BitBlender: Scalable Bloom Filter Acceleration on FPGAs with Dynamic Scheduling

Kenneth Liu
HiAccel Lab
Simon Fraser University
Burnaby, B.C., Canada
ksl24@sfu.ca

Alec Lu
HiAccel Lab
Simon Fraser University
Burnaby, B.C., Canada
fla30@sfu.ca

Zhenman Fang
HiAccel Lab
Simon Fraser University
Burnaby, B.C., Canada
zhenman@sfu.ca

***Abstract*—The Bloom filter is one of the most widely used data structures in big data analytics to efficiently filter out vast amounts of noisy data. Unfortunately, prior Bloom filter designs only focus on single-input-stream acceleration, and can no longer match the increasing data rates offered by modern networks.**

To support large Bloom filters with low false-positive rate and high throughput, we present BitBlender, a configurable and scalable multi-input-stream Bloom filter acceleration framework in HLS. To effectively share one large bit-vector on chip among all streams, we design and implement the novel arbiter and unshuffle modules to dynamically schedule conflicting accesses to execute sequentially and non-conflicting accesses to execute in parallel. To support different user configurations of the Bloom filter, we also develop an automation flow, together with an accurate performance estimator, to automatically generate the best BitBlender design. Experimental results show that, on the AMD/Xilinx Alveo U280 FPGA, BitBlender achieves a throughput up to 2,194 MQueries/s (i.e., 8.8 GB/s) for a 96Mb bit-vector with 0.01% false-positive rate. It achieves up to 10.4x speedup over a 24-thread CPU implementation and up to 4.9x speedup over a naively-duplicated multi-stream FPGA design. BitBlender will be released soon at https://github.com/SFU-HiAccel/BitBlender.

I. Introduction

In the big data era, a vast amount of information is digitized, offering opportunities for discovering new insights. However, noisy data, stemming from sources such as sensor errors and incomplete records, poses a threat to data integrity [1]. Filtering methods are crucial for addressing such issues, improving data quality by effectively identifying and eliminating noise.

Among all types of filters, the Bloom filter [2], which uses a bit-vector for fast set-membership queries, stands out as one of the most commonly used data structures for space-efficient and high-throughput data lookup and filtering. It has been widely utilized in big data analytics, such as database querying [3], networking [4], [5], and bioinformatics [6].

Due to the significant slowdown of CPU scaling, prior studies [7]–[9] have demonstrated respectable performance improvements in accelerating Bloom filters using FPGAs. Unfortunately, they mainly focus on accelerating a single input stream Bloom filter and can no longer match the increasing incoming data rate from modern networks or the host CPU.

Naively duplicating the single-stream Bloom filter design rapidly exhausts the on-chip memory resources, especially for large Bloom filters with a low false-positive rate target. A more effective approach is to share the underlying bit-vector across all input streams. However, this strategy introduces dynamic access conflicts to the same bit-vector partition from multiple streams, which leads to conservative sequential execution between all streams in a statically-scheduled HLS design.

In this work, we propose BitBlender, the first dynamically-scheduled, configurable, and scalable multi-stream Bloom filter acceleration framework. To address the issue of access conflicts in the bit-vector (due to sharing), we introduce a stream-to-partition arbiter module, which leverages priority encoding logic [10] to dynamically schedule conflicting accesses to execute sequentially and non-conflicting accesses to execute in parallel. Accordingly, we introduce a partition-to-stream module to reorganize the out-of-order (OoO, due to arbitration) partial query results back into streams, before they are aggregated to get the final output. Moreover, to prevent the subtle deadlock caused by the OoO scheduling, we introduce a ratelimit logic to prevent one stream from running too far ahead of another. The entire design is dataflowed, where each module is fully pipelined. Finally, we also develop an automation tool, with a performance estimator to accurately capture dynamic dataflow stalls, to automatically generate the best BitBlender design for a user configuration of the Bloom filter on a given FPGA.

We evaluate BitBlender for a variety of Bloom filter configurations using randomly generated input queries. On the AMD/Xilinx Alveo U280 FPGA, BitBlender achieves a throughput up to 2,194 MQueries/s (i.e., 8.8 GB/s) for a 96Mb bit-vector at a false-positive rate of 0.01%. Compared to the 24-thread CPU implementation, BitBlender achieves up to 10.4x speedup; while compared to a naively-duplicated multi-stream FPGA design, BitBlender achieves up to 4.9x speedup.

In summary, this paper makes the following contributions:

1. An automated, configurable, and scalable multi-stream Bloom filter acceleration framework named BitBlender.
2. A dynamic arbitration scheme (arbiter, unshuffle, and deadlock prevention) realized in statically-scheduled HLS.
3. A comprehensive evaluation and analysis of BitBlender.

II. Background and Motivation

A. Basics of Bloom Filters

A Bloom filter [2] is a filtering data structure that is composed of a bit-vector (BV) of length L, along with H different hash functions. It supports two basic operations: insertions and queries. The insertion of elements into a Bloom filter works

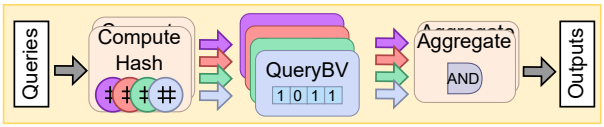

Fig. 1: Single-stream Bloom filter design, similar to [8].

as follows. Initially, the bit-vector is filled with all 0s. The element to insert is first passed into the H hash functions, yielding $i_1, i_2, ..., i_H$. Then, each of these entries in the bit-vector, $BV[i_1], BV[i_2], ..., BV[i_H]$, is set to 1.

