electronics

MDPI

Article

Finding the Top-K Heavy Hitters in Data Streams: A Reconfigurable Accelerator Based on an FPGA-Optimized Algorithm

Ali Ebrahim

Department of Computer Engineering, University of Bahrain, Sakhir P.O. Box 32038, Bahrain; ahasan@uob.edu.bh; Tel.: +973-17437029

Abstract: This paper presents a novel approach for accelerating the top-k heavy hitters query in data streams using Field Programmable Gate Arrays (FPGAs). Current hardware acceleration approaches rely on the direct and strict mapping of software algorithms into hardware, limiting their performance and practicality due to the lack of hardware optimizations at an algorithmic level. The presented approach optimizes a well-known software algorithm by carefully relaxing some of its requirements to allow for the design of a practical and scalable hardware accelerator that outperforms current state-of-the-art accelerators while maintaining near-perfect accuracy. This paper details the design and implementation of an optimized FPGA accelerator specifically tailored for computing the top-k heavy hitters query in data streams. The presented accelerator is entirely specified at the C language level and is easily reproducible with High-Level Synthesis (HLS) tools. Implementation on Intel Arria 10 and Stratix 10 FPGAs using Intel HLS compiler showed promising results—outperforming prior state-of-the-art accelerators in terms of throughput and features.

Keywords: top-k heavy hitters; data streams; Field Programmable Gate Arrays; High-Level Synthesis

Citation: Ebrahim, A. Finding the Top-K Heavy Hitters in Data Streams: A Reconfigurable Accelerator Based on an FPGA-Optimized Algorithm. *Electronics* **2023**, *12*, 2376. https://doi.org/10.3390/electronics12112376

Academic Editor: Raffaele Giordano

Received: 11 April 2023
Revised: 13 May 2023
Accepted: 23 May 2023
Published: 24 May 2023

1. Introduction

Extracting a list of the most frequently occurring items (aka. heavy hitters) from large datasets is a well-studied problem that is usually tackled with approximation techniques due to the complexity and size of the problem [1,2]. Several approximation techniques model the input as a "data stream" consisting of a sequence of items that needs to be processed in a one-pass manner at high speed and using limited memory [3]. The heavy hitter problem has applications in many fields, such as network traffic monitoring [4], website data analysis [5], and sensor networks [6]. A sub-class of the heavy hitter problem is the "top-k" problem, wherein a user would query the k most frequent items in a data stream. Examples of such queries include the top-visited websites in web data, the most frequent destination IPs in network traffic passing through a networking device, the bestselling products in retail data, etc.

In recent years, data stream algorithms have been deployed by companies such as Google, Apple, Microsoft, etc. to address several computational problems [7]. With the growing demand for high-speed data stream processing, several custom hardware accelerator architectures have emerged (see Section 2). In general, these accelerators rely on parallelism and deep pipelining to achieve the required processing throughputs. Field Programmable Gate Arrays (FPGAs) are typically used to implement such accelerators due to three main reasons: First, the flexibility of FPGAs allows one to change the accelerator hardware configuration so that it is tailored for specific stream distributions or specific user requirements. For example, some hardware configurations would favor accuracy, while other configurations would favor higher throughputs. Second, stream algorithms usually summarize the properties of the data stream in small data structures referred to as "stream summaries". The amount of fast on-chip SRAM memory in a modern FPGA is sufficient for

implementing practical stream summaries without the need to access the slower external DRAM. Third, the extensive Intellectual Property (IP) support and flexible high-bandwidth Input/Output (I/O) in FPGAs allow for the easy integration of FPGAs in edge and cloud applications that deploy data stream analytics.

With the advances in High-Level Synthesis (HLS) tools, functional hardware implementations of software algorithms can be easily and quickly realized. However, design optimizations are often necessary to achieve the optimal operation of hardware accelerators. The main goal in HLS design is to be able to implement a stall-free accelerator or, in other words, an accelerator with an Initiation Interval (II) of 1. An II of 1 means that the accelerator pipeline can process a new input item every clock cycle without stalling. An optimal stall-free accelerator is not always possible, especially if the accelerator requires complex memory accesses. For example, a hardware implementation of a hash table will not maintain an II of 1 due to the pipeline stalls needed to resolve collisions when inserting items in the table.

There are several complexities associated with designing an optimal accelerator to compute the heavy hitters in data streams, depending on the base algorithm used and the overall system requirements. For example, several heavy hitter algorithms utilize hash table data structures for counting item occurrences in a data stream, preventing a stall-free operation due to hash collisions and memory dependency. Additionally, if the accelerator is required to support the top-k item query, this adds further complexity to the design. A suitable data structure, such as a priority queue, needs to be implemented to maintain a list of the top k items. In addition, a suitable interface is needed for the host to traverse the top-k item list. Several hardware architectures have been proposed in the literature to accelerate the heavy hitter problem in data streams (See Section 2). However, there is no single elegant solution to handle the aforementioned complexities without scaling down the design. This paper presents a novel hardware adaptation of the approximate *Probabilistic* sampling algorithm in [8], which is used for the top-k item query in data streams. The presented hardware architecture aims to address design complexities in existing solutions by introducing hardware-specific optimizations at the algorithmic level. These modifications are based on intuition and favor simplicity and design scalability to facilitate the strict mapping of the algorithm into hardware. When implemented as an HLS kernel using Intel HLS compiler and targeting an Intel Stratix 10 FPGA, the proposed architecture scaled very well and achieved higher throughputs compared to all relevant existing FPGA accelerators. Furthermore, test results on synthetic and real datasets showed near-perfect accuracy—exceeding 95% in all test runs. This is a significant improvement over previously proposed scaled down accelerators that would strictly map the *Probabilistic* sampling algorithm into hardware. In short, the main contributions in this paper can be summarized as follows:

(1) A novel hardware-optimized algorithm for computing the top-k query in data streams. The algorithm is the first to deploy techniques such as fingerprinting, optimistic counting, re-hashing, and timestamping to resolve hardware-specific complexities usually associated with relevant FPGA accelerators.
(2) An HLS kernel design for the proposed optimized algorithm that can be easily reproduced using HLS tools from both major FPGA vendors (AMD/Xilinx and Intel). The HLS kernel also deploys novel optimizations at the hardware level to resolve common implementation issues such as memory dependency and data hazards.
(3) The fastest FPGA implementation compared to existing accelerators, achieving high throughputs even when the implementation has a high chip utilization.
(4) Addressing important practicality issues in kernel design such as larger key sizes (up to 128-bit), result mergeability, and parallelism.

The remainder of this paper is organized as follows: Section 2 briefly introduces the top-k query problem in data streams and discusses some of the most relevant previously published work. Section 3 briefly discusses the *Probabilistic* sampling algorithm, which is used as the basis for the top-k item query computation. Section 4 presents the proposed

optimizations for efficient hardware implementation. Sections 5 and 6 details the HLS accelerator kernel design and FPGA implementation. Section 7 presents the functional verification and evaluation of the proposed accelerator. Finally, conclusions and future work plans are summarized in Section 8.

2. Background and Related Work

2.1. The Top-k Item Query in Data Streams

Assume we have a stream S of size N, and we want to find the k most frequent items in S (known as top-k items or top-k heavy hitters). The size of the stream N is not necessarily known beforehand. A naïve exact solution for finding the top-k items can be realized using a lookup table data structure with key-count pairs in its entries. For every item hit, if the item key is in the table, its count value is updated. Otherwise, a new entry is created in the table. An additional priority queue data structure can be used to maintain a record of the current top-k items. Alternatively, the entries in the lookup table can be sorted according to their count values to extract the top-k items when the stream is exhausted or when results are required. For a general input distribution containing a large number of distinct items, finding an exact solution is impractical or even impossible due to the time and space complexity.

In most practical applications, an exact result is not required. Approximate results can be obtained using approximate algorithms that do not count every distinct item in the stream. In general, approximate algorithms only maintain a summary of the stream and a list of heavy hitter candidates that most likely contain most of the actual top-k items in the stream. There are two categories of such approximate algorithms in the literature: counter-based and sketch-based algorithms.

2.2. FPGA Implementations of Counter-Based Algorithms

In general, counter-based algorithms allocate counters in memory, only enough for counting a small subset of the overall distinct items in the stream. Each counter is a key-value tuple, where the key is an identifier for a heavy hitter candidate and the value is the estimated count for this candidate. When the stream is processed, the counters should contain all or most of the heavy hitters in the stream. Counter-based algorithms do not require an additional priority queue for computing the top-k items, as this can be performed by using a simple sort operation.

Several architectures have been proposed to accelerate item counting using FPGAs. Early works on the related field of itemset mining acceleration with FPGAs proposed implementing fine-grained systolic arrays with serially connected Processing Elements (PEs). Each PE contains a small independent memory allocated for updating a single or small number of key-value tuples [9]. A limited number of such systolic array accelerators have been specifically designed to compute the heavy hitters in data streams [10–13]. Most of these accelerators implement the popular *Space-Saving* algorithm [14]. *Space-Saving* uses m counters to monitor the first m distinct items that appear in the stream. A new incoming item, not in the any of the m counters, replaces the item with the minimum count. By doing so, frequent items with large counts should remain in some of the counters when the stream is exhausted. Although implementing *Space-Saving* is relatively simple in software, mapping it to a systolic array architecture can be tricky with complexities that limit the overall number of monitored items. Current *Space-Saving* FPGA implementations can be used to monitor hundreds to a few thousands of items using mid-capacity and large-capacity FPGA chips [11–13]. The work in [10] showed that the *Probabilistic* sampling algorithm proposed in [8] maps better to a systolic array architecture, resulting in some notable improvements compared to *Space-Saving*.

