Universidad de Concepción
Dirección de Postgrado
Facultad de Ingeniería - Programa de Magíster en Ciencias de la
Ingeniería con mención en Ingeniería Eléctrica

Acelerador hardware para búsqueda de motivos emergentes en streams de secuencias de ADN

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

ANTONIO SEBASTIÁN SAAVEDRA MONDACA
CONCEPCIÓN-CHILE
2018

Profesor Guía: Dr. Miguel Figueroa T.
Profesor Co-guía: Dra. Cecilia Hernández R.
Comisión 1: Dr. Mario Medina C.
Comisión 2: Dr. Gonzalo Carvajal B.
Dpto. de Ingeniería Eléctrica, Facultad de Ingeniería
Universidad de Concepción

Universidad de Concepción
Facultad de Ingeniería
Departamento de Ingeniería Eléctrica

Profesor Patrocinante:
Dr. Miguel Figueroa T.
Profesor Copatrocinante:
Dra. Cecilia Hernández R.

Acelerador hardware para búsqueda de motivos emergentes en streams de secuencias de ADN

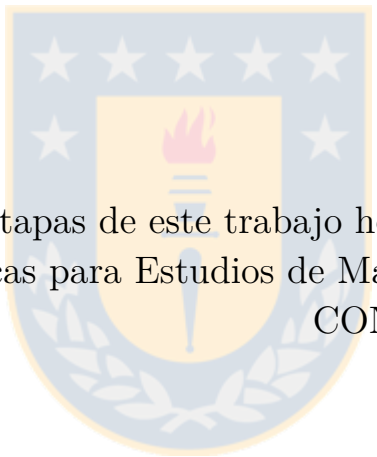
Antonio Sebastián Saavedra Mondaca

Informe de tesis de grado para optar al grado de
"Magíster en Ciencias de la Ingeniería con mención en Ingeniería Eléctrica"

Concepción, Chile
Agosto de 2016

I have no idea where this will lead us. But I have a definite feeling it will be a place both wonderful and strange.

Special Agent Dale Cooper



Durante las diferentes etapas de este trabajo he recibido el apoyo financiero del Programa de Becas para Estudios de Magíster en Chile de la agencia CONICYT del gobierno de Chile

Resumen

El descubrimiento de motivos en cadenas de ADN se define como la búsqueda de secuencias cortas de elementos compartidos en un conjunto largo de bases de nucleótidos que poseen una función biológica común. El descubrimiento de motivos entre los sitios de unión de los factores de transcripción, debido a la importancia de su función regulatoria en la expresión genética, resulta un problema de relevancia biológica. Este tipo de problemas presenta una alta complejidad computacional, especialmente debido a las dificultad de trabajar con bases de datos masivas. Las soluciones existentes en este tipo de problema se enfocan, por lo general, a plataformas en grandes clusters de alto costo, elevados tiempos de ejecución y consumo de potencia.

En este trabajo se desarrolla un acelerador hardware reconfigurable para la búsqueda de motivos emergentes en secuencias de ADN. Los motivos emergentes se definen como aquellos que cumplen requisitos establecidos de frecuencia dentro de la secuencias analizadas. Su búsqueda representa un problema biológicamente relevante que presenta altos requisitos de memoria y costos computacionales. La plataforma se propone en base a algoritmos capaces de resolver el problema de la búsqueda de elementos más frecuentes dentro de un stream de datos. Estos algoritmos utilizan estructuras de datos conocidas como sketches para realizar una aproximación al proceso de conteo para determinar los elementos más frecuentes. A diferencia de un conteo tradicional, la utilización de sketches permite resolver, a través de procesos probabilísticos, en espacio sublineal, la estimación de la frecuencia de cada elemento del stream.

Se implementaron en software los algoritmos CountSketch, Countmin, y Countmin-CU. Utilizando bases de datos biológicas públicas, se analizaron las dimensiones requeridas para operar con buena precisión y sensibilidad. El algoritmo Countmin-CU es capaz de encontrar los motivos emergentes de largos entre 10 y 20 utilizando arreglos de 65 mil contadores. El conteo tradicional requeriría sobre 100 mil millones. Se diseñó una arquitectura hardware dedicada que permite utilizar un FPGA como acelerador en un contexto de computación heterogénea. El algoritmo de streaming logra un balance adecuado entre el cómputo y los accesos requeridos a memoria permitiendo explotar el paralelismo fino de este tipo de plataforma. De esta manera, la lógica programable del FPGA con un diseño especializado nos permite reducir los costos de tiempo y el consumo de potencia de la solución. Este modelo de computación acelerada por hardware, con el FPGA nos permite trabajando con un reloj de 300MHz y consumiendo 3 Watts de potencia, nos permite alcanzar una aceleración de hasta 290 veces sobre la versión en software.

Tabla de contenidos

Lista de figuras

Lista de tablas

Acrónimos

ASIC circuito integrado de aplicación específica (del inglés *Application-Specific Integrated Circuit)*

CAM memoria accesada por contenido (del inglés *Content Addressable Memory)*

ChIP inmunoprecipitación de cromatina (del inglés *Chromatin Immunoprecipitation)*

CPU unidad de procesamiento central (del inglés *Central Processing Unit)*

CU actualización conservadora (del inglés *Conservative Update)*

DMA acceso directo a memoria (del inglés *Direct Memory Access)*

FPGA arreglo programable de compuertas (del inglés *Field Programmable Gate Array)*

GPU unidad de procesamiento gráfico (del inglés *Graphics Processing Unit)*

HDL lenguaje de descripción de hardware (del inglés *Hardware Description Language)*

HMS muestreo híbrido de motif (del inglés *Hibrid Motif Sampling)*

MEME esperanza-máximización múltiple para obtención de motivos (del inglés *Multiple Expectation Maximization for motif Elicitation)*

NGS secuenciación de alto desempeño (del inglés *Next-Generation Sequencing)*

PWM matrices de posicionamiento ponderado (del inglés *Position Weight Matrix)*

RAM memoria de acceso aleatorio (del inglés *Random Access Memory)*

RTL nivel de transferencia de registros (del inglés *Register Transfer Level)*

TF factor de transcripción (del inglés *Transcription Factor)*

TFBS sitio de unión de factor de transcripción (del inglés *Transcription Factor Binding Site)*

WAM método de vectores ponderados (del inglés *Weight Array Method)*

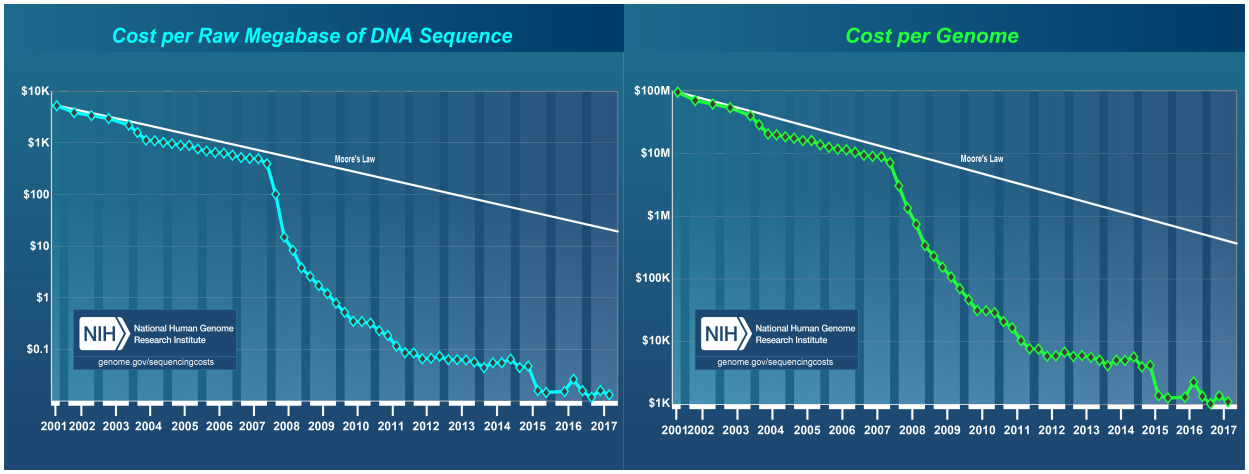
ZOOPS cero o una ocurrencia por secuencia (del inglés *Zero or One Occurence Per Sequence)*

Capítulo 1: Introducción

Dentro de las áreas de estudio de las ciencias biológicas, la genómica se ha dedicado a estudiar el funcionamiento, el contenido, la evolución y el origen de los genomas. Para esto se ha requerido el estudio en conjunto de diversas áreas del conocimiento incluyendo la biología molecular, la bioquímica, la estadística, las matemáticas, y la física, entre otras. El desarrollo tecnológico de las últimas décadas ha impactado en gran medida el desarrollo de esta área. Entre los problemas clásicos estudiados por el área se encuentra la secuenciación del ADN. Ésta consiste en la determinación del orden de las bases de pares de nucleótidos (Adenina, Citocina, Guanina y Timina) en una cadena de ADN. La secuenciación se convirtió en una herramienta práctica a partir de la utilización del método de terminación de cadena de Fred Sanger en la década de 1970. Este método clásico resulta costoso en tiempo y requiere extensa intervención manual para su utilización. Ésta fue la técnica central de secuenciación utilizada en el Proyecto Genoma Humano, que hasta el día de hoy se mantiene como el proyecto de colaboración internacional más grande dentro de la biología.

El desarrollo de los secuenciadores se vuelve fundamental luego de la finalización del Proyecto del Genoma Humano, al inicio de la década del 2000, cuando se comienza a utilizar la llamada secuenciación de alto desempeño (del inglés *Next-Generation Sequencing*, NGS). A partir de este punto, se ha reducido considerablemente el costo de la secuenciación del ADN, resultando accesible la obtención masiva de información de manera rápida. La figura 1.1 muestra el cambio en el costo de la secuenciación a través de los años. Entre los años 2007 y 2008 se observa el impacto producido a esta evolución por la introducción de la NGS. Debido a este desarrollo, en la actualidad, estas técnicas son aplicadas ampliamente en todo el espectro de la ciencias biológicas, incluyendo áreas como, por ejemplo, la investigación sobre enfermedades genéticas, el estudio de los orígenes de la humanidad, su evolución y los patrones de migración, la búsqueda de tratamientos para enfermedades actuales, el análisis de microbios y bacterias que habitan en nuestros cuerpos, o en el estudio del genoma propiamente tal, sus funciones y mecanismos de control.

La utilización de estas nuevas técnicas han traído consigo también nuevos desafíos y requerimientos que no estaban presentes en los métodos anteriores. Desde un punto de vista computacional, la secuenciación de alto desempeño implica una generación masiva de datos en altos flujos. Este flujo de alta cantidad de datos ha producido un cambio radical en la relación del área con las ciencias de la computación, convirtiendo en una necesidad el desarrollo y la implementación de nuevos sistemas computacionales, algoritmos y plataformas, que sean capaces



(a) Costo por millón de pares de bases (b) Costo por genoma

Fig. 1.1: Evolución en los costos de secuenciación de cadenas de ADN. Fuente: National Human Genome Research Institute (NIH) [1].

de manejar este mayor volumen de datos y que permitan aprovechar las ventajas que significan la secuenciación paralela a gran escala.

De esta manera, se evidencia la potencialidad del desarrollo en el ámbito de la bioinformática, enmarcada dentro de la esencia multidisciplinaria del estudio del genoma y la secuenciación de ADN. Dentro de los sistemas computacionales más utilizados en la actualidad destacan los basados en esquemas de computación en clusters. Estos sistemas permiten acelerar la ejecución de los algoritmos al distribuir sus costos computacionales en varios computadores interconectados por enlaces con bajas latencias. A pesar del alto costo de estas plataformas y su mantención, es común su utilización en el área a través de plataformas web que permiten la ejecución remota de algoritmos.

Este trabajo busca realizar un aporte al área de la bioinformática desde el diseño de hardware especializado a través del desarrollo de una plataforma hardware de aceleración computacional basada en una arquitectura dedicada. De esta manera se busca la optimización de tiempo, recursos y potencia requerida para la resolución del problema seleccionado, a través de un acelerador hardware con una arquitectura paralela. El problema seleccionado consiste en el conteo de motivos de secuencias de bases similares con una alta frecuencia de repetición en secuencias de ADN, enmarcado dentro de lo que se conoce como descubrimiento de motivos de los sitios de unión de los factores de transcripción (del inglés *TFBS motif discovery*).

1.1. Secuenciación del ADN

1.1.1. Algoritmos básicos de alineamiento

La NGS es un conjunto de herramientas de secuenciación masivamente paralelas. Estas herramientas son capaces de secuenciar varios millones de moléculas de ADN de forma simultánea y en tiempos muy inferiores a los requeridos por los secuenciadores de primera generación.

De los métodos experimentales más relevantes surgidos en esta nueva generación se encuentran los relacionados con las técnicas de inmunoprecipitación de cromatina (del inglés *Chromatin Immunoprecipitation*, ChIP). Estas técnicas consisten en seccionar una cadena larga de ADN en múltiples segmentos pequeños. Luego, a través de procesos bioquímicos que involucran la inserción de bases de nucleótidos sueltos, se puede observar qué bases se van uniendo a cada uno de estos segmentos pequeños. Esto permite en un par de etapas poseer la secuenciación de todos los segmentos pequeños de ADN. Debido a esta segmentación, esto resulta ser un proceso masivamente paralelo, y permite adquirir rápidamente todas las secuencias de segmentos pequeños. Luego a través de algoritmos computacionales (como Basic Local Alignment Search Tool, *BLAST*, o Complexity Analysis of Sequence Tracts, *CAST*) se pueden reconstruir las cadenas largas de ADN formada a partir de la unión de los segmentos más pequeños [2].

1.1.2. Descubrimiento de motivos

En el contexto de la secuenciación de ADN se conoce como *motif*, o motivos, a secuencias cortas de nucleótidos dentro de una cadena mayor con una función biológica común. El descubrimiento de nuevos motivos ha sido una de las aplicaciones más estudiadas en el área, debido a su significancia biológica y a su dificultad bioinformática. Este tipo de problemas generalmente se enfoca hacia la búsqueda de las secuencias de nucleótidos encargados de los procesos regulatorios de la expresión genética, como lo es un sitio de unión de factor de transcripción (del inglés *Transcription Factor Binding Site*, TFBS). Un factor de transcripción (del inglés *Transcription Factor*, TF) es una proteína específica cuya unión a su sitio correspondiente en una cadena de ADN es el proceso que inicia la transcripción de un gen de mayor tamaño. Estos TFBS son secuencias generalmente de tamaños de entre 8 a 28 bases de nucleótidos de largo, y consideran que los TF son capaces de reconocer en el ADN sus sitios de unión en conjuntos de secuencias similares, pero no necesariamente idénticos. A partir de la importancia de los TFBS para la regulación genética, la búsqueda de motivos entre éstos resulta un problema de

relevancia biológica [3].

La introducción de las nuevas tecnologías de secuenciamiento, como ChIP-Seq, ha permitido el análisis *in vivo* de los sitios de unión para TFs específicos. Esto ha convertido a los experimentos con ChIP en un caso de estudio para el descubrimiento de motivos [4].

1.1.2.1. Modelos de representación

Dentro del estudio de los algoritmos de descubrimiento de motivos es relevante el método o modelo para representar dichos motivos [5]. La forma más básica de representación, usada generalmente en tiempos de la secuenciación Sanger, corresponde a la enumeración de los distintos motivos, la cual sumado al conteo de estos permitía identificar sitios de unión fuertes o débiles para los distintos factores de transcripción.

Un modelo alternativo más flexible es la utilización de consensos. Esto incorpora a la enumeración la posibilidad de que existieran motivos con bases flexibles. Para esto, se expande el diccionario del ADN añadiendo nuevos caracteres que representan algún subconjunto de bases probables. Existen distintos métodos de consensos que son ampliamente utilizados.

La representación matemática más importante para motivos corresponde a las matrices de posicionamiento ponderado (del inglés *Position Weight Matrix*, PWM). Estas matrices almacenan la información de la dependencia de la frecuencia de las bases, según cual sea su posición interna dentro del motivo. La información se representa generalmente como logaritmos de probabilidades. Estas matrices se pueden obtener experimentalmente a partir de datos SELEX o computacionalmente a partir de información de ChIP-Seq, ChIP-Exo o PBM.

A pesar de ser ampliamente utilizado, el modelo de PWM presenta algunas limitaciones en lo relacionado a la dificultad de cuantificar la importancia del posicionamiento interno dentro del motivo. Por esto, han surgido modelos estadísticos más complejos que definen estas dependencias de posición internas, como lo son modelos de redes bayesianas o de Markov [6, 7].

1.1.2.2. Problemas de descubrimiento de motivos

El descubrimiento de motivos se puede separar en dos tipos de problemas diferentes. Uno consiste en la búsqueda de motivos previamente conocidos en conjuntos de datos nuevos, utilizado principalmente para buscar factores de transcripción que regulan genes específicos. Existen diversas bases de datos abiertas con PWMs para motivos conocidos de sitios de unión de factores

de transcripción, como lo son HOCOMOCO, JASPAR, TRANSFAC o footprintDB.

El otro tipo de problema es el descubrimiento *di novo* de motivos. Este consiste en la búsqueda de motivos de los cuales no se conocen los PWM para los factores de transcripción que los utilizan. Generalmente se busca identificar los motivos más sobre-representados en las secuencias de ADN donde se encuentran. El descubrimiento se realiza en base a modelos estadísticos. En la siguiente sección se desarrolla un análisis de algunas de las diversas herramientas disponibles enfocadas en este problema.

1.2. Avances en bioinformática

El desarrollo y la optimización de algoritmos dentro del área de la bioinformática, aplicaciones desde las ciencias informáticas para el procesamiento y manejo de información biológica, están cruzados por los mismos paradigmas del desarrollo computacional de los últimos años. Esto implica que los paradigmas de computación han sido llevados a plataformas con múltiples núcleos de cómputo. Estos esquemas modernos, que utilizan técnicas de paralelización, se basan en la distribución del cómputo en múltiples núcleos, reduciendo así los tiempos de ejecución de las aplicaciones sin la necesidad de aumentar las frecuencias de reloj. Sin embargo, esto requiere un cambio en el enfoque de los algoritmos y el planteamiento de los problemas en general. Para obtener rendimientos razonables usando más de un procesador, los algoritmos de interés deben poder manejar operaciones lo más independiente posibles unas de otras, y reducir los costos de comunicación entre los procesadores.

Hoy en día hay cuatro técnicas principales de optimización y paralelización de algoritmos: unidad de procesamiento central (del inglés *Central Processing Unit*, CPU) multinúcleos; instrucciones vectoriales y optimizaciones de caché; computación en clusters; y el uso de hardware especializado para la aceleración (como unidad de procesamiento gráfico (del inglés *Graphics Processing Unit*, GPU), arreglo programable de compuertas (del inglés *Field Programmable Gate Array*, FPGA) o circuito integrado de aplicación específica (del inglés *Application-Specific Integrated Circuit*, ASIC)). Todas estas técnicas han sido utilizadas dentro del ámbito de la bioinformática [8], enfocándose principalmente en los problemas de bajo nivel, generalmente consistentes en reconocimiento de patrones basados en secuencias de texto. Comúnmente, estos problemas de bajo nivel consisten en los conteos de ocurrencias de secuencias pequeñas de bases en bases de datos de secuencias de gran tamaño, los que luego son extendidos para buscar alineamientos extensos (alineamiento de secuencias) o los elementos más comunes (descubrimiento de motivos).

1.2.1. Algoritmos y plataformas para TFBS

Como se mencionó anteriormente, el descubrimiento de motivos para los factores de transcripción es un problema de gran significancia biológica, y un gran desafío computacional. Debido a sus dificultades computacionales y a su amplia utilización, muchas de las herramientas actuales de descubrimiento de motivos han sido implementadas como herramientas de código abierto accesibles a través de internet ejecutándose en grandes clusters de procesamiento. Estas herramientas funcionan a partir de las distintas fuentes de datos de ADN secuenciado, como SELEX, ChIP, ChIP-Seq, etc.

Dentro de los algoritmos principales para el descubrimiento de motivos en TFBS destaca la esperanza-máximización múltiple para obtención de motivos (del inglés *Multiple Expectation Maximization for motif Elicitation*, MEME) [9]. MEME es un algoritmo estadístico iterativo basado en el algoritmo de esperanza-maximización. Desde un punto de vista biológico, el algoritmo identifica y caracteriza motivos comunes en un conjunto de secuencias no alineados. Desde el punto de vista computacional, el algoritmo encuentra un conjunto de segmentos de caracteres que son aproximadamente similares entregando como resultados estos conjuntos agrupados en PWMs. MEME se encuentra implementado en la plataforma conocida como MEME-Suite [10], que incluye diversas herramientas para el descubrimiento de motivos. Esta plataforma permite realizar descubrimiento *di-novo* de motivos utilizando MEME modificado con el algoritmo GLAM2 que permite encontrar motivos con huecos entre las bases.

Dentro del resto de las plataformas disponibles para realizar descubrimiento de motivos se encuentran:

ChIPMunk: Herramienta desarrollada en Java, disponible tanto en código fuente como en un servidor web. Realiza el descubrimiento de motivos en secuencias de ADN o ARN.

HOMER: Herramienta de código abierto escrita en Perl. HOMER es un método diferencial de descubrimiento de motivos. Esto implica que a partir de dos conjuntos de secuencias busca identificar los elementos regulatorios que se encuentran enriquecidos en un set sobre el otro.

rGADEM: Herramienta eficiente para el descubrimiento *di novo* de motivos para datos de gran escala. Software de código abierto escrito en R.

MEME-ChIP: Plataforma web que realiza el descubrimiento *di novo* de motivos. Está diseñada para datos de ChIP-Seq de gran tamaño. Está alojada en servidores del instituto

nacional de salud de EUA (NIH).

Un análisis comparativo [11] de estas herramientas demuestra que todas, a pesar de poseer distintos métodos y algoritmos, dan resultados similares. La mejor evaluada en desempeño (en cuanto a la precisión de su resultados) es rGADEM. Como resultado general se observa un mejor desempeño en las herramientas enfocadas en la predicción de TFBSs individuales sobre las que buscan clusters de TFBSs.

1.3. Estado del arte en TFBS

En los últimos años se han presentado varios avances en cuanto al desarrollo de algoritmos para el descubrimiento de motivos en TFBS que mejoran el rendimiento de MEME estándar. Los algoritmos se pueden clasificar en dos tipos según los modelos de representación: los que utilizan secuencias de consenso y los basados en PWM. Los basados en consenso utilizan por lo general reconocimiento de patrones para buscar secuencias con ciertos esquemas conocidos. En cambio los basados en PWM generalmente utilizan técnicas estadísticas que permitan construir una PWM e iterativamente ir mejorando sus resultados para finalmente reportar los motivos de mayor relevancia.

Definición 1 (**k-mers**). *Secuencia de elementos del diccionario* $\{A, C, G, T\}$ *de largo* k.

Dentro de los algoritmos basados en secuencias de consenso destacan los algoritmos *Pair-Motif* [12] y su extensión *PairMotif+* [13], que busca disminuir sus tiempos de cómputo. Estos algoritmos plantean un método alternativo a la búsqueda por patrones que busca reducir tanto los candidatos a motivos como los k-mers analizados. Esta reducción se logra a partir de la selección desde las bases de datos de entrada de un conjunto de pares de k-mers que sean muy distintos entre ellos, los cuales se utilizan para buscar y agrupar sus similares a través de procesos estadísticos. Logran resultados comparables en precisión comparado con los algoritmos tradicionales y reducen en gran medida los tiempos de cómputo al procesar menos k-mers.

En cuanto a los algoritmos basados en PWM, la mayoría tiene buenos rendimientos con bases de datos pequeñas pero manejan con alto costo la utilización de bases de datos mayores. Por ejemplo, MEME-ChIP [14] es un algoritmo diseñado para adaptar MEME a los datos de ChIP-seq. Este algoritmo maneja de buena manera bases de datos de menor tamaño, pero al recibir como entrada bases de datos mayores, simplemente selecciona de manera aleatoria 600 secuencias entre las cuales se ejecuta. Entre otras implementaciones, EXTREME [15] tiene un muy buen

desempeño en el tiempo de cómputo pero tiene requerimientos de memoria muy grandes. Este algoritmo es similar a MEME, pero en vez de la esperanza-maximización tradicional utiliza una versión modificada del algoritmo que trabaja en línea mientras va obteniendo los datos. De esta manera reduce considerablemente los tiempos de ejecución en bases de datos grandes a costa de requerimientos de memoria mucho mayores. Algoritmos más recientes como FMotif [7], que utiliza búsqueda de patrones a través de árboles de sufijos, han logrado mejorar estas limitaciones, pero con el costo de que las optimizaciones funcionan bien para motivos cortos, sin obtener buenos desempeños al buscar motivos más largos. Algo similar ocurre con DREME [16], algoritmo diseñado para analizar bases de datos grandes de una manera eficiente en el tiempo, pero sólo busca motivos cortos, de entre cuatro y ocho bases de longitud. Este algoritmo realiza una búsqueda exhaustiva y completa de los motivos primero en el abecedario del ADN simple, para luego extender el abecedario con 11 caracteres comodines y complementar las búsquedas con modelos heurísticos similares a MEME.

Si bien la mayoría de los algoritmos están pensados para un tipo de datos de entrada proveniente de una fuente específica, hay algoritmos diseñados para utilizar múltiples fuentes de entrada. Dentro de las fuentes de datos NGS, como se ha mencionado anteriormente, las más relevantes son ChIP-seq, ChIP-exo y PBM. Dimont [17] fue diseñado como un algoritmo de alto desempeño independiente de la fuente de sus datos. Este algoritmo está basado en la aplicación de métodos probabilísticos a partir del modelo cero o una ocurrencia por secuencia (del inglés *Zero or One Occurence Per Sequence*, ZOOPS), que binariza la ocurrencia o no de un motivo en distintas secuencias, y utiliza PWMs como modelo de representación. Posee grandes costos computacionales para su ejecución con bases de datos de gran tamaño y está pensado para ser implementado en arquitecturas de clusters. A diferencia de Dimont, Slim [6] es un algoritmo que utiliza, aparte de las PWM, los modelos de árbol bayesiano y método de vectores ponderados (del inglés *Weight Array Method*, WAM). Todos estos modelos estadísticos también constituyen modelos de representación de motivos que evidencian la dependencia de cada base nucleótida dentro del motivo con las bases de sus posiciones cercanas. El algoritmo Slim (del inglés *sparse local inhomogeneous mixture model*) utiliza estos 3 modelos de una manera ponderada bajo un modelo de Markov. Ambos se encuentran disponibles en una versión desarrollada en Java dentro de la herramienta online Jstacs del servidor Galaxy de la Universidad Halle-Wittenberg.

Con el objetivo de elaborar un algoritmo que trabaje con bases de datos de ChIP-seq que sea capaz de manejar toda la información del flujo de secuencias, se desarrolló el algoritmo MCES [18] (del inglés *mining and combining emerging substrings*). Este algoritmo busca ser eficiente en términos de tiempo de cómputo y precisión en la identificación de motivos. MCES es capaz de reportar motivos de distintos largos por lo que no requiere la especificación de un largo

objetivo para su ejecución. Este algoritmo se basa en la búsqueda de k-mers emergentes definidos como aquellos que superen una frecuencia umbral previamente calculada y que se encuentran de forma mayoritaria en un conjunto de prueba por sobre uno de control. A partir de los k-mers emergentes, se realiza un proceso de combinación entre éstos para construir clusters de motivos que, a diferencia de los k-mers emergentes, consideran posibles variaciones en su composición. A partir de esta combinación se realiza un proceso posterior que permite agrupar los resultados en PWM.

Definición 2 (**Búsqueda de k-mers emergentes**). *Con un conjunto D_t de prueba y D_c de control con secuencias sobre $\Sigma = \{A, C, G, T\}$ de cardinalidad $|D_t|$ y $|D_c|$ respectivamente. Se define la frecuencia de un elemento ψ en cada conjunto como $freq(\psi, D_t)$ y $freq(\psi, D_c)$. Dado una frecuencia umbral $1/|D_t| \leq \rho_f \leq 1$, y una tasa de crecimiento $\rho_g > 1$, obtener todos los k-mers ψ de largo $k_{min} \leq k \leq k_{max}$ que cumplan $freq(\psi, D_t) \geq \rho_f$ y $\dfrac{freq(\psi, D_t)}{freq(\psi, D_c)} \geq \rho_g$.*

El algoritmo MCES está diseñado para ser implementado en clusters utilizando el paradigma de MapReduce. Esta estrategia permite distribuir sus costos en memoria entre múltiples nodos. De esta manera, el algoritmo completo se compone de la siguientes etapas:

1. **Minería de k-mers emergentes:** Este proceso se encarga de resolver el problema de *búsqueda de k-mers emergentes*. Para esto se utiliza un algoritmo de minería de datos[19] que trabaja a través de la búsqueda de patrones con un árbol de sufijos. Esta estrategia tiene un costo de memoria que, si bien es lineal en el tamaño de las bases de datos de entrada, excede la memoria disponible en los sistemas computacionales tradicionales. Para superar este impedimento, MCES combina el método de minería de k-mers con un proceso de MapReduce[20].

