $p \times m$
Output: Count-Min Sketch CM

for $i = 0$ **to** $p - 1$ **do**
    $j \leftarrow h_i(x)$
    $CM[i][j] \leftarrow CM[i][j] + 1$
return CM

3.4.3. Count-Min Sketch con actualizaciones conservadoras (CU)

Count-Min Sketch con actualizaciones conservadoras [85] (o Count-Min CU) es una variante de Count-min Sketch donde se aplica el concepto de actualización conservadora o *conservative updates (CU)* [86], en la cual solo se incrementan los contadores asociados a un flujo de red que no han sobrepasado un umbral definido, reduciendo los falsos positivos en dicha aplicación. Count-Min CU utiliza esta idea en el proceso de inserción, incrementando solo el contador con menor valor, tal como muestra el Algoritmo 9. La cota de error máxima de este sketch es la

Algoritmo 8: Consulta Count-Min Sketch

Input: Elemento x
Input: Count-Min Sketch CM de dimensiones $p \times m$
Output: Estimación de frecuencia $f(x)$

Let
    $f = \{f_i\}_0^{p-1}$
for $i = 0$ **to** $p - 1$ **do**
    $j \leftarrow h_i(x)$
    $f_i \leftarrow CM[i][j]$
return mín $(f_0, f_1..., f_{p-1})$

Algoritmo 9: Inserción Count-Min Sketch CU

Input: Elemento x
Input: Count-Min Sketch CU CU de dimensiones $p \times m$
Output: Count-Min Sketch CU CU

```
1  Let
2      f = {f_i}_0^{p-1}
3  for i = 0 to p − 1 do
4      j ← h_i(x)
5      f_i ← CU[i][j]
6  m ← mín(f_0, f_1..., f_{p−1})
7  for i = 0 to p − 1 do
8      j ← h_i(x)
9      if CU[i][j] == m then
10         CU[i][j] ← CU[i][j] + 1
11 return CU
```

Algoritmo 10: Consulta Count-Min Sketch CU

Input: Elemento x
Input: Count-Min Sketch CU CU de dimensiones $p \times m$
Output: Estimación de frecuencia $f(x)$

```
1  Let
2      f = {f_i}_0^{p-1}
3  for i = 0 to p − 1 do
4      j ← h_i(x)
5      f_i ← CU[i][j]
6  return mín (f_0, f_1..., f_{p−1})
```

misma que para Count-Min Sketch [87], pero en la práctica se ha visto que presenta un error inferior para las mismas dimensiones [85, 36]. Al igual que Count-Min Sketch, Count-Min CU usa un arreglo de $p \times m$ contadores para almacenar la frecuencia de los elementos de entrada y, como muestra el Algoritmo 10, genera una estimación encontrando el valor mínimo de los contadores asociados a un elemento de entrada.

Capítulo 4. Similitud usando índice Jaccard

4.1. Introducción

Uno de los métodos más utilizados para el cálculo de similitud entre genomas es el índice Jaccard, el cual presenta una forma sencilla de calcular la similitud usando teoría de conjuntos y la cardinalidad entre ellos.

En este trabajo se diseñó un algoritmo y acelerador en GPU y FPGA para el cálculo del índice de Jaccard utilizando un sketch HyperLogLog. Este acelerador está dividido en dos etapas: la construcción de sketches y el cómputo de la matriz de similitud. La primera etapa realiza la construcción de sketches de diferentes genomas en múltiples hebras en hardware, escribiendo el contenido final en el dispositivo de almacenamiento del host. La segunda etapa lee el contenido de los sketches y realiza una operación de unión entre todos las posibles combinaciones de genomas. El producto final de ambas etapas es la matriz triangular superior con todas las similitudes o todos los índices Jaccard. Un ejemplo de este proceso se muestra en la Figura 4.1, donde se ilustran las dos etapas del proceso, con un enfoque en los algoritmos y su implementación en el acelerador.

Las contribuciones de este trabajo son:

- Un algoritmo de streaming que utiliza paralelismo para construir sketches de genomas en aceleradores en hardware.
- La arquitectura del acelerador y el algoritmo de comunicación de datos entre el acelerador y el host.
- El uso de un criterio de selección para reducir la cantidad de similitudes calculadas utilizando las cardinalidades individuales de los conjuntos.
- Una comparación del desempeño de las implementaciones sobre infraestructura en la nube de costo similar entre ellas.

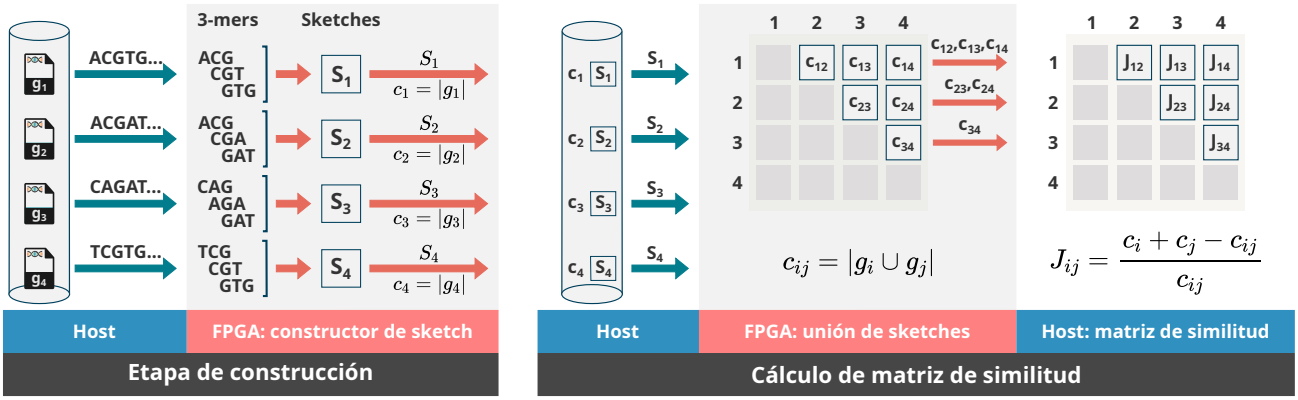


Figura 4.1: Ejemplo simplificado de la construcción de sketches y cálculo de matriz de similitud para 4 genomas, usando k-mers de largo 3. Por simplicidad, las cardinalidades se muestran como calculadas completamente en el FPGA, en la implementación el último paso se realiza en el host.

4.2. Métodos

En esta sección se presentan los métodos para el cálculo de la matriz de similitud y los algoritmos utilizados en ambas etapas de procesamiento. Además, se presentan los flujos de datos entre el host y el acelerador.

4.2.1. Índice Jaccard

El índice Jaccard o similitud de Jaccard fue propuesta por Paul Jaccard [88] y es una métrica popular en el cálculo de similitud entre genomas. Esta similitud se define con base en la cardinalidad de dos conjuntos X e Y, y más específicamente como:

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}, \tag{4.1}$$

donde $|\cdot|$ es la cardinalidad de un conjunto. Usando el principio de inclusión-exclusión es posible definir la similitud como:

$$J(X,Y) = \frac{|X| + |Y| - |X \cup Y|}{|X \cup Y|}, \tag{4.2}$$

evitando el cómputo de la intersección y permitiendo que el índice sea calculado utilizando una estructura de cardinalidad como HyperLogLog. En este trabajo, cada genoma es representado como un conjunto de k-mers canónicos.

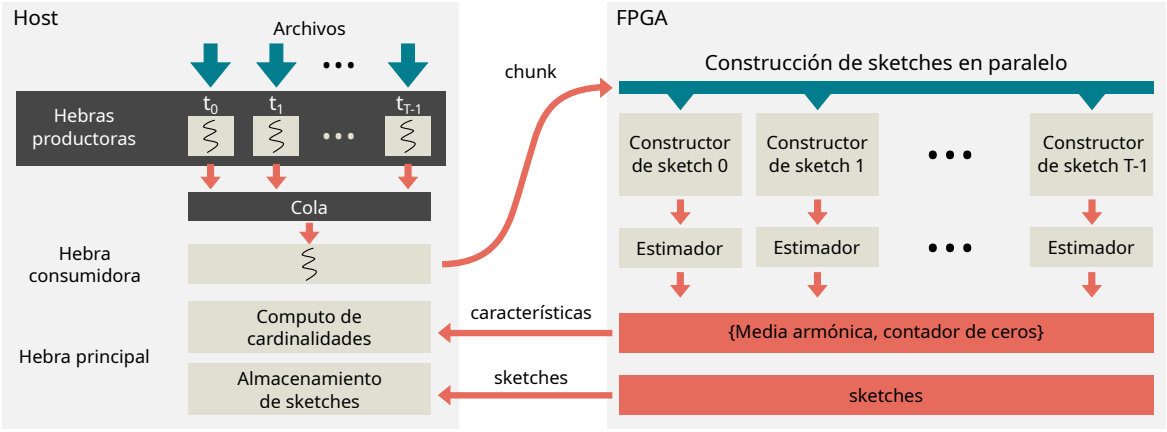


Figura 4.2: Flujo de datos en la etapa de construcción. En el host T hebras productoras leen y descomprimen datos de genomas en paralelo. Una hebra consumidora empaqueta los datos en chunks, combinando secuencias de todas las hebras productoras, y transfiriéndolos a la memoria del acelerador. El acelerador tiene T módulos de construcción de sketch, que leen los datos y construyen un sketch HLL en paralelo para cada genoma. Internamente, esta construcción se realiza con B sketches para maximizar el flujo de datos. Después de leer todo el contenido de los archivos, el módulo de construcción de sketch calcula las características necesarias para estimar la cardinalidad del genoma al genoma (media armónica y el número de buckets en cero), transfiriéndolas al host junto al contenido del sketch.

4.2.2. Etapa de construcción

La etapa de construcción corresponde a la población de los sketches HyperLogLog utilizando los genomas de la base de datos, donde cada genoma es tratado como un conjunto de k-mers canónicos. La Figura 4.2 muestra el flujo de datos entre el host y el acelerador. En el lado de host se utiliza un proceso multi-hebras utilizando un patrón consumidor/productor para solapar la lectura de genomas desde el archivo de entrada con la transferencia de estos al acelerador, y el procesamiento en el acelerador. El host utiliza T hebras productoras para leer y descomprimir los genomas desde el sistema de archivos en formato FASTA/FASTQ utilizando zlib [89], y, en el caso de la implementación en FPGA, una hebra consumidora que se encarga de empaquetar las lecturas de los productores en bloques y enviarlas al acelerador. En el caso de la implementación en GPU, debido al modelo de programación dispuesto por CUDA, las hebras productoras están encargadas de hacer la transferencia al acelerador, ya que cada hebra en hardware utiliza su propio CUDA stream. La implementación del patrón productor/consumidor en el host fue realizada utilizando la librería Jiffy [90], que permite la inserción desde diferentes hebras productoras con una sola hebra consumidora.

El Algoritmo 11 muestra el procedimiento realizado en las hebras productoras en la imple-

Algoritmo 11: Hebra productora en implementación con FPGA

Input: índice de hebra t, tamaño de bloque K, cola q, pila de archivos *files*

```
1  while len(files) > 0 do
2      f ← files.pop();
3      while not end_of_file(f) do
4          data ← f.read(K);
5          q.push(t, data);
6      q.push(t, eof_mark);
```

Algoritmo 12: Hebra consumidora en implementación con FPGA

Input: número de hebras T, tamaño de bloque K, bloques en chunk por hebra C, cola q

```
1  Let
2      offset = 0
3      Inicializa chunk como un arreglo de tamaño C × T × K
4      for i ← 0 to T − 1 do
5          Inicializa queue[i] como una cola de bloques
6  while reading files do
7      if not q.empty () then
8          t, data ← q.pop()
9          queue[t].push(data)
10     if all_nonempty(queue) then
11         for i ← 0 to T − 1 do
12             chunk[offset + i × K] ← queue[i].pop()
13         offset ← offset + T × K
14         if offset == C × T × K then
15             parallel_sketch_kernel (chunk)
```

mentación en FPGA. Cada hebra obtiene un archivo FASTA/FASTQ de un genoma desde la lista de genomas, lee y descomprime este archivo y encola cada uno de los bloques en la cola compartida con el consumidor. Posterior a cada archivo, la hebra productora encola un bloque constante que le indica al acelerador que el archivo ha terminado. El Algoritmo 12 muestra el procedimiento realizado en la hebra consumidora. Esta lee los bloques de la cola compartida y los distribuye en T colas individuales, una para cada hebra productora. Posteriormente, cuando cada cola individual posee al menos un bloque, la hebra consumidora escribe los bloques en el *chunk* de datos (Pasos 10 a 13 en Algoritmo 12), hasta completar el chunk, que luego es enviado al acelerador para su procesamiento. El Algoritmo 13 describe el proceso de lectura en la implementación en GPU, la que difiere levemente de la vista previamente. En este caso, cada

Algoritmo 13: Hebra del host en implementación con GPU

Input: índice de hebra t, tamaño de bloque K, cola q, pila de archivos *files*

```
1 while len(files) > 0 do
2     f ← files.pop();
3     while not end_of_file(f) do
4         data ← f.read(K);
5         q.push(t, data);
6     q.push(t, eof_mark);
```

Algoritmo 14: Acelerador: construcción de sketches en paralelo

Input: bloque de datos *chunk*

```
1  Let
2      largo k-mer k = 31
3      número de hebras T = 8
4      número de sketches HLL internos B = 64/T
5      for i ← 0 to T − 1 do
6          Inicializa line_buffer[i] como un arreglo de bits de tamaño 2(k + B − 1)
7          Inicializa sketch[i] como un sketch HLL de tamaño B
8  foreach chunk_word in stream do
9      for t ← 0 to T − 1 do in parallel
10         c ← chunk_word[64t : 64(t + 1) − 1]
11         sketch_builder(B, k, t, c, line_buffer[t], sketch[i])
```

hebra en el host se comunica con un CUDA stream, por lo que no es necesario dedicar una hebra a empaquetar los chunks.

Las implementaciones en GPU y FPGA son equivalentes. En ambas plataformas se utiliza una hebra hardware para cada hebra productora, realizando la lectura de los datos correspondientes e insertándolos al sketch según lo visto en el Capítulo 3. Como describe el Algoritmo 14, para cada *chunk* de datos, el acelerador extraerá los k-mers asociados a cada base, luego cada k-mer es canonizado e insertado al sketch. En el caso de la implementación en FPGA, se construyen $64/T$ sketches en paralelo para un mismo archivo. Esto se debe a un límite físico en el bus entre la memoria externa del FPGA y el chip, el cual es de 512 bits y, como cada base está codificada en caracteres ASCII de 8 bits, cada transferencia tiene un máximo de 64 bases. En este caso posterior a la inserción de todos los elementos se debe calcular la unión entre todos los sketches con información del mismo archivo, tal como se describe en los Pasos 9 y 10 del Algoritmo 15. En el caso de la implementación en GPU, se construye un solo sketch por archivo, por lo que no es necesario este último paso.

Algoritmo 15: Acelerador: sketch builder

Input: número de HLL B, largo k-mer k, índice de hebra t, sección de 64-bit chunk c, buffer de línea *line_buffer*, arreglo de 8 sketches HLL *sketch*

```
1  b ← two_bit_encode(c)
2  line_buffer ← (line_buffer ≪ 2B) | b
3  if line_buffer ≠ eof_mark then
4      for i ← 0 to B − 1 do in parallel
5          k-mer ← line_buffer[2i : 2i + 2k − 1]
6          k-mer_canon ← canonical_kmer(k-mer)
7          sketch[i].insert(k-mer_canon)
8  else
9      for i ← 0 to B − 1 do reduction in parallel
10         o_sketch ← union(o_sketch, sketch[i])
11     hmean ← harmonic_mean (o_sketch)
12     nempty ← count_zeros (o_sketch)
13     store_in_memory (hmean, nempty, o_sketch)
```

En ambas implementaciones el acelerador construye los sketches de todos los genomas indicados en la lista de entrada. Finalmente, una vez que todos los sketches son almacenados en la memoria externa del acelerador respectivo, estos son transferidos al host y escritos al sistema de archivos junto con la cardinalidad asociada a cada uno de ellos, para que la siguiente etapa pueda procesarlos.

4.2.3. Algoritmo paralelo para cómputo de matriz similitud

La similitud entre pares en un conjunto de N genomas es generalmente representada por una matriz triangular superior, con dimensiones $N \times N$ y valores normalizados entre 0 y 1. Una solución directa en una aplicación en software usando una sola hebra necesita de dos bucles anidados. Una solución utilizando múltiples hebras puede reducir el tiempo procesando varias filas de la matriz en paralelo. En ambos casos, cada hebra selecciona un genoma, o pivote, y calcula la similitud entre éste y todos los demás de forma secuencial. El diseño de acelerador propuesto utiliza un enfoque similar a la solución multihebra, realizando el cálculo de V filas de la matriz, o pivotes, en paralelo. El número máximo de pivotes está limitado por los recursos disponibles en hardware.

El Algoritmo 16 muestra el cómputo de la matriz de similitud para un conjunto de N genomas. Antes de invocar al kernel de cómputo de la matriz, el host transfiere todos los

Algoritmo 16: Acelerador: matriz de similitud

Input: número de pivotes V, número de genomas N, número de sketches a leer en paralelo D, conjunto de sketches s
Output: matriz de similitud M

```
1  Let
2      Inicializa P como un arreglo de sketches de tamaño V
3      Inicializa M como la matriz de similitud de N × N
4  for i ← 0 to N/V − 1 do
       // Carga V pivotes y calcula
       // las similitudes entre ellos
5      for j ← 0 to V − 1 do
6          x ← i × V + j
7          do in parallel
8              P[j] ← stream(s[x])
9              for k ← j to 0 do in parallel
10                 y ← i × V + k
11                 M[x, y] ←Jaccard (stream(s[x]), P[k])
       // Itera sobre el stream de entrada leyendo D sketches
       // en paralelo y calculando las similitudes
12     for x ← ((i + 1) × V)/D to N/D − 1 do
13         for k ← 0 to V − 1 do in parallel
14             y ← i × V + k
15             for n ← 0 to D − 1 do in parallel
16                 z ← x × D + n
17                 M[x, y] ←Jaccard(stream(s[z]), P[k])
18 return M
```

sketches de los genomas, construidos en la etapa anterior, a la memoria externa del acelerador. El acelerador puede almacenar V pivotes en la memoria local del chip, los que pueden ser utilizados varias veces en una misma llamada del kernel. El resto de los sketches son leídos desde la memoria externa del acelerador.

En cada iteración, el acelerador usa V pivotes para producir V filas de la matriz de similitud. Cada iteración realiza dos procesos. El primero, descrito en los Pasos 5 a 11 del Algoritmo 16, consiste en copiar los pivotes a la memoria local del acelerador, calculando en el proceso la similitud entre ellos. La similitud es calculada utilizando la Ecuación (4.2), donde $|X|$ e $|Y|$ fueron calculados en la etapa de construcción de sketches y la cardinalidad de la unión $|X \cup Y|$ es calculada mientras se realiza la operación de unión de ambos sketches. Como resultado, el

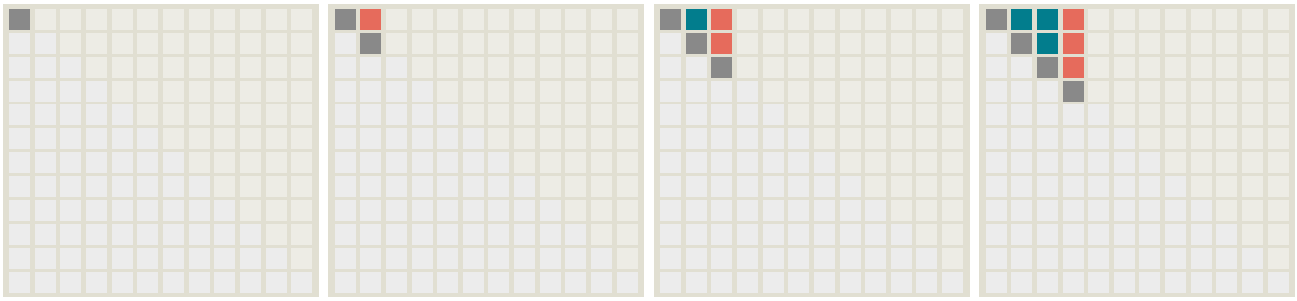
primer proceso calculará las $V(V-1)/2$ similitudes entre los V pivotes. El segundo proceso, descrito en los Pasos 12 a 15 del Algoritmo 16, consiste en el cálculo de las similitudes entre los V pivotes y el resto de los sketches. Los sketches son leídos desde la memoria externa del acelerador. En el caso de la implementación en FPGA se leen D sketches en paralelo, utilizando un único bus de 512 bits de ancho, por lo que se calculan $V \times D$ similitudes Jaccard en paralelo en el kernel, este proceso no almacena datos de los sketches en memoria interna. En la implementación en GPU, el modelo de programación no expone las interfaces de lectura a la memoria global, por lo que no es posible tener información de cómo se realiza esto.

La Figura 4.3 ilustra el proceso de cálculo de la matriz de similitud descrito en el Algoritmo 16, con los parámetros $N = 12$, $V = 4$ y $D = 2$. En la figura los rectángulos naranjos representan las similitudes que están siendo calculadas en ese momento, los rectángulos verde-azulados representan las similitudes calculadas en pasos anteriores, y los rectángulos grises en la diagonal corresponden a las similitudes entre el mismo genoma, por lo que no se calculan. La Figura 4.3a muestra la transferencia de los primeros 4 pivotes a la memoria local del acelerador, junto al cálculo de las similitudes respectivas. La Figura 4.3b muestra las iteraciones sobre todos los sketches que no son pivotes, leyendo D simultáneamente y calculando $V \times D$ similitud en paralelo. La Figura 4.3c muestra el resto del proceso: iniciando cada llamada desde el host, transfiriendo los V pivotes a la memoria local del acelerador, calculando las $V(V-1)/2$ similitudes entre ellos, y procesando el resto de los sketches asociados a cada llamada, hasta completar la matriz.

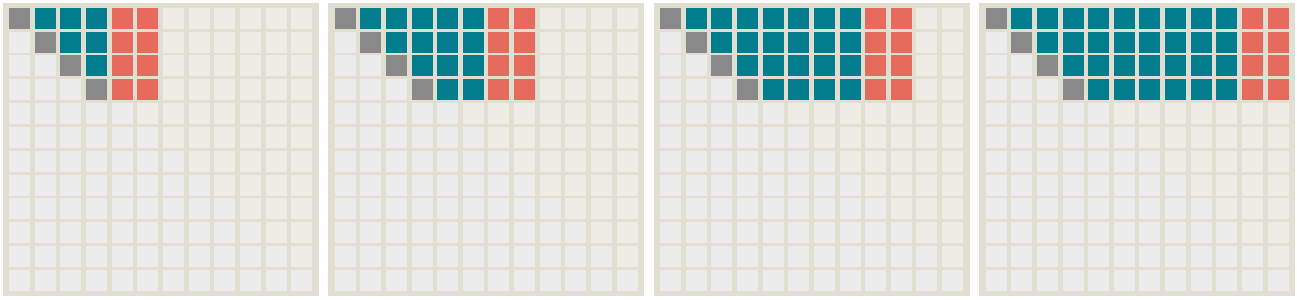
4.2.4. Criterio para selección de pares en cómputo de matriz similitud

En la mayoría de los casos, el cómputo de similitud entre genomas se realiza con el fin de buscar los genomas más parecidos dentro de una conjunto de datos. Comúnmente esto se realiza construyendo la matriz de similitud completa y luego filtrando los datos obtenidos, calculando un total de $N \times (N-1)/2$ similitudes, donde N es el número de genomas.

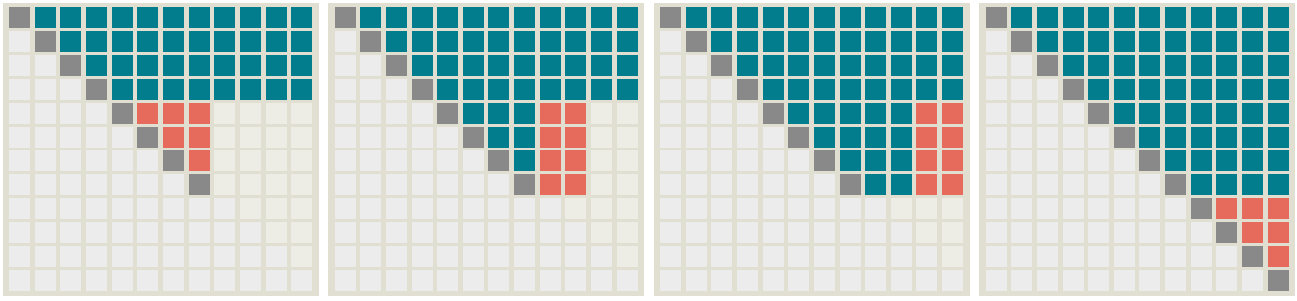
Para reducir la cantidad de similitudes que se calculan, en este trabajo se propuso utilizar un criterio de descarte, evitando calcular pares que no pueden producir similitudes sobre cierto umbral, lo cual es típicamente el criterio de búsqueda en las matrices de similitud [38]. El algoritmo selecciona los pares de genomas que son candidatos para cumplir el criterio utilizando las cardinalidades individuales de cada genoma, las que ya están disponibles desde la primera etapa, descrita en la sección anterior.



(a) Primera iteración del bucle principal: lectura de pivotes y computo de sus similitudes.



(b) Primera iteración: stream de sketches y computo de similitud.



(c) Segunda (primeros tres diagramas) y tercera (cuarto diagrama) iteraciones.

