Memory-efficient Sketch Acceleration for Handling Large Network Flows on FPGAs

Zhaoyang Han*, Yicheng Qian* Michael Zink†, Miriam Leeser*
Northeastern University *, University of Massachusetts Amherst †
Email: *zhhan,qianyich,mel@coe.neu.edu
†mzink@umass.edu,

***Abstract*—Sketch-based algorithms for network traffic monitoring have drawn increasing interest in recent years due to their sub-linear memory efficiency and high accuracy. As the volume of network traffic grows, software-based sketch implementations cannot match the throughput of the incoming network flows. FPGA-based hardware sketch has shown better performance compared to software running on a CPU when handling these packets. Among the various sketch algorithms, Count-min sketch is one of the most popular and efficient. However, due to the limited amount of on-chip memory, the FPGA-based count-Min sketch accelerator suffers from performance drops as network traffic grows. In this work, we propose a hardware-friendly architecture with a variable width memory counter for count-min sketch. Our architecture provides a more compact design to store the sketch data structure effectively, allowing us to support larger hash tables and reduce overestimation errors. The design makes use of a P4-based programmable data plane and the AMD OpenNIC shell. The design is implemented and verified on the Open Cloud Testbed running on AMD Alveo U280s and can keep up with the 100 Gbit link speed.**

***Index Terms*—FPGA, Partial Reconfiguration, P4, Network Systems**

I. Introduction

For data processing and analysis, the count-min sketch algorithm, presented by Cormode [6], has emerged as a powerful and efficient technique for approximate counting and frequency estimation tasks. It harnesses simple probabilistic hash functions to precisely characterize data sequences while using sub-linear memory space.

Count-min sketch is widely used in computer networking for tasks such as heavy hitter detection, change detection, and cardinality estimation. The heavy hitter detection problem finds network flows that occupy a significant amount of the network bandwidth. Identifying and taking action against heavy hitters improves network applications like network traffic engineering and can be applied to network intrusion detection [6], [7], [16]. Field-Programmable Gate Arrays (FPGAs) are excellent candidates for implementing sketch algorithms. The FPGA architecture is well known for its ability to implement parallelism and pipelining, which is an ideal match for the sketch approach that needs to maintain counters in parallel and process data streams in a pipelined manner. However, as network traffic proliferates over time and the bandwidth of network interfaces increases, FPGA implementations of the sketch algorithm suffer from performance drops due to limited on-chip memories.

In this paper, we present a count-min sketch implementation with a variable width counter array designed to provide more efficient memory usage. Hua, et al. proposed a variable width counter design: Bucketized Rank Indexed Counters (BRICK) [11] for 64-bit CPUs. We optimize the BRICK architecture for FPGAs. Then we develop a count-min sketch FPGA accelerator using the Protocol-Independent Packet Processing (P4) language combined with the proposed implementation, verify its correct behavior, and measure the performance on an FPGA node directly connected to a 100 Gbps network.

The contributions of this paper are:

- We present HBRICK, a hardware friendly memory design implementing variable width counter arrays on FPGA fabric based on BRICK.
- We build a data-pipelined count-min sketch architecture with HBRICK.
- We integrate a real time in-network count-min sketch accelerator with the proposed architecture into an FPGA-based NIC (AMD/Xilinx's U280) that proceses 100Gbps packet streams.

The resulting design combines P4 and High-Level Synthesis (HLS) to implement a hardware-friendly count-min sketch with a variable width counter and demonstrates the throughput advantage and reduced resource utilization of the design. The design is tested using network data in the Open Cloud Testbed (OCT) [27]. It demonstrates a design model where the FPGA can directly extract information from the packets and conduct acceleration tasks without the interference of the host. This approach can be extended to other applications where the FPGA processes data directly from the network.

The rest of this paper is organized as follows. Sec. II presents background and related work, focusing on the sketch algorithm with variable width counters and highlighting its benefits. We also discuss the recent works in combining the usage of the network domain-specific language P4 and HLS for developing network functions. In Sec. III, we investigate the algorithms of BRICK and our optimized hardware-friendly HBRICK. Sec. IV presents the combined use of P4 and HLS to easily implement a system level in-network function design that can be plugged into the FPGA-based SmartNICs. Next,

This work was funded by National Science Foundation (NSF) grants CNS-1925464, CNS-1925658, CNS-2130891, CNS-2130907 and CICI-2319962. All opinions and statements in the above publication are of the authors and do not represent NSF positions.

we present our experiments and results. We conclude with a discussion and conclusions.

II. Background

A. Sketch with Variable Width Counters

FPGAs are particularly well suited to accelerate streaming algorithms, where data are streamed into the accelerator and processed in a single pass. As the data size is much larger than the hardware memory resources, the goal is to use sub-linear memory resources compared to the number of data items to estimate a certain variable.

In this paper, we target a particular streaming data problem: detecting the heavy hitters in network flows. Here, each data item is an individual network packet. We identify a set of packets with the same five-tuple (source and destination IP addresses, source and destination transport protocol number, and transport protocol type) as a packet flow. We characterize a packet flow as a "heavy hitter" when the accumulative size of all packets within the flow surpasses a specified threshold [6].

Sketches are algorithms that condense data stream information using sub-linear memory about the data stream size. Typically, sketches comprise multiple counter arrays to approximate the frequency of elements within the stream. In the heavy hitter problem, we use the sketch algorithm to record the packet flow size. While sketches utilize smaller memory, increasing the number of entries in the counter arrays reduces the estimation error of the sketch. With a constrained memory size, reducing the number of bits used for each entry in the counter arrays allows for more entries within each array. This results in better estimation accuracy due to the increased capacity for capturing more elements in the data stream.