   Combinación de k-mers emergentes: En una primera etapa se realiza un proceso de clustering de los k-mers emergentes en el cual se agrupan los k-mers similares. Este proceso es ejecutado a través del algoritmo MCL. Éste es un algoritmo de clustering en grafos. Luego se realiza una segunda etapa de clustering dentro de los clusters anteriores que permite formar PWMs a partir de los k-mers de cada cluster. Este proceso se realiza con el mismo algoritmo.

Resultan interesantes los altos costos de memoria que entrega este proceso de minería en búsqueda de los k-mers emergentes. Estos costos son lineales con respecto al tamaño de las bases de datos de entrada. En los resultados reportados en [18] se observa que utilizando un solo núcleo sin MapReduce, se obtienen los mejores resultados en cuanto al tiempo de ejecución.

Sin embargo, para bases de datos que superan los 500 mil pares de bases, el procesador no es capaz de ejecutar el algoritmo por falta de memoria, por lo cual es necesaria la utilización de MapReduce. Se observa que, para bases de datos pequeñas, los costos de tiempo están dominados por el costo de la estrategia de MapReduce. De esta manera, la principal limitación de MCES está dada por sus requisitos de memoria. Al requerir memoria lineal con respecto a los datos de entrada, se ve obligado a utilizar MapReduce, que tiene un impacto en el tiempo de ejecución. Además del tamaño de las entradas, el algoritmo también requiere más memoria para analizar tamaños de k-mers mayores.

A partir del análisis de la estructura general del algoritmo MCES, se desprende el problema de la búsqueda de k-mers emergentes y se plantea un método para, a partir de estos k-mers, realizar el descubrimiento de motivos en TFBS. Si se redujeran las limitaciones de memoria, a través de un método de conteo que permita reducir el espacio requerido, se lograría un impacto general a todo el problema de búsqueda de motivos en TFBS. Desde este punto, esta tesis se enfoca en el problema de la búsqueda de k-mers emergentes y el estudio de algoritmos de conteo que permitan reducir el espacio requerido para encontrarlos. Utilizamos algoritmos de conteo aproximados en estructuras de datos llamadas sketches, que son capaces de realizar estimaciones de frecuencia de los elementos de un stream en un espacios de memoria sublineal.

1.4. Conclusiones

El descubrimiento de motivos entre los sitios de unión de los factores de transcripción es un problema de gran importancia biológica y de alta complejidad computacional [18]. El diseño y el desarrollo de plataformas basadas en hardware dedicado que resuelva este problema puede optimizar y mejorar los tiempos de cómputo, el área requerida y la potencia consumida de las soluciones actuales de manera significativa. Estas optimizaciones permitirían obtener un impacto positivo en la investigación biológica en el área.

1.5. Hipótesis de trabajo

Un acelerador hardware puede explotar el paralelismo de grano fino presente en algoritmos de estimación de frecuencia de elementos de un stream basado en sketch, reduciendo el tiempo de ejecución, costo y consumo de potencia del problema de selección de los motivos emergentes entre los sitios de unión de factores de transcripción en secuencias de ADN con buenos índices

de precisión y sensibilidad.

1.6. Objetivos

1.6.1. Objetivo general

Diseñar e implementar una plataforma de aceleración hardware en FPGA para el algoritmo de conteo de frecuencia para la resolución del problema de búsqueda de motivos emergentes en los sitios de unión de factores de transcripción (*TFBS*) en secuencias de ADN.

1.6.2. Objetivos específicos

- Determinar el algoritmo de conteo a utilizar considerando que se debe adecuar a la aplicación específica y debe permitir una implementación eficiente en hardware.
- Implementar el algoritmo en una plataforma software que permita la verificación del mismo a través de pruebas experimentales y la posterior comparación de desempeño con el hardware diseñado.
- Diseñar una arquitectura dedicada para el algoritmo de conteo de elementos frecuentes en un stream de datos.
- Aplicar esta arquitectura como acelerador hardware en FPGA para resolver el problema del descubrimiento de motivos emergentes a partir de secuencias de ADN adquiridas como streams de datos.

1.7. Metodología

La determinación del algoritmo a utilizar se realizó a través de un estudio bibliográfico del estado del arte en el tema. Se consideraron tanto las limitaciones propias del problema biológico a resolver como también las ventajas, desventajas y limitaciones del desarrollo de una arquitectura hardware dedicada. Para esto se estudió la existencia de implementaciones en hardware de los algoritmos propuestos y las especificaciones del problema biológico y sus bases de datos asociadas.

La implementación en software del algoritmo se programó en una plataforma basada en C/C++. Esta plataforma permite validar el algoritmo seleccionado para resolver el problema planteado. Además permitió realizar pruebas sobre el resto de los algoritmos propuestos, además de definir los parámetros necesarios para la implementación de la arquitectura dedicada. Esta implementación también permite evaluar el costo en tiempo de ejecución y potencia de una solución en software convencional. También se desarrolló software para realizar las validaciones funcionales del hardware implementado.

La arquitectura en hardware que implementa el algoritmo se diseñó como una propuesta general en primera instancia. Ésta fue verificada en cuanto a su correcto funcionamiento y uso de recursos. Los recursos requeridos fueron analizados a partir de los parámetros necesarios por el algoritmo, definidos anteriormente en software, para resolver el problema planteado. Para el diseño de la arquitectura, se explotaron las potencialidades de paralelización tanto del algoritmo como en la adquisición de datos. El diseño de la arquitectura fue plasmado en una implementación en SystemVerilog que se programó sobre un FPGA. La programación en SystemVerilog es modular y cuenta con un sistema diseñado para permitir modificar los parámetros de la implementación a alto nivel utilizando el lenguaje Python.

Con la arquitectura básica en hardware diseñada, se trabajó en su integración como plataforma de aceleración en un contexto de funcionamiento con un procesador de propósito general asociado. Para realizar esta integración se utilizaron de Xilinx su ambiente de desarrollo C++/OpenCL SDaccel con las herramientas de la suite Vivado. Esta integración permite utilizar Vivado para realizar los procesos de síntesis lógica e implementación del hardware diseñado en la tarjeta FPGA aceleradora. También permite, con SDaccel, desarrollar código en OpenCL que controla la tarjeta aceleradora y los intercambios de memoria necesarios para permitir utilizar nuestra lógica dedicada para acelerar el problema planteado.

1.8. Temario

En el resto de este documento se expone el diseño e implementación de la plataforma de aceleración hardware para el problema de búsqueda de k-mers emergentes en secuencias de ADN. El documento se estructura de la siguiente manera:

Capítulo 2: Se desarrolla la revisión bibliográfica para contextualizar el trabajo desde la bioinformática y el desarrollo de algoritmos de sketch útiles para la resolución del problema planteado.

Capítulo 3: Se presenta la elaboración de una plataforma en software que permite evaluar los distintos sketches. Además se describen las bases de datos utilizadas, y los parámetros propios para realizar una implementación que resuelva el problema.

Capítulo 4: Se explica el modelo de computación heterogénea derivada del uso de una plataforma de aceleración hardware. Se define y explica el diseño del hardware dedicado con que se implementó la solución al problema y cómo fue integrado en una plataforma de aceleración.

Capítulo 5: Se exponen los resultados obtenidos para la plataforma en software y la implementación en hardware. Se comparan los distintos algoritmos, y se presentan resultados del diseño como el uso de recursos lógicos del acelerador, su consumo de potencia y las diferencias en tiempo de ejecución.

Capítulo 6: Se presentan las conclusiones del trabajo desarrollado.

Capítulo 2: Revisión bibliográfica

2.1. Algoritmos de streaming

Para la resolución del problema de búsqueda de motivos en TFBS requerimos de un algoritmo de conteo que permita encontrar los elementos más relevantes en frecuencia de un conjunto de datos que idealmente opere en espacio sublineal. A partir de las implementaciones en hardware de algoritmos para TFBS [21] y las limitaciones generales de este tipo de plataformas, conocemos que es esencial para lograr buenos desempeños balancear la relación entre cómputo y accesos a memoria en el hardware. Un buen balance de este tipo nos permitirá explotar en forma efectiva el paralelismo de alta granularidad que nos ofrecen las plataformas como los FPGA. Por este motivo es que nos enfocaremos en los algoritmos de streaming. Los algoritmos de streaming están diseñados para el procesamiento de streams de datos, en los cuales la secuencia de elementos de entrada pueden ser solamente examinados a través de unas pocas pasadas de la información en forma de un flujo continuo de datos. Estos algoritmos requieren espacios de memoria muy inferiores al tamaño de las entradas, pero poseen restricciones en los tiempos de ejecución debido a la necesidad de operar en una o pocas pasadas del stream. Estas características los convierten en buenos candidatos para lograr el balance requerido entre cómputo y accesos a memoria.

Dentro de los algoritmos de streaming nos enfocamos en el problema de la búsqueda de elementos más frecuentes. Esto se denomina el problema de *Heavy Hitters* [22]. Existen dos versiones del problema de los Heavy Hitters, los cuales se diferencian en los límites de los errores que presentan.

Definición 3 (ℓ_1**-Heavy Hitters**). *Dado un stream de entrada A de largo n con elementos $a_j \in \{1, 2, ..., m\}$. Definimos f_i la cantidad de ocurrencias del elemento i. Dados los parámetros $0 < \varepsilon < \phi < 1$ determinar un conjunto $S \subseteq \{1, 2, ..., m\}$ para el cual si $f_i \geq \phi n$, entonces $i \in S$. Por otro lado si $f_i \leq (\phi - \varepsilon)n$, entonces $i \notin S$. Además para todo elemento $i \in S$, el algoritmo deberá entregar un estimado $\bar{f}_i$ que cumpla $|f_i - \bar{f}_i| \leq \varepsilon n$.*

Definición 4 (ℓ_2**-Heavy Hitters**). *Dado un stream de entrada A de largo n con elementos $a_j \in \{1, 2, ..., m\}$. Definimos f_i la cantidad de ocurrencias del elemento i. Definimos $F_i = \sum_{i=1}^{n} f_i^2$. Dados los parámetros $0 < \varepsilon < \phi < 1$ determinar un conjunto $S \subseteq \{1, 2, ..., m\}$ para el cual si $f_i^2 \geq \phi F_2$, entonces $i \in S$. Por otro lado si $f_i^2 \leq (\phi - \varepsilon)F^2$, entonces $i \notin S$. Además para todo elemento $i \in S$, el algoritmo deberá entregar un estimado $\bar{f}_i$ que cumpla $|f_i - \bar{f}_i| \leq \varepsilon\sqrt{F^2}$.*

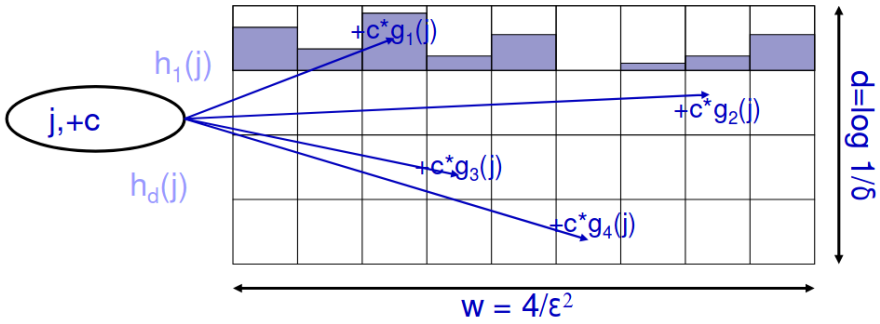
Los algoritmos para la resolución del problema de Heavy Hitters más relevantes se pueden

clasificar en dos grandes grupos [23]: los algoritmos basados en contadores, y los algoritmos basados en estructuras de datos llamadas sketches. Los basados en contadores para cumplir con las exigencias del problema[1] utilizan métodos para mantener en un menor número de contadores la información de los más relevantes, mientras que los basados en sketches lo hacen a través de respuestas aproximadas almacenadas en forma de un resumen, un sketch, del stream en memoria.

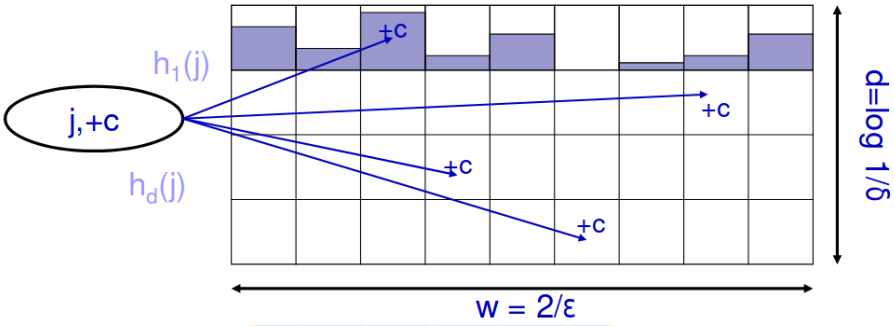
Los algoritmos en base a contadores están influenciados principalmente por un algoritmo planteado en la década de los 80. El algoritmo Majority [24], pionero en el área, resuelve el problema de búsqueda del elemento mayoritario. Este es un problema específico que es similar al Heavy Hitters, pero que, en vez de buscar un conjunto de elementos sobresalientes, busca un elemento mayoritario. Este algoritmo realiza el conteo de los elementos del stream a través de la siguiente estrategia: Se añade el primer elemento a un contador inicializado en uno; luego para todo el resto de los elementos se compara con el elemento almacenado. Si el nuevo elemento comparado es igual al almacenado, se aumenta el contador. En cambio, si es distinto que éste y el contador igual a cero, se almacena el nuevo elemento. En el último caso, de que el elemento sea distinto al almacenado, pero el contador no sea igual a cero, se decrementa su valor. Los algoritmos para Heavy Hitters basados en contadores fueron desarrollados a partir de las bases teóricas propuestas en Majority. Hay tres algoritmos relevantes en esta categoría.

El algoritmo Frequent [25] es una extensión de Majority que busca encontrar todos los elementos del stream cuya frecuencia supere la fracción $1/k$ del total de elementos a través del conteo en $k-1$ contadores. El algoritmo añade cada nuevo elemento a un contador disponible. Si encuentra un elemento presente entre ellos, lo incrementa. Si recibe un nuevo elemento y no quedan contadores disponibles, decrementa todos los contadores en uno, excepto en el caso en que alguno de los contadores tenga valor cero, caso en el cual se reemplaza el elemento almacenado por el nuevo. LossyCounting [26] es un algoritmo similar a Frequent. Este almacena tuplas constituidas por un valor límite inferior en su cuentas, y una variable Δ que almacena la diferencia entre este límite inferior y el superior. Cuando se procesa el i-ésimo elemento, si se encuentra entre los almacenados, se aumenta su límite inferior. Si no está presente, se crea una nueva tupla cuyo límite inferior se inicializa en cero y Δ en i/k. De manera periódica se eliminan todos los elementos cuyo límite superior sea inferior a $\lfloor i/k \rfloor$, garantizando de esta manera que al recorrer todo el stream estarán almacenados sólo los elementos que superen n/k en frecuencia. SpaceSaving [27] es un algoritmo similar a los anteriores en el sentido que almacena un subconjunto de los elementos del stream en contadores. Cuando se procesa un elemento

[1]Los requisitos ya mencionados de utilizar poco espacio en memoria con tiempos de cómputo pequeños para cada elemento.



(a) CountSketch



(b) Countmin Sketch

Fig. 2.1: Representación de ambos sketches. Fuente: Elaboración propia.

que está presente entre los almacenados, igual que en el resto de los casos, se incrementa su contador. Si ingresa un nuevo elemento que no está presente entre los almacenados, se reemplaza el elemento asociado al contador de menor valor por el nuevo, y se incrementa en uno este contador. De esta manera es capaz de encontrar los k elementos de mayor frecuencia utilizando k contadores.

2.2. Estructuras de conteo

Los algoritmos basados en sketches, a diferencia de los basados en contadores que son algoritmos determinísticos, permiten obtener una estimación de las ocurrencias de cada elemento del stream de datos a través de procesos probabilísticos. El sketch corresponde a una estructura de datos específica utilizada para mantener un conteo aproximado, o resumido, de los elementos vistos en el stream. Si bien es cierto que un esquema de conteo basado en algoritmos como SpaceSaving nos permite obtener los k-mers emergentes, no resulta idóneo para esta aplicación específica. Esto es debido a que requerimos de un modelo que presente mejores condiciones de ser implementado en hardware. Los algoritmos basados en contadores generalmente contienen estructuras de datos complejas y requieren realizar iteraciones de búsqueda dentro de

Tabla 2.1: Pseudo-código CountSketch

CountSketch(W, d, h) inicialización

1:	**IN:**	ancho sketch W,
		alto sketch d,
		funciones hash h y g
2:		C[1,1]...C[d,W] = 0
3:		**for** $j = 1$ to d **do**
4:		inicializar h_j
5:		inicializar g_j
6:	**OUT:**	sketch C

CountSketch(C, i) actualización

1:	**IN:**	sketch $C(W, d)$,
		elemento de entrada i
2:		**for** $j = 1$ to d **do**
3:		$C[j, h_j(\mathbf{a}_i)] = C[j, h_j(\mathbf{a}_i)] + g_j(\mathbf{a}_i)$

CountSketch(C, i) estimación frecuencia

1:	**IN:**	sketch $C(W, d)$,
		elemento a estimar i
2:	**OUT:**	$mediana_j \left(h_j(i) \cdot g_j(i)\right), j \in [1, d]$

los elementos almacenados en los contadores en cada paso. Los algoritmos basados en sketches permiten resolver de buena manera ambos requerimientos de nuestro problema, la determinación de los k-mers emergentes, y presentan mayor potencial para explotar el paralelismo en una implementación en hardware.

Como *sketch* denominamos una estructura de datos correspondiente a una proyección lineal de la entrada. Esto quiere decir que, si el stream de entrada es visto como un vector, el sketch es el producto de este vector con una matriz. Para reducir el espacio utilizado se utilizan funciones hash, también llamadas de dispersión, para determinar las proyecciones lineales. De esta manera el sketch como estructura de datos se constituye como una matriz (generalmente rala) conformada por estructuras individuales. Estas estructuras individuales corresponden a un vector fila compuesto por el dominio de salida de una función hash. Al utilizar múltiples funciones hash se conforma la matriz. Cada columna de la estructura contiene contadores, que en este contexto son llamados *buckets*. Definiremos los sketches de dimensiones $d \times W$, lo que implica que poseen d funciones hash, que distribuyen cada una los elementos de entrada a uno de sus W buckets asociados. Analizaremos los dos algoritmos basados en sketch más relevantes por sus usos en la literatura: CountSketch y Countmin Sketch. Además, introduciremos una modificación al algoritmo Countmin denominada Countmin-CU Sketch.

2.2.1. CountSketch

CountSketch[28] utiliza un sketch, como el explicado anteriormente, para obtener una aproximación al vector de ocurrencias de los elementos de un stream. El sketch está compuesto por la matriz C de $d \times W$ contadores, y dos funciones hash para cada fila: h_j que distribuye los

elementos en $[W]$, y g_j que distribuye los elementos en $\{-1, +1\}$. De esta manera cada elemento nuevo i aumentará para todo $1 \leq j \leq d$ en $g_j(i)$ el contador $C\left[j, h_j(i)\right]$. Para obtener la frecuencia estimada de cada elemento, se utiliza la mediana de entre todos los d contadores que tiene asociado dicho elemento. Si consideramos $\mathbf{a}$ como el vector de ocurrencias absolutas del stream de entrada, este algoritmo permite garantizar un error máximo limitado por la norma ℓ_2 de este vector de la forma $\varepsilon \cdot \|\mathbf{a}\|_2$ con una probabilidad de al menos $1 - \delta$. Para estas garantías es necesario utilizar un sketch con dimensiones $d = \mathcal{O}(log(4 \cdot \delta^{-1}))$ y $W = \mathcal{O}(\varepsilon^{-2})$. La figura 2.1a corresponde a una representación gráfica de este sketch. Se puede representar el algoritmo de CountSketch con el pseudo-código del cuadro 2.1.

2.2.2. Countmin Sketch

Otro algoritmo propuesto, que busca reducir el costo del tamaño del sketch a cambio de una garantía de error más relajada, es el Countmin Sketch[29]. Este algoritmo, al igual que CountSketch, está compuesto por la matriz C de $d \times W$ contadores, pero posee solamente una función hash por fila. Esta función hash es h_j encargada de la proyección lineal de los elementos. Cada elemento nuevo i aumentará para todo $1 \leq j \leq d$ en 1 el contador $C\left[j, h_j(i)\right]$. De esta manera la principal diferencia es que a diferencia de CountSketch, el algoritmo Countmin sólo posee una función hash por fila y siempre incrementa sus buckets elegidos. Para obtener la frecuencia estimada de un elemento se encuentra el mínimo de entre todos los d contadores asociados a dicho elemento. Este sketch garantiza que el rango del error estará limitado por la norma ℓ_1 de la forma $\varepsilon \cdot \|\mathbf{a}\|_1$ con una probabilidad de al menos $1-\delta$. Sin embargo para garantizar estos límites se requiere un espacio menor que CountSketch definido por $d = \mathcal{O}(log(\delta^{-1}))$ y $W = \mathcal{O}(\varepsilon^{-1})$. Una representación gráfica del sketch se puede observar en la figura 2.1b. Para definir la inicialización del sketch, el ingreso de un nuevo elemento y la estimación de su frecuencia se utiliza el pseudo-código del cuadro 2.2. Debido a que este algoritmo siempre incrementa los contadores asociados a cada elemento, posee solamente errores por sobre-estimación de frecuencia.

2.2.3. Countmin-CU Sketch

Una modificación interesante a los sketches se puede alcanzar utilizando la técnica de actualización de nominada actualización conservadora (del inglés *Conservative Update*, CU) [30]. Este método de actualización propone la idea que, en el proceso de añadir elementos a la estructura de conteo, sólo se debe incrementar su cuenta en la cantidad mínima necesaria para mantener su

Tabla 2.2: Pseudo-código Countmin sketch

Countmin(W, d, h) inicialización

1:	**IN:**	ancho sketch W,
		alto sketch d,
		funciones hash h
2:		C[1,1]...C[d,W] = 0
3:		**for** $j = 1$ to d **do**
4:		inicializar h_j
5:	**OUT:**	sketch C

Countmin(C, i) actualización

1:	**IN:**	sketch $C(W, d)$,
		elemento de entrada i
2:		**for** $j = 1$ to d **do**
3:		$C[j, h_j(\mathbf{a}_i)] = C[j, h_j(\mathbf{a}_i)] + 1$

Countmin(C, i) estimación frecuencia

1:	**IN:**	sketch $C(W, d)$,
		elemento a estimar i
2:	**OUT:**	$mínimo_j \left(h_j(i)\right), j \in [1, d]$

estimación en el valor correcto. Utilizando esta regla se demuestra que es posible mejorar las estimaciones con este tipo de algoritmos. Esta mejora se produce al reducir los errores producidos por sobre-estimación de las estructuras de conteo.

Este método de actualización ha sido aplicado a estructuras de datos como diversos algoritmos de sketches y filtros Bloom [31]. En nuestro caso específico, al tratarse de una optimización que busca reducir los errores introducidos por sobre-estimación, resulta llamativa su aplicación al algoritmo Countmin Sketch. El algoritmo Countmin-CU sketch [32] se obtiene de modificar en Countmin su actualización por la versión conservativo. En este caso, la cantidad mínima de incremento que permita mantener la aproximación precisa corresponde al incremento exclusivamente de aquellos contadores que contengan el valor mínimo del conjunto. De esta manera, el algoritmo Countmin-CU Sketch se define como el Countmin Sketch, pero con la excepción de que para ingresar un nuevo elemento, se aumentan condicionalmente solo aquellos contadores que contengan el valor mínimo entre ellos. El cuadro 2.3 presenta el pseudo-código de esta versión modificada del algoritmo Countmin.

La modificación CU también es posible incorporarla a CountSketch. Sin embargo, debido a que CountSketch produce errores por sobre-estimación y sub-estimación de las frecuencias, no posee un impacto tan significativo como con Countmin, que posee solo el tipo de error que ayuda a mejorar el complemento CU [32].

Tabla 2.3: Pseudo-código Countmin-CU sketch

Countmin-CU(W, d, h) inicialización

1:	**IN:**	ancho sketch W,
		alto sketch d,
		funciones hash h
2:		C[1,1]...C[d,W] = 0
3:		**for** $j = 1$ to d **do**
4:		inicializar h_j
5:	**OUT:**	sketch C

Countmin-CU(C, i) actualización

1:	**IN:**	sketch $C(W, d)$,
		elemento de entrada i
2:		**for** $j = 1$ to d **do**
3:		**if** $C[j, h_j(\mathbf{a}_i)] == min_j\left(h_j(i)\right), j \in [1, d]$
4:		$C[j, h_j(\mathbf{a}_i)] = C[j, h_j(\mathbf{a}_i)] + 1$

Countmin-CU(C, i) estimación frecuencia

1:	**IN:**	sketch $C(W, d)$,
		elemento a estimar i
2:	**OUT:**	$mínimo_j\left(h_j(i)\right), j \in [1, d]$

2.3. Análisis de la sensibilidad

Una comparación más directa entre los sketches propuestos en cuanto a la sensibilidad y los márgenes de error con que operan se presenta a continuación:

Recordando que en general la norma ℓ_n de un vector $\mathbf{x} \in \mathcal{R}^m$ corresponde a

$$\|\mathbf{x}\|_n = \sqrt[n]{\sum_{i=1}^{m} |\mathbf{x}_i^n|} \tag{2.1}$$

comenzamos el análisis con el caso del CountSketch.

2.3.1. CountSketch

A partir de la desigualdad de Chebyshev se pueden establecer límites para el error. Para una variable aleatoria x y una constante $c > 0$ esta desigualdad establece que, siendo $Pr[]$, $E[]$ y $Var[]$ operadores estadísticos que indican la probabilidad, esperanza y varianza respectivamente, se cumple:

$$Pr\left[|\mathbf{X} - E\left[\mathbf{X}\right]| \geq c\sqrt{Var\left[\mathbf{X}\right]}\right] \leq \frac{1}{c^2} \tag{2.2}$$

Para nuestro problema consideraremos $\mathbf{a}$ como el vector de ocurrencias que contabiliza todos los elementos a procesar por el sketch. De esta manera $\hat{\mathbf{a}}$ corresponderá al vector de ocurrencias estimadas obtenidas por el sketch, mientras que $\mathbf{a}_i$ y $\hat{\mathbf{a}}_i$ corresponderán al i-ésimo elemento de estos vectores de ocurrencia. Aplicando la ecuación 2.2 de la desigualdad de Chebyshev a $\hat{\mathbf{a}}_i$

tenemos

$$Pr\left[|\hat{\mathbf{a}}_i - \mathbf{a}_i| \geq c\sqrt{Var\left[\hat{\mathbf{a}}_i\right]}\right] \leq \frac{1}{c^2} \tag{2.3}$$

Considerando que la varianza de $\hat{\mathbf{a}}_i$ se puede delimitar por $Var\left[\hat{\mathbf{a}}_i\right] \leq \frac{\|\mathbf{a}\|_2^2}{W}$ [23], se puede plantear lo siguiente. Con W siendo el ancho del sketch, si definimos un error $\varepsilon = \frac{e^{1/2}}{W^{1/2}}$ y reemplazamos la constante $c = e^{1/2}$ la ecuación 2.3 se puede expresar como:

$$Pr\left[|\hat{\mathbf{a}}_i - \mathbf{a}_i| \geq \frac{c\varepsilon \|\mathbf{a}\|_2}{\sqrt{e}}\right] \leq \frac{1}{e} \tag{2.4}$$

Lo que determina que si se establece un tamaño de $W = \lceil \frac{e}{\varepsilon^2} \rceil$ la estimación se encontrará dentro del rango $\varepsilon\|\mathbf{a}\|_2$ con una probabilidad de al menos $1 - e^{-1}$. Para una reducción aún mayor del error es que utilizan múltiples funciones hash. Si se considera que de todos los buckets apuntados por las múltiples funciones hash se utilizará la mediana para la estimación, para obtener una estimación fuera del rango del error sería necesario que por lo menos la mitad de las estructuras inviduales estén fuera del rango. Por lo tanto, definimos una variable $\mathbf{X}$ que corresponde a las estructuras individuales que estén fuera del rango. Luego buscaremos la probabilidad de que más de la mitad del total de las estructuras estén en esta condición. Por lo tanto, si aplicamos los límites de Chernoff considerando d como el total de estructuras, y que cada una de ellas individualmente posee como probabilidad límite de error e^{-1} tenemos que:

$$Pr\left[\mathbf{X} > \frac{d}{2}\right] \leq e^{-\mathcal{O}(1)\cdot d} \tag{2.5}$$

Por lo tanto con este resultado se demuestra que si elegimos $d = \mathcal{O}(\frac{1}{\delta})$ garantizamos que $Pr\left[\mathbf{X} > \frac{d}{2}\right] \leq \delta$ con lo que con un sketch de dimensiones $W \times d$ se garantizan los márgenes del error con una probabilidad de al menos $1 - \delta$.