Figura 4.3: Ejemplo de ejecución del Algoritmo 16 para una matriz de similitud de 12×12 genomas. El ejemplo usa $V = 4$ pivotes y lecturas de $D = 2$ sketches de forma simultánea desde la memoria externa. Los cuadrados naranjos representan el cálculo de la etapa ejemplificada. Los cuadrados verde-azulados son las similitudes calculadas en pasos previos y los cuadrados grises en la diagonal son las similitudes de un genoma consigo mismo. En (a) el acelerador carga secuencialmente los primeros V pivotes y calcula las similitudes entre ellos (Pasos 5 a 11 del Algoritmo 16). En (b) el acelerador lee el stream de sketches desde la memoria externa y calcula las similitudes entre ellos y los pivotes residentes en memoria interna, calculando V filas de la matriz (Pasos 12 a 17 del Algoritmo 16). En (c) el acelerador lee un nuevo conjunto de pivotes y el stream de sketches correspondientes, finalmente el acelerador lee los últimos V pivotes, completando la matriz de similitud.

El criterio se basa en los valores posibles que puede tomar el índice de Jaccard. Como se ve en la Ecuación (4.1), el índice de Jaccard entre dos conjuntos X e Y alcanza su valor mínimo cuando la intersección entre ambos conjuntos $|X \cap Y|$ es igual a cero, lo que ocurre cuando ambos conjuntos no tienen valores en común. En este caso, la cardinalidad unión de ambos conjuntos será $|X|+|Y|$. Además, la Ecuación (4.1) tendrá su valor máximo cuando la unión de ambos conjuntos es mínima, lo que ocurre cuando un conjunto contiene completamente al otro, por lo que $|X \cap Y| = \text{máx}(|X|, |Y|)$. Por lo anterior, el valor del índice Jaccard estará entre:

$$0 \leq J(X,Y) \leq \frac{|X| + |Y| - \text{máx}(|X|, |Y|)}{\text{máx}(|X|, |Y|)}. \tag{4.3}$$

Nótese que cuando X e Y son iguales, $|X| = |Y|$ y el índice de Jaccard será igual a uno. Si asumimos que $|X| \leq |Y|$, la Ecuación (4.3) puede reducirse a:

$$J(X,Y) \leq \frac{|X|}{|Y|}. \tag{4.4}$$

Solo conociendo las cardinalidades de X e Y, la Ecuación (4.4) define un límite superior para el índice de Jaccard entre ellos $J(X,Y)$. Si solo se está interesado en aquellos conjuntos para los cuales $J(X,Y) \geq h$, donde h es un umbral predefinido, solo es necesario calcular los índices cuando:

$$h \leq J(X,Y) \leq \frac{|X|}{|Y|}, \tag{4.5}$$

lo que equivale a que $J(X,Y) \geq h$ solo si

$$|Y| \leq \frac{|X|}{h}. \tag{4.6}$$

La Ecuación (4.6) indica que cuando $|Y| > |X|/h$, el índice de Jaccard será $J(X,Y) < h$ y, por lo tanto, los conjuntos X e Y respectivos no podrán producir una similitud mayor o igual al umbral h.

En este trabajo utilizamos el criterio para reducir la cantidad de similitudes calculadas en el Algoritmo 16. Para utilizar el criterio, el host ordena los sketches de menor a mayor cardinalidad antes de subirlos al acelerador, asegurando que $|Y| > |X|$. Así, después de la lectura de los pivotes (Pasos 5 a 11 en el Algoritmo 16), el acelerador no necesita leer todos los sketches desde memoria externa, solo necesita leer aquellos que cumplan el criterio de selección definido en la Ecuación (4.6). Por lo anterior, el límite del ciclo del Paso 12 del Algoritmo 16, ya no estará definido por el número total de genomas N, sino que variará en cada llamada al kernel.

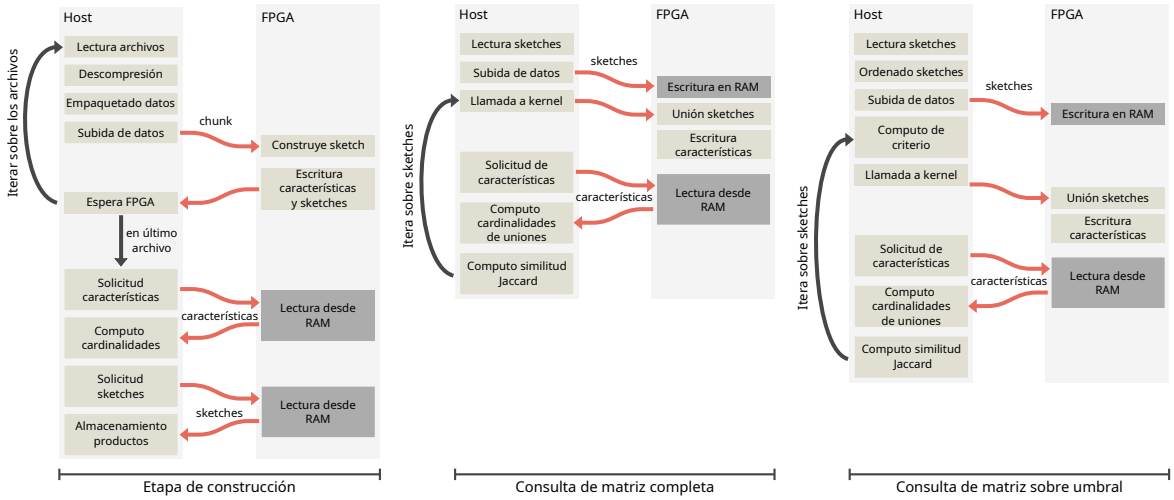


Figura 4.4: Flujo de datos de la implementación en FPGA. En la etapa de construcción, el host utiliza múltiples hilos para leer y descomprimir los archivos de genomas, para luego empaquetarlos y subirlos al FPGA, el que construye los sketches HLL para cada genoma y calcula las características usadas para la estimación de cardinalidad. Los sketches y las características son enviadas al host para calcular la cardinalidad y almacenar los sketches en memoria persistente. En la solicitud de cálculo de la matriz completa, el host sube todos los sketches al FPGA, quien calcula las características de cardinalidad para todas las uniones entre los pares de sketches, enviándolas al host para calcular la similitud de Jaccard de cada una de ellas. En la solicitud de la matriz parcial, usando el criterio de selección, el host ordena los sketches por cardinalidad antes de subirlos al FPGA. Además, para cada llamada al kernel el host evalúa el criterio indicándole al kernel hasta que columna de la matriz se debe realizar el cálculo de uniones, reduciendo el cómputo solo a los genomas capaces de cumplir el criterio.

4.3. Arquitectura en FPGA

La implementación en FPGA incluye dos kernel RTL, uno para cada una de las etapas de procesamiento descritas en la Sección 4.2. El primer kernel realiza la construcción de sketches y el cálculo de las características para la estimación de cardinalidad. El segundo kernel realiza la unión entre un conjunto de V sketches pivotes y un número de genomas definido por el host, que en caso de no utilizar el criterio es igual a N. La Figura 4.4 presenta el flujo de datos entre el host y el FPGA, en el proceso de construcción de sketches y el cálculo de la matriz de similitud. En este último se presentan las dos posibles consultas que puede realizar el host. La primera para el cálculo de la matriz de similitud completa, y la segunda para el uso del criterio. Es importante notar que el kernel soporta ambas consultas, por lo que, como se muestra en la Figura 4.4, es el host el que realiza pasos adicionales para el uso del criterio.

4.3.1. Kernel de construcción de sketches

En la etapa de construcción de sketches, el host configura el FPGA utilizando el *runtime* de Vitis, indicando el archivo binario que contiene la descripción del kernel. Luego, el host ejecuta los algoritmos 11 y 12 para leer, descomprimir y transferir secuencias de genomas al FPGA en sucesivas llamadas al kernel. Cuando el marcador de fin de archivo es detectado, el acelerador guarda el sketch en la memoria externa mientras calcula la suma armónica y el número de buckets en cero, que también son almacenados en la memoria externa. El host lee los sketches, calcula las cardinalidades, los ordena y almacena los resultados en el sistema de archivos.

La Figura 4.5 muestra la arquitectura del acelerador y el proceso visto desde el host. Como se discute en la Sección 4.2.2, el host utiliza T hebras paralelas para leer y descomprimir los archivos FASTQ/FASTA de cada genoma, construyendo *chunks* de datos con los caracteres ASCII que representan a las bases de los T archivos de genomas. El host envía los datos al acelerador en múltiples llamadas al kernel copiando un chunk de datos a la memoria externa del FPGA antes de cada llamado. A su vez, el FPGA tiene T módulos de construcción de sketches que operan en paralelo leyendo datos desde la memoria externa. El kernel lee palabras de 512 bits, limitadas por el ancho máximo del bus del acelerador. Esto corresponde a 64 bases diferentes, por lo que cada módulo recibe $B = 64/T$ bases en cada ciclo durante la lectura. El módulo *k-mer buffer*, representado con azul en la Figura 4.5, transforma cada base a una representación de 2 bits, ignorando cualquier carácter diferente a 'A', 'C', 'G', 'T', y sus versiones en minúscula. Esto nos permite evitar el procesamiento de los archivos para extraer las secuencias de ADN en el host, ya que el acelerador ignora las cabeceras, saltos de líneas y otros metadatos de los archivos FASTQ/FASTA. De acuerdo al Algoritmo 14 el módulo *k-mer buffer* implementa un buffer de línea de $2(k + B - 1)$ bits, que actúa como un registro de desplazamiento, donde k es el largo de un k-mer. En cada ciclo de reloj, el buffer recibe $2B$ bits nuevos, correspondientes a las B bases obtenidas desde el chunk de datos, y calcula B nuevos k-mers, que son utilizados para actualizar el sketch, siguiendo lo descrito en el Algoritmo 1 del Capítulo 3. Lo anterior debido a que dos k-mers adyacentes tienen $k - 1$ bases solapadas.

Para procesar los B k-mers que recibe un módulo de construcción de sketch en cada ciclo, el acelerador construye B sketches en paralelo, uno para cada uno de los k-mers de entrada. Cada k-mer es traducido a su versión canónica antes de insertarlo en el sketch, utilizando un módulo diseñado para este propósito, siguiendo el Algoritmo 18. Toda esta conversión está implementada usando un pipeline para maximizar la frecuencia de reloj del diseño.

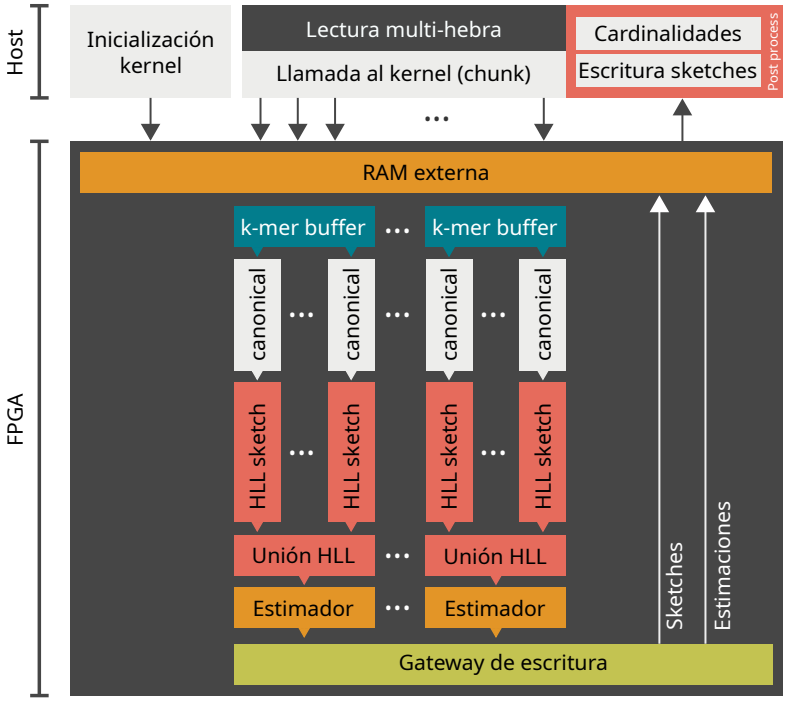


Figura 4.5: Arquitectura del kernel de construcción de sketches, incluyendo la parte del host y del acelerador. La parte gris en lado del host indica el bucle del Algoritmo 12, el cual transfiere chunks de datos al acelerador previo a cada llamada al kernel. El bloque de post-procesamiento en el host, almacena el contenido de los sketches y calcula las cardinalidades de los genomas. En el lado del hardware, el acelerador usa T módulos de construcción de sketches en paralelo, cada uno procesa B k-mers en paralelo. Durante la construcción, cada módulo lee bases desde la memoria externa al buffer de k-mers, posteriormente cada pipeline de procesamiento calcula el k-mer canónico y lo inserta a un sketch HLL. Al finalizar la lectura de un genoma completo, cada módulo hace una operación de unión entre el contenido de los B sketches HLL. El gateway de escritura recibe el resultado del estimador y el contenido del sketch para escribirlos en memoria externa del acelerador.

Cuando el módulo de construcción de sketches detecta el marcador de fin de archivo añadido por el host, envía el contenido de los B sketches a un módulo de unión que calcula el valor máximo entre los B buckets respectivos. Esta operación está completamente implementada en un *pipeline*, usando un árbol de $log_2(B)$ niveles comparadores, con una latencia de $log_2(B)$ ciclos y un flujo de un resultado por ciclo. Los sketches están implementados en memoria interna del chip, usando dos memorias de $1k \times 32$ bits, para una configuración de $1k \times 64$ bits, lo que implementa un sketch de $1k \times 16$ buckets, con 4 bits por bucket. Como la configuración de memoria permite acceder a 16 buckets por ciclo, el módulo de unión realiza la operación de estos en paralelo, por lo que el tiempo de lectura se reduce en un factor de 16. El resultado de la unión de los sketches internos se envía al módulo de estimación y al *gateway de escritura* para copiar el contenido a la memoria externa del acelerador.

Algoritmo 17: MurmurHash3 64 bits

Input: *k-mer*
Output: valor hash h

$k_0 \leftarrow k\text{-}mer \gg 33$
$k_1 \leftarrow k_0 \oplus k\text{-}mer$
$k_2 \leftarrow \texttt{0xff51afd7ed558ccd} \times k_1$
$k_3 \leftarrow k_2 \gg 33$
$k_4 \leftarrow k_3 \oplus k_2$
$k_5 \leftarrow \texttt{0xc4ceb9fe1a85ec53} \times k_4$
$k_6 \leftarrow k_5 \gg 33$
$h \leftarrow k_6 \oplus k_5$
return h

El módulo de estimación calcula la media armónica de la unión de sketches, según la Ecuación (3.6), y la cantidad de buckets vacíos. Para el cálculo de la media armónica, el módulo usa aritmética de punto fijo con 14 bits para la parte entera y 15 para la parte fraccional (Q14.15), lo que es suficiente para capturar la información por el tamaño del sketch ($2^{14} \times 4$ bits) usando en esta implementación. Los valores de media armónica y número de ceros son escritos en la memoria externa utilizando el *gateway de escritura* de la Figura 4.5. El host utiliza estos valores para calcular la cardinalidad de cada conjunto utilizando las Ecuaciones (3.7) y (3.8).

En la implementación actual, el host soporta un máximo de $T = 8$ hebras paralelas, por lo que el kernel de construcción procesa 8 genomas en paralelo utilizando 8 módulos de construcción de sketches, cada uno con $B = 8$ módulos de construcción internos de construcción de HyperLogLog.

4.3.1.1. Módulo extracción de k-mers canónicos

La Figura 4.6 muestra la arquitectura del módulo de extracción de k-mers. En cada ciclo de reloj, el módulo recibe B nuevas bases desde el *chunk* leído desde memoria externa. El módulo codifica las bases usando dos bits y almacenándolas en el buffer de línea de k-mers. Este buffer opera como un registro de desplazamiento, que desplaza B bases en cada ciclo de reloj. El buffer de línea puede almacenar $k + B - 1$ elementos, donde k es un parámetro de diseño y define el largo de k-mers. Después de cada actualización, el módulo entrega B k-mers solapados por $k - 1$ bases entre ellos. Cada k-mer es convertido a su versión canónica en paralelo, usando una secuencia de operaciones binarias y una comparación como se muestra en el Algoritmo 18. Todas las operaciones se ejecutan en un pipeline para maximizar el flujo de salida.

Algoritmo 18: Cálculo de k-mer canónico

Input: *k-mer*
Output: k-mer canónico k_{canon}
Let
  $C_1 = \texttt{0x3333333333333333}$
  $C_2 = \texttt{0x0F0F0F0F0F0F0F0F}$
  $C_3 = \texttt{0x00FF00FF00FF00FF}$
  $C_4 = \texttt{0x0000FFFF0000FFFF}$
$k_0 \leftarrow ((k\text{-}mer \gg 2) \wedge C_1) \vee ((k\text{-}mer \wedge C_1) \gg 2)$
$k_1 \leftarrow ((k_0 \gg 4) \wedge C_2) \vee ((k_0 \wedge C_2) \gg 4)$
$k_2 \leftarrow ((k_1 \gg 8) \wedge C_3) \vee ((k_1 \wedge C_3) \gg 8)$
$k_3 \leftarrow ((k_2 \gg 16) \wedge C_4) \vee ((k_2 \wedge C_4) \gg 16)$
$k_4 \leftarrow (k_3 \gg 32) \vee (k_2 \gg 32)$
$k_5 \leftarrow \neg k_4$
$k_6 \leftarrow k_5 \gg 2$
$k_{canon} \leftarrow min\{k_6, k\text{-}mer\}$
return k_{canon}

El buffer de línea también implementa un conjunto de bits de estado que indican la validez de un k-mer y el fin del procesamiento de un genoma.

4.3.1.2. Módulo HyperLogLog

La Figura 4.7 muestra la arquitectura de un módulo de HyperLogLog, perteneciente al módulo de construcción de sketches. En cada ciclo de reloj, el módulo recibe un k-mer canónico de $2k$ bits como entrada y calcula la hash asociada usando una función MurmurHash3, siguiendo

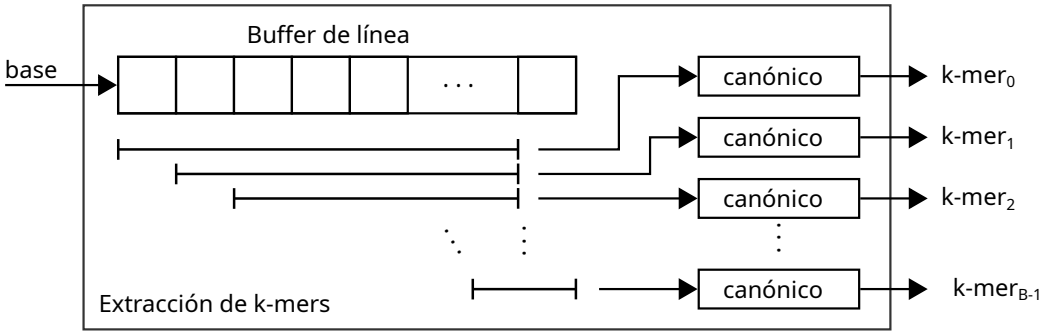


Figura 4.6: Módulo de extracción de k-mer. Cada módulo de construcción de sketches usa un buffer de línea de $k - B - 1$ bases de dos bits. En cada ciclo de reloj, el buffer de línea es actualizado con B bases, mientras el módulo entrega B k-mers nuevos. Cada k-mer es transformado a su versión canónica usando un pipeline que implementa el Algoritmo 18.

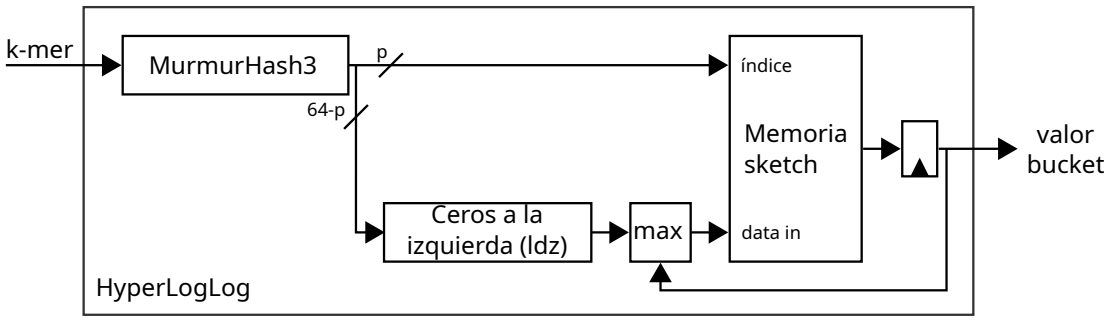


Figura 4.7: Arquitectura del módulo HLL. El componente principal es el bloque de memoria (BRAM), el módulo MurmurHash3 y el detector leading ceros. El módulo implementa lo descrito por el Algoritmo 1.

el Algoritmo 17. Los p bits menos significativos de los 64 bits de salida del módulo de función hash, se utilizan como índice del arreglo de 2^p buckets, correspondientes a la memoria del sketch.

Siguiendo el Algoritmo 1, la posición del uno más significativo de los $64-p$ bits de la hash es usada para actualizar el sketch. La Figura 4.8 muestra el circuito combinacional diseñado para obtener la posición del uno más significativo en palabras de 6 bits. La salida es un uno solo si la entrada correspondiente es uno y no hay otros unos más significativos. La salida de seis bits es codificada a un entero de tres bits con valores entre cero y cinco. El módulo HLL utiliza múltiples módulos de detección del uno más significativo en paralelo, usando un multiplexor con prioridad para obtener la posición global del uno más significativo. Luego, este valor es comparado con el presente en la memoria y, en caso de ser mayor, sobreescribe el valor del bucket.

4.3.2. Kernel de cálculo de similitud

La Figura 4.9 muestra la arquitectura del sistema heterogéneo diseñado para el cómputo de la matriz de similitud, donde el host realiza llamadas al kernel en el acelerador para procesar secciones de la matriz de similitud. El mismo kernel puede procesar las solicitudes de cálculo completo o parcial, usando el umbral del criterio de selección descrito en la Sección 4.2.4. El host controla qué solicitud realizar mediante los parámetros de ejecución del kernel.

Antes de hacer una llamada al kernel, el host sube los sketches a la memoria externa del acelerador. Para habilitar el criterio de selección, el host debe ordenar los sketches según su cardinalidad, de menor a mayor, tal como se discutió en la Sección 4.2.4. Para el cálculo de la matriz completa, el host llama al kernel con los parámetros del número de pivotes (V) y el

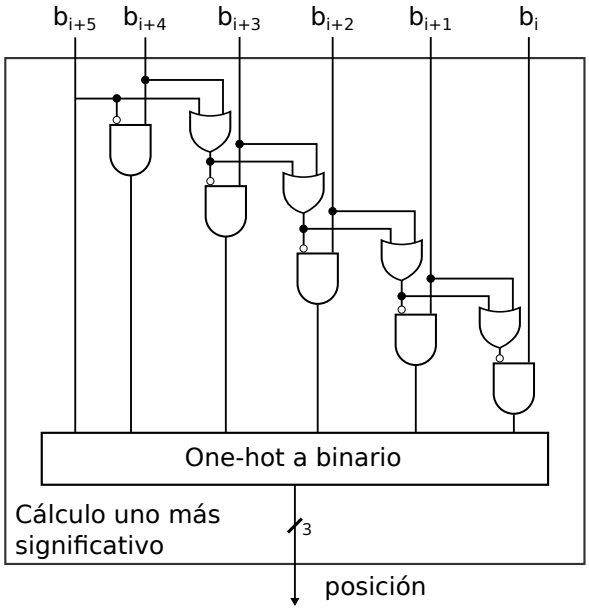


Figura 4.8: Circuito combinacional de obtención del uno más significativo. El circuito usa compuertas lógicas OR para obtener un valor en código termómetro. Luego, una etapa de compuertas AND para obtener una codificación one-hot. Finalmente, se utiliza una tabla de traducción para obtener el número binario asociado a la codificación.

puntero al primer pivote ($i \times V$ en el Algoritmo 16). El acelerador carga los pivotes desde la memoria externa, de uno en uno, copiando su contenido en memorias internas, llamadas *Pivote* en la Figura 4.9. Mientras se cargan los datos de los pivotes, los bloques de unión asociados a cada pivote calculan la unión entre éste y los datos de entrada, correspondientes a otros pivotes en esta etapa, calculando $V \times (V-1)/2$ similitudes entre ellos, como se ilustra en la Figura 4.3a. El módulo de unión es el mismo utilizado en la etapa de construcción, descrita en la Sección 4.3.1. La unión de sketches no es almacenada; ésta es transmitida a un módulo de estimación para calcular la media armónica y la cantidad de buckets en cero. El kernel escribe ambas características en la memoria externa y estas son leídas al final de cada llamada por el host para calcular la similitud de Jaccard usando la Ecuación (4.2) y las cardinalidades individuales calculadas en la etapa de construcción.

