UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

ESTIMACIÓN DE CUANTILES EN ACELERADOR HARDWARE USANDO SKETCH KLL±

POR

Felipe Martín Winkler Enríquez

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil Electrónico

Profesores Guía
Dr. Miguel Figueroa
Dra. Cecilia Hernández

Septiembre 2025
Concepción (Chile)

“It's the questions we can't answer that teach us the most. They teach us how to think. If you give a man an answer, all he gains is a little fact. But give him a question and he'll look for his own answers”

Patrick Rothfuss, The Wise Man Fear

Agradecimientos

Para comenzar, quería agradecer a mis papás: a mi papá porque fue quien más me apoyó cuando me empecé a interesar por esta área de la ingeniería, porque fue quien dio su sabiduría cada vez que me enfrentaba a un problema que no podía resolver. A mi mamá, porque fue quien me apoyó por todos estos 5 años y medio, por quien más me animaba a continuar y quien me enseñó el valor del esfuerzo. Quería agradecer a mi hermana también, porque me enseñó a tener perseverancia para enfrentar los problemas. También quería agradecer a mi tía Claudia, a mi prima Florencia y su esposo Aníbal, a los 6 porque fueron quienes más me apoyaron y animaron durante estos años. Y sin falta a la Paki y al Tito también, quienes me acompañaban a estudiar.

También quería agradecer a mis dos mejores amigos, a Joaquín Vera y a Vicente Sierra, por entenderme y su paciencia. Le agradezco a Joaquín Vera, por a pesar de vivir en ciudades distintas, seguir en contacto y por todo lo que he aprendido de él. Y le agradezco a Vicente Sierra, por enseñarme a tener paciencia y enseñarme a ser yo mismo. Le agradezco a ambos por ser los mejores amigos que nunca pensé tener.

A todos mis grupos de amigos, a mis amigos de Kunst, al Seba, el Kuncar, la Ailín, el Adriano, la Antonia y al Vice nuevamente, por su amistad, y todos los buenos tiempos que hemos pasado desde el colegio. Al grupo de D&D !d20, a la Furst, al Jumba, al Guga, a la Jose y a la isa, por todas las risas y por acompañarme a jugar mi pasatiempo favorito.

A los de la generación pandemia, especialmente al Varo por habernos enfrentado a todos los trabajos de la u, al Seba, quien me inspiró por su fascinación por la electrónica. A todos, porque no pude haber pedido un mejor grupo con el que enfrentar las adversidades de la universidad.

A los integrantes del lab VLSI durante el tiempo que estuve, a la Caro por su disposición a ayudarme y por inspirarme por su entusiasmo por el área de digitales. A la Yaime, por su apoyo en esta última etapa de la memoria. Al Joaquín, por apoyarme en todas mis locuras y apoyarnos durante estos últimos proyectos. Al profesor Miguel Figueroa, por apoyarme, guiarme y enseñarme en esta última etapa. A todos los del lab que no pude mencionar, gracias.

Sumario

El rápido crecimiento de Internet ha conllevado a un aumento en el volumen de datos que se generan, haciendo el monitoreo de tráfico un aspecto importante para la seguridad en redes. Una de las métricas que caracteriza el tráfico es el cálculo de cuantiles, la cual permite identificar patrones y anomalías en la distribución de los datos. Sin embargo, las altas velocidades de las redes actuales, la necesidad de procesamiento de grandes volúmenes de datos en tiempo real, donde los datos más recientes son importantes para predecir el comportamiento de la red representa desafíos significativos. Ante tales desafíos, restringir el análisis a una ventana de observación permite reducir la complejidad de las soluciones y priorizar el monitoreo de los datos más recientes. Esto motivo la implementación de un algoritmo de cálculo de cuantiles en un acelerador hardware basado en FPGA (Field Programmable Gate Array) que es capaz procesar un flujo de paquetes de forma continua, pero la memoria interna del acelerador limita la cantidad de datos que puede procesar. Frente a esta problemática, utilizar algoritmos basados en sketches, los cuales son estructuras probabilísticas las cuales utilizan un espacio de memoria reducida para estimar una propiedad con una cota de error, resulta en una solución atractiva.

Este trabajo presenta el diseño e implementación de un acelerador hardware basado en FPGA para la estimación de cuantiles, utilizando el algoritmo KLL± presentado por Zhao et al. [1], un sketch que admite tanto inserciones como eliminaciones, permitiendo se ajuste al comportamiento de una ventana deslizante. Se realizó la implementación de la arquitectura en una tarjeta de desarrollo Alveo U280, obteniendo un uso de memoria de 0.6 kB (kilo bytes). Se validó el diseño con trazas de red reales (MAWILab y CAIDA), logrando un error de estimación menor al 0.1% y una frecuencia de operación de 219 MHz, equivalente a una tasa de 112 Gbps (Gigabits por segundo). Los resultados demuestran que el acelerador cumple con los requisitos de precisión ante el número de paquetes para el cual fue diseñado.

Summary

The rapid growth of the Internet has led to an increase in the volume of generated data, making traffic monitoring an important aspect of network security. One of the metrics used to characterize traffic is quantile computation, which helps identify patterns and anomalies in data distribution. However, the high speeds of modern networks and the need to process large volumes of data in real time—where the most recent data is crucial for predicting network behavior pose significant challenges. Given these challenges, restricting the analysis to an observation window helps reduce the complexity of solutions and prioritize monitoring the most recent data. This motivated the implementation of a quantile computation algorithm on an FPGA (Field-Programmable Gate Array)-based hardware accelerator capable of processing a continuous packet stream. However, the accelerator's internal memory limits the amount of data it can process. To address this issue, using sketch-based algorithms probabilistic structures that employ reduced memory space to estimate a property with a bounded error provides an attractive solution.

This work presents the design and implementation of an FPGA-based hardware accelerator for quantile estimation, using the KLL± algorithm proposed by Zhao et al. [1], a sketch that supports both insertions and deletions, allowing it to adapt to a sliding window behavior. The architecture was implemented on an Alveo U280 development board, achieving a memory usage of 0.6 kB (kilobytes). The design was validated using real network traces (MAWILab and CAIDA), achieving an estimation error of less than 0.1% and an operating frequency of 219 MHz, equivalent to a throughput of 112 Gbps (Gigabits per second). The results demonstrate that the accelerator meets precision requirements for the intended number of packets.

Tabla de contenidos

Lista de figuras

Lista de tablas

Lista de algoritmos

1. Introducción

1.1. Introducción general

El rápido crecimiento de Internet ha conllevado a un aumento en el número de dispositivos interconectados, haciendo dificultoso el manejo del creciente volumen de datos generando vulnerabilidades en la red [2]. Entre las amenazas a la seguridad en redes, destacan las interrupciones, modificaciones y los ataques de denegación de servicio (DoS). Ante este tipo de riesgos, resulta fundamental aplicar técnicas de seguridad en redes; entre ellas, el monitoreo del flujo de paquetes de red, el cual permite caracterizar el tráfico y detectar patrones anómalos que puedan indicar un comportamiento malicioso o un fallo en el sistema [3].

Una de las métricas que se puede utilizar para caracterizar el comportamiento del tráfico en redes es el cuantil. Dado un set de n elementos $s_1, \dots, s_n$, donde el rank de s_i es el número de elementos en el set menores o iguales a s_i , siendo dicho rank $R(s_i)$, el cuantil q de s_i corresponde a $q = \frac{R(s_i),}{n}$ [1] .

El monitoreo de los paquetes de red para la detección de anomalías representa un desafío, debido a la necesidad de poder procesar un gran volumen de paquetes de red en tiempo real sin reducir el rendimiento de los dispositivos de red. Para resolver dicha problemática, se pueden emplear aceleradores de hardware los cuales no utilizan los recursos de los dispositivos y permiten analizar estos flujos de datos en tiempo real. Entre los aceleradores de hardware que existen, uno de los accesibles y atractivos son los FPGA (Field Programmable Gate Array) debido a su flexibilidad gracias estar compuestos de bloques lógicos cuya interconexión es programable y además proveen una alta velocidad de cómputo [4]. Estos chips permiten ejecutar algoritmos de manera rápida gracias a su paralelismo y alta frecuencia de operación. Aun así, el desempeño de un FPGA está limitado por la memoria disponible en cada chip, lo que limita la implementación de un algoritmo que procese un gran volumen de datos, es por ello por lo que se busca una alternativa por un algoritmo que utilice una menor cantidad de memoria para el cálculo de ranks.

Para solucionar dicha problemática, se pueden utilizar algoritmos conocidos como sketches, los cuales son un tipo de estructura de datos basados en hash, que pueden procesar una gran cantidad de

datos utilizando una pequeña cantidad de memoria, pero obteniendo una alta precisión de estimación [5]. Aparte se utilizan técnicas de "sampling" o muestreo de los datos, donde se reduce la cantidad de datos que se procesa utilizando una muestra representativa para reducir la memoria utilizada. Aunque esta solución sí muestra una mejora frente a la memoria utilizada, estas suelen depender del hardware de los servidores donde se implementan, lo cual genera una sobrecarga significativa al uso de recursos del servidor que ralentiza el rendimiento de este.

Con todo lo presentado anteriormente, se propone la implementación del algoritmo KLL± en un acelerador por hardware, específicamente sobre una FPGA, para realizar la estimación de ranks en tiempo real. El algoritmo KLL± es un algoritmo que a través de un componente llamado compactores, los cuales almacenan elementos, el algoritmo es una extensión del sketch KLL que soporta tanto la inserción como la eliminación de datos, lo que lo hace particularmente adecuado para flujos de datos dinámicos, como los que se encuentran en redes de datos.

En el resto de este capítulo se presentan los objetivos generales y específicos del trabajo, su alcance y limitaciones. En el capítulo 2 se presenta un análisis bibliográfico sobre sketches para la estimación de cuantiles e implementaciones en hardware cuyos algoritmos están basados en sketches. En el capítulo 3 se presenta la descripción del algoritmo a implementar. En el capítulo 4, se presenta la descripción de la arquitectura hardware. En el capítulo 5 se eligen los parámetros que se utilizaron en la implementación. La validación, uso de recursos y resultados obtenidos se presentan en el capítulo 6.

1.2. Objetivos

1.2.1. Objetivo General

Diseñar un acelerador hardware para estimación de cuantiles de propiedades de flujos de redes usando algoritmos basados en sketches que permitan inserciones y eliminaciones.

1.2.2. Objetivos específicos

- Diseñar la arquitectura del acelerador hardware.
- Escribir una implementación RTL del acelerador.
- Implementar y validar el acelerador sobre un dispositivo FPGA.
- Utilizar el acelerador para estimar propiedades estadísticas del tráfico.

1.3. Alcances y limitaciones

- Se utilizarán los dispositivos disponibles en el laboratorio VLSI para la implementación.
- Para la validación del acelerador, se utilizarán trazas de tráfico disponibles en repositorios públicos en vez de tráfico en línea.

1.4. Productos y resultados esperados

- Se entregará un informe de memoria de título explicando los procedimientos de diseño, implementación y prueba de solución.
- El diseño de la arquitectura de un acelerador hardware para estimación de cuantiles que permite inserciones y eliminaciones.

2. Revisión bibliográfica

El interés por resolver el problema de estimar cuantiles reduciendo el uso de memoria, y aumentando la precisión ha generado variados trabajos algoritmos basados en sketches.

2.1. Sketches para estimación de cuantiles

En [6] se presenta el KLL (Karnin, Lang, Liberty) permite obtener con una precisión ε los ranks y cuantiles con probabilidad de estimación δ, con una utilización de espacio de $O\left(\frac{1}{\varepsilon} log^2 log \frac{1}{\varepsilon}\right)$ en el caso fusionable. El funcionamiento del sketch es el siguiente: consiste en una jerarquía de compactores, los cuales almacenan valores y tienen un peso respectivo. Cada compactor tiene una altura asociada, donde H es el compactor de mayor altura, y el peso de cada compactor se obtiene como $\omega = 2^{h-1}$ donde h es la altura de cada compactor. Cada vez que se llena un compactor, la mitad de los valores internos ingresan al siguiente compactor en la jerarquía con cierta probabilidad, y cada vez que se llena el compactor más alto en la jerarquía, se agrega un nuevo compactor del tamaño máximo de capacidad y se reduce la capacidad de los compactores de menor jerarquía. Cuando un compactor alcanza su tamaño mínimo 2, puede ser reemplazado por un sampler. Esta estructura cumple el mismo rol que dicho compactor, al insertar un único elemento aleatorio entre $2^{H``}$ al sketch, donde H'' corresponde al número de compactores reducidos al tamaño mínimo. Para estimar los ranks el algoritmo primero cuenta el número de elementos en cada compactor que son menores o iguales a la consulta, y multiplica este contador con el peso del compactor.

