BitBlender: Scalable, High-Throughput Bloom Filter Acceleration on FPGAs with Dynamic Scheduling

KENNETH LIU, Simon Fraser University, Canada
ZHENMAN FANG, Simon Fraser University, Canada and University of Minnesota, USA

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. Bloom filters are used to accelerate many applications, including databases, networking, security, and bioinformatics. Unfortunately, prior Bloom filter designs only focus on single-input-stream acceleration, and cannot match the increasing data rates offered by modern networks. The underlying problem is reading from the bit-vector, which requires fast random access. Since accesses are random, sharing the bit-vector is difficult, due to dynamic access conflicts. However, naïvely duplicating the single-stream accelerator sacrifices scalability, due to the resource overhead of storing the entire bit-vector multiple times.

We present BitBlender, the first scalable, high-throughput multi-input-stream Bloom filter acceleration framework in HLS. BitBlender's architecture stores a large bit-vector in on-chip memories for ultra low latency. Moreover, it efficiently shares access among all input streams to allow high throughput even for large Bloom filters with low false-positive rates. To enable this efficient shared access, 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 to meet a user-provided Bloom filter specification. Experimental results show that, on the AMD/Xilinx Alveo U280 FPGA, BitBlender achieves a throughput up to 3,213 MQueries/s (i.e., 12.8 GB/s) for a 130Mib bit-vector with 0.005% false-positive rate. Compared to the highly-optimized Rust crate *fastbloom* running on 24 CPU threads, it achieves an average speedup of 8.89x, up to a maximum of 10.47x. Compared to the naïvely-duplicated multi-stream FPGA design, it achieves an average speedup of 5.30x, up to a maximum of 7.42x. BitBlender is available at https://github.com/SFU-HiAccel/BitBlender.

CCS Concepts: • **Hardware → Hardware accelerators**; *Reconfigurable logic applications*; • **Computer systems organization → Reconfigurable computing**; **High-level language architectures**.

Additional Key Words and Phrases: Bloom Filter, FPGA, High-Level Synthesis, Dynamic Scheduling, Design Automation

ACM Reference Format:
Kenneth Liu and Zhenman Fang. 2026. BitBlender: Scalable, High-Throughput Bloom Filter Acceleration on FPGAs with Dynamic Scheduling. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (April 2026), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 [16]. Filtering methods are crucial for addressing such issues, improving data quality by effectively identifying

Authors' addresses: Kenneth Liu, ksl24@sfu.ca, Simon Fraser University, Burnaby, British Columbia, Canada; Zhenman Fang, zhenman@sfu.ca, Simon Fraser University, Burnaby, British Columbia, Canada and University of Minnesota, Twin Cities, MN, USA.

Manuscript submitted to ACM

and eliminating noise. Among all types of filters, the Bloom filter [9], 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 [5], networking [10, 24], and bioinformatics [31].

Due to the significant slowdown of CPU scaling, prior studies [18–20, 22, 23, 30, 33] have demonstrated respectable performance improvements in accelerating Bloom filters using FPGAs. Unfortunately, their underlying architecture relies 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 to enable multiple input streams will rapidly exhaust 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 - and therefore, no speedup.

In this work, we propose BitBlender, the first dynamically-scheduled, configurable, high-throughput, and scalable multi-stream Bloom filter acceleration framework. We first propose an architecture template, composed of many task-parallel modules. 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 [25] to dynamically schedule conflicting accesses to execute sequentially and non-conflicting accesses to execute in parallel. Accordingly, we introduce a partition-to-stream unshuffle 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 a subtle deadlock caused by the OoO scheduling, we introduce a ratelimit logic to prevent one stream from running too far ahead of another. To maximize the effectiveness of our resource-sharing solution, we showcase several design iterations, and a highly frequency-optimized solution which achieves roughly the same frequency as the baseline FPGA design. The architecture template is dataflowed, where each module is fully pipelined.

Next, to aid users in generating their own BitBlender accelerator, we develop an automation tool which quickly identifies optimal designs for a requested Bloom filter specification. To do this, we develop a set of random forest models, to predict the resource utilization and achievable frequency of a BitBlender design. We also develop a probabilistic cycle-latency estimator, to understand and predict the pipeline stall rate. Together with the frequency predictor, this allows us to estimate the overall throughput of an instantiation of our accelerator. To support queries over a high-bandwidth network, we support automated integration with QSFP ports, with the open-sourced Aurora IP [28].

We evaluate BitBlender for a wide variety of Bloom filter configurations using randomly generated input queries. On the AMD/Xilinx Alveo U280 FPGA, BitBlender achieves a throughput up to 3,213 MQueries/s (i.e., 12.8 GB/s) for a 130Mib bit-vector, at a false-positive rate of 0.005%, exceeding the 100Gbps line rate. Compared to the 24-thread CPU implementation, BitBlender achieves up to 10.47x speedup; while compared to a naively-duplicated multi-stream FPGA design, BitBlender achieves up to 7.42x speedup.

In summary, this paper makes the following contributions:

1). A high-performance, configurable, and scalable multi-stream (i.e., input-parallel) Bloom filter accelerator template named BitBlender.

2). An automated design-generation tool, together with accurate probabilistic performance models, which quickly searches the design space for an optimal BitBlender instantiation for a user-provided Bloom filter specification.

3). A low-overhead dynamic arbitration scheme (arbiter, unshuffle, and deadlock prevention) realized in statically-scheduled HLS.
4). A comprehensive experimental evaluation and analysis of BitBlender.

2 BACKGROUND AND MOTIVATION

2.1 Basics of Bloom Filters

A Bloom filter [9] 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 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.

In order to design an algorithmically desirable Bloom filter, we wish to maximize the number of insertions (I) while simultaneously minimizing the false-positive rate (fp). In order to do this, users will require a large bit-vector, resulting in a large on-chip memory footprint. The equation that relates all four Bloom filter parameters is shown in Equation 1.

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

As shown, 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 $H = \frac{L}{I} \times \ln 2$ [10].

As a result, algorithmically desirable Bloom filters require large bit-vectors, and a configurable H to most efficiently minimize the false-positive rate, for a desired number of inserted elements I.

As will be detailed in Section 4.2.1 and Section 4.5.3, since hash functions can often be implemented efficiently on FPGAs, logic resource utilization is usually not a bottleneck for large Bloom filters. Instead, the constraining factor is the on-chip memory footprint of a large bit-vector. Therefore, this paper focuses on leveraging the otherwise-underutilized logic resources, in order to enable efficient parallel access to the shared (on-chip) memory-constrained data structure. Furthermore, since populating 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.

2.2 Baseline Single-Stream Bloom Filter Design

Fig. 1 shows our baseline single-input-stream Bloom filter query accelerator on the FPGA, similar to that in [18]. 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 pipelined design. Note, this sectioning does not affect the false-positive rate [27].

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. For clarity, we only show one copy of ComputeHash and Aggregate, and

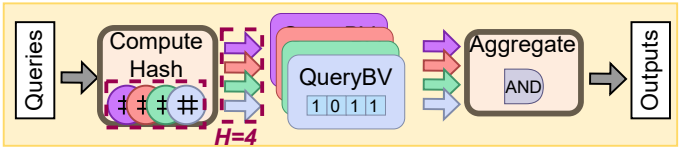


Fig. 1. Single-stream Bloom filter design, similar to [18]. Modules with bolded outlines have two duplicates.

we bold the outlines of these two modules to imply that there are actually two copies - one for each query in the pair. 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 [7], and can be easily replaced by an alternative. Next, the pair of queries for each hash function is sent to a single QueryBV unit, to look up the index in a BRAM or URAM bank - note, these banks have two ports, and therefore perform two bit-vector queries per cycle. 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.

2.3 Challenges to Multi-Stream Bloom Filter Scaling

Unfortunately, most existing Bloom filter designs [18, 23, 30] only focus on single-stream designs, 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 12.5 GB/s. Assuming a single-stream HLS (high-level synthesis) design runs at 200 MHz and each query is 32-bits long, it can only process 2 queries/cycle * 4 bytes/query * 200MHz = 1.6 GB/s, leaving about an 8x gap.

2.3.1 Challenges for naive multi-stream duplication. A naive approach to improve the query throughput via input parallelism is to simply duplicate the single-stream Bloom filter design, for multiple input 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 $fp = 0.01\%$, it needs a bit-vector length of $L = 143Mi$ with $H = 13$ 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/143Mib = 2 streams.

2.3.2 Challenges for multi-stream bit-vector sharing. To avoid 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 can be managed efficiently, allowing significant speedup potential.

2.3.3 The goal of this paper. Our goal is to design a *high-throughput, scalable, multi-stream Bloom filter accelerator* where multiple input 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.

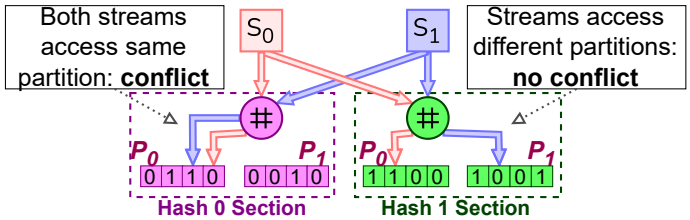


Fig. 2. Bit-vector sharing and access conflict example. S_0 and S_1 are input streams; P_0 and P_1 are partitions of each bit-vector section.

The rest of this paper is structured as follows. In Section 3, we first introduce an unoptimized architecture template of BitBlender, to illustrate the overall design. Then we introduce our novel dynamic arbitration scheme, and describe iterative optimizations to achieve high throughput while remaining configurable and scalable. We also introduce a cycle estimator to predict the dynamic stall rate of any given design, and an automation flow to quickly search the design space and generate an optimal BitBlender accelerator for a user-provided Bloom filter specification. A comprehensive evaluation is presented in Section 4. Finally, Section 5 describes related works, and Section 6 concludes the paper.

3 INITIAL BITBLENDER DESIGN AND IMPLEMENTATION

To achieve our goal, we present BitBlender, a configurable, scalable, high-throughput, 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 parallel input streams.

An overview of the BitBlender architecture template is given in Section 3.1. The arbitration scheme to enable bit-vector sharing is composed of two parts: the Arbiter, described in Section 3.2, and the Unshuffle module, described in Section 3.3. The finalized architecture template is summarized in Section 3.4. Additionally, we create a cycle-latency estimation model in Section 3.5, and a design automation tool in Section 3.6, to automatically generate an optimal BitBlender design on a given FPGA, according to the user-provided Bloom filter specification.

3.1 Initial BitBlender Architecture Template

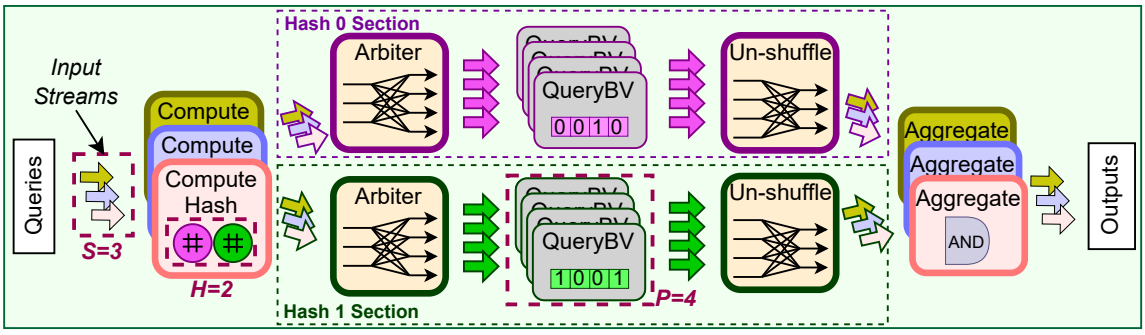


Fig. 3. Initial architecture of BitBlender. This depicts an example design with 3 input streams (S=3), 2 hash functions and 2 bit-vector sections (H=2), and 4 partitions per bit-vector section (P=4). Modules with bolded outlines have two duplicates.

An overview of our first-pass BitBlender architecture template is shown in Fig. 3 and every single module is fully pipelined with an initiation interval (II) of 1. Neighboring modules are connected using FIFOs. Each cycle, S query pairs are streamed into 2*S ComputeHash modules; modules with bolded outlines have two duplicates, so they can feed two SRAM ports, as explained in Section 2.2. 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.

When more than one stream tries to access the same bit-vector partition at once, this leads to an access conflict (shown in Fig. 2). To address these access conflicts, we introduce a stream-to-partition Arbiter module (pair) inside each section, which uses priority encoding logic [25] to schedule conflicting accesses to execute sequentially. Non-conflicting accesses to different bit-vector partitions execute in parallel.