We analyze the sizes of 588K network flows based on real network traces made public by CAIDA [3]. We calculate the minimum bit width required to store these flow sizes. As shown in Fig. 1, the real network traces contain mostly small flows. The heavy hitters, which require more memory bits for recording, constitute only a small fraction of the total flows. In other words, only a very small number of counter entries require more bits. One solution to reduce memory usage efficiently is to use variable width counters instead of fixed width. To achieve the variable width counters, extra processing cycles are commonly required to access a certain entry due to its adaptive architecture. Thus, variable width counters provide a trade-off opportunity between computing overhead and memory.

B. P4-based In-network Computing on FPGA-based SmartNICs

In-network computing aims to offload compute intensive networking tasks from the CPU to network devices like programmable NICs. In-network computing can reduce the overhead incurred by the CPU host and improve the performance in throughput and other metrics. Different kinds of hardware devices, such as AMD Pensando's DPUs, Nvidia Bluefield SmartNICs, and AMD/Xilnx's FPGAs, have been used for computing in the network. There was a lack of device-agnostic

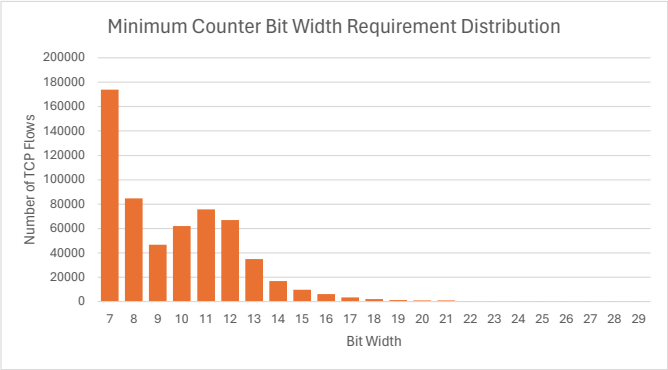

Fig. 1. Minimum Counter Bit Width Requirement Distribution of 588K TCP/UDP flows of real network traces. The largest packet flow requires 29-bit to store its total size while most flows only require 14-bit counters.

high-level abstractions in processing packets among these devices. P4, a domain-specific language originally developed to control packet forwarding in the data plane for devices like routers and switches, provides a language for programming these NIC devices at a higher level [2].

FPGAs have been a good candidate as P4-programmable devices since the beginning of the P4 language. P4-NetFPGA [12] created a workflow enabling offloading P4 functions onto FPGA-based NICs while maintaining high throughput. To maximize the capabilities of FPGAs in in-network computing, we propose a design that combines P4 and HLS. The intended network task will be segmented into one or more computational kernels, which are constructed using HLS, whereas P4 will control the packet flow through different kernels based on packet headers. We demonstrate our design as a case study of the combination.

C. Related Work

FPGA-based designs of count-min sketch with its variants have been widely studied in the past decade because of the match between the hardware device features and sketch structures. Tong et al. [23]–[25] implemented FPGA-based count-min sketch in the 2010s. They provide details on count-min sketch implementations. They analyzed the data conflicts caused by memory access latency and designed a data forwarding unit to mitigate its effect [25]. In contrast, [5] and [21] describe the design of an approximate counter that can record larger data values with less memory. Kiefer et al. [14] propose a programming model to summarize general sketch designs and their system, Scotch, generates the hardware designs automatically. Their techniques are independent of the counter array design, which our work can easily apply. Different groups have also researched other FPGA-based sketch algorithms including Hyperloglog [4], [15], [17].

More complicated sketch systems using P4 [18], [26] implemented on network devices including the Intel Tofino switch [13] have been presented in the past. P4 provides flexibility in designing sketch systems for complex network monitoring tasks. As P4 is device-independent, developers can

scale their P4 designs across the network with heterogeneous network devices. In [22], the authors propose a framework that can disaggregate complex network tasks described by P4 into multiple small functions and assign them to different types of network devices including Intel Tofino switches, AMD's FPGAs and legacy x86 CPUs. Our application also builds on the P4 language, where we use P4 to define the packet-level behaviors for network flows and combine it with HLS to generate a high-performance processing block for count-min sketch that interacts with P4. This allows us to test our design on an FPGA directly connected to the network.

III. Algorithm and Hardware Implementation

A. Count-min Sketch

In general, a streaming data problem is described as follows: For a given sequence $S_T = \{X_1, X_2, ...X_T\}$, compute a function F of the sequence: $F(S_T)$. To solve a data streaming problem while also utilizing sub-linear memory resources, an approximate estimation of the function F is generally used.

Count-min (CM) sketch is a widely used algorithm to estimate the sum of data elements. The idea is to map a data element to multiple counters using independent hash functions and then increase the value of the corresponding counters by the size of the data element. To estimate the total size of a certain flow, the algorithm will return the minimum value among all counters for that data element.

Algorithm 1 Count-Min Update

Input: S_T
Output: C
  $C_d[W] \leftarrow 0$
  for $t \leftarrow 1$ to T **do**
    $k_t, c_t \leftarrow X_t$
    for $d \leftarrow 1$ to D **do**
      $C_d[h_d(k_t)] = C_d[h_d(k_t)] + c_t$
    end for
  end for

To illustrate the algorithm, we define D hash functions $h_d \in H$. Each hash function is associated with a counter with W entries: $C_d[W] \in C$. The count-min counters are maintained as shown in Algorithm 1. The symbols k_t and c_t are denoted as the key (packet flow ID) and the size of the packet X_T. The flow ID can be determined by the five tuples introduced earlier.