En [7] se realiza una mejora de sketch KLL donde se utiliza un error multiplicativo para obtener el rank. Los autores discuten que su principal problema es que lograr garantías multiplicativas es más difícil que las garantías aditivas. La contribución de sus autores es modificar la manera en que se seleccionan el número de elementos protegidos en cada compactación, seleccionándolos aleatoriamente mediante una distribución exponencial. Sobre esta base, proponen un algoritmo que se basa principalmente en un compactor relativo, el cual procesa un flujo de n elementos y produce una salida de n/2 elementos que aproximan el flujo.

2.1.1 Sketches para estimación de cuantiles que permiten eliminaciones

Una mejora realizada a los sketches de estimación de cuantiles es la adición de incluir eliminaciones. Esta mejora se realizó para evitar datos obsoletos y, por lo tanto, obtener una representación más precisa de la estimación.

En [1] se presenta el sketch KLL±, el cual se basa en el sketch KLL con la adición de permitir eliminaciones. Para ello se utiliza la misma jerarquía de compactores que en [6], con la modificación de añadir a la etapa de compactación una etapa de eliminación. Aparte se aumenta el tamaño mínimo de los compactores de menor jerarquía a 3, para evitar perder eliminaciones en la compactación. Con respecto al sampler, se utilizan dos samplers, uno que maneja las inserciones y otro para las eliminaciones. Aparte, los autores proponen que utilizando tres sketches KLL± se puede implementar una ventana deslizante. El sketch utiliza un espacio de $O(\frac{\alpha^{1.5}}{\varepsilon} \log^2 \log(\frac{1}{\varepsilon\delta}))$, donde el parámetro δ es una pequeña constante que denota el máximo error probable, el parámetro ϵ es la precisión que tiene el sketch y α acota la proporción entre eliminaciones e inserciones. Los autores, para poner a prueba el sketch, utilizan set de datos generados de manera aleatoria y comparan los resultados con el sketch KLL.

En [8] se propone el sketch SpaceSaving±, el cual es un algoritmo determinístico que estima frecuencia, pero los autores afirman que se puede utilizar para estimar cuantiles. El funcionamiento general del sketch consiste en guardar elementos insertados e incrementar un contador de frecuencia cada vez que éste se repite, donde si la memoria está llena y llega un nuevo dato que no se ha guardado, este se intercambia por el elemento más pequeño presente en la memoria, además de guardar el contador de frecuencia relacionado al elemento eliminado en un contador de error con respecto al nuevo elemento. Mientras los contadores de frecuencia se utilizan para determinar la frecuencia de los datos presentes en el sketch, cuando se requiere obtener la frecuencia para un elemento no presente, se utiliza la suma de todos los contadores de errores como límite superior para el cálculo de frecuencia de estos elementos no presentes con un error aditivo de $\frac{\varepsilon}{2}(I - D)$ con una cantidad $\frac{2\alpha}{\varepsilon}$, donde I es la cantidad de inserciones y D las eliminaciones. Con respecto al proceso de eliminaciones, si el elemento a eliminar está presente en el sketch, reduce su contador de frecuencia y si no está, se reduce el contador de frecuencia y el de error al ítem con el mayor contador de error. Para poner a prueba el

sketch, utilizan dos conjuntos de datos para medir el desempeño se usa el error cuadrado medio, el recall, la precisión y la divergencia Kolmogorov-Smirnov.

2.2. Implementación en hardware de sketches para estimación de cuantiles

Los algoritmos mencionados en la sección 2.1 fueron desarrollados para ser implementados en software, pero como mencionamos anteriormente, se busca una implementación en hardware, para poder procesar los paquetes en tiempo real y evitar usar los recursos como servidores y otros dispositivos de red.

En [9] se realiza una implementación del sketch KLL en un acelerador hardware basado en FPGA los autores realizan modificaciones a este algoritmo, donde particularmente, en vez de que el sketch sea creado dinámicamente, se usa una estructura estática que refleja el algoritmo luego de que se hayan insertado un numero de inserciones, ellos proponen esta modificación debido a la naturaleza estática del hardware. Para los compactores, utilizan un algoritmo de clasificación en el lugar para la inserción de elementos, donde estos son ordenados a medida que van ingresados en el compactor. Para la compactación, se necesita un bit de origen aleatorio así que implementan un LFSR, el cual es un registro de desplazamiento con realimentación lineal, el cual genera palabras pseudoaleatorias de bits. Para la estimación de ranks, los autores la realizan a través de dos etapas, donde cada compactor simultáneamente cuenta el número de valores almacenados que son menores o iguales a los valores requeridos y luego para calcular el peso se desplazan estos valores con respecto a la altura de cada compactor y luego se suman.

En cambio, en [10] se discute un algoritmo basado en el algoritmo de KLL, donde implementa un algoritmo Qpipe en un switch programable que le permite obtener cuantiles directamente en el plano de datos, lo cual le permite monitorear los paquetes mientras son recibidos. Los autores proponen que el desafío de esta implementación es el encontrar estadísticos de orden en switch programables, ellos también proponen que los cuantiles y la función de distribución acumulativa ayudan a entender la estructura del tráfico de red y detectar tantas anomalías a corto plazo y cambio en la distribución a largo plazo. Los autores discuten una serie de mejoras al sketch KLL que se han realizado, donde se menciona una mejora donde se utiliza una técnica de barrido, que de amortiza el tiempo de funcionamiento del sketch KLL. Otra mejora que se discute es separar los compactores a la mitad,

y comprimir un par de compactores a la vez mientras al mismo tiempo se mantiene un nuevo par de compactores que no intercepten con los comprimidos.

Con respecto a la evaluación del algoritmo, los autores utilizan tres trazas, una de las evaluaciones principales evaluaciones fue el encontrar heavy hitters, los cuales son los elementos que superan un límite establecido. donde el algoritmo los encuentra al calcular la proporción aproximada de cada ítem. Cuando la memoria puede almacenar 100 elementos, la proporción de falsos positivos es aproximadamente 66.67% para golpeadores pesados, mientras que la proporción de positivos verdaderos es 43.75%, donde en resumen los autores mencionan que, dado suficiente memoria de 100 mil elementos, el algoritmo puede encontrar efectivamente todos los 0.01-golpeadores pesados.

2.3. Discusión

A partir de la revisión de los distintos enfoques de estimación de cuantiles, se observa que el algoritmo KLL± presenta ventajas relevantes frente a otras alternativas. A diferencia del KLL original, este sketch admite tanto inserciones como eliminaciones, lo que permite tener implementar una ventana deslizante, permitiendo la caracterización del tráfico actual.

Con respecto a dónde implementar el sketch, mientras que los switches programables permiten obtener cuantiles directamente en el plano de datos, estos enfrentan una falta de flexibilidad. Por el contrario, una implementación en un (FPGA) ofrece mayor versatilidad ya que permite aprovechar el paralelismo inherente al hardware, y ajustar de manera precisa el balance entre memoria y velocidad. Estas características hacen posible cumplir con los requisitos de procesamiento en tiempo real en redes de alta velocidad, sin depender de las restricciones impuestas por los entornos de conmutación programable como los switches programables.

3. Descripción del Sketch KLL±

A continuación, en este capítulo se describe la arquitectura y los distintos algoritmos que componen el sketch KLL, se describen su estructura, operan de compactación, inserción de elementos y la estimación de los ranks.

3.1. Estructura del sketch KLL±

El sketch KLL± [1] está basado en el sketch KLL cuya arquitectura se compone de compactores, los cuales guardan elementos de manera ordenada, y están organizados jerárquicamente, esta jerarquía luego se utiliza para obtener el rank.

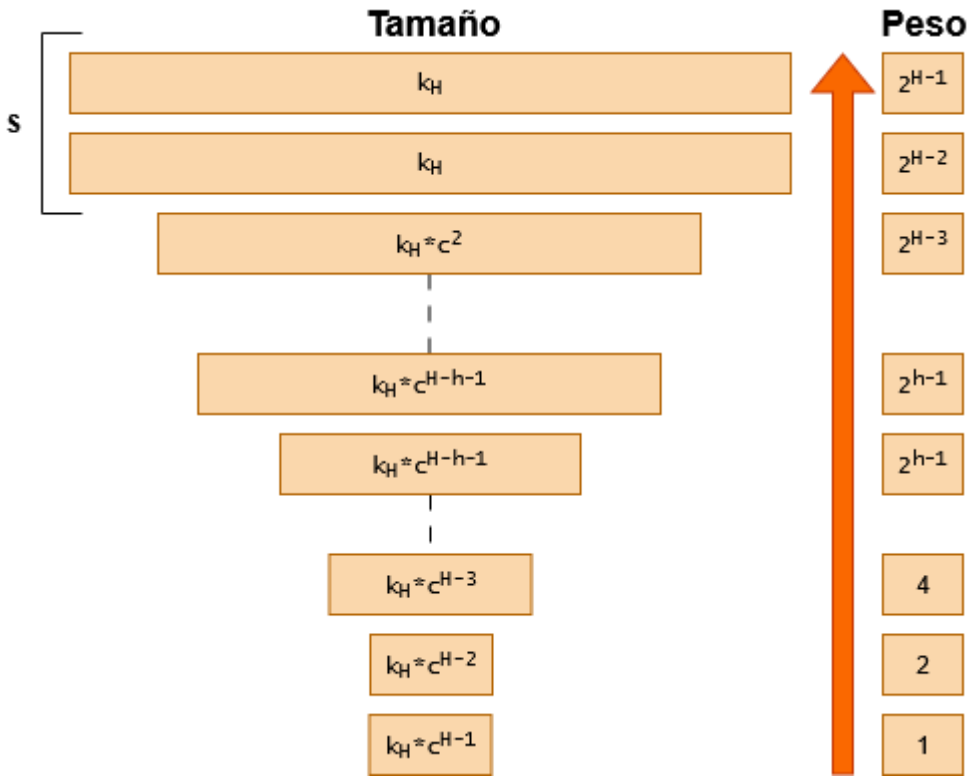

Figura 3.1: Estructura general del Sketch KLL±

Como se puede observar en la figura 3.1, cada compactor tiene dos características, tienen una altura h que define su posición en la jerarquía, donde el mayor compactor tiene tamaño H y el compactor de menor tiene una altura de 1. La segunda característica es la cantidad de elementos que puede guardar cada compactor, la cual está representada por el valor de c_h. Para calcular este valor se tiene la siguiente formula:

$$c_h = kc^{H-h},$$

donde c es el factor de reducción de la capacidad y k es el tamaño de los $s = O(\log\log\left(\frac{1}{\delta}\right))$ compactores de mayor altura, siendo esta el tamaño máximo y se calcula de la siguiente manera:

$$k = \frac{(2\alpha - 1)^{1.5}}{\epsilon\sqrt{\log\left(\frac{1}{\epsilon\delta}\right)}},$$

donde α acota la proporción entre inserciones (I) y eliminaciones (D), cuya relación está acotada por $D \leq \left(1 - \frac{1}{\alpha}\right) I$ con $\alpha \geq 1$.

Con respecto a las alturas de los compactores, la altura H va aumentando a medida que más elementos se insertan y con este aumento en el número de compactores también se va reduciendo el tamaño de cada compactor hasta llegar a un tamaño de 3.

3.2. Compactación

Cuando la cantidad de elementos que contiene un compactor es igual a su capacidad máxima k_h, este debe liberar espacio para poder recibir nuevos elementos. Para ello, éste envía la mitad de estos elementos al compactor superior. Este proceso tiene por nombre compactación, cuyo proceso consiste en dos etapas: la etapa de eliminación, donde se busca entre los elementos del compactor si hay un elemento de eliminación de igual magnitud al de inserción, si esto se cumple, se eliminan ambos elementos del compactor y no se continúa el proceso compactación, ya que el compactor no sigue a capacidad máxima. La segunda etapa es la compactación, donde, como se puede observar en

el algoritmo 1, se elige de manera aleatoria un offset donde, si el valor del offset es 0, se eligen los valores impares, y si el valor del offset es 1, se eligen los valores pares. En el algoritmo se van comparando dos elementos continuos y con respecto al offset se van moviendo al compactor superior, pero si ocurre el caso donde se compara una inserción con una eliminación, se mantienen dichos elementos en el compactor. Esto evita perder elementos como argumentan los autores en [1].

Cada vez que el compactor de mayor altura se compacta, se crea una copia de igual tamaño con altura h + 1 y se reduce el tamaño del resto compactores de menor altura con respecto a c. Como se mencionó en la sección 3.1, el tamaño mínimo de los compactores es 3; este tamaño permite evitar la pérdida de eliminaciones.