All the aforementioned systolic array accelerators rely solely on the FPGA logic resources for implementing small distributed memories to store the key-value tuples. Although they are stall-free and relatively fast, the maximum number of items that can be monitored is limited, especially with larger key integer sizes. Several works have

demonstrated that it is possible to expand the total number of monitored items using key-value store approaches based on hash tables that utilize the abundant embedded memory resources on an FPGA chip [15,16]. However, the performance of such approaches significantly suffers because of hash collisions and the pipeline stalls needed for updating the monitored items.

2.3. FPGA Implementations of Sketch-Based Algorithms

Sketch-based algorithms aim to summarize the frequency distribution of the data stream using a unique data structure referred to as a "sketch". The frequency estimation sketch can be queried to output the estimated count of a particular item key. A very popular frequency estimation sketch is the *count–min* sketch [17]. The *count–min* sketch consists of several hash tables with different hash functions. The hash tables only contain count values in their entries. When the sketch is updated with a new item hit from the stream, the item key is hashed into all of the tables, and the relevant table entries are incremented. Since the hash functions are different, item collisions will differ in all of the tables. For any queried item, the table entry with the minimum count (minimum number of collisions) represents the best count estimate for the item.

The count–min update process does not require hash collision resolution. Several FPFA accelerators implement *count–min* on the on-chip embedded memory to realize constant update time and, in some cases, stall-free operation [18–23]. As the *count–min* does not store item keys in its table entries, it cannot be directly used to solve the top-k item query problem. An additional hardware data structure is required to maintain a record of the top key-value tuples [24–26]. Only a few FPGA implementations extend the basic functionality of *count–min* to support the top-k query. The sketch accelerator in [27] uses a simple priority queue architecture. Because the queue update process is sequential, the throughput can be drastically reduced, even for small values of k. A more sophisticated and improved priority queue was later presented in the accelerator proposed in [28,29]. This accelerator uses a large portion of the available embedded memory resources for implementing the queue rather than for the sketch. The queue is implemented as a pipelined hash table with several independent buckets to allow for consecutive updates (1 update per clock cycle in most cases). This allows one to monitor a larger number of top items, but with reduced count accuracy due to the smaller sketch. In fact, the main objective of this accelerator was not to output the top-k items accurately but to estimate the entropy of the input stream using the top-k item list as a sample of the stream.

A related work deploys a hybrid approach, combining a sketch implemented on embedded memory and a systolic array implemented using the logic resources of the FPGA. The sketch is only used as a filter that passes item hits for items with counts larger than a specific threshold to the systolic array that monitors the heavy hitters [10]. While this architecture achieved good performance, it only allowed one to monitor a relatively small number of items. Another related work implements an accelerator based on an alternative sketch algorithm that supports the top-k query without the need for a priority queue [30]. The implemented *Heavy-Keeper* sketch uses hash tables similar to *count–min*; however, item keys are also stored in the table entries, allowing for the sketch to be traversed to extract the top-k items [31]. Due to the complexity in updating the *Heavy-Keeper* sketch, the item update process required several clock cycles (II larger than 1).

2.4. Summary of Existing FPGA Implementations

Table 1 summarizes the most relevant existing FPGA implementations of data stream heavy-hitter detection algorithms. Implementations are classified into three categories: systolic array, hash table, and sketch implementations. These categories are compared according to the following attributes:

Design: A stalling design means that the accelerator cannot guarantee that an item is processed in a single clock cycle, resulting in a slow FPGA implementation.
Key Size: Smaller key sizes limit the possible applications.

Top-k Query Support: Several implementations do not directly support the top-k query as an additional priority queue is needed.
Result Readout Style: Using a buffer implemented as a memory block allows the results to be easily copied and processed by a host. However, a large buffer increases latency and host processing time. A First-In First-Out (FIFO) interface is present in accelerators that are only capable of outputting results for individual item count queries.

Table 1. Summary of FPGA implementations of data stream heavy-hitter detection algorithms.

Ref.	Type	Algorithm	Stalling Design?	Key Size (Bits)	Top-k Query?	Result Readout
[10]	Systolic Array	Probabilistic [8]	No	32	Yes	Small Buffer
[11–13]	Systolic Array	Space-Saving [14]	Yes	32	No	FIFO
[15,16]	Hash Table	Cukoo Hash [32]	Yes	104	Yes	Large Buffer
[18–23]	Sketch	Count–Min [17]	No	32-128	No	FIFO
[30]	Sketch	Heavy-Keeper [31]	Yes	32	Yes	Large Buffer

From Table 1 we can see that there is no single FPGA implementation that excels in all attributes. Therefore, the presented work focuses on filling this gap by addressing all the attributes needed to realize a fast and practical FPGA accelerator specifically tailored for the top-k query in data streams.

3. Base Algorithm: Probabilistic Sampling

Our proposed approach for finding the top-k items borrows several ideas from the *Probabilistic* sampling algorithm in [8]. We first briefly introduce *Probabilistic* sampling in this section and, later, we detail the proposed hardware-specific optimizations in Section 4.

Probabilistic sampling is generally considered fast and efficient, and can approximate a list of the top-k items in data streams. The idea behind *Probabilistic* sampling is very simple and as follows: the data stream is divided into rounds of size r that are processed separately. The algorithm uses m counters to count the first m distinct items that appear in the round (see Figure 1). Hits from items not registered in the m counters are discarded. A hash table with key-value entries is a fast and simple method for counting the sampled items. At the end of a round, the k items with the largest round counts are extracted and stored in a list. The process is then repeated in later rounds of the stream. The final list of approximate top-k items is obtained by merging the top-k lists at the end of each round. Only the items with the highest round counts make it to the final list. For duplicate item records, only one entry with the highest round count is stored in the merged list.

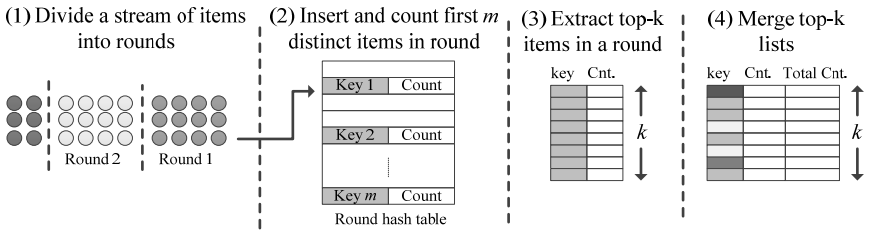


Figure 1. Probabilistic sampling algorithm.

In addition to the highest round count for an item in the top-k list, an extra total accumulate count field can also be stored for each item. This field represents an underestimated value for the item count. The value in this field is obtained by accumulating the round counts of items appearing in the top-k list in consecutive rounds. If an item is recorded as a top item in all rounds, then the total count estimate is the actual count of this item.

4. Proposed Approach: Optimizations for Efficient Hardware Implementation

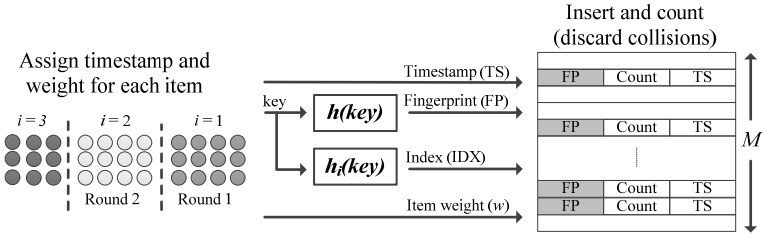
4.1. Updating the Round Table

To implement the *Probabilistic* algorithm on an FPGA, a hash table is needed for counting items in a round. The hash table can be implemented as a RAM block. The aim is to store as many key-value pairs as possible using the available on-chip memory. In addition, a stall-free operation is necessary to achieve high throughput. The main obstacle in achieving a stall-free operation is the item collision issue. There are some trivial methods to minimize hash collisions. For example, we can increase the load factor of the hash table by using a RAM block with M buckets to count m distinct items ($M > m$). Additionally, it is possible to use an additional bloom filter data structure to decrease the number of table updates [33]. However, using these methods will increase the memory usage and limit the number of items that can be monitored in a round.

Optimizing memory accesses and the RAM block geometry can also enhance the performance of a hash table. For example, breaking the RAM block into several pipelined banks allows for several concurrent memory accesses [16,33]. However, real data streams are typically skewed, causing contention on some of the memory banks and, as a result, preventing meaningful performance gains. While there are some methods that have been proposed to reduce contention on the memory banks when processing has skewed streams [34,35], there is no efficient solution that guarantees a stall-free operation.

Instead of strictly mapping the round table update process in *Probabilistic* models into hardware, we modify this process to address the aforementioned shortcomings associated with hardware hash tables. The remainder of this section details the proposed optimizations for the round table update process. Figure 2 shows the pseudo code for the optimized *update_table()* function, which returns the round count for a particular item hit in a round. The function has four arguments:

(1) FP: item fingerprint.
(2) TS: timestamp.
(3) IDX: table index generated by a hash function.
(4) w: weight of an item ($w = 1$).




```
Function update_table(FP, TS, IDX, w):
    c ← 0
    if table[IDX].TS ≠ TS then
        table[IDX].FP ← FP
        table[IDX].TS ← TS
        table[IDX].count ← w
        c ← w
    else if table[IDX].FP = FP then
        table[IDX].count ← table[IDX].count + w
        c ← table[IDX].count
    return c
```

Figure 2. The proposed optimized function for updating the round table.