Querying whether an element exists in a Bloom filter works similarly. First, the input query is hashed, yielding H indices. Then, it looks up these H bits, and performs an **AND** operation i.e., $BV[i_1] \wedge BV[i_2] \wedge ... \wedge BV[i_H]$, to produce the output. An output of 0 means the input is definitely not in the Bloom filter, i.e., it is safe to filter out this input data. An output of 1 means the input is probably in the Bloom filter, i.e., it assumes the input data is useful and does not filter it out, with a false-positive rate *fp*, described by Equation 1 [11]:

$$fp \approx (1 - e^{-HI/L})^H \quad (1)$$

The query false-positive rate *fp* decreases monotonically with L (bit-vector length), and increases monotonically with I (number of insertions). Its relationship with H (number of hash functions) is more complicated: it reaches a global minimum when $H = L/I \times \ln 2$ [4]. Since building a Bloom filter with insertions is usually a one-time effort, in this paper, we focus on accelerating the throughput of Bloom filter queries with a variety of L, H, I, and *fp* combinations to match the line rate.

B. Baseline Single-Stream Bloom Filter Design

Fig. 1 shows our baseline single-stream Bloom filter query accelerator on the FPGA, similar to that in [8]. The core QueryBV units perform lookups into the bit-vector, which is stored in BRAMs and URAMs on the FPGA. The bit-vector is divided into H disjoint sections, so that each hash function sends the computed index to its own bit-vector section (QueryBV unit) to avoid access conflict and achieve a fully piplined design. Note this sectioning does not affect the false-positive rate [11]. Moreover, each BRAM or URAM bank has two ports and can perform two bit-vector queries per cycle.

Here is the overall dataflow. In each cycle, an input query pair (two queries) is streamed in from an off-chip memory channel or a high-bandwidth network port. Next, each query is sent to H parallel hash functions to compute the lookup indices in the bit-vector in a fully pipelined fashion. The current hash function is the widely-used 32-bit MurmurHash3 [12], and can be easily replaced by an alternative. Finally, the lookup results from H QueryBV units are aggregated together to get the final output for each input query. This whole design is fully pipelined and can perform two bit-vector queries per cycle.

C. Challenges to Multi-Stream Bloom Filter Scaling

Unfortunately, most existing Bloom filter designs [7]–[9] only focus on single-stream design and cannot match the increasing rate (bandwidth) of incoming data from modern SmartNIC modules or off-chip memory. For example, a 100 Gbps network port on Alveo U280 FPGA board can transfer input data at a rate of 10 GB/s. Assuming a single-stream HLS (high-level synthesis) design runs at 200 MHz and each query is 32-bit long, it can only process 2 queries/cycle * 4 bytes/query * 200MHz = 1.6 GB/s, leaving about a 6x gap.

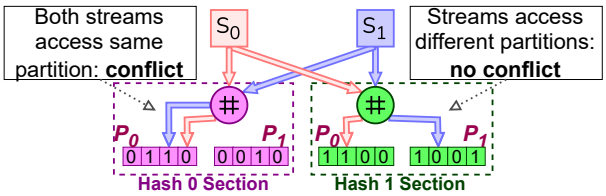

Fig. 2: Bit-vector sharing and access conflict example

Challenges for naive multi-stream duplication. A naive approach to improve the query throughput is to simply duplicate the single-stream Bloom filter design for multiple streams. However, the on-chip BRAM and URAM capacity will limit the number of streams that it can duplicate, especially when a large bit-vector length (L) is needed to keep the false-positive (fp) rate low. For example, for a Bloom filter built with 8 million insertions ($I = 8M$), to keep a $fp = 0.02\%$, it needs a bit-vector length of $L = 144M$ with $H = 9$ hash functions. The total capacity of on-chip memory (BRAMs + URAMs) on Alveo U280 FPGA board is only about 41 MB, meaning it can duplicate at most 41MB/144Mb = 2 streams.

Challenges for multi-stream bit-vector sharing. To avoid the excessive on-chip memory usage, another alternative is to let all streams share one copy of the bit-vector (with H disjoint sections) as shown in Fig. 2. Each bit-vector section is further divided into P partitions to allow parallel stream access. However, the hashed indices from multiple streams may conflict by accessing the same bit-vector partition, inside each bit-vector section. For these potential access conflicts, a statically-scheduled HLS design generated by vendor HLS tools (such as Vitis HLS) conservatively assumes the conflicts always happen. Thus, multiple streams would be scheduled to access bit-vector sections in a sequential order, leading to the same throughput as the single-stream design. In reality, access conflicts are unlikely, allowing significant speedup potential.