Para acelerar el cómputo de similitud entre pivotes, el kernel debe leer la mayor cantidad posible de buckets de un mismo sketch de forma simultánea. Los sketches son almacenados en memoria interna, la que tienen una dimensión máxima de 512×64 bits en FPGA UltraScale+ de Xilinx. Como se discute en el Capítulo 6, el tamaño del sketch HyperLogLog utilizado en esta implementación es de $2^{14} \times 4$-bit, por lo que es posible leer 16 buckets en paralelo al usar memorias de 64-bits de ancho. Para almacenar un sketch completo de 2^{14} buckets, se utilizan dos memorias en paralelo, pudiendo leer o escribir 32 buckets en paralelo. Por esto, el kernel

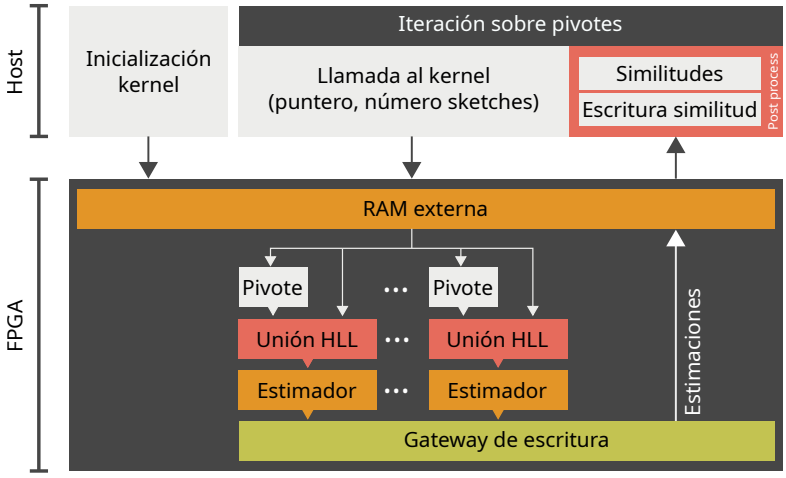


Figura 4.9: Arquitectura del kernel de cálculo de similitud. El host inicializa el kernel subiendo los datos de los sketches ordenados a la memoria externa del acelerador, los que fueron calculados en la etapa de construcción de sketches. Para cada iteración del bucle principal del Algoritmo 16, el host ejecuta la llamada al kernel, el que copia V pivotes al acelerador, guardándolos en una memoria interna del chip (etiquetada como Pivote en la figura) y calculando las uniones entre los pivotes. Posteriormente, el kernel lee un stream de sketches desde la memoria externa, calculando las similitudes entre ellos y los pivotes, calculando V filas de la matriz de similitud. Por simplicidad, la figura solo muestra un módulo de unión por pivote, aun cuando el kernel lee D sketches de forma simultánea, por lo que para cada pivote se calculan D uniones en paralelo.

tiene una interfaz de lectura de memoria externa de 128 bits para los pivotes, que permite leer 32 buckets cada ciclo y calcular la unión entre éste y los pivotes anteriores. La implementación actual soporta un máximo de $V = 80$ pivotes, limitados por los recursos de ruteo disponibles en el FPGA.

En la segunda parte de una llamada al kernel, éste calcula las uniones entre un conjunto de sketches con los pivotes previamente almacenados en memoria interna, calculando V filas de la matriz de similitud, como se muestra en la Figura 4.3b. Esta parte de una llamada al kernel implementa el bucle de los Pasos 12 a 17 del Algoritmo 16, leyendo D sketches en paralelo en cada iteración, calculando la unión entre todos ellos y los V pivotes residentes en el chip. La Figura 4.3b ilustra una ejecución de esta parte de la llamada. A diferencia de los pivotes, estos sketches no son almacenados en memoria interna y solo son utilizados para calcular las uniones con los pivotes. Por simplicidad, la Figura 4.9 solo muestra un módulo de unión por pivote, pero realmente para cada pivote se realizan D uniones en paralelo. En cada llamada al kernel, el host entrega un puntero al primer sketch y la cantidad de sketches a leer desde memoria externa. Para calcular la matriz de similitud completa, el host debe indicar la cantidad de sketches faltantes para completar las filas de la matriz correspondientes a los pivotes, como se muestra en el Paso

12 del Algoritmo 16. Cuando se usa el criterio de selección para solo calcular las similitudes sobre un umbral h, el número de sketches a leer es determinado por la Ecuación (4.6) por lo que el último sketch debe tener a lo más una cardinalidad de $|X|/h$, donde X es el pivote con mayor cardinalidad almacenado en el acelerador.

El valor de D está limitado por los recursos lógicos y de ruteo del FPGA, y por el ancho máximo del bus de comunicación con memoria externa, que es de 512 bits en la plataforma donde se implementó este trabajo. Como es posible leer 32 buckets de los pivotes por ciclo, el acelerador puede leer hasta 128 bits de un mismo sketch, así, el acelerador utiliza el bus de 512 bits para leer $D = 4$ sketches en paralelo desde la memoria externa.

Los resultados de las implementaciones descritas en esta sección se presentan en la Sección 6.4.

4.4. Implementación en GPU

La implementación en GPU incluye las mismas dos etapas de procesamiento que la implementación en FPGA. Por el modelo de programación, cada etapa tiene más de un kernel CUDA para realizar el procesamiento. La etapa de construcción de sketches utiliza un kernel para extraer los k-mers asociados a cada base de un genoma, posteriormente un segundo kernel mantiene en memoria un sketch HyperLogLog insertando cada k-mer válido en él. La segunda etapa calcula la similitud entre un conjunto de genomas pivotes y el resto de genomas del conjunto de prueba. Al igual que en la implementación en FPGA, la implementación en GPU puede utilizar el criterio de selección para reducir el número de cálculos de similitud. Adicionalmente, para la segunda etapa se presentan dos implementaciones: una versión naif que lee los pivotes desde la memoria externa, y una versión usando memoria compartida para almacenar los pivotes, de forma similar al kernel de cálculo de similitud de la implementación en FPGA.

4.4.1. Etapa de construcción de sketches

La implementación en GPU utiliza dos kernel de CUDA para insertar elementos a un sketch HyperLogLog, tal como muestra la Figura 4.10. El primer kernel recibe un chunk de datos de un genoma (c en Algoritmo 19) desde el host, codifica cada base en dos bits y genera un k-mer canónico asociado a cada una de las bases. Este proceso, descrito en el Algoritmo 19, se

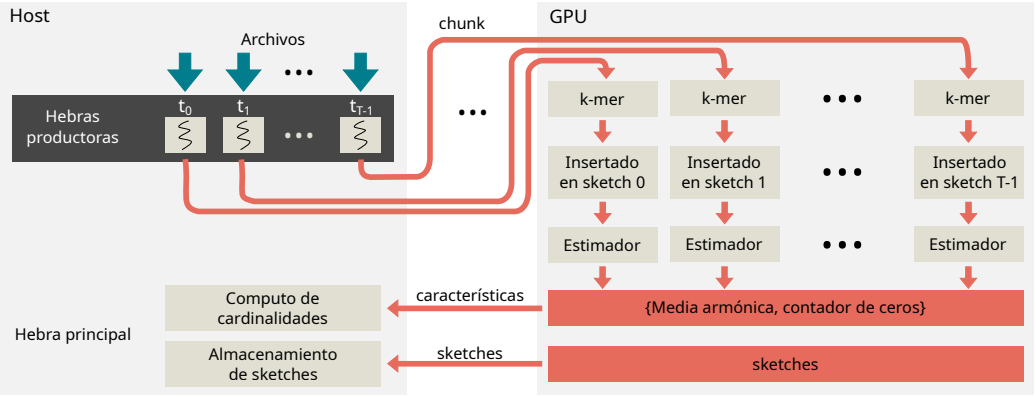


Figura 4.10: Flujo de datos en la etapa de construcción. En el host T hebras leen y descomprimen los datos de genomas en paralelo. Cada hebra envía los datos a la GPU y encola las operaciones para ejecutar en un stream CUDA los kernels necesarios para insertar los elementos al sketch HyperLogLog. Un kernel de estimación calcula las características para calcular las cardinalidades de cada genoma y las envía al host para su procesamiento.

ejecuta en paralelo en n hebras CUDA, escribiendo los resultados de cada k-mer en memoria externa, junto con un indicador de validez, usado para descartar k-mers asociados a cabeceras o caracteres diferentes a una base genómica.

El segundo kernel lee, en n hebras CUDA, los k-mers canónicos con el indicador de validez respectivo desde memoria externa, como describe el Paso 2 del Algoritmo 20. Cada hebra aplica la función hash al k-mer correspondiente, calculando el índice y número de ceros a la izquierda. Estos datos son utilizados para insertar el elemento al sketch HyperLogLog, según el Algoritmo 1. Como todas las inserciones de k-mers van a un único sketch, es posible que existan colisiones entre ellos. Para manejar esto, la modificación del contenido del sketch se realiza con la operación atómica `atomicMax`, modificando el valor solo si éste es mayor al almacenado previamente.

En la ejecución de esta etapa, el host lee datos de los archivos de genomas en un bucle, tal como describe el Algoritmo 13, iterando sobre el contenido y realizando múltiples llamadas a los dos kernels anteriores. Una vez que todo el contenido del genoma fue procesado, el acelerador llama al tercer kernel.

El tercer kernel, llamado Estimador en la Figura 4.10, lee el sketch HyperLogLog desde la memoria externa del GPU y lo recorre para calcular la media armónica y el número de buckets en cero. Este proceso se realiza usando un árbol de reducción donde solo un hilo calcula el valor final y lo escribe en memoria externa para que el host pueda leerlo. Al igual que en la implementación en FPGA, el host escribe en el almacenamiento persistente el contenido de los sketches y las cardinalidades de cada uno de los genomas.

Algoritmo 19: Pseudocódigo kernel extracción de k-mers

Input: chunk de datos c, tamaño chunk n, largo k-mer k
Output: valor k-mer *output*, k-mer valido *valid*

```
1  for idx = 0 to n − k do
2      v ← true
3      o ← 0
4      for i = 0 to k do
5          d ← c[idx + i]
6          o_i ← 0
7          if d == 'A' then
8              o_i ← 0
9          else if d == 'C' then
10             o_i ← 1
11         else if d == 'G' then
12             o_i ← 2
13         else if d == 'T' then
14             o_i ← 3
15         else
16             v ← 0
17         o ← (o ≪ 2) + o_i
18     output[idx] ← canonical(o)
19     valid[idx] ← v
```

4.4.2. Etapa de cálculo de similitud

La etapa de cálculo de similitud realiza un proceso similar al visto en la Sección 4.3.2 para la implementación en FPGA. En este caso, se presentan dos soluciones, ambas realizan el cálculo de la matriz de similitud por partes. En la primera implementación a cada hebra CUDA se le asigna una fila de la matriz de similitud y se itera sobre el resto de la matriz para calcular la fila completa. Este enfoque no es eficiente, ya que hay muchas operaciones de copia de memoria repetidas. Para disminuir el tiempo de ejecución, en este trabajo se planteó una implementación basada en el mismo algoritmo propuesto para la implementación en FPGA, el que utiliza pivotes y está descrito en el Algoritmo 21. En este caso, cada bloque de CUDA lee un pivote asociado a una fila de la matriz de similitud. El pivote es almacenado en la memoria compartida del bloque, que es accesible por todas las hebras del bloque. Posteriormente, las hebras que son parte del bloque calculan en paralelo las similitudes entre el pivote y el resto de los genomas, descrito en el Algoritmo 21 como el bucle del Paso 5. La naturaleza del cómputo de la suma

Algoritmo 20: Pseudocódigo kernel inserción HLL

Input: chunk de datos c, tamaño chunk n, largo k-mer k, valor k-mer $kmers$, k-mer válido $valid$, sketch HLL mem, bits precisión sketch HLL p
Output: sketch HLL mem

```
1 for idx = 0 to n − k do
2     if valid[idx] then
3         h ← hash(kmers[idx])
4         v1 ←< h63, h64−p >2
5         v2 ←< h63−p, h0 >2
6         z ← ldz(v2)
7         mem[idx] ← atomicMax{mem[idx], z}
```

armónica y la cantidad de ceros es secuencial. Para acelerar esto en CUDA, el sketch HLL se divide entre las hebras del bloque, calculando ambos valores para cada segmento, indicado como *parallel_ merge* en el Algoritmo 21. Este cálculo se realiza para el sketch unión y para ambos sketches originales (A y B). Posteriormente, el kernel realiza una reducción en paralelo, sumando los valores parciales de las sumas armónicas y la cantidad de ceros. En el paso final, indicado por los Pasos 12 a 15 del Algoritmo 21, solo una de las hebras del bloque CUDA calcula la estimación de cardinalidad para cada sketch HLL y el índice Jaccard asociado a ellos. Esta implementación sigue el Algoritmo 16, siendo una implementación en GPU del mismo algoritmo implementado en FPGA.

En paralelo al producto de la unión generado por este kernel, se calculan las cardinalidades individuales de cada sketch. Así, una vez obtenida la cardinalidad de la unión, el acelerador calcula la similitud de Jaccard, escribiéndola en la memoria externa. Finalmente, el host almacena los resultados parciales en el almacenamiento persistente.

Los resultados de las implementaciones descritas en esta sección se presentan en la Sección 6.4.

Algoritmo 21: Pseudo código kernel de cálculo de similitud usando pivotes según Algoritmo 16

Input: número de sketches n, offset sketch m, índice bloque b_idx, sketches HLL mem
Output: memoria escritura resultados $output$

```
1  Let
2      pivot as HLL sketch
3  pivot ← mem[b_idx]
4  if b_idx < n then
5      for j = 0 to m do
6          r ← parallel_merge(mem[j], pivot)
7          sincronización de hebras
8          {sum_m, zeros_m} ← parallel_reduction(r)
9          {sum_A, zeros_A} ← parallel_reduction(mem[j])
10         {sum_B, zeros_B} ← parallel_reduction(pivot)
11         sincronización de hebras
12         C_m ← cardinality(sum_m, zeros_m)
13         C_A ← cardinality(sum_A, zeros_A)
14         C_B ← cardinality(sum_B, zeros_B)
15         output[m * b_idx + j] ← jaccard(C_m, C_A, C_B)
```

Capítulo 5. Similitud usando JSD

5.1. Introducción

La divergencia de Jensen-Shannon (JSD) es una métrica utilizada para calcular la similitud entre genomas dentro de otras etapas de procesamiento, como comparación de genomas [33] o estudios de asociación de genomas completos [34]. Esta métrica utiliza la distribución de dos conjuntos de datos, o genomas en este caso, para medir la similitud entre ellos mediante el cálculo de sus entropías individuales y conjuntas.

En este trabajo se diseñó un acelerador en GPU y FPGA para estimar la divergencia de Jensen-Shannon entre pares de genomas. Para esto, se utilizaron sketches de estimación de frecuencia, además de un arreglo de colas de prioridad basado en los algoritmos y aceleradores propuestos por Soto et al. [59]. Este trabajo sigue el patrón de la implementación para cálculo de similitud de Jaccard descrita en el Capítulo 4, utilizando dos etapas: una de construcción y una de cálculo de similitud. En la etapa de construcción, el acelerador puebla los arreglos de colas de prioridad con los k-mers más frecuentes de cada genoma, almacenando el contenido de cada una de estas estructuras en el almacenamiento persistente. En la segunda etapa, de la misma forma que en el Capítulo 4, se utilizan pivotes para construir la matriz de similitud, realizando una operación de unión entre los pares de colas y calculando las entropías individuales y conjuntas para estimar la divergencia de Jensen-Shannon asociada a cada par.

Las contribuciones presentes en este trabajo son:

- Estudio de la contribución de los elementos más frecuentes a la entropía de un genoma.
- Estudio del efecto de muestreo o *sampling* en genomas para el cálculo de similitud.
- Un algoritmo de streaming para la captura de los elementos más frecuentes de genomas en hardware.
- Implementación en GPU y FPGA de estructuras de datos probabilísticas para cálculo de entropía.
- Operación de unión en arreglos de colas de prioridad.

5.2. Métodos

En esta sección se presentan los métodos utilizados para el cálculo de la matriz de similitud usando la divergencia de Jensen-Shannon.

5.2.1. Divergencia de Jensen–Shannon

La divergencia de Jensen-Shannon es una métrica de similitud basada en la divergencia de Kullback-Leibler (KLD) [91], la cual evalúa la semejanza entre las distribuciones de probabilidad de dos conjuntos. A diferencia de Kullback-Leibler, la divergencia de Jensen-Shannon es simétrica, por lo que es posible utilizarla como una medida de similitud. Así, dada dos distribuciones de probabilidad P y Q, la divergencia de Jensen-Shannon será:

$$JSD(P\|Q) = \frac{1}{2}KLD(P\|M) + \frac{1}{2}KLD(Q\|M), \tag{5.1}$$

donde $M = \frac{P+Q}{2}$ y KLD corresponde a la divergencia de Kullback-Leibler. Además,

$$KLD(P\|Q) = \sum_{x\in X} p(x)\log\frac{1}{q(x)} - \sum_{x\in X} p(x)\log\frac{1}{p(x)} = H(P,Q) - H(P), \tag{5.2}$$

por lo tanto, es posible definir la divergencia de Jensen-Shannon con base en la entropía de P y P, junto a la entropía de la unión de ambos conjuntos como:

$$JSD(P\|Q) = H(P,Q) - \frac{1}{2}\left(H(P) + H(Q)\right). \tag{5.3}$$

5.2.2. Entropía de Shannon

La entropía es una medida de incertidumbre que indica la cantidad de información que contiene una distribución de datos. Ésta se define con base en la probabilidad que cada estado dentro de la distribución como:

$$H = -\sum_{i=1}^{N} p_i \log p_i, \tag{5.4}$$

donde N es la cantidad de estados. En una muestra de datos, como lo es un genoma, la probabilidad de cada elemento depende de la disponibilidad de datos, los que se pueden asociar a una

distribución empírica, de la cual se puede obtener una entropía empírica. La entropía empírica está dada por:

$$H = -\sum_{i=1}^{N} \frac{m_i}{M} \log \frac{m_i}{M}, \tag{5.5}$$

donde m_i es el número de ocurrencias del i-ésimo elemento del conjunto y M es número total de elementos en la muestra. Como es posible observar de la ecuación anterior, cada elemento único aporta $\frac{m_i}{M}$ a la entropía total, lo que significa que el mayor aporte estará dado por los elementos más frecuentes de todo el conjunto. Es por esto que Soto et al. [92] propusieron el uso de los elementos más frecuentes, o Top-K, para calcular una entropía representativa del conjunto. En este caso, la entropía es dividida en dos términos,

$$H = -\left[\sum_{i=1}^{K} \frac{m_i}{M} \log \frac{m_i}{M} + \sum_{i=K+1}^{N} \frac{m_i}{M} \log \frac{m_i}{M}\right], \tag{5.6}$$

donde el primer término corresponde al aporte de los elementos más frecuentes a la entropía del conjunto. Así, es posible estimar la entropía del conjunto de datos como la entropía de los elementos más frecuentes [37] de la siguiente forma:

$$\hat{H} = -\sum_{i=1}^{K} \frac{m_i}{L} \log \frac{m_i}{L}, \tag{5.7}$$

donde $L = \sum_{i=1}^{K} m_i$.

5.2.3. Muestreo de k-mers

Los genomas, representados como un conjunto de k-mers, tienen una gran cantidad de elementos distintos. La base de datos utilizada en este trabajo tiene una media de 12 925 647 k-mers, para $k = 31$, lo que implica un alto uso de memoria para almacenar la frecuencia de cada elemento distinto. Además, como se ve en la Sección 6.5, el cálculo de la entropía solo usando los elementos más frecuentes no genera una buena aproximación de ésta. Para esto, en este trabajo se propuso hacer un muestreo aleatorio de los k-mers, reduciendo el número de elementos distintos, pero manteniendo la distribución de los genomas. Un enfoque que ya ha sido utilizado en genómica en trabajos como ntCard [52] o KmerStream [93].

El muestreo utiliza una función hash para distribuir de forma uniforme los k-mers. Luego, aquellos k-mers que generen una salida de la función hash con s bits iguales a cero, son utilizados,

Algoritmo 22: Muestreo de k-mer

Input: k-mer x
Output: flag validez v

```
1 h ← hash(x)
2 if < h_s, h_0 >_2 == 0 then
3     v ← 1
4 else
5     v ← 0
6 return v
```

mientras el resto se descarta, lo que corresponde a un muestreo en un factor de 2^s. Esta lógica se ve en el Algoritmo 22, usando un *flag* para indicar la validez de un k-mer y el posterior uso o no de este.

5.2.4. Arreglo de colas de prioridad (PQA)

El arreglo de colas de prioridad (Priority Queue Array, PQA), propuesto por Soto et al. [59] es una estructura que reemplaza una cola de prioridad, por una tabla hash con una cola de prioridad de tamaño fijo en cada bucket. Esta estructura reduce el tiempo de inserción en un factor igual a la cantidad de colas a usar. El error introducido por esta alternativa es pequeño y permite calcular la entropía del conjunto de forma efectiva [59]. El Algoritmo 23 describe el proceso de inserción, donde se aplica una función hash al elemento de entrada y se usan los R bits menos significativos para seleccionar la cola a usar en el arreglo, tal como se ilustra en la Figura 5.1. El resto de los bits se usa como un identificador del elemento y es usado para diferenciar los elementos presentes dentro de la cola. El par identificador-frecuencia es comparado con los elementos de la cola seleccionada, reemplazando, insertando o descartandolo según corresponda, como indica el Algoritmo 23 entre los Pasos 2 y 6. Este proceso tiene un tiempo de $O(n)$, ya que es necesario comparar el par de entrada con los n elementos presentes en la cola. En una implementación en hardware, dependiendo de la cantidad de elementos por cola, es posible realizar la inserción en solo un ciclo, reduciendo la complejidad temporal a $O(1)$.

El PQA es utilizado para almacenar los elementos más frecuentes de un conjunto. Como la estructura almacena un identificador de cada elemento ingresado, es posible realizar una operación de unión entre PQA. Lo anterior permite el cálculo de características conjutas entre estructuras, como la entropía. El Algoritmo 24 describe la operación de unión, donde se genera un PQA de hasta $2^R \times 2n$ elementos, lo que ocurre cuando ambos PQA no comparten ningún

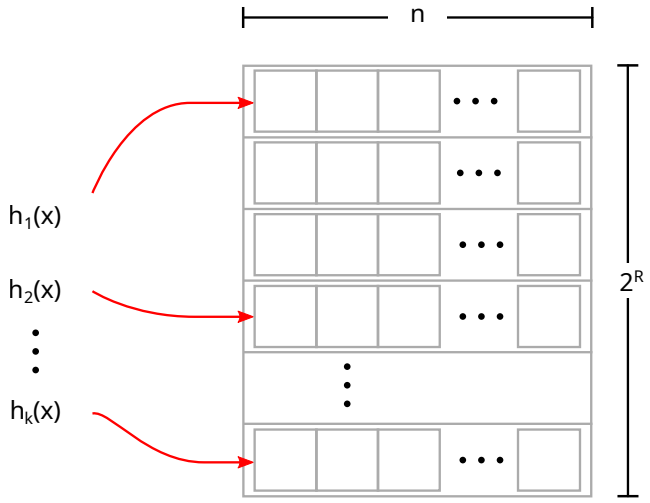


Figura 5.1: Ilustración de inserción a arreglo de colas de prioridad.

elemento, ejecutando solo los Pasos del 7 al 12 del Algoritmo 24.

5.2.5. Algoritmo de construcción de colas

La etapa de construcción corresponde a la creación de un arreglo de colas de prioridad para cada genoma, almacenando las frecuencias de los elementos más frecuentes de cada genoma, los cuales son necesarios para estimar la entropía. La Figura 5.2 muestra el flujo de datos entre el host y el acelerador, incluyendo la extracción de archivos en paralelo en T hebras del host y su procesamiento en el acelerador con T módulos de construcción independientes. Cada módulo de construcción también realiza el cálculo de entropía.

En el host, cada hebra lee y descomprime un genoma utilizando zlib [89]. En la implementación en FPGA, el host usa una hebra consumidora para empaquetar los bloques de lectura en un único bloque de datos para transferir al FPGA. En la implementación en GPU, cada hebra se comunica de forma independiente con un CUDA stream, por lo que no hay necesidad de empaquetar los datos. La lectura de datos en ambas implementaciones, sigue el proceso descrito en el Capítulo 4 en los Algoritmos 11, 12 y 13. El procesamiento en el acelerador difiere del Capítulo 4, por la diferencia en los sketches a utilizar.

En el FPGA, el flujo de datos de entrada se divide en T módulos de procesamiento diferentes. Cada módulo realiza una inserción al sketch de frecuencia, una actualización al arreglo de colas de prioridad y una actualización parcial de los acumuladores necesarios para el cálculo de

Algoritmo 23: Inserción arreglo de colas de prioridad (PQA)

Input: Elemento x, frecuencia elemento e
Input: Arreglo colas de prioridad PQ de dimensiones $2^R \times n$
Output: Arreglo colas de prioridad PQ

```
1  h ← h(x)
2  idx ←< h_{R-1}, h_0 >_2
3  tag ←< h_31, h_R >_2
4  if tag ∈ PQ[idx] then
5  |  for j = 0 to n − 1 do
6  |  |  if PQ[idx][j].tag == tag then
7  |  |  |  PQ[idx][j].value ← e
8  else if x > mín(PQ[idx]) then
9  |  PQ[idx][n − 1] ← x
10 sort(PQ[idx])
11 return PQ
```

entropía. Este proceso, ejemplificado en la Figura 5.2, se repite hasta que el módulo detecta una marca de fin de archivo, tras lo cual se escribe el contenido del arreglo de colas a la memoria del acelerador. En paralelo a la escritura en memoria, el módulo calcula la entropía asociada, la cual se escribe en memoria en una posición siguiente a la última dato del arreglo de colas.