Algoritmo 1: Algoritmo de compactación para un compactor de altura h

```
Sort(Compactor_list[h]);
Discarded = Elimination ((-x,x) pairs in Compactor_list[h]);
if Discarded then
    break; //No se descarta debido a que el compactor deja de estar
lleno
Offset = random(0,1);//Selección del offset entre 0 y 1
//Proceso de compactación
for (i=0; i < Compactor_list[h].length; i = i+2) do
    if sign(Compactor_list[h][i] == Compactor_list[h][i+1]) then
        Push(Compactor_list[h][I+offset]to Compactor_list[h+1]
//Envia el elemento
    else
       //Se mantienen las parejas de signos distintos
       Keep(Compactor_list[h][i], Compactor_list[h][i+1]) in TEMP
end
Compactor_list[h].clear();
Compactor_list[h] ← TEMP;
return
```

Tabla 3.1 Compactación de compactor de tamaño 3

Elementos guardados en un compactor de tamaño 3	Resultado de la compactación
3 inserciones o 3 eliminaciones	Mantiene un elemento en el compactor y transfiere al compactor superior un elemento
2 inserciones y 1 eliminación	Mantiene la eliminación en el compactor y transfiere al compactor superior una inserción
1 inserción y 2 eliminaciones	Mantiene la inserción en el compactor y transfiere al compactor superior una eliminación

3.3. Inserción de elementos

Los nuevos elementos entran al sketch a través del compactor de menor altura. Mientras más elementos se insertan al sketch, más compactores se crean y la cantidad de compactores de tamaño 3 aumenta en número; los autores proponen el reemplazo de estos compactores por dos samplers, un sampler para inserciones y otro para eliminaciones. Esto ocurre cuando el compactor de menor tamaño reduce su tamaño a 3; Este es reemplazado por dos samplers, ambos de altura H'' y a medida que más compactores reducen su tamaño a 3, estos en vez son eliminados y la altura correspondiente a ambos sampler aumenta.

Como se puede observar en la tabla 3.1, se tiene que los compactores de tamaño mínimo H'' solo transfieren un elemento cada vez que compactan, debido a este comportamiento, se puede reemplazar los compactores de tamaño mínimo por dos sampler, un sampler que maneja inserciones y un sampler que maneja eliminaciones. En este trabajo nos basamos en el funcionamiento interno del sampler definido en [6], esto debido a errores [1], no obstante, los mismos autores mencionan que se basaron en el sampler del sketch KLL para implementar este sketch.

Cada vez que se ingresa un nuevo dato, éste entra con un peso w. Si la suma del peso nuevo

w y el peso interno v es menor o igual a $2^{H''}$, el elemento interno del sampler es reemplazado por el nuevo elemento con una probabilidad de $\frac{w}{w+v}$. No obstante, sin importar si se cumple o no la condición de reemplazar el nuevo valor, el nuevo peso interno es igual a la suma del peso interno anterior más el nuevo peso ingresado. Aparte del caso anterior, si la suma del peso interno y el peso del nuevo elemento es igual a $2^{H''}$, se inserta el elemento ingresado (antes de ser reemplazado por el nuevo valor como sucede en el caso anterior) al compactor de altura H'' + 1, o sea, el compactor de arriba. El último caso que puede ocurrir con un nuevo elemento en el compactor es el caso donde la suma de ambos pesos sea mayor a $2^{H''}$, en ese caso se descarta el elemento con el mayor peso y se mantiene el elemento cuyo peso es el menor. El elemento con mayor peso puede ingresar al compactor de mayor peso con probabilidad $\frac{\max(v,w)}{2^{H''}}$.

3.4. Estimación de ranks

El objetivo principal del sketch es el cálculo del rank. Para ello, se tiene una consulta de un elemento x, y se debe comparar dicho elemento con cada elemento en cada compactor. Para calcular el rank de un elemento se utiliza un algoritmo que, por cada ítem por cada compactor en el sketch, calcula el peso de cada compactor. En [1] se establece que dicho peso corresponde a la siguiente formula:

$$w_h = 2^{h-1},$$

donde w_h es el peso del compactor de altura h; luego de calcular el peso del compactor, lo multiplica por el signo del ítem, lo guarda en un mapa que contiene cada ítem positivo y finalmente suma el peso en un contador. Luego, para calcular los cuantiles, se utiliza el mapa de pesos ordenados, donde para calcular el cuantil respectivo de cada elemento, se suma el peso respectivo del elemento y se le suma el peso acumulado anterior y se divide por el peso total.

Algoritmo 2: Algoritmo de estimación de cuantiles

```
Result = map();
//Mapa de item por magnitud en orden
ItemWeightSortedMap = OrderedMap< item, weights >
TotalWeight = 0;
for all compactors do
    For all items in compactor[h] do
        Weight = sign(item) *
        ItemWeightSortedMap[(abs(item))] += weight;
        TotalWeight += weight
    End
End
//Peso acumulado
PrevW = 0
//Revisar el vector de ítems ordenados en orden ascendente
for < item, weights > in ItemWeightSortedMap do
    Result[item] = (weights + PrevW)/TotalWeight;
    PrevW += weights;
End
//El resultado son los cuantiles, si se multiplica por TotalWeight, se obtienen
los ranks
return Result;
```

4. Arquitectura del acelerador

El sketch KLL± está compuesto por elementos dinámicos, pero para la arquitectura del acelerador hardware se tomó la decisión de diseñar el acelerador sin los elementos dinámicos para mantener la naturaleza estática del hardware.

4.1. Arquitectura principal

Para implementar el sketch, como ya se mencionó anteriormente, para quitar los elementos dinámicos se decidió usar la estructura resultante después de una cantidad determinada de inserciones, esto implica que la arquitectura se compone de H – H'' compactores y un sampler de altura H''. Todos estos parámetros se discuten en la siguiente sección. Para los compactores se tomó la decisión de usar memorias tipo Block RAM. Para simplificar y facilitar el proceso, se decide usar dos Block RAM, una para elementos tipo inserción y la otra para elementos tipo eliminación, esto permite simplemente usar magnitudes. También permite acceder en un mismo ciclo a elementos negativos y positivos. Aparte de los compactores, se incluye en la estructura dos módulos de generador de números pseudoaleatorios, uno para el elemento aleatorio del sampler y otro para la compactación de los compactores. También para evitar perder elementos durante las distintas etapas del compactor, se decidió incluir una cola FIFO, cuyas siglas provienen del inglés first input first output, donde el primer elemento en ser guardado es el primer elemento en ser leído. Está memoria permite evitar la pérdida de elementos mientras los como actores están ocupados compactando. Finalmente, se implementó también un módulo de estimación de rank.

En la figura 4.1 se observa la estructura general, donde se observa un ejemplo de 5 compactores y ambos samplers. Para la implementación, como se puede observar, se tienen dos entradas de elementos, una para inserciones y otra para eliminaciones. La arquitectura fue diseñada basándose en cómo funciona una ventana móvil, más específicamente cómo funciona la ventana móvil presentada en [1], la cual es una implementación alterna del sketch KLL±. En esta implementación alterna se insertan la mitad de los elementos y luego se insertan la otra mitad mientras se eliminan los primeros elementos. Es por eso por lo que se diseñó con esto en mente, donde primero solo hay inserciones hasta llegar a la mitad de los datos y luego comienzan las eliminaciones e inserciones, alternándose hasta que se acaben los datos.

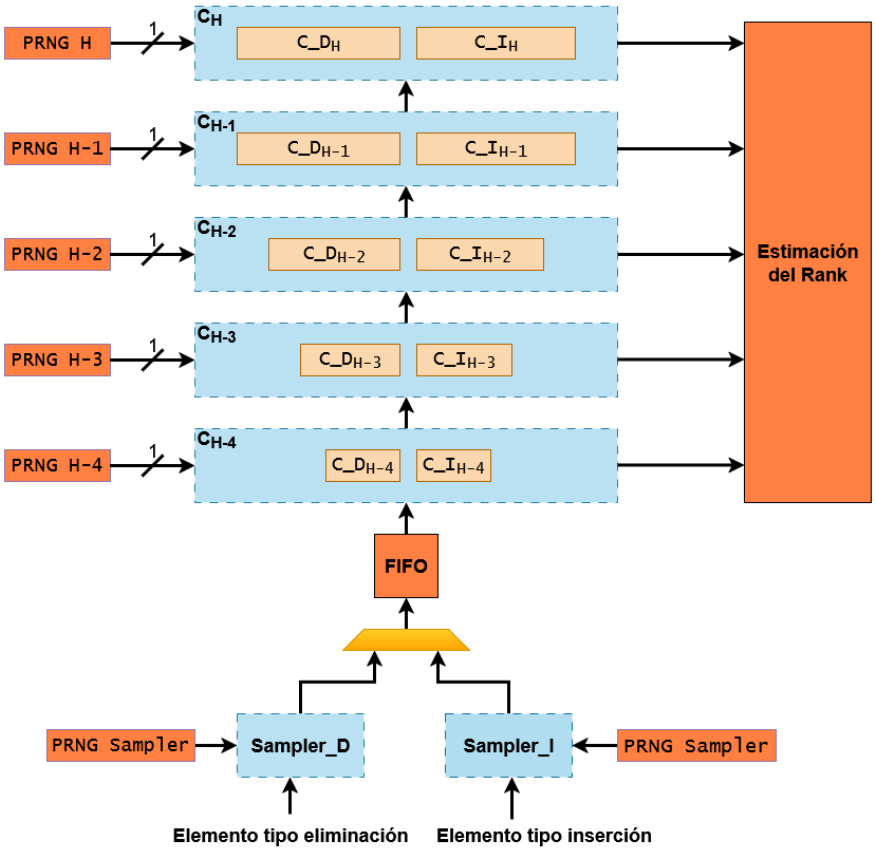

Figura 4.1 Arquitectura general del acelerador

4.2. Compactores

Como ya se mencionó anteriormente, los compactores están compuestos por dos Block RAM, el de elementos tipo inserción y el de elementos tipo eliminación. En esta sección, se utiliza el término compactor para referirse al conjunto de ambos vectores. Un compactor tiene 5 modos de funcionamiento:

1. Modo de inserción: inserta los nuevos elementos en las Block RAM de forma ordenada.

2. Modo de eliminación: compara los elementos tipo eliminación y tipo inserción en ambas Block RAM, eliminando aquellos que coinciden

3. Modo de reordenamiento: reorganiza los elementos tras el proceso de eliminación, eliminando los espacios vacíos resultantes.

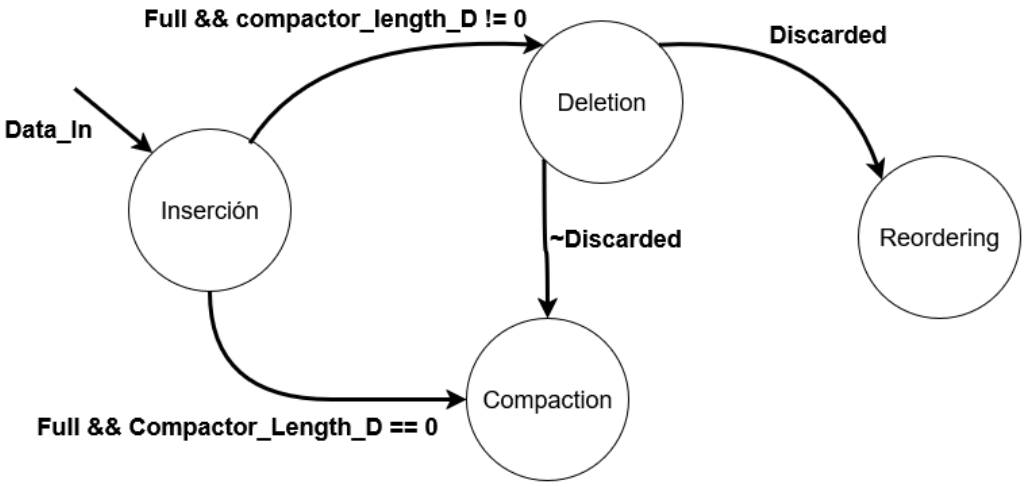

Figura 4.2: Interacción general entre los modos de funcionamiento del compactor

4. Modo compactación: transfiere los elementos de un compactor al compactor superior en la jerarquía.

5. Modo estimación: aplica una búsqueda binaria para determinar la cantidad de elementos menores o iguales al elemento consultado.

4.2.1 Inserción

Con respecto a la etapa de inserción de elementos al compactor, se tienen dos modos: el primero, el modo inserción, es cuando no hay elementos en el compactor y se insertan los elementos directamente. El segundo, modo de ordenamiento, donde se decidió mantener los elementos ordenados de menor a mayor con el algoritmo in-place que se observa en el algoritmo 3, para la implementación del ordenamiento, se basó en la implementación realizado en [11], incluyendo el algoritmo 3. Como se explicará en la sección de compactación, se envían los elementos de uno en uno y se insertan solo si hay espacio disponible. No obstante, este espacio disponible depende de la suma entre la cantidad de elementos entre ambas Block RAM. También es necesario reconocer si el dato entregado al compactor es de tipo I o tipo D, pero una vez insertado el dato, se guarda solamente su magnitud.

Para guardar los elementos se utilizó una Block RAM. Esto implica que los elementos están disponibles en el siguiente ciclo de reloj. Es por ello por lo que se decidió implementar como una máquina de estados, donde se decide si se inserta directamente, o si se debe buscar ordenar los

elementos.