To query the flow information for a given key k, the algorithm compares all the corresponding values from all counters and returns the minimum value. It can be described as:

$$F(S_T, k) = min(C_d[h_d(k)]), \forall d \in D \quad (1)$$

The combination of the design parameters D and W defines the estimation error bound of the CM sketch. The theoretical CM error bounds are as follows. For a given $D = log(1/\delta)$, $W = 2/\epsilon$ and any threshold ϕ, with the probability of $1 - \delta$, the sketch algorithm will not falsely over-estimate any flow with size lower than $\phi - \epsilon$ of the entire stream. In other words, mathematically, the overestimation error of the count-min sketch will not exceed $1 - \delta$.

B. Challenges in Bucketized Rank Indexed Counters (BRICK)

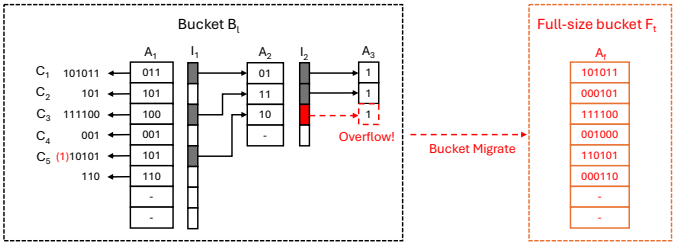

Fig. 2. Bucket design of BRICK Architecture and Bucket Migration. The red parts indicate the bucket overflow migration when there is not enough A_3 sub-counter bits.

BRICK [11], a variable width counter design, is established based on the idea that not all data entries in the counter will use up all the bits assigned. On the contrary, it assumes that most of the data entries will only require an average number of bits. The entire BRICK counter system is constructed from N buckets, totaling $W = Nk$ entries. In each counter bucket, the first level contains k sub-counters with average bitwidth, while subsequent levels A_i contain $k_i < k$ sub-counters. We refer to these subsequent level sub-counters as optional levels and A_1 as the base level. If we properly select the base level bitwidth as the average bitwidth of the entire data, most of the data will only use up the base level sub-counters. Only a small portion of the data will require optional level sub-counters, i.e. $A_2, A_3, ...$ As a result, we achieve reduced memory utilization through this multi-level memory counter design. Fig. 2 depicts a specific three-level counter bucket comprising k counter entries. When an entry with address C is recorded, the correct bucket is determined by C/k, and its relative position within the bucket is $C_i = C \bmod k$. The first three bits of the entry are stored in the base sub-counter. If the value exceeds three bits, it automatically overflows to the optional levels.

To connect the data at different levels of the counters, the BRICK approach uses the technique called rank indexing. Rank indexing is based on 1-bit bitmaps I. For an L-level design, there will be $L-1$ bitmaps. Each bitmap has the same number of entries as its level. If we denote I_i as the bitmap of i-th level A_i, $I_i[p]$ denotes whether the sub-counter entry $A_i[p]$ extends to the next level A_{i+1}. If the bit is set to true (shown as a shaded block in the figure), the rank $q = rank(I_i[p])$ is entry p 's next-level location. The $rank(I_i[p])$ function returns the number of ones in the current bitmap I_i in the range $[0, p]$. For example, in Fig. 2, the entry C_3 at the first level has its corresponding bit at bitmap I_1 set to be true (shaded). It is the second shaded box of the entire bitmap I_1, i.e. its rank is 2. Therefore the entry C_3 has an extension to the next level, $A_2[2]$.

The design divides the counter into small buckets and assumes that data is normally distributed so that data items with large values will fall into different buckets on average.

However, real data is often skewed and not normally distributed. Another important scheme of the BRICK design is to handle the insufficient number of sub-counters. As the sub-counters for higher levels are limited, the worst case is the sub-counters at the higher level are not enough to accommodate all the large data items. In other words, more than expected large flows fall into the same bucket. In Fig. 2, the sub-counter A_3 overflows as it reaches the capacity of A_3. When this happens, the BRICK design will copy all the data in the bucket to a spare, full width bucket. In [11], the authors provide concrete mathematical analysis in determining the width and number of entries for each level of the sub counters to reduce the possibility of overflows.

The BRICK design was originally designed for CPU-based sketch counters. When network functions like sketch implementations are offloaded to FPGA-based SmartNICs, it is necessary to accommodate such variable width counter designs onto the FPGA to improve memory efficiency and estimation accuracy. A few challenges need to be addressed for implementing such design on an FPGA to achieve high performance:

- As multiple cycles are required to read a complete entry from the counters, **data hazards** are introduced and cause increasing errors in the sketch estimation.
- An **inefficient shift operation** happens during the counter update stage. For example, in Fig. 2, if C_2 at level A_1 is about to extend to the next level, it should fill the position $A_2[2]$. If $A_2[2]$ is already occupied from earlier counter operations, a shift of $A_2[2]$ and $A_2[3]$ is required.
- There is an **inefficient bucket migration strategy**. The BRICK design requires a few full width buckets and every entry of the bucket has the worst-case width. This is an inefficient use of the memory as all the data in the original bucket have an extra copy in the full-size bucket. In addition, extra time is required for copying the data.