Introducing 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 into streams. Second, inside each bit-vector partition, the outputs for multiple streams could be out-of-order (OoO). For example, data from stream 1 may be 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; meanwhile in hash section 1, stream 0 may go far ahead of stream 1. This could lead to a subtle deadlock involving the Aggregate modules for stream 0 and stream 1, which will be detailed in Section 3.2.2. To prevent the deadlock, we introduce ratelimiting 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.

In order to implement the dynamic scheduling, the Arbiter and Unshuffle modules need to make decisions based on metadata from each packet. The metadata that we require is summarized in Table 1. The partition index of each packet is computed in the ComputeHash module. The stream index and input index of each packet is determined by the loading logic.

Table 1. Required metadata for packets in BitBlender

Metadata Name	Range	Description
Partition Index (pidx)	$[1, P]$	Which partition of the bit-vector this packet accesses.
Stream Index (sidx)	$[1, S]$	The input stream from which this packet originated.
Input Index (iidx)	$[1, \infty]$	The numbering of this input, within input stream $\#sidx$.

The rest of this discussion makes frequent reference to the BitBlender configuration parameters. For the readers' convenience, the configuration parameters are summarized in Table 2.

Table 2. BitBlender configuration parameters: H, S, P, D, L, I, and fp

Config Parameter	Description
H	The number of hash functions to compute for each query.
S	The number of parallel input streams. This is specific to BitBlender.
P	How many partitions each bit-vector section is split into. This is specific to BitBlender.
D	The ratelimiting distance permitted by the Arbiter. This is specific to BitBlender.
L	The length, in bits, of the entire bit-vector.
I	The number of insertions to the Bloom filter.
fp	The false-positive rate of the Bloom filter.

3.2 Stream-to-Partition Arbiter Design

3.2.1 Basic Arbiter Design. The stream-to-partition Arbiter aims to dynamically schedule conflicting accesses (i.e.,requests 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 4, II=1).

Algorithm 1 Pseudo HLS code for stream-to-partition Arbiter

```
1:  function Arbiter(in_hash[S], out_hash[P])
2:      min_iidx = 0; slowest_stm; input_indices[S]; //Ratemonitoring, based on iidx metadata
3:      hash_buf[S] //buffer packets in registers, to allow explicit control logic
4:      while true do //pipelined, II=1
5:          for s in 0 to S do //read from each stream, unrolled
6:              //backpressure until hash_buf[s] is available
7:              if (!hash_buf[s].valid && !in_hash[s].empty()) then
8:                  hash_buf[s].value = in_hash[s].read()
9:                  hash_buf[s].valid = true
10:         for p in 0 to P do //write to each partition, unrolled
11:             //search all streams for a packet targeting partition p.
12:             //In a conflict, priority is given to the stream with smaller stream-idx.
13:             for s = slowest_stm; /*(S iterations)*/; s = (s+1) % S; do //unrolled
14:                 if (hash_buf[s].valid && hash_buf[s].value.pidx == p &&
15:                     hash_buf[s].value.iidx ≤ min_iidx + D //Ratelimiting
16:                 ) then
17:                     if out_hash[p].write_nb(hash_buf[s].value) then //write to partition p
18:                         hash_buf[s].valid = false
19:                         input_indices[s] = hash_buf[s].value.iidx //Ratemonitoring
20:                         break
21:         min_iidx = min(input_indices) //Ratemonitoring
22:         slowest_stm = argmin(input_indices) //Stream prioritization
```

Each cycle, it first reads up to S hashes, and buffers them in the `hash_buf` registers (lines 5-9). Next, for each partition p (line 10), it implements a priority encoder [25] (lines 13-16) to concurrently search all streams to find the ones accessing this partition. When there are access conflicts, it gives priority to the slowest stream, by manipulating the stream loop-order (on line 13). Finally, it attempts to write the packet from the chosen stream index (stored in the `hash_buf` registers) to the output FIFO of partition p (line 17). Here, the `write_nb` refers to a non-blocking write, meaning that if the FIFO is full, the write will not execute, and the if-condition will be false. If the write is successful, it releases the corresponding `hash_buf` entry (line 18). The ratelimiting and ratemonitoring (lines 15, 19, 21) are required for deadlock prevention, as discussed in Section 3.2.2.

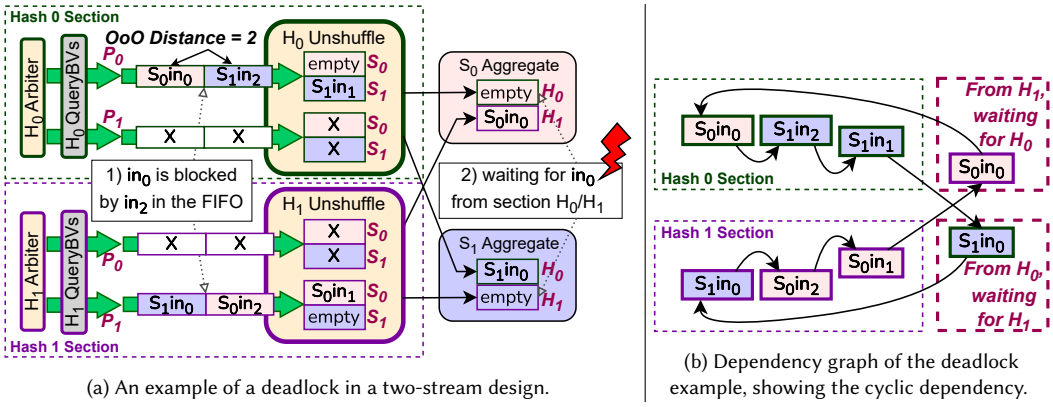
The actual throughput of the Arbiter, i.e., how many queries it dynamically processes per cycle, depends on how often the design backpressures. Backpressure occurs when the `hash_buf[s]` buffer entry is not released yet and thus the Arbiter is blocked from reading input indices from stream `s` (lines 7-8). Since corresponding `hash_buf` buffer entries are released when they are written out (lines 17-18), 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, the non-prioritized streams will not be written out in the current cycle, and will have to wait for the next cycles (lines 13 and 20). Second, if an output index FIFO is full due to backpressure caused by downstream modules (line 17). Third, if the ratelimiting logic (line 15) decides to pause a stream that is running too far ahead; the ratelimiting logic is designed to prevent deadlocks, which will be explained in Section 3.2.2.

A comprehensive analysis of expected throughput can be found in Section 3.5.

3.2.2 Deadlock Prevention in Arbiter. The dynamic arbitration leads to out-of-order (OoO) scheduling between multiple streams, which could potentially cause deadlocks. Note that for packets *which originate from the same stream*, they cannot become out of order.

An example deadlock is shown in Fig. 4a, with an accompanying dependency graph in Fig. 4b. As illustrated, in the top hash section H_0, after the Arbiter and QueryBV modules (on the left), stream 1 is running 2 packets ahead of stream 0 - that is, there is an out-of-order distance of 2. S_1in_0 is already in the S_1 Aggregate stage, S_1in_1 is in the Unshuffle stage, while S_1in_2 and S_0in_0 are waiting in the FIFO of bit-vector partition H_0-P_0.

Meanwhile, in the bottom hash section H_1, stream 0 and stream 1 are in the opposite situation. Here, stream 0 is running 2 packets ahead of stream 1: S_0in_0 is already in the S_0 Aggregate stage, S_0in_1 is in the Unshuffle stage, while S_0in_2 and S_1in_0 are waiting in the FIFO of bit-vector partition H_1-P_1.



(a) An example of a deadlock in a two-stream design.

(b) Dependency graph of the deadlock example, showing the cyclic dependency.

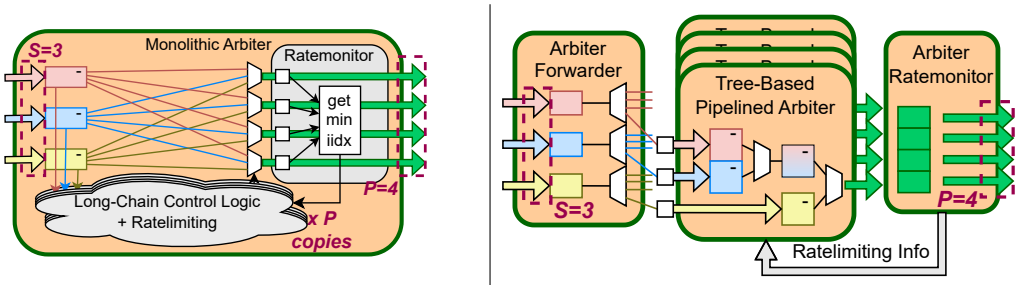
Fig. 4. Deadlocking in a multi-stream design: cyclic dependency example with two streams S_0 and S_1.

Now S_0 Aggregate has S_0in_0 from hash section H_1, and is waiting for S_0in_0 from hash section H_0. This is illustrated in the long, leftwards dependency arrow at the top of Fig. 4b. However, following the dependency chain in Fig. 4b, we see that S_0in_0 from H_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. In the same manner, 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 the ratelimiting logic into the Arbiter to prevent one stream from going too far ahead of another. To do this, each cycle it monitors all of the packets being sent out, and keeps track of the input index for the slowest stream. Each packet then compares its own input index to the slowest one, to get an OoO distance, and the packet is paused if the distance exceeds a threshold D. In the example in Fig. 4a, 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.

3.2.3 Frequency-Optimized Arbiter. As shown in Fig. 5a, the monolithic Arbiter implementation presented in Alg. 1 has a frequency issue: for each output partition, it uses a long logic-chain to decide which stream to prioritize in each cycle. This long logic delay results in a low achievable frequency. Worse, this delay scales with S. To alleviate these frequency bottlenecks, we propose an optimized design, shown in Figure 5b.

Optimization #1: logic separation. The optimized implementation separates the behaviour of the Arbiter into three stages: (1) forwarding, that implements the stream-to-partition packet separation; (2) priority arbitration, that decides which stream's packet will be sent to each partition; and (3) ratemonitoring, that keeps track of how much



(a) Unseparated monolithic Arbiter (b) Separated tree-based Arbiter

Fig. 5. Arbiter module designs ("-" denotes ratelimited modules).
(a): unoptimized implementation with long control paths.
(b): optimized implementation with separated logic and tree-based arbitration.

data each stream has sent (i.e., the current input index for each stream). This separation of logic into different modules simplifies the control logic of each module, which improves local routing congestion.

Optimization #2: tree-based structuring. On top of logic separation, our optimized design uses a **tree-based** Arbiter as opposed to a **monolithic** Arbiter. In our tree-based Arbiter, arbitration is performed on the streams in a pairwise manner. As shown in Figure 5b, packets from S_0 and S_1 are arbitrated first, and the winner is compared against a packet from S_2 for the final output. The number of stages in the tree in a general case is $\lceil log_2(S) \rceil$, since it is implemented as a balanced binary tree. Each node in the tree is specified as a separate module in the task hierarchy, and each node implements ratelimiting and priority arbitration, prioritizing packets from slower streams. Nodes communicate via FIFOs to localize the required logic-chain. This implementation allows a control logic of constant size within each node of the tree, no longer scaling with the number of streams, S. In this way, it unlocks greater parallelization, without creating a frequency bottleneck.

Optimization #3: multi-cycle exit condition. The final frequency optimization is to implement the exit-condition of the pipelined main loop for each of these modules as a multi-cycle path. In Alg. 1, we introduced the overall idea of the Arbiter, but simplified the details by ignoring the exit-condition of the main loop (line 4, in Alg 1). However, there must be an exit condition so that the design can complete, to signify a batch of data has been processed, and to re-initialize metadata counters for the next batch. The exit condition for the II=1 loops are computed by counting the number of data packets that have finished processing. However, a naïve implementation of this would compute and update the exit-condition every cycle. This creates a long combinational path, wherein the execution of iteration $i + 1$ is dependent on knowing if iteration i will write all current packets. In other words, it requires the control logic to decide, within only one cycle, if the loop breaks or iterates.

Our solution to this frequency bottleneck is shown in Alg. 2. Here, we describe the exit condition as a multi-cycle logic path, which relaxes timing constraints. To accomplish this, we calculate the number of completed packets by having two layers of counters: we distribute one counter per partition (line 2, `pkt_counter[P]`), and another total counter (line 3, `total_pkts_sent`). Partition-level counters are incremented by either zero or one each cycle, depending on if a packet is written to the corresponding partition (lines 15-16). The total counter is computed over P iterations of the loop, by accumulating one partition-level counter per iteration (line 5, `finish_check_pidx`). After P cycles, we have added all partition-level counters into the total counter. At this point, if the total counter indicates we have processed the expected number of packets, we set the finish condition to true (lines 7-8), and the loop will break on the next iteration. Otherwise, we restart the exit condition computation, to reinitialize the multi-cycle path (lines 9-11).