2.3.2. Countmin Sketch

Si analizando el caso de Countmin Sketch, tenemos que, a diferencia de CountSketch, al sumar siempre en los contadores de los buckets correspondientes y realizando la estimación con el mínimo entre los sumadores asociados a un elemento, se da la particularidad de que la estimación siempre será igual o superior al conteo real $\hat{\mathbf{a}}_i \geq \mathbf{a}_i$. Con esto en consideración comenzaremos el análisis a partir de la desigualdad de Markov, que plantea para una variable x y una constante $c > 0$

$$Pr\left[x > c \cdot E\left[x\right]\right] \leq \frac{1}{c} \tag{2.6}$$

A partir de un análisis también presentado en [23], conocemos que para la estimación $\hat{\mathbf{a}}_i$ su esperanza esta limitada, para un sketch de largo W, por:

$$E\left[\hat{\mathbf{a}}_i\right] \leq \mathbf{a}_i + \frac{\|\mathbf{a}\|_1}{W} \tag{2.7}$$

Por lo tanto podemos plantear la desigualdad de Markov aplicada a nuestra estimación $\hat{\mathbf{a}}_i$ de la siguiente manera:

$$Pr\left[\hat{\mathbf{a}}_i > \mathbf{a}_i + \frac{c\|\mathbf{a}\|_1}{W}\right] \leq \frac{1}{c} \tag{2.8}$$

Si reemplazamos c por e y definimos $W = \lceil \frac{e}{\varepsilon} \rceil$ para un error $0 < \varepsilon < 1$ obtenemos como resultado

$$Pr\left[\hat{\mathbf{a}}_i > \mathbf{a}_i + \varepsilon\|\mathbf{a}\|_1\right] \leq \frac{1}{e} \tag{2.9}$$

De manera similar al caso de CountSketch, este resultado nos demuestra que cada estructura individual (o fila del sketch) realiza una estimación dentro del límite planteado de error con una probabilidad inferior a $1 - e^{-1}$. Definiremos como $\mathbf{Y}_i$ el evento de que la estructura i-ésima entregue un estimado dentro del margen de error determinado, y $\mathbf{Y}$ como el evento buscado en que el estimado considerando todas las estructuras individuales se encuentre dentro del rango de error. Por lo tanto buscamos encontrar la probabilidad $Pr[\mathbf{Y}]$. Como consideraremos el mínimo de todas las estimaciones individuales como la estimación global esta probabilidad se convierte en

$$Pr\left[\mathbf{Y}\right] = Pr\left[\exists i \mathbf{Y}_i\right] = 1 - Pr\left[\forall i \bar{Y_i}\right] \tag{2.10}$$

Como cada estructura entrega un estimado independiente con probabilidad de error de $1 - e^{-1}$, y se poseen d estructuras obtenemos

$$Pr\left[\mathbf{Y}\right] \leq 1 - \frac{1}{e^d} \tag{2.11}$$

A partir de esto definimos $\delta = e^{-d}$ de tal manera que la probabilidad de que el estimado global se encuentre dentro del margen de error aceptado sea de $1 - \delta$. Luego resolvemos para d y obtenemos que la cantidad de estructuras individuales necesarias son $d = \lceil ln(\delta^{-1}) \rceil$.

2.3.3. Conclusiones del análisis

Desde un punto de vista formal, CountSketch obtiene un vector de ocurrencias estimadas $\hat{\mathbf{a}}$ el cual, con una probabilidad de al menos $1 - \delta$, contiene un error menor a $\varepsilon \left\|\mathbf{a}\right\|_2$. Por otro lado Countmin Sketch obtiene un vector de ocurrencias estimado que, con al menos la misma probabilidad, contiene un error menor a $\varepsilon \left\|\mathbf{a}\right\|_1$.

Hay que considerar que para todo vector $\mathbf{a} \in \mathcal{R}^n$ por propiedad de las normas se cumple que $\left\|\mathbf{a}\right\|_2 \leq \left\|\mathbf{a}\right\|_1 \leq \mathcal{O}(n^{1/2}) \left\|\mathbf{a}\right\|_2$. Esto evidencia que el CountSketch garantiza un error inferior al Countmin. Sin embargo el espacio requerido por ambos es distinto. El uso de espacio para CountSketch es de $\mathcal{O}(\frac{1}{\varepsilon^2} \cdot log(\frac{1}{\delta}))$ mientras que para CountMin es de $\mathcal{O}(\frac{1}{\varepsilon} \cdot ln(\frac{1}{\delta}))$.

2.3.4. Uso de actualización conservativa

El uso de CU busca reducir el error introducido en el conteo aproximado del sketch por sobre-estimación de la frecuencia. Se estima que permite reducir la magnitud de estos errores en un factor de 1.5 [31].

Para analizar los márgenes de sensibilidad en Countmin-CU sketch partiremos analizando la probabilidad de ocurrencia de un falso positivo [33]. Si consideramos una versión generalizada del sketch que posea contadores de solamente 1 bit, tenemos que, para que un elemento ψ esté contenido en el sketch, debe ocurrir $h_d(\psi) = 1$, para todos los valores de d. De esta manera la probabilidad de que un elemento no este contenido en el sketch después de añadir el primer elemento corresponde a $(1-\frac{1}{W})$. Luego de introducir N elementos únicos en el sketch, la probabilidad de no encontrar un elemento se vuelve $(1-\frac{1}{W})^N$. El recíproco de esta probabilidad, $1-(1-\frac{1}{W})^N$, indica la probabilidad de que una línea particular se encuentre en 1 después de N ingresos de elementos únicos. En el caso de Countmin Sketch, para obtener un elemento falso positivo necesitamos que d índices de d líneas distintas se encuentren con valor 1. De esta manera llegamos a la probabilidad de que, luego de N ingresos de elementos únicos, ocurra un falso positivo FP[2]:

$$FP(N) = \left(1 - \left(1 - \frac{1}{W}\right)^N\right)^d \tag{2.12}$$

A partir de esta probabilidad de falsos positivos en función de la cantidad de elementos distintos insertados N, se define la probabilidad de falsos positivos pasados, FPP, como la tasa promedio de falsos positivos hasta el elemento insertado N:

$$FPP(N) = \frac{\sum_{i=1}^{N} FP(i)}{N} \tag{2.13}$$

Utilizando las definiciones de $\mathbf{a}_i$ y $\hat{a}_i$ como la frecuencia de un elemento en el sketch y nuestra estimación de ésta respectivamente, si $\mathbf{a}_i > 1$ se puede garantizar que

$$Pr\left[|\hat{\mathbf{a}}_i - \mathbf{a}_i| > k\right] < FPP(A_k) \tag{2.14}$$

Donde A_i denota la cantidad de elementos que han sido insertados por lo menos i veces. Esta garantía se debe al hecho de que el error es exclusivamente de sobre-estimación. Denotando como D_i la cantidad de elementos que se estiman que aparezcan por lo menos i veces en el stream, podemos definir que para $i = 1$, $D_1 = A_1$, debido a que para una única inserción nunca

[2]Siempre y cuando las d funciones hash sean independientes entre ellas.

va a existir una sobre-estimación. Se puede estimar lo límites superiores de la esperanza de A_i a través de la fórmula recursiva:

$$E\left[A_i\right] = D_i + \sum_{j=1}^{i-1} \left(D_j - D_{j+1}\right) \cdot FPP\left(A_{i-j}\right) \tag{2.15}$$

Si hasta la inserción de un elemento se han producido k colisiones, es evidente que la última colisión no puede entregar una estimación menor a k. Por lo tanto la probabilidad del error la podemos limitar por $FP(A_k)$, en vez de $FPP(A_k)$:

$$Pr\left[|\hat{\mathbf{a}}_i - \mathbf{a}_i| > k\right] < FP(A_k) \tag{2.16}$$

Y combinando esta probabilidad con la esperanza de 2.15 podemos plantear que:

$$Pr\left[|\hat{\mathbf{a}}_i - \mathbf{a}_i| < k\right] > 1 - FP(A_k) \tag{2.17}$$

Para comparar este análisis con los parámetros estándar utilizados por los demás sketches, consideramos $k = \lceil \varepsilon N \rceil$ y $FP(A_{\lceil \varepsilon N \rceil}) = \delta$. Finalmente tenemos que:

$$Pr\left[|\hat{\mathbf{a}}_i - \mathbf{a}_i| < \varepsilon N\right] > 1 - \delta \tag{2.18}$$

Sobre todo con $FP(A_{\lceil \varepsilon N \rceil}) = \delta$ podemos demostrar que es posible mantener un valor de δ pequeño requiriendo valores pequeños de d. Esto presenta una mejora desde el punto de vista teórico contra los otros sketches.

El uso de algoritmos CU no garantiza matemáticamente un mejor desempeño comparado con las versiones tradicionales. Este análisis es una aproximación al análisis de sensibilidad presentado en [33]. Las técnicas tradicionales de análisis de los sketches no se pueden aplicar directamente al caso de los algoritmos modificados con CU. Sin embargo, ésta demuestra ser una aproximación validada experimentalmente, al igual que la estimación de la reducción en un factor de 1,5 del error de sobre estimación presentada en [31].

A partir del desarrollo de los distintos algoritmos en la plataforma en software desarrollamos comparaciones experimentales del desempeño de los tres algoritmos presentados, que permitieron comprobar las conclusiones de estos análisis.

2.4. Funciones hash

Como garantía de los límites de los errores de nuestros algoritmos es necesario que las funciones hash utilizadas cumplan con ciertos requisitos. Para el caso de CountSketch es necesario

utilizar una función hash que sea universal 2. Para Countmin Sketch se requiere que sea universal. Una familia $\mathscr{H}$ de funciones hash se define como universal si es que está constituida por un conjunto de funciones de $\mathscr{U} \mapsto w$ si se garantiza que cumple, para cualquier distintos $x, y \in \mathscr{U}$ que

$$Pr_{h \in \mathscr{H}}\left[h(x) = h(y)\right] \leq \frac{1}{w} \tag{2.19}$$

Una familia de funciones hash $\mathscr{U} \mapsto w$ se define como **universal 2** si es que además, con $x, y \in \mathscr{U}$, para cualquier $s, t \in [w]$, satisface

$$Pr_{h \in \mathscr{H}}\left[h(x) = s \wedge h(y) = t\right] = \frac{1}{w^2} \tag{2.20}$$

Debido a la facilidad con que se puede implementar en hardware[34], se utilizará la familia de funciones hash H3[35]. Esta es una familia universal 2. Se define formalmente como sigue.

Definición 5 (H3). *Dado el espacio $\mathscr{U}$ de elementos de entrada de i bits de largo y el espacio w de direcciones de j bits de largo. Denominamos por Q el conjunto de todas las matrices booleanas $i \times j$ posibles. Para $q \in Q$, denominamos $q(a)$ a su a-ésima fila, y para una entrada u, denominamos como $u(a)$ su a-ésimo bit. La función $h_q(x) : U \mapsto w$ se define como*[3]

$$h_q(x) = x(1) \cdot q(1) \oplus x(2) \cdot q(2) \oplus ... \oplus x(i) \cdot q(i) \tag{2.21}$$

El conjunto $\{h_q | q \in Q\}$ se define como la familia de funciones hash H3.

Esta familia de funciones resulta eficiente de implementar. Esto es debido a que las operaciones aritméticas que requiere son operaciones bit a bit las cuales se implementan de manera directa en el hardware. Las operaciones que requiere son AND y XOR lógicos, y la cantidad es proporcional a la cantidad de bits que contenga la llave de entrada.

2.5. Aplicaciones e implementaciones de sketches

2.5.1. Aplicacion de sketches para TFBS

Dentro del amplio rango de aplicaciones de estos sketches de conteo en espacio sublineal, se han desarrollado aplicaciones dentro del estudio de TFBS. El algoritmo *khmer*[36] es una

[3]Los operadores $\cdot$ y $\oplus$ corresponden a las operaciones lógicas bit a bit AND y XOR respectivamente.

aplicación de Countmin Sketch para el problema de conteo de k-mers. El objetivo de este algoritmo es el conteo de k-mers presente en un stream de datos, que si bien no corresponde exactamente al problema de búsqueda de k-mers emergentes, demuestra que estas estructuras de conteos son adecuadas para contar k-mers de secuencias biológicas. El algoritmo está enfocado en el conteo de k-mers para valores grandes de k, por ejemplo alrededor de $k = 32$. Para este tipo de largos demuestran que los vectores de conteos son mayormente ralos, existiendo dentro de las secuencias de ADN solamente un conjunto pequeño de los 4^{32} k-mers posibles. En estas condiciones, otros algoritmos de conteo pierden efectividad.

2.5.2. Aceleración hardware para TFBS

Dentro de la literatura especializada, la aceleración hardware especializada para algoritmos bioinformáticos se encuentra distribuida principalmente entre plataformas basadas en GPU y FPGA. La mayoría de estas plataformas se enfocan en la implementación de los problemas de bajo nivel de reconocimiento de patrones en las secuencias de bases de ADN. Las ventajas más evidentes de estos sistemas son las relaciones entre desempeño y costo, o desempeño y potencia consumida, sobre todo si se les compara con los grandes clusters donde se implementan tradicionalmente los algoritmos.

2.5.2.1. Aceleración basada en GPU

Considerando las implementaciones en GPU de aceleradores hardware para bioinformática, la mayor cantidad está enfocada en el problema de alineación de secuencias. Existen múltiples implementaciones del algoritmo BLAST en GPU, como GPU-BLAST, CUDA-BLASTP, TERA-BLAST, NCBI-BLAST, entre otros.

Dentro del problema del descubrimiento de motif, destaca la implementación de un algoritmo modificado para adecuarse de manera adecuada a la arquitectura de una GPU presentada en [37]. El algoritmo presentado, llamado muestreo híbrido de motif (del inglés *Hibrid Motif Sampling*, HMS) esta implementado en una tarjeta NVIDIA utilizando herramientas CUDA. A partir de modelos estadísticos, la tarea de calcular las probabilidades a cada motivo es costoso computacionalmente. El algoritmo es iterativo de tipo Monte Carlo. Aprovechándose de que algunos cálculos se pueden realizar de manera simultanea, se acelera la ejecución con una GPU, reduciendo así los tiempos de ejecución. Comparando con el algoritmo HMS original implementado en una CPU, la versión acelerada alcanza tiempos de ejecución del orden de 10 veces

inferiores [38].

2.5.2.2. Aceleración basada en FPGA

Al igual que en el caso de las GPU, las aplicaciones en FPGA están mayoritariamente concentradas a los problemas de bajo nivel de alineamiento. Entre las herramientas más populares se encuentran implementaciones de BLAST en FPGA [39]. En esta implementación se presenta como cuello de botella principal la cantidad disponible de memoria interna del FPGA (Xilinx Virtex-5) y los tiempos de transferencia de datos en la entrada, limitando así la posible aceleración teórica máxima, pero aún alcanzando importantes velocidades. Otra implementación de BLAST realizada en un sistema de múltiples FPGA de bajo costo (COPACOBANA 500) que cuenta con 120 Xilinx Spartan-3 [40] consigue un desempeño con aceleraciones importantes al compararla con sistemas basados en CPU, pero no alcanza a las implementaciones en FPGA con mayor lógica disponible (como la mencionada en [39]).

Como mirada general a la aceleración hardware a través de FPGA, es necesario categorizar los algoritmos en términos de sus requerimiento de cómputo y memoria [21]. En bioinformática, la mayoría de problemas son altamente costosos en cómputo con volúmenes grandes de datos. La flexibilidad y adaptabilidad de los FPGA permiten convertir algoritmos dominados por sus costos de cómputo en algoritmos dominados por sus costos de anchos de bandas de comunicación. Ejemplos de estos algoritmos son los de alineamiento de secuencias múltiples (BLAST, CAST, etc). Sin embargo otros algoritmos de bioinformática siguen siendo limitados por sus costos de cómputo a pesar de las flexibilidades de los FPGA. Éstos incluyen algoritmos de áreas diversas tales como predicción genética, predicción de estructuras de ARN y proteínas, construcción de árboles filogenéticos, y otros análisis de secuencias de ADN o proteínas. Aún cuando las principales limitaciones son las conexiones físicas entre FPGA y memorias, el uso de FPGA trae grandes mejoras al desempeño, incluidos los tiempos de ejecución y costos energéticos, con respecto a otras opciones de implementaciones.

2.5.3. Implementaciones en hardware de algoritmos de sketch

Dentro de la literatura especializada se encuentran trabajos que implementan en una arquitectura hardware dedicada algún algoritmo de sketch, concebidos para aplicaciones de análisis de tráfico de redes.

El algoritmo Countmin Sketch fue implementado en hardware utilizando una FPGA para

análisis de tráfico en [41]. El problema a resolver es la búsqueda de elementos frecuentes en streams de datos de comunicación en los enlaces de las redes. Si bien no es una aplicación dentro del área de la bioinformática, presenta una propuesta de arquitectura para la implementación del algoritmo que ha sido verificada y validada en un FPGA Xilinx Virtex-7 XC7VX1140 con un Countmin Sketch de dimensiones $65{,}536 \times 5$.

Una implementación similar [42] utiliza Countmin Sketch para realizar extracción de características del tráfico de red. Está implementado en un FPGA Xilinx Virtex II XC2V1000 que utiliza un reloj de 270MHz clock. Las dimensiones del sketch implementado son de $16{,}384 \times 4$.

Durante el desarrollo de esta tesis se publicaron dos trabajos relevantes sobre implementaciones de sketches en FPGA con hardware dedicado. El primer trabajo propone una implementación en un FPGA Virtex UltraScale de Xilinx de CountSketch con dimensiones $1{,}048{,}576 \times 8$ que permite realizar detección de heavy hitters en tráfico de red en un enlace Ethernet de 100Gbits/s [43]. El otro trabajo propone arquitecturas genéricas para accelerar con hardware Countmin y K-ary Sketch [44]. Se plantea también la incorporación de técnicas de postprocesamiento al sketch que permitan realizar la detección tanto de heavy hitters, como de heavy change. Reportan su mejor desempeño con un sketch de dimensiones $32{,}768 \times 5$. Este hardware puede operar en enlaces Ethernet de 150 Gbits/s para detección de heavy hitters y en 100 Gbits/s para detección de heavy change.

Capítulo 3: Bases de datos y plataforma en software

En este capitulo se describe el desarrollo e implementación la plataforma en software. En primera instancia se describen y caracterizan las bases de datos de secuencias de ADN utilizadas para probar los algoritmos. Luego se describe las implementaciones de los algoritmos CountSketch, Countmin Sketch y Countmin-CU Sketch realizadas en lenguaje C/C++.

En conjunto, el software diseñado con las bases de datos permiten la evaluación de los algoritmos aplicado al problema específico propuesto y presenta resultados de validación para la implementación en hardware.

3.1. Descripción de la base de datos

La plataforma software desarrollada utiliza como datos de entradas base de datos en forma de stream que están conformadas por secuencias de ADN. Las bases de datos utilizadas corresponden a información provenientes de procesos de secuenciación ChIP-seq.

Debido a que nuestro problema, la búsqueda de k-mers emergentes, se encuentra enmarcado en los procesos de descubrimiento de motivos de TFBS, hay sectores dentro de los genomas que resultan de más interés para el estudio que otros. En concreto, los métodos de secuenciación ChIP-seq entregan ciertas áreas en un genoma que han sido enriquecidas, que son denominadas como *peaks*. Estas áreas corresponden a aquellas en que el ADN ha interactuado con proteínas. En el caso de que esta proteína sea un TF, los peaks corresponden a potenciales TFBS. Para el descubrimiento de motivos entre TFBS se utilizan las áreas peaks como la señal del entrada donde buscar los elementos más frecuentes. Las áreas que no corresponden a los peaks son utilizados como señales de control.

Nuestra plataforma ha sido probada con 5 bases de datos biológicas. Todas estas fueron obtenidas a partir de la información de peaks en secuencias ChIP-seq obtenidas del repositorio MGA presentado en [45]. Cuatro de estas bases de datos corresponden a las de mayor dimensión entre las utilizadas en [18]. Al igual que en dicho trabajo, utilizamos la información de los peaks ChIP-seq. De esta manera las bases de datos están conformadas por múltiples secuencias de bases de ADN, cada una de las cuales está centrada en un peak. A partir del centro del peak,

Tabla 3.1: Dimensiones bases de datos. Fuente: Elaboración propia.

Base de datos	Tipo	Secuencias	Largo	Total bases
E2f1	prueba	20.696	200	4.139.200
Esrrb	prueba	21.644	200	4.328.800
Tcfcpl1	prueba	26.908	200	5.381.600
Ctcf	prueba	39.601	200	7.920.200
Control	control	43.283	200	8.656.600
Hepg2	prueba	252.660	200	50.532.000
Hepg2ctr	control	505.320	200	100.064.000

se seleccionan los 100 pares de bases hacia cada lado, obteniendo de esta manera secuencias de 200 bases de longitud. La información de las secuencias fueron extraídas de la base de datos mESC [46] (del inglés mouse embryonic stem cell), correspondiente a peaks en secuencias de células madre de embriones de ratón. Basado en el gen Esrrb, esta base de datos de prueba contiene 21.644 secuencias de 200 bases de longitud.

Además de las bases de datos públicas de ChIP-seq Encode, replicamos este mismo proceso para obtener bases de datos a partir de los genes E2f1, Ctcf y Tcfcpl1, también con información genética de ratones de mESC. Para obtener una base de datos de mayores dimensiones, se generó una señal de prueba a partir del genoma humano. Para esto se utilizó el gen de Encode Hepg2-Pol2Pk.

Para los conjuntos de control se utilizaron secuencias de los respectivos genomas que también poseen 200 bases de longitud cada uno. A diferencia de los conjuntos de prueba, que se encontraban centrados en las zonas peaks, la bases de datos de control se obtienen comenzando a 400 bases de distancia de estas zonas peak.

La tabla 3.1 presenta un resumen de las dimensiones de estas bases de datos.

A continuación analizaremos los vectores de ocurrencia en estas bases de datos para k-mers de distintos largos en el rango de trabajo, y además aproximaremos las frecuencias de umbral esperadas para cada largo.

3.1.1. Vectores de ocurrencias

A partir del análisis de sensibilidad presentado en la sección 2.3 se desprende la importancia del análisis de los vectores de ocurrencia y los valores de sus normas. Utilizando un contador

exacto desarrollado en el software se realizó el cálculo de los vectores de ocurrencia para los distintos largos. Este contador exacto se implementa con los $\approx 4^k$ contadores para los motivos de largo k, garantizando así obtener el vector de ocurrencias $\mathbf{a}^k$ de todos los elementos de largo k de manera exacta sin error.

3.1.1.1. Base de datos Esrrb

El siguiente análisis lo realizaremos a partir del conjunto de prueba de la base de datos Esrrb. En la figura 3.1 se observa la distribución de los 100 k-mers más relevantes para largos de $k = 10$, $k = 15$ y $k = 20$. Como es de esperarse las distribuciones presentan dependencias relevantes al largo. En general para k más pequeños, los elementos más recurrentes presentan más ocurrencias que para k mayores. A pesar de esto es evidente para todos los largos que la gran mayoría de elementos poseen frecuencias muy bajas encontrándose la gran mayoría de elementos solo una o dos veces en el total de secuencias.

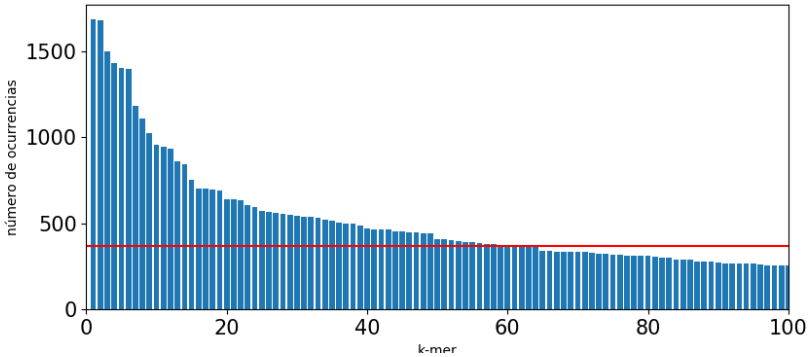
A partir de estos vectores de ocurrencias se obtienen las normas ℓ_1 y ℓ_2 para cada largo. En la tabla 3.2 se presentan los resultados de los resultados de estos cálculos. La tabla incluye el conteo del total de k-mers presentes en la base de datos. Además presenta la cantidad de k-mers diferentes que hay presentes y la cantidad de k-mers que poseen una sola ocurrencia en toda la base de datos.

Se observa una diferencia muy significativa entre ambas normas. La razón evidente de esta diferencia de tanta importancia entre ambas normas es el peso que poseen los elementos con una ocurrencia dentro del vector. Estos elementos tienen un impacto mayor en la norma ℓ_1 al influir sumando su cantidad mientras que para la norma ℓ_2 tienen un impacto menor de aproximadamente la raíz cuadrada de su cantidad.

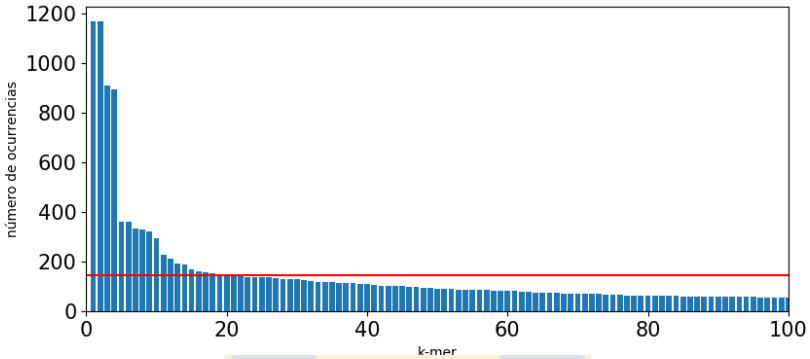
3.1.1.2. Otras bases de datos

En el anexo final de este documento se encuentran los gráficos de las distribuciones de los k-mers más frecuentes para el resto de las bases de datos.

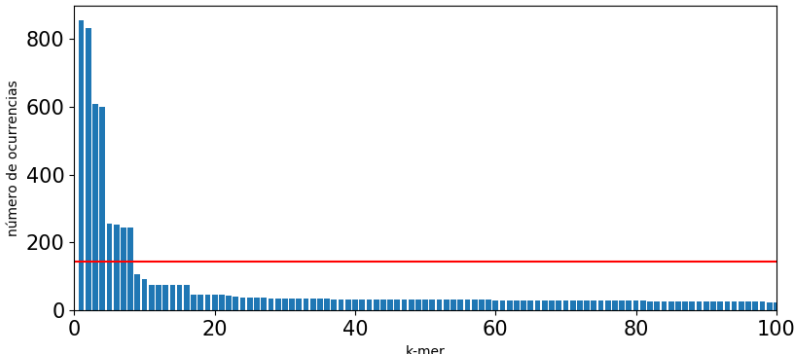
A partir del análisis de las figuras 7.1, 7.2 y 7.3, en comparación con 3.1, se puede comprobar, que a pesar de ser bases de datos de tamaños diferentes, la distribución de los k-mers con mayor frecuencia es similar. Todas las bases de datos poseen distribuciones con heavy hitters, como es de esperarse en secuencias que corresponden a peaks de TFBS.



(a) Esrrb $k = 10$



(b) Esrrb $k = 15$



(c) Esrrb $k = 20$

Fig. 3.1: Distribución de ocurrencias en 100 k-mers más frecuentes en Esrrb. Fuente: Elaboración propia.