Como se observa, en el algoritmo 3, se tienen dos casos, o el dato leído es mayor al dato a insertar y se mueven los elementos ya ubicados en la memoria, o se inserta el nuevo. En el primer caso se escribe en la memoria dos veces y en el otro caso solo una, por lo tanto, es necesario utilizar dos etapas, una donde escribe el dato leído en la dirección de escritura y otra donde se escribe un cero.

Aparte de ello, se necesita tener un registro del tamaño del compactor, un registro de la cantidad de elementos que se insertan y finalmente una señal que avise que hay nuevos elementos disponibles.

Algoritmo 3: Algoritmo sort in place

```
// Xin Es un arreglo con n elementos a insertar
//Se utiliza el Compactor I como ejemplo
wr_ptr = Compactor_I.length() + Xin.length - 1; //Se comienza escribiendo en el
primer espacio sin elementos
rd_ptr = Compactor_I.length()  - 1; //Se lee el mayor dato
Compactor_I.resize(Compactor_I.length()  + Xin.length);
Nleft = Xin.length;
While nleft > 0 do
    If Compactor_I[rd_ptr] ≥ Xin[nleft] then
        // Se reordenan los elementos para poder insertar el dato
        Compactor_I[wr_ptr] = Compactor_I[rd_ptr]; //Se escribe el dato leído
        Compactor_I[rd_ptr] = 0;
        wr_ptr = wr_ptr -1;
        rd_ptr = rd_ptr – 1;
    Else then
        Compactor_I[wr_ptr] = Xin[nleft]; //Se escribe el nuevo dato
        Nleft = Nleft – 1;
        Wr_ptr = rd_ptr + nleft;
    end
end
return;
```

4.2.2 Eliminación

Una vez que el compactor está lleno con elementos, es necesario compactar el compactor, pero para ello primero debe pasar por el proceso de eliminación. Como se mostró en la figura 4.2, no es necesario pasar por esta etapa si solo hay elementos de un solo tipo. En el caso contrario, donde si hay elementos para eliminar, se deben leer ambas Block RAM.

La eliminación comienza cuando el compactor está lleno y hay elementos en ambas Block RAM. La forma en que funciona el algoritmo de eliminación es que se empieza con cuatro punteros, dos punteros de lectura y dos de escritura, un par para cada Block RAM. Estos punteros son inicializados en 0, donde se irán leyendo los elementos y buscando elementos iguales.

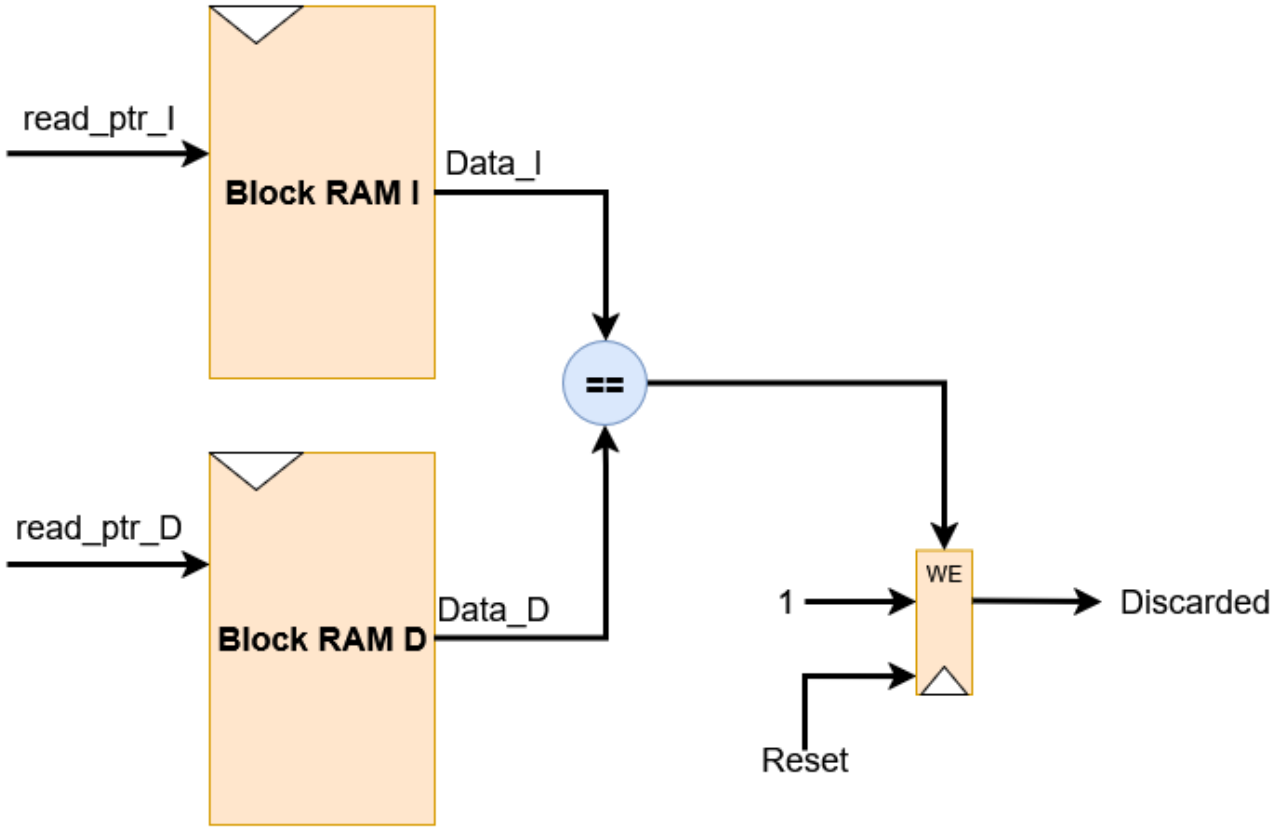

Figura 4.3: Circuito de identificación para eliminación

Como se observa en la figura 4.3; La forma en que se van buscando elementos es que primero se compara si son iguales; si son iguales, se aumentan ambos punteros y la señal Discarded se activa. Esta se mantiene prendida para que el compactor sepa que al finalizar la etapa debe pasar a la etapa del reordenamiento. También, ya que se eliminan elementos, se aprovecha de reducir el tamaño del compactor al momento en qué se elimina un dato al escribir en esa dirección un dato valor 0 en ambas Block RAM. Si los elementos no son iguales, se aumentan los compactores dependiendo de qué dato es mayor a otro. Si el elemento de la Block RAM I es menor al elemento de la Block RAM D se aumenta el puntero I y si es mayor, se aumenta el puntero D. De esa forma, se va recorriendo ambas Block RAM con respecto a la magnitud de los elementos y comparándolos.

Debido a que existe la posibilidad de que se llegue al final de una de las Block RAM y no de la otra, terminar este proceso no es tan simple como terminar cuando ambos punteros sean igual a la cantidad de elementos. Es por ello por lo que se tienen 3 condiciones en que se termina el modo de eliminación:

- Si el puntero de la Block RAM D es igual a la cantidad de elementos y el dato leído es menor que el dato leído de la Block RAM I.
- Si el puntero de la Block RAM I es igual a la cantidad de elementos y el dato leído es menor que el dato leído de la Block RAM D.
- Y si ambos punteros son igual a la cantidad de elementos de ambas Block RAM.

Una vez se termina el modo eliminación, se revisa el valor de `discarded` y se decide a qué etapa continuar. Si la señal `discarded` es 1, los elementos deben ser reordenados, modo que se revisa en el capítulo 4.2.3. Y si no se eliminaron, se compacta, modo que se revisa en el capítulo 4.2.4.

4.2.3 Reorden

En el caso donde se eliminaron elementos, ambas Block RAM deben ser reordenadas. El proceso de reordenado es simple: se inicializan los punteros en 0, y tal como se observa en la figura 4.4, se va aumentando el puntero de lectura y solo se aumenta el de escritura cuando un dato es distinto de 0. Con respecto a las señales, `read_enable` se mantiene activa y `write_enable` se activa

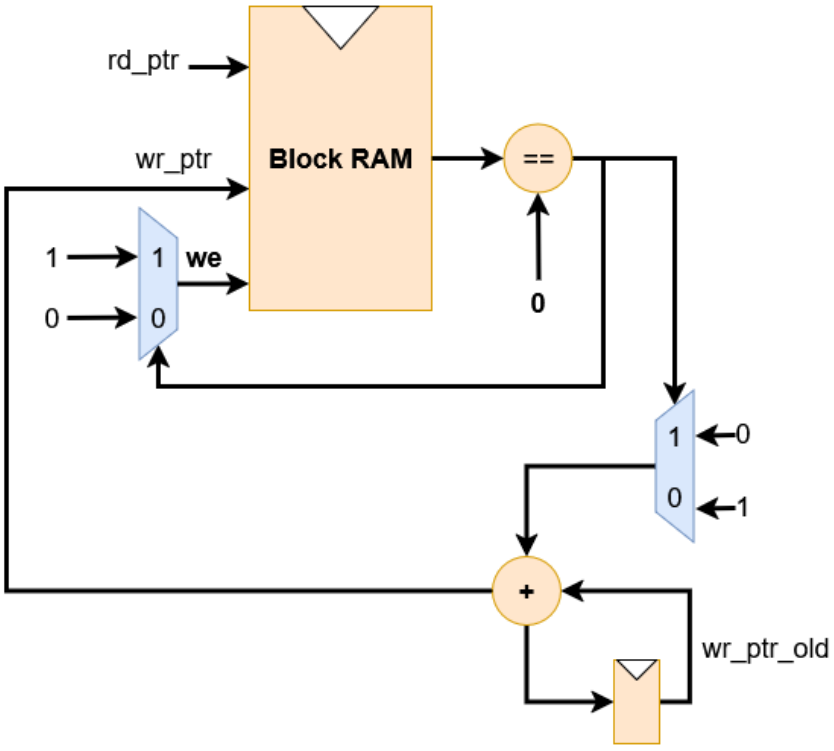

Figura 4.4: Circuito de punteros para reordenamiento

solo si un dato es distinto de 0.

4.2.4 Compactación

Cuando el compactor está lleno y no se descartó ningún elemento, se inicia el proceso de compactación, considerando el algoritmo N mencionado en [1], se necesita la cantidad de elementos de ambas Block RAM y un offset de origen aleatorio, pero el origen del bit de offset se discute en la sección 4.3.

Cuantos elementos deben ser enviados y si se debe mantener una pareja de elementos (una inserción y una eliminación) depende de la cantidad de elementos en cada Block RAM, por lo que tenemos 3 casos:

- Si solo hay elementos en una sola Block RAM, deben ser enviados la mitad de los elementos y se elimina el resto.

- Si hay elementos en ambas Block RAM, pero la cantidad de elementos tipo eliminación es una cantidad par, tan solo se debe enviar la mitad de la cantidad de elementos de cada Block RAM, por ejemplo, si el compactor es de tamaño 14 y la Block RAM D tiene 4 elementos y la otra tiene 10 elementos, se enviarán 2 elementos tipo D y 5 elementos tipo I.

- La tercera opción es donde la cantidad de elementos tipo D es impar, porque en ese momento se cumple la condición de mantener un dato tipo D y un dato tipo I. Basándose en cómo se comporta el algoritmo 1, los elementos que se deben mantener siempre son los de menor magnitud en ambas Block RAM, esto ya que en el algoritmo 1 se van leyendo el vector en base a parejas. Así que para ello se tiene el registro Keep, el cual, si es 1, se mantendrán dos primeros elementos de cada Block RAM, y en los otros casos mencionados anteriormente Keep será 0.

El siguiente registro que se debe calcular es la cantidad de elementos a enviar al siguiente compactor, para ello es simplemente la mitad elementos guardados en cada Block RAM. Teniendo la cantidad de elementos a enviar, falta elegir cuáles se van a enviar. Para ello, como dice el algoritmo 1, se deben elegir entre una pareja de datos con respecto a un offset, pero debido a que no se puede leer dos elementos de una Block RAM a la vez, se debe buscar una alternativa, para ello, como el ejemplo que se muestra en la figura 4.5, se observa que, sin importar el valor del offset, las posiciones de datos a enviar entre ambas Block RAM nunca coinciden. Siguiendo el ejemplo, si se leen ambas Block RAM desde el primer dato, si el offset es 0 solo se debe enviar el 7 y si el offset es 1 se envía el 4. Esto implica que se pueden tener dos etapas, una donde se envían inserciones y otra etapa donde se envían eliminaciones, y cuál se envía en cada etapa se elige a través del offset.