Algorithm 2 Pseudo HLS code for multi-cycle exit condition

```
1:  function ARBITER(in_idx[S], out_idx[P]], D)
2:      pkt_counter[P] //num. of packets sent, for each partition.
3:      finished_processing = 0, finish_check_pidx = 0, total_pkts_sent = 0;
4:      while (!finished_processing) do //pipelined, II=1
5:          total_pkts_sent += pkt_counter[finish_check_pidx++];
6:          if finish_check_pidx == (P-1) then
7:              if (total_pkts_sent == TOTAL_EXPECTED_PKTS) then
8:                  finished_processing = 1;
9:              else
10:                 total_pkts_sent = 0;
11:                 finish_check_pidx = 0;
12:         //... Reading/Writing/Ratemonitoring logic goes here ...
13:         for p in 0 to P do //write to each partition, unrolled
14:             //... Other Writing logic goes here ...
15:             if out_idx[p].write_nb(idx_buf[s_out].value) then
16:                 pkt_counter[p] += 1;
```

Table 3. Quality of results with alternative Arbiter implementations

H	*P*	*S*	*D*	*L*	Version	Freq	LUTs	FFs	DSPs	BRAMs	URAMs
3	**4**	**3**	**16**	**24**	Unseparated Monolithic	222	12.0%	7.7%	3.0%	18.9%	7.5%
					Separated Monolithic	225	12.9%	8.0%			
					Separated Tree, Singlecycle	220	13.3%	8.3%			
					Separated Tree, Multicycle	225	11.7%	7.7%			
5	**6**	**5**	**16**	**40**	Unseparated Monolithic	103	18.2%	10.1%	9.5%	20.5%	15.6%
					Separated Monolithic	138	21.3%	11.5%			
					Separated Tree, Singlecycle	218	25.0%	14.0%			
					Separated Tree, Multicycle	225	24.6%	14.0%			
7	**8**	**8**	**16**	**56**	Unseparated Monolithic	53	24.7%	12.5%	18.7%	30.5%	17.5%
					Separated Monolithic	77	33.5%	16.1%			
					Separated Tree, Singlecycle	177	52.5%	29.0%			
					Separated Tree, Multicycle	224	50.7%	28.8%			

Optimization results. In Table 3, we show a frequency and scalability ablation study, in which we place and route the entire BitBlender architecture under varying configurations and Arbiter implementations. All values shown are taken from post-route reports. As shown, both the unseparated monolithic and separated monolithic Arbiter designs show a marked decrease in frequency as the BitBlender parameters grow. Meanwhile, the tree-based Arbiter with logic separation, and a single-cycle exit condition is able to achieve a good quality of results (QoR), except in the largest design parameters. Finally, the tree-based Arbiter with logic separation, and a multi-cycle exit-condition is able to achieve a high frequency regardless of the parameters. The small variation in the achieved frequency can be attributed to randomness inherent in the place-and-route heuristics. In our most optimized version, the worst delay on a logic path is practically independent of our BitBlender parameters - *H*, *P*, *S*, *D*, and *L*.

However, these frequency improvements do not come for free. As shown, the logic resources required for the tree-based designs grow quickly, and can double relative to the unseparated monolithic design. Regardless, the optimization is well worth the logic cost, as continuing to scale the monolithic design results in unroutable nets - the design becomes limited by congestion and routability, rather than by logic usage. Additionally, the resource overhead comes in the form of LUTs and FFs - not memory resources, which are the constraining resource for our Bloom filter design.

Another trade-off of the tree-based Arbiter is that it produces a longer cycle-latency between the arbitration logic and the ratemonitor. The effects of this will be discussed further in Section 3.5.

3.3 Partition-to-Stream Unshuffle Design

3.3.1 Basic Unshuffle Design. The partition-to-stream Unshuffle module aims to reorganize the out-of-order (OoO) query results from P partitions, back to in-order query results in S streams, to prepare for the final aggregation. Alg. 3 shows pseudocode describing its fully pipelined design, and Fig. 6a shows a diagram of the initial Unshuffle implementation.

Algorithm 3 Pseudo HLS code for partition-to-stream unshuffle

```
1:  function UNSHUFFLE(in_value[P], out_value[S], D)
2:      value_buf[P][S][D] //buffer packets in registers, to allow explicit control
3:      next_iidx[S] = 0 //track the next input index to send to each stream
4:      while true do //pipelined, II=1
5:          for p in 0 to P do //read from each partition, //unrolled
6:              peek = in_value[p].peek()
7:              d = peek.iidx % D //get the buffer idx for this packet's input index
8:              if (!value_buf[p][peek.sidx][d].valid) then //backpressure until this buffer entry is available
9:                  in_value[p].read() //when the buffer entry is available, consume from FIFO
10:                 value_buf[p][peek.sidx][d].valid = true
11:                 value_buf[p][peek.sidx][d].value = peek
12:         for s in 0 to S do //write to each stream, unrolled
13:             //search all partitions for a packet from stream s: for each input index, only one packet can be valid across all partitions
14:             for p in 0 to P do //unrolled
15:                 d = next_iidx[s] % D //get the buffer idx in which we will find the packet with iidx == next_iidx
16:                 if (value_buf[p][s][d].valid  &&  value_buf[p][s][d].value.iidx == next_iidx[s]) then
17:                     //write the chosen packet, which contains a bit from the bit-vector, to stream s
18:                     if out_value[s].write_nb( value_buf[p][s][d].value ) then
19:                         value_buf[p][s][d].valid = false
20:                         next_iidx[s] += 1
21:                         break
```

Each cycle, it first peeks up to P QueryBV values (lines 5-6), checking metadata to decide their assigned spot in the `value_buf` buffer (line 7). Then, if a packet's spot is available, Unshuffle consumes the packet and buffers it (lines 8-11). This `value_buf` buffer is of size $P \times S \times D$ (line 2), to hold up to D packets for each stream, from each partition. The D dimension is introduced to address the deadlock issue, which will be explained further in Section 3.3.2. Next, for each stream s, it concurrently searches all partitions in the buffer for the next packet to send (lines 12-21). Since this next packet corresponds to a unique *input_idx* for each stream, there can only be one valid packet for each stream (as explained by the comment on line 13). Therefore, only one partition could hold the packet, so there will be no access conflicts. As a result, there is no prioritization logic required, as opposed to the Arbiter module. Finally, it attempts to write the chosen bit-vector value to the output FIFO of stream s (line 18). If the write is successful, it releases the corresponding `value_buf` buffer entry, and increments the `next_iidx` for stream s (lines 19-20).

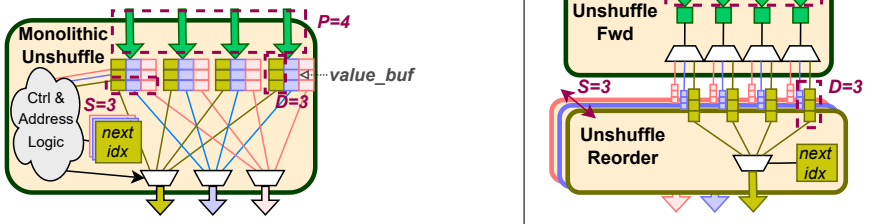
As with the Arbiter, the actual throughput of the Unshuffle module depends on how often the design backpressures. Backpressure occurs when a `value_buf` buffer entry is not released yet and the incoming value is mapped to buffer in the same entry. Like the Arbiter, `value_buf` entries are released when they are written out, so backpressure only occurs when Unshuffle cannot output data. This can happen in two ways. First, when an output FIFO is full, due to backpressure from downstream Aggregate modules. Second, when the next packet to send is not yet in the `value_buf`, due to the packet scheduling from the Arbiter.

A comprehensive analysis of expected throughput can be found in Section 3.5.

3.3.2 Deadlock Prevention in Unshuffle. The deadlock prevention logic in the Unshuffle module goes hand-in-hand with the ratelimiting logic in the Arbiter. To accomodate the ratelimiting distance D that is enforced by the Arbiter, the Unshuffle module has a buffer for each input stream to hold up to D values, as described in Section 3.3.1, to unblock its input FIFO from potential OoO packets, caused by faster streams. For example, in Fig. 4a, if the Unshuffle module had a buffer of size D=2 for each partition and stream rather than D=1, it would solve the illustrated deadlock, because the blocking packets, S_0in_2 and S_1in_2, could be consumed.

3.3.3 Frequency-Optimized Unshuffle. For the Unshuffle implementation in Alg. 3, we note that although it seems to implement a similar logic pattern to the Arbiter, it does not suffer from the long logic-chain that was optimized by the tree-based structuring from Section 3.2.3. This is because the Arbiter must assign a priority among several choices, where the priority can change between iterations; in contrast, the Unshuffle write logic is searching for a unique packet which has the correct data. There is no dynamically-changing prioritization among choices, and therefore, no long logic chain. However, like the Arbiter, the Unshuffle unit can be viewed as performing two jobs, as explained shortly. Thus, we adopt the same logic-separation optimization.

Optimization #1: logic separation. By separating the implementation into separate modules (similar to what we did with the Arbiter in Section 3.2.3), the control logic is simplified, which improves frequency. We separate the Unshuffle unit into two stages, as shown in Figure 6: (1) forwarding, which reorganizes packets from partitions back to their originating streams, and (2) reordering, which serializes the input indices of the packets from the P incoming partitions, to each stream.



(a) Monolithic Unshuffle, with $value\_buf$

(b) Separated Unshuffle, with FIFOs

Fig. 6. Unshuffle unit implementations.
(a): unoptimized implementation with complicated addressing.
(b): optimized implementation with separated logic and simplified buffering.

Optimization #2: FIFOs instead of buffers. This bifurcation of logic also reveals the path to another optimization. As mentioned in Section 3.2.2, the data packets within any given stream never change order. Therefore, within each of the P partitions, after the packets are separated according to the originating streams, the packets are then in-order within the partition. Thus, after the partition-to-stream separation is performed by the Unshuffle-forwarding unit, it is possible to remove the fully-partitioned $value\_buf$ buffer from the Unshuffle-reorder module. Instead, we implement it as a simple FIFO of size D as shown in Figure 6b. This greatly simplifies the next-packet logic, replacing it with a FIFO read operation from P partitions, and a small comparison between the P packets - since the correct data will always appear at the head of one of the input FIFOs.

Note that since the modules in Unshuffle do not need metadata counters that need to be reset, these modules do not need any reset logic. Therefore, in contrast to the Arbiter's frequency optimizations we do not adopt any multi-cycle exit-conditions.

Table 4. Quality of results with alternative Unshuffle implementations

H	*P*	*S*	*D*	*L*	Version	Freq	LUTs	FFs	DSPs	BRAMs	URAMs
3	**4**	**3**	**16**	**24**	Unseparated, Buf-based	211	14.1%	9.3%	3.0%	18.9%	7.5%
					Separated, Buf-based	225	14.2%	9.4%	3.0%		
					Separated, FIFO-based	225	11.7%	7.7%	3.0%		
5	**6**	**5**	**16**	**40**	Unseparated, Buf-based	(failed)	N/A	N/A	N/A	N/A	N/A
					Separated, Buf-based	155	29.4%	18.5%	9.5%	20.5%	15.6%
					Separated, FIFO-based	225	24.6%	14.0%	9.5%		
7	**8**	**8**	**16**	**56**	Unseparated, Buf-based	(failed)	N/A	N/A	N/A	N/A	N/A
					Separated, Buf-based	134	64.3%	41.0%	18.7%	30.5%	17.5%
					Separated, FIFO-based	224	50.7%	28.8%	18.7%		

Optimization results. Table 4 shows an ablation study in which we place and route the entire BitBlender architecture under varying configurations and Unshuffle implementations. All values shown are taken from post-route reports. As shown, the unseparated Unshuffle implementation faces significant difficulty in routing, and is unable to complete place-and-route for even moderately-sized BitBlender configurations. The separated Unshuffle implementation is able to complete place-and-route, but the buffer-based implementation uses significantly more LUTs and FFs, due to the logic searching and comparing every buffer entry in a $P \times D$ sized buffer. This leads to higher congestion and lower achievable frequency when compared to the FIFO-based implementation.

Moving from the buffer-based implementation to the FIFO-based implementation is a strong improvement with no noticeable downsides. This is because the buffer-based implementation allows for random access into the large packet buffer, but after we optimize the design with logic separation, the random access is restricted to only P FIFO heads.