En GPU, cada stream CUDA procesa un archivo, aplicando un kernel de codificación sobre los datos, luego un kernel de inserción al sketch de frecuencia y finalmente uno de actualización al arreglo de colas. Al término del procesamiento de un archivo, se copia el contenido del arreglo de colas a la memoria del host. En esta implementación no se calcula la entropía en la etapa de construcción, pues requeriría leer todos los elementos del arreglo de colas nuevamente.

Los Algoritmos 25 y 26 muestran una descripción en alto nivel de las operaciones realizadas por los aceleradores, indicado que el cálculo de entropía solo se realiza en FPGA con el *flag* *isFPGA*. El Algoritmo 25 muestra la etapa de división por genoma de las palabras de 512 bits de entrada al acelerador (Paso 10) y el envío de estas al módulo de construcción de PQA en el Paso 11. El Algoritmo 26 muestra en detalle las operaciones que realiza el módulo de construcción de PQA, realizando la extracción de k-mers canónicos (Pasos 4 y 5), el muestreo (Paso 6) y la inserción al sketch de cuentas y posteriormente al PQA en los Pasos 7 y 8. La escritura a memoria y el cálculo de entropía se realizan en los Pasos 10 a 14 del Algoritmo 26. Las Secciones 5.3 y 5.4 detallan la arquitectura e implementación en FPGA y GPU respectivamente.

Algoritmo 24: Unión de PQA

Input: Arreglo colas de prioridad $PQ1$ y $PQ2$ de dimensiones $2^R \times n$
Output: Unión de arreglos uPQ de dimensiones $2^R \times 2n$

```
1  for i = 0 to 2^R − 1 do
2      for j = 0 to n − 1 do
3          for k = 0 to n − 1 do
4              if PQ1[i][k].tag == PQ2[i][j].tag then
5                  uPQ[i][j].value ← PQ1[i][k].value + PQ2[i][j].value
6                  uPQ[i][j].tag ← PQ1[i][k].value
7          if PQ1[i][j].tag ∉ uPQ[i] then
8              uPQ[i][j].tag ← PQ1[i][j].tag
9              uPQ[i][j].value ← PQ1[i][j].value
10         if PQ2[i][j].tag ∉ uPQ[i] then
11             uPQ[i][n + j].tag ← PQ2[i][j].tag
12             uPQ[i][n + j].value ← PQ2[i][j].value
13 return uPQ
```

5.2.6. Algoritmo paralelo para cómputo de matriz de similitud

En la similitud usando JSD, el cálculo de la matriz de similitud requiere del cálculo de la entropía de las uniones de PQA, necesarias para evaluar la Ecuación (5.3). Este proceso se realiza usando el enfoque de pivotes descrito en el Capítulo 4, el que permite paralelizar el cómputo de las uniones al almacenar un número de PQA en memoria interna del acelerador y calcular uniones en paralelo. El enfoque de pivotes para el cálculo de similitud Jaccard no puede ser utilizado de forma directa en este caso, debido a que los arreglos de colas de prioridad son de mucho mayor tamaño que los sketches de cardinalidad. De esta forma, en esta sección se describe la estrategia de pivotes para similitud con JSD usando submatrices en GPU y múltiples instancias de un kernel en FPGA.

En la implementación en GPU, la matriz de similitud es dividida en submatrices debido a restricciones de memoria. En este caso, el acelerador recibe dos conjuntos de arreglos de colas, uno para utilizar como pivotes y el otro para hacer las operaciones de unión contra esos pivotes. Este enfoque aumenta la cantidad de copias a memoria, pero habilita el uso de conjuntos de datos de mayor tamaño. El Algoritmo 27 describe este proceso en alto nivel del cálculo de una submatriz, donde se realiza la carga de pivotes y el cálculo de JSD entre ellos en los Pasos 4 a 7 del Algoritmo 27, para luego procesar el resto de los PQA que son leídos desde memoria externa y guardar el resutado en la posición correspondiente de la matriz (Pasos 8 a 11). El uso

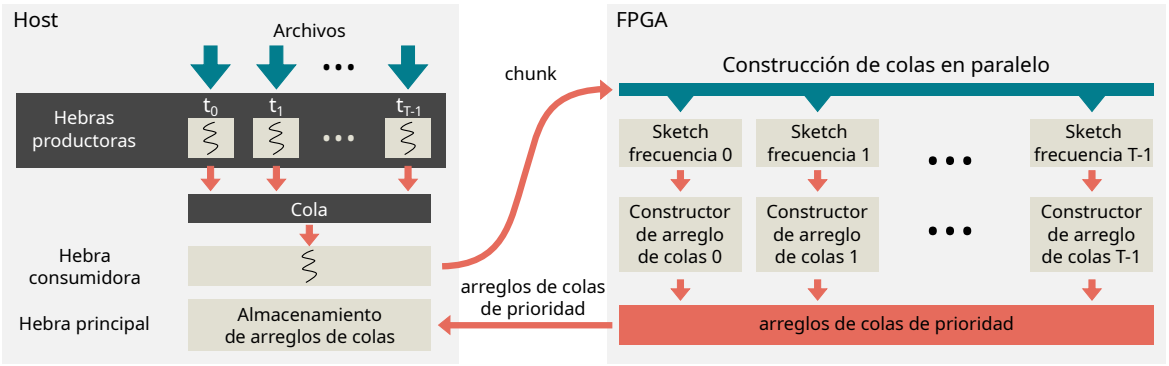


Figura 5.2: Flujo de datos en la etapa de construcción. En el host T hebras productoras leen y descomprimen datos de genomas en paralelo. Una hebra consumidora empaqueta los datos en chunks, combinando secuencias de todas las hebras productoras, y transfiriéndolos a la memoria del acelerador. El acelerador tiene T módulos de construcción de colas, que leen los datos, extraen los k-mers de las secuencias y los insertan en un sketch de frecuencia en paralelo. Cada inserción en el sketch tiene como salida la estimación de frecuencia, la cual es insertada al arreglo de colas correspondiente. Al finalizar el proceso de construcción del arreglo de colas, el kernel transfiere el contenido a la memoria externa del acelerador para ser leídos desde el host al final de esta etapa.

de submatrices requiere de la coordinación del host, el cual debe realizar las copias de memoria necesarias y las llamadas al kernel con los punteros correspondientes. El procesamiento interno de cada submatriz es equivalente al presentado en el Capítulo 4 para el cálculo de similitud Jaccard, donde cada bloque de CUDA lee un pivote de la submatriz y calcula las uniones y divergencias de Jensen-Shannon entre dicho pivote y los arreglos de colas correspondientes.

Los aceleradores con FPGA presentan una configuración de memoria diferente a las GPU, ya que en éstos es posible acceder de forma paralela e independiente a las interfaces de memoria disponibles. Por esto, es posible realizar el cálculo de la matriz de similitud usando submatrices en paralelo con múltiples instancias de un kernel procesando datos desde diferentes interfaces, lo que es explicado con mayor detalle en la Sección 5.3.2. En alto nivel, el host distribuye los arreglos de cola de prioridad entre las memorias del acelerador, y llama a los kernels indicando un grupo de pivotes a usar. En cada llamada, cada instancia del kernel copia los mismos pivotes a memoria interna (Pasos 5 a 10 de Algoritmo 28) y lee un conjunto de arreglos de colas de prioridad distinto en cada instancia, eso se muestra en los Pasos 11 a 14 del Algoritmo 28. Este enfoque es similar al implementado en GPU, pero realizando el cálculo de múltiples submatrices en paralelo. El Algoritmo 28 detalla este proceso, incluyendo el cálculo de la divergencia de Jensen-Shannon para cada par y la escritura en la posición correspondiente de la matriz (Paso 14).

Algoritmo 25: Acelerador: construcción de sketches en paralelo

```
   Input: bloque de datos chunk
 1 Let
 2     largo k-mer k = 31
 3     número de hebras T = 8
 4     for i ← 0 to T − 1 do
 5         Inicializa line_buffer[i] como un arreglo de bits de tamaño 2k
 6         Inicializa sketch[i] como un sketch de frequencia de tamaño p × m
 7         Inicializa pq[i] como un arreglo de colas de prioridad de tamaño n × 2^R
 8 foreach chunk_word in stream do
 9     for t ← 0 to T − 1 do in parallel
10         c ← chunk_word[64t : 64(t + 1) − 1]
11         queue_builder(k, c, line_buffer[t], sketch[i], pq[i])
```

En ambas implementaciones la salida del acelerador es una lista de similitudes, la cual es leída desde el host y escrita en el medio de almacenamiento.

5.3. Arquitectura en FPGA

La implementación en FPGA para cálculo de similitud usando divergencia de Jensen-Shannon (JSD) consta de dos etapas de procesamiento, cada una con un kernel de procesamiento principal. En la primera etapa, se definieron tres kernels, uno de lectura desde memoria externa, uno de escritura de resultados y uno de procesamiento. El kernel de procesamiento implementa el algoritmo de alto nivel descrito en los Algoritmos 25 y 26. La segunda etapa corresponde al cálculo de la matriz de similitud, la que utiliza tres kernels para lectura, escritura y procesamiento. En este caso se utilizan cuatro instancias de cada kernel para mejorar la distribución del uso de recursos en el acelerador. El Algoritmo 28 describe el procesamiento en alto nivel del kernel.

5.3.1. Kernel de construcción de arreglos de colas

En la etapa de construcción de colas, el host descomprime los genomas, los empaqueta y transfiere a la memoria externa del acelerador, añadiendo una marca de fin de cada archivo. La transferencia y ejecución de los kernels en el acelerador se realizan usando el *framework* Vitis de Xilinx, el cual utiliza OpenCL para la comunicación con el acelerador.

Algoritmo 26: Acelerador: construcción de arreglo de colas (queue builder)

Input: largo k-mer k, sección de 64-bit chunk c, buffer de línea *line_ buffer*, sketch frecuencia *sketch*, arreglo de colas de prioridad *pq*

```
1  b ← two_bit_encode(c)
2  line_buffer ← (line_buffer ≪ 2B) | b
3  if line_buffer ≠ eof_mark then
4  |  k-mer ← line_buffer[2i : 2i + 2k − 1]
5  |  k-mer_canon ← canonical_kmer(k-mer)
6  |  if sample(k-mer_canon) then
7  |  |  f_est ← sketch.insert(k-mer_canon)
8  |  |  pq.insert(f_est)
9  else
10 |  if isFPGA then
11 |  |  entropy ← compute_entropy(pq)
12 |  |  store_in_memory (pq, entropy)
13 |  else
14 |  |  store_in_memory (pq)
```

La implementación en el acelerador se divide en tres kernels comunicados por *AXI-Streams*:

- Kernel HLS de lectura: controla la lectura desde memoria externa y la separación en flujos de 64 bits cada uno.
- Kernel RTL de procesamiento: ocho instancias para procesar los datos de entrada llenando los arreglo de colas de prioridad.
- Kernel HLS de escritura: ocho instancias, una por cada kernel de procesamiento, para escribir el contenido de las colas a memoria externa.

El kernel de procesamiento realiza la construcción de un arreglo de colas de prioridad para ocho bases en paralelo (64 bits), tal como es descrito en el Algoritmo 26. El kernel implementa la construcción de k-mers, el muestreo usando una función hash, la inserción a un sketch Count-Min (seleccionado según los resultados presentados en el Capítulo 6), la inserción al arreglo de colas y el cálculo de entropía. La Figura 5.3 describe el flujo de datos en el sistema y la interacción con el host usando un pivote por kernel y cuatro interfaces a memoria externa. El módulo de construcción de k-mers sigue los lineamientos descritos en la Sección 4.3 para el cálculo de similitud Jaccard, pero usando una función hash MurmurHash3 de 32 bits, descrita en el Algoritmo 29. El módulo de muestreo corresponde a una función hash que marca un k-mer como válido solo si se cumple que los N bits menos significativos son iguales a cero, como se

Algoritmo 27: Matriz de similitud JSD en GPU

Input: número de pivotes V, número de genomas N, conjunto de sketches s, conjunto de pivotes p
Output: matriz de similitud M

Let
  Inicializa P como un arreglo de sketches de tamaño V
  Inicializa M como la matriz de similitud de $V \times N$
`// Carga` V `pivotes y calcula las similitudes entre ellos`
for $x \leftarrow 0$ **to** $V - 1$ **do**
  $P[x] \leftarrow stream(p[x])$
  for $y \leftarrow 0$ **to** x **do in parallel**
    $M[x, y] \leftarrow JSD\ (stream(p[x]),\ P[y])$
`// Itera sobre el stream de entrada`
for $i \leftarrow 0$ **to** $N - 1$ **do**
  $x \leftarrow i + V$
  for $y \leftarrow 0$ **to** $V - 1$ **do in parallel**
    $M[x, y] \leftarrow JSD(stream(s[x]), P[y])$
return M

indica en la Sección 5.2.3, esto reduce los datos en un factor de 2^N aproximadamente. El resto de los módulos diseñados para este kernel se describen a continuación.

5.3.1.1. Módulo Count-Min (CMS)

El módulo que implementa el sketch Count-Min, puede recibir hasta ocho k-mers válidos por ciclo, el realizar más de una inserción en paralelo incrementa los recursos de memoria asociados al sketch, ya que es necesario usar memorias espejo para permitir este acceso múltiple. En este trabajo, en la mayoría de los ciclos no será necesario insertar los ocho k-mers, porque, debido al muestreo, muchos de ellos no serán válidos. Por esta razón, el acceso al módulo fue serializado, añadiendo un buffer de entrada y la lógica asociada para detener el pipeline, hasta que todos los datos en el buffer sean insertados. La Figura 5.3 ilustra esto con la señal *ready*. Para cada dato de entrada al módulo se aplican d funciones hash, una por fila del sketch. El módulo usa los m bits menos significativos para indexar el bloque de memoria o BlockRAM (BRAM) correspondiente a la fila. La escritura se realiza en un pipeline de tres ciclos, en el primero realiza la lectura del contador, en el segundo se incrementa este valor en uno según lo descrito en el Capítulo 3, y en el tercer ciclo se realiza la escritura en la BRAM. Este proceso genera una actualización errónea cuando el mismo dato es insertado en dos ciclos consecutivos,

Algoritmo 28: Matriz de similitud JSD en FPGA

Input: número de pivotes V, número de genomas N, conjunto de sketches s
Output: matriz de similitud M

```
1  Let
2      Inicializa P como un arreglo de sketches de tamaño V
3      Inicializa M como la matriz de similitud de N × N
4  for i ← 0 to N/V − 1 do
       // Carga V pivotes y calcula las similitudes entre ellos
5      for j ← 0 to V − 1 do
6          x ← i × V + j
7          P[j] ← stream(s[x])
8          for k ← 0 to j do in parallel
9              y ← i × V + k
10             M[x, y] ←JSD (stream(s[x]), P[k])
       // Itera sobre el stream de entrada
11     for x ← (i + 1) × V to N do
12         for k ← 0 to V − 1 do in parallel
13             y ← i × V + k
14             M[x, y] ←JSD(stream(s[x]), P[k])
15 return M
```

ya que la lectura de memoria se realiza antes de que la primera actualización se escriba. Esto se subsana adelantando la lectura, lo que se traduce en la implementación como un multiplexor para seleccionar la fuente a incrementar entre la salida de la memoria (ciclo 1) o el registro que contiene el incremento (ciclo 2), eso se omitió en la Figura 5.4 por simplicidad. Una vez que cada contador es incrementado, las estimaciones de cada fila, son comparadas en un árbol de reducción para encontrar el mínimo, que corresponde a la estimación del sketch. Esto último ilustrado en el lado derecho de la Figura 5.4.

5.3.1.2. Módulo arreglo de colas de prioridad (PQA)

El módulo PQA o de arreglo de colas de prioridad, implementa la estructura PQA descrita en la Sección 5.2.4. Este módulo almacena los elementos de mayor frecuencia estimados por el sketch Count-Min. Cada estimación de frecuencia válida es acompañada por un identificador del elemento, que corresponde a la hash utilizada en la primera fila del módulo de Count-Min. El módulo utiliza los R bits menos significativos de la hash para seleccionar la cola de prioridad donde se insertará la estimación. El resto de los bits son almacenados para identificar el elemento

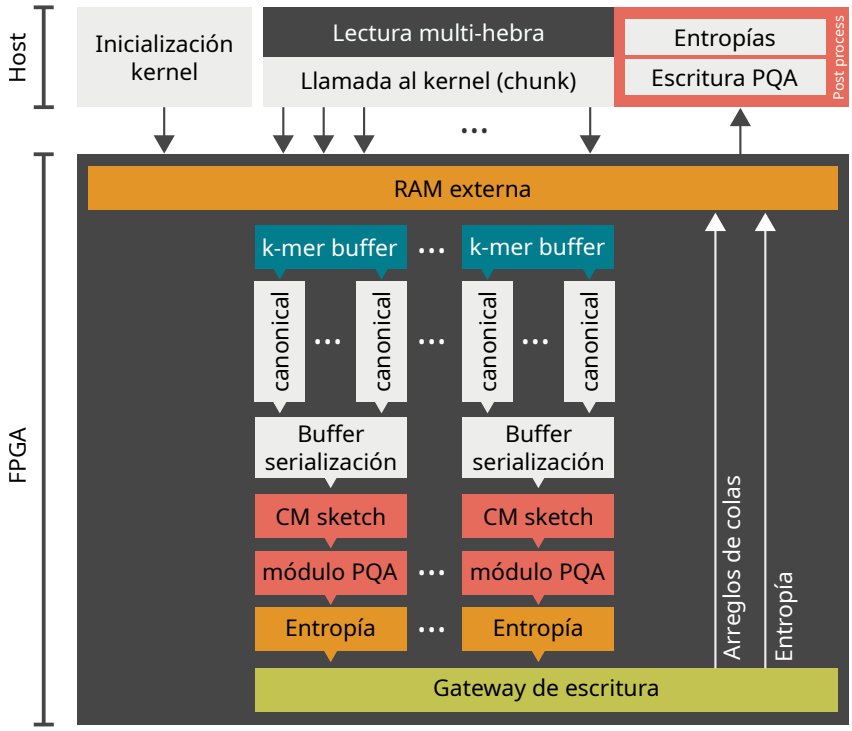


Figura 5.3: Arquitectura general del kernel de construcción de arreglos de colas de prioridad.

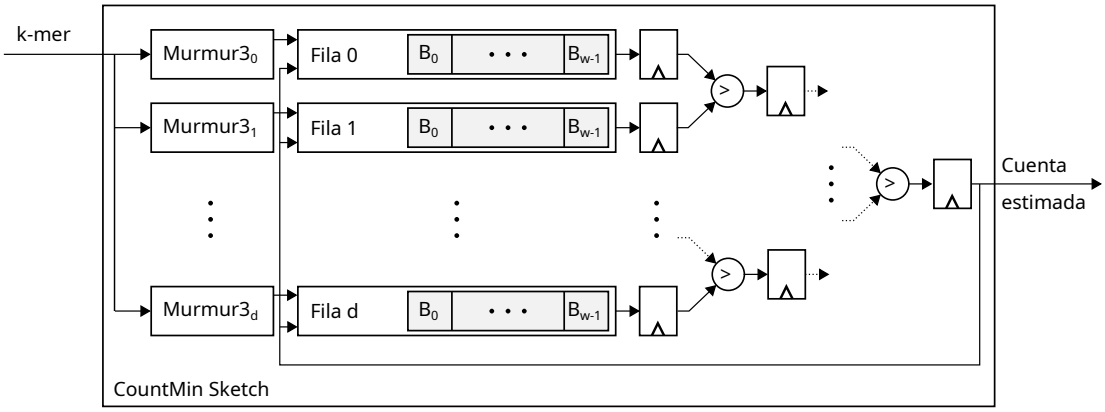


Figura 5.4: Arquitectura módulo Count-Min sketch.

Algoritmo 29: MurMurHash3

Input: Elemento de entrada x
Input: Número de bytes de elemento de entrada len
Input: Semilla $seed$
Output: Hash h

Let
  $c1 = \texttt{0xcc9e2d51}$
  $c2 = \texttt{0x1b873593}$
  $r1 = \texttt{15}$
  $r2 = \texttt{13}$
  $m = \texttt{5}$
  $n = \texttt{0xe6546b64}$
  $h = seed$
for k ***in*** $four\_bytes\_chunk(x)$ **do**
  $k \leftarrow k \times c1$
  $k \leftarrow (k \ll r1)|(k \gg (32 - r1))$
  $k \leftarrow k \times c2$
  $h \leftarrow h \oplus k$
  $h \leftarrow (h \ll r2)|(h \gg (32 - r2))$
  $h \leftarrow h \times m + n$
$h \leftarrow h \oplus len$
$h \leftarrow h \oplus (h \gg 16)$
$h \leftarrow h \times \texttt{0x85ebca6b}$
$h \leftarrow h \oplus (h \gg 13)$
$h \leftarrow h \times \texttt{0xc2b2ae35}$
$h \leftarrow h \oplus (h \gg 16)$
return h

dentro de la cola, permitiendo la actualización de elementos ya existentes. La Figura 5.5 muestra el flujo de datos en el módulo, indicando las memorias y la selección de la cola a utilizar. La actualización de la cola de prioridades requiere de un ordenamiento de los elementos. Para esto, el módulo diseñado lee todos los datos de la cola seleccionada, luego compara el identificador y contador de cada uno de ellos con los de entrada. Un circuito combinacional analiza las comparaciones de igualdad y menor que, activando las señales de escritura de cada una de las memorias, seleccionando como fuente el dato actual, el de la derecha (de mayor frecuencia) o el de entrada. De esta forma, cada inserción es atendida en un ciclo de reloj. La Figura 5.6 muestra la arquitectura del submódulo de inserción, donde *mux ctrl* es el circuito combinacional que realiza la selección de la entrada a escribir en cada uno de los elementos de la cola seleccionada.

Al final de la inserción de k-mers válidos, el arreglo de colas pasa al estado de lectura,

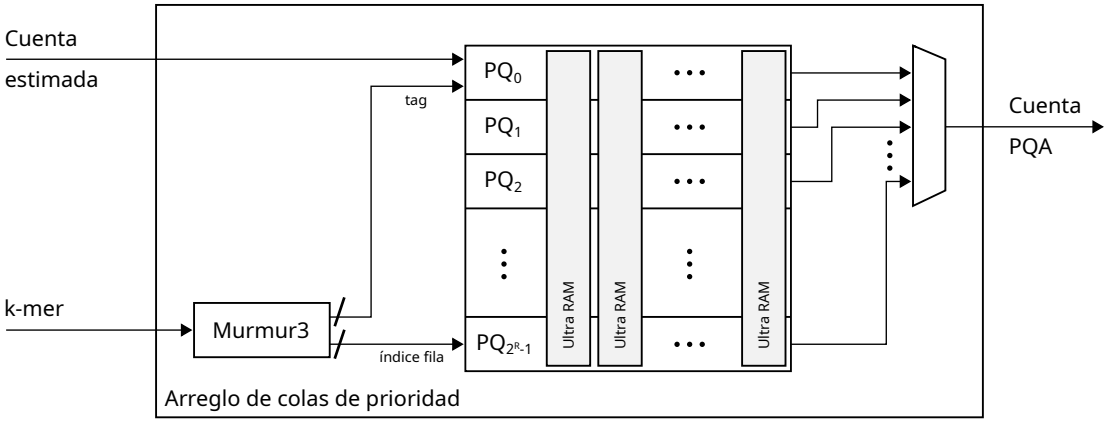


Figura 5.5: Arquitectura módulo arreglo de colas de prioridad.

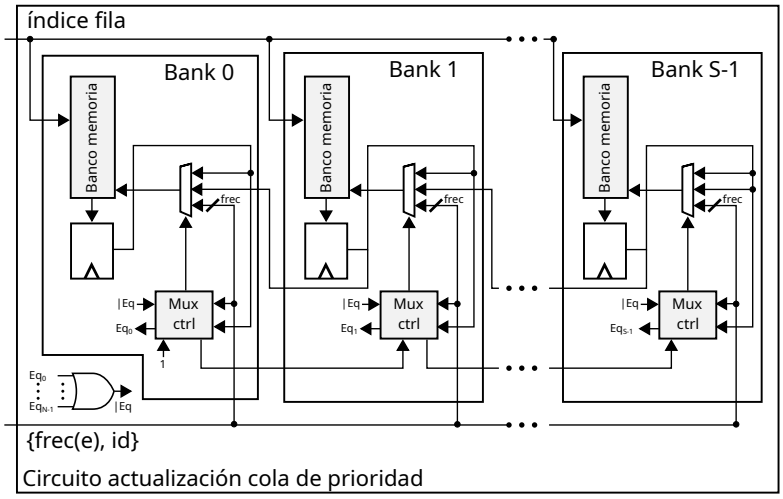


Figura 5.6: Arquitectura módulo arreglo de colas de prioridad.

donde cada una de las filas es empaquetada y transferida al kernel de escritura en memoria. En paralelo, los datos son enviados al módulo de cálculo de entropía, el cual escribe su resultado en la posición de memoria siguiente a la última fila del PQA.