Goal of this paper. *Our goal is to design a high-throughput scalable multi-stream Bloom filter accelerator where multiple streams can dynamically share the same bit-vector using vendor HLS: for the majority of the time, multiple streams should run in parallel unless there are true access conflicts.*

III. BitBlender Design and Implementation

A. Overall BitBlender Design and Novelties

To achieve our goal, we present BitBlender, a configurable and scalable multi-stream Bloom filter acceleration framework. The core idea is to design and implement a novel dynamic arbitration scheme in vendor HLS to efficiently share the bit-vector between S number of input streams.

An overview of our BitBlender architecture is shown in Fig. 3 and every single module is fully pipelined with an initiation interval (II) of 1. Neighbor modules are connected using FIFOs. Each cycle, S query pairs are streamed into

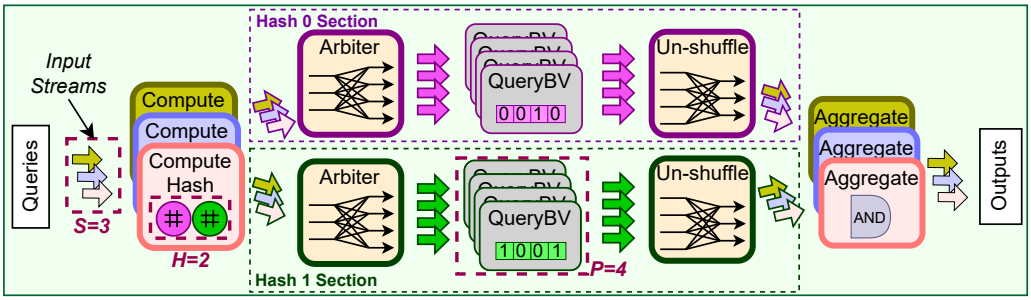

Fig. 3: Overall architecture of BitBlender. It depicts an example design with 3 input streams (S=3), 2 hash functions and 2 bit-vector sections (H=2), and 4 partitions per bit-vector section (P=4). Modules with bolded outlines have two duplicates.

2*S ComputeHash modules; modules with bolded outlines have two duplicates, for the reason explained in Section II-B. Each ComputeHash module computes H hashed indices for an input query and sends these indices to query H disjoint bit-vector sections. Each bit-vector section is further divided into P partitions to enable parallel queries from multiple streams.

To address the access conflict to the same bit-vector partition caused by multiple streams, we introduce a stream-to-partition arbiter module (pair) inside each section, which uses priority encoding logic [10] to schedule conflicting accesses to execute sequentially. Note non-conflicting accesses to different bit-vector partitions execute in parallel.

Meanwhile, this dynamic arbitration creates two new challenges for the aggregate module (pair) that aggregates the query results from all hash sections for each stream. First, the outputs from each hash section are now organized in bit-vector partitions, instead of streams. To address this issue, we introduce a partition-to-stream unshuffle module (pair) inside each section to reorganize the query results in streams. Second, inside each bit-vector partition, the outputs for multiple streams are now out-of-order (OoO), e.g., stream 1 may be processed a few items ahead of stream 0 in a specific partition, depending on which partitions the hashed indices map to. Even worse, in hash section 0, stream 1 may go far ahead of stream 0; while in hash section 1, stream 0 may go far ahead of stream 1. This could lead to a subtle deadlock between two aggregate modules for stream 0 and stream 1, which will be detailed in Section III-B2. To prevent the deadlock, we introduce a ratelimit logic inside the arbiter module such that it can control the maximum distance (D) one stream can go ahead of another; meanwhile, in the unshuffle module, to unblock its input FIFO, we buffer up to D outputs for each stream.

Finally, we also develop a design automation tool, with an accurate performance estimator to capture dynamic dataflow stalls, to automatically generate the best BitBlender design on a given FPGA under the user configuration of the Bloom filter.

B. Stream-to-Partition Arbiter Design

1) Basic Arbiter Design: The stream-to-partition arbiter aims to dynamically schedule conflicting accesses to the same bit-vector partition to execute sequentially, while scheduling non-conflicting accesses to execute in parallel. Alg. 1 shows the pseudo code for its fully pipelined design (line 3, II=1).

Alg. 1 Pseudo HLS code for stream-to-partition arbiter

```
1:  function ARBITER(in_idx[S], out_idx[P]], D)
2:      idx_buf[S] //buffer indices in registers for explicit control logic
3:      while true do //pipelined, II=1
4:          for s in 0 to S do //read from each stream, unrolled
5:              //backpressure till idx_buf[s] is consumed
6:              if (!idx_buf[s].valid && !in_idx[s].empty()) then
7:                  idx_buf[s].value = in_idx[s].read()
8:                  idx_buf[s].valid = true
9:          for p in 0 to P do //write to each partition, unrolled
10:             //search all streams for BV-idx mapping to partition p,
    priority given to the stream with smaller stream-ID when conflict
11:             for s in S to 0 do //unrolled
12:                 if (idx_buf[s].value in idx_range(p) &&
13:                 ratelimit_ok(s, D) && idx_buf[s].valid)  then
14:                     s_out = s //choose a stream-idx to write out
15:             //write the chosen stream-idx to partition p
16:             if out_idx[p].write_nb(idx_buf[s_out].value) then
17:                 idx_buf[s_out].valid = false
```

Each cycle, it first reads up to S valid hashed indices and buffers them in the *idx_buf* registers (lines 4-8). Next, for each partition p (line 9), it implements a priority encoder [10] (lines 10-14) to concurrently search all streams to find the right one accessing this partition. When there are access conflicts, it gives priority to the stream with the smallest stream ID; in our optimized implementation, it gives priority to the slowest stream by manipulating the stream loop order (line 11). Finally, it writes the chosen stream index (buffered in *idx_buf* registers) to the output index FIFO of partition p and releases the corresponding *idx_buf* buffer entry (lines 15-17).