C. Design of Hardware-friendly Bucketized Rank Indexed Counters (HBRICK)

To address the above issues, we propose the Hardware-friendly Bucketized Rank Indexed Counters (HBRICK).

The main issue causing **data hazards** in the BRICK design is its recursive indexing algorithm, which results in variable cycles to update the counter based on the element's bitwidth. Larger elements with more sub-counters require more cycles. To resolve this, we propose a parallel indexing architecture, as shown in Fig. 3. Instead of recursively indexing the sub-counters, we separate the process into an indexing phase and an update phase. We combine the indexing arrays I_i into a single large array I. During the indexing phase, We use I to generate the indices for all layers together through the rank operation. With the full-size indexing array I, we can index all layers in parallel and fetch the results from sub-counters simultaneously. Our universal index array design simplifies index management. Although the bitmap I requires more memory space than in BRICK, this allows us to control the entire update process at a fixed clock cycle T_c, solely based on the number of entries k in each bucket. Given this fixed overhead, we create a Data Forwarding Unit (DFU) for each bucket, essentially a FIFO with a depth T_c. This unit consolidates multiple accesses to the same entry into a single access over consecutive T_c cycles. Algorithm 2 presents the details of HBRICK.

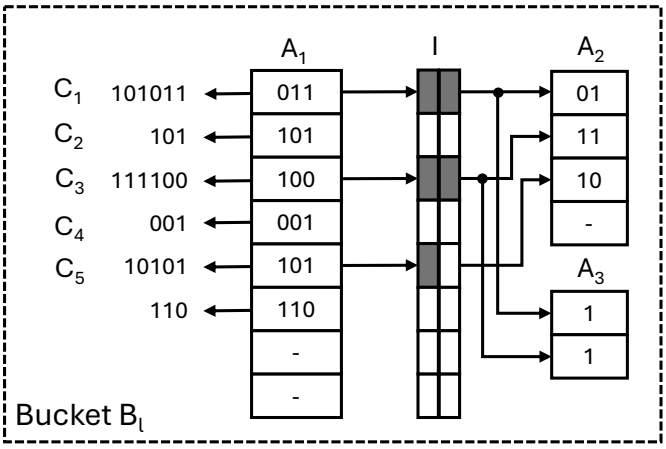

Fig. 3. HBRICK's optimized bucket design

Algorithm 2 HBRICK Access

Input: Index i
Output: Count C
  $C \leftarrow 0$
  Bucket $B \leftarrow i \div N$
  $j \leftarrow i \mod k$
  $S \leftarrow rank(j, I_B)$
  for $l \leftarrow 1$ to L **do**
    Note: Iterations execute in parallel
    $C+ = A_l[S_l]$
  end for

To improve the **inefficient shift operation**, we employ data packing techniques for the optional level sub-counters. We pack all the data in optional levels into single data words respectively, transforming the data shift operation into an atomic bit-wise shift. While this reduces the total number of entries k in each bucket, it also decreases the processing overhead and helps improve the total throughput. Additionally, since the access for an optional level value is a single word, we can store this word in a local register during consecutive accesses. This approach avoids the data dependency for BRAM reads and writes. However, the drawback of this design is that it is constrained by the maximum bitwidth of the BRAMs, which is 72 bits.

To **efficiently address bucket overflows**, we propose integrating a BRAM-based fully associative memory block into the design. This addition enables the rapid identification of overflowed entries across the entire counter set. Each entry in the bucket is assigned a dirty bit in an array V; this dirty bit indicates whether the entry has been evicted from the bucket.

By combining the index array and the dirty bit array, we save additional memory space compared to the original BRICK architecture. According to [11], for real network trace with millions of packets, BRICK requires $J = 100$ extra full width buckets with k entries each to accommodate overflowed data items. Usually, there are only 1 or 2 overflows that happen per bucket. As a result, HBRICK requires only J entries in total to migrate the overflows, compared to $k * J$ total entries required in BRICK.

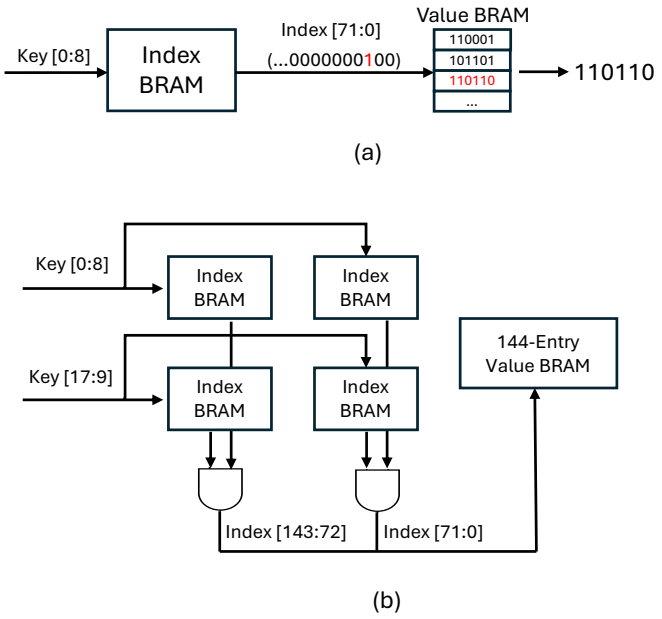

Fig. 4. HBRICK Associative Memory Design. (a) demonstrates a single mapping between the 9-bit key and 72 entries. (b) shows an extension of the associative memory with a larger key width and capacity.