3.4 Final Optimized Architecture

After optimizing the Arbiter and Unshuffle modules, the architecture template is composed of many highly scalable modules, allowing the design to operate at a high frequency for a very wide range of possible configurations. The final architecture diagram is shown in Figure 7. Here, the Arbiter and Unshuffle modules are split into their frequency-optimized counterparts, distinct from the initial architecture in Figure 3. Additionally, a summary of each module in the design is given in Table 5.

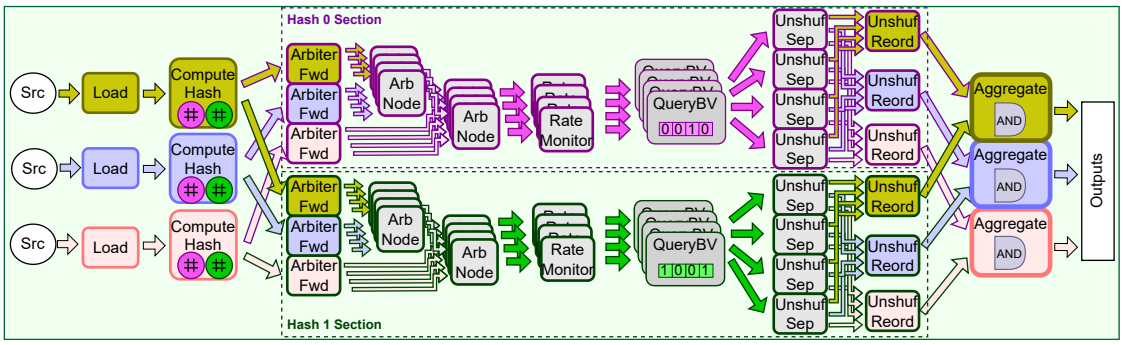


Fig. 7. Final architecture, including all frequency-improved modules. This depicts an example design with 3 input streams (S=3), 2 hash functions and 2 bit-vector sections (H=2), and 4 partitions per bit-vector section (P=4). Modules with bolded outlines have two duplicates.

Table 5. Module Descriptions and Data Forwarding Behaviour

Module Name	In | Out	Data Forwarding Behaviour
Load	1 | 1	All valid input packets are forwarded to the output.
Compute Hash	1 | H	If all H outputs are not full, an input packet is consumed and copied to each of the H outputs.
Arbiter Forwarder	1 | P	Input packets' target partition idx is inspected, and forwarded to the corresponding output.
Arbiter Node	2 | 1	Two input packets' input idx are inspected, and ratelimited if it's greater than $min\_iidx + D$. If both packets are valid and not ratelimited, the smaller input idx is forwarded to the output. If only one is valid and not ratelimited, it is forwarded to the output.
Ratemonitor	1 | 1	Input packets' input idx are inspected to update min_iidx, which is the largest $input\_idx$ that has been received by all streams. This min_iidx is given to all Arb Nodes.
QueryBV	1 | 1	All valid input packets are forwarded to the output.
Unshuffle Separator	1 | S	Input packets' stream idx are inspected, and forwarded to the corresponding output.
Unshuffle Reorder	P | 1	Each input packet's input idx is compared with $next\_iidx$. If they are equal, then that packet is sent to the output, and $next\_iidx$ is incremented.
Aggregate	H | 1	If all H inputs are valid, they are combined and sent to the output.

3.5 Cycle-Latency Estimator

In this section, we develop a mathematical model to predict the throughput of a given BitBlender configuration. That is, given some combination of H, S, P, and D, we will describe and predict the expected data forwarding patterns, and dynamic stalling behaviour. As explained earlier, every single module in BitBlender is fully pipelined with an initiation interval (II) of 1, and the entire design runs in a dataflow fashion. In an ideal case, BitBlender could achieve a throughput of $2 \times S$ queries per cycle (2 queries per cycle, for every stream). However, due to dynamic dataflow stalls, the actual throughput is:

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

To characterize the dataflow stall rate, we will start by explaining that the stall rate of BitBlender is well-estimated by only considering the behaviour of the Arbiter. Next, we will analyze the expected cycle latency of the Arbiter. Finally, we define the cycle-efficiency, to easily describe the expected dynamic stalling behaviour of BitBlender.

3.5.1 Dynamic Dataflow Stalls in BitBlender. A dataflow stall occurs in a module if it pauses a valid packet. Therefore, we do not consider a module to be stalling if all of its output FIFOs are full - we consider this stall to be due to some downstream module. Therefore, the dataflow stall rate of BitBlender is determined by modules whose data forwarding behaviour may pause valid packets even if an output FIFO is *not* full. As shown in Table 5, this includes only ComputeHash, Arbiter Node, Unshuffle Reorder, and Aggregate.

However, the ComputeHash module can only spawn a pipeline bubble if at least one of its outputs are full, but another output is not full - meaning a downstream module stalled a sufficient number of times in its execution. Similarly, the Aggregate module can only spawn a pipeline bubble if at least one of its inputs are empty - meaning an upstream module stalled a sufficient number of times in its execution. Both of these modules may only introduce pipeline bubbles as a result of other modules stalling. The Unshuffle Reorder module may pause a valid packet if the input index of the packet is not correct. However, the scheduling of packets is handled by the Arbiter logic - so the Reorder module's stall rate is also directly influenced by the Arbiter. Therefore, we can reasonably estimate that all dataflow stalls can be traced back to originate from the Arbiter.

3.5.2 Predicting Cycle Latency. In order to understand the scheduling behavior of the Arbiter, we will refer to its description in Section 3.2.1, accompanied by Alg. 1.

To recap, our initial Arbiter design has S input FIFOs, and P output FIFOs. Its behaviour is summarized as follows:

- **Reading Logic:** Each cycle, it reads up to S input packets (zero or one from each input FIFO).
- **Ratelimiting Logic:** It ratelimits packets if the packet has an $input\_idx > min\_iidx + D$, where min_iidx is the current input index of the slowest stream.
- **Prioritization Logic:** The Arbiter assigns each packet a priority. The highest priority is given to the packet with the lowest $input\_idx$.
- **Routing Logic:** For each of its P outputs, it checks which of the non-ratelimited packets need to be sent there. It sends the packet which has the highest priority.

To aid our analysis, we define a "set of inputs" to the Arbiter as the S packets that have the same $input\_idx$. For example, the 3rd set of inputs is the 3rd packet from each stream, as illustrated in Figure 8. In the following paragraphs we describe how we model the runtime (in cycles) of a BitBlender architecture.

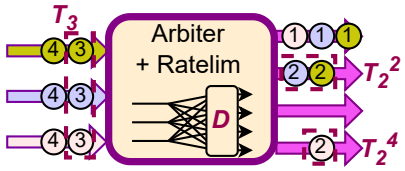


Fig. 8. Illustrations of input-sets (T_k) and partition-input-sets (T_k^p), to develop the analytical cycle-latency model.

Let T_k be the k'th set of inputs, as illustrated in Figure 8. The **reading logic** dictates that we consume at most one set of inputs per cycle. Hence, the earliest that T_k can be read is on cycle k. By the **ratelimiting logic**, none of the packets from T_k can be sent out until all packets from $T_{(k-D)}$ have been sent. The **prioritization logic** states that, in the event of an access conflict, we always try to send the packets with the lowest $input\_idx$. We model this by saying that each packet in T_k that maps to partition p cannot be sent out until all of the packets that also map to partition p, in all previous sets (T_{k-1} and lower) have been sent. Note, this interpretation of the prioritization logic is not entirely faithful to the true behavior of BitBlender, in which out-of-order input indices actually *can* appear on a partition. Despite this minor difference, the resulting cycle-latency model turns out to be extremely accurate.

Finally, by the **routing logic**, each packet in T_k is mapped to a partition. The number of cycles required for T_k to send depends on how many access conflicts exist in T_k - that is, how many packets in T_k target the same partition. **Define** T_k^p to be the set of packets from T_k with partition index p, as illustrated in Figure 8. Then, it takes $|T_k^p|$ cycles for the k'th set of inputs to send to partition p, since these packets must be serialized by the Arbiter.

Based on this analysis, we summarize our cycle-latency model in Equation 3, for input sets where $k \geq 1$. Since $FinalSendCycle$ is a recurrence relation on k, for consistency we define $FinalSendCycle(p, 0) = FinalSendCycle(p, -1) = \ldots = 0$.

$$FinalSendCycle(p,k) = \max \left\{ \begin{array}{l} \max_{\rho \in [1,P]} FinalSendCycle(\rho, k-D) + FeedbackLatency \\ FinalSendCycle(p, k-1) \\ k \end{array} \right\} + |T_k^p| \tag{3}$$

In this equation, $FinalSendCycle(p, k)$ denotes the final cycle in which a packet from T_k is sent to partition p. The first term (i.e., the multi-line *max* statement) predicts the cycle at which the k'th set of inputs *begins* to send to partition p. The second term (i.e., $|T_k^p|$) predicts how many cycles it takes for that set of inputs to finish sending, modelling the **routing** logic. The first line of the outer *max* statement models the **ratelimiting** behaviour. The second line models the **prioritization** logic, and the final line models the **reading** logic, as described previously.

Note that, when modelling the ratelimiting behavior, we add a $FeedbackLatency$ term. This term is added to model the frequency optimizations introduced in Section 3.2.3 - there are a few cycles of delay for the ratelimiting logic (implicit in the nodes of the Arbiter-tree) to receive rate-data from the Ratemonitor. In practice, we determine $FeedbackLatency$ based on the HLS-reported latency of the Arbiter nodes and the Ratemonitor.

To use Equation 3, we must find the value of the term $|T_k^p|$. But since we cannot know the partition index of each packet ahead of time, we will model it with a probability distribution.

We assume that our hash functions have a uniform output distribution. With this assumption, the partition index of each packet is also a uniform probability distribution. A set of inputs consists of S packets, where we consider each packet to be labelled with its partition index, $p \in [1, P]$. Then by definition, the number of packets in T_k^p is the number of packets in T_k that are labelled with p. We can view the labelling of the packets as a series of S independent Bernoulli trials (i.e., pass/fail trials [15]), where a trial is passed if the packet is labelled p, and failed if it is labelled with anything other than p. This shows that $|T_k^p|$ (i.e., the number of packets in T_k^p) can be described by a binomial probability distribution [15], with S trials, each having a success-probability of $1/P$:

$$Pr(|T_k^p| = n) = \binom{S}{n} \times (\frac{1}{P})^n \times (\frac{P-1}{P})^{S-n} \tag{4}$$

Henceforth, we consider $|T_k^p|$ to be a random variable, described by the binomial distribution shown in Equation 4. As shown, this probability distribution depends only on the number of streams, S, and the number of partitions, P. In Figure 9, we graph the probability distribution for $|T_k^p|$, as shown in Equation 4. As we can see, the entire graph shifts to the right as the number of streams, S, increases. On the other hand, the graph shifts to the left when the number of partitions, P, increases. Hence, the odds of an access conflict (i.e., the probability that $|T_k^p| \geq 2$) increases with S, and decreases with P. With this model, we can use Equation 3 to estimate the cycle-runtime of BitBlender.

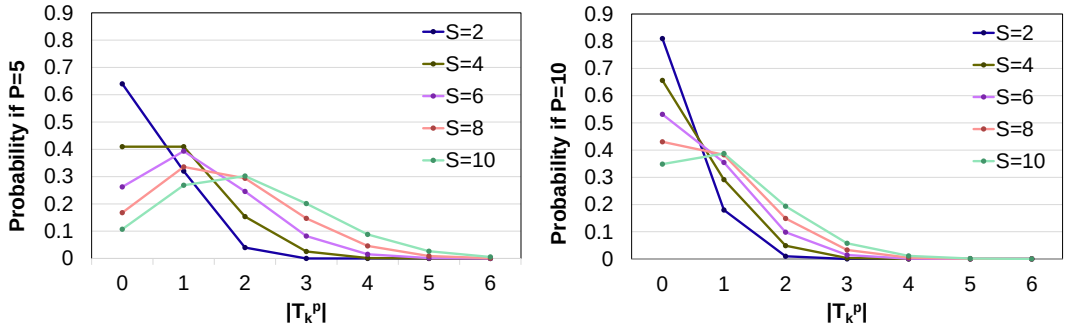


Fig. 9. Example probability distributions of $|T_k^p|$, with different P and S.

3.5.3 Predicting Stall Rate and Efficiency. The stall-rate model presented in Equation 3 is difficult to simplify further, due to the max function. However, since it is an additive recurrence relation, it approximately boils down to a sum over

many samples of the random variable $|T_k^p|$. By the law of large numbers [15], as the number of samples, n, increases, the expected value of all n samples tends towards the expected value of one sample multiplied by n. Practically speaking, this means that the expected runtime should vary minimally between different randomly-drawn input sequences.