The actual throughput of the arbiter, i.e., how many queries it dynamically processes per cycle, depends on how often the design backpressures. Backpressure occurs when the *idx_buf[s]* buffer entry is not released yet and thus the arbiter is blocked from reading input indices from stream s (line 6). Since corresponding *idx_buf* buffer entries are released when they are written out (lines 16-17), backpressure only occurs when the arbiter does not output data from a given stream. This could happen for three reasons. First, during multi-stream access conflicts, non-chosen streams will not be written out in the current cycle and have to wait for the next cycles (lines 10-14). Second, an output index FIFO is full due to backpressure caused by downstream modules (line 16). Third, the ratelimit logic (line 13) decides to pause a stream that runs too far ahead; the ratelimit logic is designed to prevent deadlocks,

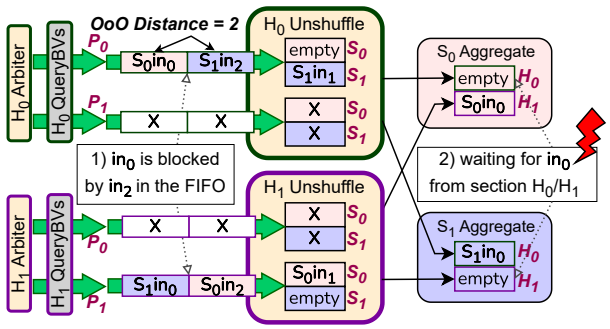

Fig. 4: Deadlock example in a multi-stream design

which will be explained in Section III-B2.

2) Deadlock Prevention in Arbiter: The dynamic arbitration leads to out-of-order (OoO) scheduling between multiple streams, which could cause potential deadlocks in the downstream aggregate modules. Note within each stream, the scheduling is still in order. An example is shown in Fig. 4.

In hash section H_0, after the arbiter and QueryBV modules, stream 1 is running 2 steps ahead of stream 0: S_1in_0 is already in the S_1 aggregate stage, S_1in_1 is in the unshuffle stage, S_1in_2 and S_0in_0 are waiting in the FIFO of bit-vector partition H_0-P_0. Likewise, for hash section H_1, stream 0 is running 2 steps ahead of stream 1: S_0in_0 is already in the S_0 aggregate stage, S_0in_1 is in the unshuffle stage, S_0in_2 and S_1in_0 are waiting in the FIFO of bit-vector partition H_1-P_1.

Now stream 0 aggregate is waiting for S_0in_0 from section H_0 and stream 1 aggregate is waiting for S_1in_0 from section H_1 to finish their aggregation between all hash sections. On the other hand, S_0in_0 is waiting and blocked by S_1in_2 in the H_0-P_0 FIFO; S_1in_2 is blocked by S_1in_1 in H_0 unshuffle; and S_1in_1 is blocked by S_1in_0 in S_1 aggregate. Likewise, S_1in_0 is also chain blocked by S_0in_0 in S_0 aggregate. This forms a circular dependence, which leads to a deadlock.

To prevent the deadlock, we introduce a ratelimit logic into the arbiter to prevent one stream from going too far ahead of another. To do this, each cycle, it monitors all the priority streams that have been scheduled to the partitions, and gets the location that the slowest stream has advanced into. Each stream then compares its own location to the slowest one to get an OoO distance, and pauses itself if the distance exceeds a threshold D. In the example in Fig. 4, the arbiter allowed D=2; instead, D=1 could prevent the deadlock as there would be no S_1in_2 blocking S_0in_0 in the H_0-P_0 FIFO.

C. Partition-to-Stream Unshuffle Design

1) Basic Unshuffle Design: The partition-to-stream unshuffle aims to reorganize the OoO query results in partitions back to in-order query results in streams to prepare for the final aggregation. Fig. 5 shows its fully pipelined design and we omit its pseudo code due to space constraints.

Each cycle, it first reads up to P QueryBV values and buffers them in the $value\_buf$ registers if there are entries available. This buffer is of size $P \times S \times D$, to hold up to D values for each stream from each partition. The D dimension is introduced to address the deadlock issue, which will be explained further