The design of the associative memory also utilizes the on-chip BRAM. The FPGA dual-port BRAMs are 36Kb memory on AMD Alveo cards and each can be configured as 512 entries with 72-bit wide data. In other words, we can create a simple array with 9-bit addresses to store 72-bit wide data. However, this design will lead to very sparse memory usage. We applied a similar rank index technique used in our bucket design to construct our associative memory since we only need hundreds of entries while the key space is large. For a given key, we use the 72-bit BRAM value to store whether such key is in the associative memory, as shown in Fig. 4, and indicate the location of the value. In HBRICK, we applied the key width based on W and the capacity based on the possibility of overflow. With the above designs, we demonstrate the counter update algorithm in Algorithm 3.

Fig. 5 shows the overall design of the entire HBRICK counter. The optimized bucket design has three stages. At the pre-processing stage, the access to the same entry will be combined. The indexing stage will calculate the correct indices to access the correct BRAMs among all levels. At last, the values are fetched from the BRAMs and reconstructed into

Algorithm 3 HBRICK Update

Input: Index i, New Value C
  Bucket $B \leftarrow i \div N$
  $j \leftarrow i \mod k$
  $S \leftarrow rank(j, I_B)$
  if WIDTH EXPAND **then**
    if OVERFLOW **then**
      $V_B[j] \leftarrow 1$
    else
      Counter Shift: $A[S] >> d_l$
    end if
  end if
  Update Base Sub-counters:
  $A_1[j] = C[:]$
  Update Optional Sub-counters:
  for $l \leftarrow 2$ to L **do**
    Note: Iterations execute in parallel
    $A_l[S_l] \& = C[:]$
  end for
  Write A_l back to BRAM

the final value. If the dirty bit is set, access to the associative memory is needed.

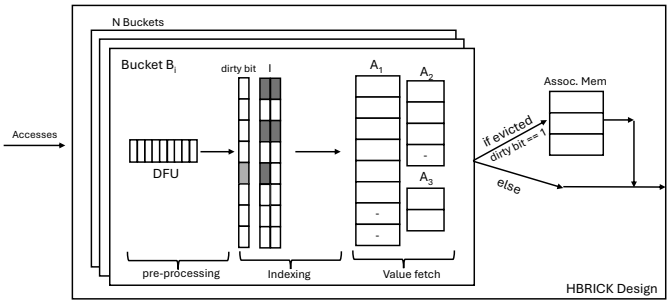

Fig. 5. Overall HBRICK architecture and data paths. The optimized bucket design includes three stages: pre-processing, indexing, and value fetching.

IV. System Implementation

To demonstrate the use of our HBRICK counter in the count-min sketch algorithm and to measure its performance, we develop an in-network count-min sketch implementation with HBRICK and implement it on an FPGA-based NIC.

We implement the entire system on an FPGA node in the Open Cloud Testbed (OCT) [27], where an AMD Alveo U280 FPGA is used as a SmartNIC attached both to the host CPU and to the 100Gbps network to receive and process incoming packets. The experimental setup is described in Sec. V. The system is generated with support from the toolchain developed by the OCT operators [9]. The OCT provides a framework that allows network researchers to easily program their P4 applications onto the FPGA through the use of AMD's P4 compiler. In our case, we extended the usage of this framework and combined it with HLS to achieve a much more complicated design. We showcase the usage of two different high-

level abstractions under the network processing framework and explore how to connect two different toolchains.

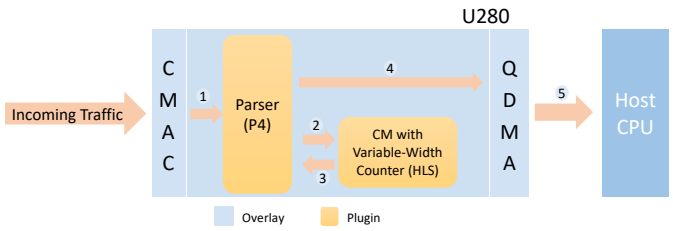

Fig. 6. System Architecture for HBRICK-based Count-min Sketch. Data on the numbered path: 1. raw packet; 2. packet key; 3. estimate value; 4&5 packet forward to the host.

This system implementation is broken up into two parts: front end and back end. On the back end, we use the count-min sketch with HBRICK counter generated by HLS as the back end kernel recording and estimating the flows. The front end is a P4 block that controls the packet's path, and extracts header fields as the flow key. It also helps reconstruct the packets and forward the packet to the host. Fig. 6 shows the system architecture of our design. We make use of the open source OpenNIC shell [1] as the fundamental framework, which offers MAC and DMA support for IO. Our application is a plugin block that sits between the CMAC and QDMA blocks of the OpenNIC shell.