To predict the stall rate of a given BitBlender configuration, we sample the random variable $|T_k^p|$ many times and iteratively compute $FinalSendCycle(t, k)$ for increasing k, to compute one possible value of $FinalSendCycle(p, K)$ for each partition $p \in [1, P]$. Here, K represents the total number of input query-pairs per stream. The largest $FinalSendCycle(p, K)$ across all partitions p then represents the estimated number of cycles needed to process $2 \times K \times S$ queries. From this, we may derive the $stall\_rate$ as defined in Equation 2; however, it is perhaps more intuitive to define the **efficiency** of a BitBlender design, as shown in Equation 5. Note that this definition of efficiency is equivalent to the factor of $\frac{1}{(1+stall\_rate)}$, from Equation 2. The efficiency represents the percentage of cycles spent performing useful work, as opposed to cycles spent waiting for dynamic scheduling overhead to resolve.

$$Efficiency = \frac{number\_of\_inputs}{runtime\_cycles} \approx \frac{K}{\max_{p \in [1,P]} FinalSendCycle(p, K)} \tag{5}$$

3.6 Automation Support for BitBlender

To allow users to easily generate optimized BitBlender hardware designs to meet their desired Bloom filter specifications, we develop a design automation toolflow shown in Fig. 10.

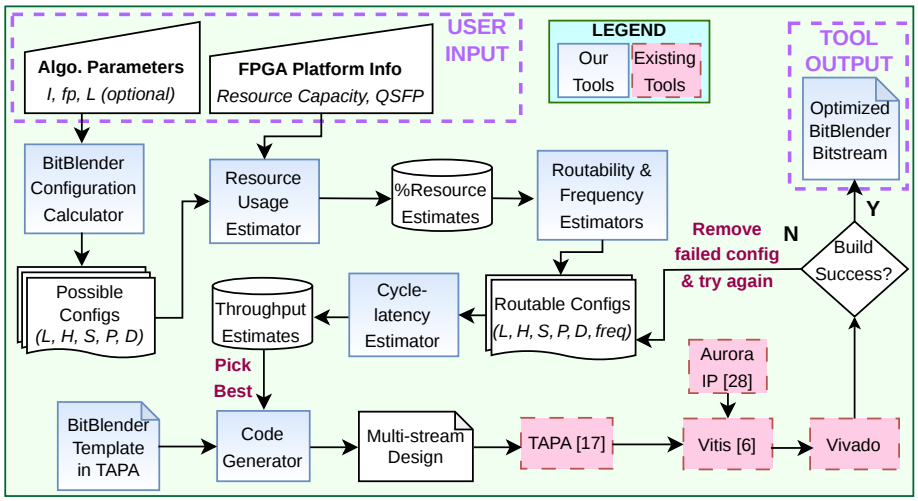


Fig. 10. Flowchart of design automation for BitBlender, to quickly search the design space and generate the optimized design.

The automated design space exploration tool takes the FPGA resources and Bloom filter parameters as input from the user, including whether or not the user wants to target a QSFP-based design or a DRAM-based design, as discussed in Section 3.7. The Bloom filter parameters include the number of expected inserts (I) to the Bloom filter, the desired false-positive rate (fp), and optionally the bit-vector length (L). Our configuration calculator then proposes many valid BitBlender configuration combinations, including the bit-vector length (L), number of hash functions (H), number of streams (S), number of partitions per bit-vector section (P), and ratelimiting distance (D).

Next, we use a random forest regressor model to predict the FPGA resources required for each proposed BitBlender configuration. Then, the predicted resources are divided by the user-provided FPGA resource capacity, to estimate the percentage-resource usages. We use a machine learning model rather than an analytical model for the resource estimator, because it is difficult to estimate the computational resources that each module requires, as they scale nonlinearly with the BitBlender configuration. Moreover, during the placement and routing of the design, the resource usages may be automatically optimized depending on the size and complexity of each module.

From there, we use frequency-predictor model, together with the probabilistic cycle-latency estimator which was described in Section 3.5.2, to predict the total throughput of the BitBlender design. The design with the highest estimated throughput is passed into the back-end building process. The building process begins with TAPA [17], which performs automatic coarse-grained floorplanning optimizations for the input design. It then calls Vitis and Vivado to generate the FPGA bitstream. Depending on user preference, it may link with the Aurora IP for QSFP support, as described in Section 3.7. If Vivado fails to generate the bitstream, the user is prompted to try again with the next best design.

3.6.1 Training the Resource and Frequency Estimators. To train the random forest regressors for resource and frequency estimation, we built 238 total BitBlender designs. These designs are split into four sweeps, where each sweep iterates through several values of H and a single other configuration parameter, while holding the other parameters the same. When a parameter was not being swept, its value was fixed as S=8, P=9, D=16, and the bit-vector section length (i.e., L/H) = 16.

For example, for each H from 3 to 11, we would also vary S from 2 to 9, to sweep through several numbers of streams while holding P=9, D=16, and the bit-vector section length as 16. We also swept through $P \in [4, 12]$, $D \in \{2, 4, 8, 16\}$, and section lengths $\in \{12, 16, 20, 24\}$. Finally, we also performed a sweep through $H \in [3, 15]$ with S=6, P=8, D=16, and a section length of 16. This resulted in 238 total builds, 148 of which completed placement and routing.

3.7 QSFP Port Integration

Modern datacenter FPGAs are equipped with high-bandwidth networking interface to support 100-Gigabit data rates and beyond. For example, many of the FPGAs in the AMD/Xilinx Alveo family (e.g., the U280 [2], U50 [1], U200, and U250 [3]) support QSFP28 ports, which are capable of reaching a 100Gbps data rate.

To support end-to-end streaming of queries from a network, we also add support for integrating the QSFP Port. To enable this, we use the Aurora IP module [28]. The Aurora IP is a prepackaged RTL IP, designed to interface between the FPGA fabric and a Gigabit transceiver, such as the QSFP28 port. It communicates with the FPGA fabric using the lightweight AXI-stream standard [8], which is realized in HLS with the `hls::stream` API. Based on whether the user specifies to add QSFP support, the code generator and build tool will automatically adjust the typical compilation flow of TAPA [17] to link our HLS-generated BitBlender IP and the Aurora IP. Unfortunately, this poses a slight challenge to TAPA's automated floorplanning, which is done on the BitBlender architecture before the linking step, and is therefore unaware of the Aurora IP. Furthermore, the Aurora IP relies on unscalable clocks in the FPGA fabric, which makes routing more difficult. To alleviate this problem, our resource estimators add a constant overhead when the user requests QSFP port integration, targeting a smaller design that is more likely to route.

4 RESULTS AND ANALYSIS

4.1 Experimental Setup

Baseline CPU implementation. We compare against the popular, high-performance, and actively-developed Rust crate fastbloom [32]. As of April 2025, this is one of the most popular Rust crates for Bloom filters, with over 240k downloads. It achieves high performance by being implemented as a partial blocked Bloom filter, in which the bit-vector is partitioned into cache-friendly blocks, and queries are localized in blocks to achieve low memory latency. For a fair false-positive rate comparison, we configure fastbloom to use 512-bit blocks, and we insert I elements into the Bloom filter prior to our random querying phase.

We write a benchmark to measure the querying time with 24 threads, and compile it using optimization level 3, link time optimization specified as "thin", and one codegen unit. Further, during compilation, we specify the `target-cpu=native`, and `target-feature=+avx2,+fma` rust flags, to ensure the resulting binary is able to leverage all SIMD intrinsics. We measure the runtime on a 14nm 12-core Xeon Silver 4214 CPU, with 24 hyper-threads and a 16.5MB L3 cache.

Hardware platform and software tools. We evaluate our BitBlender accelerator designs on the 16nm AMD-Xilinx Alveo U280 FPGA [2] 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 2.2.

For all FPGA-based designs built by us, including the naïve multi-stream designs, we use the automated floorplanning optimization tool TAPA [17] to improve timing closure, and Vitis v2022.1 [6] for hardware synthesis. All FPGA results are measured via on-board execution, and the resource utilization is from post place-and-route reports. The runtimes are measured using cycle counters built into the accelerator. For a fair comparison between the FPGA implementations, all bitstreams are generated in a fully automated manner, with no manual tweaks to the build strategies. That is, the place and route tool uses the default placement and routing strategies, and no manual floorplanning is done for any of the designs.

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.

4.2 Overall Performance Speedup of BitBlender

Table 6 shows the overall performance, measured in million-queries-per-second, for auto-generated BitBlender designs under various Bloom filter configurations. In this table, all BitBlender designs are generated using our automation tool, as described in Section 3.6. For each design point, we prompt the design automation tool to provide us with the top 5 designs, generate all of them, and report the single best-performing design in this table. In total, we attempted to generate 40 HBM-based BitBlender designs, and 40 QSFP-based BitBlender designs.

To generate the naïve multi-stream designs, we use a set of standard Bloom filter configuration equations [11, 27], to calculate the most space-efficient bit-vector size and corresponding number of hash functions, given the expected number of insertions and desired false-positive rate. We then attempt to build the resulting design with several different numbers of streams, S, and take the best result.

Compared to the 24-thread CPU baseline, BitBlender achieves an average speedup of 8.89x and a peak speedup of 10.47x. Compared to the naive multi-stream FPGA design, BitBlender achieves an average speedup of 5.30x and a peak speedup of 7.42x.

Table 6. 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 (bits)	H	S	P	D	LUTs	FFs	BRAM	URAM	DSP	(MHz)
4M	0.01%	BitBlender-HBM	3155 (9.53x)	96Mi	6	8	10	16	56.4%	31.6%	38.9%	35.0%	18.1%	211
		BitBlender-QSFP	3037 (9.18x)	115Mi	5	8	11	16	53.2%	30.3%	43.7%	42.2%	15.1%	211
		Naive multi-stream	832 (2.51x)	78Mi×2	13	2	-	-	18.4%	9.6%	57.7%	54.2%	6.3%	208
		24-thread CPU	331 (1x)	73.1Mi	13	-	-	-	-	-	-	-	-	-
4M	0.005%	BitBlender-HBM	3213 (9.77x)	130Mi	5	9	10	16	58.1%	32.1%	48.9%	45.8%	17.0%	208
		BitBlender-QSFP	2551 (7.75x)	130Mi	5	7	10	16	43.6%	24.8%	50.9%	45.8%	13.3%	186
		Naive multi-stream	784 (2.38x)	84Mi×2	14	2	-	-	19.0%	9.7%	61.4%	58.3%	6.8%	196
		24-thread CPU	329 (1x)	78.6Mi	14	-	-	-	-	-	-	-	-	-
4M	0.002%	BitBlender-HBM	3145 (9.83x)	132Mi	6	8	11	16	60.1%	33.0%	44.9%	45.0%	16.0%	204
		BitBlender-QSFP	2658 (8.31x)	112Mi	7	7	9	16	52.2%	29.5%	50.2%	40.8%	18.5%	206
		Naive multi-stream	892 (2.79x)	80Mi×2	16	2	-	-	16.2%	8.9%	56.0%	53.3%	4.8%	223
		24-thread CPU	320 (1x)	85.9Mi	15	-	-	-	-	-	-	-	-	-
6M	0.01%	BitBlender-HBM	3098 (10.46x)	144Mi	6	8	11	16	61.3%	33.7%	49.0%	50.6%	18.1%	201
		BitBlender-QSFP	2684 (9.07x)	144Mi	6	7	10	16	50.1%	28.1%	55.5%	50.0%	15.9%	203
		Naive multi-stream	450 (1.52x)	104Mi×1	13	1	-	-	13.8%	7.9%	40.8%	37.9%	3.2%	225
		24-thread CPU	296 (1x)	109.7Mi	13	-	-	-	-	-	-	-	-	-
6M	0.005%	BitBlender-HBM	2821 (9.86x)	136Mi	8	7	9	16	57.3%	31.5%	55.6%	46.7%	21.1%	212
		BitBlender-QSFP	2299 (8.04x)	135Mi	9	6	8	16	53.0%	29.9%	57.3%	45.9%	20.4%	213
		Naive multi-stream	380 (1.33x)	112Mi×1	14	1	-	-	14.1%	8.0%	43.2%	40.8%	3.4%	190
		24-thread CPU	286 (1x)	117.9Mi	14	-	-	-	-	-	-	-	-	-
6M	0.002%	BitBlender-HBM	2466 (9.00x)	153Mi	9	7	8	16	59.7%	32.2%	66.2%	50.6%	23.8%	194
		BitBlender-QSFP	2313 (8.44x)	153Mi	9	6	8	16	50.6%	28.1%	68.4%	50.6%	20.4%	199
		Naive multi-stream	422 (1.54x)	128Mi×1	16	1	-	-	14.7%	8.2%	48.0%	46.7%	3.9%	211
		24-thread CPU	274 (1x)	128.9Mi	15	-	-	-	-	-	-	-	-	-
8M	0.01%	BitBlender-HBM	2286 (9.14x)	160Mi	10	6	7	16	50.0%	27.1%	44.5%	62.5%	22.7%	205
		BitBlender-QSFP	1907 (7.63x)	162Mi	9	5	7	16	40.3%	22.2%	47.0%	61.9%	17.0%	207
		Naive multi-stream	408 (1.63x)	143Mi×1	13	1	-	-	14.4%	7.9%	52.4%	48.8%	3.2%	204
		24-thread CPU	250 (1x)	146.3Mi	13	-	-	-	-	-	-	-	-	-
8M	0.005%	BitBlender-HBM	2052 (8.38x)	165Mi	11	5	7	16	45.7%	24.2%	45.7%	61.9%	20.8%	217
		BitBlender-QSFP	1925 (7.86x)	165Mi	11	5	6	16	43.0%	23.5%	52.4%	57.3%	20.8%	214
		Naive multi-stream	390 (1.59x)	154Mi×1	14	1	-	-	14.7%	8.0%	55.7%	52.5%	3.4%	195
		24-thread CPU	245 (1x)	157.3Mi	14	-	-	-	-	-	-	-	-	-

Table 6 also lists the generated BitBlender configuration parameters (L, H, S, P, and D) by our automation tool, as well as resource utilization and frequency for the FPGA designs. As seen, BitBlender achieves a performance speedup over the naive multi-stream FPGA implementation, by using computational resources (LUTs and FFs) to manage access to the shared on-chip bitvector. A detailed resource breakdown is shown in Section 4.2.1. The number of modules instantiated, as well as the logic resources used by each module, scale with the choice of the following configuration parameters: H, S, P, and D. The memory usage is dedicated entirely to holding the bit-vector, and scales with L only.