5.3.1.3. Módulo entropía

El cálculo de entropía se realiza durante el proceso de escritura a memoria del contenido del arreglo de colas. En cada ciclo de reloj el módulo de entropía recibe ocho elementos, correspondientes a una fila del arreglo. Cada elemento ingresa a un módulo de cálculo de logaritmo en base dos. El que se implementó siguiendo lo descrito en el Algoritmo 30. Este módulo usa un módulo

Algoritmo 30: Algoritmo cálculo logaritmo base 2

Input: frecuencia e
Output: valor logaritmo v

1 **Let**
2 | Inicializa LUT como un arreglo de N
3 | **for** $i \leftarrow 0$ **to** $N-1$ **do**
4 | | $LUT[i] \leftarrow log_2(i/N)$
5 $v_{int} \leftarrow ldz(c)$
6 $idx \leftarrow int(c \wedge ((1 \ll v_{int}) - 1))$
7 $v_{frac} \leftarrow LUT[idx]$
8 $v_{frac} \leftarrow v_{frac} + (c - idx)\ (LUT[idx+1] - LUT[idx])$
9 **return** $\{v_{int}, v_{frac}\}$

Algoritmo 31: Algoritmo cálculo recíproco multiplicativo con Newton-Raphson

Input: cuenta c, iteraciones it
Output: salida v

1 **Let**
2 | Inicializa LUT como un arreglo de N
3 | **for** $i \leftarrow 1$ **to** $N-1$ **do**
4 | | $LUT[i] \leftarrow 1/i$
5 $v \leftarrow ldz(c)$
6 $x_0 \leftarrow LUT[(c \gg v) + 1]$
7 **for** $i \leftarrow 0$ **to** $it-1$ **do**
8 | $x_i \leftarrow 2 - (b\ x_0)$
9 | $x_0 \leftarrow x_0\ x_i$
10 **return** x_0

ldz para detectar el número de ceros a la izquierda en la entrada, lo que también corresponde a la parte entera del logaritmo (Paso 5 del Algoritmo 30). Con el número de ceros, la entrada se normaliza a un número entre uno y dos, con este valor se obtiene la parte fraccionaria desde la LUT (Paso 7). Luego, con los bits descartados en el paso anterior, el módulo realiza una interpolación lineal para mejorar el resultado de la parte fraccionaria, como se describe en el Paso 8 del Algoritmo 30. El resultado final es la concatenación entre la parte entera y la parte fraccionaria del logaritmo.

Las salidas de los módulos de logaritmo son multiplicadas por la entrada de los mismos y acumuladas usando un árbol de reducción para calcular la Ecuación (5.7). La que se reescribe

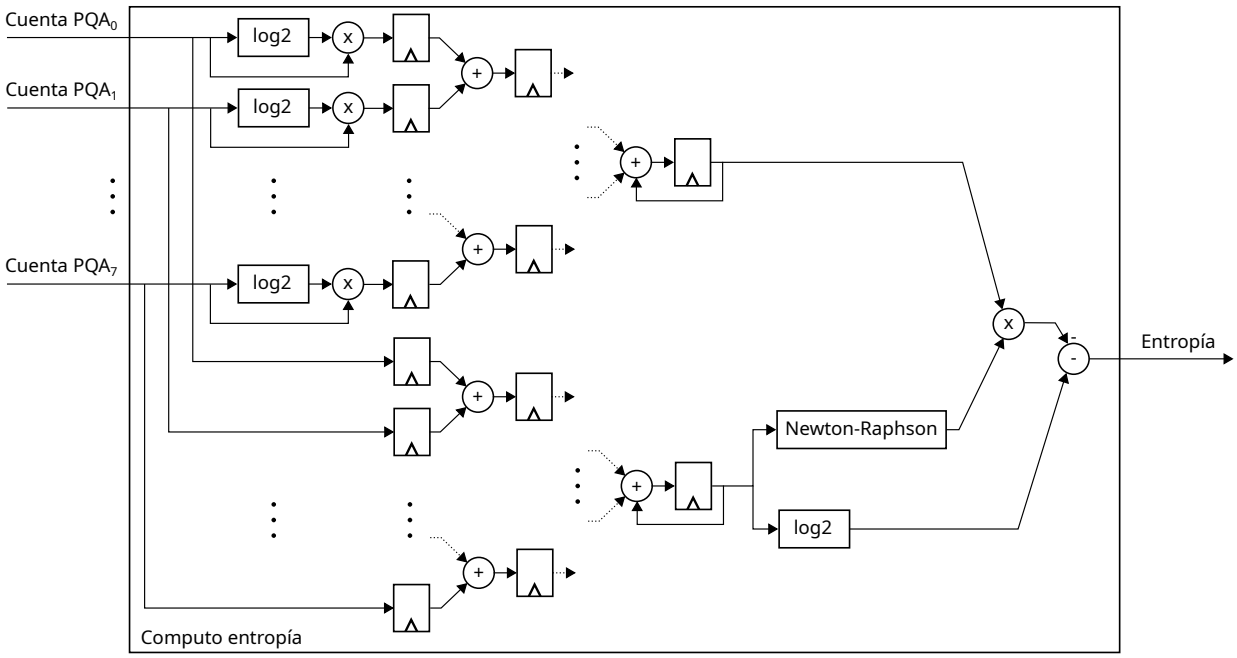


Figura 5.7: Arquitectura módulo entropía.

para reducir el número de operaciones como:

$$\hat{H} = -\frac{1}{M}\sum_{i=1}^{K} m_i \log m_i + \log M. \tag{5.8}$$

La división final se realiza calculando el inverso multiplicativo de M con Newton-Raphson, descrito en el Algoritmo 31. Newton-Raphson fue implementado usando una LUT para obtener la suposición inicial, descrita en los Pasos 5 y 6 del Algoritmo 31. Seguido del proceso iterativo para mejorar la estimación, descrito en los pass 8 y 9 del Algoritmo 31. La Figura 5.7 ilustra el flujo de datos entre los módulos descritos y las operaciones necesarias para el cálculo de la Ecuación (5.8).

5.3.2. Kernel de cálculo de similitud

El kernel de cálculo de similitud realiza la operación de unión entre arreglos de colas de prioridad, para esto el kernel tiene dos estados, los cuales se ejecutan de forma secuencial en cada llamada. El primer estado es la lectura de los pivotes, donde cada PQA es copiado a una memoria interna junto a la entropía correspondiente, la que está en la posición de memoria siguiente al último elemento del arreglo. En paralelo a la lectura desde memoria externa, el kernel

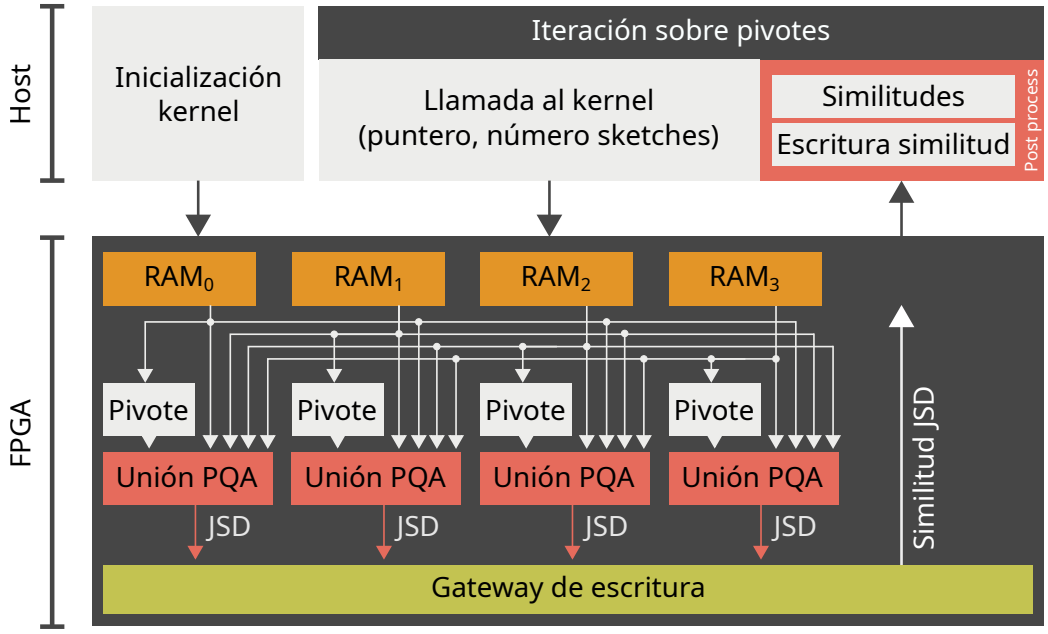


Figura 5.8: Arquitectura kernel cálculo matriz de similitud usando JSD. El diagrama muestra solo un pivote por instancia a modo de ejemplo.

calcula la unión estas colas con los pivotes que residen en memoria interna en ese momento. Este proceso se repite hasta completar la lectura de los V pivotes. En el segundo estado se lee un conjunto de arreglos de colas desde memoria externa, calculando la unión de las colas con los pivotes residentes, la entropía asociada a esta unión y la divergencia de Jensen-Shannon. El módulo envía como salida al kernel de escritura la divergencia y un identificador del pivote y número de cola que generaron ese resultado. La Figura 5.8 muestra la arquitectura del kernel, indicando los módulos de procesamiento y el flujo de datos asociado. El módulo de entropía es el mismo presentado en la Sección 5.3.1.3, pero utilizando 16 entradas en lugar de 8.

Con el objetivo de mejorar el ruteo, cada instancia del kernel está conectada a un kernel de lectura de memoria externa, con cuatro interfaces de lectura. El kernel utiliza una interfaz de lectura a memoria externa para la copia de pivotes. En el segundo estado, el kernel utiliza las cuatro interfaces para calcular la unión entre colas en distintas memorias externas con los pivotes residentes en memoria interna. Esta decisión de diseño reduce los recursos de ruteo internos, permitiendo una mejor distribución de los módulos en el posicionamiento en el acelerador.

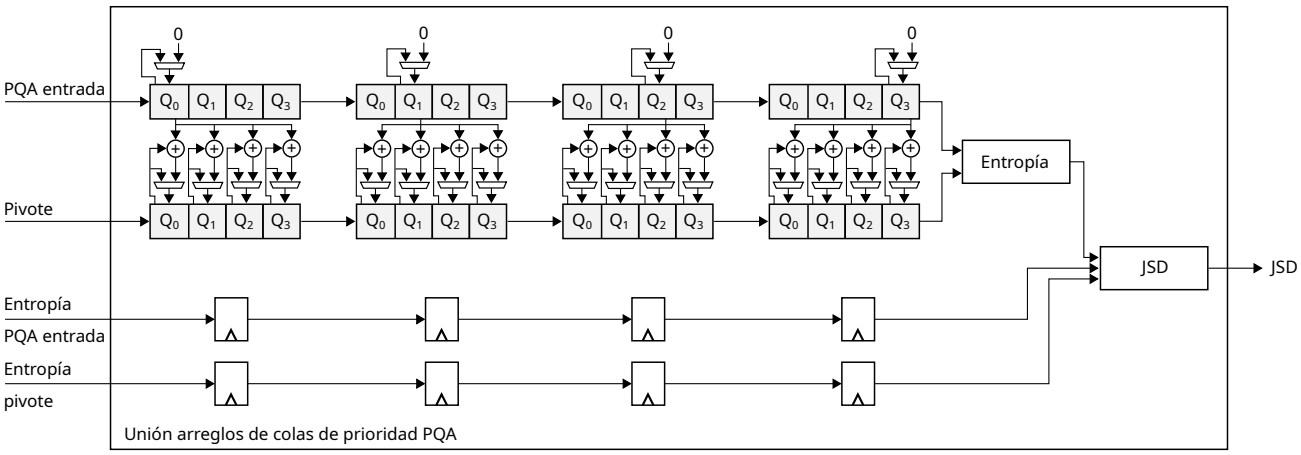


Figura 5.9: Arquitectura módulo unión de arreglos de colas de prioridad. Ejemplo de pipeline de procesamiento con colas de cuatro elementos, en cada ciclo se realiza la comparación de un elemento de la cola de entrada contra todos los elementos del pivote.

5.3.2.1. Módulo unión PQA

El módulo de unión de PQA, realiza la operación de unión entre los contadores de las colas que sean equivalentes en dos PQA. El tamaño de la unión varía entre $2^R \times n$, cuando los PQA tienen una coincidencia perfecta entre sus elementos y $2^R \times 2n$ cuando no hay ninguna coincidencia entre ellas.

En cada ciclo el módulo recibe dos colas, de ocho elementos cada una, realiza una búsqueda de identificadores de una cola en otra, acumulando los contadores en caso de coincidencias. Este proceso se implementó con un pipeline de ocho etapas, una por cada elemento de la cola de prioridad, como se ve en la Figura 5.9 en un ejemplo de cuatro elementos. Después de 16 ciclos, el resultado es un arreglo de entre 8 y 16 elementos, el cual es utilizado como entrada al módulo de entropía. Para cada unión parcial se acumula el término $m_i \log m_i$ para evaluar la Ecuación (5.8) una vez que se reciban todas las colas del PQA. Posteriormente, el módulo de unión utiliza las entropías individuales asociadas a cada PQA y la entropía de la unión para calcular la divergencia de Jensen-Shannon según la Ecuación (5.3).

Los resultados de las implementaciones descritas en esta sección se presentan en la Sección 6.5.

5.4. Implementación en GPU

Al igual que en la implementación en FPGA, la implementación en GPU se divide en dos procesos: la construcción de colas de prioridad y el cálculo de similitud usando JSD. Cada uno de ellos fue implementado como un proceso separado, siguiendo una lógica similar a la utilizada en el cálculo de similitud Jaccard presentada en el Capítulo 4. En la primera etapa, un kernel realiza la extracción de k-mers, otro kernel inserta estos k-mers al sketch Count-Min, generando una estimación asociada a cada k-mer. Posteriormente, un tercer kernel inserta cada estimación en el arreglo de colas de prioridad. En la etapa de cálculo de similitud, un kernel calcula las divergencias de Jensen-Shannon para un subconjunto de los genomas. Esto se debe a las limitaciones de memoria del acelerador y el tamaño de los PQA. Esta etapa realiza el cálculo de la matriz completa procesando submatrices y usando el enfoque de pivotes propuesto en el Capítulo 4.

5.4.1. Etapa de construcción de arreglos de colas

La implementación de la etapa de construcción de arreglos de colas en GPU se realizó con tres kernel CUDA. El primer kernel es para extraer los k-mer desde el conjunto de bases de entrada. Este kernel realiza el mismo procedimiento descrito en el Algoritmo 19 de la Sección 4.4, donde cada hebra lee los k valores siguientes a cada base, codificando cada base en dos bits y calculando el k-mer canónico asociado. Posteriormente, el kernel escribe cada k-mer en memoria externa, además de un *flag* por k-mer que indica la validez del mismo, con el fin de poder identificar aquellos caracteres de los datos de entrada que no corresponden a una codificación de bases de ADN.

El segundo kernel lee cada uno de los k-mer junto al *flag* de validez por k-mer. Si el k-mer es válido, se aplican las d funciones hash y se incrementa el contador asociado en cada línea del sketch de cuentas, usando la operación atómica `atomicAdd`, lo que se describe en los Pasos 2 a 7 del Algoritmo 32. Cada estimación del sketch es almacenada en un bloque de memoria asociado a un conjunto de colas, que serán la fuente del próximo kernel. Esta operación se realiza manteniendo un puntero asociado a cada bloque e insertando las estimaciones de forma atómica usando la operación `atomicCAS`, descrito en el Paso 10 del Algoritmo 32. Esta operación ordena las inserciones en siguiente kernel, permitiendo reducir el tiempo de ejecución asociado a la lectura de datos. El Algoritmo 32 muestra el pseudocódigo de la implementación de este kernel

Algoritmo 32: Pseudo código kernel inserción Count-Min sketch

Input: tamaño chunk n, largo k-mer k, valor k-mer *kmers*, k-mer valido *valid*, sketch CM *cm*, número filas CM d, $\log_2$ número buckets por fila CM w, bloque de estimaciones para arreglo de colas *pq_ block*

Output: sketch CM *cm*, bloque de estimaciones para arreglo de colas *pq_ block*

```
1  for idx = 0 to n − k do
2      if valid[idx] then
3          min ← 0xFFFFFFFF
4          for j = 0 to p − 1 do
5              h_j ← hash_j(kmers[idx])
6              bucket ←< h_w, h_0 >_2
7              cm[j, bucket] ← atomicAdd(cm[j, bucket], 1)
8              if min < cm[j, bucket] then
9                  min ← cm[j, bucket]
           // Inserta estimación a bloque de colas
10         pq_block[h_0].append(value : min, tag : h_0)
11 return cm, pq_ block
```

en GPU.

El tercer kernel realiza la inserción de las estimaciones según lo descrito en el Algoritmo 33. Para esto, cada hebra CUDA lee una cola de prioridad a memoria interna (Pasos 3 y 4 del Algoritmo 33), luego lee cada una de las estimaciones en el bloque de memoria escrito por el kernel anterior, insertando todas las estimaciones a la cola correspondiente, según la descripción indicada en los Pasos 5 a 9 el Algoritmo 23. En cada inserción, el kernel verifica si el identificador del elemento ya está en la cola, en cuyo caso actualiza la estimación, o lo inserta si la estimación es mayor al mínimo ya almacenado. Una vez que el kernel procesa todas las estimaciones, la cola de prioridad es copiada a la posición de memoria correspondiente.

Tras esto, el host copia el arreglo de colas a su memoria, continuando el proceso con otro genoma. Este proceso se realiza en ocho streams CUDA, donde cada uno procesa genomas en paralelo.

5.4.2. Etapa de cálculo de similitud

El cálculo de la matriz de similitud se realiza con un kernel que realiza la operación de unión entre dos conjuntos de arreglos de colas. En cada llamada al kernel el host pasa dos punteros

Algoritmo 33: Pseudocódigo kernel inserción arreglo de colas de prioridad

Input: arreglo de colas de prioridad pq, número elementos por cola n, bloque de estimaciones para arreglo de colas pq_block, número de estimaciones en bloque s, índice bloque b_idx
Output: arreglo de colas de prioridad pq

```
1  Let
2      Inicializa LPQ como una cola de prioridad de n elementos
   // Copia cola de prioridad a memoria interna
3  for i = 0 to n do
4      LPQ[i] ← pq[b_idx, i]
   // Inserción de estimaciones
5  for idx = 0 to s do
6      if pq_block[idx, i].value > pq_min then
7          for i = 0 to n do
8              LPQ.insert(pq_block[idx, i])
9              pq_min ← mín(pq_block[idx, i])
10 return pq
```

como argumentos, los que indican los conjuntos de pivotes y de datos a utilizar. Cada bloque CUDA copia a memoria interna uno de los pivotes, y luego procesa la totalidad de los datos del otro conjunto, calculando las uniones entre cada pivote y los datos de entrada. Como indica el Algoritmo 34, este proceso se realiza comparando cada elemento del pivote con elementos en la cola de entrada, combinando los contadores en caso de coincidencia (Pasos 19 a 21 del Algoritmo 34) o considerándolos de forma individual (Pasos 22 a 24 del Algoritmo 34). Durante este proceso se acumulan los componentes $\sum m_i \log m_i$ y $\sum m_i$, utilizados para el cálculo de las entropías individuales y conjuntas, descritas en los Pasos 30 a 32 que evaluan la Ecuación (5.8). Finalmente, el Paso 33 del Algoritmo 34 realiza el cálculo de la divergencia de Jensen-Shannon según la Ecuación (5.3).

Este kernel fue diseñado para recibir punteros diferentes para pivotes y datos de entrada, permitiendo su uso para el cálculo de submatrices, que luego son concatenadas para generar la matriz completa de similitud.

Los resultados de las implementaciones descritas en esta sección se presentan en la Sección 6.5.

Algoritmo 34: Pseudocódigo kernel cálculo de similitud

Input: número de arreglos de colas de entrada N, conjunto de sketches s, conjunto de pivotes p, número elementos por cola n, número de colas 2^R, matriz de similitud M, índice bloque b_idx

Output: matriz de similitud M

Let
  Inicializa P como un arreglo de n elementos
`// Copia de cola de prioridad a memoria interna`
for $i = 0$ **to** 2^R **do**
  for $j = 0$ **to** n **do**
    $P[i, j] \leftarrow p[b\_idx][i, j]$
for $h = 0$ **to** N **do**
  $milogmi_m \leftarrow 0$
  $milogmi_p \leftarrow 0$
  $milogmi_l \leftarrow 0$
  $mi_m \leftarrow 0$
  $mi_p \leftarrow 0$
  $mi_l \leftarrow 0$
  for $i = 0$ **to** 2^R **do**
    for $j = 0$ **to** n **do**
      $overwrite \leftarrow$ `False`
      for $k = 0$ **to** n **do**
        if $P[i, j].tag == s[h][i, k].tag$ **then**
          $overwrite \leftarrow$ `True`
      if $overwrite$ **then**
        $count_m \leftarrow P[i, j].value + s[h][i, k].value$
        $milogmi_m \leftarrow milogmi_m + count_m \log_2(count_m)$
      else
        $milogmi_m \leftarrow milogmi_m + P[i, j].value \log_2(P[i, j].value)$
        $milogmi_m \leftarrow milogmi_m + s[h][i, k].value \log_2(s[h][i, k].value)$
      $milogmi_p \leftarrow milogmi_p + P[i, j].value \log_2(P[i, j].value)$
      $milogmi_l \leftarrow milogmi_l + s[h][i, k].value \log_2(s[h][i, k].value)$
      $mi_m \leftarrow mi_m + P[i, j].value + s[h][i, k].value$
      $mi_p \leftarrow mi_p + P[i, j].value$
      $mi_l \leftarrow mi_l + s[h][i, k].value$
  $H_m \leftarrow \log_2(mi_m) - \frac{milogmi_m}{mi_m}$
  $H_p \leftarrow \log_2(mi_p) - \frac{milogmi_p}{mi_p}$
  $H_l \leftarrow \log_2(mi_l) - \frac{milogmi_l}{mi_l}$
  $M[b\_idx, h] \leftarrow H_m - \frac{H_p + H_l}{2}$
return M

Capítulo 6. Resultados

6.1. Introducción

Este capítulo muestra la base de datos utilizada, la infraestructura de pruebas, las métricas para evaluar el desempeño de las implementaciones y los resultados logrados en cada uno de los trabajos presentados en este informe. La implementación en FPGA de ambos trabajos fue diseñada utilizando la plataforma Vitis de Xilinx, que habilita el desarrollo de aplicaciones heterogéneas con componentes en software y hardware, utilizando un FPGA Virtex UltraScale+ de Xilinx. Las implementaciones en GPU fueron desarrolladas usando el *Software Development Kit* (SDK) de CUDA. Todas las pruebas se realizaron utilizando instancias EC2 de *Amazon Web Services* (AWS).

6.2. Amazon Web Services

Amazon Web Services es un servicio de cómputo en la nube que ofrece diferentes configuraciones para múltiples aplicaciones, desde almacenamiento de grandes volúmenes de datos, a instancias con aceleradores hardware para algoritmos de *High Performance Computing* (HPC). Dentro del servicio de cómputo elástico en la nube (Elastic Compute Cloud, EC2), AWS usa un sistema de instancias de máquinas con diferentes configuraciones. Esto posibilita realizar pruebas del mismo software en diferentes configuraciones de hardware. Las características y precios de las instancias utilizadas en este trabajo se muestran en la Tabla 6.1. En todos los casos se seleccionaron instancias con un precio similar, para poder realizar una comparación del costo de cómputo en cada implementación. Las implementaciones aceleradas en FPGA se realizaron en instancias F1 de AWS EC2 (f1.2xlarge). Estas instancias equipan un FPGA XCVU9P Ultrascale+ de Xilinx con 64 GiB de memoria externa. El host tiene un procesador Intel Xeon con 8 hebras físicas. Para las implementaciones en GPU se utilizó una instancia G5 (g5.4xlarge) equipada con una GPU NVIDIA Ampere A10G y un procesador AMD EPIC de 16 hebras físicas en el host. Para la comparación en software se utilizó una instancia C5 (c5.9xlarge), las que tienen procesadores Intel Xeon Platinum de 36 hebras físicas. También, a modo de comparación, se usó una instancia c5.2xlarge que tiene un precio menor y la misma cantidad de hebras que una instancia f1.2xlarge. Todas las instancias usan un almacenamiento GP3 [94] con un límite de 20 000 operaciones de entrada/salida por segundo. Además, con el fin de hacer una

Tabla 6.1: Instancias de AWS EC2 utilizadas en este trabajo. Todas las instancias utilizan procesadores virtualizados y el número de vCPUs indicado se refiere a la cantidad de hebras en paralelo que puede utilizar una máquina. La instancia f1.2xlarge usa un FPGA Xilinx XCVU9P UltraScale+. La instancia g5.4xlarge usa una GPU A10G de NVIDIA. Los precios y características son válidos en junio del 2022.

Instancia	Familia CPU	vCPUs	RAM (GiB)	Acelerador hardware	Precio (USD/hour)
f1.2xlarge	Intel Xeon	8	122	XCVU9P	1,65
g5.4xlarge	AMD EPYC	16	24	A10G	1,62
c5.9xlarge	Intel Xeon Platinum	36	72	–	1,53
c5.2xlarge	Intel Xeon Platinum	8	16	–	0,34

comparación válida, el ancho de banda se limitó a 212 MiB/s, el cual es el límite soportado por las instancias f1.2xlarge [95]. Esto último se analizó con más detalle en la Sección 6.4.3.

6.3. Base de datos RefSeq de NCBI

El Centro Nacional para la Información Biotecnológica (National Center for Biotechnology Information, NCBI) posee una base de datos de genomas secuenciados llamada Reference Sequence o RefSeq. Esta base de datos es pública y está disponible para descargar desde su sitio web [96]. Ésta está compuesta por 128 110 genomas al primero de junio de 2022, donde el más pequeño consta de 220 bases de ADN y el más grande tiene 40 054 341 269 bases. La media y mediana son 12 925 678 y 4 158 070 respectivamente.