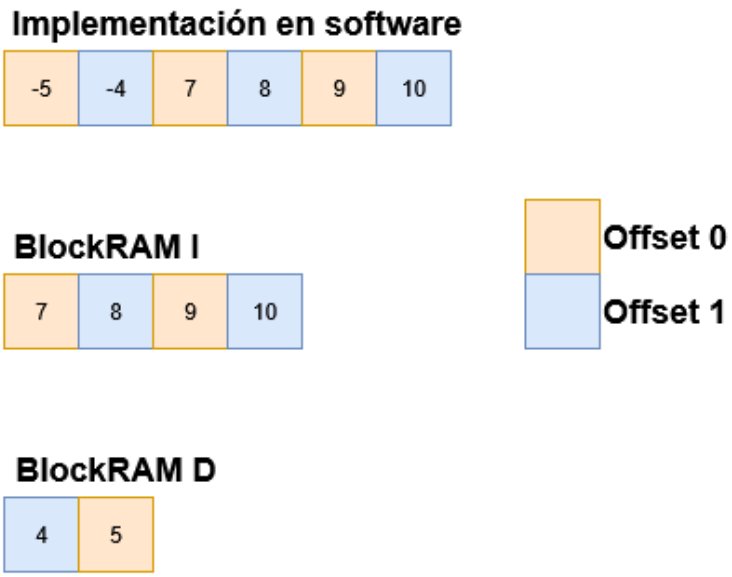

Figura 4.5: Ejemplo de selección de elementos a enviar al siguiente compactor

Para enviar elementos de un compactor a otro, este tiene que comunicarse con el compactor de siguiente altura, para ello se tienen las señales data_I y data_D, las cuales le avisan al compactor h+1 que tiene una inserción o eliminación para que reciba respectivamente. Cuando recibe uno de esos elementos, devuelve una señal Received, la cual avisa que se recibió el dato y se puede pasar a la siguiente etapa. Pasando a la siguiente etapa se debe esperar a que el compactor termine de procesar los elementos, puede ocurrir que el compactor h + 1 comience a compactar mientras el compactor h todavía tiene elementos para enviar, eso hace que el compactor h espere todo el proceso, por lo tanto, si el compactor h – 1 comienza a compactar también y el compactor h está ocupado esperando, también espera al compactor h, esto genera que todos los compactores esperen, es por ello que se implementó una cola FIFO, para guardar elementos mientras se espera que los compactores estén desocupados.

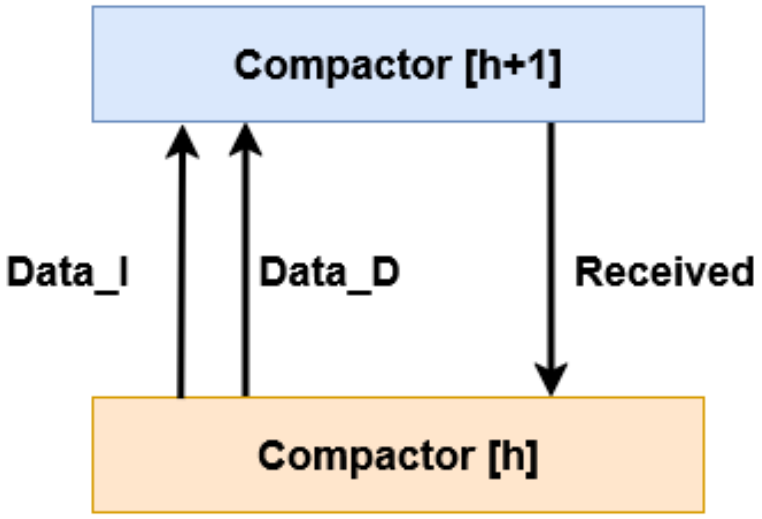

Figura 4.6: Interfaz de comunicación entre compactores

Una vez que se termina el proceso de compactación, en vez de borrar los elementos contenidos en las Block RAM, simplemente se mueve el puntero de tamaño a 0 o 1, dependiendo de Keep. Así simplemente, cuando nuevos elementos sean ingresados en la memoria, se reemplazan los elementos antiguos.

4.2.5 Estimación

Para la estimación de ranks, se envía una señal a cada compactor que indica que se realizará una consulta y se enviará el elemento a consultar; cuando los compactores reciban dicha señal, todos empezarán a buscar el elemento consultado, para ello, se aprovechó de que los elementos están ordenados y se utilizó una búsqueda binaria para encontrar el primer elemento mayor que el elemento consultado, ya que la ubicación de este dato es la cantidad de datos menores o iguales al dato consultado. Pero el algoritmo de búsqueda tradicional está pensado para que encuentre un valor exacto, entonces se realizó una pequeña modificación al algoritmo, como se observa en el algoritmo 4, para que la búsqueda se detenga en el siguiente elemento más cercano mayor al consultado. Este proceso se realiza en ambas Block RAM.

Algoritmo 4: Búsqueda binaria modificada

```
Input: query_element -> valor de 64 bits que se consulta
//Se utiliza el Compactor I como ejemplo
rd_ptr = 0;
Left = 0; //Puntero al inicio de la Block RAM
Right = compactor_lenght – 1; // Puntero al último dato de la Block RAM
While (Left < Right) do
    rd_ptr = (Right + Left) >> 1;
    if ( Compactor_I[rd_ptr] <= query_element) do
        Left = rd_ptr + 1;
    end else do
        Right = rd_ptr;
    end
End
return Left;
```

Una vez se termine el proceso de búsqueda, se resta el valor obtenido en la Block RAM I con el valor obtenido en la Block RAM D. En este caso, se utiliza el formato de magnitud signo, el cual utiliza un bit para indicar el signo de la palabra. Una vez el compactor obtiene el resultado final, le avisa al módulo de estimación que se tiene un resultado válido.

El módulo de estimación se basó en la arquitectura de [11]; en la arquitectura de estimación de la implementación del sketch KLL, se recibe el valor estimado de cada compactor, se multiplica por el peso del compactor respectivo y se va sumando a un registro; cuando se terminan de sumar todos los resultados de cada compactor, se indica que terminó la estimación. Para la implementación de este trabajo, se añadió que se sume o reste el resultado obtenido de cada compactor y luego se multiplica por su peso. Puede ocurrir que el resultado de la estimación sea negativo; para ello los autores en [1] indican que este resultado debe ser reemplazado por 0.

Como se puede observar, esta implementación de estimación de rank, no es similar a la propuesta en [1]. Se decidió usar este algoritmo sobre el propuesto ya que se desea poder consultar un dato a la vez y el algoritmo 2 está diseñado para obtener todos los ranks de una lista.

4.3. Generador de números pseudoaleatorios

Como se ha mencionado anteriormente, cada compactor requiere un bit aleatorio para la etapa de compactación. Asimismo, el sampler necesita una palabra de varios bits aleatorios para seleccionar un elemento a insertar a la estructura. Es por ello que es necesario implementar un generador de números pseudoaleatorios. Como desafío, se decidió utilizar un generador basado en un mapa caótico. Dicho generador está compuesto de una función con números resultantes de 64 bits donde solo se utilizan los 16 bits menos significativos. En este trabajo, se decidió implementar el mapa caótico implementado en [12], cuya función es la siguiente:

$$x_{n+1} = \lambda \times x_n \times (1 - x_n) + \mu$$

La principal debilidad de la implementación hecha en [12] es la velocidad del reloj obtenida, la cual es muy baja para la implementación del acelerador, es por ello que se decidió modificar la arquitectura añadiendo paralelismo, donde en cada etapa se realiza una sección de la aritmética. Debido a la cantidad de operaciones que se realizan, lo primero que se puede concluir es que se deben usar varias etapas para el pipeline, pero para este proceso no es un problema, pues como los mapas caóticos están siendo constantemente realimentados, no se debe esperar por todas las etapas por un valor válido, permitiendo esto ser una solución preferible a la implementación realizada en [12].

En [12], para implementar la multiplicación, se propone un algoritmo donde se multiplica un número de 64 bits, separado en cuatro palabras de 16 bits y una palabra de 72 bits, separada en 3 palabras de 24 bits, esto se realiza para que se pueda implementar en un DSP48E1 que tienen como input máximo palabras de ese largo. Pero también los autores proponen utilizar 64 bits para cada valor, es por ello por lo que, para multiplicar, se extiende el signo de una de las palabras de 64 bits.

Los autores también argumentan que, al usar 64 bits para ambos números, como el resultado esperado es de 128 bits, se trunca a 64 bits, pero como se mencionó anteriormente, uno de los argumentos de la multiplicación es de 72 bits, entonces se pueden descartar los 72 bits menos significativos. Todo esto resultaría en lo siguiente,

$$X * Y = 2^{96} \times X_a Y_w + 2^{72} \times X_b Y_w + 2^{48} \times X_c Y_w + 2^{80} \times X_a Y_x + 2^{56} \times X_b Y_x + 2^{64} \times X_a Y_y + 2^{64} \times X_a Y_y + 2^{40} \times X_b Y_y + 2^{48} \times X_a Y_z$$

donde X es una palabra de 72 bits separada en X_a, X_b, X_c, donde cada una representa 24 bits de X. Además, Y es una palabra de 64 bits compuesta por Y_w, Y_x, Y_y, Y_z, donde cada palabra representa 16 bits de Y. Con todo esto en consideración, se puede realizar cada multiplicación en una etapa y hacer las sumas de los resultados de la multiplicación en varias etapas, aparte como el desplazamiento es constante, se puede manejar los bits manualmente para no manejar una palabra de 64 bits completa.

Como se observa en la función del mapa caótico propuesto anteriormente, se tienen dos multiplicaciones, así que se debe utilizar la técnica de multiplicación dos veces, aparte se deben realizar dos restas.

El resultado de esta implementación es que PRNG consiste en 9 etapas. También se estableció una señal tipo enable en cada registro, para así, cada vez que un módulo necesite un valor aleatorio, puede pedirle a este módulo que genere uno nuevo.

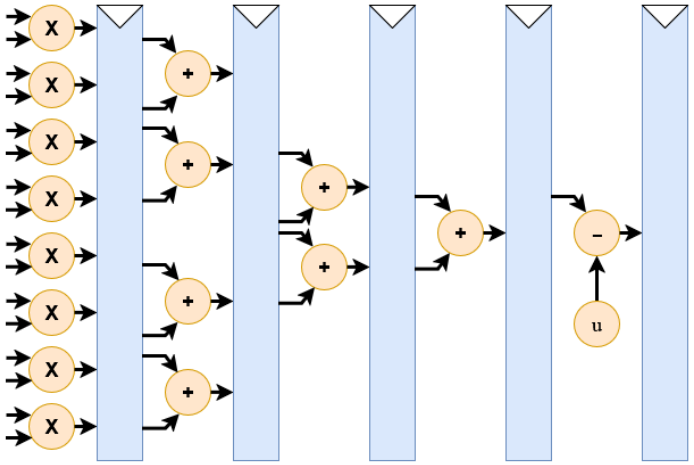

Figura 4.7: Fracción de pipeline implementado para la multiplicación de dos elementos

4.4. Sampler

Cómo fue explicado en el capítulo 3, cuando los compactores alcanzan la capacidad de almacenamiento mínima, pueden ser reemplazados por dos sampler de tamaño igual a los compactores de tamaño mínimo que reemplazo. Ahora, como esta implementación del sketch es una estructura estática por la cual ya pasaron N elementos, el tamaño del sampler también será estático, aparte, para esta implementación solo se considera la entrada de elementos con peso 1, en otras palabras, solo entraran elementos nuevos. El sampler consiste en dos operaciones principales, enviar un dato cada $2^{H''}$ elementos que intentan entrar al sampler y elegir aleatoriamente con respecto a $\frac{1}{1+v}$ que dato enviar entre esos nuevos elementos.

La primera operación se implementa a través de un simple contador, el cual le suma uno a un registro cada vez que se tiene un elemento válido, la cual avisa cuándo llega un nuevo dato. Con respecto a la segunda operación, se requiere realizar una división, pero la división no es una operación soportada en los FPGAs, por lo tanto, se decidió implementar una tabla LUT (Look up Table), cuya entrada es v+1 donde v es el peso actual que varía entre 0 y H´´, así que esta tabla contiene todos los posibles resultados de v + 1 en 13 bits en formato u1.12, lo que significa que se usa el primer bit para representar la parte entera y los otros doce bits para representar la parte fraccionaria.

Cuando el sampler recibe un nuevo dato, si el registro de peso v es igual a cero, se guarda ese valor en el registro del peso interno v; si es distinto a cero, se debe comparar el valor $\frac{1}{1+v}$ con el resultante de la PRNG. Una vez el peso v alcanza su máximo valor, se avisa al módulo de la cola FIFO que un nuevo dato está disponible. En el caso que la cola FIFO este llena, el elemento se perderá.

4.5. Cola FIFO

Debido a que los compactores tienen varias etapas, durante estas etapas los compactores no pueden recibir nuevos elementos, por lo tanto, es necesario mantener una memoria que guarde elementos para ser procesados luego, y así evitar la pérdida de nuevos elementos. En esta arquitectura se decidió utilizar una sola FIFO, debido a que el compactor que recibe los elementos nuevos es de capacidad 4 y no puede pasarse de esta capacidad, al usar dos FIFOs, una para elementos tipo eliminación y otro tipo inserción, sería necesario tomar decisiones de quién inserta primero y quién

después, en el caso de que el primer compactor tenga capacidad 3 de 4. Por lo tanto, se decidió usar una sola cola FIFO que acepta ambos tipos de elementos.