This table also highlights the variance in the place and route tools. The critical path of the FPGA implementations is consistently within the query modules, due to long chains of cascaded BRAMs or URAMs acting as a single memory bank. However, while all designs managed to achieve around 200 MHz, there is a noticeable variance in the achieved frequencies, due to the high resource demand and fully-automated design flow. The final frequencies vary from 190MHz to 225MHz.

The generated configurations for the QSFP-based BitBlender designs achieve slightly lower throughput compared to the HBM designs for the reasons explained in Section 3.7.

4.2.1 Resource Breakdown of BitBlender. In Table 7, we show the per-module resource cost of the same BitBlender-HBM designs that are shown in Table 6. Note that all of the BRAM and URAM usage is used in the QueryBV module; to save

Table 7. Per-module resource breakdown. For each module, the reported cost is the sum of resources across all instances in the design. Load and Aggregate are negligible (less than 1%) and thus not reported. BRAM/URAM are used solely in QueryBV; they are already reported in Table 6 and thus not repeated here. Arbiter is subdivided into arbiter forward (Fwd), tree-based pipelined arbiter (Node), and ratemonitor (RateMon) components. Unshuffle is subdivided into the unshuffle separator (Sep) and reorder (Reord) components.

I	fp	L (bits)	H	S	P	D	Resource	Compute	Arbiter				QueryBV	Unshuffle		
								Hash	Fwd	Node	RateMon	Total		Sep	Reord	Total
4M	0.01%	96Mi	6	8	10	16	LUT	9.22%	2.07%	22.01%	14.95%	39.03%	1.03%	2.22%	4.16%	6.38%
							FF	13.61%	0.56%	13.93%	0.76%	15.25%	0.34%	0.84%	1.00%	1.84%
							DSP	18.10%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
4M	0.005%	130Mi	5	9	10	16	LUT	9.06%	2.02%	23.80%	14.60%	40.42%	1.08%	2.17%	4.09%	6.26%
							FF	13.04%	0.54%	14.91%	0.67%	16.12%	0.29%	0.81%	0.95%	1.76%
							DSP	17.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
4M	0.002%	132Mi	6	8	11	16	LUT	6.23%	2.33%	25.33%	17.00%	44.66%	1.15%	2.55%	4.73%	7.28%
							FF	9.79%	0.67%	18.15%	0.96%	19.78%	0.41%	1.09%	1.28%	2.37%
							DSP	16.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
6M	0.01%	144Mi	6	8	11	16	LUT	9.27%	2.24%	24.35%	16.35%	42.94%	1.35%	2.45%	4.55%	7.00%
							FF	13.77%	0.57%	15.57%	0.82%	16.96%	0.36%	0.94%	1.10%	2.04%
							DSP	18.10%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
6M	0.005%	136Mi	8	7	9	16	LUT	10.53%	2.21%	20.32%	15.66%	38.19%	1.25%	2.30%	4.35%	6.65%
							FF	15.04%	0.62%	12.28%	0.86%	13.76%	0.40%	0.84%	1.01%	1.85%
							DSP	21.10%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
6M	0.002%	153Mi	9	7	8	16	LUT	11.93%	2.30%	20.47%	16.05%	38.82%	1.49%	2.32%	4.45%	6.77%
							FF	16.21%	0.68%	11.83%	0.87%	13.38%	0.38%	0.80%	0.98%	1.78%
							DSP	23.80%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
8M	0.01%	160Mi	10	6	7	16	LUT	11.33%	2.00%	15.03%	13.74%	30.77%	1.54%	1.95%	3.77%	5.72%
							FF	15.05%	0.63%	8.36%	0.83%	9.82%	0.40%	0.65%	0.81%	1.46%
							DSP	22.70%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
8M	0.005%	165Mi	11	5	7	16	LUT	10.51%	1.94%	12.30%	13.16%	27.40%	1.75%	1.88%	3.58%	5.46%
							FF	13.86%	0.60%	6.68%	0.90%	8.18%	0.45%	0.62%	0.77%	1.39%
							DSP	20.80%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

space, we do not repeat the BRAM and URAM information presented in Table 6. Additionally, the Load and Aggregate modules together make up less than 1% of LUT and FF usage - thus these modules are not reported in Table 7, either.

First, as shown, all of the DSP utilization is consumed by the hash computation, and the arbitration logic do not use DSP resources. In BitBlender, DSP is typically the least used resource and not the bottleneck. Even in naive multi-stream designs, as shown in Table 6, DSP is typically the least used resource and not the bottleneck.

Second, among DSP, LUT, and FF, LUT is usually the most consumed resource. The majority of the LUT resource cost is due to the scheduling overhead required by the arbitration, especially the tree-based pipelined arbiter (denoted as Arbiter Node in Table 7) and ratemonitor. When the number of streams (S) and partitions (P) are higher, the relative cost of arbitration increases, to allow for even higher throughput.

Third, the FF resource cost is mostly spent between arbitration and the hash computation. When the number of hash functions (H) is higher, the ComputeHash module starts to dominate FF resource utilization.

This resource breakdown data again highlights the advantage of BitBlender - in large, memory-constrained Bloom filter designs, the LUT resources are heavily underutilized by the naively-duplicated multi-stream design (see Table 6 and also later in Table 9). In our dynamic arbitration scheme, we leverage these otherwise-unused LUT resources to manage concurrent accesses to the bit-vector, resulting in an overall throughput improvement.

4.2.2 Performance of BitBlender on a Real-World Dataset. In order to profile BitBlender on a sample of real-world data, we use the Wikimedia organization's dataset [12], representing Wikipedia pages that were flagged to be revised. We profile this data on two different BitBlender configurations: 4M-0.01%, and 8M-0.005% - i.e., the smallest and largest configurations of BitBlender in Table 6, respectively.

In the former configuration, this dataset is processed 20% slower than our randomized synthetic inputs, at approximately 2,524 MQueries/sec. In the latter configuration, it is 13% slower, at approximately 1,785 MQueries/sec. This dataset is slower than the purely-random synthetic inputs because the dataset contains many duplicate entries (the same articles are often flagged for revision many times). These duplicate entries have identical hashes, and therefore repeatedly access the same bit-vector partition, leading to increased access-conflicts in our arbitration logic.

4.3 Analysis for Different Bloom Filter Configurations

Here we analyze the BitBlender throughput when we vary one of its design parameters: H (number of hash functions), S (number of streams), P (number of partitions), and D (ratelimiting distance). Unless otherwise stated, by default, we set H=7, S=8, P=9, D=16, and the bit-vector section length (i.e.,L/H, bits per hash function) to 2^{24} - approximately 16 million. This default configuration is capable of achieving I=4M insertions under a false positive rate of fp=0.002%.

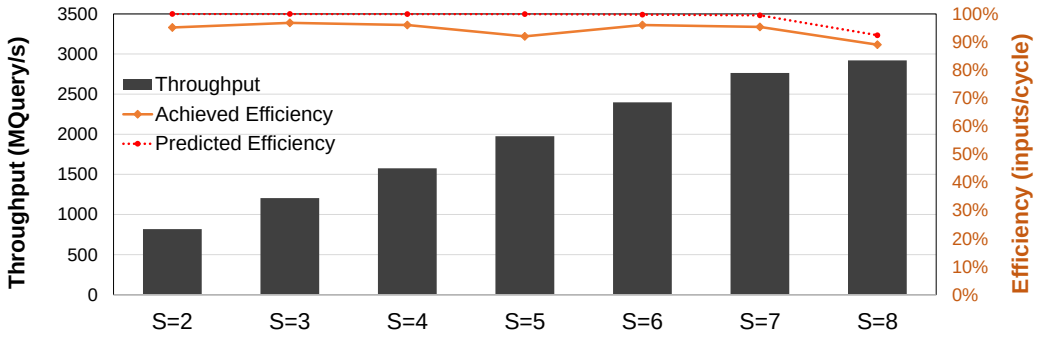


Fig. 11. BitBlender throughput at different S

#Streams impact. Shown in Fig. 11, the throughput of BitBlender increases roughly linearly with S. However, when S approaches P (9, in this case), the throughput plateaus slightly because the efficiency (defined in Equation 5) decreases. Here, we also see that the cycle-performance model we developed in Section 3.5.2 slightly overestimates efficiency if the efficiency is high. The efficiency dip at $S = 5$ is not predicted by our model. This can be attributed to the change in topology in the Arbiter-tree: with 4 or 6 inputs, the tree is much more balanced than it is with 5 inputs; this imbalance results in slightly decreased efficiency. However, the efficiency dip at $S = 8$ is predicted by our model, as shown by the predicted efficiency following the achieved efficiency.

Table 8 also lists the corresponding resource utilization and frequency when S changes. At this BitBlender configuration, the single-stream design resource utilization is dominated by BRAM (47.74%) and URAM (40.83%), which limits the naive duplication of streams. In our BitBlender design, each additional stream adds about 7% more LUTs, 4% more FFs, and 3% more DSPs. Note the BRAM and URAM utilization does not change, as the bit-vector is shared among all streams. When S increases, the achievable frequency does not change much; however, at S=9, the placer cannot find a solution due to overutilization.

#Hashes impact. Shown in Fig. 12, the throughput is practically independent of H. Any differences in throughput can be explained by the difference in achieved frequency; the efficiencies of these designs are almost identical, hovering

Table 8. BitBlender resource utilization at different S.
*S=1 shows a baseline single-stream implementation, **without** the Arbiter and Unshuffle units.

S	LUTs	FFs	BRAM	URAM	DSP	Frequency
1*	9.32%	6.18%	47.74%	40.83%	1.24%	220 MHz
2	19.52%	11.02%			5.32%	216 MHz
3	24.74%	13.54%			7.96%	209 MHz
4	30.39%	16.53%			10.59%	205 MHz
5	37.13%	20.02%			13.23%	220 MHz
6	43.24%	23.93%			15.87%	208 MHz
7	51.56%	28.40%			18.51%	207 MHz
8	60.15%	33.34%			21.14%	205 MHz

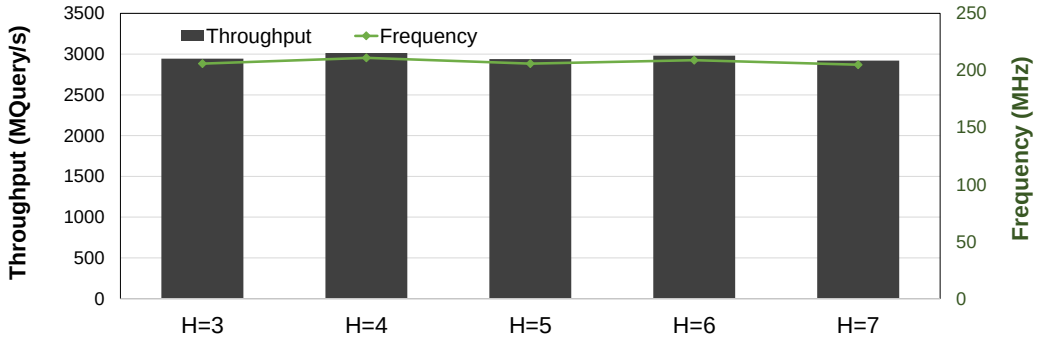


Fig. 12. BitBlender throughput at different H, with P=9, S=8, D=16

between 89.0%, and 89.4%. Meanwhile, our cycle estimator predicts an efficiency of 92.4% for all of these designs: the expected cycle latency is independent of H, as explained in Section 3.5.2. The expected cycle latency is independent of H, as explained in Section 3.5.2.