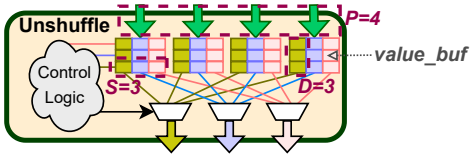

Fig. 5: Unshuffle module design

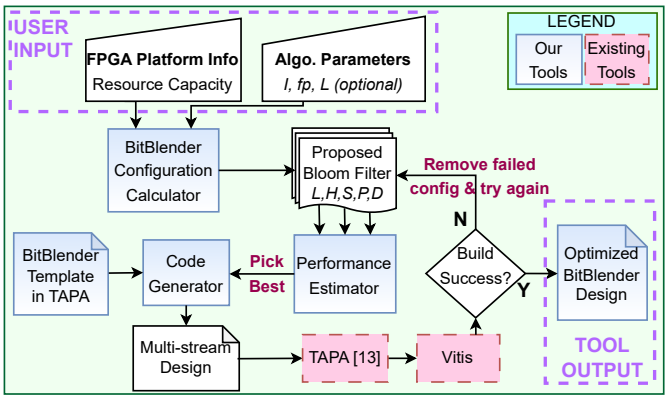

Fig. 6: Flowchart of design automation for BitBlender

in Section III-C2. Next, for each stream s, it concurrently searches all partitions in the buffer for the corresponding value to send. Note only one partition could hold the value, so there can be no access conflict. Therefore, as opposed to the arbiter module, there is no prioritization logic required. Finally, it writes the chosen bit-vector value to the output FIFO of stream s, and releases the corresponding $value\_buf$ buffer entry.

As before, the actual throughput of the unshuffle module depends on how often the design backpressures. Backpressure occurs when a $value\_buf$ buffer entry is not released yet and the incoming value is mapped to buffer in the same entry. Like the arbiter, $value\_buf$ entries are released when they are written out, so backpressure only occurs when unshuffle cannot output data. This only happens when an output FIFO is full, due to backpressure from downstream aggregate modules.

2) Deadlock Prevention in Unshuffle: The deadlock prevention logic in the unshuffle module goes hand-in-hand with the ratelimit logic in the arbiter. To accomodate the ratelimit distance D in the arbiter, the unshuffle module has a buffer for each input stream to hold up to D values, as described in Section III-C1, to unblock its input FIFO from potential blocking caused by faster streams. In the example in Fig. 4, if the unshuffle module had a buffer of size D=2 for each stream rather than D=1, it would solve the illustrated deadlock, because the blocking S_0in_2 and S_1in_2 could be consumed.

D. Automation Support for BitBlender

To allow users to easily generate optimized BitBlender hardware designs based on their own configurations, we develop a design automation toolflow shown in Fig. 6.

It takes the FPGA resources and Bloom filter parameters as inputs, including the desired false-positive rate (fp), the number of expected inserts (I) to the Bloom filter, and optionally the bit-vector length (L). Our configuration calculator then proposes multiple valid Bloom filter configuration combinations,

including the bit-vector length (*L*), number of hash functions (*H*), number of partitions per bit-vector section (*P*), number of streams (*S*), and ratelimit distance (*D*).

Next, our performance estimator selects the best-performing Bloom filter design configuration. Together with our pre-built BitBlender HLS design templates developed in TAPA tasks [13], our code generator generates the optimized multi-stream design. Finally, TAPA [13] performs automatic floorplanning optimization for the generated design and calls Vitis to generate the FPGA bitstream. If it fails to generate the bitstream, our tool selects the next best-performing design.

Performance estimator. As explained earlier, every single module in BitBlender is fully pipelined with II=1 and the entire design runs in a dynamic dataflow fashion. Ideally, it could achieve a throughput of $2 \times S$ per cycle. Due to dynamic dataflow stalls, its actual throughput is calculated as:

$$queries\_per\_cycle = 2 \times S/(1 + stall\_rate) \quad (2)$$

The dynamic dataflow stalls are caused by backpressure in the design. As explained in Section III-B1 and III-C1, the root cause of backpressure is in the arbiter module, which causes OoO scheduling and rate mismatch between multiple streams. All other modules merely propagate stalls that are initiated by the arbiter. Therefore, we estimate the dynamic dataflow $stall\_rate$ by generating a few thousand randomized inputs to emulate the average stalls per cycle in the arbiter, under a different combination of its design parameters: *S*, *P*, and *D*.

To evaluate the accuracy of our performance estimator, we compare its predictions against the measured queries per cycle of the BitBlender designs, across all configurations presented in Section IV-C. Our results show an average relative error of 2.00%, and a maximum relative error of 5.56%.

IV. Results and Analysis

A. Experimental Setup

Baseline CPU implementation. We develop a well-optimized multi-threaded CPU implementation in C++, which provides more configurability and outperforms alternative open-source CPU implementations, including bloomd and rbloom [14], [15], by more than 1.5x. This is because our implementation is optimized for integers (not strings), and the alternatives did not scale to use multiple threads properly. The hash function currently used is the widely-used and high-performance 32-bit MurmurHash3 [12], and can be easily replaced by an alternative. All the input query data are 32-bit wide and randomly generated. We compile the program using g++ with the *-Ofast* and *-march=native* optimization flags and measure its performance on a 14nm 12-core Xeon Silver 4214 CPU, with 24 hyper-threads and 16.5MB L3 cache.

Hardware platform and software tools. We evaluate our BitBlender accelerator designs on the 16nm AMD-Xilinx Alveo U280 FPGA [16] board, which has a 100 Gbps network interface. In addition, we also compare to the naive multi-stream FPGA designs by duplicating multiple copies of the single-stream design described in Section II-B. We use the automated floorplanning optimization tool TAPA [13] to improve timing closure, and Vitis v2021.2 [17] for hardware synthesis. All FPGA results are measured on board and the resource utilization is from post place-and-route reports.