The code snippet for the P4 function is shown in Listing 1. We explore the usage of the P4 extern functions as the connecting point between the HLS-generated blocks and P4 blocks. In particular, we utilize the P4 extern functions as the wrapper of our HLS function and create a standard interface to exchange the necessary information between blocks. Through the use of the high-level language P4, we have significantly reduced the effort required for designing and implementing network functions. The P4 code requires only about 300 lines, 50% of which is the definition of headers. Our HBRICK-based count-min sketch also requires about 400 lines of HLS code. The abstraction of the P4 language demonstrates a powerful design model for extracting necessary information directly from the network packets. The FPGA kernel can then bypass the unnecessary data copy in the host CPU. In our case, the extracted information is packet headers. Alternatively, it can also be information in the packets' payloads. For example, we can easily use this design model to extract images from the network packet and execute machine learning inference on FPGAs directly.

```
...
parser MyParser(packet_in pkt, out headers hdr){
  apply {
    pkt.extract(hdr)};
}
control HHD_Processing(inout headers hdr) {
//  Declare exteranl sketch engine
  UserExtern<in_tuples_t, out_tuples_t>(4) sketch;
...
  apply {
//  Prepare input for the sketch engine
...
    //Call the backend sketch engine
    sketch.apply(tuple_in, tuple_out);
//  Define post-process behaviours:
//  Setting the user-defined header field
    if (tuple_out.flow_size > THRESHOLD)
      hdr.isHitter = 1;
    else
      hdr.isHitter = 0
    forwardPacket();
  }
}
...
```

Listing 1. P4-based front-end

V. Experiments and Results

In this section, we describe the testing environment, OCT, and test results of our design. First, we quantify our results in reducing BRAM utilization. Next, we provide a design accuracy profile comparing our HBRICK to the original BRICK. Finally, we present end-to-end throughput results from testing in a real network with actual network packets.

A. Testing

For testing, we request an FPGA node from the Open Cloud Testbed [10]. The node is connected to OCT infrastructure through 100G network links for network tests. We load our implementation onto the FPGA and employ it to receive and identify packets from the network. The packets are real time traces from the Center for Applied Internet Data Analysis (CAIDA) that were collected at the Equinix-Chicago backbone link in January, 2016 [3]. We replay these packets from another OCT node that connects to the same network via 100Gbps port as our FPGA node. The FPGA node processes all the incoming packets from the 100Gb link and then forwards them to the host, where we verify the received packets and measure the performance. The host is running with DPDK [8] as the NIC driver which allows it to keep up with the throughput.

B. BRAM Utilization and Performance

We have implemented the design of HBRICK under different configurations to find the optimal choices of the number of layers. We know that increasing the number of counters and layers will result in better estimation accuracy and increased memory utilization. The increased memory utilization will decrease the maximum operating frequency of the module and thus affect the throughput. Also, the increased number of memory layers will lead to large overhead and lower frequency. Therefore, one goal is to determine the optimal trade-off that achieves both high throughput and low estimation error.

First, we investigate the optimal selection of the number of layers. Table I illustrates the impact of the hierarchy of counters on frequencies and estimated processing overheads. As the number of layers increases, memory utilization becomes higher, but this comes at the cost of reduced processing speed

TABLE I
DIFFERENT HBRICK MEMORY LEVELS FOR 2^{15} ENTRIES.

Levels	Freq. (MHz)	Overhead (cycles)	BRAMs
2	412	10	132
3	**397**	**14**	**114**
4	213	18	106
5	74	22	99

and increased overhead. Therefore, we choose the three level HBRICK counters for our performance measurements.

Fig. 7 shows a drop in clock frequency as the number of entries in each hash table increases. As the depth of the table increases, the BRAM size also grows. The figure illustrates that when the BRAM exceeds a certain threshold, the operating frequency drops significantly due to the extended data path required to access the larger BRAM. Reshaping the counters array and dividing it into multiple memory banks can slightly increase the frequency, but the clock frequency still experiences a reduction.

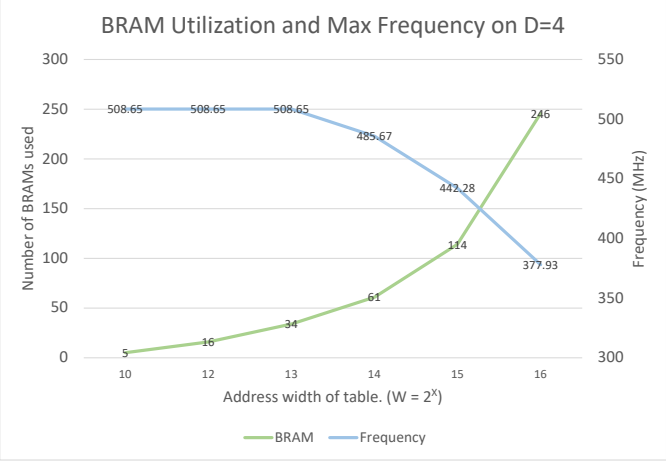

Fig. 7. BRAM utilization and F_{max} of Our Design with fixed number of hashes $D = 4$ while increasing the size (W) of each counter.

TABLE II
BRAM UTILIZATION COMPARISON BETWEEN DIFFERENT IMPLEMENTATIONS

Implementations	Freq. (MHz)	Overhead (cycles)	BRAMs
Regular CM	453	2	153
Original BRICK	156	42	103
Ours	**397**	**14**	**114**

Based on the above results, we select $D = 4$ and $W = 2^{15}$ with three layers $L = 3$ of variable counters as our design configuration where the design can run at a maximum 397MHz. It provides a relatively low overestimation error while maintaining a high operating frequency. Using a total of 4 hash functions provides good end-to-end throughput results. Assuming network packets of 64 bytes (the minimum size) as our input stream and that our pipeline can process one network packet in each cycle, our design can achieve a $195Gbps$ throughput with this frequency.

Finally, we implement the other two count-min sketches. One is with the traditional counters and the other is with the original BRICK counters. Table. II shows the BRAM utilization and its max frequency achieved. The table shows that our implementation outperforms the original BRICK in frequency and achieves about 18.9% BRAM utilization improvement.