Si comparamos los valores obtenidos a partir de los vectores de ocurrencias del resto de las bases de datos, en comparación con la base de datos Esrrb, podemos comprobar que las carácteristicas son similares. En general, los valores obtenidos son proporcionales a la cantidad de secuencias que posee cada base de datos. La base de datos a partir del genoma humano Hepg2 posee una menor predominancia de los elementos con una sola ocurrencia dentro de la base, lo que disminuye la relación entre ambas normas. Sin embargo esto es esperable debido a

Tabla 3.2: Características de la base de datos Esrrb. Fuente: Elaboración propia.

Largo k	Total k-mers	k-mers distintos	k-mers únicos	Norma ℓ_1	Norma ℓ_2
10	4.112.360	816.855	$\approx 24\,\%$	4.134.004	9.781
11	4.090.716	1.849.149	$\approx 52\,\%$	4.112.360	6.705
12	4.069.072	2.864.368	$\approx 76\,\%$	4.090.716	5.092
13	4.047.428	3.463.379	$\approx 89\,\%$	4.069.072	4.197
14	4.025.784	3.730.220	$\approx 95\,\%$	4.047.428	3.676
15	4.004.140	3.837.165	$\approx 97\,\%$	4.025.784	3.353
16	3.982.496	3.875.840	$\approx 98\,\%$	4.004.140	3.124
17	3.960.852	3.885.529	$\approx 98\,\%$	3.982.496	2.950
18	3.939.208	3.882.675	$> 99\,\%$	3.960.852	2.810
19	3.917.564	3.873.912	$> 99\,\%$	3.939.208	2.696
20	3.895.920	3.861.929	$> 99\,\%$	3.917.564	2.598

la mayor cantidad de k-mers presentes en la base de datos.

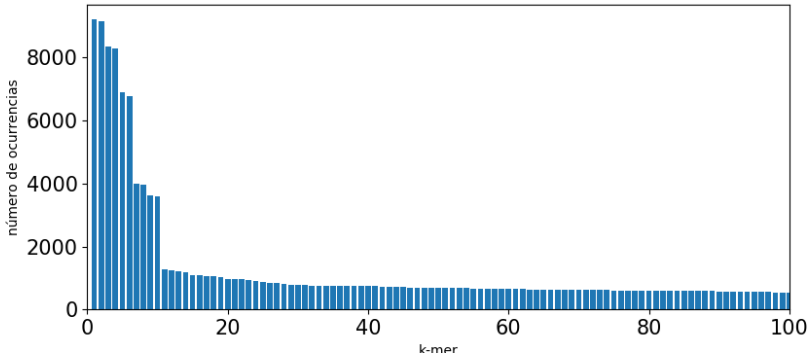
3.1.1.3. Bases de datos de control

Con las bases de datos de prueba con caracteristicas similares, presentamos el análisis de la base de datos de control. En la figura 3.2 observamos la distribución de los 100 k-mers más relevantes para los mismos largos analizados en Esrrb.

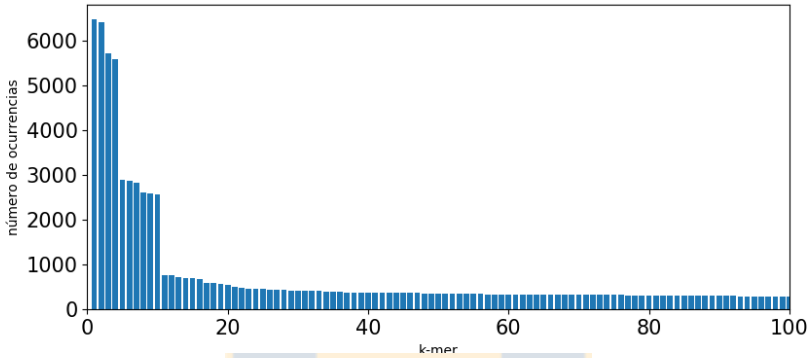
Como se puede observar en las distribuciones, al igual que en las bases de datos de prueba, la base de datos de control presenta sus k-mers distribuídos de manera uniforme, con algunos datos que son sobresalientes por su frecuencia. Como es de esperarse por su tamaño, los datos más frecuentes poseen frecuencias mayores con respecto a los heavy hitters de la base de datos de prueba.

La distribución de los k-mers se explica por la naturaleza de las secuencias utilizadas en la base de datos y la función que cumplen. Al estar fuera de las zonas peaks de los genomas, debería haber menos presencia de k-mers similares repetidos donde se unan las proteínas TF. La función que cumple esta base de datos y el proceso de control en general es filtrar de entre los k-mers heavy hitters aquellos que tienen relevancia en las zonas alejadas de los peaks. De esta manera se pueden eliminar aquellos k-mers que son aleatoriamente más frecuentes, a diferencia de aquellos que efectivamente se relacionan con TF como TFBS.

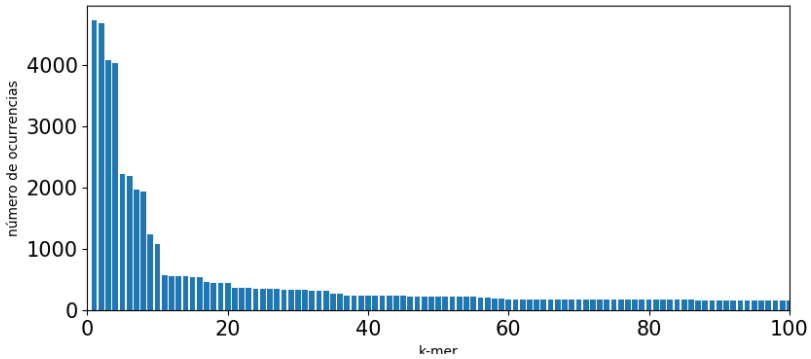
De esta manera, los k-mers heavy hitters en la base de datos de control resultan ser elementos



(a) Control $k = 10$



(b) Control $k = 15$



(c) Control $k = 20$

Fig. 3.2: Distribución de ocurrencias en 100 k-mers más frecuentes en control. Fuente: Elaboración propia.

aleatoriamente muy frecuentes, que en su mayoría son o bases repetidas, o un par de bases intercaladas que se repiten. De esta manera k-mers como $AAAAAAAA$ o $ATATATAT$, por mostrar un par de ejemplos, se encuentran de manera muy frecuente en todo el genoma, y no solo en las áreas peak. Debido a que este tipo de k-mers no son significativos en la zonas peaks de mayor manera que en las zonas de control, no cumplen con el requisito de crecimiento para considerarse motivos emergentes y son eliminados en el proceso de control.

3.1.2. Conclusiones del análisis

En la distribución de los datos, los elementos de una única repetición poseen tal peso que producen una diferencia significativa entre ambas normas. Este peso se va incrementando mientras aumenta el largo de los k-mers, siendo niveles cercanos o superior al 90 %. Esto coincide con la disminución del valor de la norma ℓ_2 a medida que se aumenta el largo k. Al estar los márgenes de error de las estimaciones limitados por las normas ℓ_1 y ℓ_2 para Countmin y CountSketch respectivamente, se requerirán valores de ε muy diferentes para aproximar con la misma precisión. El valor de ε tiene un impacto directo en la relación entre el espacio requerido y sensibilidad de los resultados.

Desde el punto de vista lógico, esto se puede explicar debido a la naturaleza unilateral del error en el Countmin. Estos elementos con una ocurrencia son vistos como ruido en nuestra tarea de buscar los más frecuentes. Este ruido en Countmin es exclusivamente aditivo ya que siempre se incrementan los contadores. En cambio, debido a la segunda función hash, en el CountSketch puede incrementar o decrementar los contadores respectivos. Debido a que estos elementos de ruido son muy dominantes en todo el conjunto de elementos, tiene un impacto muy perjudicial el error unilateral con este tipo de bases de datos.

La introducción de los incrementos condicionales con el proceso de actualización CU en Countmin-CU incorpora un método que permite mejorar el desempeño del algoritmo precisamente en el error de sobre-estimación que posee. La reducción de este factor de error es relevante precisamente por que es un error unilateral; es decir es el único error que posee. En el capítulo de resultados se analizan los resultados experimentales del software desarrollado que permiten comprobar estos planteamientos teóricos.

3.1.3. Frecuencias umbral para motivos emergentes

A partir de la diferencia de los valores en las distribución de los elementos más frecuentes en función del largo de los motivos a buscar, es necesario encontrar frecuencias umbral específicas para cada largo distinto. Estas frecuencias definirán los elementos que serán considerados como frecuentes y los que no. Utilizaremos el método más común [18] para calcular la probabilidad de ocurrencia de motivos.

En primera instancia se buscará encontrar la probabilidad de ocurrencia en una secuencia dada de alguna de las instancias de un motivo de largo k que soporta hasta dm mutaciones en

Tabla 3.3: Frecuencias de umbral para distintos largos. Fuente: Elaboración propia.

Largo k	10	11	12	13	14	15	16	17	18	19	20
E2f1	351	294	245	153	143	138	138	138	138	138	138
Esrrb	368	308	257	161	150	145	145	145	145	145	145
Tcfcpl1	457	382	319	200	186	180	180	180	180	180	180
Ctcf	673	563	470	294	274	265	265	265	265	265	265

sus bases. Esta probabilidad, P_{occ}, se obtiene a partir de la siguiente ecuación de probabilidades

$$P_{occ} = \sum_{i=0}^{dm} \binom{dm}{i} p^i (1-p)^{dm-i} \times \frac{1}{\binom{k}{i} \times 3^i} \tag{3.1}$$

El valor de p definido entre $0 < p < 1$ corresponde a un parámetro que define la variación aceptada del motivo. Un motivo con p menor indica que su conservación es mayor, es decir que la diferencia entre sus ocurrencias son menores. Por el contrario, una conservación menor indica mayores diferencias entre sus ocurrencias y se expresa con un valor de p mayor. Luego se define otro parámetro $0 \leq q \leq 1$, que representa la cantidad de secuencias que contienen instancias de motivos. Este parámetro q generalmente se utiliza para adaptarse a alguno de los modelos de representación utilizados en MEME [47]. De esta manera las frecuencias umbrales se calculan como

$$\rho_f = q \cdot P_{occ} \tag{3.2}$$

Utilizando los valores de $q = 0{,}8$ y $p = 0{,}7$ [47], se calculó la tabla 3.3 donde se muestran las frecuencias de umbral estimadas para cada largo en nuestro rango. Para esto se utilizó la ecuación 3.2 y se definió el umbral como ρ_f multiplicado por la cantidad de k-mers distintos analizados para el largo (proveniente de la tabla 3.2). Además, cabe señalar que la figura 3.1 muestra la ubicación de este umbral dentro de las distribuciones señaladas como la línea roja horizontal.

3.2. Descripción del software

La plataforma en software presenta una implementación de los 3 algoritmos considerados a partir del estudio teórico del estado del arte. Estos algoritmos son CountSketch [28], Countmin [29] y Countmin-CU [32]. Estos tres algoritmos son similares en el sentido de que son algoritmos de búsqueda de elementos más frecuentes en un stream de datos en espacio sublineal y se implementan a través de una estructura de datos específica, denominada sketch. El sketch

se plantea como alternativa a realizar el conteo de manera directa, lo cual requeriría un contador dedicado para cada posible elemento del stream. La desventaja de este conteo directo es que, dependiendo de la aplicación, requeriría una cantidad de contadores exponencial en el largo de los k-mers.

En efecto, si consideramos el problema planteado, con respecto a secuencias de ADN y motivos de entre 10 y 20 bases de longitud, el conteo directo requeriría $\sum_k 4^k, k \in [10, 20]$ contadores. Los sketches representan una alternativa que permite encontrar en espacios mucho menores los elementos más frecuentes a cambio de un costo de confiabilidad de los resultados a través de procesos probabilísticos.

La estructura de datos sketch se asemeja a una matriz de contadores, donde cada fila corresponde al dominio de salida de una función hash, o de dispersión. De esta manera, cada elemento que ingresa al sketch es procesado por una función hash distinta para cada fila del esquema, la cual lo distribuye probabilísticamente sobre alguna las columnas. Se denomina como bucket a cada elemento individual dentro de esta matriz.

Como fue mencionado en la sección 2.2, la principal diferencia entre los esquemas se encuentra en que Countmin Sketch utiliza en cada bucket un acumulador simple, el que se incrementa cada vez que una función hash asigna un elemento al bucket en cuestión. En cambio, CountSketch posee una segunda función hash que, luego de la distribución de la primera, define si el contador correspondiente se incrementa o decrementa en una unidad. Countmin-CU utiliza el mismo modelo que Countmin, con la diferencia que cada bucket posee un acumulador condicional que solamente es aumentado su valor si corresponde al mínimo dentro del conjunto.

3.2.1. Características del software

El software fue desarrollado utilizando el lenguaje C/C++. Es capaz de utilizar como stream de entradas bases de datos biológicas que consistan en secuencias de caracteres, independiente de su largo.

El sketch propiamente tal está almacenado en estructuras C++ std::vector de dos dimensiones. Estas estructuras son arreglos de datos dinámicos, que en este caso almacenan elementos enteros. Se implementaron estos arreglos como una clase que posee las tres funciones básicas de un sketch: inicialización, actualizar contadores de elementos y estimar su frecuencia.

Para las funciones hash se realizó una implementación de la familia de funciones H3 descrita

en la sección 2.4. Estas funciones reciben como entrada un std::string. A partir del string se obtienen los carácteres que lo conforman y se extraen, utilizando máscaras binarias, 2 bits que permiten representar los valores ASCII del alfabeto del ADN de manera única. Con estos bits se realizan las operaciones con una serie de semillas generadas al inicializar el sketch.

Los k-mers detectados como heavy hitters son almacenados en una estructura std::unordered_map. Esta estructura de almacenamiento dinámica es utilizada para almacenar los heavy hitters mientras se van detectando con su respectiva frecuencia. Además permite, durante el proceso de control, comparar los k-mers del stream con los elementos guardados, y eliminar los que corresponda.

El resto de las operaciones del software, como los procesos de entrada y salida, fueron desarrollados con las bibliotecas estándar de C/C++. El software desarrollado puede desarrollar las funciones de:

- Realizar la búsqueda de motivos emergentes en bases de datos de secuencias de ADN.
- Obtener el conteo absoluto de los k-mers presentes en la base de datos.
- Calcular el desempeño de los sketches.
- Medir el tiempo de ejecución de los algoritmos.
- Generar datos de validación para la plataforma hardware.

3.2.2. Desempeño de los algoritmos

Para analizar el desempeño de nuestros sketches, utilizaremos dos métricas: la precisión, y la sensibilidad. En un proceso de recuperación de información se definen estas métricas a partir de las fracciones de los elementos recuperados que resultan relevantes. Si denominamos como VP, FP, FN a los verdaderos positivos, falsos positivos y falsos negativos respectivamente se define formalmente la precisión y la sensibilidad:

$$Precisión = \frac{VP}{VP + FP} \tag{3.3}$$

$$Sensibilidad = \frac{VP}{VP + FN} \tag{3.4}$$

En nuestro caso se utilizarán como referencia los valores obtenidos a partir del conteo absoluto del vector de ocurrencias elaborado para el análisis de la base de datos. Con aquel vector de ocurrencias, y el umbral ρ definido, se determinarán qué elementos corresponderán a los elementos relevantes en nuestra prueba. Luego del vector de estimaciones entregado por las implementaciones de los algoritmos, se utilizará el mismo umbral y se considerarán como positivos aquellos elementos que lo superen. Los elementos de entre los positivos reportados por el sketch que efectivamente se encuentren entre los elementos relevantes serán considerados como verdaderos positivos. El resto de los elementos positivos reportados por el sketch, y que no encuentren dentro de los relevantes corresponden a falsos positivos. De esta manera, la precisión corresponde a la razón entre los verdaderos positivos y la suma entre verdaderos positivos y falsos positivos. La sensibilidad corresponde entonces a la razón entre verdaderos positivos y total de elementos relevantes.

3.2.3. Optimizaciones de la plataforma

Con el fin de poseer una medición de tiempo de ejecución en software que represente un uso adecuado y optimizado a la arquitectura de la CPU a utilizar, se incorporaron las siguientes técnicas de optimización al código:

- Implementación de múltiples sketch para k-mers de distinto largo en múltiples hebras utilizando OpenMP.
- Implementación de las funciones hash paralelos vectorizables con OpenMP SIMD.
- Optimización O3 del compilador GCC.

Capítulo 4: Arquitectura

En esta sección se presenta a la arquitectura diseñada para resolver el problema de la búsqueda de k-mers emergentes a través del uso del algoritmos de sketch. Esta arquitectura fue diseñada como un acelerador hardware y fue implementada en un FPGA Kintex-7 Ultrascale de Xilinx, en la tarjeta de desarrollo KCU1500 de Xilinx. Se presentan los detalles de la arquitectura diseñada y de la integración de la plataforma en el contexto de aceleración hardware.

4.1. Sistema global

4.1.1. FPGA como acelerador hardware

La arquitectura dedicada diseñada para el FPGA se utiliza como un acelerador hardware. El modelo de aceleración hardware significa que el FPGA actúa como hardware dedicado a esta tarea específica trabajando en conjunto con un procesador de propósito general. En este contexto, una CPU en conjunto con un coprocesador con hardware dedicado permite la ejecución de una tarea computacional específica (en nuestro caso, la búsqueda de k-mers emergentes para TFBS en secuencias de ADN) de manera más eficiente que si se ejecutara sólo en software. Este esquema de cómputo requiere de una interfaz de comunicación entre el procesador de propósito general, llamado host, y el acelerador hardware.

La utilización de un procesador de propósito general con un hardware especializado definen una plataforma de computación heterogénea. La ventaja principal de utilizar una plataforma de este tipo sobre una implementación en software es que el software permite traducir el algoritmo a código secuencial que se ejecuta en la arquitectura general fija del procesador. En cambio, una implementación en FPGA consiste en un arquitectura especialmente adaptada para ejecutar el algoritmo. De esta manera, se pierde la posibilidad de ejecutar código de propósito general por una arquitectura específica para este algoritmo, mejorando su tiempo de ejecución, consumo de potencia y uso de recursos. Otra plataforma de aceleración hardware usada tradicionalmente es la GPU. Una GPU, por un lado, es un procesador con una arquitectura fija que ejecuta secuencias de código, pero por el otro lado la arquitectura está diseñada para resolver un clase específica de problemas de manera más eficiente que una CPU.

El modelo de computación heterogénea consiste en la utilización de un host, en conjunto

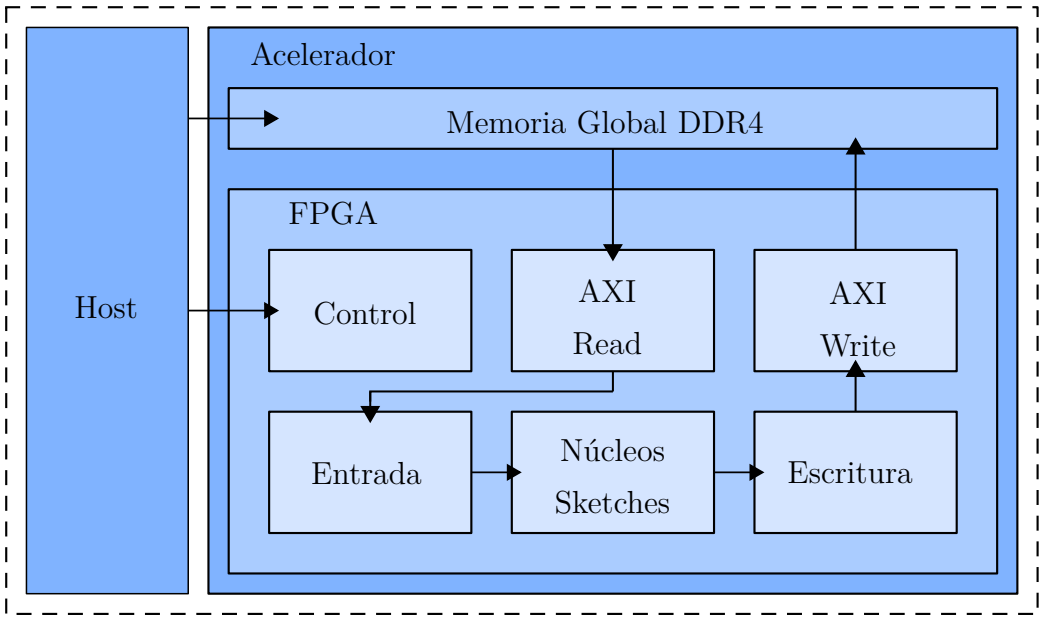


Fig. 4.1: Arquitectura general del sistema. Fuente: Elaboración propia.

con un acelerador hardware. Esto permite transferir código con alta carga computacional del host al acelerador, para de esta manera aprovechar las optimizaciones introducidas por el uso de una arquitectura dedicada. Este modelo heterogéneo requiere que la CPU sea la encargada de controlar el acelerador desde su espacio de usuario y sea capaz de controlar las transferencia de datos necesaria para ejecutar secciones del algoritmo en el acelerador.

4.1.2. Arquitectura general del sistema

En nuestro caso específico, la tarjeta de desarrollo KCU1500 de Xilinx se integra utilizando como canal físico de comunicación un bus PCIexpress de 3ra generación (abreviado como PCIe). En cuanto a la interacción entre el software y el hardware dedicado, se utiliza una interfaz en el lenguaje de programación C++/OpenCL utilizando el ambiente de desarrollo SDAccel de Xilinx y su integración con las herramientas tradicionales de diseño, síntesis lógica e implementación en FPGA de la suite Vivado.

La arquitectura global del acelerador hardware consta de 3 partes principales:

- Interfaz y protocolo de comunicación con OpenCL a través de PCIe.
- Núcleos de los sketches.
- Hardware adicional de comunicación entre la interfaz PCIe y los núcleos de los sketches.

El núcleo del sketch consiste de la arquitectura dedicada diseñada e implementada para la resolución del problema propuesto a través del algoritmo Countmin-CU Sketch. Esta implementación está realizada a nivel nivel de transferencia de registros (del inglés *Register Transfer Level*, RTL) y escrita en el lenguaje de descripción de hardware (del inglés *Hardware Description Language*, HDL) SystemVerilog. Para realizar la búsqueda de k-mers emergentes, el núcleo del sketch explota el paralelismo propio del algoritmo de sketch, trabajando en paralelo con cada una de las filas del sketch, incluyendo el cálculo de las funciones hash de dispersión y los procesos de actualización en memoria.

El proceso de comunicación entre el host y el FPGA, incluyendo el control del acelerador y el traspaso de datos y memoria, se realiza a través de un puerto PCIe. Para la programación de esta interfaz, tanto del control desde el host, como de los controladores y protocolos de comunicación en hardware, fue utilizado la plataforma de desarrollo de Xilinx SDAccel. Este ambiente de desarrollo permite controlar el acelerador hardware desde un modelo de programación de alto nivel a través de funciones de OpenCL. Desde el punto de vista del hardware implementa un protocolo de acceso directo a memoria (del inglés *Direct Memory Access*, DMA) sobre PCIe.

Aparte de los dos componentes mencionados se desarrolló en hardware una serie de módulos adicionales que permiten ajustar las interfaces de entrada y salida de la comunicación FPGA/host para adecuarla al núcleo del sketch y recibir los datos como un stream.

La figura 4.1 muestra la arquitectura general del sistema. En ella se puede observar el host y el acelerador hardware compuesto por su sistema de memoria DDR y el chip FPGA. Dentro del FPGA se encuentran los módulos de comunicación AXI para acceder a los espacios de memoria global, los núcleos de los sketches, y la lógica necesaria para conectar estos núcleos con las interfaces de comunicación.

4.1.3. Arquitectura general del algoritmo

La entrada al sistema corresponde a los streams de datos correspondientes a las bases de datos. El problema de búsqueda de motivos emergentes utiliza el algoritmo de Countmin-CU sketch para encontrar los elementos más frecuentes, o heavy hitters, en el stream de datos de prueba. Estos k-mers heavy hitters pueden ser analizados para distintos largos de k. Tal y como fue planteado en la plataforma desarrollada en software, el proceso de búsqueda de los heavy hitters depende de este largo k. Esto quiere decir que se debe analizar el stream para cada largo distinto usando un sketch distinto. Para explotar el paralelismo en la adquisición de datos, se

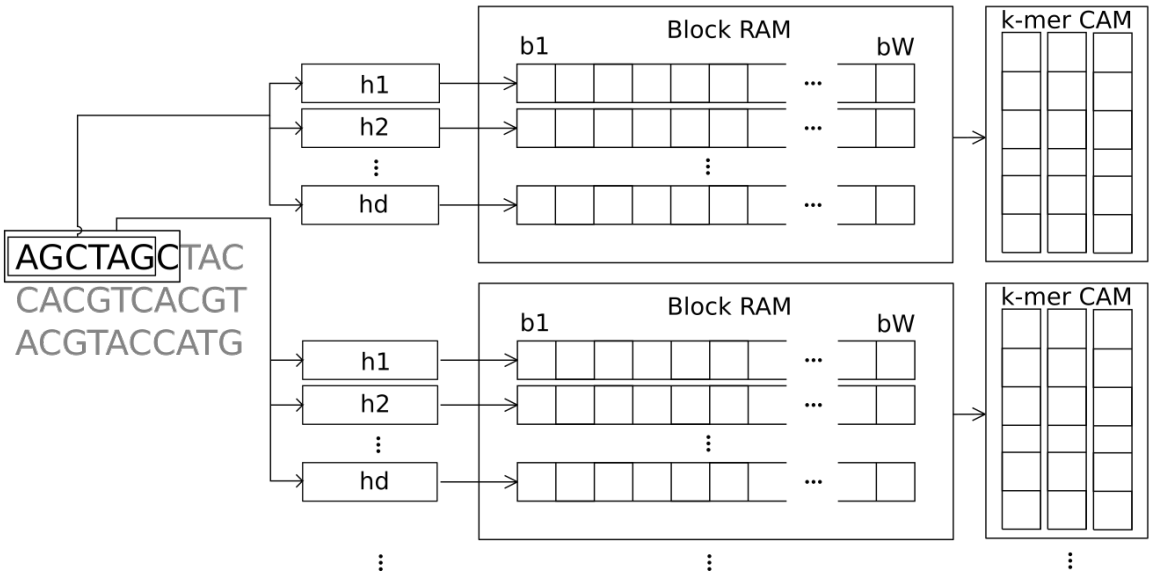


Fig. 4.2: Arquitectura propuesta para el sketch. Fuente: Elaboración propia.

utilizaron múltiples sketches en paralelo. De esta manera, cada sketch distinto podrá realizar de manera paralela e independiente con el resto el conteo para k-mers de un largo específico. Con un sketch por largo se puede explotar de manera efectiva el paralelismo en la adquisición de datos. De la misma manera, debido al funcionamiento de los sketches, poseer más de un sketch por largo requeriría lógicas complejas de coherencia de datos y verificación de las estimaciones de frecuencias que impedirían poder paralelizar múltiples sketches con los mismos elementos de entrada. La figura 4.2 muestra el detalle da la arquitectura propuesta para el algoritmo. En ella se observa que cada largo distinto de k-mer posee su propio sketch, con sus funciones hash y memoria para almacenar k-mers.

De esta manera, la arquitectura contiene para cada largo distinto de k el hardware dedicado a la implementación del Countmin-CU sketch y un sistema de memoria para el almacenamiento de los heavy hitters. La implementación de múltiples sketch permite operar de manera paralela sobre los mismos datos de entrada, solo considerando los largos distintos, sin introducir la necesidad de trabajar con múltiples pasadas del stream. Para todos los distintos largos, durante esta pasada del stream se insertan los elementos en el sketch y se detectan los elementos heavy hitters. Estos heavy hitters son entonces almacenados en un sistema de memoria para su almacenamiento. Cada sketch en paralelo tiene su respectiva memoria para los heavy hitters de su largo específico. El acceso a estas memorias se realiza en base a su contenido, los k-mers, por eso nos referiremos a ellas como memoria accesada por contenido (del inglés *Content Addressable Memory*, CAM).

El proceso de control se realiza una vez que se ha completado el análisis de la base de datos

de prueba y se detectaron y almacenaron los heavy hitters. Este proceso involucra el análisis del stream de datos de control, y, de la misma manera, realizar las operaciones en paralelo para distinto largos trabajando con las unidades de almacenamiento de heavy hitters. La salida del algoritmo corresponde a los k-mers emergentes encontrados luego de ambos procesos para todos los largos de k analizados.