6.4. Similitud usando Jaccard

El desempeño de las etapas de construcción de sketches y computo de la matriz de similitud, fue evaluado usando cuatro algoritmos e implementaciones diferentes:

Dashing [7]: Una implementación del estado del arte para cálculo de similitud entre genomas. Dashing utiliza una implementación multihebras e instrucciones SIMD para alcanzar un alto desempeño en procesadores multi núcleo actuales. En este trabajo se utilizó la versión 1.0 de Dashing, disponible en `https://github.com/dnbaker/dashing`.

SF-GPU: Es una implementación directa de la etapa de construcción de sketches y del cálculo de similitud utilizando la plataforma CUDA de NVIDIA. La etapa de construcción utiliza el Algoritmo 11 para leer los genomas en el host, el cual se implementa con dos kernel CUDA, uno para extraer y codificar k-mers, y otro para calcular las hashes e insertar los elementos al sketch HLL. Un tercer kernel calcula la suma armónica y el número de ceros desde la memoria del sketch, almacenándolos en memoria global para que el host pueda leerlos y calcular la cardinalidad de cada uno de ellos. En esta implementación cada hebra del host se comunica con un stream CUDA diferente. En la etapa de cálculo de similitud, cada línea de la matriz es ejecutada por una hebra de CUDA, leyendo los sketches desde la memoria global, calculando las uniones de ellos y los coeficientes de Jaccard para cada par de la matriz.

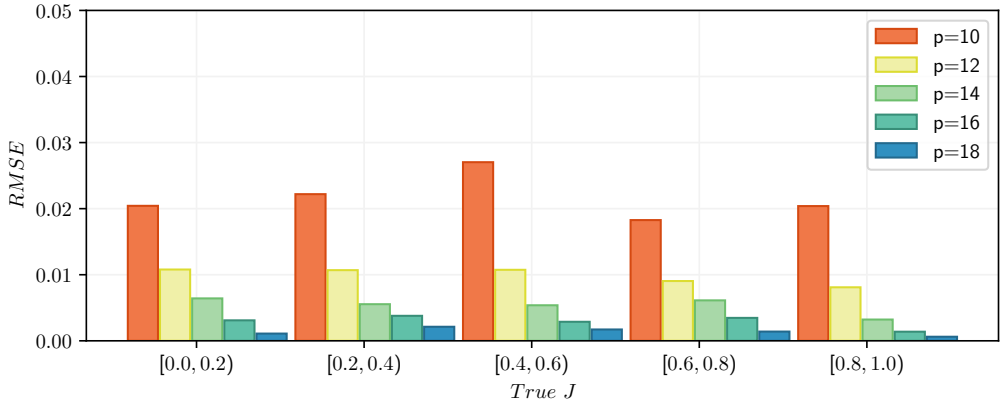
JACC-GPU: Es una implementación optimizada de SF-GPU, la que utiliza el algoritmo de cómputo de la matriz de similitud con pivotes descrito en el Algoritmo 16. Esta implementación disminuye la cantidad de transferencias a la memoria global, reduciendo el tiempo de cómputo al compararlo con SF-GPU. En esta implementación cada bloque de CUDA lee un pivote a la memoria interna y, posteriormente, todas las hebras del bloque respectivo calculan en paralelo las uniones entre el sketch y el resto de la línea de la matriz. Esta implementación se describe en más detalle en la Sección 4.4.

JACC-FPGA: Es la implementación principal de este trabajo. Esta implementa los Algoritmos 11-16 en FPGA, siguiendo lo descrito en la Sección 4.3.

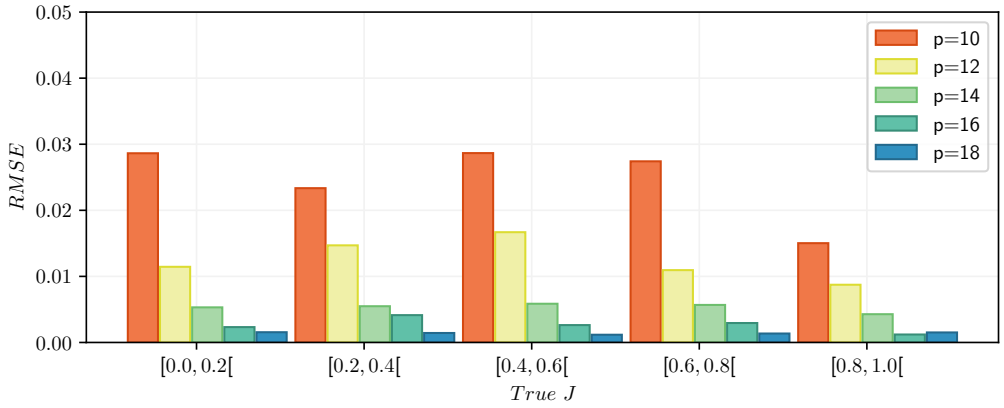
6.4.1. Parámetros HLL

El primer experimento evalúa el impacto del tamaño del sketch en la estimación de los coeficientes de Jaccard, comparado con el valor real. El tamaño del sketch depende del número de buckets, que es 2^p, y del número de bits por bucket. Esta evaluación compara los resultados obtenidos con Dashing y con la implementación presentada en este trabajo. Es importante notar que SF-GPU, JACC-GPU y JACC-FPGA son numéricamente equivalentes, por lo que son evaluadas en conjunto en esta subsección. El algoritmo de HLL implementado usa la corrección de estimación por baja ocupación vista en el Capítulo 3. Además, los resultados fueron comparados con la misma implementación usando el estimador de HLL++, el cual es más complejo, pero entrega mejores resultados que HLL [84].

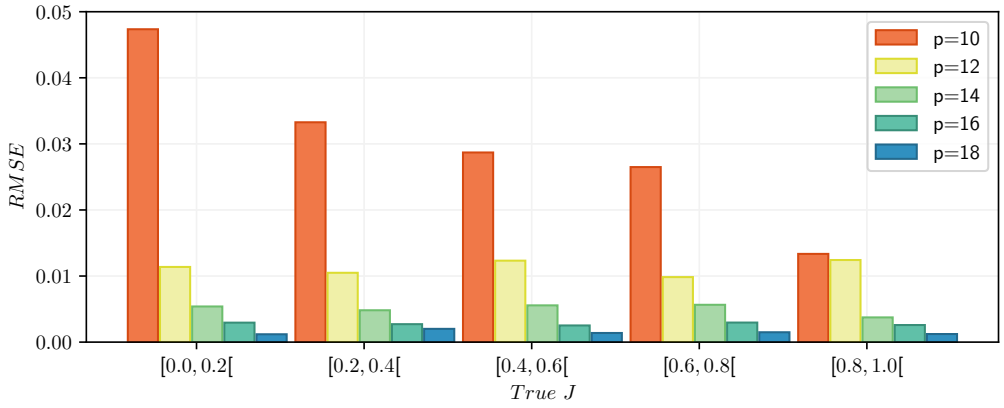
La Figura 6.1 muestra el error cuadrático medio (Root-Mean-Square Error, RMSE) de la



(a) Dashing



(b) HLL con buckets de 5 bits



(c) HLL con buckets de 4 bits

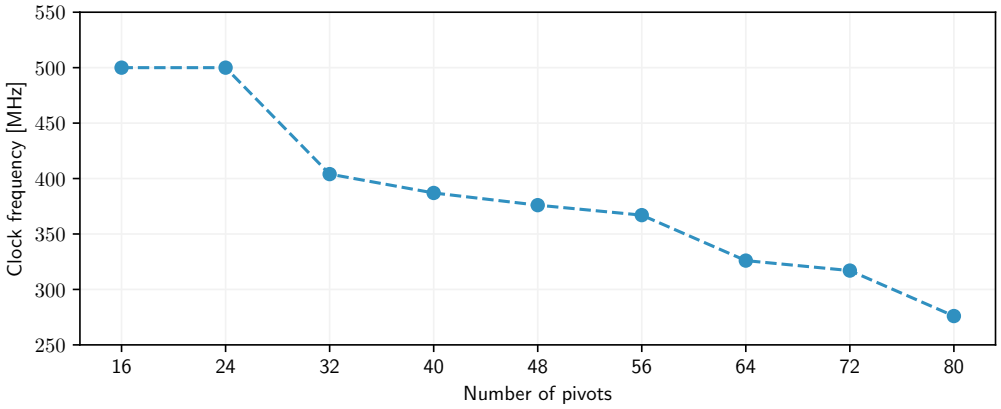
Figura 6.1: Error cuadrático medio (Root-mean-square error, RMSE) de la similitud Jaccard usando cardinalidades estimadas con sketches HLL y usando la cardinalidad exacta como referencia. Las pruebas presentadas incluyen 5 valores de p entre 10 y 18 para los sketches. El error fue calculado usando los mismos 400 pares de genomas utilizados por Dashing [7].

estimación de similitud Jaccard versus la real, para Dashing y para la implementación propuesta. Esta comparación se realizó con 5 valores de p entre 10 y 18, considerando los 400 pares de genomas utilizados por los autores de Dashing para sus propias evaluaciones [7]. Este conjunto de genomas abarca todo el rango de posibles similitudes, entre 0 y 1. Los gráficos en la Figura 6.1 muestran el RMSE para los diferentes valores de p, con 5 rangos de valores Jaccard. La Figura 6.1a muestra el error obtenido usando Dashing, el cual, al ser una implementación en software, utiliza 8 bits por bucket. En una implementación en hardware, como la propuesta en este trabajo, la cantidad de bits por bucket es configurable, donde uno de los objetivos de la implementación es utilizar el mínimo tamaño posible, para reducir el uso de memoria y posibilitar el acceso a la mayor cantidad de buckets en paralelo, como se discutió en el Capítulo 4. Las Figuras 6.1b y 6.1c muestran los errores de la implementación propuesta para buckets de 4 y 5 bits. Como es esperable, el error de estimación disminuye al aumentar el valor de p. Para valores de $p \leq 12$ el uso de menos bits por bucket incrementa el error, en $p > 12$ los resultados en 4, 5 bits y Dashing son equivalentes, siendo menor a 0.01 en todo el rango de similitudes Jaccard. Por esto seleccionamos sketches de 2^{14} buckets con 4 bits por bucket. Tanto el diseño en GPU como en FPGA están parametrizados, por lo que es sencillo modificar el tamaño de los sketches.

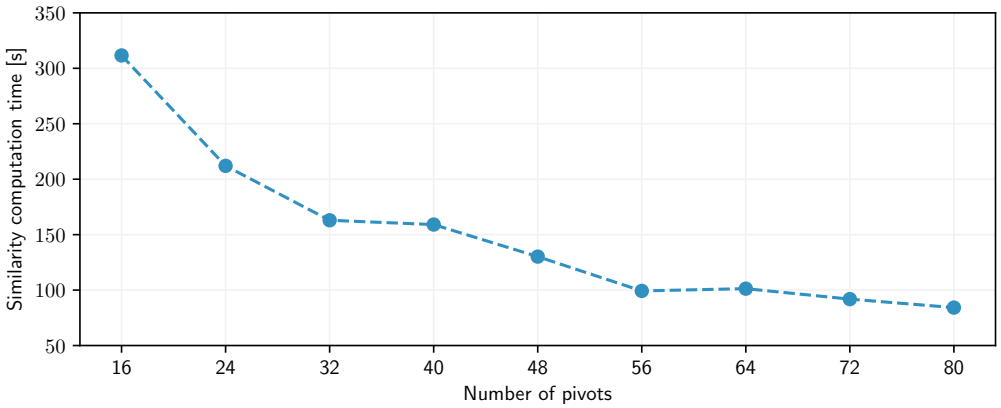
6.4.2. Número de pivotes

El número de pivotes usados por el kernel indica el paralelismo alcanzado por el kernel durante la etapa de cálculo de la matriz de similitud usando el algoritmo propuesto en este trabajo, lo que afecta directamente el tiempo de ejecución. Cuando se calculan V filas de la matriz de similitud, cada llamada al kernel calcula $V \times L + V \times (V-1)/2$ similitudes, donde V es el número de pivotes y L el número de sketches restantes en las líneas de la matriz. Como la implementación del algoritmo en FPGA realiza V uniones en paralelo, el tiempo de ejecución de una llamada al kernel es proporcional a $V + L$. Además, en el cálculo global, un valor mayor de V reduce el número de llamadas necesarias para completar el cálculo de la matriz de similitud.

Teóricamente, el valor máximo de pivotes en la implementación en FPGA es de 128, que corresponde al número de ciclos necesarios para leer un sketch completo desde memoria de $2^{14} \times 4$ bits, usando un bus de 512 bits. Sin embargo, el incremento de pivotes conlleva un incremento en la memoria interna y lógica necesaria, lo que reduce la frecuencia del reloj máxima a la que puede operar el acelerador. Además, como el incremento de pivotes aumenta el número de uniones realizadas en paralelo, se incrementa el uso de recursos lógicos y de memoria, por lo



(a) Frecuencia de reloj del kernel de cálculo de similitud.



(b) Tiempo de ejecución del kernel de cálculo de similitud.

Figura 6.2: Desempeño del kernel de cálculo de similitud versus el número de pivotes usando la base de datos RefSeq completa. Para (a) Frecuencia de reloj en MHz, y (b) tiempo de ejecución del kernel en segundos.

que el ruteo entre ellos se vuelve más complejo. Con un número mayor a 80 pivotes, el software de síntesis Vitis, es incapaz de rutear el diseño para un FPGA XCVU9P, el cual está presente en las instancias f1.2xlarge de AWS.

La Figura 6.2 muestra el desempeño de la implementación en FPGA en la etapa de cálculo de similitud para un número de pivotes entre 16 y 80. La figura muestra la frecuencia de reloj y el tiempo de ejecución en una instancia f1.2xlarge. Idealmente, al duplicar el número de pivotes, el tiempo de ejecución debería reducirse a la mitad. Sin embargo, como se explicó anteriormente, el incremento en el número de pivotes, aumenta la complejidad del diseño, lo que se traduce en una reducción de la frecuencia máxima de reloj que puede alcanzar el acelerador. La Figura 6.2a muestra una reducción consistente en la frecuencia de reloj, comenzando en 500 MHz para 16 y

Tabla 6.2: Tiempo de ejecución de las etapas de construcción de sketches y computo de similitud con la base de datos RefSeq completa. Usando diferentes algoritmos e instancias de AWS. El reporte de desempeño para JACC-FPGA fue en una instancia f1.2xlarge y para SF-GPU/JACC-GPU en una instancia g5.4xlarge. El reporte de desempeño de Dashing se realizó en una instancia c5.2xlarge y c5.9xlarge, ambas optimizadas para cómputo intensivo, además, a modo de comparación en las instancias f1.2xlarge y g5.4xlarge. El tiempo de ejecución reportado es en segundos. En la etapa de cálculo de similitud, el tiempo incluye el cálculo de la matriz de similitud completa.

Implementación	Instancia	Construcción de sketches (s)	Cálculo de similitud (s)
JACC-FPGA	f1.2xlarge	2388	85
JACC-GPU	g5.4xlarge	2257	340
SF-GPU	g5.4xlarge		2315
Dashing	c5.9xlarge	2257	4954
Dashing	f1.2xlarge	6847	23 571
Dashing	g5.4xlarge	2452	8325
Dashing	c5.2xlarge	4756	15 517

24 pivotes y llegando a 276 MHz para 80 pivotes. Aun así, la Figura 6.2b muestra que impacto en el tiempo de ejecución, dado por el incremento en el paralelismo y por el aumento del número de pivotes, es mayor al de la reducción en la frecuencia de reloj. La reducción en el tiempo de ejecución no es lineal al incremento en el número de pivotes, pero el menor tiempo se alcanza al usar 80 pivotes. Como no es posible probar el diseño con un número de pivotes mayor a 80, la implementación reportada corresponde a la que utiliza 80 pivotes.

6.4.3. Desempeño de implementaciones

En esta sección se presenta el desempeño de la solución propuesta en GPU y FPGA, comparada con la implementación en software.

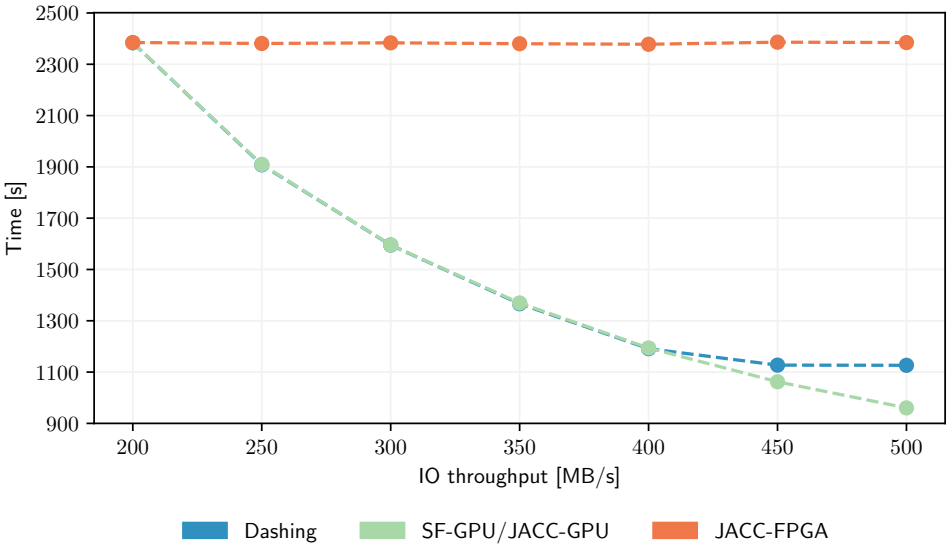
El primer experimento corresponde a la medición del tiempo transcurrido en el host (*wall clock*) para correr ambas etapas del algoritmo de forma separada. Este experimento incluye la construcción de los sketches para la base de datos RefSeq completa, y el cálculo de la totalidad de la matriz de similitud. Al comienzo de la etapa de construcción de sketches, el host ordena la lista de genomas por peso, para mejorar el balance de carga entre las hebras productoras, descritas

en el Algoritmo 11. Dashing también fue ejecutado en las instancias f1.2xlarge y g5.4xlarge, para entregar una mejor comparación entre la relación de los host y los aceleradores conectados a ellos.

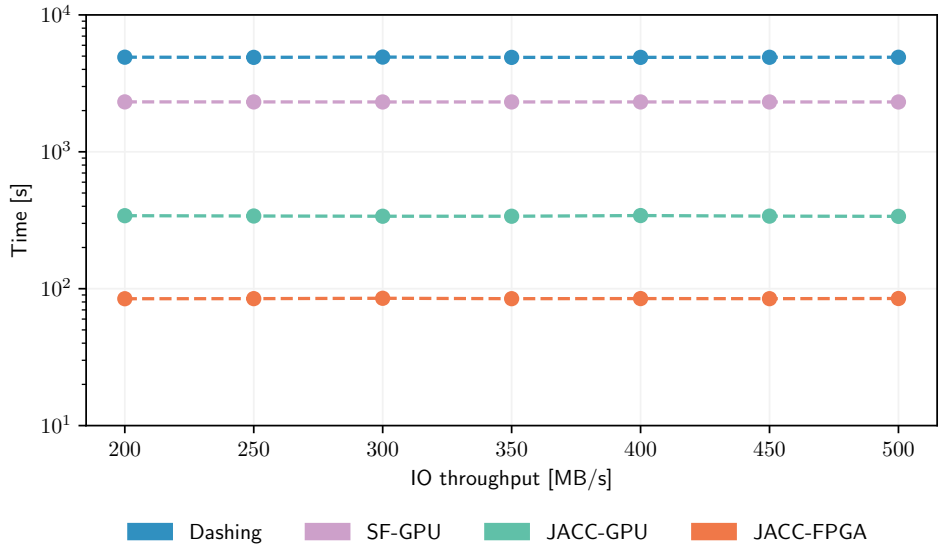
La Tabla 6.2 reporta la mediana del tiempo de 10 ejecuciones de cada etapa. En todos los casos la desviación absoluta de la mediana (median absolute deviation, MAD) es menor al $0,2\,\%$ del valor de la mediana en la etapa de construcción y menor al $0,1\,\%$ en la etapa de cálculo de similitud. En la primera etapa, JACC-FPGA, corriendo en una instancia f1.2xlarge, construye los sketches HLL para todos los genomas del conjunto y calcula sus cardinalidades en aproximadamente 39,8 minutos, lo que es 2,9 veces más rápido que Dashing ejecutándose en el mismo host. También, es el doble de rápido que Dashing sobre una instancia c5.2xlarge, la que tiene el mismo número de hebras que la f1.2xlarge, pero en una serie optimizada para cómputo. Al ejecutar Dashing en una instancia c5.9xlarge con 36 hebras físicas, la etapa de construcción tarda 37,6 minutos, lo que representa una reducción del $10\,\%$ del tiempo frente a JACC-FPGA. De la misma forma, ST-GPU y JACC-GPU en una instancia g5.4xlarge, completan la etapa de construcción en 37,6 minutos. En la misma instancia (g5.4xlarge) Dashing tarda un $10\,\%$ más que las implementaciones en GPU.

En la segunda etapa, tanto Dashing como las implementaciones en GPU y FPGA leen los sketches y las cardinalidades calculados en la primera etapa desde el dispositivo de almacenamiento, para posteriormente calcular las uniones entre ellos y luego las similitudes Jaccard. Como se muestra en la Tabla 6.2, JACC-FPGA calcula la matriz de similitud en solo 85 segundos. En el mismo proceso, Dashing tarda más de 6,5 horas en el host de la instancia f1.2xlarge, más de 4,3 horas en la instancia c5.2xlarge y casi 1,4 horas en la instancia c5.9xlarge. Esto corresponde a una aceleración de 58 veces sobre Dashing en la instancia más rápida considerada, y más de 277 veces sobre Dashing en el mismo host. La implementación SF-GPU tarda 38,6 minutos en una instancia g5.4xlarge, mientras que JACC-GPU, que utiliza el mismo algoritmo de JACC-FPGA, calcula la matriz de similitud en 5,7 minutos en la misma instancia. Así, JACC-FPGA es cuatro veces más rápido de JACC-GPU y 27 veces más que ST-GPU. La alta aceleración alcanzada por JACC-FPGA está dada por el alto nivel de paralelismo temporal y espacial alcanzado por la arquitectura, el cual permite calcular 320 uniones de sketch en paralelo (80 pivotes con 4 sketches de entrada en paralelo).

Una observación importante, que explica el tiempo transcurrido en la etapa de construcción de sketches en las implementaciones más rápidas (JACC-FPGA, JACC-GPU y Dashing en la instancia c5.9xlarge), es el tamaño total de la base de datos, que asciende a 464,4 GiB. A una velocidad de lectura máxima de 212 MiB/s, el tiempo mínimo de lectura es de 2243 segundos, el



(a) Construcción de sketches.



(b) Computo de similitud.

Figura 6.3: Desempeño de las etapas de construcción de sketches y cálculo de similitud versus la velocidad de lectura/escritura del dispositivo de almacenamiento conectado al host. Durante la construcción de sketches, el tiempo de ejecución de JACC-GPU y Dashing en la instancia c5.9xlarge disminuyen al incrementar la velocidad de transferencia. El tiempo de ejecución de JACC-FPGA no disminuye porque está limitado a 212 MiB/S para las instancias f1.2xlarge. En la etapa de cálculo de similitud, el tiempo de ejecución de las tres implementaciones está limitado por la capacidad de cómputo, por lo que es independiente de la velocidad de acceso al dispositivo de almacenamiento.

cual es cercano a los tiempos reportados en la Tabla 6.2. Esto permite inferir que el desempeño de estas implementaciones está limitado por el tiempo de entrada y salida, y que el procesamiento está casi completamente solapado con la lectura de datos. El tiempo de ejecución ligeramente superior de JACC-FPGA se explica por la limitación en la velocidad de lectura de 212 MiB/s de las instancias f1.2xlarge, mientras que en las instancias g5.4xlarge y c5.9xlarge es de 594 MiB/s y 1187 MiB/s respectivamente, lo que las permite operar a la máxima velocidad de lectura configurada.

Para validar la hipótesis anterior, en este trabajo se realizó un experimento donde gradualmente, se incrementó la velocidad de lectura del dispositivo de almacenamiento desde 200 MiB/s hasta 500 MiB/s, lo que también aumento el costo de almacenamiento [95]. La Figura 6.3a muestra el tiempo de ejecución para la etapa de construcción de sketches en las tres implementaciones más rápidas en función de la velocidad de lectura. Como era esperado, el tiempo de ejecución de JACC-FPGA permanece constate, debido a la limitación en el ancho de banda de la instancia. Las otras dos implementaciones (JACC-GPU y Dashing) reducen su tiempo de ejecución junto al incremento de la velocidad de lectura. Sobre los 400 MiB/s, Dashing comienza a ser limitado por la capacidad de cómputo de la instancia c5.9xlarge, alcanzando un tiempo de ejecución mínimo de 1100 segundos. Por otro lado, JACC-GPU continúa disminuyendo el tiempo de ejecución con el incremento de la velocidad de lectura durante todo el experimento. La Figura 6.3b muestra un gráfico semilogarítmico del tiempo de ejecución de la etapa de cálculo de la matriz de similitud versus la velocidad de lectura. En este caso, el tiempo de ejecución permanece constante en todas las implementaciones, ya que el tamaño de los archivos a leer es menor a 2 GiB, por lo que todas las implementaciones están limitadas por capacidad de procesamiento.