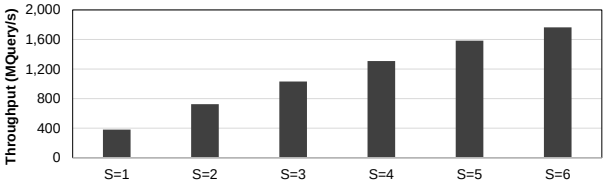

Fig. 7: BitBlender throughput at different *S*

In all cases, the throughput is calculated by the total number of queries, divided by the time taken in our CPU or FPGA kernel to process the queries.

B. Overall Performance Speedup of BitBlender

Table I shows the overall performance, measured in million-queries-per-second, for auto-generated BitBlender designs under various user Bloom filter configurations. Compared to the 24-thread CPU baseline, BitBlender achieves an average speedup of 9.15x and a peak speedup of 10.4x. Compared to the naive multi-stream FPGA design, BitBlender achieves an average speedup of 3.6x and a peak speedup of 4.9x.

Table I also lists the generated Bloom filter configuration parameters (*L*, *H*, *P*, *S*, and *D*) by our automation tool, as well as resource utilization and frequency for the FPGA designs.

C. Analysis for Different Bloom Filter Configurations

Here we analyze the BitBlender throughput when we vary one of its design parameters in *H* (number of hash functions), *S* (number of streams), *P* (number of partitions), and *D* (rate-limit distance). Unless otherwise stated, by default, we set *H*=9, *S*=6, *P*=8, *D*=16, and the bit-vector section length (per hash function) to 16 million, i.e., the biggest design in Table I.

#Streams impact. Shown in Fig. 7, the throughput increases roughly linearly when *S* increases. Table II also lists the corresponding resource utilization and frequency when *S* changes. For multi-stream designs, each additional stream adds about 8%-9% more LUTs, 6% more FFs, and 3% more DSPs. Since the single-stream design does not require our dynamic arbitration solution, the resource increase from *S*=1 to *S*=2 is bigger than the other *S* increments. Note the BRAM and URAM utilization does not change, as the bit-vector is shared among all streams. When *S* increases, the frequency gradually decreases; when *S*=7, the timing closure fails as it consumes too many resources and the router is unable to find a valid solution. The frequency is primarily constrained by logic delays in the arbiter and unshuffle modules, which grows with *S* (and *P*). The maximum number of streams realizable in BitBlender is constrained by the resource usage of the design configuration and the timing closure.

#Hashes impact. Shown in Fig. 8, the throughput decreases slightly when *H* increases. This is mainly caused by the frequency drop due to more complicated floorplanning for larger designs: with regards to queries per cycle, the differences are negligible.

TABLE I: BitBlender auto-generated design performance and resource comparison. I = # inserted elements, fp = false-positive rate, L = bit-vector length, H = # hash functions, S = # streams, P = # partitions per section, and D = ratelimit distance.

User Config		Design	Throughput	Generated Config					Resource Usage					Frequency
I	fp		(MQueries/s)	L	H	S	P	D	LUTs	FFs	BRAM	URAM	DSP	(MHz)
4M	0.01%	BitBlender	**2,194 (8.0x)**	96M	6	8	8	16	56.29%	36.06%	44.94%	30.00%	16.00%	158
		Naive multi-stream	812 (3.0x)	80M×2	10	2	-	-	11.85%	6.95%	67.71%	50.00%	3.24%	207
		24-thread CPU	274 (1x)	96M	6	-	-	-	-	-	-	-	-	-
4M	0.001%	BitBlender	**1,872 (9.7x)**	128M	8	6	8	16	51.92%	33.12%	56.55%	40.00%	16.00%	152
		Naive multi-stream	831 (4.3x)	96M×2	12	2	-	-	12.37%	7.05%	79.12%	60.00%	3.77%	212
		24-thread CPU	193 (1x)	128M	8	-	-	-	-	-	-	-	-	-
8M	0.2%	BitBlender	**1,870 (8.5x)**	112M	7	6	8	16	46.53%	29.74%	50.69%	35.00%	14.01%	152
		Naive multi-stream	399 (1.8x)	112M×1	7	1	-	-	9.94%	6.23%	50.32%	35.00%	1.24%	199
		24-thread CPU	220 (1x)	112M	7	-	-	-	-	-	-	-	-	-
8M	0.02%	BitBlender	**1,764 (10.4x)**	144M	9	6	8	16	57.14%	36.51%	62.00%	45.00%	18.00%	146
		Naive multi-stream	360 (2.1x)	144M×1	9	1	-	-	10.43%	6.31%	61.83%	45.00%	1.51%	191
		24-thread CPU	170 (1x)	144M	9	-	-	-	-	-	-	-	-	-

TABLE II: BitBlender resource utilization. I=8M, fp=0.02%.