4.1.4. Herramienta de automatización de Python

Para tener flexibilidad en las implementaciones de los sketches con respecto a los parámetros de las implementaciones, se diseñó una herramienta que automatiza la generación código en SystemVerilog. Esta herramienta permite, de esta manera, generar los códigos fuentes de distintos sketches con distintos parámetros. Utilizando el lenguaje script Python, la herramienta posibilita la generación automática de código SystemVerilog ajustado a los parámetros requeridos a través de scripts sencillos de modificar. Estos scripts son los encargados de recibir los parámetros y con ellos ajustar el código RTL escrito.

Los parámetros que permite modificar estas herramienta son los siguientes:

Sketch: Permite seleccionar entre CountSketch, Countmin y Countmin-CU sketch.

Largo k: Permite seleccionar el largo k de bases de los elementos de entrada.

Filas d: Permite seleccionar la cantidad d de filas del sketch.

Columnas W: Permite seleccionar el ancho W de cada fila.

Bits por bucket: Permite seleccionar la cantidad de bits de cada contador del sketch.

Límite rho: Permite seleccionar el umbral de heavy-hitters.

Bits por bases: Permite modificar la cantidad de bits necesarios para representar una base.

Tamaño CAM: Permite definir el tamaño de una memoria CAM.

Conjuntos CAM: Permite definir la cantidad de conjuntos que posee una memoria CAM.

El uso de esta herramienta no solo permite modificar los parámetros de los sketches, sino que además permite que la implementación se ajuste especialmente a estos parámetros implementando un mecanismo de reconfiguración estática. Esto quiere decir que, por ejemplo, si se

implementan múltiples sketches de distintos largos, cada sketch estará construido con las dimensiones óptimas para su largo específico, y no tendremos que usar generalizaciones para que el mismo hardware se utilice para todos los largos.

Esta herramienta permite modificar los parámetros del núcleo del sketch diseñado con la arquitectura dedicada, pero se encuentra limitada al realizar la integración con la plataforma de aceleración de SDAccel. Los módulos que implementan la lógica necesaria para utilizar nuestra arquitectura con el esquema de aceleración no se encuentran incluidos dentro de lo que permite generar estos scripts, por lo que los módulos de interfaz deben ser programados manualmente.

4.2. Implementación del modelo de computación

La utilización de un procesador de propósito general en conjunto con un acelerador hardware dedicado define un modelo de cómputo heterogéneo. Desde el espacio de usuario brindado por el procesador host y su sistema operativo, es posible la ejecución de software que utiliza el FPGA para realizar aceleración hardware dedicada a ciertas tareas específicas. En este trabajo se utilizó el ambiente de desarrollo de Xilinx SDAccel para la integración de la tarjeta de desarrollo KCU1500 como una plataforma de aceleración hardware.

La herramienta SDAccel utiliza el lenguaje OpenCL como estándar de programación para el modelo de computación heterogénea entre CPU y FPGA. A través de estos estándares se crea un modelo de programación que permite transferir datos entre la CPU host y los dispositivos de aceleración hardware. En este contexto de computación heterogénea la CPU host es la encargada de controlar la plataforma de aceleración, asignar buffers para transferir datos entre la memoria propia y la global, y definir la carga computacional del FPGA.

La funcionalidad del ambiente de desarrollo SDAccel se resumen en la figura 4.3. SDAccel permite acelerar, a través de la utilización de un lenguaje de alto nivel, una gran variedad de algoritmos a partir de bibliotecas de hardware prediseñadas para este propósito. Esto permite volver accesible a lenguajes de alto nivel la aceleración que brindan los FPGA, escondiendo los requisitos de altos tiempos de desarrollo. Además de la posibilidad de utilizar los aceleradores de las bibliotecas para la aceleración directa de códigos C++/OpenCL, SDAccel permite integrar hardware personalizado para utilizar como acelerador. Por lo tanto SDAccel también puede ser utilizado como un puente para integrar hardware dedicado diseñado a nivel de RTL en la ejecución de código C/C++ utilizando directivas de OpenCL.

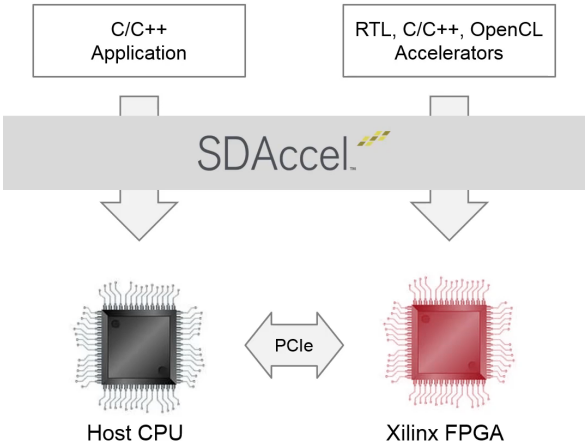


Fig. 4.3: Esquema de funcionalidad de SDAccel. Fuente: Documentación SDAccel Xilinx [48]

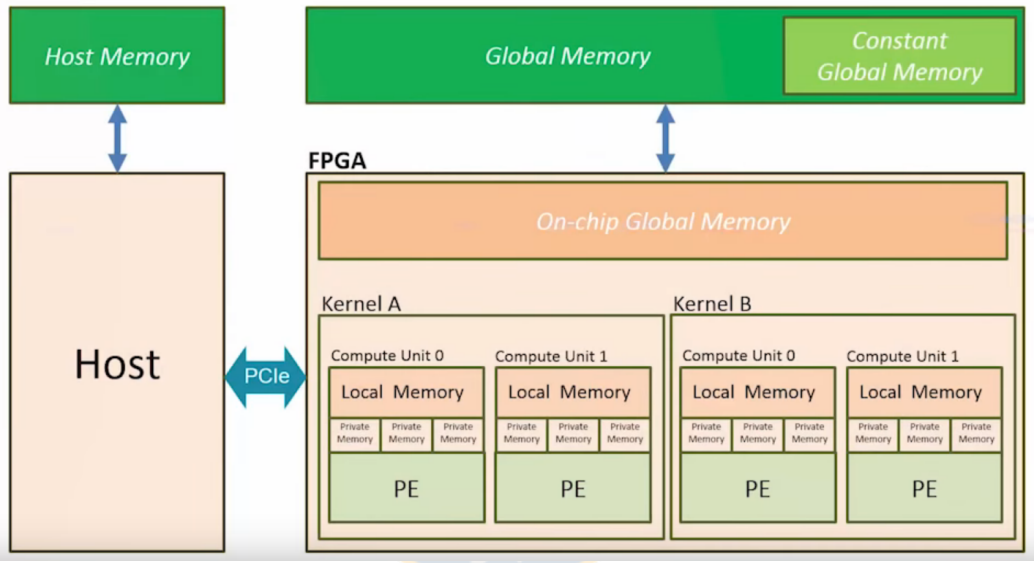


Fig. 4.4: Esquema de memoria para computación heterogénea. Fuente: Documentación SDAccel Xilinx [49]

4.2.1. Esquemas de memoria

El modelo de computación heterogénea implementado por la plataforma de aceleración en conjunto con OpenCL utiliza un sistema de memorias definido que permite compartir datos entre el host y el acelerador. La figura 4.4 representa este esquema de memoria. Este modelo contiene espacios de memoria presentes tanto en el host como en el FPGA, con la conexión física entre ambos a través de un puerto PCIe. Además de las memorias compartidas, el esquema incluye distintas unidades funcionales, llamados kernels, en el FPGA acelerador con memorias privadas.

La memoria del host, *host memory* en el diagrama, es el sistema de memoria que posee la CPU host. Esta está compuesta por toda la memoria que posea la CPU en todas sus jerarquías.

Este espacio de memoria es privado para el host, y es desde donde lee o hacia donde escribe datos en las transferencias con el FPGA. De los espacios de memoria de la plataforma de aceleración, el de nivel más alto de la jerarquía corresponde a la memoria global (*global memory*). Este espacio de memoria está implementado en memoria externa al chip del FPGA en sistemas de memoria de acceso aleatorio (del inglés *Random Access Memory*, RAM) presentes en las plataformas de aceleración. En nuestro caso, la tarjeta KCU1500 posee 16GB de memoria DRAM DDR4 que son utilizados como memoria global. Este es un espacio de memoria donde tanto el host como el FPGA tienen acceso para lectura y escritura. El host puede escribir y leer buffers entre su memoria privada y la memoria global a través del puerto PCIe. Los kernels pueden acceder a ella a través de los pines de entrada y salida del chip FPGA. El modelo de programación permite incluir dentro de la memoria global un espacio dedicado para almacenamiento de constantes (denominado *constant global memory*). Este espacio de memoria es un subespacio dentro de la memoria global donde únicamente el host tiene permisos de escritura. De esta manera el FPGA solo puede leer los datos ahí almacenados.

El esquema de referencia del uso de memorias incluye dentro de las memorias del FPGA más bajo en la jerarquía una memoria global dentro del FPGA (*On-chip global memory*) y memorias locales para cada unidad de procesamiento (*Local memory*). La memoria global dentro del chip se implementa en memorias blockRAM y permiten compartir datos entre los distintos kernels de procesamiento instanciados en el FPGA. Además, se utilizan para la construcción de buffers para la lectura de datos desde la memoria global en DRAM. De esta manera, se establece una jerarquía de memoria en el FPGA, que al igual que las jerarquías estándar de una CPU, permiten tener memorias con latencias más bajas cercanas a las unidades de procesamiento, pero con menor capacidad de almacenamiento.

Nuestro uso de la plataforma de aceleración no se realizó a través del uso de las bibliotecas de kernels de aceleración de Xilinx para código OpenCL. Al utilizar un solo kernel propio, la utilización de los recursos de memoria fue personalizada para ajustarse de mejor manera al algoritmo. Sin embargo el modelo de memoria planteado se mantiene para las jerarquías altas y el canal PCIe de comunicación.

4.2.2. Kernel RTL

Para poder tener un control fino de la arquitectura, el uso de relojes, y poder explotar el paralelismo inherente del algoritmo, realizamos nuestra plataforma de aceleración utilizando una arquitectura propia. SDAccel permite, a través de los denominados Kernel RTL, integrar

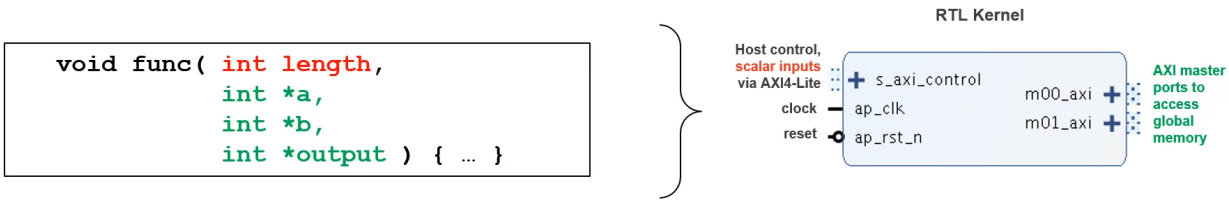


Fig. 4.5: Representación de las interfaces del kernel RTL. Fuente: Elaboración propia.

diseños de hardware a nivel RTL a partir de lenguajes HDL a los esquemas de control del acelerador y transferencia de memoria propias del modelo de programación con OpenCL. De esta manera, se permite integrar dentro de un código en OpenCL la ejecución de un núcleo propio de hardware diseñado a nivel RTL. Esto permite la utilización de lógica personalizada para realizar los procesos de aceleración.

La integración de diseños propios en códigos OpenCL se realiza, desde el punto de vista del software, a través del equivalente de un llamado a una función. En esta abstracción, el kernel RTL se representa como el llamado a una función sin valor de retorno. Desde este punto de vista, la función puede recibir como argumentos punteros a memoria, además de argumentos escalares fijos de configuración. La aplicación en OpenCL se encarga de la configuración del FPGA, del traspaso de información a las memorias globales, de la ejecución de los kernels, y de volver a dejar disponible el FPGA al terminar su uso.

Desde el punto de vista del hardware, el kernel cuenta con dos tipos canales AXI de comunicación: Un controlador AXI-Lite esclavo encargado de los procesos de control del acelerador, y canales AXI-MM para el intercambio de datos de memoria. A través del canal AXI-Lite esclavo, la CPU host puede iniciar la ejecución del kernel, monitorear su estado, entregar las direcciones de los punteros a la memoria global, y opcionalmente escribir argumentos escalares. Aparte del único AXI-Lite para la comunicación de control, el kernel debe contar con entre 1 y 16 canales AXI-4 maestros del tipo MM (*Memory mapped*) para la comunicación con el espacio global de memoria y el kernel. Cada interfaz maestra AXI-4 MM posee direcciones de memoria de 64 bits, cuya dirección base en la memoria global es brindada por el host a través la interfaz AXI-Lite esclavo. A parte de los canales de comunicación en el FPGA, se implementan también el controlador del bus PCIe y los sistemas de control del kernel acelerador. El sistema de control es el encargado de manejar las señales que indican el comienzo de la ejecución del kernel y de generar las señales que confirman su final.

El esquema de la figura 4.5 representa las dos partes de esta interfaz. Por un lado, desde el punto de vista del software, está la visión del kernel como un llamado a una función sin retorno, la cual recibe los punteros a memoria que corresponden a los buffers de lectura y escritura.

Por el lado del hardware, está el kernel RTL empaquetado como un IP core con los canales de comunicación descritos como entradas y salidas.

Además de los canales AXI de comunicación, un kernel RTL debe contar con por lo menos una señal de reloj y de reinicio. El reloj primario es usado para manejar los canales AXI a 300MHz, y posee una señal de reinicio activa bajo. Se puede incorporar un segundo dominio de reloj en el caso de que el kernel deba funcionar a una velocidad distinta que los canales AXI. Si se utiliza un segundo reloj, se debe incorporar una segunda señal de reinicio, activo baja, en el dominio de este segundo reloj.

La integración del Kernel RTL con el código en OpenCL se realizó a través de una herramienta disponible en SDAccel llamada RTL Kernel Wizard. Este es un wizard ejecutado a través de una interfaz gráfica cuyo objetivo es brindar un método directo de empaquetar IPs RTL propias como un kernel RTL integrado a SDAccel. El wizard permite crear de manera automática a nivel de RTL módulos hardware que contienen la interfaz AXI-Lite esclavo incluyendo toda la lógica de los registros de control, un kernel IP de ejemplo para ser reemplazado con el kernel propio, las interfaces de comunicación AXI4-MM maestras. Además el wizard crea un archivo de proyecto de Vivado que integra toda la jerarquía de módulos generados y provee herramientas de prueba y simulación para el kernel de ejemplo. Finalmente el wizard genera un código de ejemplo en C/C++ OpenCL para ejecutar el kernel RTL generado.

4.3. Arquitectura del núcleo del sketch

La parte central de nuestro diseño hardware corresponde al núcleo diseñado para la implementación del algoritmo Countmin-CU Sketch. Si bien nuestro sistema explota el paralelismo en la adquisición de datos al utilizar múltiples núcleos de sketch para los múltiples largos a analizar, cada uno de estos núcleos de manera interna también esta diseñado para explotar el paralelismo propio del algoritmo.

En primer lugar, se propone que cada fila del sketch sea trabajada en paralelo. Debido a que cada una de las filas del sketch se puede manejar de manera independiente del resto, se ajustó el hardware para que trabaje con éstas en paralelo. En concreto, cada fila requiere poseer una función hash asociada a ésta, su espacio independiente de almacenamiento, y su lógica de actualización. En segundo lugar, cada una de estas unidades de funcionamiento se implementa como un pipeline profundo para maximizar la frecuencia del reloj del sistema. De esta manera, cada elemento nuevo que ingrese al sketch será distribuido hacia cada pipeline correspondiente

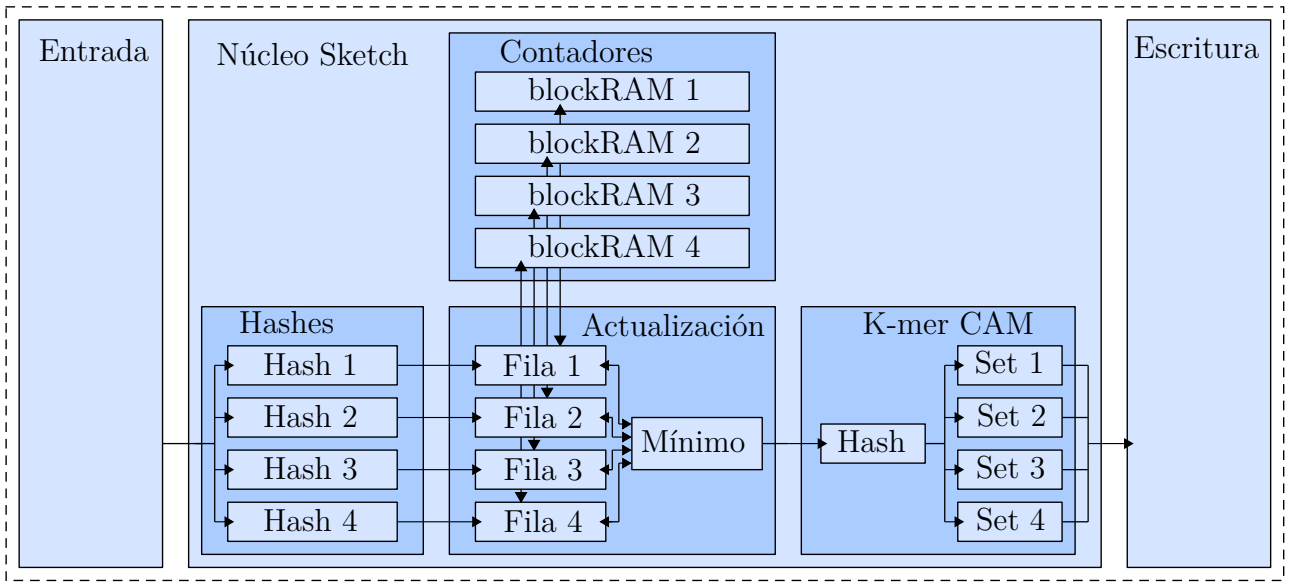


Fig. 4.6: Arquitectura de un núcleo de sketch. Fuente: Elaboración propia.

a cada una de las d filas presentes en el sketch.

Un esquema de la arquitectura básica de un núcleo de sketch se presenta en la figura 4.6. La arquitectura utiliza memoria blockRAM interna del chip FPGA para implementar el almacenamiento que contiene la matriz de contadores del sketch. Éstas fueron generadas utilizando el IP de Xilinx Block Memory Generator y son instanciadas individualmente por filas para tener acceso paralelo a ellas. Cada línea de procesamiento contiene una función hash de la familia H3 y una unidad lógica de actualización de la filas, ambas implementadas en pipelines profundos. Un módulo global a todas las filas del sketch es el encargado de computar la frecuencia estimada. Esta estimación para el algoritmo Countmin-CU se realiza calculando el mínimo valor de entre todos los contadores apuntados por las funciones hash. Aparte del cálculo de la frecuencia estimada, se utilizará un comparador para determinar si cada elemento corresponde o no a un heavy hitter. Adicionalmente se diseño un sistema de memoria para almacenar los elementos identificados como heavy hitters.

De esta manera cada núcleo de sketch se compone de los siguientes módulos de procesamientos:

- d funciones hash de la familia H3 implementadas en pipelines paralelos.
- d unidades de actualización del sketch en pipelines paralelos.
- d filas de almacenamiento de W contadores implementadas en blockRAM.
- Un módulo encargado de calcular el mínimo entre los contadores seleccionados.

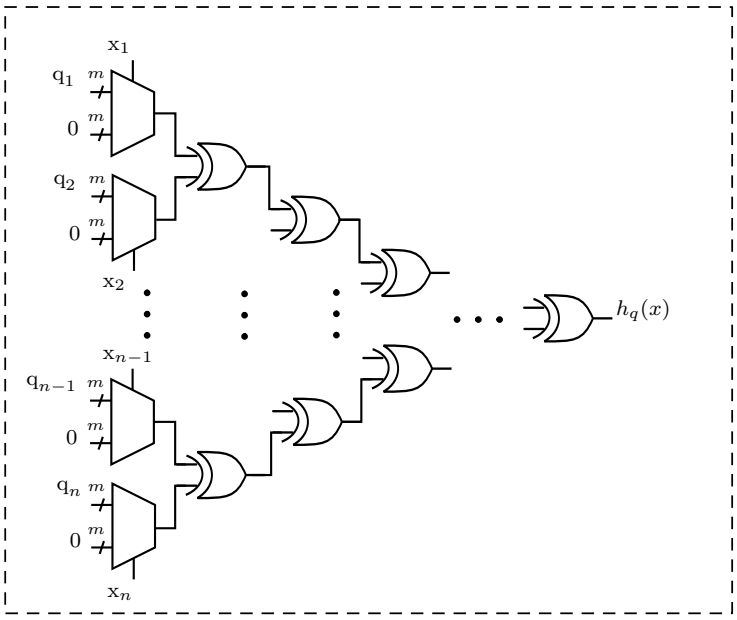


Fig. 4.7: Esquema de implementación de la función H3 en hardware. Fuente: Elaboración propia.

- Un sistema de memoria diseñado para almacenar los heavy hitters.
- Un módulo top que contenga las unidades funcionales, realice la comparación con el umbral de heavy hitters, y posea registros de desplazamientos del largo de los pipeline de linea.

En el resto de esta sección explicaremos en detalle cada uno de estos componentes.

4.3.1. Funciones hash

Como fue definido en la sección 2.4, las funciones hash que utiliza la implementación en hardware corresponden a la familia de funciones H3. En la ecuación 2.21 se define la operatoria matemática necesaria para el cálculo de su valor. En nuestro sistema, cada una de las d filas del sketch implementan una función H3, con distinto juegos de semillas. Las semillas corresponden a números aleatorios con valores entre 0 y W. Los elementos de entrada a la función hash son denominados llaves. Cada función hash contiene una semilla de $\lceil log_2 W \rceil$ bits por cada bit que poseen sus llaves de entrada.

A partir de la ecuación, se observa que ésta consiste en una serie de operaciones lógicas AND entre un bit de la llave de entrada y su semilla asociada. Estas operaciones AND son independiente entre ellas. El valor de la función es la aplicación de la operación lógica XOR entre cada uno de estos resultados. Por lo tanto, la función se puede calcular como el AND entre cada bit de la llave y su semilla correspondiente en paralelo, seguido por un árbol de reducción

binario que ejecuta los XOR bit a bit. Este esquema permite explotar el paralelismo presente en la función e implementarlo en un pipeline profundo para que no se generen limitaciones de tiempo dentro de este módulo. Debido a que las operaciones AND de la primera etapa son entre un sólo bit y un arreglo de bits, se implementa a través de multiplexores de dos entradas. En los multiplexores el bit de la llave se utiliza para seleccionar la salida entre la semilla original o un vector de ceros del mismo largo.

La arquitectura de esta implementación se puede observar en la figura 4.7. La figura no muestra los registros de pipeline, que se encuentran presente entre cada etapa XOR. Si definimos las llaves con un largo de n bits, la arquitectura utiliza n multiplexores y el pipeline tiene una latencia total de $\lceil log_2 n \rceil + 1$ ciclos. Para la implementación particular con $W = 16384$ y para largo $k = 15$ tendríamos llaves de 30 bits, con 30 semillas de 14 bits cada una. Con estas dimensiones se requeriría un total de 6 etapas de pipeline para obtener la salida de 14 bits.

Estas operaciones lógicas a nivel de bit son simples de implementar en hardware y no requiere unidades aritméticas complejas (como lo serían sumadores o multiplicadores). Como fue mencionado anteriormente, la posibilidad de realizar una implementación óptima fue el criterio preferido para seleccionar la familia H3 para ser utilizada. Debido a que el sistema completo con sketches en paralelo requiere una cantidad considerable de funciones hash en paralelo, el bajo uso de recursos lógicos de la implementación se vuelve relevante. Una implementación de 11 sketches de dimensiones $d \times W$ requiere un total de $d \times 11$ funciones hash, para tamaños de llaves que varían entre 20 y 40 bits.

La generación del código HDL para la implementación de las funciones hash para diferentes parámetros en nuestro flujo de trabajo se encuentra automatizado a través de los scripts en Python ya mencionados.

4.3.2. Estimación de frecuencia

Para realizar la estimación de la frecuencia de un elemento visto en el stream de datos el algoritmo Countmin-CU, al igual que Countmin, utiliza el valor mínimo de los contadores del sketch asociados a dicho elemento. Esta operación se realiza sobre una cantidad de valores igual a la cantidad de filas.

En nuestra arquitectura se utilizan redes de ordenamiento, del inglés *sorting network*, para calcular el mínimo (o la mediana para el caso de CountSketch). Estas redes consisten en unidades que comparan dos elementos y los ordenan dependiendo de cuál es menor.

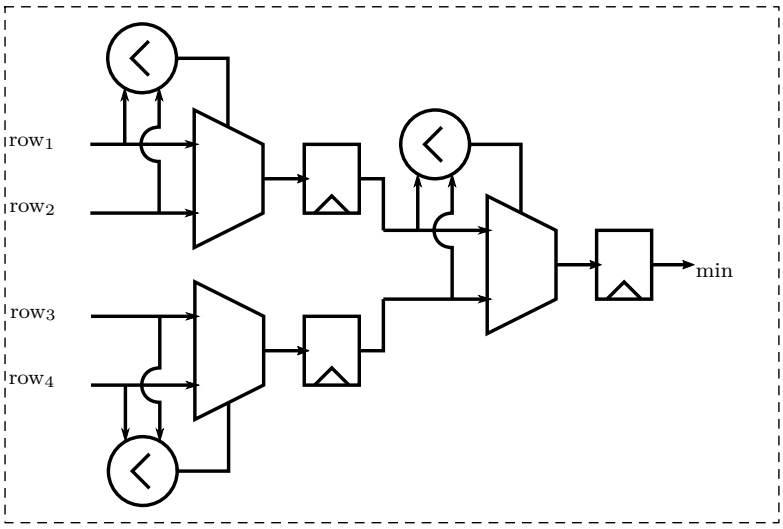


Fig. 4.8: Red de ordenamiento para el cálculo del mínimo para 4 entradas. Fuente: Elaboración propia.

En nuestro caso, como buscamos el mínimo, la red de ordenamiento selecciona de entre dos elementos el menor, descartando el mayor. De esta manera, formamos una red binaria que en cada etapa reduce a la mitad la cantidad de elementos y almacena los menores. Luego de $\lceil log_2 d \rceil$ etapas se obtiene el mínimo global.

La figura 4.8 muestra el circuito que implementa esta operación para un valor de $d = 4$. Cada sketch posee uno de estos circuitos que reciben las entradas de cada una de sus filas. Con estas dimensiones, se utilizan tres unidades de ordenamiento, cada una compuesta por un comparador y un multiplexor. La red de ordenamiento está implementada en un pipeline con el objetivo de trabajar con frecuencias de reloj altas.

Al igual que las funciones hash, la generación del código de SystemVerilog para este módulo esta incluido en los manejados automáticamente por los scripts generadores.

4.3.3. Actualización de filas del sketch

Las operaciones sobre el sketch se realizan en cada fila de manera independiente. En nuestra arquitectura, el proceso de ingreso de un nuevo elemento al sketch y la estimación de su frecuencia se ejecutan en un mismo pipeline. La lógica requerida para estos procesos se asocia a una línea de la matriz del sketch. Esta línea está almacenada en memoria blockRAM dentro del chip del FPGA. Los bloques de blockRAM correspondientes fueron configurados utilizando el IP Core Block Memory Generator de Xilinx. Estas memorias fueron configurados como memorias RAM de puerta dual simple, una puerta de lectura, y otra de escritura, independientes entre

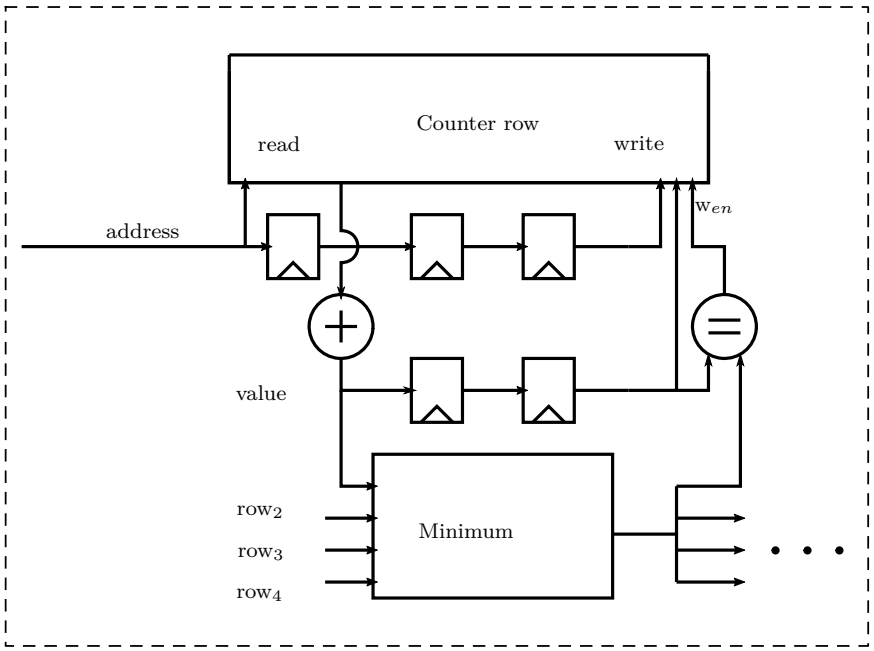


Fig. 4.9: Lógica de actualización de filas del sketch. Fuente: Elaboración propia.

ellas. Las puertas tienen una latencia de un ciclo de reloj. Esto quiere decir que al solicitar la información de una dirección específica a la puerta de lectura, en el siguiente ciclo se obtiene dicho dato.