Debido a que debemos saber reconocer entre estos 2 distintos tipos de elementos, se crean dos módulos aparte de la cola FIFO: uno que recibe elementos de ambos sampler y envía los datos a la cola FIFO indicando de qué sampler recibió el elemento, y otro que maneje la salida de elementos de la FIFO y le indique al compactor de menor altura si es una inserción o eliminación,

Para manejar la distinción entre los dos tipos de elementos, se utiliza un bit que distingue entre uno y el otro, donde el último bit, si es 0, es una eliminación, y si el último bit es 1 es una inserción. El módulo, FIFO_input crea esta nueva palabra con el bit de signo y se comunica con la cola FIFO a través de la señal put_in, la cual solo activa si la FIFO no está llena, a través de la señal.

Para sacar valores de la FIFO depende de si el compactor 0 si no está ocupado, y si la cola FIFO tiene elementos, entonces activa la señal get. Una vez se obtiene el dato, se revisa el último bit y se envía el dato con la señal Data_in_I si es una inserción, y si es una eliminación, se envía la señal Data_in_D.

```
1 if(In_Data == Del_Data) begin
2     //Si son iguales, se borra y se leen los siguientes
3     //Insert
4     rd_ptr_delete_I = rd_ptr_delete_I_old + 1;
5     wr_ptr_delete_I = wr_ptr_delete_I_old + 1;
6     //Delete
7     rd_ptr_delete_D = rd_ptr_delete_D_old + 1;
8     wr_ptr_delete_D = wr_ptr_delete_D_old + 1;
9     //Se borra
10    we = 1'b1;
11 end else if(In_Data < Del_Data) begin
12    //Insert
13    rd_ptr_delete_I = rd_ptr_delete_I_old + 1;
14    wr_ptr_delete_I = wr_ptr_delete_I_old + 1;
15    //Delete
16    rd_ptr_delete_D = rd_ptr_delete_D_old;
17    wr_ptr_delete_D = wr_ptr_delete_D_old;
18    we = 1'b0;
19 end else begin
20    //Insert
21    rd_ptr_delete_I = rd_ptr_delete_I_old;
22    wr_ptr_delete_I = wr_ptr_delete_I_old;
23    //Delete
24    rd_ptr_delete_D = rd_ptr_delete_D_old + 1;
25    wr_ptr_delete_D = wr_ptr_delete_D_old + 1;
26    we = 1'b0;
27 end
```

Extracto de código 1: Código en SystemVerilog de búsqueda de elementos a eliminar

```
REORDERING:
begin
    if(Data_out_ram == 0) begin
        wr_ptr = wr_ptr_old;
        we = 0;
        Data_in_ram = 0;
    end else begin
        Data_in_ram = Data_out_ram;
        wr_ptr = wr_ptr_old + 1;
        we = 1;
    end
    re = 1;
    rd_ptr = rd_ptr_old + 1;
    if(rd_ptr_old == compactor_length) begin
        next_state = IDLE;
    end else begin
        next_state = REORDERING;
    end
    compactor_busy = 1;
end
```

Extracto de código 2: Código en SystemVerilog de reordenamiento de elementos en compactor

```
else if(Rank_In_Ready[0] && !processed_rank[0]) begin
    processed_rank[0] <= 1'b1;
    if(sign_h_0 == 1) begin
        rank_estimated <= rank_estimated - (rank_h_0 << 4);
    end else begin
        rank_estimated <= rank_estimated + (rank_h_0 << 4);
    end
end
```

Extracto de código 3: Extracto de código en SystemVerilog de sección de estimación de rank

```
always_ff @(posedge clk) begin
    if(data_sam_in_in) begin
        data_out <= {1'b1, data_in_in};
    end else if(data_sam_del_in) begin
        data_out <= {1'b0, data_in_in};
    end
    put_in <= (data_sam_in_in || data_sam_del_in) && !full_in;
end
```

Extracto de código 4: Extracto de código en SystemVerilog para insertar elementos en la cola FIFO

5. Selección de parámetros

Como fue explicado en el capítulo 3, se debe determinar la estructura del sketch. La forma en que se determina la estructura del sketch KLL± es la determinación de la altura máxima H, la capacidad k de cada compactor, la altura máxima del sampler H´´ y el parámetro s, que representa la cantidad de compactores de altura H a H – s (o los s compactores más altos), cuya capacidad debe ser reemplazado por el parámetro k. Para obtener estos parámetros, se deben elegir el parámetro de precisión ε, el parámetro de probabilidad de falla δ, el parámetro de la razón de reducción de la capacidad de los compactores c y, finalmente, el parámetro de α. Como se explicó en el capítulo 4, se utilizará la estructura resultante después de cierta cantidad de inserciones, aspecto que se debe tener en cuenta al elegir los parámetros; aparte, se elegirán los parámetros de la capacidad de la cola FIFO y el generador de números aleatorios.

5.1. Selección de la cantidad de datos n y la razón α

El primer parámetro por seleccionar es la cantidad de datos de entrada al sketch. Como los autores presentan en [1], el parámetro n se obtiene de la siguiente manera:

$$n = I + D,$$

donde I es la cantidad de inserciones que recibe el sketch y D los elementos (que ya fueron insertados) que se eliminarán del sketch, es por ello por lo que se debe primero calcular el parámetro α. Para esta implementación se desea que la arquitectura se comporte como una ventana móvil, más específicamente a la implementación alternativa del KLL±, la cual es la "Fixed size sliding window" [1] o ventana móvil de tamaño fijo. Esta implementación alternativa funciona con tres vectores de datos de capacidad ω, donde está el vector active, vector BackUp y el vector Under-Construction, donde se van insertando inicialmente valores a al vector active, cuando se insertaron $\frac{\omega}{2}$ datos, se siguen insertando datos en el vector active y en el vector BackUp. Cuando el vector Active se completa con ω datos (por lo tanto, el vector BackUp tiene $\frac{\omega}{2}$ datos), se comienzan a insertar en el vector BackUp y UnderConstruction, aparte, el vector Active comienza a eliminar los primeros datos que fueron insertados. Para "emular" este comportamiento, se deben realizar en 2 etapas, la primera etapa solo se

inserta $\frac{I}{2}$ elementos, y la segunda a medida que se insertan la otra mitad de los datos, se elimina la primera mitad. Por lo tanto, el valor de α es 2, ya que se eliminan la mitad de los datos tipo Inserción. Este valor nos permitirá luego calcular D en función de I y obtener n.

Ahora, para elegir el valor de I, se consideran las trazas MAWILab [13], una de las trazas con las que se evaluará el diseño. Estas trazas en promedio tienen 100 millones de paquetes en una ventana de observación de un minuto. Se decide utilizar una ventana de observación de 30 segundos, para facilitar la evaluación del diseño al momento de procesar los paquetes, obteniendo un valor de I de 50 millones de datos.

Finalmente, se calcula D, cuyo valor es:

$$D = \left(1 - \frac{1}{\alpha}\right) \times I$$

$$D = 0.5 \times I = 25\ millones\ de\ elementos$$

Obteniendo finalmente que el parámetro n es de 75 millones de elementos.

5.2. Selección de la proporción de reducción de capacidad c

Para determinar el valor de c, los autores de [1] proponen que éste debe variar entre $\frac{1}{2}$ y 1; aparte de ello, los autores no proponen otra forma de determinar el valor, aparte de que, para probar el sketch, los autores utilizan el valor de $c = \frac{2}{3}$. Es por ello por lo que se elige simplemente elegir este valor, el cual se utiliza en la tabla 5.2.

5.3. Selección de la precisión ε y probabilidad de fallo δ

Para seleccionar ambos parámetros, se toma en consideración la cantidad de memoria que utilizará con las distintas combinaciones de los parámetros, para mostrarlo de manera numérica, se considerará que cada elemento, o sea, el tamaño del largo de cada paquete, es de 64 bits. En la tabla 5.1, se observa la fórmula que los autores en [1] propusieron para la obtención de H, H'', k y s. Para calcular el valor de s, se multiplica por dos la expresión para tener una mayor precisión en las estimaciones realizadas.

Tabla 5.1: Tabla de fórmulas de parámetros

Parámetro:	Formula:
Altura máxima del sketch (H)	$H = log_2\left(\frac{n}{c * k}\right) + 2$
Altura máxima del Sampler (H'')	$H'' = H - (\frac{log_2(3) - log_2(k)}{log_2(c)})$
Capacidad de elementos del compactor H (k)	$k_H = \frac{(2\alpha-1)^{1.5}}{\varepsilon\sqrt{\log(\frac{1}{\varepsilon\delta})}}$
Cantidad de top compactores con capacidad k (s)	$O(\frac{\alpha^{1.5}}{\varepsilon\sqrt{\log\left(\frac{1}{\delta}\right)}})$

Al observar la tabla 5.2, se pueden observar los distintos valores de ε y δ y los parámetros resultantes. Entre estos parámetros resultantes, se calcula el espacio total en kilobytes y el parámetro "Top compactores", que es el número de compactores que se tendrá en la estructura. Lo primero que se puede ver es que cuando $\boldsymbol{\varepsilon}$ varía entre 0,1 y 0,05 se tiene un valor de k bastante pequeño y un sampler bastante "grande", lo cual resultaría en una implementación que consume pocos recursos, pero el error esperado sería muy alto, del 10%. En cambio, si se observa cuando $\boldsymbol{\varepsilon}$ tiene el valor de 0,001, un error bastante pequeño, se obtiene un k mas alto, donde se consumen más recursos, pero se obtienen mejores resultados y la altura del sampler es menor, permitiendo más inserción de elementos. Como desafío, se eligió el error $\boldsymbol{\varepsilon}$ con valor de 0,001, significando que el valor de error esperado es cercano a 0,001.

Con respecto a $\boldsymbol{\delta}$, se eligió el valor de tener una probabilidad de fallo de 0,001, esto debido a que es la opción que consume menos memoria de entre los distintos valores de $\boldsymbol{\delta}$. Finalmente se tiene que la estructura tiene que ser de un $\boldsymbol{k_H}$ de 1164, una altura máxima de 19, un sampler de altura 4 y los top 6 compactores tendrán capacidad de $\boldsymbol{k_H}$**.** Se destaca también que como fue mencionado en el capítulo 4, se utilizaron dos memorias por compactor, resultando en que se consumirá 56 KB de memoria. En la tabla 5.3, se muestra cuántos elementos y cuánta memoria se espera que consuma cada compactor. Estos 15 compactores necesitan 15 bits de origen aleatorio para cada compactación, ya que la implementación del generador de números pseudoaleatorios genera palabras al azar de 16 bits, se tiene suficiente para tan solo necesitar un solo módulo para los compactores. Con respecto al

sampler, este es de altura 4, por lo tanto, la LUT que se debe realizar, debe tener 16 valores. Aparte se tienen dos sampler, pero como se recibe una inserción o una eliminación en un ciclo del reloj, solo se necesitara un generador para ambos sampler.

Tabla 5.2: Diferentes tamaños de la estructura del Sketch KLL±

ε	δ	k_H	H	H''	Top compactores	s	Espacio total en KB
0,1	0,1	21	25	20	5	3	0,46
0,1	0,05	19	25	20	5	4	0,41
0,1	0,005	16	25	21	4	4	0,32
0,1	0,002	15	25	21	4	4	0,30
0,1	0,001	15	25	22	3	3	0,26
0,05	0,1	38	24	18	6	3	0,86
0,05	0,05	36	24	18	6	4	0,80
0,05	0,005	31	24	19	5	5	0,66
0,05	0,002	29	24	19	5	5	0,62
0,05	0,001	28	24	19	5	5	0,60
0,005	0,1	314	21	9	12	3	7,54
0,005	0,05	301	21	10	11	4	7,18
0,005	0,005	266	21	10	11	5	6,35
0,005	0,002	255	21	10	11	6	6,08
0,005	0,001	248	21	10	11	6	5,92
0,002	0,1	742	20	6	14	3	17,78
0,002	0,05	713	20	6	14	4	17,10
0,002	0,005	638	20	7	13	5	15,30
0,002	0,002	614	20	7	13	6	14,69
0,002	0,001	598	20	7	13	6	14,32
0,001	0,1	1426	19	4	15	3	34,19
0,001	0,05	1375	19	4	15	4	32,98
0,001	0,005	1239	19	4	15	5	29,73
0,001	0,002	1195	19	4	15	6	28,68
0,001	0,001	1164	19	4	15	6	27,92

Tabla 5.3: Estructura de sketch resultante

Altura del Compactor	Capacidad de elementos	Espacio en KB
19	1164	9,31
18	1164	9,31
17	1164	9,31
16	1164	9,31
15	1164	9,31
14	1164	9,31
13	103	0,83
12	69	0,56
11	46	0,37
10	31	0,26
9	21	0,18
8	14	0,11
7	9	0,08
6	6	0,05
5	4	0,03

5.4. Selección de la altura de la FIFO

El ultimo parámetro a seleccionar es cuántos elementos puede guardar la FIFO. Como fue explicado anteriormente, esta cola se utiliza para evitar la pérdida de elementos, pero está la pregunta de cuál debe ser el largo de dicha cola: si es una cola con poca capacidad se seguirán perdiendo elementos y si es una cola con una gran capacidad, puede consumir muchos recursos si esta se sobreestima. Por lo tanto, es ideal poder encontrar un tamaño de cola que no pierda una cantidad de elementos significativa y tampoco este sobreestimada.