4.1.1. Optimistic Counting

As mentioned earlier, the main obstacle in achieving a stall-free operation is item collisions when counting the first m distinct items in a round. Rather than increasing the hash table load factor to decrease item collisions, we opt for an "optimistic counting" approach that ignores item collisions. The idea behind optimistic counting is simple and can be described in four steps as follows:

(1) An item hit is hashed to generate an index (IDX) to a bucket in the table.
(2) If the indexed bucket is empty, create an entry for the item in the bucket.
(3) If the indexed bucket already contains an entry for the item, increment the round count of the item.
(4) If the indexed bucket contains an entry for another item, discard the item hit.

We can see that the optimistic counting approach utilizes all M buckets for item counting opposite to the conventional hash table approach, which only inserts m items in the M buckets ($M > m$). The first item to be hashed into an empty bucket will stick in the bucket for the remainder of the round. Item collisions are totally ignored in favor of sampling more items and achieving a stall-free operation in hardware. We refer to the optimized function as "optimistic" because it assumes, or in other words, hopes that there will be no item collisions before, at least, the m items are inserted into the table. Due to the simplicity of this approach, we can count a significantly larger number of items using the same amount of FPGA embedded memory compared to a conventional hash table with a large load factor. In addition, to allow for a stall-free operation, we can also argue that the proposed simplified item counting technique can result in better accuracy compared to a hash table with a large load factor when the amount of embedded memory is limited. This is mainly attributed to the larger number of sampled items using the same amount of memory.

4.1.2. Fingerprinting

As the on-chip memory in FPGAs is generally limited, it is very important to optimize memory usage to be able to sample as many items as possible. Most of the available accelerators discussed in Section 3 limit the key size of an item to 4 bytes. While this is sufficient for some applications (example: IP address in IPv4), there are other applications that require larger key sizes (example: 128-bit IP address in IPv6). Storing the full item keys in the round table buckets will limit the total number of buckets possible with the available memory, especially for larger key sizes. As we are only interested in maintaining a record of the top-k items, there is no need to store the keys for all the sampled items. Alternatively, we can store a unique fingerprint (FP) of the item in the round table [31]. This fingerprint is generated by a hash function and is much smaller in size compared to the actual key (see Figure 2). Simply, when calling the optimized *table_update()* function, the function matches the generated item fingerprint to the fingerprint stored in the indexed table bucket to decide if the round count should be incremented.

4.1.3. Round Re-Hashing

As our simplified optimistic counting technique ignores item collisions, we need to consider the case when two or more heavy hitter items generate the same index early in the round. Only one of these heavy hitters will be registered in a round and considered as a top-k candidate. In addition, there is a possibility of different items generating the same table index as well as the same fingerprint.

To rectify the issue of colliding heavy hitters, we propose round re-hashing. Basically, the seed for the hash function used to generate the item index is randomly generated for each round (labelled $h_i(key)$ in Figure 2). The idea behind round re-hashing is simple and as follows: if two heavy hitter items collide in a round, it is highly unlikely that the same items will collide with each other again in another round, as they will likely generate different hashes.

4.1.4. Using a Timestamp for Reduced Latency

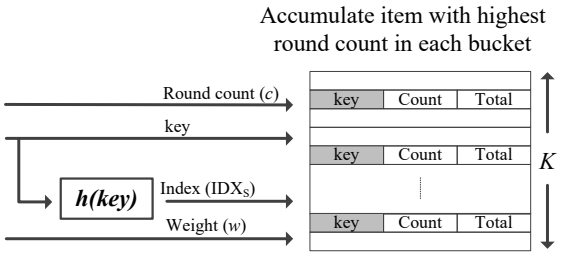
Another issue that needs to be addressed when updating the round table is the RAM block reset needed in between rounds. As we are going to use the same RAM block for counting items in different rounds, all buckets must be reset before starting a new round. This is time-consuming, as the memory locations in the RAM block need to be reset sequentially. The latency of the accelerator will significantly increase, especially if the RAM block is large and the chosen round size is not sufficiently large relative to the RAM block size.

To address this issue, we propose using a unique timestamp for every round. Items in the same round will be assigned the same timestamp. When an item is first registered in a table bucket, the current timestamp is also stored in the bucket (see Figure 2). When updating the table, if an indexed bucket has a timestamp different to the current timestamp, the bucket is considered empty and can be used to register a new item. This way, there is no need to reset the RAM block after each round. To avoid significantly increasing the memory usage of the round table, only small numbers should be used for the timestamp. For example, if 1 byte is allocated for the timestamp, this allows one to run 255 rounds before a reset of the RAM block is required.

4.2. Updating the Heavy Hitter Summary

After counting items in a round, the top-k frequent items in the round should be extracted and stored in a list. In hardware, the process of updating the top-k item list should run concurrently with the table update process to achieve a stall-free operation. Typically, a priority queue data structure is used to maintain a record of the top-k items; however, FPGA implementations of such data structures can be complex and inefficient, especially if a stall-free operation is required (see Section 2).

We propose a data structure that is entirely different to a priority queue. We call this data structure the "heavy hitter summary". The summary is a hash table with K buckets, where K is much larger than k but still much smaller than M. Figure 3 shows the pseudo code for the *update_summary()* function, which is called after the *update_table()* function in Figure 2.



Function `update_summary`$(key,\ IDX_S,\ c,\ w)$**:**
- **if** $summary[IDX_S].key = key$ **then**
  - $summary[IDX_S].total \leftarrow summary[IDX_S].total + w$
- **if** $summary[IDX_S].count < c$ **then**
  - $summary[IDX_S].count \leftarrow c$
  - **if** $summary[IDX_S].key \neq key$ **then**
    - $summary[IDX_S].key \leftarrow key$
    - $summary[IDX_S].total \leftarrow w$

Figure 3. Updating the heavy hitter summary.

Each bucket in the summary contains three fields: an item key, round count, and total accumulate count for the item. The *update_summary()* function arguments are the key for the current item hit, the bucket index (IDX_s) generated by the hash function, the item's round

count returned by the *update_table()* function, and the weight of the item. The procedure for updating the summary can be summarized as follows:

(1) An item hit is hashed to generate an index (IDX_s) to a bucket in the summary.
(2) If the key matches the key stored in the indexed bucket, the accumulate count is incremented. The stored round count is also updated in case the current round count is larger than the stored count.
(3) If the key is different than the stored key in the indexed bucket, the bucket is updated with the new item only if the new item round count is larger than the stored round count.

The *update_summary()* function is called for every item hit in the stream. The summary is only reset when the stream is exhausted. The function basically splits the stream into K sub-streams using the hash function. Only the heaviest item from each sub-stream is maintained in the relevant bucket (the item with the largest round count). When K is sufficiently large relative to k, most of the actual top-k item should stick in some of the summary's buckets. Querying the top-k items from the summary is simple and involves the following: first the items in the summary are sorted according to their accumulate count. Then, the top-k items are extracted. The latency incurred from the sorting operation should not affect the performance when implementing the proposed summary in hardware. This is because, in practical applications, a top-k query is only issued intermittently after very large intervals of streaming activity. Additionally, K is generally small, and the sorting operation can be efficiently completed in software by a host.

5. HLS Kernel Architecture

This section details the design of an FPGA accelerator implementing the proposed algorithm in Section 4. We draw the readers' attention to the Intel HLS documentation [36], as the remainder of this section uses technical terminology that may be specific to Intel HLS design flow. As the presented design only deploys standard pragmas, the design can be easily migrated to other HLS tools (for example, AMD/Xilinx Vitis HLS). In addition, with minor modifications, the accelerator can be deployed as a kernel in Intel heterogeneous computing tools such as the following: Intel FPGA SDK for OpenCL and Intel OneAPI toolkit. Since these tools use the same core compiler technology as Intel HLS, results should be the same regardless of which Intel design tool is used.

5.1. Architecture Overview

The accelerator is designed as a kernel that operates alongside a host CPU, which is the typical hardware setup in streaming applications. The kernel is implemented as a slave component that is controlled by a host through a memory mapped slave interface (see Figure 4). In Intel HLS compiler, the "hls_avalon_slave_component" attribute can be used to infer a slave interface compatible with the Avalon bus specification. The host launches the kernel to process a single round of the stream. There are some memory mapped registers that need to be setup by the host before launching the kernel, including the following: the round size (r), the timestamp of the round (TS), and the hash function seed needed for generating the round table index (IDX), as explained in Section 4. The kernel contains two memory blocks, one represents the round table with M memory locations and the other represents the heavy hitter summary with K memory locations. Both memory blocks are implemented as simple dual-port RAM using the M20K embedded memory resources in Intel FPGAs. A simple dual-port memory has a read port dedicated to reading (load operations) and a write port dedicated to writing (store operations). Using the relevant component macros in Intel HLS compiler, the heavy hitter summary is specified as a slave memory with read access granted to the host. By doing so, the compiler inserts arbitration logic at the read port of the heavy hitter summary to allow the host to read the summary when results are required. For a simpler arbitration logic, we prevent host access during a round when the kernel is active.