La Figura 6.4 muestra un perfil del tiempo de ejecución de JACC-FPGA en la instancia f1.2xlarge. En la etapa de construcción de sketches, el tiempo fue medido en la hebra consumidora. Se utilizaron llamadas asincrónicas al kernel, permitiendo que casi el 100 % del tiempo de procesamiento en el FPGA esté solapado con el tiempo de procesamiento en el host. Como muestra la Figura 6.4, solo el 0,2 % del tiempo es utilizado en llamadas al kernel, que incluyen copias de memoria y espera de resultados. A pesar de que la implementación solapa la lectura de los archivos con el empaquetamiento, la hebra consumidora pasa el 32 % del tiempo esperando por nuevos datos de las hebras productoras, lo que confirma que el tiempo de esta etapa está limitado por la velocidad de lectura del dispositivo de almacenamiento. En la etapa de cálculo de la matriz de similitud, el tiempo fue medido en el hilo principal. En este caso, el host pasa el 10,8 % del tiempo leyendo los sketches desde el dispositivo de almacenamiento y 26,5 % almacenando la matriz de similitud. Del 62,7 % restante, un 14,3 % se utiliza en llamadas al

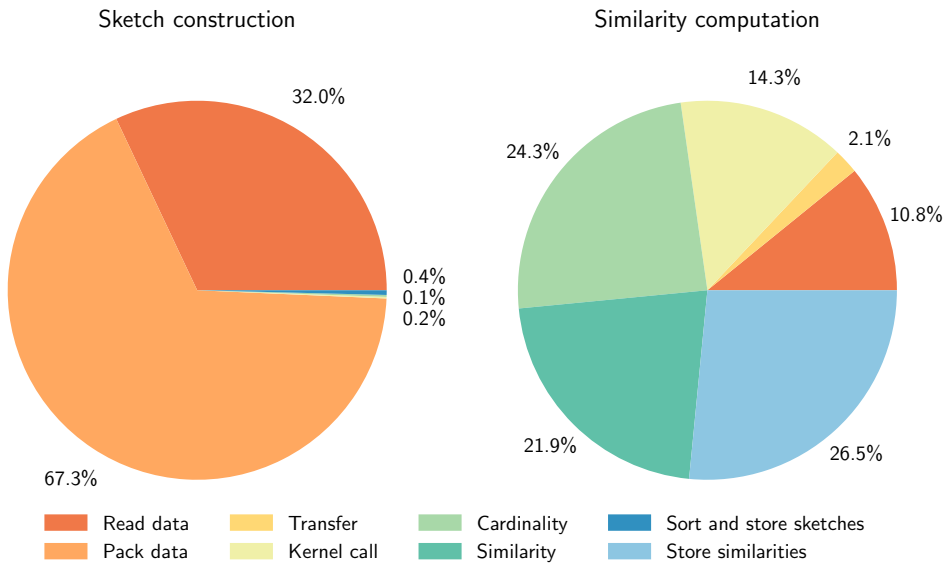


Figura 6.4: Perfil del tiempo de ejecución en el host para las dos etapas de procesamiento con JACC-FPGA. En la etapa de construcción el tiempo fue medido en la hebra consumidora y en la de cálculo de similitud en la hebra principal. Las llamadas al kernel incluyen la copia de los datos hacia y desde los buffers del acelerador y el tiempo de espera al finalizar la ejecución de una llamada al kernel. Las llamadas al kernel son asíncronas, por lo que el tiempo de ejecución en el acelerador está mayoritariamente solapado con las tareas del host.

kernel y un 48,4 % en el host. Esto último es relevante, ya que a pesar de que la mayor parte del procesamiento se realiza en el acelerador, el host sigue utilizando un gran porcentaje del tiempo total, demostrando el bajo poder de cómputo de en el host de las instancias F1, comparados con las instancias C5 y G5. También se puede ver esto en la Tabla 6.2, donde el tiempo de procesamiento de Dashing en la instancia f1.2xlarge es significativamente mayor que en la instancia c5.2xlarge, a pesar de ambas tener 8 vCPUs.

6.4.4. Desempeño del criterio de selección

En esta sección se evalúa el impacto del uso del criterio presentado en el Capítulo 4 para reducir el número de similitudes a calcular, centrándose solo en aquellos pares que pueden generar similitudes Jaccard sobre cierto umbral. El impacto del criterio de selección se midió en JACC-GPU, JACC-FPGA y una implementación multi-hebra en software del Algoritmo 16 en una instancia c5.9xlarge. Esta implementación se desarrolló en C++ usando la implementación de HyperLogLog acelerada en SIMD de la librería Bonsai, desarrollada por los mismos autores de Dashing, y disponible en `https://github.com/dnbaker/bonsai`. Como el concepto de pivotes

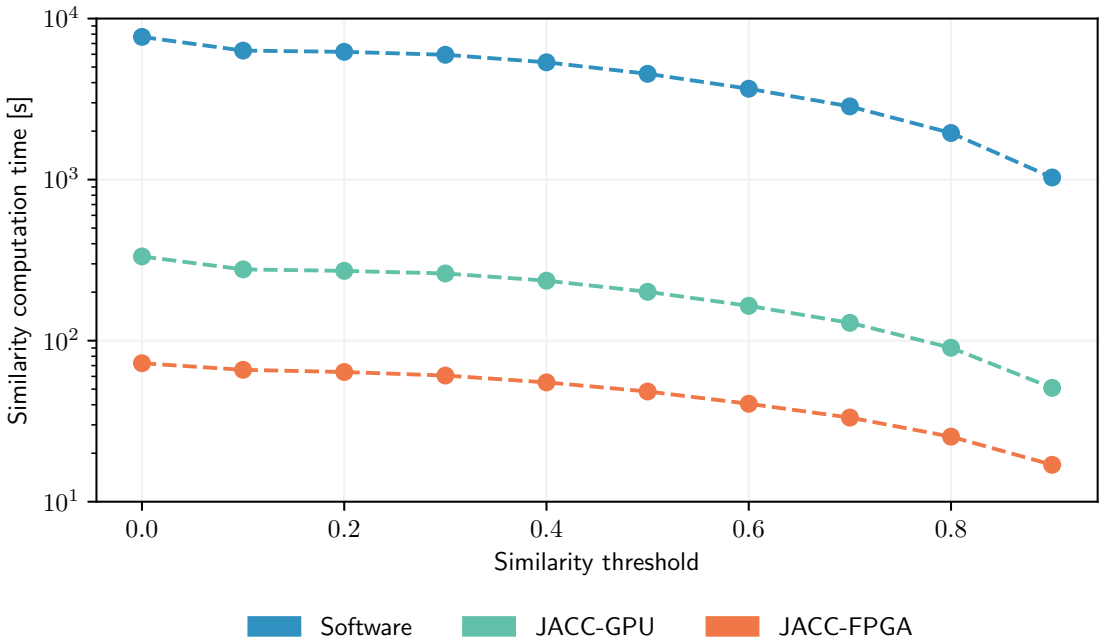


Figura 6.5: Tiempo de ejecución del Algoritmo 16 usando el criterio de selección descrito en el Capítulo 4, en función del umbral de similitud. Para la versión en software se creó una implementación propia usando la librería de HLL de Dashing, llamada Bonsai, que explota el paralelismo usando múltiples hebras e instrucciones de vectorización. La implementación en software corre en una instancia c5.9xlarge, JACC-GPU en una instancia g5.4xlarge y JACC-FPGA en una instancia f1.2xlarge. El tiempo no incluye el tiempo inicial, asociado a lectura, ordenamiento de sketches y carga de datos al acelerador en el caso de JACC-GPU y JACC-FPGA

solo es válido en aceleradores hardware y solo es posible leer un sketch a la vez por hebra, esta implementación utiliza $V = D = 1$, mientras paraleliza el procesamiento con OpenMP usando las 36 hebras disponibles. En todos los casos se utilizó el criterio para limitar el número de iteraciones en el Algoritmo 16.

La implementación en software lee las cardinalidades y los sketches HLL generados por Dashing para cada genoma de la base de datos RefSeq, luego los ordena de acuerdo a sus cardinalidades y luego ejecuta múltiples consultas para calcular las similitudes Jaccard para los pares de interés, variando el umbral entre 0 y $0,9$. La lectura y ordenamiento de los sketches tarda 81 segundos en la instancia c5.9xlarge. En el caso de las implementaciones en GPU y FPGA el host lee todos los sketches de un único archivo, generado en la etapa de construcción, el cual ya se encuentra ordenado. En estas implementaciones, el host sube el archivo y en cada

llamada al kernel se calcula la cantidad de sketches que cumplen el umbral, indicándole al acelerador hasta que punto debe realizar comparaciones. El proceso de lectura y subida de los datos al acelerador toma cerca de 11 segundos en las instancias f1.2xlarge y g5.4xlarge.

La Figura 6.5 muestra el tiempo de ejecución para las tres implementaciones en función del umbral h usado en el criterio de selección, definido en la Ecuación (4.6). En el gráfico, un umbral $h = 0$ significa que no se realiza una selección de pares, por lo que se calcula toda la matriz. En todas las implementaciones, el tiempo disminuye al aumentar el umbral, lo que es esperable debido a la reducción de pares a calcular, siguiendo el Algoritmo 16. Para umbrales superiores a $h = 0,4$, la implementación en software es más lenta que Dashing procesando la matriz de similitud completa, esto se debe a las optimizaciones a nivel de hebras que utiliza Dashing que no están incorporadas en esta implementación. Para umbrales mayores, el criterio de selección alcanza un mejor desempeño, para $h = 0,8$, la implementación en software es 2,5 veces más rápida que Dashing y 4 veces mayor que sí misma para $h = 0$.

El impacto en el tiempo de ejecución de JACC-GPU y JACC-FPGA es similar al de la implementación en software, pero los beneficios disminuyen en umbrales mayores a 0,7. Una razón es que el criterio se aplica sobre el conjunto de sketches que procesa el acelerador, por lo que es posible que pares que no cumplen el criterio sean incluidos por la granularidad de procesamiento en el acelerador. La segunda razón es que la reducción de operaciones solo está asociada al procesamiento, por lo que los tiempos fijos de llamadas y transferencias de datos comienzan a ser más relevantes en umbrales altos, por ejemplo para umbrales mayores a $h > 0,7$ el tiempo en JACC-FPGA es menor a 30 segundos y en JACC-GPU es menor a 100 segundos. Aun así, para un umbral $h = 0,8$ JACC-FPGA calcula el conjunto de pares en 25 segundos, alcanzando una aceleración de 2,9 veces sobre el cálculo de la matriz completa. De forma similar, JACC-GPU tarda 80 segundos cuando $h = 0,8$, lo que representa una aceleración de 3,7 veces sobre el cálculo de la matriz de similitud completa.

6.4.5. Uso de recursos en FPGA

La Tabla 6.3 muestra los recursos usados por los kernels que implementan las dos etapas de procesamiento en FPGA. Para el kernel de construcción de sketches, los recursos con mayor utilización son los bloques DSP, que son usados en operaciones aritméticas como multiplicaciones y sumas. El kernel instancia 64 módulos de cálculo de MurmurHash3, 8 constructores de sketches con ocho módulos HLL cada uno, cada módulo MurmurHash3 utiliza 24 DSP: 10 por cada multiplicación de 64-bits y 4 en otras operaciones. En total se utiliza un 22,4 % de los DSP

Tabla 6.3: Utilización de recursos en FPGA para cada kernel del proceso de cálculo de similitud de Jaccard.

Recurso	Construcción de sketches	Cálculo de similitud	Disponibilidad
LUTs	92 077	349 315	1 180 984
Registers	111 800	354 420	2 364 480
BRAMs	324	179	2160
DSP slices	1536	0	6840

disponibles en el FPGA XCVU9P. Este kernel usa 256 bloques de memoria interna (BRAM) para implementar los sketches y 68 BRAMs en memorias FIFO para las interfaces de lectura y escritura a memoria externa. El kernel utiliza un total de 15 % de las BRAMs disponibles en el FPGA. El resto de los recursos presenta un uso menor al 8 %.

El kernel de cálculo de similitud utiliza un 29,5 % de las LUTs, principalmente en lógica y operaciones aritméticas en las entradas de los módulos de unión entre sketches HLL. El kernel usa el 15 % de los registros, mayoritariamente en pipelines y buffers para aumentar el *fan out* de los buckets de entrada. Solo se utiliza un 8 % de las memorias disponibles, siendo necesarias dos BRAM por pivote y 19 como memorias FIFO en las interfaces de lectura y escritura a la memoria externa. A pesar de la baja utilización de memoria, no fue posible aumentar la cantidad de pivotes sobre 80 por limitaciones en los recursos disponibles para ruteo, tal como se discutió en la Sección 6.4.2.

6.5. Similitud usando Jensen–Shannon

Para el trabajo de estimación de similitud usando divergencia de Jensen-Shannon, se evalúan dos implementaciones, una en GPU y una en FPGA. No se realizó una comparativa con una versión en software debido a la ausencia de implementaciones eficientes en este ámbito. Las implementaciones utilizas son:

JSD-GPU: Implementación en dos etapas en la plataforma CUDA de NVIDIA según lo detallado en la Sección 5.4. En la etapa de construcción, la implementación extrae los k-mers asociados a cada base de ADN, insertándolos en un sketch de cuentas y luego en el arreglo de colas de prioridad. En la etapa de cálculo de similitud se realiza la unión de los arreglos de colas de prioridad, realizando el cálculo en submatrices debido a restricciones

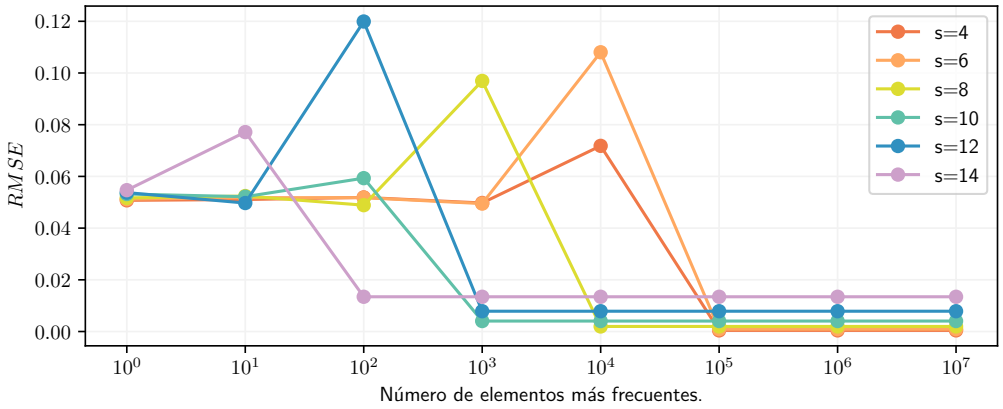
de memoria y al gran tamaño de los arreglos de colas de prioridad.

JSD-FPGA: Implementación en FPGA utilizando la plataforma Vitis de Xilinx y una descripción RTL de dos kernels de procesamiento, según lo descrito en la Sección 5.3. Esta implementación realiza la construcción de arreglos de colas de prioridad en la primera etapa, usando un sketch de cuentas y serializando su acceso. En la etapa de cálculo de similitud se utilizan varias instancias para procesar datos desde diferentes interfaces de memoria en paralelo.

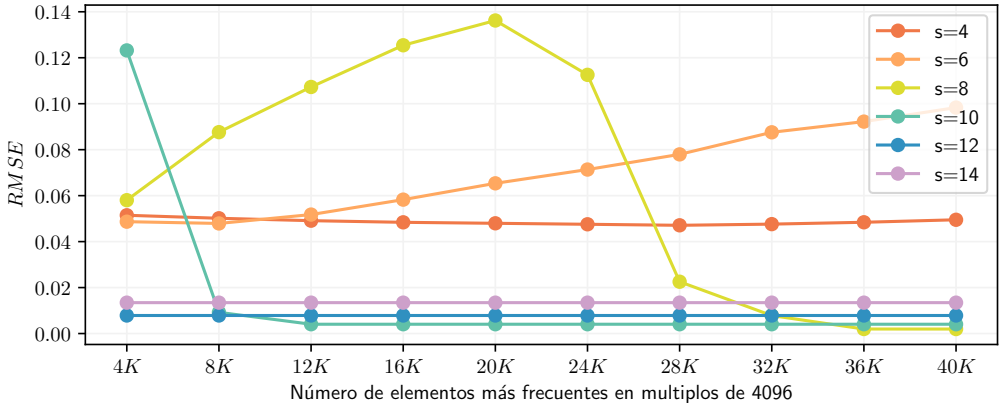
6.5.1. Selección de parámetros

Siguiendo el modelo propuesto por Soto et al. [92] y descrito en la Sección 5.2.2, es posible calcular la entropía de un conjunto usando una buena estimación de los elementos más frecuentes. La estimación de los elementos más frecuentes se puede realizar con un sketch de cuentas y una cola de prioridad para almacenar las estimaciones. En el mismo trabajo se puede ver cómo un arreglo de colas de prioridad (PQA) puede ser utilizada para este fin, introduciendo un error bajo y disminuyendo notablemente el tiempo de inserción. En el primer experimento, evaluamos el tamaño del PQA necesario para obtener un error tolerable considerando los 400 pares de genomas usados en la evaluación de la similitud de Jaccard. En conjunto, evaluamos el comportamiento usando un muestreo de datos con un valor de s entre 4 y 14 bits, siguiendo lo descrito en la Sección 5.2.3. Como se ve en la Figura 6.6a, a mayor muestreo el error converge en una cantidad de elementos en el PQA menor. De este resultado se infiere que los menores errores se obtienen cuando, independientemente del muestreo, la mayor parte de los elementos se encuentran en la cola de prioridad. De esta forma buscamos un rango tolerable de errores, con tamaños de PQA posibles de utilizar en una implementación en hardware. La Figura 6.6b muestra los errores en múltiplos de 4096 ($4k$) elementos en una cola de prioridad perfecta, mientras la Figura 6.6c muestra el mismo rango pero usando un PQA. Con esta información se decidió utilizar un PQA de $32K$ o $8 \times 4K$ elementos.

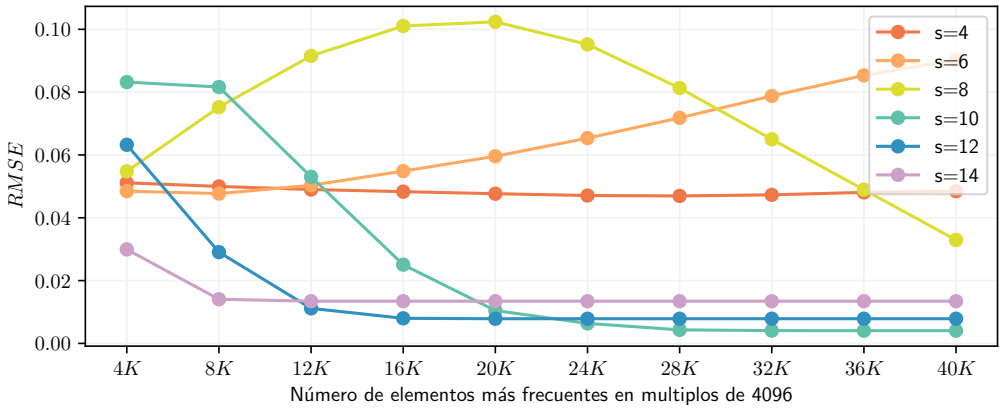
Posteriormente, evaluamos el error de estimación de JSD introducido al usar un sketch de cuentas. La Tabla 6.4 muestra la comparativa de tres sketches: Count sketch (CS), Count-Min (CM) y Count-Min CU (CM-CU), con parámetros $d \in [5, 7, 9]$ y $w \in [4K, 8K, 16K]$. Es posible ver que los errores son muy pequeños, confirmando la acción del muestreo y la elección de realizar la estimación de la entropía usando solo los elementos más frecuentes. De igual forma, siguiendo los errores presentados en la Tabla 6.4, se utilizó un sketch Count-Min de $5 \times 16K$ con un muestreo de 10 bits.



(a) Potencias de 10 entre 10^0 y 10^8.



(b) Multiplos de $4K$ (4096) elementos.



(c) Arreglo de colas de prioridad con $4K$ colas de 1 a 10 elementos cada una.

Figura 6.6: Error cuadrático medio (RMSE) de similitud usando divergencia de Jensen-Shannon (JSD). Las Figuras (a) y (b) usan estructuras exactas, la Figura (c) utiliza un arreglo de colas construido con una función MurmurHash3. Cada prueba fue realizada con un muestreo s entre 4 y 14 bits. Los errores fueron calculados respecto a la similitud JSD exacta de cada par de genomas, considerando los 400 pares utilizados por Dashing [7].

Tabla 6.4: Error cuadrático medio (Root-mean-square error, RMSE) de la similitud usando Jensen-Shannon con muestreo entre 8 y 12 bits; sketches Count sketch, Count-Min sketch y Count-Min CU de diferentes dimensiones; PQA de $8 \times 4K$ elementos. El error fue calculado usando los mismos 400 pares de genomas utilizados por Dashing [7]. El cálculo de la entropía solo considera los elementos más frecuentes, según la Ecuación (5.7).

s	w	RSME								
		CS			CM			CM-CU		
		$d=5$	$d=7$	$d=9$	$d=5$	$d=7$	$d=9$	$d=5$	$d=7$	$d=9$
8	4K	0,1733	0,1695	0,1623	0,1366	0,1355	0,1340	0,1147	0,1119	0,1101
	8K	0,1502	0,1407	0,1321	0,1102	0,1060	0,1018	0,0986	0,0963	0,0943
	16K	0,1174	0,1078	0,1009	0,0851	0,0801	0,0770	0,0834	0,0797	0,0774
9	4K	0,1298	0,1176	0,1070	0,0645	0,0564	0,0505	0,0477	0,0429	0,0404
	8K	0,0875	0,0725	0,0605	0,0293	0,0217	0,0168	0,0273	0,0209	0,0165
	16K	0,0464	0,0321	0,0248	0,0118	0,0093	0,0086	0,0118	0,0094	0,0086
10	4K	0,0855	0,0703	0,0573	0,0253	0,0172	0,0120	0,0231	0,0164	0,0117
	8K	0,0428	0,0293	0,0199	0,0076	0,0048	0,0042	0,0070	0,0047	0,0042
	16K	0,0153	0,0093	0,0058	0,0041	0,0041	0,0041	0,0041	0,0041	0,0041
11	4K	0,0437	0,0300	0,0215	0,0078	0,0055	0,0051	0,0077	0,0055	0,0051
	8K	0,0170	0,0100	0,0071	0,0051	0,0050	0,0050	0,0051	0,0050	0,0050
	16K	0,0068	0,0053	0,0051	0,0050	0,0050	0,0050	0,0050	0,0050	0,0050
12	4K	0,0223	0,0111	0,0088	0,0078	0,0077	0,0079	0,0078	0,0078	0,0079
	8K	0,0089	0,0079	0,0079	0,0078	0,0079	0,0079	0,0078	0,0079	0,0079
	16K	0,0079	0,0078	0,0079	0,0081	0,0079	0,0079	0,0079	0,0079	0,0079

6.5.2. Desempeño de implementaciones

El desempeño de las implementaciones es evaluado en el tiempo que tarda cada una de ellas en conseguir el resultado. En esta sección se evalúan las dos implementaciones descritas con anterioridad: JSD-GPU y JSD-FPGA. En todos los casos se muestra la mediana del tiempo en 10 ejecuciones. En la primera etapa, ambas implementaciones tardan un tiempo similar: 2390 segundos en GPU y 2520 en FPGA, según lo presentado en la Tabla 6.5. Como se evaluó en la Sección 6.4.3, el tiempo de procesamiento en la etapa de construcción está dominado por el tiempo de acceso al dispositivo de almacenamiento. En este caso, el tiempo es mayor al tiempo

Tabla 6.5: Tiempo de ejecución de las etapas de construcción de colas de prioridad y computo de divergencia de Jensen-Shannon para la base de datos RefSeq completa. Usando diferentes algoritmos e instancias de AWS. El reporte de desempeño para JSD-FPGA es en una instancia f1.2xlarge y para JSD-GPU en una instancia g5.4xlarge. El tiempo reportado es en segundos y la etapa de cálculo de JSD incluye la matriz completa.

Implementación	Instancia	Construcción de colas (s)	Cálculo de JSD (s)
JSD-FPGA	f1.2xlarge	2520	1240
JSD-GPU	g5.4xlarge	2390	2385

de construcción del otro método, presentado en la Tabla 6.2, debido al mayor tamaño de los arreglos de colas, que alcanzan los $31,28$ GiB, con un tiempo asociado de escritura de 151 segundos.

Como muestra la Tabla 6.5, en la etapa de cálculo de la matriz de similitud la implementación en GPU, JSD-GPU, tarda 2385 segundos, mientras la implementación en FPGA tarda 1240 segundos, cerca de la mitad del tiempo de la implementación en GPU. Este tiempo se explica por el grado de paralelismo presente en el diseño, usando múltiples instancias del módulo principal y el diseño en pipeline del algoritmo de unión entre colas, ilustrado en la Figura 5.9.

Las implementaciones en FPGA usaron un reloj de 250MHz para la comunicación con memoria externa. El kernel de construcción de arreglos de colas usó un reloj de 308MHz, mientras el kernel de similitud uno de 320MHz.

6.5.3. Uso de recursos en FPGA

La Tabla 6.6 muestra los recursos utilizados por la implementación en FPGA para ambas etapas de procesamiento. En el kernel de construcción de arreglos de colas de prioridad, el recurso con mayor uso son las BRAM, utilizadas en interfaces de memorias, pero principalmente en los sketches Count-Min. Cada sketch necesita de 5×16 BRAM, cada módulo fue implementado como un doble buffer, duplicando sus recursos para permitir una inserción continua de datos. Así, cada kernel usa 160 BRAMs, con total para la implementación de 1280, lo que sumado al resto de memorias utilizan el $73,42\,\%$ de las memorias disponibles en el chip. De igual forma, cada arreglo de colas de prioridad utiliza ocho UltraRAMs (URAM), con un total de 16 por kernel y 128 para la implementación completa. El uso de DSPs alcanza un $14,04\,\%$ del total disponible

Tabla 6.6: Utilización de recursos en FPGA para los dos kernels utilizados en el proceso de cálculo de divergencia de Jensen-Shannon.