S	LUTs	FFs	BRAM	URAM	DSP	Frequency
1	10.43%	6.31%	62.00%	45.00%	1.5%	191 MHz
2	22.78%	14.21%			6.03%	183 MHz
3	30.80%	18.92%			9.02%	174 MHz
4	38.68%	24.17%			12.01%	160 MHz
5	48.37%	30.04%			15.00%	158 MHz
6	57.14%	36.51%			18.00%	146 MHz
7	68.35%	43.38%			20.99%	(did not route)

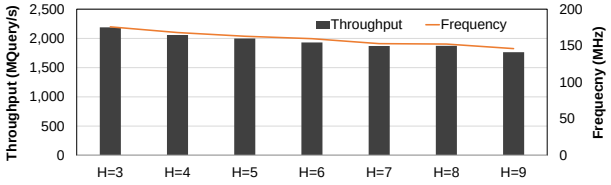

Fig. 8: BitBlender throughput at different H

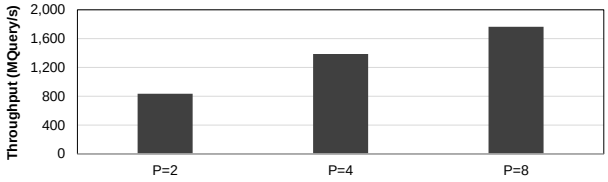

Fig. 9: BitBlender throughput at different P

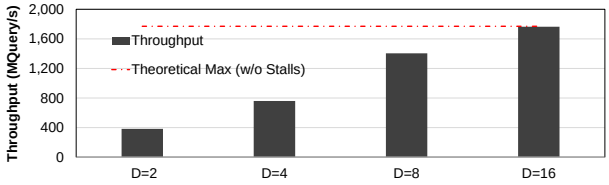

Fig. 10: BitBlender throughput at different D

#Partitions impact. Shown in Fig. 9, the throughput increases when P increases, since BitBlender's parallelism is constrained by $min(S, P)$ and S=6 in this case. Based on this calculation, comparing P=4 to P=8, one might expect a performance gain of 1.5x. However, the gain is closer to 1.3x, because the P=8 design achieves a lower frequency than P=4.

Ratelimit distance impact. Shown in Fig. 10, the throughput increases when D increases. A smaller D leads to less opportunities for OoO scheduling in the arbiter, and thus a higher percentage of dataflow stalls. On the other hand, a larger D leads to a larger buffer size and more resource utilization in the unshuffle module. Therefore, we choose D=16, which only causes about 0.2% of cycles to be stalls in the dynamic dataflow.

D. Comparison with Previous FPGA Accelerator Designs

Table III compares BitBlender against prior FPGA designs: [8], an early work accelerating single-stream counting Bloom filters; [7], state-of-the-art SRAM-based Bloom filter accelerator on FPGA, employing high-frequency hash functions in RTL design; and [9], state-of-the-art DRAM-based Bloom filter accelerator, using radix sorters to order off-chip memory accesses. We note that the FPGAs used in these studies are smaller than our AMD/Xilinx Alveo U280 FPGA: Sateesan et al. target a Virtex Ultrascale+ in [7]; Kang et al. target a Virtex-7 in [9]; and Harwayne-Gidansky et al. target a Virtex-4 in [8]. None of these works are open-sourced for us to reproduce results on a larger FPGA. However, note that all of these prior studies target a single-stream design, and therefore, their throughput will be capped at two queries per cycle, even in a larger FPGA with more resources.

TABLE III: Performance comparison with prior work

Work	H	Throughput (MQueries/s)	Frequency (MHz)	L	I @ fp = 0.1%
BitBlender (ours, SRAM)	8	1,890	158	128M	9.2M
	9	1,763	146	144M	10.3M
	12	1,515	150	96M	6.9M
[7] (SRAM)	12	925	462	0.25M	18k
[8] (SRAM)	8	410	205	16k	1.1k
[9] (DRAM)	8	47	250	512M	36.8M

Compared to previous SRAM-based FPGA studies [7], [8], BitBlender supports orders-of-magnitude larger bit-vectors and maximum number of inserts under the same false-positive rate. At the same time, BitBlender achieves about 2.0x and 4.6x higher throughput than [7] and [8]. Compared against the DRAM-based FPGA implementation [9], BitBlender achieves about 40.2x higher throughput while supporting a similar scale of bit-vector size, which is only 4x smaller.

V. Related Work

Bloom Filter acceleration on FPGA. Besides Bloom filter acceleration studies presented in Section IV-D, Khairy et al. proposed the first HLS-based Bloom filter accelerator and achieved a throughput of 0.77 queries/cycle [18]. Other studies utilized Bloom filter as part of a larger accelerator on FPGA, such as flow-table lookups for software-defined networking [19], equi-join operations in databases [20], and DNA sequence comparisons [21]. However, they only support

single-stream acceleration. BitBlender is the first work to support scalable multi-stream acceleration for Bloom filters.

Dynamic scheduling in vendor HLS. In [22], Du et al. proposed a pipelined shuffle module in their SpMV accelerator to dynamically arbitrate and resolve access conflicts to the dense vector buffered on-chip. This is similar to our arbiter module. However, our Bloom filter design is more complicated, as it requires downstream unshuffle modules to reorganize the OoO data in partitions back into in-order streams and requires deadlock prevention logic to avoid potential deadlocks between multiple streams; neither of these happens in [22].