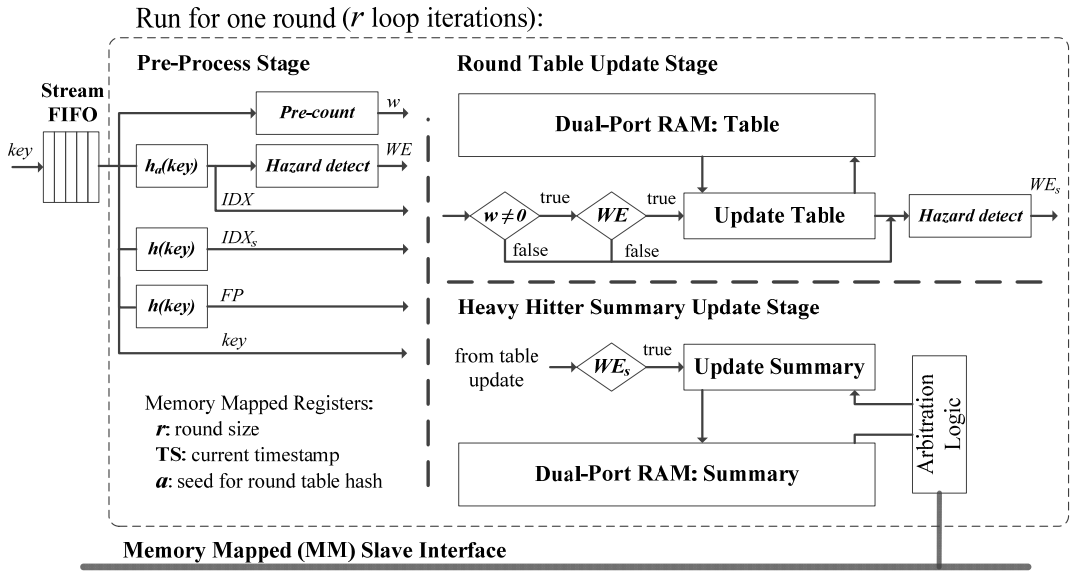


Figure 4. Architecture overview of the HLS kernel.

After launching the kernel, the kernel runs for r iterations. In every iteration a blocking read operation from an input stream FIFO is performed to read the item keys from the data stream. The kernel pipeline can be divided into three main stages: the pre-process stage, the round table update stage, and the heavy hitter summary update stage. In the pre-process stage, the input keys are processed to generate the hashes required in the proposed algorithm. In addition, some control signals are generated to efficiently handle data hazards in later stages in the pipeline. The round table update stage basically executes the function in Figure 2, while the heavy hitter summary update stage executes the function in Figure 3. While a functional hardware implementation of the proposed algorithm in Section 4 is very straightforward using HLS, achieving an II of 1 requires some design optimizations, which will be detailed in the remainder of this section.

5.2. Round Table Load-Store Logic

A naïve HLS code for the *update_table()* function in Figure 2 will certainly result in a pipeline with an II larger than 1 when compiled. The pipeline will not be able to process an input item every clock cycle mainly due to memory dependency and the read-modify-write operation performed when updating a bucket in the round table. With dual-port RAM, it is possible to perform load and store operations at the same time; however, the minimum latency of a RAM block is 1 clock cycle. This means that consecutive updates to the same bucket will create a data hazard issue. To prevent functional failure due to memory dependency, the HLS compiler inserts stalling logic in the pipeline and increases II to a number larger than the RAM block latency. Therefore, the best possible II for a naïve HLS implementation is 2. In addition, the performance will further suffer if the round table is large—consisting of many FPGA RAM primitives that are physically distanced apart on the chip. The compiler may decide to further increase II or reduce the maximum operating frequency (fmax) to meet timing requirements.

HLS tools usually support special pragmas to relax memory dependency. For example, the "ivdep safelen (m)" pragma can be used to tell the compiler that there will be no memory dependency for at least m loop iterations. We only need $m = 2$ to achieve an II of 1. However, when m is sufficiently larger than the RAM block latency, the compiler will be able to schedule the memory load and store operations further apart in the pipeline to achieve higher fmax, and this is particularly useful when the RAM block is large [37].

Relaxing memory dependencies and specifying a safe dependence distance m for the compiler does not guarantee functional correctness. The programmer needs a mechanism to guarantee that no such dependencies will occur in the first place to prevent functional

failure. There are several solutions available in the literature for handling dependencies at run-time when updating frequency estimation sketches (Examples: [19,21,37,38]). None of the available solutions are directly applicable to our proposed algorithm. Therefore, we propose a custom solution for handling memory dependencies when updating the round table according to the *update_table()* function in Figure 2. The solution is based on a Load-Store Queue (LSQ) that precedes the round table update stage in Figure 4. The LSQ mainly performs two tasks—labelled "pre-count" and "hazard detect" in Figure 4.

5.2.1. Pre-Count: Forward Weight Accumulation

Memories constructed using the logic blocks of the FPGA do not have read and write latencies as in memories constructed with the embedded RAM resources of the FPGA. The first step in our proposed solution for handling memory dependencies is to pre-count the occurrences of item keys in a small buffer constructed using the logic blocks. We refer to this technique as "forward weight accumulation". The circuit used for forward weight accumulation is shown in Figure 5. The circuit consists of a shift register of size m. All registers have parallel connections to the weight accumulation logic.

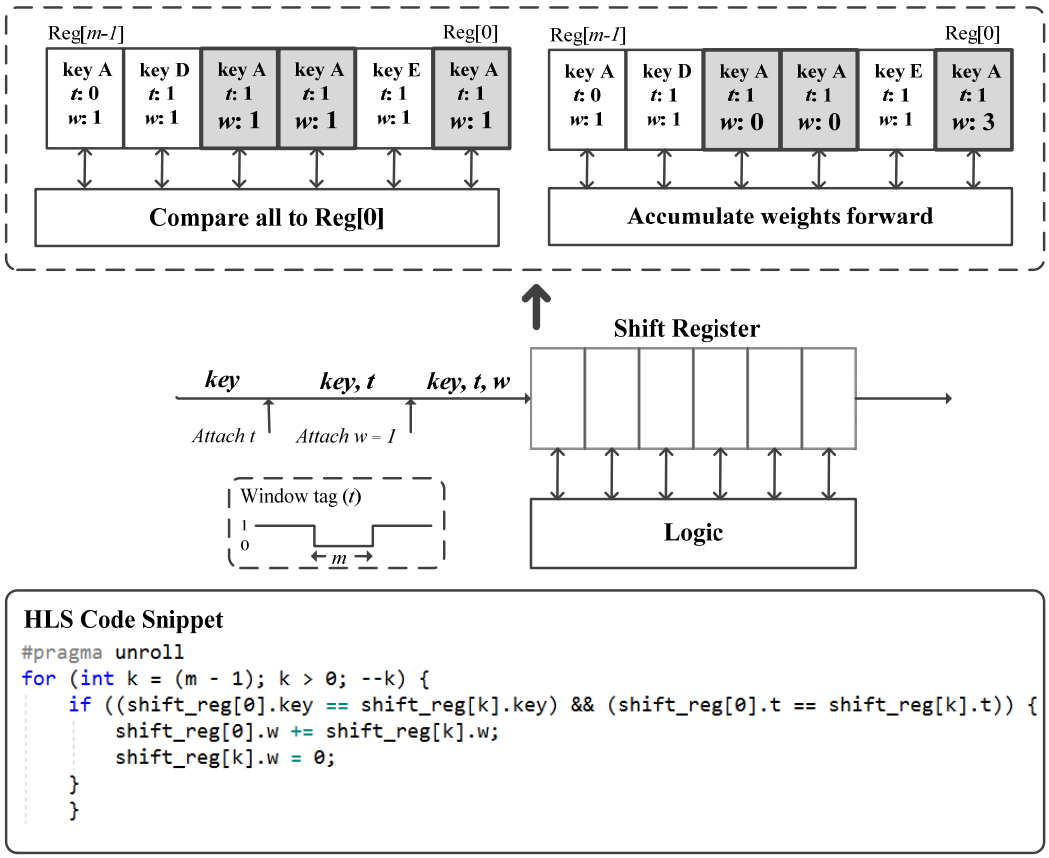


Figure 5. Pre-count: breaking memory dependency for m loop iterations using forward weight accumulation.

When the kernel is launched, the item keys are read from the input FIFO. The kernel will first wrap each item key in a data structure containing a wight variable (w), which is initialized to 1 (see Figure 5). In addition to the key and initial weight of 1, the kernel will wrap a variable t that is assigned to either 0 or 1 depending on the order of the item in the stream. The variable t is a tag used for dividing consecutive keys into groups or windows of size m; each key in the same window is assigned the same tag. Basically, the value of t is inverted in every m loop iteration. When passing items through the shift register, the weight accumulation logic will compare the key in Reg[0] to the keys in all

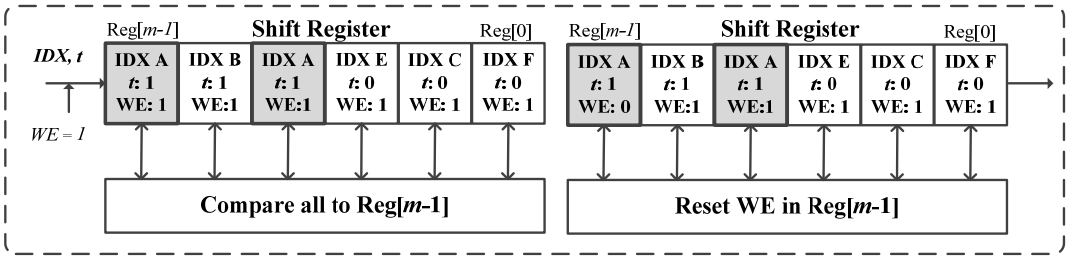
other registers. The weight of any identical key belonging to the same window will be accumulated forward in Reg[0] before it is reset to zero. Any key with a weight of zero is regarded as a "bubble" and can be discarded in later stages in the pipeline when it exits the shift register. By inserting bubbles in the pipeline, memory dependencies are eliminated for at least m loop iterations.

Implementing forward weight accumulation is straightforward with HLS as it only requires a simple loop unroll pragma (see the HLS code snippet in Figure 5). It should be noted that a previously proposed solution deployed a similar backward weight accumulation technique that does not require the stream to be divided into windows with alternating tags [37]. The idea was to keep accumulating wights backward in Reg[$m-1$] to insert as many bubbles as possible into the pipeline. While the solution is perfect for simple frequency estimation sketches, it is not suitable for the algorithm presented in this paper. This is mainly because, in skewed data streams, the weights of frequent items will likely keep accumulating in the shift register for long intervals in the case of using a backward accumulation technique. This is highly undesirable for our sampling algorithm because updates of frequent items are delayed in the shift register and may fail to stick in any of the round table buckets when they eventually exit the shift register.