La mejor manera de seleccionar la altura es elegir distintas alturas de la FIFO y ver cuántos elementos se pierden, pero para ello se tiene que saber cuándo se considera un dato perdido. En este caso se define como dato perdido si el sampler intenta enviar un dato y la cola FIFO está llena, resultando en que ese dato simplemente se pierde.

Para contar los elementos perdidos, se utilizará la implementación de la arquitectura sobre un dispositivo FPGA, pero se realizará una modificación a la arquitectura presentada para que permita contar paquetes perdidos, donde se utilizara un simple contador, el cual cuenta cada vez que se cumplen las condiciones mencionadas en el párrafo anterior y se modificara la consulta para entregar el valor del contador en vez de la consulta. Con respecto a la inserción de paquetes de internet, se simulará la latencia en la recepción de elementos con respecto al tamaño de cada paquete. Donde se considerará que se recibirá que un paquete de tamaño mínimo se puede recibir en un solo ciclo de reloj, pero si un paquete es de mayor tamaño, por ejemplo, tres veces el tamaño mínimo, se considerará que se requieren 3 ciclos de reloj adicional, para la recepción del siguiente paquete.

Para la selección de los distintos tamaños de la cola FIFO, se utilizarán trazas de trafico de red las cuales provienen de MAWILab [13]: el cual es un repositorio utilizado para validar métodos de detección de anomalías. Aparte, se tienen 2 trazas del repositorio CAIDA (Center for Applied Internet Data Analysis) [14,15], dichas trazas fueron recolectadas entre Equinix y Chicago. Se utilizarán 4 trazas en este experimento, elegidas especialmente por la cantidad de paquetes por traza. Los resultados se pueden observar en las tablas 5.5 a 5.8.

Tabla 5.4: Tamaño de las trazas de red utilizadas

Nombre de la traza	Número de paquetes	Número de paquetes (considerando Inserciones y eliminaciones)
Mawi-2016	99.795.401	149.693.102
Mawi-2017	122.397.817	183.596.726
Mawi-2018	80.707.468	121.061.202
Mawi-2019	67.530.945	101.296.418
Mawi-2020	123.897.490	185.846.235
Mawi-2021	39.532.058	59.298.087
Chicago-2011	27.049.728	40.574.592
Chicago-2016	32.472.976	48.709.464

Tabla 5.5: Perdida de paquetes para Chicago 2016

Tamaño de la cola FIFO	Cantidad de paquetes perdidos	Porcentaje de paquetes perdidos
128	6.669	0,0137%
512	6.037	0,0124%
2048	4.799	0,0099%
4096	3.155	0,0065%

Tabla 5.6: Perdida de paquetes para Mawi 2018

Tamaño de la cola FIFO	Cantidad de paquetes perdidos	Porcentaje de paquetes perdidos
128	7.290	0,0060%
512	6.023	0,0050%
2048	4.758	0,0039%
4096	3.158	0,0026%

Tabla 5.7: Perdida de paquetes para Mawi 2020

Tamaño de la cola FIFO	Cantidad de paquetes perdidos	Porcentaje de paquetes perdidos
128	7.557	0,0041%
512	6.042	0,0033%
2048	4.806	0,0026%
4096	3.166	0,0017%

Tabla 5.8: Perdida de paquetes para Mawi 2021

Tamaño de la cola FIFO	Cantidad de paquetes perdidos	Porcentaje de paquetes perdidos
128	7.218	0,0122%
512	6.057	0,0102%
2048	4.790	0,0081%
4096	3.152	0,0053%

Con respecto a los resultados, la cantidad de paquetes perdidos considera también la perdida de eliminaciones, con respecto a la cantidad de paquetes de cada traza, estos se pueden observar en la tabla 5.4. Con respecto a la cantidad de elementos perdidos, se observa que con una cola de tamaño 128 o 512, se pierde sobre el alrededor de 7000 paquetes; mientras ese número es alto,

porcentualmente se observa que es 0,01%, lo cual es una cantidad pequeña de elementos. Mientras que con colas de tamaño 2048 y 4096 se pierde bajo ese porcentaje, pero se ve que la diferencia porcentual entre ambas es pequeña, especialmente para las trazas de Mawi 2020 y 2018, esto se debe a la cantidad de elementos que tienen a diferencia de las otras trazas que son de menor tamaño. Es por ello por lo que finalmente se elige la FIFO de tamaño 2048, debido a que la diferencia con la de tamaño 4096 es mínima y también debido a que la cantidad de elementos que se pierden no son representativos con respecto al tamaño de la traza.

6. Validación, implementación y resultados de desempeño

6.1. Validación de la arquitectura

Ya con la arquitectura diseñada y los parámetros, seleccionados el próximo objetivo es la implementación, pero antes de implementar, se validó la arquitectura a través de tres tipos de simulación: la simulación funcional, simulación funcional post síntesis y simulación funcional post implementación. Inicialmente, se utilizaron valores generados de manera aleatoria, pero para simulaciones posteriores se utilizaron 6 trazas de red, mencionadas en la tabla 5.4. en la que puede observar la cantidad de paquetes de cada una de las trazas. Aparte, para realizar las simulaciones se utilizó la versión de vivado 2022.2.

Para realizar las validaciones, primero se programó la arquitectura en C++ usando los algoritmos propuestos en el capítulo 4. Esta implementación en software permite facilitar la validación de la arquitectura y la selección de parámetros. Para la corroboración de la arquitectura, se comparó con los algoritmos propuestos por los autores en [1], por ejemplo, se comparó el cálculo del rank a través de la arquitectura propuesta en el capítulo 4.2.5 con el algoritmo 2 y el algoritmo de compactación de propuesto en 4.2.4 con el algoritmo 1. Aparte, como se observa en la figura 6.1 obtenida de [1], la cual es un ejemplo mostrado por los autores, se usó este ejemplo también para la corroboración de la lógica. Para la validación de la selección de parámetros, se compararon los resultados del cálculo del rank con los resultados de los ranks reales, en otras palabras, se calculó el error absoluto normalizado, el cual se calcula de la siguiente manera:

$$\frac{\left|R(x) - \hat{R}(x)\right|}{n},$$

donde R(x) es el cálculo del rank real del valor x, $\hat{R}(x)$ es el cálculo del rank estimado del valor x y n es el número total de paquetes no eliminados. Los elementos utilizados para la validación de la implementación en software se utilizaron valores generados al azar, pero luego para comparar los resultados con la implementación en hardware, se utilizaron principales las trazas mencionadas en la sección 5.4. El resultado final fue que se consiguió un error menor a 0.001, lo cual era lo deseado.

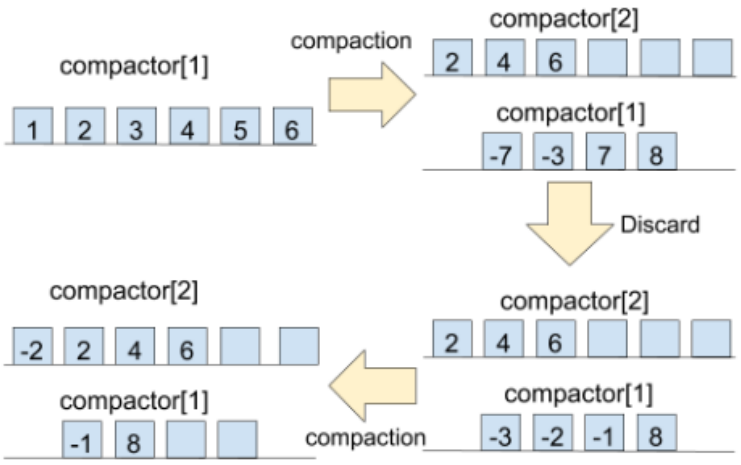

Figura 6.1: Ejemplo de funcionamiento del sketch KLL±

Una vez validada la arquitectura, se comenzó con la simulación funcional, cuyo objetivo principal es verificar la lógica implementada. También es la simulación más rápida entre las tres, ya que no toma en cuenta el tiempo que requiere el hardware para computar. Lo primero que se realizó fue la corrección de errores encontrados en la implementación de software. Luego de ambas implementaciones, se compararon a través de procesar el mismo set de datos, donde se verificó que los compactores enviaran la misma cantidad de datos al otro compactor, que el sampler envíe una cantidad correcta de datos a la cola FIFO. Finalmente, se compararon los resultados de la obtención del rank entre ambas implementaciones, donde se encontraron valores iguales.

Con respecto a la simulación funcional post-síntesis, la cual simula el hardware considerando el tiempo que requiere para computar, para validar esta simulación, se comparó con la simulación behaivoral, corroborando efectivamente que los resultados son iguales.

6.2. Implementación, uso de recursos y frecuencia de reloj

Para la implementación de la arquitectura, esta se realizó en la tarjeta Alveo u280 a través de la interfaz de vitis, una plataforma de desarrollo de aceleradores diseñada por Xilinx, en ella se implementaron cuatro kernels. Un kernel implementa la arquitectura presentada en este trabajo y los otros tres se utilizan para la comunicación entre el host y la arquitectura para enviar las inserciones, eliminación y realizar consultas.

Tabla 6.1: Utilización de recursos de la arquitectura

Recurso	Recursos usados	Disponible	Recursos usados en porcentaje
LUT	21.479	1.303.680	1,65
Memoria Distribuida	126	600.960	0,02
Registros	12.926	2.607.360	1,87
Block RAMs	75	2.016	3,92
DSPs	32	9024	0,35

Tabla 6.2: Utilización de recursos considerando el sistema conectado al host

Recurso	Recursos usados	Disponible	Recursos usados en porcentaje
LUT	142.953	1.303.680	10,97
Memoria Distribuida	8.512	600.960	1,42
Registros	177.337	2.607.360	6,80
Block RAMs	272	2.016	13,49
DSPs	36	9.024	0,40

La tarjeta en la que fue implementada la arquitectura contiene un chip UltraScale+ XCU280. La tarjeta está conectada a un servidor, el cual tiene un procesador Intel Core i9-10980XE de 18 núcleos, 128GB de memoria RAM DDR4 y un disco duro SSD de tipo NVMe de 1TB de capacidad, aparte tiene el sistema operativo Ubuntu 20.04.6 LTScon un kernel Linux 5.15.0.

Se observa en la tabla 6.1 la utilización de recursos de la arquitectura diseñada en el capítulo 4. Se observa en primer lugar la alta utilización de look up tables. También se observa la gran cantidad de Block RAM utilizadas. Esto debido a que se diseñó la arquitectura de 64 bits y se necesita más de una Block RAM para guardar esa cantidad de bits, aparte de que se utilizan dos memorias por compactor. Aparte, se observa el uso de DSPs, los cuales provienen del diseño de las PRNGs.

La velocidad del reloj obtenida es de 219 MHz, que corresponde a un flujo aproximado de 219 millones de paquetes. Esta velocidad está limitada por el camino crítico, el cual es el cálculo del puntero que indica al último dato escrito de los compactores de mayor capacidad.

6.3. Estimación de ranks

Para la estimación de ranks, se compararon los errores absolutos normalizados obtenidos mediante la implementación (KLL±_SH) y la implementación de la arquitectura en el acelerador (KLL±_Hw). Las consultas se realizaron en la segunda etapa de inserción de datos al sketch, o sea, cuando se están insertando y eliminando paquetes. Se llevaron a cabo 12 consultas; para determinar que valores consultar, se calculó el rank real de cada traza y se seleccionaron aleatoriamente 3 ranks con respecto a los cuantíales de los resultados obtenidos, o sea, 3 valores de ranks entre el percentil 0 al 25, 3 entre el 25 y el 50, hasta llegar al percentil 99. Para realizar la comparación de errores, como se explicó al principio del capítulo, ambas se harán a través del error absoluto normalizado $\frac{|R(x)-\hat{R}(x)|}{n}$.