Considerando los algoritmos de la sección 2.2, los procesos de actualización de los distintos algoritmos son simples aritméticamente. A diferencia de los algoritmos de CountSketch y Countmin Sketch, en quienes para cada fila siempre se actualiza el valor obtenido, Countmin-CU Sketch realiza un incremento condicional. Esto significa que solamente cambia los valores en aquellas filas cuyos valores correspondan al mínimo de entre todas las filas. Esto implica la necesidad de realizar el cálculo del mínimo dentro de la lógica de actualización de la fila, introduciendo una dependencia de datos entre el resultado del mínimo y la escritura en blockRAM del valor actualizado. La figura 4.9 esquematiza la lógica de actualización de línea utilizada para el algoritmo Countmin-CU. Este hardware es responsable de determinar si el contador de la fila asociada debe ser aumentado, en cuyo caso escribe en la misma memoria el valor actualizado. La figura muestra la arquitectura de actualización de una sola fila, sin embargo, el módulo de cálculo del mínimo es un circuito global que recibe sus entradas de cada una de las d filas del sketch.

El módulo de actualización de fila recibe la salida h_q proveniente de su función hash asociada. Este valor corresponde a la dirección con la que se accede a la memoria blockRAM donde está almacenada la fila completa. Al recibir el dato al ciclo siguiente, éste ingresa al pipeline del módulo del mínimo. Como es necesario esperar $\lceil log_2 d \rceil$ ciclos de latencia, tanto el valor

incrementado como la dirección de memoria de donde fue leído son introducidos a registros de desplazamiento. Luego de obtener el valor del mínimo, éste es comparado con el valor almacenado. Esta comparación controla la señal de habilitación de escritura en la memoria blockRAM. Si el mínimo es igual al valor almacenado, este valor es escrito en la dirección dada por los registros de desplazamiento. Si el valor almacenado y el mínimo no son iguales, no se realiza la escritura en la memoria. Los ciclos de latencia introducen una dependencia de datos entre el cálculo del mínimo y la escritura, por lo que se debe aplazar el proceso de escritura como fue descrito. Este aplazamiento temporal de la escritura puede introducir errores de coherencia de datos en el caso en que elementos a una distancia igual o menor que la latencia del mínimo sean distribuidos por las funciones hash a una misma dirección de memoria. Según fue cuantificado experimentalmente en la sección 5.1.1.2, este error resulta despreciable para la búsqueda de k-mers emergentes.

En total el proceso de actualización de fila del sketch toma $\lceil log_2 d \rceil + 1$ ciclos de latencia. En cuanto a su uso de recursos, tiene relevancia la cantidad de bloques de memoria blockRAM que utiliza. La cantidad de bits utilizados por los contadores tiene un impacto en el uso de memoria. El tamaño de los contadores, si bien es un parámetro dentro de lo que se pueden modificar con los scripts de generación de HDL, éstos deben ser configurados dentro de la herramienta del IP utilizando las herramientas de Vivado. La definición del tamaño de los contadores depende principalmente de la base de datos. En nuestro caso, se definieron de un tamaño de 11 bits, a pesar de necesitar estrictamente 9 para nuestra base de datos, como manera de ampliar la implementación a otras bases de datos. Con este parámetro, en total se necesitan d filas de blockRAM, con puertas de 11 bits de ancho y direcciones de $\lceil log_2 W \rceil$ bits de ancho. Esto quiere decir, direcciones de 14 bits para $W = 16384$. Una fila de este tamaño requiere $176kb$ de almacenamiento que por la herramienta son distribuidos en 5 bloques de 36kb de blockRAM.

4.3.4. Detección de los heavy hitters

Para cada elemento de entrada desde el stream de datos, el algoritmo Countmin-CU estima su frecuencia como el mínimo de sus d contadores asociados. Nuestra arquitectura utiliza el módulo de cálculo del mínimo descrito en esta sección e incluido dentro del pipeline de la actualización de filas. Debido a que el mínimo se calcula con el valor leído más su incremento en uno, la salida del módulo del mínimo corresponde a la estimación de frecuencia actualizada del elemento en el sketch.

Fuera del lazo central del pipeline de la actualización de línea, la estimación de frecuencia es

comparada con una frecuencia umbral ρ_f. Este umbral de frecuencia es establecido en el proceso de generación automatizada de los códigos HDL y es definido como un parámetro. En la sección 3.1.3 se detalló el trasfondo biológico de los umbrales definidos en nuestra implementación, que fueron posteriormente validados con los programas en software.

Los elementos cuya estimación de frecuencia superan dicho umbral son los seleccionados como heavy hitters. Para su posterior procesamiento en el proceso de control, es necesario preservar tanto el k-mer heavy hitter, como su frecuencia estimada. Debido a que el k-mer proviene de un stream de datos que no se volverá a leer, es necesario almacenar los k-mer que se encuentran actualmente dentro del flujo de procesamiento. Esto se realiza en registros de desplazamientos del largo total de la concatenación de los pipeline de la función hash y la actualización de fila del sketch. Dichos registros de desplazamiento almacenan una cantidad de bits que dependen del largo k en cada sketch. Por lo tanto, almacenan $2 \times k$ bits y poseen un largo de $(\lceil log_2(2 \times k) \rceil + 1) + (\lceil log_2 d \rceil + 1)$ ciclos, correspondiente a la suma de la latencia de ambos módulos mencionados.

4.4. Almacenamiento de heavy hitters

El almacenamiento de los heavy hitters es necesario para el proceso de control. En este proceso se leen elementos del stream de datos de control y se comparan con los heavy hitters almacenados. Debido a la necesidad de buscar si cada elemento del stream se encuentra o no dentro de los heavy hitters, se requiere una arquitectura de memoria que permita realizar esta comparación de manera eficiente. Para resolver esas necesidades se utilizó una memoria CAM implementada como una memoria asociativa por conjuntos.

Para acceder a esta memoria, el elemento es distribuido en un espacio de menor dimensionalidad por una función hash. Debido a la probabilidad de que multiples elementos colisionen en una misma dirección de memoria, cada dirección está asociada a un subconjunto de elementos. Esta arquitectura permite realizar la comparación del elemento de entrada con los almacenados en dos procesos:

- El cálculo de la función hash asociada.
- La comparación del elemento con los almacenados en el subconjunto apuntado por la dirección de memoria

De esta manera, se realiza la comparación bit a bit de los elementos con un número altamente reducido de posibilidades, muy inferior al total de heavy hitters guardados.

La arquitectura de estas memorias es similar al diseño del sketch. Posee un pipeline de acceso compuesto por funciones hash, almacena los datos en blockRAM, y posee una lógica de actualización. Se utilizaron la funciones hash de la misma familia H3 para la distribución de las direcciones. Se implementó la memoria con un espacio de 64 conjuntos de 4 elementos asociados a cada uno. Para poder realizar las comparaciones con los 4 elementos del mismo conjunto en paralelo, se utilizan 4 bloques de memoria blockRAM de 64 elementos cada una. De esta manera la memoria asociativa de 64 conjuntos de 4 elementos es equivalente a un sketch de dimensiones $W = 64$ y $d = 4$, con políticas de actualización distintas.

El módulo posee 3 modos de funcionamiento: uno para almacenar elementos, uno para realizar el proceso de control, y otro para leer los elementos almacenados.

4.4.1. Almacenamiento de heavy hitters

Este modo de funcionamiento se utiliza mientras se lee el stream de la señal de prueba. Como fue mencionado anteriormente, el almacenamiento de los heavy hitters debe guardar el k-mer heavy hitter y su frecuencia estimada. Con k-mer nos referimos a almacenar la representación de la secuencia de k bases nucleótidas utilizando la representación de 2 bits por base. Aparte del k-mer se recibe su frecuencia estimada divida por dos.

Al recibir los datos, el k-mer es ingresado a la función hash, mientras que la frecuencia estimada es almacenada en registros de desplazamiento. Al obtener el resultado de la función hash se compara en paralelo con los k-mers almacenados en cada uno de los 4 espacios de memoria asociados. Si existe una coincidencia con alguno de los k-mers almacenados, se escribe en el mismo espacio de memoria actualizando su valor de frecuencia estimado.

En en el caso de que no exista coincidencia con los elementos guardados en el conjunto apuntado, se escribe en el primer sitio vacío disponible. Un contador, inicializado en cero, es el encargado de indicar en cuál de los 4 espacios de memoria del conjunto se debe escribir. En el caso de que se utilicen todos los espacios vacios, el contador cíclico vuelve a apuntar al espacio inicial, sobre-escribiendo el primer elemento guardado en caso de que esto ocurra, implementando una política de reemplazo FIFO.

4.4.2. Proceso de control

Cuando finaliza el proceso de lectura del stream de prueba, comienza el proceso de control. En este proceso se lee el stream de datos de control. Estos datos no son procesados en el sketch sino que se procesan en esta unidad de memoria. Según la definición de búsqueda de k-mer emergentes, se deben conservar aquellos elementos que sean heavy hitters (los cuales ya están presentes en la memoria) y que la razón entre sus frecuencias en el conjunto de prueba y el conjunto de control sea superior a un parámetro ρ_g definido como crecimiento. Es decir, para que un k-mer ψ sea emergente debe cumplir $|freq(\psi, D_t)|/|freq(\psi, D_c)| > \rho_g$. Esto es equivalente a plantear que su frecuencia en el conjunto de control debe ser superior a $|freq(\psi, D_t)|/\rho_g$.

En nuestra implementación en hardware, el valor de frecuencia que almacenamos para cada heavy hitter corresponde a su frecuencia estimada por el sketch dividido por el crecimiento ρ_g. Dado que para la aplicación actual el crecimiento tiene un valor de $\rho_g = 2$, esta división se implementa con un desplazamiento lógico de bits. En este modo de funcionamiento, la actualización de los valores de los k-mers analizados se realiza a través de operaciones de resta. Esto quiere decir que cada k-mer presente en el stream, se busca dentro de los valores almacenados. En el caso de haber una coincidencia, se sustrae uno al valor de frecuencia almacenada y se actualiza en la memoria. En el caso de que no hayan coincidencias, no se realizan cambios en las memorias. Si un k-mer almacenado alcanza una frecuencia de cero con este método, quiere decir que corresponde a un heavy hitter que no es un k-mer emergente, por lo que se deja en cero su frecuencia y se elimina el k-mer.

Con este funcionamiento de la memoria se garantiza que al finalizar la lectura del stream de control se encuentren almacenados sólo aquellos elementos que corresponden a k-mer emergentes.

4.4.3. Lectura de k-mers emergentes

Al finalizar el proceso de control, es necesario enviar desde la FPGA al computador host el resultado de los k-mer emergentes. Para realizar este proceso el módulo posee la capacidad de leer elementos almacenados dentro de la memoria. Para realizar esto, el módulo posee una puerta de entrada para recibir direcciones de memoria de consulta, y una puerta de salida que permite entregar los datos almacenados en dicha dirección de memoria. Este proceso posee un ciclo de latencia.

Cuando el módulo se encuentra en este modo de funcionamiento, escribirá en cada ciclo en la

puerta de salida los 4 elementos almacenados en el conjunto apuntado por la dirección recibida en el puerta de entrada el ciclo anterior. Estos datos son empaquetados en 64 bits para cada elemento, tanto el k-mer correspondiente como su frecuencia. De esta manera, los 4 elementos de un conjunto son empaquetados en 256 bits, lo que permite trabajar con los resultados en la interfaz de software con cada elemento como un entero doble.

Este proceso de lectura está controlado por un módulo externo y la única funcionalidad de la memoria en este modo de operación es leer de la dirección requerida, y escribir en la puerta de salida. Estas puertas de entrada y salidas del módulo no son utilizadas por el resto de los modos de operación.

4.5. Módulos adicionales

4.5.1. Lógica de comunicación con el host

El hardware encargado de la comunicación entre la plataforma de SDAccel y el FPGA acelerador es creado automáticamente por el entorno de programación. En nuestro caso, en que incorporamos nuestra lógica diseñada en la forma de un kernel RTL, esto implicó la generación de los controladores de PCIe, los canales de comunicación de datos, y los canales de comunicación de control. Los canales de comunicación incluyen la implementación de los protocolos estándar de comunicación, la generación de buffers de memoria necesarios, y la conexión hacia el espacio de usuario a través de protocolos AXI-4, estándar de los IP de Xilinx.

Desde el espacio de usuario del kernel RTL, la conexión se realiza a través del canal AXI-4. Desde el punto de vista del desarrollo del hardware, si bien el hardware controlador de esta interfaz es invisible al programador, algunos parámetros de los módulos de escritura y lectura son modificables desde el código en SystemVerilog. Estos parámetros modificables incluyen la cantidad de transacciones, las dimensiones de dichas transacciones, y la presencia o no de buffers de entrada/salida que permitan recibir a una tasa constante. En nuestra plataforma, las dimensiones están adecuadas en la lectura para recibir los datos de las dimensiones de las bases de datos de prueba. No se generan los buffers FIFO automáticamente para la lectura desde el FPGA debido a que se utilizaron buffers personalizados por a las necesidades particulares del sketch. En la salida las dimensiones de las transferencias se ajustan al tamaño de las memorias de almacenamiento de heavy hitters. En la salida sí se utilizan buffers generados automáticamente, debido a que la escritura desde el FPGA al canal de comunicación se realiza de manera

continua.

4.5.2. Módulo de entrada al sketch

La lógica personalizada del FPGA se conecta a través de un canal AXI-4 a los canales de comunicación con el host. Como se mencionó recién, el canal AXI-4 no posee un buffer que permita almacenar todo el tamaño de la base de datos y garantizar la lectura de un dato nuevo cada ciclo. Esto significa que se debe utilizar todo el protocolo AXI-4 para la lectura, incluyendo las señales de recepción de datos validos.

A partir de este módulo de lectura desde el canal AXI-4, se generó un buffer FIFO para almacenar los datos recibidos. Este buffer permite recibir datos mientras son entregados a los sketches en forma de stream de k-mers. El buffer está implementado en memoria blockRAM. La dimensión de salida del buffer es tal que permite almacenar una secuencia individual de la base de datos. Esto quiere decir que para una base de datos con secuencias de 200 bases de largo, tendrá 400 bits para poder almacenar los dos bits por base de una secuencia individual. El buffer permite almacenar una cantidad programable de secuencias. Sin embargo, al requerir por múltiples ciclos una misma secuencia, no se requiere una gran profundidad de almacenamiento. Esto es debido a que la lectura desde el buffer es más lenta que la velocidad promedio de escritura.

Hardware especializado se dedica a separar de cada secuencia los k-mers de distintos largos y entregarlos individualmente a la entrada de su sketch correspondiente. Para cada k, se entregan los primeros k elementos en el primer ciclo. El segundo k-mer corresponde al formado desplazándose una base más adelante. Así, para secuencias de 200 bases de largos, luego de $200 - k$ ciclos se habrá procesado la totalidad de la secuencia. Los sketches con valores de k mayores reciben antes su último k-mer a analizar, y deben esperar a que finalice el k-mer más pequeño antes de avanzar hacia la siguiente secuencia. Por ejemplo, si para la misma secuencia de 200 bases se buscan k-mer entre los largos $10 \leq k \leq 20$, luego de 180 ciclos ingresará al pipeline del sketch de largo $k = 20$ el último k-mer de la secuencia. Recién luego de 190 ciclos de reloj ingresará el último k-mer al sketch para largo $k = 10$, lo que permite en el siguiente ciclo analizar una nueva secuencia.

4.5.3. Módulo de escritura al host

A diferencia del módulo de lectura, para la escritura desde el FPGA hacia los espacios de memoria global a través del protocolo AXI-4 si se instanció un buffer FIFO generado de manera automática por las herramientas. Este buffer garantiza la posibilidad de entregar al canal de comunicación un nuevo dato en cada ciclo de reloj, sin preocuparse de la velocidad de transferencia entre el canal y la memoria global. El módulo encargado de entregar la información a escribir a dicho canal es el encargado de recibir la información desde las distintas memorias asociativas por conjuntos que contienen los k-mers emergentes. Este módulo realiza la tarea de solicitar a las unidades de memoria, dirección por dirección, cada uno de los datos almacenados, y de entregarlos para la escritura de vuelta al host a los módulos de comunicación generados desde SDAccel. A pesar de la presencia mayoritaria de espacios de memoria desocupados en las estructuras de almacenamiento, se implementó este esquema de escritura para poder adecuarse a la necesidad de realizar un número fijo de transferencias, propio del modelo de computación heterogénea implementado con SDAccel.

Capítulo 5: Resultados

5.1. Resultados Software

5.1.1. Desempeño sketches.

En esta sección se presentan los resultados del uso la plataforma en software, con las bases de datos y los umbrales de heavy hitters definidos en el capitulo 3. Comparamos el desempeño de los algoritmos, según las métricas definidas en el mismo capítulo, para distintos tamaños. Los tamaños corresponden a sketches de 4×16.384, 7×16.384, y 7×32.768. Estos tamaños fueron aproximados con los márgenes de error teóricos y comprobado su funcionamiento con los distintos sketches.

Para el cálculo de la sensibilidad y la precisión de cada algoritmo se realizó la búsqueda de heavy hitters y el proceso de control para todos los largos $10 \leq k \leq 20$. La sensibilidad y precisión reportados en esta sección fueron calculados considerando los k-mers emergentes de los distintos largos. Con los conjuntos formado por unión de los k-mers emergentes de cada largo y su equivalente estimado con el sketch correspondiente se buscaron los verdaderos positivos para obtener estas métricas.

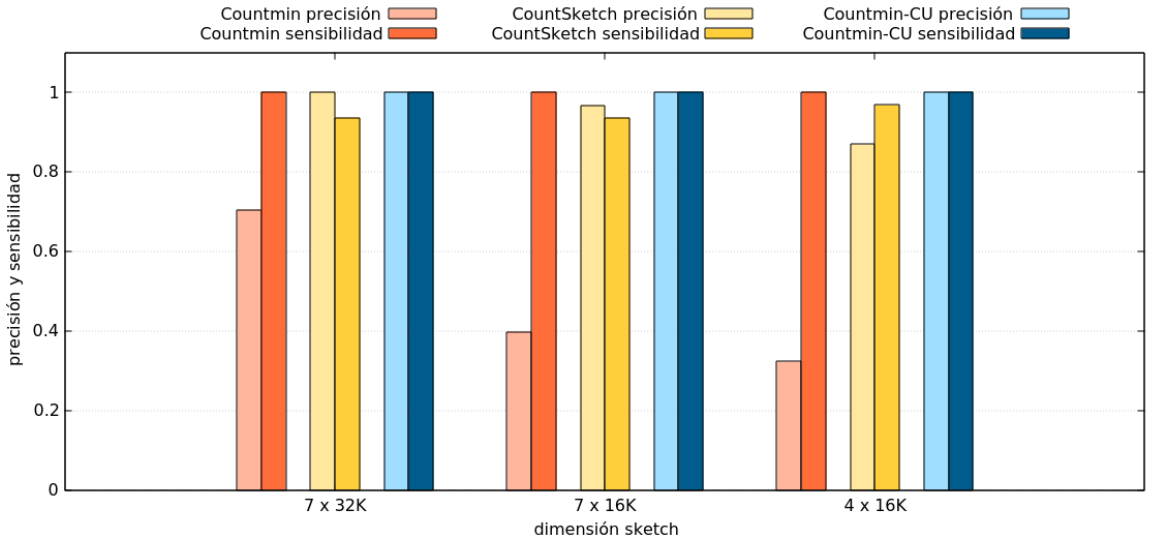


Fig. 5.1: Precisión y sensibilidad para base de datos Esrrb. Fuente: Elaboración propia.

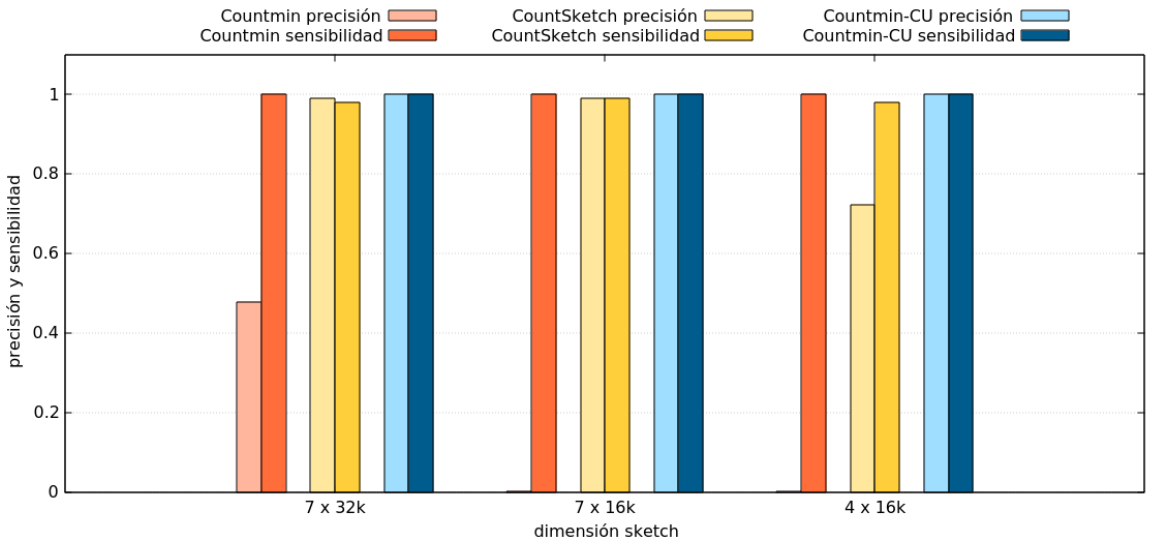


Fig. 5.2: Precisión y sensibilidad para base de datos Tcfcpl1. Fuente: Elaboración propia.

5.1.1.1. Resultados

Los resultados de la base de datos Esrrb de 21.644 secuencias se observan en la figura 5.1. Para el tamaño mayor de memoria, se observa que los tres sketches se comportan relatavimente con un buen desempeño, siendo Countmin el con peor precisión. Si reducimos la cantidad de contadores por fila, pasando a los sketches de 7×16.384, el algoritmo Countmin Sketch posee una caída grande en su precisión.

Estos resultados evidencian lo expuesto teóricamente, en el sentido de que debido al alto valor de la norma ℓ_1 se requieren valores de ε muy pequeños para garantizar una buena precisión con Countmin Sketch. Para tener márgenes de error comparables con el CountSketch requeriríamos valores mayores de ε. Esta diferencia se ve de manera explicita al comparar Countmin con CountSketch en estas dimensiones de 7×16.384. A diferencia de Countmin, CountSketch no presenta caídas significativas en su precisión.

Como es de esperarse, Countmin posee una sensibilidad perfecta para cada tamaño. Esto es debido a la mencionada naturaleza unilateral del error. Este error de sobreestimación produce que se reduzca su precisión, al encontrar más elementos como heavy hitters de los que realmente son, pero no deja ningún elemento heavy hitter fuera de los seleccionados.

La introducción de la actualización CU presenta una mejora significativa en la reducción del error de sobre estimación del sketch Countmin, logrando una precisión incluso superior al de CountSketch para nuestras bases de datos con valores de d inferiores. Esto se comprueba experimentalmente analizando los resultados para sketches de dimensiones 4×16.384. Para esta

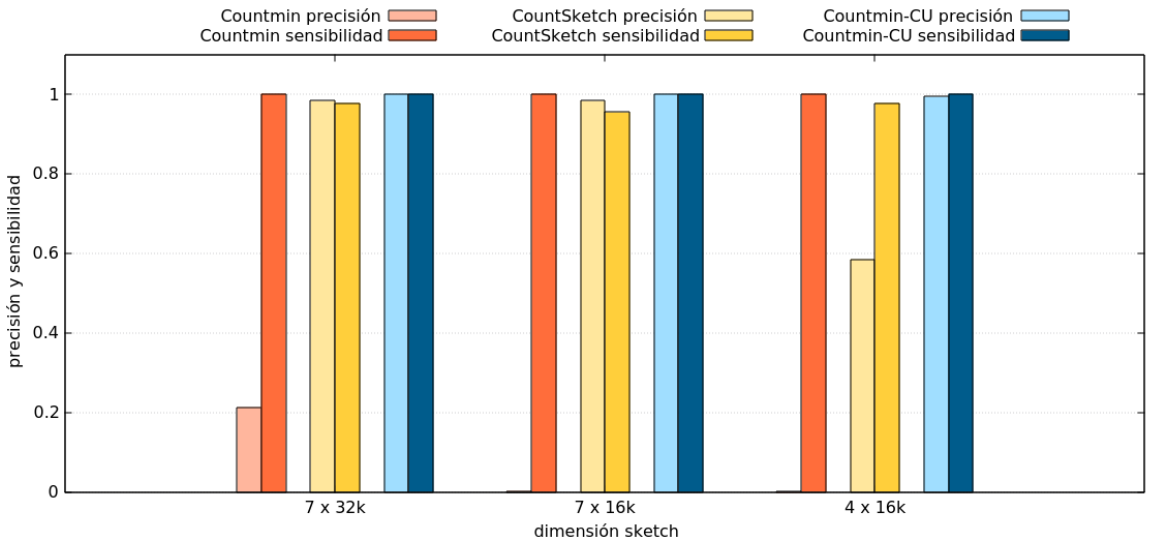


Fig. 5.3: Precisión y sensibilidad para base de datos Ctcf. Fuente: Elaboración propia.

dimensión, Countmin-CU mantiene su nivel de precisión, mientras que CountSketch presenta una disminución significativa. Countmin-CU mantiene la sensibilidad ideal de Countmin. CountSketch posee una sensibilidad no ideal, debido a poseer errores de sobre y subestimación. Sin embargo posee sensibilidad cercana a la ideal para los distintos tamaños.

Las figuras 5.2 y 5.3 muestran los resultados para los mismos tamaño de sketches para las bases de datos Ctcf y Tcfcpl1 respectivamente. En ellas se observa el mismo patrón de comportamiento mencionado anteriormente para los datos Esrrb. La base de datos E2f1, al ser utilizada con el mismo conjunto de control, no presenta k-mers emergentes. Si bien posee heavy hitters según la definición de su umbral, que son correctamente detectado por los algoritmos, estos son eliminados en el proceso de control.

En la tabla 5.1 presentamos de manera más detallada los resultados obtenidos para la base de datos más grande utilizada, Ctcf, de 39.601 secuencias.

5.1.1.2. Error y dependencia de datos

Como fue explicado en la sección 4.3.3, la arquitectura utilizada para la actualización de las filas del sketch contiene un operación que puede introducir errores de coherencia de datos en su pipeline. En esta sección analizaremos el impacto de dichos errores sobre los resultados obtenidos por el sketch.

Para probar el impacto en el desempeño utilizamos una métrica de error relativo. Se define

Tabla 5.1: Especificación de resultados para Ctcf. Fuente: Elaboración propia.

Largo k	10	11	12	13	14	15	16	17	18	19	20
Heavy Hitters	36	40	41	47	49	42	38	32	24	16	13
Heavy Hitters CS	37	39	40	45	49	43	38	31	23	16	13
Heavy Hitters CM	49	51	58	212	315	287	222	180	177	124	84
Heavy Hitters CM-CU	36	40	41	48	50	42	38	32	24	16	13
Emergentes	26	30	31	37	39	32	28	22	14	6	3
Emergentes CS	26	29	30	35	40	33	29	21	14	5	3
Emergentes CM	38	41	54	181	273	242	196	148	140	107	66
Emergentes CM-CU	26	30	31	38	40	32	28	22	14	6	3

Tabla 5.2: Error de estimación promedio para Countmin-CU. Fuente: Elaboración propia.