5.2.2. Data Hazard Detection

With forward weight accumulation, we resolve the memory dependency issue associated with consecutive items with identical keys in the stream. There is still one minor issue that needs to be resolved before safely using the "ivdep safelen (m)" pragma in the kernel's HLS code. The issue arises from the fact that different keys may generate the same round table index (IDX). While this is not a problem when the keys are distanced apart in the stream or when a key is already registered in the indexed bucket, a data hazard occurs when two keys that are less than m cycles apart generate the same index to an empty bucket. When the empty bucket is evaluated for the first key update, a memory store operation is initiated to register the key in the bucket. Due to memory latency, the bucket may still be interpreted as empty when evaluated for the second key update—initiating an incorrect memory store operation.

To resolve this data hazard, the table indexes are first pre-processed by an LSQ similar to the one used for forward weight accumulation. The LSQ consists of a shift register of size m and some data hazard detection logic (see Figure 6). When passing the indexes to the shift register, the kernel wraps a Write Enable (WE) variable as well as the same window tag t used in forward weight accumulation. The data hazard detection logic compares the index in Reg[$m-1$] to all the indexes in the other registers. If any identical index that belongs to the same window is detected, the WE variable in Reg[$m-1$] is reset. In later stages in the pipeline, any key update with WE = 0 is discarded.



HLS Code Snippet

```
#pragma unroll
        for (int k = 0; k < (m - 1); ++k) {
            if ((shift_reg[m - 1].IDX == shift_reg[k].IDX) && (shift_reg[m-1].t == shift_reg[k].t)) {
                shift_reg[m - 1].WE = 0;
            }
        }
```

Figure 6. Resolving data hazards before the round table update stage.

5.3. Heavy Hitter Summary Load-Store Logic

The next stage in the pipeline is the heavy hitter summary update stage, which will execute the *update_summary()* function in Figure 3. This function takes the item key, summary index (IDX_S), the item weight w, and the round count c.

As memory dependencies occur due to identical keys in the stream that are already resolved by weight accumulation, there is only one minor data hazard that needs to be resolved before executing the *update_summary()* function. The data hazard occurs when two item hits with different keys are less than m cycles apart and generate the same summary index. In particular, if the indexed bucket stores a round count smaller than the round count of both items and the round count of the second key is larger than the round count of the first key. When the bucket is evaluated for the first item, a store operation will be initiated to replace the stored item with the smaller round count. Due to memory latency, an incorrect memory store operation may be also initiated for the second item. To resolve this data hazard, another LSQ is used after the round table update stage. The LSQ in Figure 7 consists of a shift register of size m and some data hazard detection logic. The summary indexes for the items are wrapped with the associated round counts as well as a Write Enable (WE_S) variable before being passed to the shift register of the LSQ. The data hazard detection logic compares the index in Reg[$m-1$] to all the indexes in the other registers. If any identical index with a larger round count is present, the WE_S signal in Reg[$m-1$] is reset. Any item with $WE_S = 0$ is discarded before updating the heavy hitter summary.

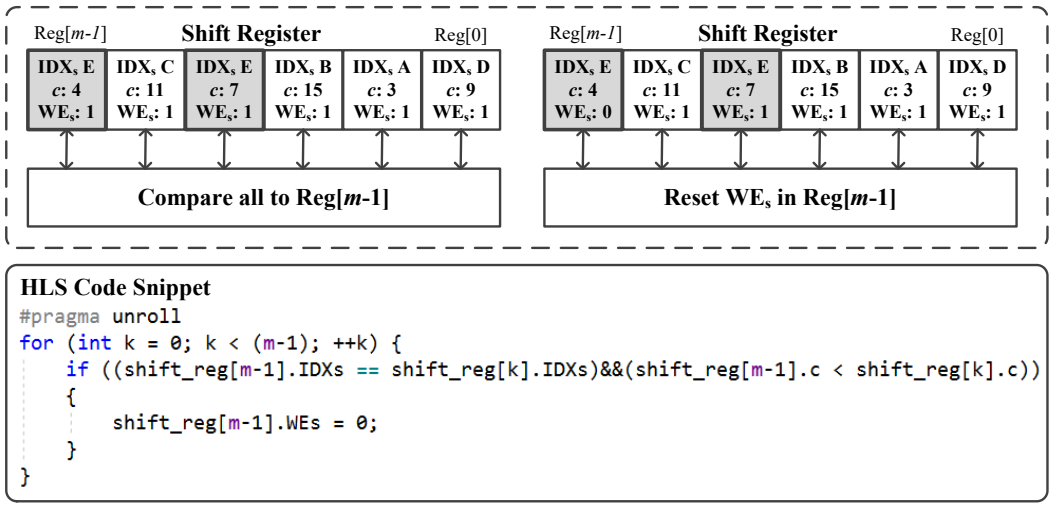


Figure 7. Resolving data hazards before the heavy hitter summary update stage.

By applying the aforementioned LSQs, a stall-free HLS kernel for the proposed algorithm can be easily implemented. The kernel can be invoked by a host to process as many rounds as needed. However, an additional minor tweak is needed to safely process multiple rounds. The shift registers of the LSQs need to be flushed at the end of a round to ensure that all item hits are processed. To accomplish this, the kernel runs for a number of extra iterations at the end of the round. During these iterations, the kernel passes dummy item keys with zero weights to flush the shift registers before the next round.

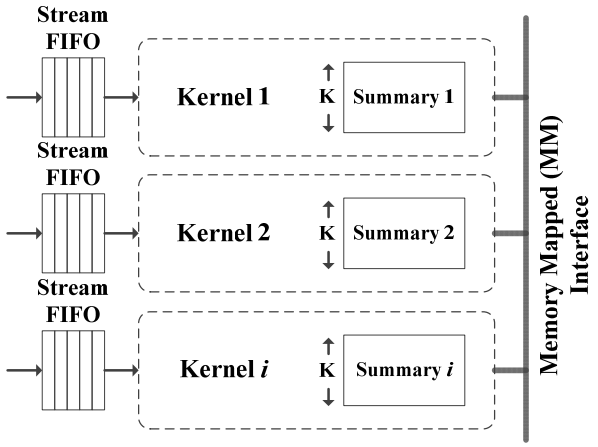
5.4. Support for Parallel Sub-Streams

Achieving high performance when mapping algorithms into hardware can be performed via deep pipelining and task parallelism. Task parallelism involves vectorizing the input into distinct sets that are processed using separate kernels. Results from these separate kernels can then be merged. By applying task parallelism, the throughput can be significantly increased without the need to operate a kernel hardware at its fmax. Task parallelism is particularly useful to exploit the high throughputs supported by the on-chip

High Bandwidth Memory (HBM) available in some of the trending high-end FPGAs [39]. In addition, FPGAs with high-speed transceivers may require several replicas of a kernel to be able to saturate the available bandwidth.

In order to apply task parallelism, the algorithm should support vectorization. In general, data stream frequency estimation sketches, such as the *count–min* sketch, can be vectorized by splitting the stream into several sub-streams that update separate sketches with identical geometry. Merging results from distributed *count–min* sketches with identical geometry is simple as it only involves the entry-wise summarization of the sketch tables. In hardware, this can be difficult, as the host needs to access every table entry in the sketch, which can span most of the embedded RAM on the FPGA chip. Based on the lack of attempts reported in the literature, there have been few attempts to implement parallel frequency estimation sketches in a single FPGA chip [19,21]. These parallel systems only support a general update–query model, where the frequency of individual items is sequentially acquired by the host. This simple model does not support the top-k item query as it does not allow the host to access the sketch memory or any priority queue paired with the sketch.

One of the important advantages of the algorithm proposed in this paper is the ease of vectorizing the algorithm in hardware. First, the data stream is naturally divided into rounds that can be processed independently by several kernel replicas (see Figure 8). Second, the host only needs to access the smaller heavy hitter summary data structure in each kernel, which is implemented as a slave memory. Third, the merging of heavy hitter summaries is very simple, especially if the same hash function is used for all the summaries. The pseudo code in Figure 8 shows how two summaries from two kernel replicas can be merged when the same hash function and the same round sizes are used in the different kernel replicas. The merge process can be performed efficiently by the host as it only has a time complexity of O(n), where ($n = K$). In addition, if the heavy hitter summary configuration is changed in the HLS code to grant both read and write access to the host, the merge process can be performed in-place without the need to copy the summaries to the host's memory.




```
// summary 2 merged into summary 1
Function merge_summaries(summary1, summary2, K):
    for i = 0 to K − 1 do
        if summary1[i].key = summary2[i].key then
            summary1[i].total ← summary1[i].total + summary2[i].total
            if summary1[i].count < summary2[i].count then
                summary1[i].count ← summary2[i].count
        else if summary1[i].count < summary2[i].count then
            summary1[i] ← summary2[i]
```

Figure 8. Scaling performance with parallelism.

6. FPGA Implementation

This section reports the FPGA implementation results when compiling the proposed kernel using Intel HLS compiler and Intel Quartus Prime synthesis tools. The midrange Arria 10 GX 1150 FPGA was selected as a target device. This is the largest chip from the Arria 10 family, with 427,200 Adaptive Logic Modules (ALMs), 1518 Digital Signal Processing (DSP) blocks, and 2713 M20K embedded RAM blocks. As the kernel mainly consumes embedded RAM for implementing the round table and the heavy hitter summary, we focus on validating the notion that the throughput of the kernel is sustained even when scaling the size of the RAM blocks to large portions of the available on-chip memory.