Tabla 6.4: Comparación de errores para Mawi-2016

N° Consulta	KLL±_SH	KLL±_Hw
1	0,00088088	0,00088088
2	0,00088088	0,00088088
3	0,00088088	0,00088088
4	0,00071360	0,00071360
5	0,00234812	0,00234812
6	0,00159660	0,00159660
7	0,00234816	0,00234816
8	0,00158016	0,00158016
9	0,00038292	0,00038292
10	0,00000320	0,00000320
11	0,00000320	0,00000320
12	0,00000320	0,00000320
Media	0,00096848	0,00096848

Tabla 6.5: Comparación de errores para Mawi-2017

N° Consulta	KLL±_SH	KLL±_Hw
1	0,00240308	0,00240308
2	0,00240308	0,00240308
3	0,00240308	0,00240308
4	0,00031304	0,00031304
5	0,00031304	0,00031304
6	0,00016912	0,00016912
7	0,00015528	0,00015528
8	0,00135760	0,00135760
9	0,00135760	0,00135760
10	0,00000000	0,00000000
11	0,00000000	0,00000000
12	0,00000000	0,00000000
Media	0,00090624	0,00090624

Tabla 6.6: Comparación de errores para Mawi-2018

N° Consulta	KLL±_SH	KLL±_Hw
1	0,000279160	0,00027916
2	0,000279160	0,00027916
3	0,000279160	0,00027916
4	0,000279160	0,00027916
5	0,000495000	0,000495
6	0,000495000	0,000495
7	0,002301960	0,00230196
8	0,000430320	0,00043032
9	0,000194120	0,00019412
10	0,000337200	0,0003372
11	0,000724560	0,00072456
12	0,000632240	0,00063224
Media	0,000560587	0,000560587

Tabla 6.7: Comparación de errores para Mawi-2019

N° Consulta	KLL±_SH	KLL±_Hw
1	0,002813280	0,002813280
2	0,002813280	0,002813280
3	0,002813280	0,002813280
4	0,000135880	0,000135880
5	0,000135880	0,000135880
6	0,000135880	0,000135880
7	0,000135880	0,000135880
8	0,000728040	0,000728040
9	0,001657760	0,001657760
10	0,000626720	0,000626720
11	0,000990160	0,000990160
12	0,000990160	0,000990160
Media	0,001164683	0,001164683

Tabla 6.8: Comparación de errores para Mawi-2020

N° Consulta	KLL±_SH	KLL±_Hw
1	0,000914840	0,000914840
2	0,000914840	0,000914840
3	0,000914840	0,000914840
4	0,000668440	0,000668440
5	0,003595880	0,003595880
6	0,003595880	0,003595880
7	0,000221440	0,000221440
8	0,001696080	0,001696080
9	0,000361725	0,000361725
10	0,000699640	0,000699640
11	0,002253760	0,002253760
12	0,000933120	0,000933120
Media	0,001397540	0,001397540

Tabla 6.9: Comparación de errores para Mawi-2021

N° Consulta	KLL±_SH	KLL±_Hw
1	0,00072356	0,00072356
2	0,00072356	0,00072356
3	0,00072356	0,00072356
4	0,00066844	0,00066844
5	0,00255452	0,00255452
6	0,00255452	0,00255452
7	0,00022144	0,00022144
8	0,00075256	0,00075256
9	0,000369	0,000369
10	0,00069964	0,00069964
11	0,00225376	0,00225376
12	0,00067712	0,00067712
Media	0,001076807	0,001076807

Tabla 6.10: Comparación de errores para Chicago-2011

N° Consulta	KLL±_SH	KLL±_Hw
1	0,00118880	0,00118880
2	0,00118880	0,00118880
3	0,00184928	0,00184928
4	0,00223824	0,00223824
5	0,00097772	0,00097772
6	0,00207184	0,00207184
7	0,00033024	0,00033024
8	0,00361772	0,00361772
9	0,00185992	0,00185992
10	0,00173428	0,00173428
11	0,00000004	0,00000004
12	0,00000004	0,00000004
Media	0,00142141	0,00142141

Tabla 6.11: Comparación de errores para Chicago-2016

N° Consulta	KLL±_SH	KLL±_Hw
1	0,00229700	0,00229700
2	0,00012544	0,00012544
3	0,00078900	0,00078900
4	0,00173036	0,00173036
5	0,00002360	0,00002360
6	0,00243284	0,00243284
7	0,00108700	0,00108700
8	0,00108700	0,00108700
9	0,00074920	0,00074920
10	0,00000320	0,00000320
11	0,00000320	0,00000320
12	0,00000320	0,00000320
Media	0,00086092	0,00086092

Cómo se observa en las tablas anteriores, se cumple la cota del error en casi todas las consultas, donde son las consultas cercanas al cuantil 0.5 las cuales suelen estar arriba de la cota de error, mientras que las consultas de un cuantil mayor tienen un menor error. En promedio también se obtienen valores cercanos a 0,001.

7. Discusión, conclusiones y trabajo futuro

La propuesta de este trabajo fue la implementación de un sketch de cuantiles que permite eliminaciones en un acelerador hardware. La arquitectura que se diseñó fue una modificación del sketch presentado en [1], donde los algoritmos fueron modificados para adaptarse a la naturaleza de un acelerador. Considerando los resultados presentados en el capítulo 6.3, se puede concluir que los errores observados dentro de los límites esperados para el parámetro seleccionado $\varepsilon = 0{,}001$. La mayoría de los errores son menores que esta cota de error, mientras que aquellos que la superan se mantienen próximos al valor establecido.

Con respecto al diseño, la decisión de usar dos memorias por compactor fue una decisión acertada, ya que durante la etapa de compactación es necesario comparar simultáneamente dos elementos. Si se hubiese utilizado una única memoria, esta operación habría requerido dos ciclos de reloj, dado que cada ciclo permite la lectura de un solo elemento.

Con respecto al uso de recursos, la arquitectura fue diseñada para procesar paquetes de 64 bits, lo que explica que los valores reportados en la tabla 6.1 sean relativamente altos. Sin embargo, en la práctica se utiliza aproximadamente solo un 1% de los recursos disponibles. Por otro lado, la frecuencia de operación del reloj es menor en comparación con la implementación del KLL reportada en [11], aunque sigue siendo suficiente para manejar las velocidades típicas de redes comerciales.

Otra decisión de diseño fue implementar un generador de números pseudoaleatorios (PRNG) basado en un mapa caótico utilizando la técnica de pipeline. En [12], la implementación reporta una frecuencia de operación de 93 MHz; sin embargo, debido a que en nuestra arquitectura se requería una frecuencia mayor, se optó por implementar el PRNG en paralelo. Los resultados mostraron que esta implementación paralela no afectó la frecuencia total del sistema. En cuanto al uso de recursos, como se observa en la tabla 6.1, se emplearon DSPs para las operaciones de multiplicación necesarias en el generador.

La principal consideración del diseño presentado en este trabajo es la necesidad de guardar los paquetes insertados. Durante la validación de la arquitectura, estos paquetes se guardaron en un archivo temporal y luego se leían para ser eliminados en la siguiente etapa. En un escenario de

implementación real en un flujo de paquetes de internet, este almacenamiento podría realizarse en una memoria RAM.

Una limitación relevante es la capacidad de la FIFO, especialmente en comparación con la implementación reportada en [11]. En la arquitectura presentada en este trabajo, la cola FIFO tiene una altura de 4096, significativamente mayor que en [11], lo que se debe a la segunda etapa de inserción de paquetes. Durante esta etapa, se van eliminando elementos ya insertados, cuyo proceso requiere recorrer completamente las Block RAM en búsqueda de elementos a eliminar. Como resultado, los compactores de mayor capacidad requieren más ciclos de reloj antes de poder recibir más elementos, haciendo que el resto de compactores queden a la espera y no puedan recibir nuevos paquetes.

Como trabajo futuro, se puede implementar el sketch KLL± como ventana deslizante, el cual fue propuesto en [1], el cual usa tres de estos sketches en forma de ventana deslizante, facilitando el análisis de flujo de datos y la posibilidad de usar una mayor ventana de observación, para detectar anomalías en una mayor franja de tiempo. Aparte, se puede analizar la inclusión de combinarlo con algoritmos de análisis de base de datos, ya que esta implementación permite realizar consultas cada cierto tiempo, estas consultas se pueden guardar en una base datos y así se pueden realizar análisis más extensos luego.

Referencias

[1] F. Zhao, S. Maiyya, R. Wiener, D. Agrawal, and A. El Abbadi, "KLL± approximate quantile sketches over dynamic datasets," Proceedings of the VLDB Endowment, vol. 14, no. 7, pp. 1215-1227, 2021.

[2] H. Han, Z. Yan, X. Jing, and W. Pedrycz, "Applications of sketches in network traffic measurement: A survey," *Information Fusion*, vol. 82, pp. 58–85, 2022. doi: 10.1016/j.inffus.2021.12.007.

[3] A. K. Marnerides, A. Schaeffer-Filho, and A. Mauthe, "Traffic anomaly diagnosis in Internet backbone networks: A survey," *Computer Networks*, vol. 73, pp. 224–243, 2014.

[4] P. Papaphilippou and W. Luk, "Accelerating database systems using FPGAs: A survey," in 2018 28th Intl. Conf. on Field Programmable Logic and Applications (FPL). IEEE, 2018, pp. 125–1255.

[5] H. Han, Z. Yan, X. Jing, and W. Pedrycz, "Applications of Sketches in Network Traffic Measurement: A Survey," Information Fusion, vol. 82, pp. 58–85, Jan. 2022

[6] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," 2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS), pp. 71-82, 2016, doi:10.1109/FOCS.2016.17

[7] G. Cormode, Z. Karnin, E. Liberty, J. Thaler, and P. Vesel`y, "Relative error streaming quantiles," in Proceedings of the 40th ACM SIGMOD SIGACT-SIGAI Symposium on Principles of Database Systems, 2021, pp. 96–108.

[8] F. Zhao, D. Agrawal, A. E. Abbadi, and A. Metwally, "SpaceSaving ±," Proceedings of the VLDB Endowment, vol. 15, no. 6, pp. 1215–1227, Feb. 2022, doi: https://doi.org/10.14778/3514061.3514068.

[9] C. Gallardo-Pavesi, Y. Fernandez, J. E. Soto, C. Hernández and M. Figueroa, "A Hardware Accelerator for Quantile Estimation of Network Packet Attributes," 2024 27th Euromicro Conference on Digital System Design (DSD), Paris, France, 2024, pp. 114-121, doi: 10.1109/DSD64264.2024.00024.

[10] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "Qpipe: Quantiles sketch fully in the data plane," in Proc. of the 15th Intl. Conf. on Emerging Networking Experiments and Technologies, 2019, pp. 285–291.

[11] C. S. Gallardo Pavesi, "Acelerador hardware para estimación de cuantiles en tráfico de redes," Memoria de Título, Departamento de Ingeniería Eléctrica, Universidad de Concepción, Concepción, Chile, 2024.

[12] A. Pande and J. Zambreno, "A chaotic encryption scheme for real-time embedded systems: design and implementation," *Telecommun Syst*, vol. 48, no. 3-4, pp. 213-223, 2011. doi: 10.1007/s11235-011-9460-1

[13] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, “Mawilab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking,” in ACM CoNEXT ’10, Philadelphia, PA, December 2010.

[14] Anonymized Internet Traces 2011,” https://catalog.caida.org/dataset/passive_2011_pcap.

[15] Anonymized Internet Traces 2016,” https://catalog.caida.org/dataset/passive_2016_pcap.

UNIVERSIDAD DE CONCEPCION – FACULTAD DE INGENIERIA
RESUMEN DE MEMORIA DE TITULO

Departamento	**:** Departamento de Ingeniería Eléctrica
Carrera	: Ingeniería civil electrónica
Nombre del memorista	: Felipe Winkler Enríquez
Título de la memoria	**:** Estimación de cuantiles en acelerador hardware usando sketch KLL±
Fecha de la presentación oral	: 03/09/2025
Profesor(es) guía	: Miguel Figueroa, Cecilia Hernández
Profesor(es) revisor(es)	: Mario Medina
Concepto	**:**
Calificación	**:**

Resumen (máximo 200 palabras)

El crecimiento de Internet y la dependencia de las redes hacen del monitoreo de tráfico y la detección de anomalías tareas esenciales. Una métrica clave es el cálculo de cuantiles, útil para identificar patrones y anomalías en la distribución de datos. Sin embargo, las altas velocidades de las redes actuales y la obsolescencia rápida de los datos plantean grandes desafíos para el procesamiento en tiempo real.

Este trabajo presenta el diseño e implementación de un acelerador hardware en FPGA para la estimación eficiente de cuantiles en flujos de red dinámicos, basado en el algoritmo KLL±, que soporta inserciones y eliminaciones, adaptándose a entornos de alta variabilidad. La arquitectura optimiza recursos mediante estructuras estáticas, compactores jerárquicos y generadores pseudoaleatorios basados en mapas caóticos.

La validación con trazas de red reales mostró un error absoluto normalizado menor al 0,1% y una frecuencia de operación de 219 MHz. Estos resultados confirman que el acelerador cumple con los requisitos de precisión y velocidad para aplicaciones de monitoreo en redes de alta velocidad.