In Fig. 13, we show the throughput under slightly lower-throughput configurations, which leaves resources to build for larger values of H. Again, the differences in throughput are mostly explained by differences in achieved frequency. The efficiencies of these designs varies slightly more, but still remains mostly static, ranging from 90.9% to 94.7%. Meanwhile, our cycle estimator predicts an efficiency of 99.8%, overestimating the true efficiency.

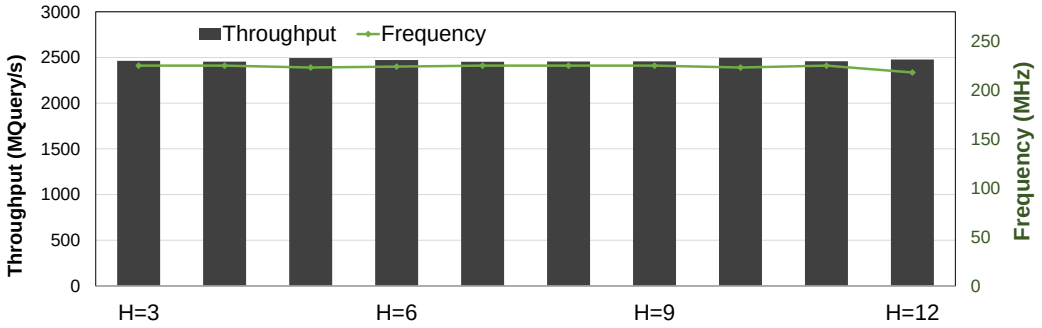


Fig. 13. BitBlender throughput at different H, with P=8, S=6, D=16, L/H=10

#Partitions impact. Shown in Fig. 14, the throughput increases when P increases. Intuitively, this makes sense, since BitBlender's parallelism is constrained by $min(S, P)$. The throughput increase roughly follows the curve of the achieved efficiency - any deviations are due to variations in clock frequency.

This figure highlights that our cycle-performance model is very accurate when the efficiency is low, but it still overestimates higher efficiency values. As shown, the efficiency does continue to improve from increasing P, but it slowly plateaus as P increases above S.

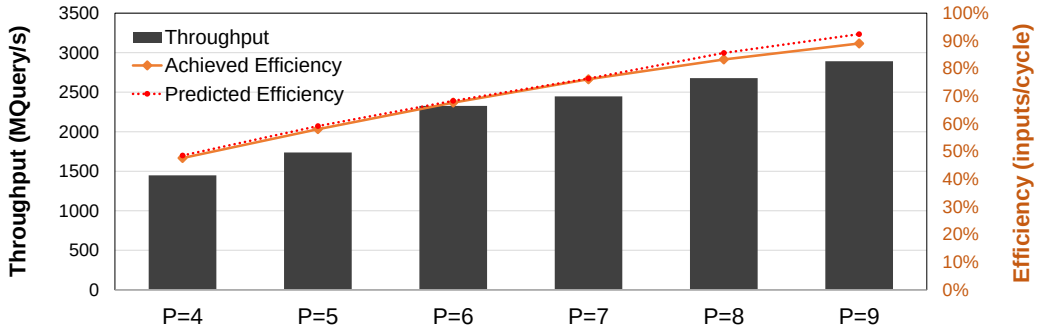


Fig. 14. BitBlender throughput at different P

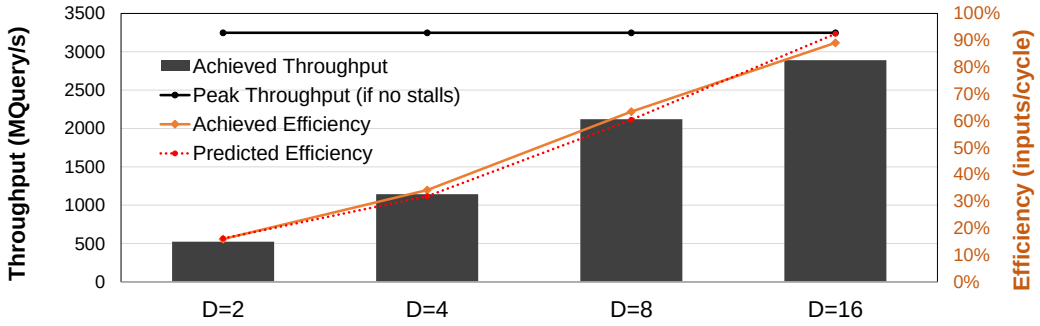


Fig. 15. BitBlender throughput at different D

Ratelimit distance impact. Shown in Fig. 15, the throughput increases when D increases. A smaller D leads to less opportunities for OoO scheduling in the Arbiter, and thus a lower efficiency, as it becomes more likely that a stream is ratelimited, and must wait for other streams to progress. On the other hand, a larger D leads to more complicated logic in the Ratemonitor module. In fact, the HLS scheduler is unable to find a valid schedule for the Ratemonitor, when D=32. Nonetheless, even with P=9 and S=8, the efficiency we can achieve with D=16 is just above 89%.

4.4 Automation Tool Performance Analysis

Here we analyze our design automation tool, which was described in Section 3.6. In Fig. 16, we use the designs built with the automation tool in Section 4.2 to compare the throughput estimation of our automation framework against the final achieved throughput. This figure shows that the end-to-end throughput estimation closely matches the majority of designs, showing that both the frequency estimator and the cycle latency model are reasonably accurate. The average relative error across all of these builds is 4.1%, with a maximum of 11.7%, and a minimum of 0.1%.

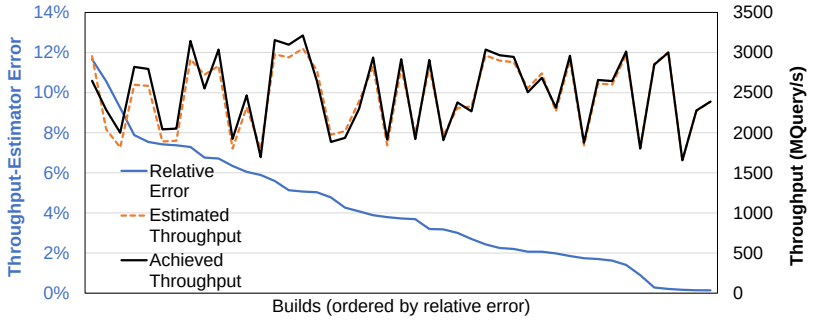


Fig. 16. BitBlender estimated end-to-end throughput vs achieved throughput

4.4.1 Cycle Latency Estimator Analysis. In this section, we evaluate the cycle-latency estimator over the 148 successful builds, which were detailed in Section 3.6.1. These builds sweep through BitBlender configurations in a similar fashion to Section 4.3. These 148 builds were run on FPGA with seven randomly-generated batches of inputs. The seven runtimes were averaged to get the achieved efficiency of each design. In Fig. 17, the achieved efficiency of on-board execution is compared against the efficiency predicted by our cycle-latency estimator from Section 3.5. Additionally, we report the absolute error and relative error between the predictions and ground truth. The results are ordered in decreasing absolute error.

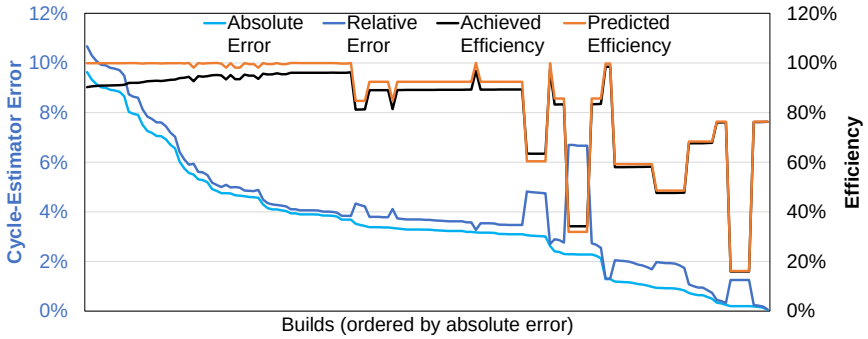


Fig. 17. BitBlender estimated cycle-efficiency vs achieved cycle-efficiency

This illustrates the accuracy of our cycle-latency estimator. Across all builds, the maximum relative error is 10.7%, while the average relative error is 4.1%. The maximum absolute error is 9.6%, and the average absolute error is 3.5%. We believe the difference between our predicted efficiency and the achieved efficiency is due to several factors.

The first factor is additional latency in the Arbiter, caused by extra pipeline registers inserted by the automated floorplanning tool, TAPA [17]. These pipeline registers are inserted to divide long routing paths and increase frequency. This would explain the erratic efficiency differences in Fig. 13, where we do not expect any efficiency differences.

The second factor is that our cycle-latency estimator only models the behaviour of *a single* Arbiter, and neglects the fact that we have H running together. The achievable throughput would be dictated by the worst-performing Arbiter of the group. We also do not model the behaviour of the Unshuffle module, which might generate a few pipeline stalls of its own. The final factor is that there seems to be a constant cycle overhead, which could be the result of the multi-cycle exit paths of our modules, from Sections 3.2.3 and 3.3.3. These two factors would explain the tendency to over-estimate only when the estimated efficiency is very high, because this constant overhead prohibits the design from approaching 100% efficiency.

4.5 Comparison with Previous FPGA Accelerator Designs

This section compares BitBlender against prior state-of-the-art Bloom filter accelerators on FPGA. Here, we compare against three works: Sateesan et al. [30], Wada et al. [33], and Kang et al. [23]. The works by Sateesan et al. [30] and Wada et al. [33] are both high-throughput SRAM-based Bloom filter accelerators, profiling the microarchitecture of hash functions and query engines in an RTL design. The work from Kang et al. [23] is a state-of-the-art DRAM-based Bloom filter accelerator, where the bit-vector is stored in off-chip DRAM. Their work profiles using radix sorters to enable bursting in the off-chip bit-vector accesses. Further discussion of these works can be found in Section 5.1.

We note that the FPGA used in the DRAM-based accelerator [23] is smaller than our AMD/Xilinx Alveo U280 FPGA: they target a Virtex-7. This work is not open-sourced for us to reproduce results on a larger FPGA. However, even on a larger board, their query throughput is limited by accesses to the off-chip bit-vector: there is only so much bursting that can be enforced, so while they may support larger Bloom filters, their achievable query throughput is very difficult to improve. The FPGAs used in the SRAM-based works are as follows. Sateesan et al. target a VU7P Virtex Ultrascale+ FPGA in [30], while Wada et al. target a VU9P Virtex Ultrascale+ FPGA in [33]. These FPGAs have a comparable number of resources to our Alveo U280 evaluation platform, although the U280 is the largest. The available resource counts of these FPGAs are shown in Table 9. In addition, we showcase the maximum number of duplications for two sample naïvely duplicated multi-stream designs, to illustrate scalability constraints, discussed further in Section 4.5.2.

Table 9. Resource comparisons between Alveo U50, U250, U280, VU7P, and VU9P FPGAs [4], and illustration of when scaling Bloom filter designs is logic bound and when it is on-chip memory bound: the maximum number of naive stream duplicates is limited by the smaller number of duplicates calculated based on available logic and memory resources (highlighted in red).

FPGA	LUTs	FFs	BRAM	URAM	DSP	Max #streams with H=6, L=6Mi		Max #streams with H=13, L=143Mi	
						logic-only	memory-only	logic-only	memory-only
U50	0.87M	1.74M	1344	640	5952	22	37	11	1
U250	1.34M	2.75M	2000	1280	11508	34	71	16	3
U280 (used by us)	1.30M	2.61M	2016	960	9024	33	56	16	2
VU9P (used in [33])	1.18M	2.36M	2108	938	6840	30	56	14	2
VU7P (used in [30])	0.79M	1.58M	1406	625	4560	20	37	10	1

We note that all of these prior studies target a single-stream design, or a naïvely-duplicated multi-stream design. Therefore, they face a very harsh capacity/throughput tradeoff, even in a larger FPGA with more resources. The throughput in [30] is determined only by the frequency, and therefore will not increase even on a larger board. In [33], their naïvely-duplicated multi-streamed architecture achieves high throughput by severely limiting the bit-vector section length - it is limited to 144k bits, and therefore they cannot scale to support a large Bloom filter with a high number of insertions. Furthermore, this work neglects to run the generated design on FPGA, which ignores the challenges of achieving high off-chip memory data rates. Their reported throughput is an estimate, based on their expected pipeline efficiency and achieved build frequency.