As there are many parameters that can be varied in the kernel's configuration, we fix some of these parameters to practical values for simpler analysis. First, the size of the heavy hitter summary K is fixed to 2^{15} in all configurations. This size should be selected according to the number of top-k items that needs to be monitored. Making K unnecessarily large may increase the time for result readout without gaining meaningful improvements in accuracy. So, if we are aiming to report somewhere near the top-1000 items, a fixed size of K = 2^{15} is a reasonable choice that should provide a good balance between accuracy and result readout time. The size of the round count variable c is fixed to 16-bit, the total accumulate count filed in the heavy hitter summary's buckets to 32-bit, the item fingerprint FP to 16-bit, and finally, the round timestamp TS to 8-bit.

As can be seen from Figure 4, there are three hash circuits that need to be implemented in the kernel. The quality of the hash function used will affect the accuracy of the system. We opt for using a simple and efficient hash function that can be implemented with the least amount of FPGA resources. We use the "binary multiplicative" hash function described and analyzed in [40] for the three hash circuits in the kernel. This hash function only requires a single multiplier when implemented in hardware. The operation of the hash function is described as follows: assume we have L-bit integers that need to be mapped to l-bit integers. Using an odd integer seed a, the hash function is defined as:

$$h_a(x) = (a.x) mod\ 2^L / 2^{L-l}$$

Table 2 reports the FPGA post place-and-route implementation results for nine different configurations of the kernel. In these configurations, the size of the round table M and the size of the item key are varied. Three sizes of M were considered: 2^{17}, 2^{18}, and 2^{19}, and three key sizes were considered: 32-bit, 64-bit, and 128-bit. In all configurations, the safe dependence distance m was fixed to 8. Additionally, a target fmax of 400 MHz and a target II of 1 were specified to the compiler when compiling all configurations.

Table 2. Implementation results on Intel Arria 10 GX 1150 FPGA.

M	Key Size (Bits)	Fmax (MHz)	Resource Utilization		
			ALM	DSP	M20K
2^{17}	32	417	1577	6	480 (18%)
	64	399	2294	18	544 (20%)
	128	379	3493	45	672 (25%)
2^{18}	32	351	1719	6	800 (30%)
	64	351	2356	18	864 (32%)
	128	321	3616	45	992 (37%)
2^{19}	32	277	2032	6	1440 (53%)
	64	261	2567	18	1504 (55%)
	128	265	3890	45	1632 (60%)

With the proposed optimizations, the compiler achieved an II of 1 in all configurations, meaning that the maximum throughput in items/s is equivalent to the reported fmax. We can see from Table 2, that the throughput of the kernel can reach up to 417 million items/s in

the smallest configuration. All configurations achieve throughputs that can be considered optimal for this mid-range FPGA family. The largest configuration, spanning 60% of the available embedded RAM on the FPGA chip, achieved a throughput of 265 million items/s. These significantly high numbers are mainly attributed to the simplicity of the synthesized logic and the proposed memory dependency handling technique that allowed for efficient pipelining.

It should be noted that the synthesis tools may report slightly different results for the same configuration when compiled multiple times. The results in Table 2 are obtained from a single compilation process for each configuration.

7. Evaluation

7.1. Accuracy Validation

As the accuracy of the proposed accelerator will be mainly bounded by the amount of on-chip memory allocated for the round table, it is important to validate that the available embedded memory in a typical FPGA chip is sufficient for achieving good accuracy when processing practical data streams. The same Arria 10 GX 1150 FPGA is used as a baseline for accuracy analysis. The amount of embedded RAM available on this chip can be considered in the mid-range of modern FPGA devices. From Table 2, the kernel configurations with the 32-bit key size are selected for accuracy analysis. These configurations are simulated using both synthetic and real datasets. After each simulation run, the heavy hitter summary is sorted to extract the top-1000 items, which are compared to exact results. Two metrics are used in the analysis, as defined below:

Accuracy: The number of correctly identified top-k items divided by k.
Avg. Count Error: The average of relative count errors calculated in all reported items.

Figure 9 reports the accuracy and average error for synthetic data streams. The datasets were generated as Zipfian distributions [41]. The size of all datasets was fixed to 10^7 items, and the Zipfian parameter (α) was varied from 1.0 to 1.6 in intervals of 0.2. The range of α was selected to cover typical values of data skew in several types of real data streams [42]. There are several factors that would influence the best choice of round size r [8]. For simplicity, r was fixed to M in all simulation runs (M—size of the round table in the kernel).

From Figure 9 we can see that all kernel configurations achieved near-perfect accuracy. The worst-case accuracy exceeded 98%, and the worst-case average count error was below 4%.

To further verify the kernel's performance, four different real datasets were also used for accuracy analysis. The properties of the datasets are summarized in Table 3. All of these datasets are easily available and widely used in the analysis of itemset mining algorithms. *Retail* contains market basket transactions data from an anonymous Belgian retail store [43]. *Kosarak* is a collection of click-stream data from a Hungarian online news portal [44]. *Chainstore* contains customer transactions from a major grocery store in California, USA [45]. Finally, *BMS2* contains click-stream data from an anonymous webstore [45]. All of these datasets were originally structured as transactional datasets consisting of several separated transactions, each containing a number of integer items. For the purpose of stream item counting, these datasets were pre-processed to merge the transactions into a serial stream of items. Figure 10 reports the accuracy and average error count results when processing these real data streams.

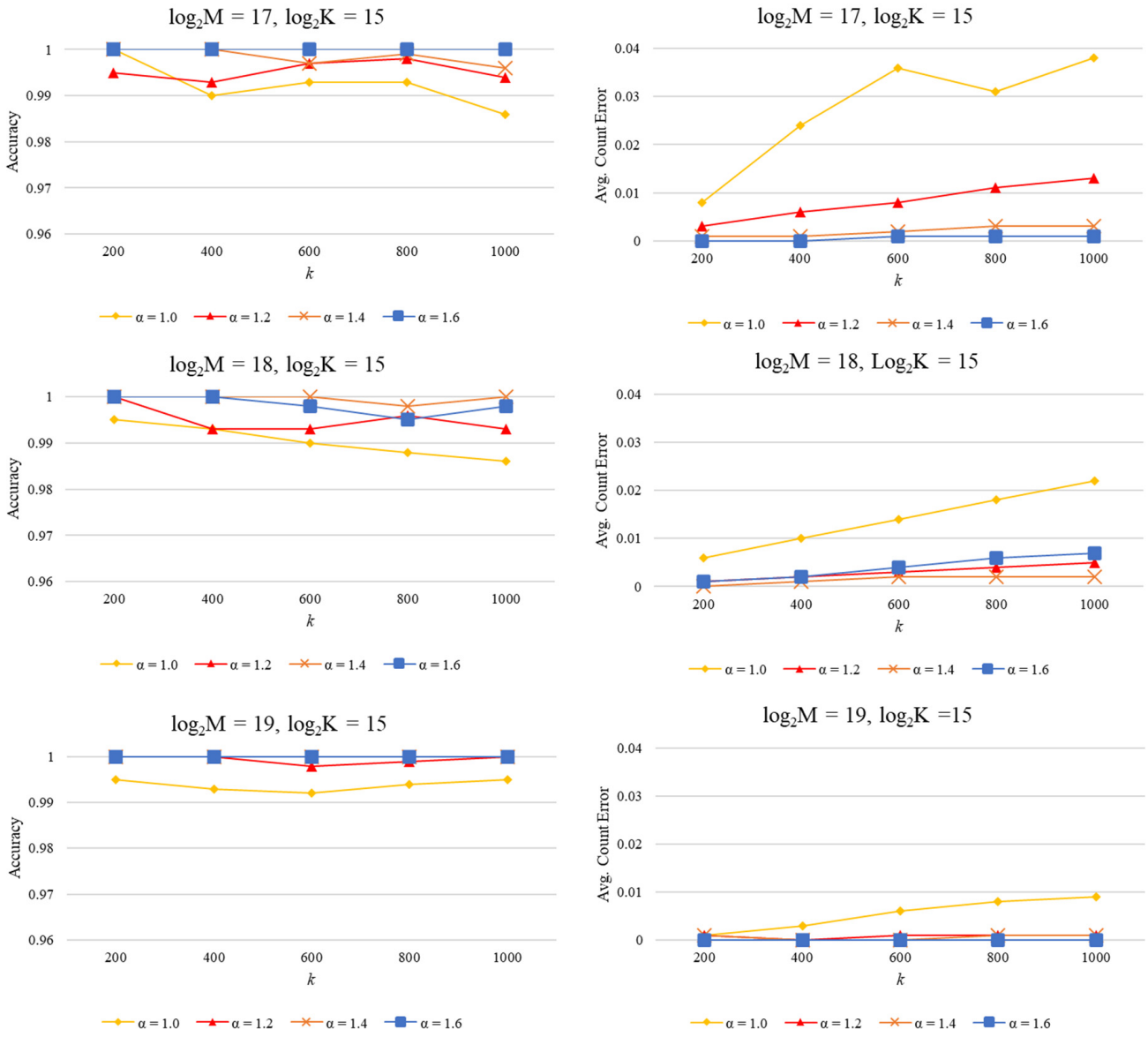


Figure 9. Results for synthetic data streams modelled as Zipfian distributions.

Table 3. Real datasets.

Dataset	Distinct Items	Size
Retail	16,469	908,399
Kosarak	41,270	8,019,015
Chainstore	46,086	8,042,879
BMS2	3340	358,278

We can see from Figure 10 that the kerel maintained a high level of accuracy when processing the real datasets. The worst-case accuracy exceeded 95%, and the worst-case average error count was 2%.