Recurso	Construcción de colas	Cálculo de JSD	Disponibilidad
LUTs	75 219	757 827	1 180 984
Registers	105 181	108 353	2 364 480
BRAMs	1586	750	2160
URAMs	128	192	960
DSP slices	960	4032	6840

y está dado principalmente por las operaciones necesarias para el cálculo de las funciones hash, donde cada una utiliza 24 DSPs. Cada kernel instancia cinco módulos de MurmurHash3, uno por cada fila del sketch Count-Min, con un total 960 DSP en la implementación. La utilización del resto de los recursos es bajo el 7 %.

El kernel de cálculo de similitud utiliza un $64,17\,\%$ de LUTs disponible, este uso se da principalmente por el módulo de cálculo de entropía y los 16 módulos de logaritmo que éste instancia. El uso de registros alcanza un $4,58\,\%$ debido a los buffers de entrada para distribuir los datos en el kernel y en los pipelines internos de cada módulo. Esta implementación tiene un uso intensivo de memoria, llegando a usar 20 % de las URAM y un $34,72\,\%$ de las BRAM disponibles en el chip. El número de URAM está directamente relacionado a la cantidad de pivotes utilizada, donde cada uno de ellos requiere de ocho URAM, por lo que para los seis pivotes utilizados por interfaz se requieren 192 memorias. En el caso de las BRAM, el uso está distribuido entre las memorias FIFO de entrada y salida de los módulos de lectura y escritura, respectivamente, y en las tablas de cálculo de los logaritmo en el módulo de entropía. Además, el módulo tiene un uso de un $58,95\,\%$ de los DSPs disponibles, dado por las operaciones necesarias para el cálculo de entropía.

En esta implementación, a diferencia de la implementación del cálculo de similitud usando Jaccard, el número de pivotes está limitado por el uso de recursos, en particular por el uso de LUTs y la complejidad del ruteo que implica un uso alto de recursos. Debido a esta razón, el número de pivotes máximo que se puedo alcanzar en este chip fue de seis por interfaz, con un total de 24 pivotes.

Capítulo 7. Conclusiones

Este trabajo presenta el diseño de dos aceleradores para el cálculo de similitud entre genomas usando el paradigma de FPGA como servicio (FPGA-as-a-service). Incluyendo el diseño de algoritmo para el procesamiento y transferencia eficiente de datos genómicos. Las soluciones propuestas tratan los genomas como conjuntos de k-mers y, utilizando estructuras de datos probabilísticas, permiten estimar métricas de similitud como la similitud Jaccard o la similitud basada en la divergencia de Jensen-Shannon. Las arquitecturas propuestas utilizan los recursos disponibles en FPGA actuales, explotando el paralelismo de grano grueso y fino, haciendo uso extensivo de pipelines para acelerar las operaciones de inserción y consulta en las estructuras de datos, y en las estimaciones respectivas. Se utilizaron algoritmos de streaming para maximizar el desempeño, para poder explotar la alta capacidad de cómputo que tiene como contra la reducida cantidad de memoria interna en los aceleradores, principalmente en FPGAs. Además, en la similitud Jaccard, este trabajo propuso un criterio de descarte rápido de pares que, por su cardinalidad, no pueden generar cierta similitud, permitiendo reducir el tiempo de cómputo en los casos de uso más habituales para este tipo de aplicaciones.

En la similitud Jaccard, los resultados experimentales muestran que la aceleración en hardware es efectiva, acelerando la construcción y el cálculo de la matriz de similitud. Comparado con el estado del arte en la misma instancia de AWS, la implementación en FPGA es 2,9 veces más rápida en la etapa de construcción y 277 veces en el cálculo de la matriz de similitud. Sin embargo, debido a limitaciones físicas en la capacidad de cómputo y ancho de banda del medio de almacenamiento en las instancias f1.2xlarge, el desempeño en la etapa de construcción está limitado. Por esto, la etapa de construcción en la implementación en FPGA es 10 % más lenta que Dashing corriendo en una instancia con un costo por hora similar (g5.4xlarge). Igualmente, en la etapa de cálculo de la matriz de similitud, el acelerador es 58 veces más rápido que Dashing y cuatro veces más rápido que la implementación en GPU optimizada. Es esperado que esta diferencia aumente al aumentar la cantidad de genomas analizados, ya que el cálculo de similitud entre pares tiene una complejidad cuadrática.

En la similitud usando divergencia de Jensen-Shannon, los resultados muestran una aceleración de $1,9$ veces entre la implementación en FPGA y GPU para el cálculo de similitud. En esta solución no existe una implementación de referencia en el estado del arte, por lo que la implementación de la similitud de Jaccard puede ser utilizado como una referencia. En la etapa de construcción el tiempo está dominado por el tiempo de lectura de datos desde el medio de almacenamiento, con un componente adicional de escritura de los arreglos de cola, los cuales

son un 1600 % mayores que los sketches HLL utilizados para estimar la similitud Jaccard. En la etapa del cálculo de la matriz de similitud, la implementación en FPGA es casi dos veces más rápida que su equivalente en GPU, debido a las diferencias en el modelo de programación y al mayor paralelismo alcanzado en la implementación en FPGA.

Las implementaciones se ejecutan sobre un FPGA UltraScale+ XCVU9P de Xilinx, presentes en las instancias F1 de AWS EC2. La implementación de Jaccard utiliza menos de un 30 % de los recursos del chip, permitiendo la inclusión de más unidades de procesamiento para datos genómicos. En el caso de la similitud usando Jensen-Shannon, el uso de recursos supera el 60 %. Ambas implementaciones presentan un error de estimación bajo. Sin embargo, la implementación usando similitud Jaccard es más rápida y tiene un menor uso de memoria en resultados intermedios entre etapas. Por esto, debería ser la implementación utilizada por defecto, exceptuando aquellos casos donde sea necesario contrastarlo con otra métrica, o donde Jaccard no sea tan efectivo, como en aquellos genomas muy grandes [31].

De los resultados obtenidos, es posible afirmar que el uso de aceleradores como FPGA y GPU es efectivo en aplicaciones de genómica, ayudando a reducir los tiempos de procesamiento con un nivel de costo similar. Sin embargo, del análisis realizado, es posible notar que el foco de las plataformas de *cloud computing*, particularmente AWS, está en plataformas de aceleración basadas en GPU, dificultando el desarrollo de aplicaciones heterogéneas con coprocesadores en FPGA por las limitaciones descritas anteriormente en el *host*. Esto último se ve en las limitaciones en ancho de banda en instancias F1, estudiadas en la Sección 6.4.3, y los múltiples lanzamientos de nuevos tipos de instancias con GPU [97].

Una conclusión importante de este trabajo es la validación del uso de algoritmos de streaming basados en sketches para el diseño de aceleradores en hardware. Estos algoritmos permiten aprovechar el paralelismo disponible en un FPGA, sin la necesidad de almacenar todos los datos a procesar. Un ejemplo de esto es el arreglo de colas de prioridad o PQA, el que se diseñó para mantener los elementos más frecuentes en memoria con un tiempo de inserción bajo. Esta estructura permitió el cálculo de la entropía de un genoma a una alta velocidad, al permitir la inserción de un dato por ciclo. Las alternativas a esto eran implementar un algoritmo de ordenamiento con el que el acelerador habría tardado varios ciclos de reloj en insertar un elemento, o usar un arreglo sistólico con un alto uso de recursos que habría limitado enormemente la cantidad de elementos a mantener en la estructura y la cantidad de genomas procesados en paralelo.

La continuación de este trabajo puede ser el desarrollo de una plataforma en la nube para

el uso de los aceleradores diseñados usando el paradigma FPGA-as-a-service. Esto permitiría contrastar de forma rápida secuencias provistas por usuarios con una base de datos, lo que podría ser útil en aplicaciones de clustering, mapeo de secuencias, o búsqueda de los genomas más parecidos en la base de datos. También sería posible evaluar el desempeño de los aceleradores al usar datos metagenómicos, lo que podría ser útil para acelerar aplicaciones de taxonomía. Otros trabajos derivados son la combinación de Jaccard y Jensen-Shannon en un mismo acelerador permitiendo combinar ambas métricas para hacer más robusto el cálculo de similitud. O la creación de una biblioteca de algoritmos con aceleración en hardware para habilitar la creación de pipelines de procesamiento, lo que sumado a la capacidad de reconfiguración dinámica que posee la plataforma Vitis de Xilinx, podría ayudar en aplicaciones relacionadas con genómica y en otras áreas que requieran procesamiento de grandes volúmenes de datos. Adicionalmente, sería interesante evaluar el desempeño de aceleradores descritos en lenguajes de alto nivel como HLS o OpenCL, con los que se podría reducir el tiempo de desarrollo. Por último, diseñar arquitecturas multilenguaje para concentrar los esfuerzos de diseño en las etapas más importantes.

Anexo A. Publicaciones

JACC-FPGA: A hardware accelerator for Jaccard similarity estimation using FPGAs in the cloud

Publicado el 2023 en la revista Future Generation Computer Systems, presenta el diseño de algoritmos y aceleradores para el cálculo de la similitud de Jaccard en genomas. Este trabajo incluyó lo descrito en el Capítulo 4 y los resultados presentados en el Capítulo 6.

A high-throughput hardware accelerator for network entropy estimation using sketches

Publicado el 2021 en IEEE Access, presenta la estimación de la entropía de flujos de red usando una buena estimación de los elementos más frecuentes y asumiendo una distribución uniforme en el resto de los datos. En este trabajo se diseñó un acelerador capaz de funcionar a una frecuencia de reloj de 400 MHz, usando sketches de cuenta y un arreglo de colas de prioridad (PQA). El método propuesto en este trabajo presentó un error del 1,5 %.

A hardware accelerator for entropy estimation using the top-k most frequent elements

Presentado en 2020 en Euromicro Conference on Digital System Design y seleccionado como el "*Outstanding paper of the DSD 2020 Programme*". Este trabajo presentó la estructura PQA, evaluando su desempeño en el almacenamiento de los elementos más frecuentes de un flujo de red. Esto se acompañó con el diseño de un acelerador capaz de funcionar con una frecuencia de reloj de 354 MHz y obteniendo un error de estimación del 17 %.

Hardware acceleration of k-mer clustering using locality-sensitive hashing

Presentado el 2019 en Euromicro Conference on Digital System Design, presenta un acelerador para clustering de *motif* de ADN. Este trabajo implementa sketch MinHash usando un arreglo sistólico, procesando un k-mer por ciclo a una frecuencia de 350 MHz.

Referencias

[1] David Koslicki and Daniel Falush. Metapalette: A k-mer painting approach for metagenomic taxonomic profiling and quantification of novel strain variation. *MSystems*, 1(3):e00020–16, 2016.

[2] Ncbi reference sequence (refseq) database release 212. Visitada: 2022-06-01.

[3] Alexandre Almeida, Stephen Nayfach, Miguel Boland, Francesco Strozzi, Martin Beracochea, Zhou Jason Shi, Katherine S Pollard, Ekaterina Sakharova, Donovan H Parks, Philip Hugenholtz, et al. A unified catalog of 204,938 reference genomes from the human gut microbiome. *Nature biotechnology*, 39(1):105–114, 2021.

[4] Cuiping Li, Dongmei Tian, Bixia Tang, Xiaonan Liu, Xufei Teng, Wenming Zhao, Zhang Zhang, and Shuhui Song. Genome Variation Map: a worldwide collection of genome variations across multiple species. *Nucleic Acids Research*, 49(D1):D1186–D1191, 11 2020.

[5] Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13(7):1–11, 07 2015.

[6] Nicola Cadenelli, Zoran Jakšić, Jordà Polo, and David Carrera. Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94:148–159, 2019.

[7] Daniel N Baker and Ben Langmead. Dashing: fast and accurate genomic distances with hyperloglog. *Genome biology*, 20(1):1–12, 2019.

[8] Thomas D. Wu, Jens Reeder, Michael Lawrence, Gabriel Becker, and Matthew J. Brauer. Gmap and gsnap for genomic sequence alignment: Enhancements to speed, accuracy, and functionality. *Methods in molecular biology*, 1418:283–334, 2016.

[9] Yan Gao, Yongzhuang Liu, Yanmei Ma, Bo Liu, Yadong Wang, and Yi Xing. abPOA: an SIMD-based C library for fast partial order alignment using adaptive band. *Bioinformatics*, 37(15):2209–2211, 11 2020.

[10] Nauman Ahmed, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. Gpu accelerated api for alignment of genomics sequencing data. In *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 510–515, 2017.

[11] Sayan Goswami, Kisung Lee, Shayan Shams, and Seung-Jong Park. Gpu-accelerated large-scale genome assembly. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 814–824, 2018.

[12] Huiren Li, Anand Ramachandran, and Deming Chen. Gpu acceleration of advanced k-mer counting for computational genomics. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–4, 2018.

[13] Robin Kobus, André Müller, Daniel Jünger, Christian Hundt, and Bertil Schmidt. *MetaCache-GPU: Ultra-Fast Metagenomic Classification*. Association for Computing Machinery, New York, NY, USA, 2021.

[14] Arun Subramaniyan, Jack Wadden, Kush Goliya, Nathan Ozog, Xiao Wu, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Accelerated seeding for genome sequence alignment with enumerated radix trees. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 388–401, 2021.

[15] Yatish Turakhia, Gill Bejerano, and William J Dally. Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly. *ACM SIGPLAN Notices*, 53(2):199–213, 2018.

[16] Yatish Turakhia, Sneha D. Goenka, Gill Bejerano, and WIlliam J. Dally. Darwin-wga: A co-processor provides increased sensitivity in whole genome alignments with high speedup. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 359–372, 2019.

[17] Nathaniel Mcvicar, Chih-Ching Lin, and Scott Hauck. K-mer counting using bloom filters with an fpga-attached hmc. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 203–210, 2017.

[18] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–135, 2019.

[19] Amazon. Amazon EC2 F1 instances, Visitada: 2022-02-15.

[20] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. Seedex: A genome sequencing accelerator for optimal alignments in subminimal space. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 937–950, 2020.

[21] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanović, David A. Patterson, and Anthony D. Joseph. Fpga accelerated indel realignment in the cloud. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 277–290, 2019.

[22] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. When FPGA meets cloud: A first look at performance. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020.

[23] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H. Oh, Krste Asanovic, Jae W. Lee, and Lisa Wu Wills. Genesis: A hardware acceleration framework for genomic data analysis. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 254–267, 2020.

[24] Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome biology*, 17(1):1–14, 2016.

[25] Sairam Behera, Jitender S Deogun, and Etsuko N Moriyama. Minisoclust: Isoform clustering using minhash and locality sensitive hashing. In *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 1–7, 2020.

[26] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology*, 33(6):623–630, 2015.

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

[28] Mateusz Forc, Wiktor Kuśmirek, and Robert M Nowak. De novo genome assembly for third generation sequencing data. In *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018*, volume 10808, page 108083D. International Society for Optics and Photonics, 2018.

[29] David Moi, Laurent Kilchoer, Pablo S Aguilar, and Christophe Dessimoz. Scalable phylogenetic profiling using minhash uncovers likely eukaryotic sexual reproduction genes. *PLoS computational biology*, 16(7):e1007553, 2020.

[30] Alexis Criscuolo. On the transformation of minhash-based uncorrected distances into proper evolutionary distances for phylogenetic inference. *F1000Research*, 9, 2020.

[31] Natapol Pornputtapong, Daniel A Acheampong, Preecha Patumcharoenpol, Piroon Jenjaroenpun, Thidathip Wongsurawat, Se-Ran Jun, Suganya Yongkiettrakul, Nipa Chokesajjawatee, and Intawat Nookaew. Kitsune: A tool for identifying empirically optimal k-mer length for alignment-free phylogenomic analysis. *Frontiers in bioengineering and biotechnology*, 8:556413, 2020.

[32] Nithya Ramakrishnan and Ranjan Bose. Analysis of healthy and tumour dna methylation distributions in kidney-renal-clear-cell-carcinoma using kullback–leibler and jensen–shannon distance measures. *IET systems biology*, 11(3):99–104, 2017.

[33] Gregory E Sims, Se-Ran Jun, Guohong A Wu, and Sung-Hou Kim. Alignment-free genome comparison with feature frequency profiles (ffp) and optimal resolutions. *Proceedings of the National Academy of Sciences*, 106(8):2677–2682, 2009.

[34] Xuan Guo. Js-ma: A jensen-shannon divergence based method for mapping genome-wide associations on multiple diseases. *Frontiers in genetics*, 11:507038, 2020.

[35] XiaoFei Zhao. BinDash, software for fast genome distance estimation on a typical personal laptop. *Bioinformatics*, 35(4):671–673, 07 2018.

[36] Antonio Saavedra, Hans Lehnert, Cecilia Hernández, Gonzalo Carvajal, and Miguel Figueroa. Mining discriminative k-mers in dna sequences using sketches and hardware acceleration. *IEEE Access*, 8:114715–114732, 2020.

[37] Javier E Soto, Thomas Krohmer, Cecilia Hernandez, and Miguel Figueroa. Hardware acceleration of k-mer clustering using locality-sensitive hashing. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 659–662. IEEE, 2019.

[38] Yuval Bussi, Ruti Kapon, and Ziv Reich. Large-scale k-mer-based analysis of the informational properties of genomes, comparative genomics and taxonomy. *PloS one*, 16(10):e0258693, 2021.

[39] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[40] Ahmed M Moustafa, Arnav Lal, and Paul J Planet. Comparative genomics in infectious disease. *Current Opinion in Microbiology*, 53:61–70, 2020.

[41] Samuel Lipworth, Karina-Doris Vihta, Kevin Chau, Leanne Barker, Sophie George, James Kavanagh, Timothy Davies, Alison Vaughan, Monique Andersson, Katie Jeffery, et al. Ten-year longitudinal molecular epidemiology study of escherichia coli and klebsiella species bloodstream infections in oxfordshire, uk. *Genome medicine*, 13(1):1–13, 2021.

[42] Alexis Criscuolo. A fast alignment-free bioinformatics procedure to infer accurate distance-based phylogenetic trees from genome assemblies. *Research Ideas and Outcomes*, 5:e36178, 2019.

[43] Cene Gostinčar. Towards genomic criteria for delineating fungal species. *Journal of Fungi*, 6(4):246, 2020.

[44] Rhys PD Inward, Kris V Parag, and Nuno R Faria. Using multiple sampling strategies to estimate sars-cov-2 epidemiological parameters from genomic sequencing data. *Nature Communications*, 13(1):5587, 2022.

[45] Will PM Rowe. When the levee breaks: a practical guide to sketching algorithms for processing the flood of genomic data. *Genome biology*, 20(1):1–12, 2019.

[46] Will PM Rowe and Martyn D Winn. Indexed variation graphs for efficient and accurate resistome profiling. *Bioinformatics*, 34(21):3601–3608, 2018.

[47] Roderick Bovee and Nick Greenfield. Finch: a tool adding dynamic abundance filtering to genomic minhashing. *Journal of Open Source Software*, 3(22):505, 2018.

[48] Amir Joudaki, Alexandru Meterez, Harun Mustafa, Ragnar Groot Koerkamp, André Kahles, and Gunnar Rätsch. Aligning distant sequences to graphs using long seed sketches. *Genome Research*, pages gr–277659, 2023.

[49] Can Kockan, Kaiyuan Zhu, Natnatee Dokmai, Nikolai Karpov, M Oguzhan Kulekci, David P Woodruff, and S Cenk Sahinalp. Sketching algorithms for genomic data analysis and querying in a secure enclave. *Nature methods*, 17(3):295–301, 2020.

[50] F. P. Breitwieser, D. N. Baker, and S. L. Salzberg. Krakenuniq: confident and fast metagenomics classification using unique k-mer counts. *Genome Biology*, 19(1):198, Nov 2018.

[51] Antonio Saavedra, Cecilia Hernández, and Miguel Figueroa. Heavy-hitter detection using a hardware sketch with the countmin-cu algorithm. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 38–45, 2018.

[52] Hamid Mohamadi, Hamza Khan, and Inanc Birol. ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, 33(9):1324–1330, 01 2017.

[53] Chun-Pei Cheng, Kuo-Lun Lan, Wen-Chun Liu, Ting-Tsung Chang, and Vincent S. Tseng. DeF-GPU: Efficient and effective deletions finding in hepatitis B viral genomic DNA using a GPU architecture. *Methods*, 111:56–63, 2016. Big Data Bioinformatics.

[54] Sneha D. Goenka, Yatish Turakhia, Benedict Paten, and Mark Horowitz. Segalign: A scalable gpu-based whole genome aligner. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020.

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

[56] Da Tong and Viktor K. Prasanna. Sketch acceleration on fpga and its applications in network anomaly detection. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):929–942, 2018.

[57] Minjin Tang, Mei Wen, Junzhong Shen, Xiaolei Zhao, and Chunyuan Zhang. Towards memory-efficient streaming processing with counter-cascading sketching on fpga. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[58] Monica Chiosa, Thomas B Preußer, and Gustavo Alonso. Skt: A one-pass multi-sketch data analytics accelerator. *Proceedings of the VLDB Endowment*, 14(11):2369–2382, 2021.

[59] Javier E. Soto, Paulo Ubisse, Yaime Fernández, Cecilia Hernández, and Miguel Figueroa. A high-throughput hardware accelerator for network entropy estimation using sketches. *IEEE Access*, 9:85823–85838, 2021.

[60] Niranjan Nagarajan and Mihai Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14(3):157, 2013.

[61] Bonnie Berger and Yun William Yu. Navigating bottlenecks and trade-offs in genomic data analysis. *Nature Reviews Genetics*, 24(4):235–250, 2023.

[62] Antonio Saavedra, Cecilia Hernández, and Miguel Figueroa. Heavy-hitter detection using a hardware sketch with the countmin-cu algorithm. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 38–45. IEEE, 2018.

[63] Xilinx. Virtex UltraScale+. urlhttps://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html, 2019.

[64] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. Graph processing on FPGAs: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697*, 2019.

[65] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[66] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[67] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[68] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[69] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[70] Ward Douglas Maurer and Ted G Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[71] Andrii Gakhov. *Probabilistic Data Structures and Algorithms for Big Data Applications*. BoD–Books on Demand, 2019.

[72] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003.

[73] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[74] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[75] Daniel Ting, Jonathan Malkin, and Lee Rhodes. Data sketching for real time analytics: Theory and practice. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3567–3568, 2020.

[76] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. Fast concurrent data sketches. *ACM Transactions on Parallel Computing*, 9(2):1–35, 2022.

[77] Federica Ventruto, Marco Pulimeno, Massimo Cafaro, and Italo Epicoco. On frequency estimation and detection of heavy hitters in data streams. *Future Internet*, 12(9):158, 2020.

[78] Theophilus Wellem, Yu-Kuen Lai, Chao-Yuan Huang, and Wen-Yaw Chung. A flexible sketch-based network traffic monitoring infrastructure. *IEEE Access*, 7:92476–92498, 2019.

[79] Javier E. Soto, Cecilia Hernández, and Miguel Figueroa. JACC-FPGA: A hardware accelerator for Jaccard similarity estimation using FPGAs in the cloud. *Future Generation Computer Systems*, 2022.

[80] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[81] A Appleby. Murmurhash3 http://code. google. com/p/smhasher/wiki, 2014.

[82] Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. ntHash: recursive nucleotide hashing. *Bioinformatics*, 32(22):3492–3494, 07 2016.

[83] Thomas Wang. Integer Hash Function., 2014.

[84] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 683–692, New York, NY, USA, 2013. Association for Computing Machinery.

[85] Amit Goyal and Hal Daumé III. Approximate scalable bounded space sketch for large data nlp. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 250–261, 2011.

[86] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002.

[87] Younes Ben Mazziane, Sara Alouf, and Giovanni Neglia. A formal analysis of the count-min sketch with conservative updates. *arXiv preprint arXiv:2203.14549*, 2022.

[88] Paul Jaccard. The distribution of the flora in the alpine zone.1. *New Phytologist*, 11(2):37–50, 1912.

[89] P. Deutsch and J-L. Gailly. RFC 1950 (Informational).

[90] Dolev Adas and Roy Friedman. A fast wait-free multi-producers single-consumer queue. In *23rd International Conference on Distributed Computing and Networking*, ICDCN 2022, page 77–86, New York, NY, USA, 2022. Association for Computing Machinery.

[91] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

[92] Javier E. Soto, Paulo Ubisse, Cecilia Hernández, and Miguel Figueroa. A hardware accelerator for entropy estimation using the top-k most frequent elements. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 141–148, 2020.

[93] Páll Melsted and Bjarni V Halldórsson. Kmerstream: streaming algorithms for k-mer abundance estimation. *Bioinformatics*, 30(24):3541–3547, 2014.

[94] Amazon EBS volume types. Visitada: 2022-06-01.

[95] Amazon EBS–optimized instances. Visitada: 2022-06-01.

[96] Nuala A O'Leary, Mathew W Wright, J Rodney Brister, Stacy Ciufo, Diana Haddad, Rich McVeigh, Bhanu Rajput, Barbara Robbertse, Brian Smith-White, Danso Ako-Adjei, et al. Reference sequence (refseq) database at ncbi: current status, taxonomic expansion, and functional annotation. *Nucleic acids research*, 44(D1):D733–D745, 2016.

[97] Amazon EBS volume pricing. Visitada: 2023-08-08.