TABLE III
AVERAGE ABSOLUTE ERROR ON DIFFERENT DATASETS

Datasets	Skewness	CM	Original BRICK	Ours
Zipf	0	1748.4	116.7	**114.3**
Zipf	0.25	1712.3	119.5	**121.6**
Zipf	0.5	1361.2	142.5	**142.5**
Zipf	0.75	1642.5	154.3	**156.1**
Zipf	1	472.5	125.3	**125.3**
Zipf	1.25	129.6	104.4	**102.2**
Zipf	1.5	6.3	8.3	**7.4**
CAIDA-1	-	432.5	105.8	**105.3**

C. Design Accuracy Profiling

In this section, we use two types of datasets to measure the accuracy and miss rate of our implementation. The first dataset consists of real network traces captured by CAIDA in 2016, containing 3 million packets. The second dataset is synthetically generated to follow the Zipf distribution [20]. For the Zipf distribution, we generate multiple datasets with varying degrees of skewness, determined by the exponent s. The skewness ranges from 0 to 1.5, with all Zipf datasets containing 3 million packets. As the skewness increases, the total number of flows decreases, resulting in more heavy flows.

We measure the performance of the sketches using the average absolute error, defined as the average error between the real and estimated sizes of all flows. Our design is compared to the traditional count-min sketch (CM), which employs a fixed width counter. Both designs use a similar number of BRAMs.

As shown in Table III, our design demonstrates superior accuracy when the data is skewed. Given the similar total memory usage, overflows are more likely to occur when using the fixed width counters, leading to significant errors. However, as skewness continues to increase, our variable width counter design also shows increased error. When the skewness exceeds 1.25, predominantly heavy flows cause our design to suffer from overflows. Despite this, our design performs well with real world CAIDA datasets and Zipf distribution when the skewness is around 1.

D. End-to-End Results

We implemented the entire design with the selected configurations and tested it on OCT FPGA nodes as described in Sect. IV. Our overall design achieves the resource utilization for each block shown in Table. V.

In Table. IV, we compare our design with other FPGA implementations. Among all the listed implementations, we measured the real-time throughput through the existing FPGA-based SmartNIC framework. Other implementations provide their throughput based on the generated design frequency. Our measured throughput of this design is about 92Gbps on

TABLE IV
PERFORMANCE COMPARISON OF VARIOUS FPGA-BASED DESIGNS

Design	Counter Type	Device	Theoretical Throughput (Gbps)	Real-time (Gbps)	D, W	$1-\delta$
[19]	exact	Xilinx Virtex 2 XC2V1000	61	-	4, 2^{12}	-
[25]	exact	Xilinx Virtex Ultrascale XCVU440	155	-	5, 2^{16}	-
[25]	exact	Xilinx Virtex Ultrascale XCVU440	146	-	10, 2^{15}	0.999
[15]	exact	Xilinx Alveo U250	575 (with three CM kernels)	-	-	0.99
[21]	approx.	Xilinx Virtex UltraScale+	196	-	4, 2^{16}	0.98
[21]	approx.	Xilinx Virtex UltraScale+	212	-	5, 2^{15}	0.99
HBRICK	exact	Xilinx Alveo U280	195	92/100	4, 2^{15}	0.98

TABLE V
OVERALL RESOURCE UTILIZATION

	BRAM	LUTs	URAM	DSPs
OpenNIC shell	16.54%	8%	1%	0
P4 Front end	4.6%	3.4%	0	0
Sketch engine	30.6%	1%	0	0

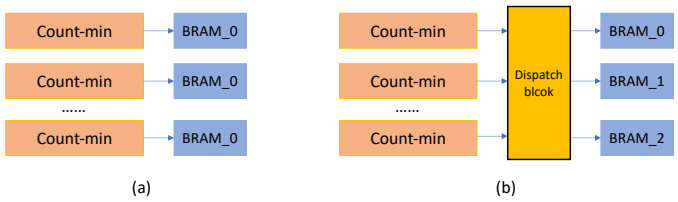

Fig. 8. Shared Memory for Parallel Engines. (a) Pessimistic architecture; (b) Optimistic architecture.

one Ethernet port of the AMD U280. Our design also shows a comparable design frequency and theoretical maximum throughput with other implementations. In [21], the authors implement an approximate counter design called HSAC, which adopts the Simple Active Counter (SAC), a counter that uses a representation similar to floating-point numbers. As shown in Table. IV, our design demonstrates better accuracy compared to this approximation-based counter. The design presented in [15] shows high performance results using multiple count-min kernels and demonstrating cumulative performance. Their architecture primarily addresses data conflicts between these parallel kernels, using normal counters with universal bitwidth. Our design is orthogonal to their architecture and can further enhance memory efficiency. A detailed discussion will be covered in the next section.

Overall, our design demonstrates comparative performance in accuracy and throughput with reduced memory usage compared to other FPGA-based implementations.

VI. DISCUSSION AND FUTURE WORK

This paper presents a novel variable width counter architecture for count-min sketch algorithm with reduced memory resource utilization compared to traditional count-min sketch implementations. In Sec. V, we demonstrate the advantage of using our design. Currently, we have implemented a single sketch engine on an Alveo U280 FPGA. An improvement we plan for the future is to increase the number of parallel engines using straightforward duplication. In [15], the authors refer to this as a "pessimistic" architecture and they propose an "optimistic" architecture that uses shared memory as shown in Fig. 8 with really high-performance as shown in Table. IV. Our design is a natural fit for this parallel architecture as we have already divided the counters into numerous small buckets. For future work, we plan to explore our design's performance with parallel engines.