Base de datos	Esrrb	Tcfcpl1	Ctcf
E_{HH} software	0,09 %	0,11 %	0,15 %
Precisión software	1	1	0,994
Sensibilidad software	1	1	1
E_{HH} pipeline	0,12 %	0,12 %	0,16 %
Precisión pipeline	1	1	0,994
Sensibilidad pipeline	1	1	1

el error relativo de estimación producido por el sketch como:

$$E(x_i, \hat{x_i}) = \frac{|x_i - \hat{x_i}|}{x_i} \times 100 \tag{5.1}$$

donde x_i corresponde a la frecuencia real del elemento i, y $\hat{x_i}$ es la frecuencia de dicho elemento estimada por el sketch. Para estimar el error relativo en la ejecución del algoritmo, definimos el error de estimación promedio como la media geométrica de la estimación de error puntual sobre todo el conjunto de heavy hitters:

$$E_{\text{HH}} = \sqrt[n]{\prod_{i=1}^{n} E(x_i, \hat{x_i})}, \forall i \in \text{HH} \tag{5.2}$$

donde HH es el conjunto de todos los elementos de la base de datos que son heavy hitters según su frecuencia real. La restricción del promedio del error al conjuntos de los heavy hitters fue hecha para medir la distancia a los resultados válidos. Estos cálculos de error fueron realizados con métricas propuestas en [50].

En la tabla 5.2 se observa la estimación del error promedio para el algoritmo Countmin-CU en su versión en software, y la versión con el pipeline con dependencia de datos para las 3 bases de datos con motivos emergentes. En la tabla se observa que, a pesar de que ocurren colisiones que aumentan el error promedio, su impacto en la frecuencia estimada de los heavy hitters es a lo más de 0.03 %. Este error no es suficiente para producir cambios en el proceso de detección de heavy hitters ni en la búsqueda de k-mers emergentes.

5.1.1.3. Conclusiones

A partir de estos resultados de las implementaciones en software desarrolladas se puede señalar en primera medida que los resultados experimentales se ajustan a los esperados por el análisis teórico. Estos resultados experimentales permiten validar la implementación en hardware.

Se confirma que una diferencia significativa entre las normas ℓ_1 y ℓ_2 significa un impacto importante en términos de la relación entre desempeño y espacio requerido por el sketch. Debido a esto, y la caracterización realizada de la base de datos, definimos que, para el problema particular planteado, el algoritmo Countmin-CU se ajusta de mejor manera que CountSketch y Countmin Sketch. La utilización de Countmin-CU permite obtener mejores resultados en menor espacio comparado con sus pares.

Para los análisis siguientes mantenemos como tamaños seleccionados de sketches 4×16.384 para Countmin-CU, 7×16.384 para CountSketch, y el mayor de 7×32.768 para Countmin Sketch, ya que son los parámetros que entregan resultados más similares entre los trés sketches.

El error introducido por el uso del pipeline en hardware que no resuelve la dependencia de datos es menor. Produce solo una variación mínima en la frecuencia estimada y no influye en las métricas de precisión ni sensibilidad con ninguna base de datos. De esta manera, al ser un error despreciable, se decide mantener la arquitectura hardware como fue planteada. Se descarta de esta manera la opción de introducir técnicas de que permitirían eliminar las incoherencias de actualizaciones, pero que tendrían un impacto en el desempeño de la arquitectura.

5.1.2. Tiempos de ejecución en software.

Utilizando las dimensiones seleccionadas para cada sketch recién definidos, medimos el tiempo de ejecución de cada algoritmo para las distintas bases de datos. La tabla 5.3 presenta los resultados de ejecución de los algoritmos.

Tabla 5.3: Tiempos de ejecución (s). Fuente: Elaboración propia.

Base de datos	E2f1	Esrrb	Tcfcpl1	Ctcf
CountSketch	26.00	27.17	29.90	53.44
Countmin	11.74	13.41	15.05	22.48
Countmin-CU	8.64	9.72	11.61	15.59

Esta medición fue realizada a solamente el proceso correspondiente al algoritmo del sketch y el proceso de control propiamente tal. Esto quiere decir que se realizó una versión modificada del programa que mide el tiempo, desde que se encuentran cargadas las bases de datos en memoria RAM, hasta que finaliza el proceso de control, sin realizar ni el conteo real de los k-mers ni los cálculos de desempeño.

Las mediciones fueron realizadas en un servidor equipado con procesador Intel Xeon E5-2630 con 12 núcleos virtuales (capaces de ejecutar 12 hebras simultáneamente) a una frecuencia máxima de 2.80GHz, utilizando el kernel de 4.13.0-39 de Linux. La configuración posee 64GB de memoria RAM DDR3 con un bus de memoria de 1333MT/s, capaz de realizar 1333 millones de transferencias por segundo.

Como es de esperarse en la arquitectura de esta CPU, el tiempo de ejecución aumenta para bases de datos de mayor tamaño. Los tiempos de ejecución también están determinados por los tamaños de sketches utilizados, debido a su impacto en el uso de memoria. De esta manera, se observa que Countmin Sketch requiere más tiempo para ejecutarse que Countmin-CU Sketch. A pesar de que ambos sketches realizan prácticamente las mismas operaciones, Countmin utiliza un sketch 7×32.768, mientras que Countmin-CU está reducido a 4×16.384. Debido a las arquitecturas de memorias jerárquicas de la CPU, el cambio de un sketch de W =16.384 a uno W =32.768 no tiene un impacto mayor. Sin embargo, aumentar la cantidad de filas de $d = 4$ a $d = 7$ tiene un impacto significativo, debido a que se requieren calcular esa cantidad de funciones hash extra para cada k-mer del stream.

Siguiendo en la misma línea, CountSketch es el algoritmo que más tiempo demora en general. Esto debido a que, además de poseer $d = 7$ como Countmin, requiere calcular dos funciones hash en cada fila de cada sketch por elemento. Este impacto resulta más pronunciado para las bases de datos más grandes.

En la tabla 5.4 se presenta un análisis de los tiempos de ejecución de los algoritmos para distintos tamaños de sketches. Para esto se ejecutaron los algoritmos con distintas dimensiones,

Tabla 5.4: Tiempos de ejecución para Ctcf (s). Fuente: Elaboración propia.

Dimensión	4×16.384	7×16.384	7×32.768
CountSketch	33.46	53.44	54.16
Countmin	17.43	20.87	22.48
Countmin-CU	15.59	21.44	21.49

todos utilizando la misma base de datos *Ctcf*. De esta manera se confirma el hecho de que aumentar la cantidad de funciones hash poseen un impacto mayor que cambiar de una dimensión de W =16.384 a W =32.768. Estos resultados también permiten confirmar que los tiempos de ejecución de Countmin y Countmin-CU son cercanos, a pesar de obtener resultados de precisión muy diferentes. CountSketch sigue siendo el algoritmo que requiere mayor tiempo de ejecución.

Para comparar los requisitos de tiempo sin la utilización de las optimizaciones que incluye uso de múltiples hebras e instrucciones vectorizadas, utilizamos una versión del programa solamente optimizada con la opción O3 del compilador GCC. Esta versión tardó 125,76 segundos en ejecutar el algoritmo Countmin-CU 4×16.384 para los 11 largos distintos con su correspondiente proceso de control correspondientes a la base de datos Ctcf. Comparando este resultado con la versión multi-hebra vectorizada, se evidencia una aceleración levemente superior a 8 veces. La arquitectura del procesador utilizado posee ejecución paralela de 12 hebras. El algoritmo utiliza 11 hebras en paralelo, por lo tanto su aceleración ideal corresponde a 11, considerando que es posible ejecutar todas las hebras requeridas en paralelo.

5.2. Resultados implementación hardware

5.2.1. Uso de recursos

5.2.1.1. Comparación entre sketches.

Considerando las dimensiones seleccionadas para cada algoritmo de sketch, se implementó y analizó el uso de recurso lógicos utilizados en el FPGA para un sketch genérico de largo $k = 15$. La tabla 5.5 muestra este resultado, indicando además su precisión y sensibilidad con la base de datos Esrrb. Para conformar esta tabla, se utilizó la tarjeta de desarrollo de Digilent Genesys 2 con una FPGA Xilinx Kintex-7 XC7K325T. La utilización de esta tarjeta, a través de la suite Vivado, nos permitió probar de manera aislada la arquitectura de cada sketch sin tener que

Tabla 5.5: Uso de recursos de un sketch para $k = 15$. Fuente: Elaboración propia.

Sketch	CM	CS	CM-CU
Dimensiones	7x32k	7x16k	4x16k
% LUTs	0.61	0.62	0.31
% Regs	0.55	0.63	0.29
% BRAM	15.7	7.86	4.49
Precisión	0.70	0.96	1
Sensibilidad	1	0.93	1

integrarla al contexto de aceleración hardware con SDaccel. El objetivo de esto es dimensionar el impacto de la utilización de diferentes sketches en cuanto a su uso de recursos. Los valores están representados en porcentaje con respecto a la lógica total disponible para dicho FPGA.

Como se observa en la tabla 5.5, el uso de recursos esta fuertemente determinado por el tamaño del sketch. Esto quiere decir, que independientemente del algoritmo de actualización propio de cada tipo de sketch, es el tamaño de la implementación lo que determina el mayor o menor uso de recursos. En este sentido, el recurso lógico más importante resulta ser la cantidad de memorias blockRAM disponibles.

Debido a sus requisitos de memoria inferiores para almacenar la matriz de contadores, Countmin-CU sketch alcanza un mejor desempeño consumiendo menos recursos. Sin embargo, para sketches de las mismas dimensiones el uso de recursos resulta equivalente. En particular, Countmin-CU y Countmin, para las mismas dimensiones, poseen diferencias mínimas en la implementación. La diferencia más grande en cuanto al software es la actualización condicional en CU, pero esta operación utiliza un recurso lógico que también posee la implementación en hardware de Countmin. Por lo tanto, requiere cambiar internamente conexiones de los recursos lógicos ya utilizados para realizar la actualización condicional.

Por otra parte, CountSketch sí requiere mayor cantidad de recursos para un sketch de las mismas dimensiones. En particular, dado que el algoritmo utiliza una función hash para distribuir la entrada en los contadores y otra para definir el signo de su actualización, se requiere el doble de módulos de hash por cada sketch. Esto se observa en una mayor utilización de LUTs y registros. La utilización de blockRAM, al no ser usadas para las funciones hash, no se afectada. Sin embargo, el uso de recursos sigue manteniéndose muy bajo, inferior al 1 % en aquellos recursos lógicos que afecta.

Tabla 5.6: Resumen uso de recursos. Fuente: Elaboración propia.

Recurso	Utilizados	Disponibles	Uso (%)
LUT	13.078	663.360	1,97
LUTRAM	1.869	293.760	0,64
FF	13.336	1.326.720	1,01
BRAM	264	2.160	12,22

5.2.1.2. Sistema completo

A continuación se analizará el uso de recursos del sistema completo implementado en el FPGA para ser utilizado como acelerador hardware. A diferencia de los resultados anteriores, este sistema incluye los 11 sketches en paralelo, uno para cada largo de k-mers de entre 10 y 20. Además, incluye el uso del sistema de memoria para almacenar k-mers, y toda la interfaz para la comunicación con el host y sus sistemas de control. Los sketches implementados cuentan con los siguientes parámetros: $d = 4$ filas con ancho W =16.384, contadores de 12 bits, representación de entradas en 2 bits por base, memoria CAM de 64 conjuntos con 4 elementos cada uno.

Como se mencionó anteriormente, la implementación del sistema completo se realizó utilizando la plataforma de Xilinx KCU1500 que cuenta con un FPGA Kintex-7 Ultrascale XCKU115. La tabla 5.6 presenta un resumen de la utilización de recursos para el sistema completo. Además presenta los resultados indicando los valores disponibles de los recursos lógicos del FPGA utilizado para la implementación.

Como es de esperarse, el recurso lógico más utilizado son las memorias blockRAM, que alcanzan un 12 % de las disponibles. Cabe señalar que el chip Ultrascale XCKU115 es de alto desempeño, diseñado para aplicaciones de gran tamaño. Para ejemplificar, el 12 % de uso en este chip requiere de 264 bloques de memoria, lo que supera el total disponible en el chip Kintex-7 de la tarjeta Genesys 2. El uso del resto de los recursos se encuentra inferior al 2 %. El sistema no utiliza bloques DSP, debido a que los algoritmos no poseen operaciones aritméticas que los requieran.

Para analizar en mayor detalle el uso de recursos, la tabla 5.7 presenta un desglose jerárquico de toda la arquitectura. Los módulos AXI son los módulos propios de la interfaz de comunicación con el host a través de la abstracción de Kernel-RTL de SDaccel. De estos módulos, *AXI-Ctl* corresponde a la comunicación e implementación de las lógicas de control. *AXI-Com* contabiliza los recursos utilizados por los módulos de lectura y escritura a la memoria global de datos. Tanto cada sketch de distinto largo, como cada una de sus memorias para almacenar k-mers, fueron

Tabla 5.7: Uso de recursos por módulo. Fuente: Elaboración propia.

Módulos	LUTs	REG	CARRY8	F7MUX	BRAM
AXI-Ctl	128	180	-	-	-
AXI-Com	605	1357	25	-	-
Total AXI	733	1537	25	-	-
Sketches	6193	8471	33	-	220
CAMs	1545	2154	72	-	38
Entrada	4349	901	-	12	6
Escritura	259	258	-	-	-
Total Kernel	12346	11784	105	12	264
Total Sistema	13079	13321	130	12	264

contabilizados juntos. Esto se presentan en la tabla como *Sketches* y *CAMs* respectivamente. La tabla también presenta los módulos de entrada a nuestro núcleo, y de salida en la escritura hacia el host.

Como se mencionó anteriormente, el recurso más relevante en cuanto a su uso son las memorias blockRAM. Del análisis del desglose jerárquico se observa que el uso de éstas se da en los módulos de los sketches, y en las memorias de almacenamiento de los k-mers. Una menor cantidad se utiliza para implementación de un buffer de entrada en el módulo correspondiente.

Los sketches también son las unidades jerárquicas que utilizan mayor cantidad de bloques LUT y REG. Esto debido a que poseen en su interior las funciones hash, y registros de desplazamientos para almacenar los valores y k-mers en toda la profundidad del pipeline.

Bloques de multiplexores dedicados, *F7MUX*, son utilizados para implementar la lógica que implementa la entrada a nuestro núcleo desde las puertas AXI. Estos multiplexores se utilizan dentro de las máquinas de estado encargadas de seleccionar, de la secuencia de bases, los k-mers de los largos correspondiente que van a cada sketch.

Además, el sistema utiliza algunas unidades lógicas encargadas de optimizar la propagación de *carry* en operaciones de suma o resta. Éstas se encuentran particularmente concentrados en la lógica encargada de realizar la resta requerida en el proceso de control en las unidades de almacenamiento de k-mers. También están presentes en los circuitos de actualización de línea de los sketches.

Como fue mencionado anteriormente, no se utilizan bloques DSP en ningún módulo de toda la jerarquía.

Tabla 5.8: Tiempos de ejecución hardware. Fuente: Elaboración propia.

Base de datos	E2f1	Esrrb	Tcfcpl1	Ctcf
Tiempo kernel (ms)	41,23	42,09	45,07	53,21
Aceleración	209,55	231,88	257,59	292,99

Tabla 5.9: Consumo de potencia dinámica por recurso. Fuente: Elaboración propia.

Recurso	Potencia (mW)	Potencia (%)
BRAM	1.074	62
Logic	249	14
Signals	257	15
Clocks	153	9
Total	1.732	

5.2.2. Resultados de tiempo

Para la medición de los tiempos de ejecución en hardware se utilizó la misma implementación de Countmin-CU Sketch 4×16.384 con la que fue reportado el uso de recursos del sistema completo. Esta implementación utiliza un reloj de 300MHz tanto en la interfaz de comunicación y control del hardware, como en el kernel propiamente tal. Las mediciones de los tiempos de ejecución fueron realizadas dentro de la plataforma SDaccel con sus herramientas de reportes.

La tabla 5.8 presenta los resultados obtenidos en cuanto a los tiempos de ejecución del la plataforma. Estos resultados de tiempos de ejecución para las distintas bases de datos fueron comparados con los obtenidos en la tabla 5.3 en su columna correspondiente Countmin-CU para obtener los valores de aceleración.

Para las bases de datos probadas, el factor de aceleración se mueve entre 209 para la base de datos más pequeña y 292 para la mayor. Si consideramos que para todas las bases de datos se utilizan los mismos datos de control, se puede notar que el tiempo de ejecución en hardware se comporta de manera prácticamente lineal con respecto al tamaño de la base de datos de prueba. El tiempo que demora el proceso de control es fijo para las distintas bases de datos y es de 28,43ms.

Tabla 5.10: Consumo de potencia dinámica por módulo. Fuente: Elaboración propia.

Módulos	Potencia (mW)	Potencia (%)
AXI-Ctl	3	<1
AXI-Com	33	1
Sketches	1.081	36
CAMs	486	16
Entrada	124	4
Escritura	5	<1
Total	1.732	58

5.2.3. Uso de potencia

Utilizando la herramienta XPower Analyzer de Xilinx fueron estimados los requisitos de consumo de potencia para del sistema completo, implementando la arquitectura y la interfaz de comunicación a una velocidad de reloj de 300MHz.

Este consumo llega a 3,01W. De esta potencia 1,73W, el 58 %, corresponde a la disipación dinámica. El 42 % restante, 1,28W, corresponde a pérdidas estáticas. La tabla 5.9 presenta un desglose del consumo dinámico de potencia por recursos lógicos. En ella se observa que las memorias blockRAM consumen sobre el 62 % de la potencia dinámica total. Esto es consecuente con la utilización de recursos, donde se evidencia el fuerte predominio en la utilización de blockRAM sobre el resto en nuestro diseño.

La tabla 5.10 muestra el consumo de potencia dinámica desglosado en los módulos jerárquicos de las distintas unidades funcionales. Esta jerarquización corresponde a la misma que explicada para la tabla 5.7 sobre el uso de recursos.

Los sketches y las memorias de almacenamiento de k-mers son las unidades funcionales que más potencia consumen. Esto se explica porque presentan un mayor uso de recursos, sobre todo en cuanto a memorias blockRAM, que como hemos visto en la tabla 5.9 consumen la mayor parte de la potencia dinámica. Todo el control de los canales de comunicación AXI presenta un consumo de aproximadamente un 1 %. La mayor parte de esta potencia es disipada en los módulos de comunicación de datos, especialmente el de escritura. Si comparamos el módulo de lectura de datos AXI con el de escritura, vemos que consumen 1mW y 32mW respectivamente. Esta diferencia es consecuencia de que, como fue explicado en la sección 4.5, el módulo de escritura implementa una memoria como un buffer tipo FIFO y el de lectura no. De manera inversa, nuestro módulo de entrada consume más potencia que el equivalente de escritura. Esto

es debido a la misma decisión de implementación, ya que en este nivel el módulo de entrada implementa un buffer con blockRAM y el de escritura no.

En nuestro diseño la potencia estática alcanza valores altos. Además, representa un porcentaje muy significativo del total. Esto se puede explicar por dos motivos. En primera instancia, la serie 7 Ultrascale de Xilinx utiliza el nodo tecnológico de transistores de 20nm. En los nodos tecnológicos más pequeños es esperado que aumente el impacto de la potencia estática. Con transistores de dimensiones más pequeños aumentan las corrientes de fuga tanto en la puerta como a través de su canal. Estas corrientes son las responsables de la disipación estática, por lo que ésta aumenta en las tecnologías más pequeñas. Sin embargo, transistores más pequeños requieren voltajes umbral de activación más pequeños por lo que requieren ser alimentados con menor energía, reduciendo así su potencia dinámica. Por lo tanto, para los nodos tecnológicos actuales es esperable que el consumo estático sea un mayor porcentaje del total.

También hay que considerar que en nuestro diseño, la ocupación del chip es pequeña. La potencia dinámica es proporcional a ésta, mientras que la potencia estática es proporcional al tamaño total del chip. Por otro lado, nuestro sistema esta dominado por el uso de los recursos de blockRAM. Según lo expuesto por Xilinx, las blockRAM realizan una contribución significativa a la disipación estática total del dispositivo [51]. Además, las optimizaciones al consumo de potencia estática en estos recursos lógicos se han centrado en técnicas de clock gating que permitan desactivar los sectores de blockRAM que no son utilizados. Sin embargo en nuestro sistema se utiliza constantemente la gran mayoría de las memorias blockRAM usadas en el diseño, por lo que ese tipo de optimización no contribuiría a la reducción de su consumo de potencia estática.

5.3. Escalamiento para streams de mayor dimensiones

Para probar la respuesta de nuestras implementaciones a la utilización de bases de datos de mayores dimensiones, se utilizó como stream de entrada la base de datos Hepg2, caracterizada en la sección 3.1. Esta base de datos fue utilizada para probar tanto el software como la plataforma de aceleración hardware. Esta base de datos consiste en información ChIP-seq de secuencias genéticas humanas. El conjunto de prueba es de 252.660 secuencias, cada una de 200 bases de longitud. El set de control también cuenta con 252.660 secuencias de 200 bases de largo, que al igual que para las bases de datosde mESC, fue conformada con las secuencia partiendo a 400 bases de distancia del centro de los peaks ChIP-seq.

Tabla 5.11: Tiempos de ejecución con Hepg2. Fuente: Elaboración propia.

Medición	Tiempo
Tiempo software	135,17 (s)
Tiempo hardware	324,64 (ms)
Aceleración	471,81

Como ha sido reportado para esta base de datos [52], los motivos de TFBS son de mayor largo que los presentes en las bases de datos de mESC. Para las otras bases de datos se buscó en los largos $10 \geq k \geq 20$. Para estresar más la arquitectura, utilizaremos más sketches en paralelo. De esta manera, para las pruebas con Hepg2 se utilizaron sketches para largos de k en el rango $10 \geq k \geq 28$. La figura 5.4 muestra el desempeño del algoritmo utilizando para el conteo aproximado los distintos sketches con distintas dimensiones con la base de datos Hepg2. Como se puede observar en dicha figura, a pesar de tratarse de una base de datos más de seis veces superior en tamaño que las anteriores, con las mismas dimensiones de 4×16.384, el algoritmo Countmin-CU posee precisión y sensibilidad perfectas. Sin embargo, el máximo valor de los contadores alcanza valores superiores que en los otros casos. En la figura 7.4 se observa que el k-mer más frecuente posee alrededor de 50 mil ocurrencias. Debido a esto, los contadores de la matriz de 4×16.384 deberán ser de 16 bits cada uno. En cambio, para las bases de datos de mESC con 12 bits eran suficientes.

La tabla 5.11 muestra las mediciones de tiempos de ejecución, tanto para software como para hardware, con la base de datos Hepg2. Los tiempos en software se aumentaron considerablemente con respecto a las resultados anterior. Esto se debe, a parte del tamaño de las bases de datos,

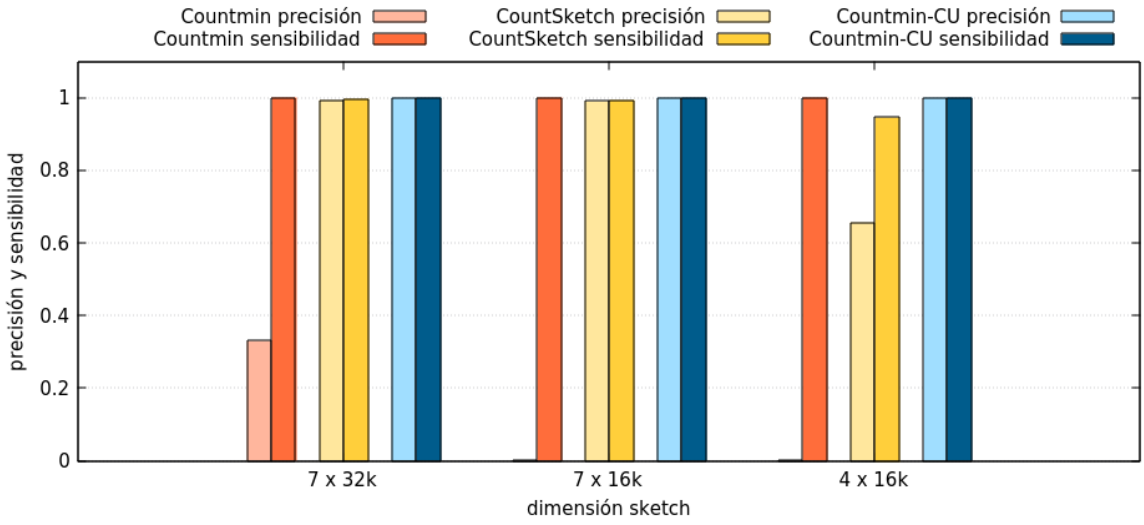


Fig. 5.4: Precisión y sensibilidad para base de datos Hepg2. Fuente: Elaboración propia.

Tabla 5.12: Comparación en la utilización del FPGA. Fuente: Elaboración propia.

Recurso	mESC (%)	Hepg2 (%)
LUT	1,97	2,44
LUTRAM	0,64	0,72
FF	1,01	1,31
BRAM	12,22	29,91

Tabla 5.13: Comparación en el consumo de potencia. Fuente: Elaboración propia.

Potencia	mESC	Hepg2
Dinámica (W)	1,73	3,91
Estática (W)	1,28	1,32
Dinámica (%)	58	75
Estática (%)	42	25
Total (W)	3,01	5,23

a que el número sketches ejecutándose en paralelo supera a las hebras disponibles en la CPU. Como la arquitectura en hardware posee un flujo de procesamiento de 1 k-mer por ciclo, el aumento del tiempo es linear con respecto al tamaño del stream de entrada. De esta menare el acelerador hardware ejecuta la búsqueda de k-mers emergentes en la base de datos Hepg2 más de 471 veces más rápido que nuestra plataforma optimizada en software.

Para profundizar en la comparación de ambas dimensiones de implementación, se analizó el uso de recursos y el consumo de potencia. La comparación del uso de recursos lógicos del chip FPGA se muestra en la tabla 5.12. Como es de esperarse, la utilización de recursos se incremento considerablemente, especialmente para las memorias blockRAM. A pesar de que las dimensiones de cada sketch se mantiene igual, se utilizan más blockRAM debido a la mayor cantidad de sketches en paralelos, cada uno con su memoria CAM respectiva. A pesar de que la cantidad de sketches paralelos se incrementó de 11 a 19, el uso de blockRAM se incrementó más de 2,44 veces. Este incremento mayor al increment de sketches en paralelo se debe al aumento del tamaño de contadores de 12 a 16 bits. La tabla 5.13 compara el consumo de potencia de ambas implementaciones. Para la implementación con mayores dimensiones se incrementó el consumo de potencia más de 1,73 veces. Debido a que la implementación utiliza un área mayor del chip FPGA, el porcentaje de consumo debido a las pérdidas estáticas se redujo. Esto se debe a que estas pérdidas estáticas, al depender principalmente del tamaño del chip, no se incrementaron significativamente, como es el caso del consumo dinámico.

Capítulo 6: Conclusiones

La implementación de los algoritmos de sketch en una plataforma de computación heterogénea formada por un procesador de propósito general y un acelerador hardware dedicado implementado en un FPGA permite resolver el problema de la búsqueda de motivos emergentes en bases de datos de secuencias de ADN. El circuito diseñado es capaz de implementar los algoritmos de CountSketch y Countmin-CU para encontrar los elementos más frecuentes, heavy hitters, en un stream formado por secuencias de ADN. Además, este circuito también realiza el proceso de análisis de la bases de datos de control necesario para completar la búsqueda de motivos emergentes. Los arquitecturas dedicadas diseñadas para los sketches son aplicables a otros problemas de detección de heavy hitters, como lo es el análisis del tráfico de redes de comunicación o los procesos de minería dentro del procesamiento del uso de lenguaje natural. Se desarrolló una herramienta utilizando Python que brinda una interfaz de usuario a nivel de scripts que permite modificar los parámetros de las implementaciones en lenguaje HDL de las arquitecturas diseñadas.

La plataforma de computación homogénea se configuró utilizando una tarjeta de desarrollo KCU1500 de Xilinx que posee un FPGA Kintex-7 UltraScale XCKU115-2FLVB2104E. Se utilizó el ambiente de desarrollo SDaccel de Xilinx para desarrollar la interfaz de comunicación entre el software y el acelerador hardware. Este ambiente permite utilizar el FPGA como acelerador hardware para código desarrollado en lenguaje OpenCL, conectado a través de un puerto PCIe. Esta plataforma permitió integrar nuestro hardware dedicado diseñado empaquetado como un núcleo RTL.