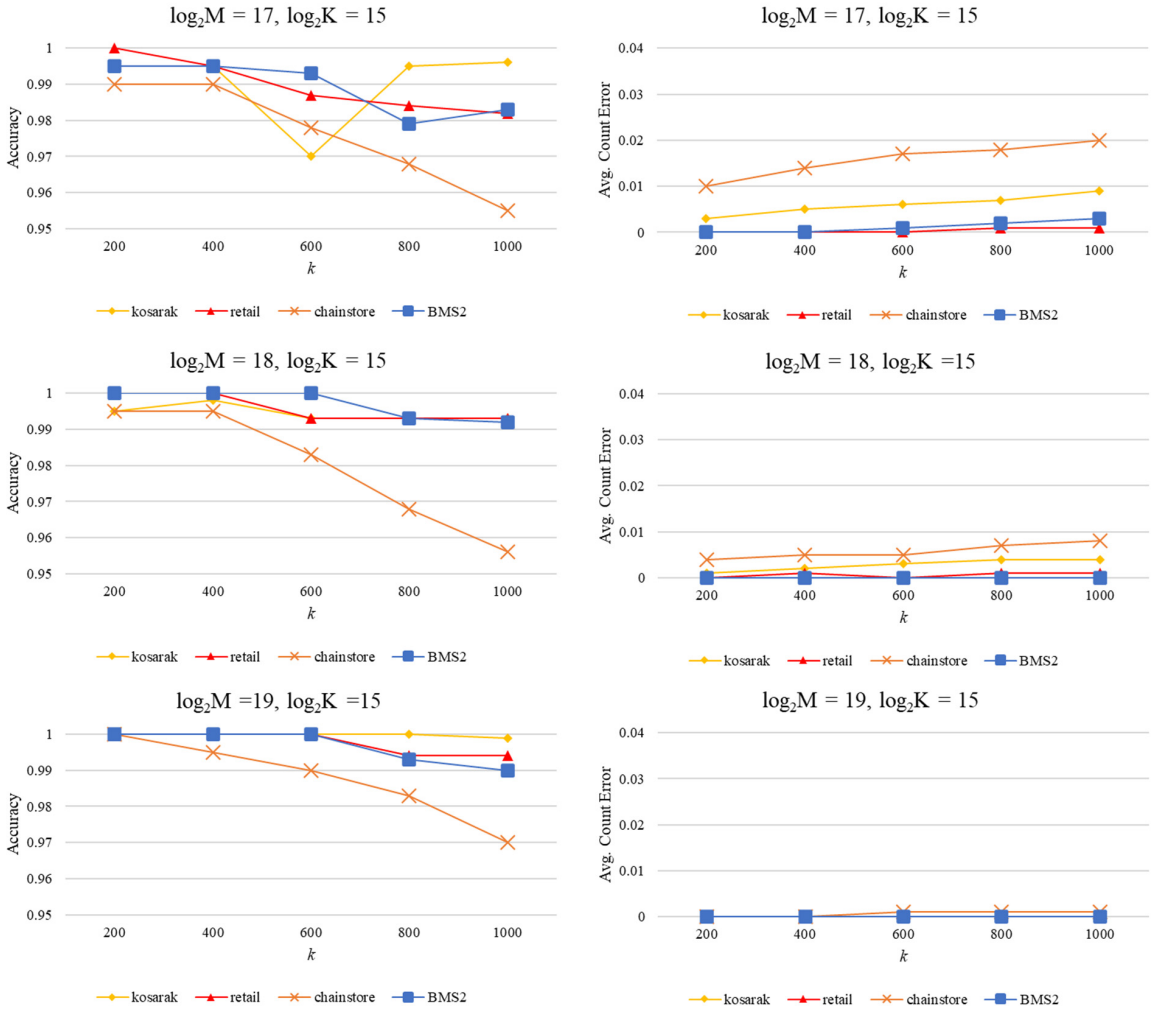


Figure 10. Results for real datasets.

7.2. Comparison with the State-of-the-Art

This section compares the proposed accelerator kernel to the fastest FPGA accelerators previously proposed in the academic literature (see Table 4). As discussed in Section 2, there are a limited number of FPGA accelerators specifically designed for the top-k query problem in data streams. Therefore, we extend the comparison to include generic data stream sketches. These sketches are simple and do not monitor any item key as they do not include a priority queue data structure. Four different metrics are used in the comparison:

Chip utilization: The highest chip utilization percentage of any resource type in the accelerator implementation (mainly logic resources for systolic array implementations and embedded memory for sketch implementations).
Monitored items: The number of item keys that are monitored by the accelerator. In Table 4, the relevant entries are labelled with "none" for generic sketch accelerators without a priority queue.
Key size: The size of the item key in bits. Smaller key sizes will limit the applications of the accelerator. For example, monitoring IPv6 addresses requires 128-bit item keys.
Throughput: The accelerator processing speed is measured in million items/s. It should be noted that, in some of the relevant publications of the accelerators (Table 4), the throughputs were reported in bits/s and calculated by multiplying the update rate by the item key size or by the network packet size in sketches targeting networking applications.

We use the largest kernel configuration from Table 1 for comparison ($M = 2^{19}$ $K = 2^{15}$). Several accelerators in Table 4 are implemented using high-end AMD/Xilinx UltraScale

and UltraScale+ FPGA devices. Since our work was based on Intel FPGA technology, the proposed accelerator was re-implemented on a Stratix 10 FPGA for better comparison with previous implementations on high-end FPGAs. When compiling for a Stratix 10 FPGA as a target, the target fmax was set to 600 MHz, and the safe dependence distance m was set to 16.

Table 4. Comparison with published work.

Implementation	Chip Utilization (%)	Monitored Items	Key Size (bits)	Throughput (M Items/s)
Arria 10 (Proposed)	60	32,768	128	265
Stratix 10 (Proposed)	6	32,768	128	542
Arria 10 [10]	51	300	32	276
Arria 10 [11]	40	1200	32	174
Virtex UltraScale+ [18]	8	none	96	415
Stratix 10 [21]	11	none	32	503
Virtex UltraScale [23]	16	none	128	456
UltraScale+ MPSoC [29]	24	2400	32	354

We can see from Table 4 that the proposed accelerator is far more efficient when compared to accelerators that support the top-k query [10,11,29], as it allows one to monitor a significantly larger number of items with larger keys. In fact, the only other accelerator to support 128-bit keys is the accelerator in [23], which is slower than our proposed accelerator and does not support the top-k item query. When implemented on a Stratix 10 FPGA, the proposed accelerator has a 25% higher throughput compared to the average throughout of competing accelerators implemented on high-end FPGAs. The proposed accelerator is also 8% faster than the fastest competing accelerator.

The presented accelerator was designed using an HLS design flow, which usually leads to performance penalties in favor of better design productivity compared to Register Transfer Level (RTL) design flows. Although most of the accelerators in Table 4 were designed and optimized using RTL, the presented accelerator outperformed all of the other accelerators. The fact that the proposed accelerator outperformed existing FPGA accelerators was mainly attributed to the simplicity of the synthesized hardware. Introducing careful modifications to the implemented algorithm facilitated the resolution of several design complexities. While, in a previous section, we presented an empirical analysis based on synthetic and real datasets to validate the accuracy of the optimized algorithm, we note that further mathematical analysis is required to formally define the error bounds and other metrics, such as the time and memory complexity of the optimized algorithm.

8. Conclusions and Future Work

This paper presented the design and implementation of an FPGA HLS accelerator kernel for computing the top-k heavy hitters in data streams. The kernel is based on a novel hardware-optimized algorithm, allowing for an easily achieved pipelined datapath with an initiation interval of 1. The proposed algorithm incorporates several optimizations, such as fingerprinting, optimistic counting, re-hashing, and timestamping to address several hardware-specific complexities that usually limit the performance of data stream item-counting accelerators. In addition, several FPGA-specific design tweaks that resolve memory dependency issues when implementing the accelerator kernel on an FPGA have been presented. These tweaks deploy unique Load-Store Queues (LSQs) that can be easily implemented using HLS code. When synthesized for Intel FPGA devices using Intel HLS compiler and targeting the Arria 10 and Stratix 10 FPGA families, the resulting synthesized hardware was very simple—mainly consuming the embedded memory resources of the FPGA. Hardware synthesis of several configurations of the kernel showed a variety of promising results: First, the high throughput of the kernel was sustained, even for configurations consuming up to 60% of the on-chip memory. Second, the high throughput and low logic footprint were also sustained when scaling the item key size processed by the kernel from 32-bit to 128-bit. Third, accuracy analysis based on synthetic and real datasets showed

that near-perfect results are achievable even using the on-chip memory capacities available in mid-density FPGA families. Finally, compared to existing state-of-the-art accelerators, the proposed accelerator is the fastest—with throughput exceeding 540 million items/s. It is also notably superior in terms of features, as it has a larger key size, larger number of monitored heavy hitters, and supports task parallelism.

Future work will first focus on further validation of the proposed algorithm as well as formally defining the error bounds, time, and memory complexity. In addition, we will explore porting the proposed kernel to a cloud application using FPGAs and different types of accelerators, such as Graphics Processing Units (GPUs).