Our design uses P4 controlling packet processing while offloading the sketch task to an external function implemented using HLS. It demonstrates a design methodology of using the network domain-specific language P4 block to describe network behaviors of packet processing and HLS to achieve high throughput processing units. This methodology also allows us to orchestrate network flows among multiple HLS-based compute blocks using P4. Another direction for future work is to exploit and design more complicated network traffic monitoring systems with the combination of P4 and HLS.

VII. CONCLUSION

In this paper we present HBRICK, a novel hardware architecture for count-min sketch with variable width counters, which allow for larger traffic flows to be handled by FPGAs processing data traffic at the edge. The original design for variable width counters ignored data hazards introduced by its recursive update mechanism and inserted extra processing overhead. In contrast, the HBRICK architecture solves this problem by redesigning the counters and supporting hardware. We implemented our design on FPGAs connected to 100Gb Ethernet network connections. We combine a hardware implementation of the count-min sketch implementation, described in HLS, with a P4 packet processing front end and demonstrate the ability to keep up with the line rate on the network.

Our implementation and experiments demonstrate one application of network-attached FPGAs with the combination usage of the two high-level abstractions P4 and HLS. Our design constitutes a case study of a potential framework where network-attached FPGA can bypass the host CPU it attaches to and directly extract the necessary information from the network packets through P4 and process them using HLS.

In the future, we plan to improve the performance of our sketch architecture by exploiting parallelism at several levels. This parallelism will not be limited to a single FPGA, but will be implemented across multiple network-connected FPGAs in the data center to construct a complete networking monitoring system.

REFERENCES

[1] AMD OpenNIC Project. https://github.com/Xilinx/open-nic, 2023. [Online; accessed 07-30-2023].

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[3] The CAIDA UCSD anonymized 2016 internet traces. https://www.caida.org/data/passive/passive_2016_dataset. Accessed on 30-07-2023.

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

[5] Grigorios Chrysos, Odysseas Papapetrou, Dionisios Pnevmatikatos, Apostolos Dollas, and Minos Garofalakis. Data stream statistics over sliding windows: How to summarize 150 million updates per second on a single node. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 278–285. IEEE, 2019.

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

[7] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 61–72, 2002.

[8] DPDK. https://www.dpdk.org, 2023. [Online; accessed 07-30-2023].

[9] Zhaoyang Han, Suranga Handagala, Kalyani Patle, Michael Zink, and Miriam Leeser. A Framework to Enable Runtime Programmable P4-enabled FPGAs in the Open Cloud Testbed. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6. IEEE, 2023.

[10] Suranga Handagala, Miriam Leeser, Kalyani Patle, and Michael Zink. Network Attached FPGAs in the Open Cloud Testbed (OCT). In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6. IEEE, 2022.

[11] Nan Hua, Bill Lin, Jun Xu, and Haiquan Zhao. Brick: A novel exact active statistics counter architecture. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 89–98, 2008.

[12] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4 to netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–9, 2019.

[13] Intel Corporation. Intel programmable ethernet switch - tofino series. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. Accessed: Insert date here.

[14] Martin Kiefer, Ilias Poulakis, Sebastian Breß, and Volker Markl. Scotch: Generating fpga-accelerators for sketching at line rate. *Proceedings of the VLDB Endowment*, 14(3):281–293, 2020.

[15] Martin Kiefer, Ilias Poulakis, Eleni Tzirita Zacharatou, and Volker Markl. Optimistic Data Parallelism for FPGA-Accelerated Sketching. *Proceedings of the VLDB Endowment*, 16(5):1113–1125, January 2023.

[16] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247, 2003.

[17] Amit Kulkarni, Monica Chiosa, Thomas B. Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. HyperLogLog Sketch Acceleration on FPGA. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 47–56, August 2020. ISSN: 1946-1488.

[18] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 334–350, New York, NY, USA, 2019. Association for Computing Machinery.

[19] David Nguyen, Gokhan Memik, Seda Ogrenci Memik, and Alok Choudhary. Real-time feature extraction for high speed networks. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 438–443. IEEE, 2005.

[20] David MW Powers. Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*, 1998.

[21] Arish Sateesan, Jo Vliegen, Simon Scherrer, Hsu-Chun Hsiao, Adrian Perrig, and Nele Mentens. Speed Records in Network Flow Measurement on FPGA. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 219–224, August 2021. ISSN: 1946-1488.

[22] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 571–592, 2021.

[23] Da Tong and Viktor Prasanna. Online heavy hitter detector on FPGA. In *2013 International conference on reconfigurable computing and FPGAs (reconfig)*, pages 1–6. IEEE, 2013.

[24] Da Tong and Viktor Prasanna. High Throughput Sketch Based Online Heavy Hitter Detection on FPGA. *ACM SIGARCH Computer Architecture News*, 43(4):70–75, April 2016.

[25] Da Tong and Viktor K. Prasanna. Sketch Acceleration on FPGA and its Applications in Network Anomaly Detection. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):929–942, April 2018. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[26] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.

[27] Michael Zink, David Irwin, Emmanuel Cecchet, Hakan Saplakoglu, Orran Krieger, Martin Herbordt, Michael Daitzman, Peter Desnoyers, Miriam Leeser, and Suranga Handagala. The Open Cloud Testbed (OCT): A platform for research into new cloud technologies. In *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, pages 140–147. IEEE, 2021.