Sobre una base de datos de 39.601 secuencias de 200 bases cada una, esta plataforma es capaz de encontrar los motivos emergentes para largos $10 \leq k \leq 20$ utilizando en paralelo sketches Countmin-CU de dimensiones 4×16.384. Este proceso requiere bajo 54 milisegundos utilizando un reloj de 300MHz. Comparado con una implementación en software desarrollada en C++ utilizando OpenMP para ejecución en multiples hebras e instrucciones de vectorización SIMD, este tiempo de ejecución alcanza un factor de aceleración de 290 veces. Utilizando el FPGA Kintex-7 UltraScale con los 11 sketches en paralelo, añadido a la lógica de la interfaz de comunicación a través de PCIe, se alcanza un 12 % de uso de las memorias blockRAM y menores al 2 % de los recursos LUT y registros disponibles. Además, el FPGA consume una potencia de aproximadamente 3W.

Dentro de las principales limitaciones de la plataforma desarrollada se encuentra el hecho

de que, a pesar de poseer la interfaz en Python que permite ajustar a alto nivel los parámetros de la implementación para cada núcleo de sketch, la comunicación con el host a través de las interfaces utilizadas incluye configuraciones que están programadas dentro del diseño de los módulos en SystemVerilog, que evitan que la plataforma sea adaptable de manera sencilla a, por ejemplo, bases de datos de otros tamaños. A partir de esto se plantea como trabajo futuro:

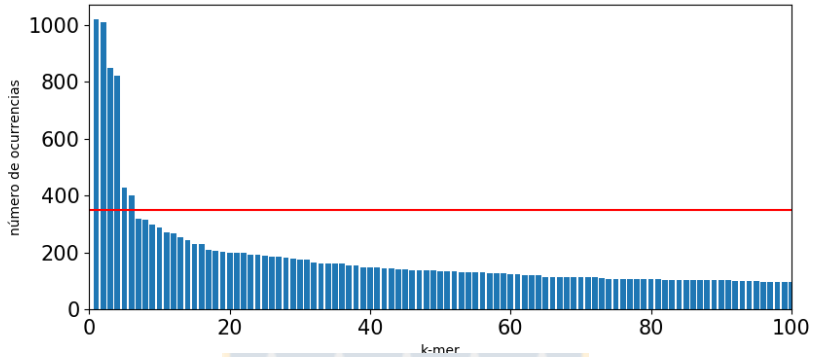
- Un análisis en profundidad de las herramientas disponibles en las interfaces de comunicación de Xilinx para el desarrollo de un sistema de comunicación que no dependa de configuraciones establecidas dentro del lenguaje HDL. En particular, la utilización de Kernels RTL permite la utilización de valores escalares como configuración, que se podrían integrar en la plataforma desarrollada.
- Dentro de las posibilidades entregadas por estas herramientas, también se encuentra el uso de dos dominios de reloj dentro de un kernel RTL. Se puede analizar la posibilidad de ejecutar el kernel del sketch con un reloj más rápido que el estándar de la comunicación de 300MHz.

Más allá de las plataformas de programación, el trabajo futuro más relevante consiste en analizar la posibilidad de desarrollar en hardware, o integrado a la plataforma homogénea, el proceso de clustering y combinación de los k-mers emergentes para formar motivos. De esta manera se podría utilizar la salida de nuestros procesamiento, que corresponde a los k-mers emergentes agrupados en distintos largos, y realizar todo el proceso de descubrimiento de motivos en TFBS en una plataforma única.

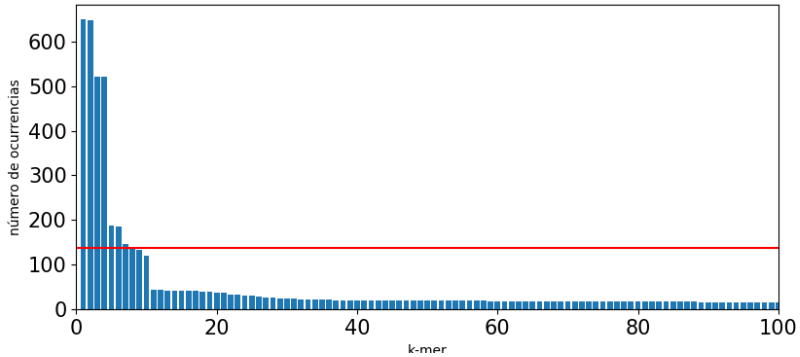
Capítulo 7: Anexo: Bases de datos

7.1. Distribución E2f1.

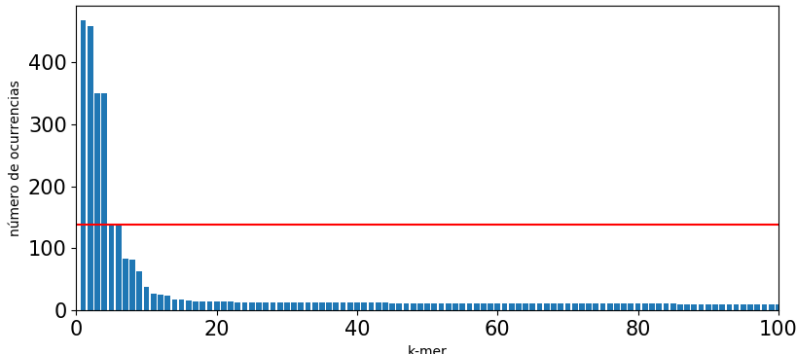
Fuente: Elaboración propia.



$k = 10$



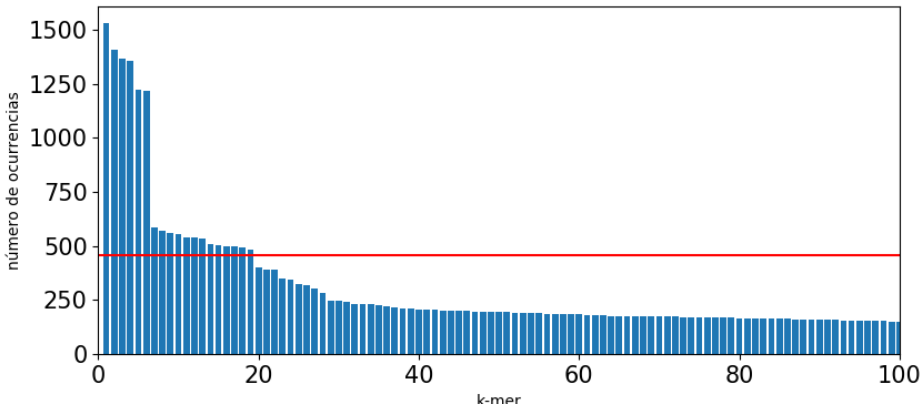
$k = 15$



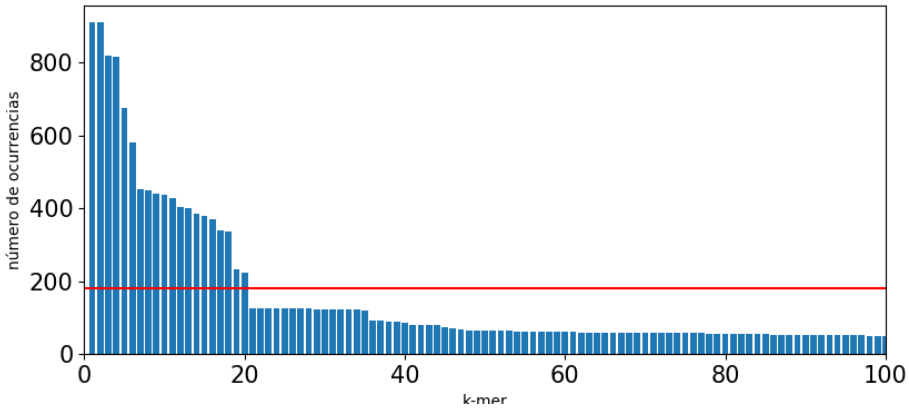
$k = 20$

7.2. Distribución Tcfcpl1

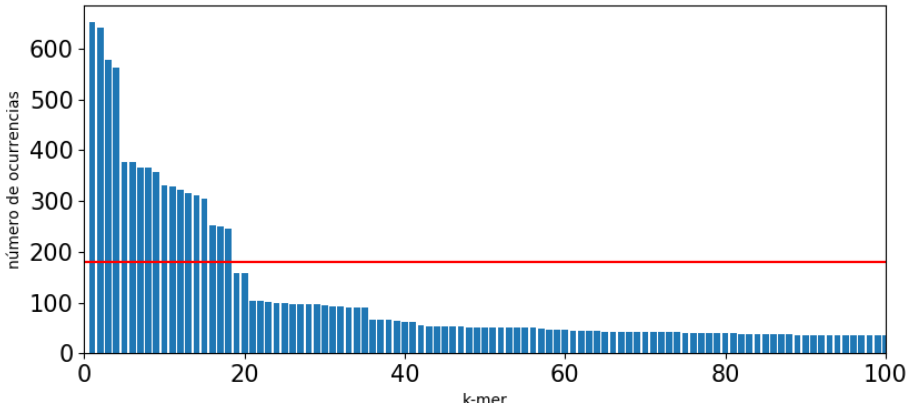
Fuente: Elaboración propia.



$k = 10$



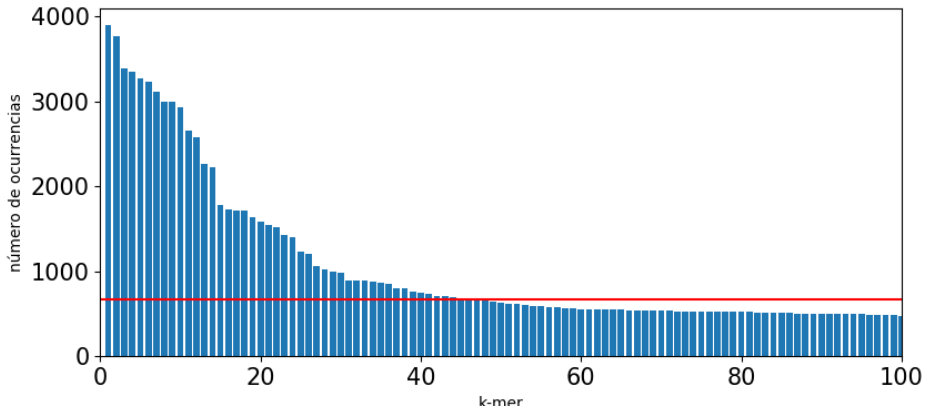
$k = 15$



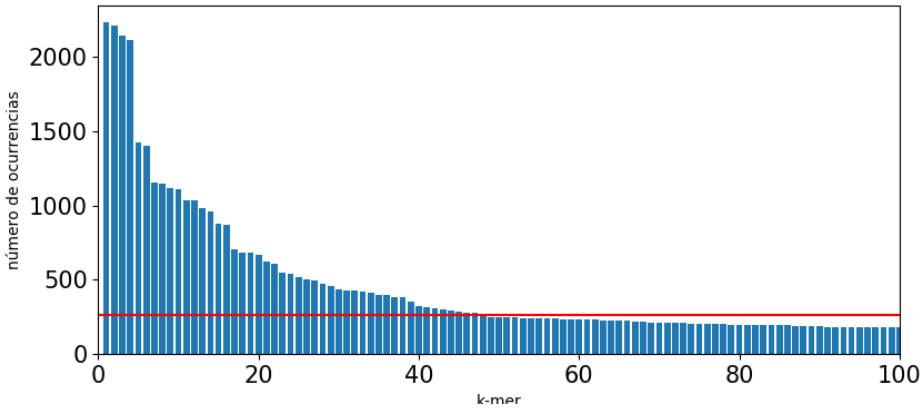
$k = 20$

7.3. Distribución Ctcf

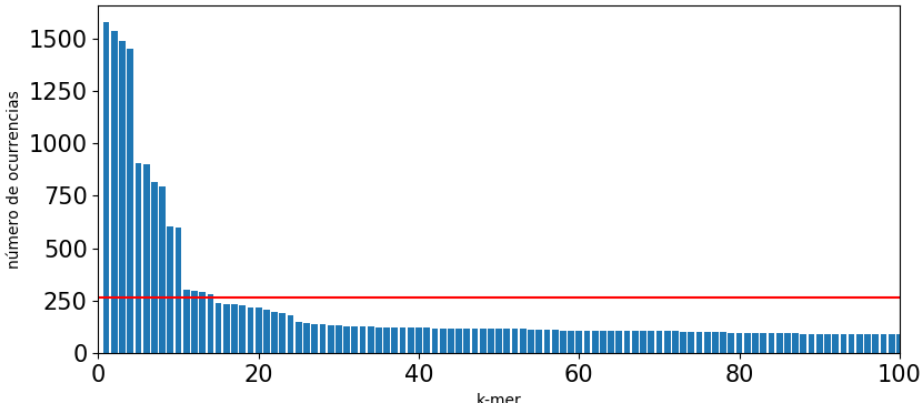
Fuente: Elaboración propia.



$k = 10$



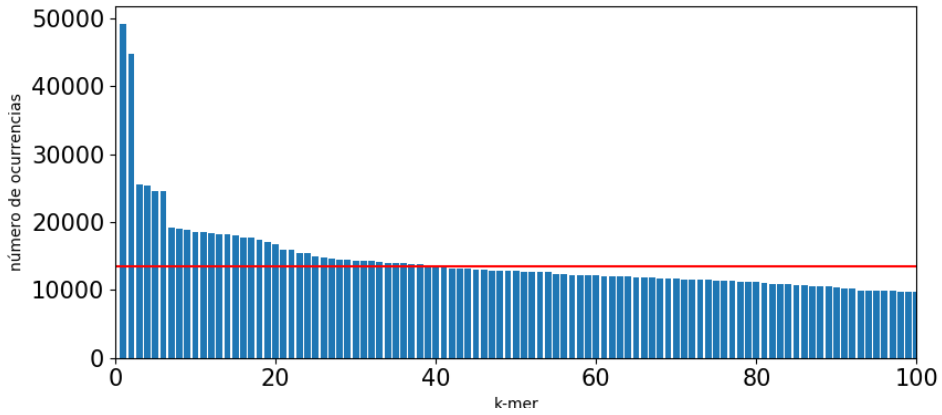
$k = 15$



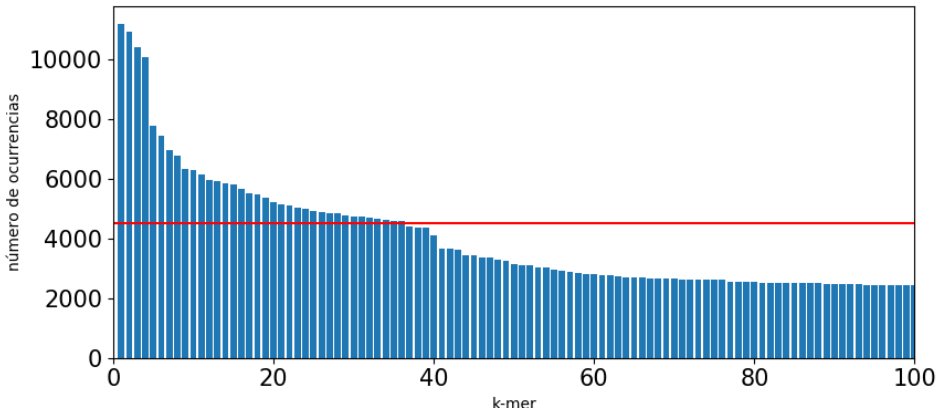
$k = 20$

7.4. Distribución Hepg2

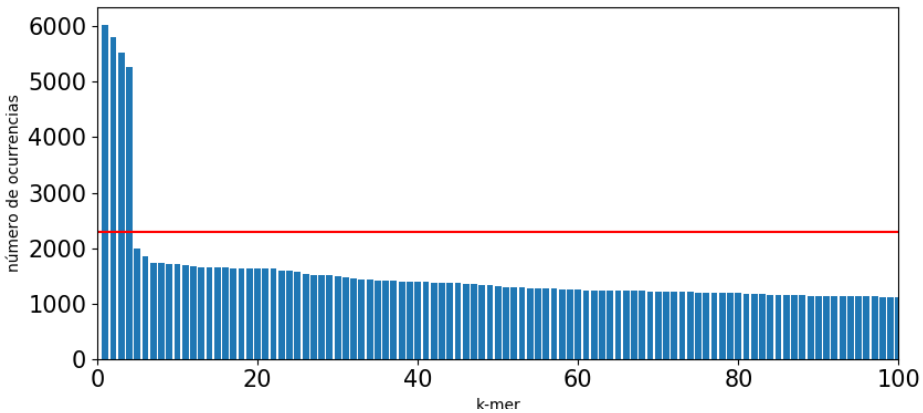
Fuente: Elaboración propia.



$k = 10$



$k = 20$



$k = 28$

Bibliografía

[1] K. Wetterstrand. (2017, Oct.) DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP). [Online]. Available: www.genome.gov/sequencingcostsdata.

[2] P. Collas and J. A. Dahl, "Chop it, chip it, check it: the current status of chromatin immunoprecipitation," *Frontiers in Bioscience*, vol. 13, pp. 929–943, 2008.

[3] J.-Y. Zhu, Y. Sun, and Z.-Y. Wang, *Genome-Wide Identification of Transcription Factor-Binding Sites in Plants Using Chromatin Immunoprecipitation Followed by Microarray (ChIP-chip) or Sequencing (ChIP-seq).* Totowa, NJ: Humana Press, 2012, pp. 173–188. [Online]. Available: https://doi.org/10.1007/978-1-61779-809-2_14

[4] F. Zambelli, G. Pesole, and G. Pavesi, "Motif discovery and transcription factor binding sites before and after the next-generation sequencing era," *Briefings in Bioinformatics*, vol. 14, no. 2, pp. 225–237, 2013. [Online]. Available: http://dx.doi.org/10.1093/bib/bbs016

[5] V. Boeva, "Analysis of genomic sequence motifs for deciphering transcription factor binding and transcriptional regulation in eukaryotic cells," *Frontiers in Genetics*, vol. 7, p. 24, 2016. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fgene.2016.00024

[6] J. Keilwagen and J. Grau, "Varying levels of complexity in transcription factor binding motifs," *Nucleic Acids Research*, vol. 43, no. 18, p. e119, 2015. [Online]. Available: +http://dx.doi.org/10.1093/nar/gkv577

[7] C. Jia, M. B. Carson, Y. Wang, Y. Lin, and H. Lu, "A new exhaustive method and strategy for finding motifs in chip-enriched regions," *PLOS ONE*, vol. 9, no. 1, pp. 1–13, 01 2014. [Online]. Available: https://doi.org/10.1371/journal.pone.0086044

[8] D. Langenkämper, T. Jakobi, D. Feld, L. Jelonek, A. Goesmann, and T. W. Nattkemper, "Comparison of acceleration techniques for selected low-level bioinformatics operations," *Frontiers in Genetics*, vol. 7, p. 5, 2016. [Online]. Available: http://europepmc.org/articles/PMC4748744

[9] T. Bailey and C. Elkan, "Fitting a mixture model by expectation maximization to discover motifs in biopolymers." in *International Conference on Intelligent Systems for Molecular Biology*, 1994, pp. 28–36.

[10] T. L. Bailey, M. Boden, F. A. Buske, M. Frith, C. E. Grant, L. Clementi, J. Ren, W. W. Li, and W. S. Noble, "Meme suite: tools for motif discovery and searching," *Nucleic Acids Research*, vol. 37, no. 2, pp. W202–W208, 2009.

[11] N. Jayaram, D. Usvyat, and A. C. R. Martin, "Evaluating tools for transcription factor binding site prediction," *BMC Bioinformatics*, Nov 2016. [Online]. Available: https://doi.org/10.1186/s12859-016-1298-9

[12] Q. Yu, H. Huo, Y. Zhang, and H. Guo, "Pairmotif: A new pattern-driven algorithm for planted (l, d) dna motif search," *PLOS ONE*, vol. 7, no. 10, pp. 1–10, 10 2012. [Online]. Available: https://doi.org/10.1371/journal.pone.0048442

[13] Q. Yu, H. Huo, Y. Zhang, H. Guo, and H. Guo, "Pairmotif+: A fast and effective algorithm for de novo motif discovery in dna sequences," *International Journal of Biological Sciences*, vol. 9, no. 1449-2288, pp. 412–424, 4 2013. [Online]. Available: https://doi.org/10.1371/journal.pone.0048442

[14] P. Machanick and T. L. Bailey, "Meme-chip: motif analysis of large dna datasets," *Bioinformatics*, vol. 27, no. 12, pp. 1696–1697, 2011. [Online]. Available: +http://dx.doi.org/10.1093/bioinformatics/btr189

[15] D. Quang and X. Xie, "Extreme: an online em algorithm for motif discovery," *Bioinformatics*, vol. 30, no. 12, pp. 1667–1673, 2014. [Online]. Available: +http://dx.doi.org/10.1093/bioinformatics/btu093

[16] T. L. Bailey, "Dreme: motif discovery in transcription factor chip-seq data," *Bioinformatics*, vol. 27, no. 12, pp. 1653–1659, 2011. [Online]. Available: +http://dx.doi.org/10.1093/bioinformatics/btr261

[17] J. Grau, S. Posch, I. Grosse, and J. Keilwagen, "A general approach for discriminative de novo motif discovery from high-throughput data," *Nucleic Acids Research*, vol. 41, no. 21, p. e197, 2013. [Online]. Available: +http://dx.doi.org/10.1093/nar/gkt831

[18] Q. Yu, H. Huo*, X. Chen, H. Guo, J. S. Vitter, and J. Huan, "An efficient algorithm for discovering motifs in large dna data sets," *IEEE Transactions on NanoBioscience*, vol. 14, no. 5, pp. 535–544, July 2015.

[19] J. Fischer, V. Heun, and S. Kramer, *Optimal String Mining Under Frequency Constraints*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 139–150. [Online]. Available: https://doi.org/10.1007/11871637_17

[20] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[21] G. Chrysos, E. Sotiriades, C. Rousopoulos, K. Pramataris, I. Papaefstathiou, A. Dollas, A. Papadopoulos, I. Kirmitzoglou, V. J. Promponas, T. Theocharides, G. Petihakis, and J. Lagnel, "Reconfiguring the bioinformatics computational spectrum: Challenges and opportunities of fpga-based bioinformatics acceleration platforms," *IEEE Design Test*, vol. 31, no. 1, pp. 62–73, Feb 2014.

[22] D. P. Woodruff, "New Algorithms for Heavy Hitters in Data Streams," *CoRR*, vol. abs/1603.01733, 2016. [Online]. Available: http://arxiv.org/abs/1603.01733

[23] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," *The VLDB Journal*, vol. 19, no. 1, pp. 3–20, Feb 2010. [Online]. Available: https://doi.org/10.1007/s00778-009-0172-z

[24] R. S. Boyer and J. S. Moore, *MJRTY—A Fast Majority Vote Algorithm*. Dordrecht: Springer Netherlands, 1991, pp. 105–117. [Online]. Available: https://doi.org/10.1007/978-94-011-3488-0_5

[25] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, no. 1, pp. 51–55, Mar. 2003. [Online]. Available: http://doi.acm.org/10.1145/762471.762473

[26] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 346–357. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287369.1287400

[27] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of the 10th International Conference on Database Theory*, ser. ICDT'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 398–412. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30570-5_27

[28] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3 – 15, 2004, Automata, Languages and Programming. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397503004006

[29] M. Muthukrishnan and G. Cormode, "Approximating data with the count-min sketch," *IEEE Software*, vol. 29, pp. 64–69, 2011.

[30] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 270–313, 2003.

[31] A. Goyal and H. Daumé, III, "Approximate scalable bounded space sketch for large data nlp," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP '11. Stroudsburg, PA, USA: Association for Computational Linguistics, 2011, pp. 250–261. [Online]. Available: http://dl.acm.org/citation.cfm?id=2145432.2145461

[32] A. Goyal, H. Daumé, III, and G. Cormode, "Sketch algorithms for estimating point queries in nlp," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, ser. EMNLP-CoNLL '12. Stroudsburg, PA, USA: Association for Computational Linguistics, 2012, pp. 1093–1103. [Online]. Available: http://dl.acm.org/citation.cfm?id=2390948.2391070

[33] G. Einziger and R. Friedman, "A formal analysis of conservative update based approximate counting," in *2015 International Conference on Computing, Networking and Communications (ICNC)*, Feb 2015, pp. 255–259.

[34] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, Dec 1997.

[35] J. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143 – 154, 1979. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000079900448

[36] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown, "These are not the k-mers you are looking for: Efficient online k-mer counting using a probabilistic data structure," *PLOS ONE*, vol. 9, no. 7, pp. 1–13, 07 2014. [Online]. Available: https://doi.org/10.1371/journal.pone.0101271

[37] P. Zandevakili, M. Hu, and Z. Qin, "Gpumotif: An ultra-fast and energy-efficient motif analysis program using graphics processing units," *PLOS ONE*, vol. 7, no. 5, pp. 1–6, 05 2012. [Online]. Available: https://doi.org/10.1371/journal.pone.0036865

[38] M. Hu, J. Yu, J. M. G. Taylor, A. M. Chinnaiyan, and Z. S. Qin, "On the detection and refinement of transcription factor binding sites using ChIP-Seq data," *Nucleic Acids Research*, vol. 38, no. 7, pp. 2154–2167, 2010. [Online]. Available: +http://dx.doi.org/10.1093/nar/gkp1180

[39] E. Sotiriades and A. Dollas, "A general reconfigurable architecture for the blast algorithm," *The Journal of VLSI Signal Processing*, vol. 48, no. 3, pp. 189–208, 2007.

[40] L. Wienbrandt, S. Baumgart, J. Bissel, C. M. Y. Yeo, and M. Schimmler, "Using the reconfigurable massively parallel architecture copacobana 5000 for applications in bioinformatics," *Procedia Computer Science*, vol. 1, no. 1, pp. 1027 – 1034, 2010, iCCS 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050910001158

[41] D. Tong and V. Prasanna, "High throughput sketch based online heavy hitter detection on fpga," *SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 70–75, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2927964.2927977

[42] D. Nguyen, G. Memik, S. O. Memik, and A. Choudhary, "Real-time feature extraction for high speed networks," in *International Conference on Field Programmable Logic and Applications, 2005.*, Aug 2005, pp. 438–443.

[43] J. F. Zazo, S. Lopez-Buedo, M. Ruiz, and G. Sutter, "A single-fpga architecture for detecting heavy hitters in 100 gbit/s ethernet links," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2017, pp. 1–6.

[44] D. Tong and V. K. Prasanna, "Sketch acceleration on fpga and its applications in network anomaly detection," *IEEE Transactions on Parallel & Distributed Systems*, vol. 29, no. 4, pp. 929–942, April 2018. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TPDS.2017.2766633

[45] R. Dréos, G. Ambrosini, R. Groux, R. C. Périer, and P. Bucher, "Mga repository: a curated data resource for chip-seq and other genome annotated data," *Nucleic acids research*, vol. 46, no. D1, pp. D175–D180, 2017.

[46] X. Chen, H. Xu, and P. Yuan, "Integration of external signaling pathways with the core transcriptional network in embryonic stem cells," *Cell*, vol. 133, no. 6, pp. 1106–1117, June 2008. [Online]. Available: http://dx.doi.org/10.1016/j.cell.2008.04.043

[47] Y. Zhang and P. Wang, "A fast cluster motif finding algorithm for chip-seq data sets," *BioMed Research International*, 2015. [Online]. Available: http://dx.doi.org/10.1155/2015/218068

[48] "Sdaccel profiling and optimization guide," Xilinx, Tech. Rep. UG1207, July 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1207-sdaccel-optimization-guide.pdf

[49] "Sdaccel environment user guide," Xilinx, Tech. Rep. UG1023, July 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1023-sdaccel-user-guide.pdf

[50] G. T. Minton and E. Price, "Improved concentration bounds for count-sketch," *CoRR*, vol. abs/1207.5200, 2012. [Online]. Available: http://arxiv.org/abs/1207.5200

[51] S. Kolluri, "White paper: Ultrascale architecture low power technology overview," Xilinx, Tech. Rep. WP451, October 2015. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp451-ultrascale-pwr-reduction.pdf

[52] J. Wang, J. Zhuang, S. Iyer, X.-Y. Lin, M. C. Greven, B.-H. Kim, J. Moore, B. G. Pierce, X. Dong, D. Virgil *et al.*, "Factorbook. org: a wiki-based database for transcription factor-binding data generated by the encode consortium," *Nucleic acids research*, vol. 41, no. D1, pp. D171–D176, 2012.