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Cormode, G.; Hadjieleftheriou, M. Finding frequent items in data streams. *VLDB Endow.* **2008**, *1*, 1530–1541. [CrossRef]
2. Manerikar, N.; Palpanas, T. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data Knowl. Eng.* **2009**, *68*, 415–430. [CrossRef]
3. Muthukrishnan, S. *Data Streams: Algorithms and Applications*; Now Publishers Inc.: Norwell, MA, USA, 2005.
4. Harrison, R.; Cai, Q.; Gupta, A.; Rexford, J. Network-wide heavy hitter detection with commodity switches. In Proceedings of the Symposium on SDN Research, Los Angeles, CA, USA, 28–29 March 2018; pp. 1–7.
5. Liu, B. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*; Springer: Berlin/Heidelberg, Germany, 2011; Volume 1.
6. Shrivastava, N.; Buragohain, C.; Agrawal, D.; Suri, S. Medians and beyond: New aggregation techniques for sensor networks. In Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, Baltimore, MD, USA, 3–5 November 2004; pp. 239–249.
7. Cormode, G.; Yi, K. *Small Summaries for Big Data*; Cambridge University Press: Cambridge, UK, 2020.
8. Demaine, E.D.; López-Ortiz, A.; Munro, J.I. Frequency estimation of internet packet streams with limited space. In Proceedings of the European Symposium on Algorithms, Rome, Italy, 17–21 September 2002; pp. 348–360.
9. Bustio-Martínez, L.; Cumplido, R.; Letras, M.; Hernández-León, R.; Feregrino-Uribe, C.; Hernández-Palancar, J. FPGA/GPU-based acceleration for frequent itemsets mining: A comprehensive review. *ACM Comput. Surv.* **2021**, *54*, 179. [CrossRef]
10. Ebrahim, A.; Khalifat, J. Fast approximation of the top-k items in data streams using FPGAs. *IET Comput. Digit. Technol.* **2023**, *17*, 60–73. [CrossRef]
11. Ebrahim, A.; Khlaifat, J. An Efficient Hardware Architecture for Finding Frequent Items in Data Streams. In Proceedings of the IEEE International Conference on Computer Design (ICCD), Hartford, CT, USA, 18–21 October 2020; pp. 113–119.
12. Sun, Y.; Wang, Z.; Huang, S.; Wang, L.; Wang, Y.; Luo, R.; Yang, H. Accelerating frequent item counting with FPGA. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 26–28 February 2014; pp. 109–112.
13. Teubner, J.; Muller, R.; Alonso, G. Frequent item computation on a chip. *IEEE Trans. Knowl. Data Eng.* **2010**, *23*, 1169–1181. [CrossRef]
14. Metwally, A.; Agrawal, D.; El Abbadi, A. Efficient computation of frequent and top-k elements in data streams. In Proceedings of the International Conference on Database Theory, Edinburgh, UK, 5–7 January 2005; pp. 398–412.
15. Sha, M.; Guo, Z.; Wang, K.; Zeng, X. A High-Performance and Accurate FPGA-Based Flow Monitor for 100 Gbps Networks. *Electronics* **2022**, *11*, 1976. [CrossRef]
16. Pontarelli, S.; Reviriego, P.; Maestro, J.A. Parallel d-pipeline: A cuckoo hashing implementation for increased throughput. *IEEE Trans. Comput.* **2015**, *65*, 326–331. [CrossRef]
17. Cormode, G.; Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* **2005**, *55*, 58–75. [CrossRef]
18. Sateesan, A.; Vliegen, J.; Scherrer, S.; Hsiao, H.-C.; Perrig, A.; Mentens, N. Speed records in network flow measurement on FPGA. In Proceedings of the 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), Dresden, Germany, 30 August–3 September 2021; pp. 219–224.
19. Chiosa, M.; Preußer, T.B.; Alonso, G. SKT: A One-Pass Multi-Sketch Data Analytics Accelerator. *Proc. VLDB Endow.* **2021**, *14*, 2369–2382. [CrossRef]
20. Tang, M.; Wen, M.; Shen, J.; Zhao, X.; Zhang, C. Towards memory-efficient streaming processing with counter-cascading sketching on FPGA. In Proceedings of the ACM/IEEE Design Automation Conference (DAC), Virtual, 20–24 July 2020; pp. 1–6.
21. Kiefer, M.; Poulakis, I.; Breß, S.; Markl, V. Scotch: Generating fpga-accelerators for sketching at line rate. *Proc. VLDB Endow.* **2020**, *14*, 281–293. [CrossRef]

22. Saavedra, A.; Hernández, C.; Figueroa, M. Heavy-hitter detection using a hardware sketch with the countmin-cu algorithm. In Proceedings of the 2018 21st Euromicro Conference on Digital System Design (DSD), Prague, Czech Republic, 29–31 August 2018; pp. 38–45.
23. Tong, D.; Prasanna, V.K. Sketch acceleration on FPGA and its applications in network anomaly detection. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *29*, 929–942. [CrossRef]
24. Kohutka, L.; Nagy, L.; Stopjaková, V. A novel hardware-accelerated priority queue for real-time systems. In Proceedings of the 2018 21st Euromicro Conference on Digital System Design (DSD), Prague, Czech Republic, 29–31 August 2018; pp. 46–53.
25. Chen, W.; Li, W.; Yu, F. A hybrid pipelined architecture for high performance top-K sorting on FPGA. *IEEE Trans. Circuits Syst. II Express Briefs* **2019**, *67*, 1449–1453. [CrossRef]
26. Yan, D.; Wang, W.-X.; Zuo, L.; Zhang, X.-W. A novel scheme for real-time max/min-set-selection sorters on FPGA. *IEEE Trans. Circuits Syst. II Express Briefs* **2021**, *68*, 2665–2669. [CrossRef]
27. Zazo, J.F.; Lopez-Buedo, S.; Ruiz, M.; Sutter, G. A single-fpga architecture for detecting heavy hitters in 100 gbit/s ethernet links. In Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, Mexico, 4–6 December 2017; pp. 1–6.
28. Soto, J.E.; Ubisse, P.; Fernández, Y.; Hernández, C.; Figueroa, M. A high-throughput hardware accelerator for network entropy estimation using sketches. *IEEE Access* **2021**, *9*, 85823–85838. [CrossRef]
29. Soto, J.E.; Ubisse, P.; Hernández, C.; Figueroa, M. A hardware accelerator for entropy estimation using the top-k most frequent elements. In Proceedings of the Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, 26–28 August 2020; pp. 141–148.
30. Gou, X.; Zhang, Y.; Hu, Z.; He, L.; Wang, K.; Liu, X.; Yang, T.; Wang, Y.; Cui, B. A sketch framework for approximate data stream processing in sliding Windows. *IEEE Trans. Knowl. Data Eng.* **2022**, *35*, 4411–4424. [CrossRef]
31. Yang, T.; Zhang, H.; Li, J.; Gong, J.; Uhlig, S.; Chen, S.; Li, X. HeavyKeeper: An accurate algorithm for finding Top-k elephant flows. *IEEE/ACM Trans. Netw.* **2019**, *27*, 1845–1858. [CrossRef]
32. Pagh, R.; Rodler, F.F. Cuckoo hashing. *J. Algorithms* **2004**, *51*, 122–144. [CrossRef]
33. Cho, J.M.; Choi, K. An FPGA implementation of high-throughput key-value store using Bloom filter. In Proceedings of the Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test, Taiwan, China, 28–30 April 2014; pp. 1–4.
34. Chen, X.; Tan, H.; Chen, Y.; He, B.; Wong, W.-F.; Chen, D. Skew-oblivious data routing for data intensive applications on FPGAs with HLS. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 5–9 December 2021; pp. 937–942.
35. Kiefer, M.; Poulakis, I.; Zacharatou, E.T.; Markl, V. Optimistic Data Parallelism for FPGA-Accelerated Sketching. *Proc. VLDB Endow.* **2023**, *16*, 1113–1125. [CrossRef]
36. Intel®High Level Synthesis Compiler Pro Edition: User Guide. Available online: https://www.intel.com/content/www/us/en/docs/programmable/683456/21-4/pro-edition-user-guide.html (accessed on 1 March 2023).
37. Ebrahim, A. High-Level Design Optimizations for Implementing Data Stream Sketch Frequency Estimators on FPGAs. *Electronics* **2022**, *11*, 2399. [CrossRef]
38. Preußer, T.B.; Chiosa, M.; Weiss, A.; Alonso, G. Using DSP Slices as Content-Addressable Update Queues. In Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), Gothenburg, Sweden, 31 August–4 September 2020; pp. 121–126.
39. Huang, H.; Wang, Z.; Zhang, J.; He, Z.; Wu, C.; Xiao, J.; Alonso, G. Shuhai: A Tool for Benchmarking High Bandwidth Memory on FPGAs. *IEEE Trans. Comput.* **2021**, *71*, 1133–1144. [CrossRef]
40. Dietzfelbinger, M.; Hagerup, T.; Katajainen, J.; Penttonen, M. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms* **1997**, *25*, 19–51. [CrossRef]
41. Zipf, G.K. *Human Behavior and the Principle of Least Effort*; Martino Fine Books: Eastford, CT, USA, 1949.
42. Cormode, G.; Muthukrishnan, S. Summarizing and mining skewed data streams. In Proceedings of the the 2005 SIAM International Conference on Data Mining, Newport Beach, CA, USA, 21–23 April 2005; pp. 44–55.
43. Brijs, T.; Swinnen, G.; Vanhoof, K.; Wets, G. Using association rules for product assortment decisions: A case study. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, 15–18 August 1999; pp. 254–260.
44. Frequent Itemset Mining Dataset Repository, University of Helsinki. Available online: http://fimi.cs.helsinki.fi/data/ (accessed on 2 October 2021).
45. Fournier-Viger, P.; Lin, J.C.-W.; Gomariz, A.; Gueniche, T.; Soltani, A.; Deng, Z.; Lam, H.T. The SPMF open-source data mining library version 2. In Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Riva del Garda, Italy, 19–23 September 2016; pp. 36–40.