Dynamic HLS compiler. Another orthogonal line of work is the recent dynamic HLS compiler toolflow. For example, in [23], Josipović et al. introduced Dynamatic, a dynamically-scheduled HLS compiler, which synthesizes a datapath consisting of elastic components [24] that can tolerate dynamic data-rates. Their results showcased a greatly improved cycle-performance for dynamic workloads, at the cost of increased resources and degraded clock frequency. In this work, we demonstrate a dynamic arbitration scheme with our proposed arbiter and unshuffle modules, together with the deadlock prevention logic, in statically-scheduled vendor HLS.

VI. Conclusion

In this paper, we introduce BitBlender, a configurable and scalable multi-stream Bloom filter acceleration framework in HLS. To address access conflicts in the on-chip bit-vector, caused by sharing it among multiple streams, we have developed a novel dynamic arbitration scheme in statically-scheduled vendor HLS, which includes arbiter and unshuffle modules, together with deadlock prevention logic. Based on the user-defined Bloom filter configuration and FPGA specification, we have developed an automation tool to generate the best-performing Bloom filter accelerator on FPGA. Experiments on the Alveo U280 FPGA show that BitBlender achieves a throughput up to 2,194 MQueries/s (i.e., 8.8 GB/s) for a 96Mb bit-vector with 0.01% false-positive rate, which is close to the line rate of its 100 Gbps (i.e., 10 GB/s) network interface. On average, BitBlender achieves 9.15x speedup over an optimized 24-thread CPU implementation and 3.6x speedup over a naively-duplicated multi-stream FPGA design.

Acknowledgements

This work was supported in part by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; CFI John R. Evans Leaders Fund and BC Knowledge Development Fund; Huawei Canada, AMD-Xilinx; and the Paderborn Center for Parallel Computing, Germany (for equipment access).

References

[1] D. García-Gil, J. Luengo, S. García, and F. Herrera, "Enabling smart data: Noise filtering in big data classification," *Information Sciences*, vol. 479, pp. 135–152, 2019.

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, p. 422–426, jul 1970.

[3] AMD/Xilinx, "Vitis database library," 2024, last accessed March 16, 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-database.html

[4] A. Broder and M. Mitzenmacher, "Survey: Network applications of Bloom filters: A survey." *Internet Mathematics*, vol. 1, 11 2003.

[5] B. Maggs and R. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 52–66, 07 2015.

[6] H. Stranneheim, M. Käller, T. Allander, B. Andersson, L. Arvestad, and J. Lundeberg, "Classification of DNA sequences using Bloom filters," *Bioinformatics*, vol. 26, no. 13, pp. 1595–1600, 05 2010.

[7] A. Sateesan, J. Vliegen, J. Daemen, and N. Mentens, "Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications," *Microprocessors and Microsystems*, vol. 93, p. 104619, 2022.

[8] J. Harwayne-Gidansky, D. Stefan, and I. Dalal, "FPGA-based SoC for real-time network intrusion detection using counting Bloom filters," in *IEEE Southeastcon*, 2009, pp. 452–458.

[9] S. Kang, T. S. Ganesh Nerella, S. Uppoor, and S.-W. Jun, "Bunch-bloomer: Cost-effective bloom filter accelerator for genomics applications," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 9–16.

[10] M. M. Mano and M. D. Ciletti, *Digital Design (4th Edition)*. USA: Prentice-Hall, Inc., 2006.

[11] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable Bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[12] A. Appleby, "smhasher," 2016. [Online]. Available: https://github.com/aappleby/smhasher

[13] L. Guo, Y. Chi, J. Lau, L. Song, X. Tian, M. Khatti, W. Qiao, J. Wang, E. Ustun, Z. Fang, Z. Zhang, and J. Cong, "TAPA: A scalable task-parallel dataflow programming framework for modern FPGAs with co-optimization of HLS and physical design," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, dec 2023.

[14] A. Dadgar, "bloomd," 2022. [Online]. Available: https://github.com/armon/bloomd

[15] K. Hanke, "rbloom," 2023. [Online]. Available: https://github.com/KenanHanke/rbloom

[16] AMD/Xilinx, "Alveo U280 data center accelerator card data sheet (ds963)," 2024. [Online]. Available: https://docs.xilinx.com/r/en-US/ds963-u280/Summary

[17] ——, "Vitis unified software platform," 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development

[18] R. Khairy, M. Safar, and M. W. El-Kharashi, "Bloom filter acceleration: A high level synthesis approach," in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2017, pp. 1–6.

[19] E. Kaljic, A. Maric, and P. Njemcevic, "Bloom filter based acceleration scheme for flow table lookup in SDN switches," in *2022 XXVIII International Conference on Information, Communication and Automation Technologies (ICAT)*, 2022, pp. 1–6.

[20] B. He, M. Xue, S. Liu, and W. Luo, "Bloom filter-based parallel architecture for accelerating equi-join operation on FPGA," *Electronics*, vol. 10, no. 15, 2021.

[21] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain, "Mercury BLASTP: Accelerating protein sequence alignment," in *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 2. New York, NY, USA: Association for Computing Machinery, jun 2008.

[22] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped FPGAs using HLS: A case study on SpMV," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 54–64.

[23] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 127–136.

[24] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 657–662.