Table 10. Performance comparison with prior work. * denotes performance under a special case.

Work	H	Query Rate (GB/s)	Frequency (MHz)	E2E Latency Per-Query (ns)	L	I @ fp = 0.1%	I @ $fp = 10^{-12}$
BitBlender (ours, SRAM)	5	12.85	208	582.7	130Mi	4,704k	172k
	11	8.21	217	497.7	165Mi	11,999k	1,331k
Wada et al. [33] (SRAM)	40	7.78	162	(not reported)	5.48Mi	265k	100k
Sateesan et al. [30] (SRAM)	12	5.55	462	4.1*	0.25Mi	18k	2.3k
BunchBloomer [23] (DRAM)	8	0.19	250	(not reported)	512Mi	36,752k	2,156k

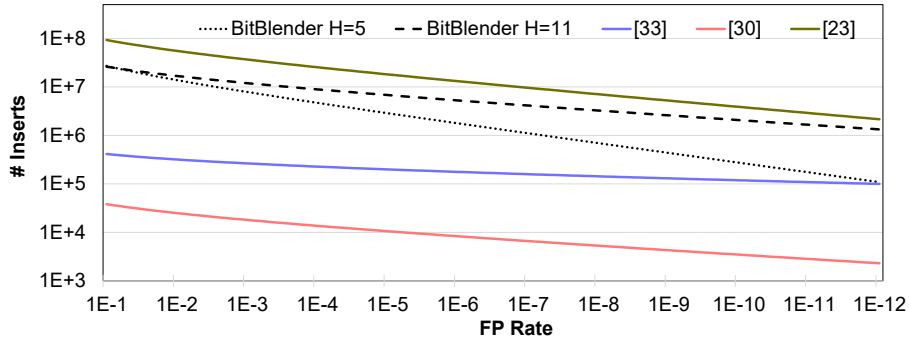


Fig. 18. Bloom filter comparison with prior work. x-axis shows the range of the false-positive (FP) rate, rightwards is better. y-axis shows the number of inserts the Bloom filter can support under a specific FP rate, higher is better.

Table 10 shows the performance of BitBlender against these prior FPGA designs. Additionally, Fig. 18 compares the supported Bloom filter size: namely, it shows the maximum number of insertions (I) at different false-positive rates (fp) for each accelerator design. The two BitBlender configurations shown here are (1) the fastest and (2) the largest Bloom filters previously shown in Table 6.

When prioritizing throughput (i.e., for the smaller Bloom filter), BitBlender can achieve up to 2.32x and 1.65x higher query rates than [30] and [33], respectively, while still supporting more insertions at the same false positive rate. Relative to the DRAM-based FPGA implementation [23], the fastest BitBlender configuration achieves about 68x higher throughput, while supporting anywhere from 4.6x to 20x fewer insertions, depending on the false positive rate.

On the other hand, when prioritizing the Bloom filter size, BitBlender can achieve orders-of-magnitude larger Bloom filters than [30] and [33], while still outperforming their processing rate. Compared against the DRAM implementation [23], the largest BitBlender configuration achieves about 43x higher throughput while supporting 1.6x to 3x fewer insertions.

4.5.1 Per-Query Latency Comparison. Our work aims to improve the throughput by introducing extra arbitration logic, which adds overhead for per-query latency. In the compared designs, our end-to-end latency per query is 121 and 108 cycles, respectively. At our achieved frequency, this corresponds to 582.7 and 497.7 ns, as shown in Table 10.

The latency breakdown of our design is further shown in Table 11. In this table, the first two data rows correspond to the two BitBlender designs in Table 10, while the last row illustrates a special case when the section length is a power-of-two. This table illustrates that the hash computation costs over half of our latency when the section length is not a power-of-two. The arbiter adds a latency overhead between 14 cycles and 26 cycles, while the multi-stream aggregation and output logic occupies 13 cycles. Other modules such as load, queryBV, and unshuffle only take 2 to 3 cycles.

Table 11. BitBlender latency breakdown (cycles per module). The first two configurations correspond to our BitBlender configurations reported in Table 10. The last configuration shows a special case where the section length is a power of two, allowing us to apply the same modulo optimization used in the work from Sateesan et al. [30].

BitBlender Configuration					**Per-module latency (cycles)**						**Total latency**
H	L	S	P	Section length (L/H)	Load	ComputeHash	Arbiter	QueryBV	Unshuffle	Aggregate	
5	130Mi	9	10	26Mi	2	74	26	3	3	13	121
11	165Mi	5	7	15Mi		73	14				108
7	112Mi	8	8	16Mi		11	23				55

The latency from the hash computation is actually due to a high-latency modulo operation: in order to retain a valid address into a bit-vector section, the hash output is modded by the section length, L/H. When the section length is a power-of-two, this modulo operation can be implemented significantly cheaper using a right shift. As shown in the design with $L/H = 16$ (last row), the latency of ComputeHash is reduced from more than 70 cycles to 11 cycles and our end-to-end latency is cut roughly by half.

The studies from Wada et al. [33] and Kang et al. [23] do not report latency information. In the work from Sateesan et al. [30], they are able to achieve very low query latency by designing a high-frequency and low-latency hash function for a much smaller bit-vector size. However, the latency reported in their work is operating within a special case: this accelerator is only profiled on designs where the bit-vector section length is a power-of-two. As such, when the hash-output is modded by the bit-vector section length, this operation costs almost no runtime - their reported latency is 2 cycles. It is unclear what the latency would be in a more general case, although it would be lower (i.e., better) than that of BitBlender. Also note that Sateesan et al. [30] only support very small bit vector sizes, approximately two orders-of-magnitude smaller than ours.

4.5.2 Resource Efficiency Comparison. To perform a performance-per-resource comparison between Bloom filter implementations, it may be tempting to directly divide the throughput by the number of resources used. For the sake of illustration, the direct comparison is shown in Table 12, where throughput per resource is measured in kB/s per LUT and per kb SRAM. However, this comparison results in skewed data that heavily prefers small Bloom filters, which are less algorithmically desirable (e.g., the work from Sateesan et al. [30] supports a Bloom filter size that is almost two orders-of-magnitude smaller than ours). This is because a larger Bloom filter consumes a lot more hardware resources to implement, while the throughput of the single-stream design does not change relative to Bloom filter size; therefore, when the throughput is directly normalized, the small Bloom filters get a much higher throughput per resource number.

For a more fair comparison between Bloom filter implementations, we present the throughput normalized by two factors: 1) FPGA resource utilization, and 2) A Bloom filter algorithmic "goodness" metric. This Bloom filter goodness metric is computed by multiplying the number of insertions (I) by the negative log of false-positive rate (fp), as shown in Equation 6. The higher the "goodness", the better.

$$goodness = I * -log(fp). \tag{6}$$

Normalizing by this goodness metric rewards large Bloom filters; meanwhile, normalizing by resource usage rewards small Bloom filters. Normalizing by both resource utilization and this goodness metric allows for a more fair tradeoff analysis. In Table 13, we compare the normalized throughput for each accelerator, which is given by $throughput * goodness/\#resources\_used$, where throughput is measured in GB/s, and resources are measured in the raw resource utilization (not as a percentage of available resources). Note, the *goodness* is dependent on the false-positive rate, which is why we report the normalized metric at two different values of *fp*. As shown, our design can be configured to be competitive with Sateesan et al. [30] in terms of both logic efficiency, and memory efficiency, while remaining more scalable - their high throughput relies on achieving high frequency in a small design. We also significantly outperform other designs by Wada et al. [31] and BunchBloomer [22].

4.5.3 More Discussion on Resource Bottleneck and Scaling Limitation. As presented in Section 4.2.1, for large Bloom filters, the resource bottleneck to scale the multi-stream designs is the available on-chip memory to support a large bit vector, instead of the logic resources. As shown in Table 6, the naive multi-stream design can often only duplicate 1 or 2

Table 12. Comparison of throughput normalized by only resource usage, ignoring algorithmic parameters. Higher is better.

Design	Throughput per LUT (kB/s per LUT)	Throughput per kb SRAM (kB/s per kb SRAM)
BitBlender (H=5)	16.97	79.28
BitBlender (H=11)	13.78	40.17
Wada et al. [33]	18.96	20.36
Sateesan et al. [30]	8,219.02	19,263.33
BunchBloomer [23]	0.73	5.63

Table 13. Comparison of throughput normalized by resource usage and algorithmic goodness. Higher is better.

Design	Throughput × Goodness per LUT (fp = 0.1%)	Throughput × Goodness per LUT (fp = 10^{-12})	Throughput × Goodness per kb SRAM (fp = 0.1%)	Throughput × Goodness per kb SRAM (fp = 10^{-12})
BitBlender (H=5)	401.44	22.14	1,875.58	103.46
BitBlender (H=11)	495.93	220.04	1,446.16	641.67
Wada et al. [33]	15.07	22.75	16.19	24.43
Sateesan et al. [30]	443.83	226.85	1,040.22	531.67
BunchBloomer [23]	79.97	18.76	621.18	145.76

streams due to the high BRAM and URAM usage, while the LUT is significantly underutilized. Note this is not just the case for our specific FPGA (Alveo U280), but also true for other common datacenter FPGAs listed in Table 9.

For small Bloom filter designs that require less on-chip memory to hold the bit vector, the resource bottleneck could potentially shift from the memory resources to logic resources; note this is not the focus of this paper though. An illustration of the scaling bottleneck for different FPGAs is shown in the last four columns of Table 9. These columns show the maximum number of naive duplicates for two different designs: a low-memory design, targeting a small Bloom filter with H=6 and L=6Mi; and a high-memory design, targeting a large Bloom filter with H=13 and L=143Mi. In the low-memory design, the duplication opportunity is often constrained by logic utilization, and therefore the bit-vector sharing scheme of BitBlender is not advantageous. However, in the high-memory design, the duplication opportunity is constrained by memory utilization, which is how BitBlender is able to provide a speedup: by leveraging the otherwise-unused logic resources to provide dynamic sharing of the constrained memory resources. In practice, based on the single-stream design resource utilization and the total available FPGA resources, one can determine whether naive scaling would be bounded by logic resources or memory resources (based on the highest utilization percentage). BitBlender enables users to better scale large Bloom filters.

4.6 Comparison with GPU Accelerator Designs

In [26], McCoy et al. implement an array of optimized Bloom filter alternatives on GPU. Their work focuses on feature-rich filters to provide support for a broader set of applications. However, their filter designs add these additional features at the cost of performance. Therefore, for a fair comparison, we simply compare to their GPU-optimized Bloom filter variant.

On an NVIDIA Tesla V100 GPU (on 12nm technology node, slightly lower than our Alveo U280 FPGA), their Bloom filter implementation achieves approximately 4,000 MQueries/sec of throughput, for designs with bit-vector lengths (L) ranging from 64 Mib - 1 Gib. In terms of power efficiency, we found that their design consumes about 100W of power, meaning their performance-per-watt is about 40 MQueries/sec/W.

In comparison, according to Vivado's power estimation, the power consumption of BitBlender ranges from 33W to 40W for the designs shown in Table 6. The smallest BitBlender design shown (i.e., 4M-0.01%) consumes about 35W, giving it a performance-per-Watt of around 90 MQueries/sec/W. The largest BitBlender design (i.e., 8M-0.005%)

consumes only 33W, giving it a performance-per-Watt of around 62 MQueries/sec/W. So our FPGA design is about 1.5x to 2.25x more energy efficient.

5 RELATED WORK

5.1 Bloom Filter Acceleration on FPGA

Harwayne-Gidansky et al. proposed the first single-stream Bloom filter accelerator, and achieve a throughput of 2 queries per cycle at approximately 210 MHz [18], for a peak query input rate of 1.6 GB/s. Other studies utilized Bloom filters as part of a larger accelerator on FPGA, such as flow-table lookups for software-defined networking [22], equi-join operations in databases [19], and DNA sequence comparisons [20]. However, they only support single-stream acceleration.

In [33], Wada et al. implement a high-throughput, byte-stream-based Bloom filter. In this design, each new byte is combined with several previous bytes to define a new query, as a rolling window over the byte-stream. They focus on developing fast streaming hash functions by microarchitecting BRAM-based and DSP-based hashing engines. Their performance modelling predicts strong throughput numbers (up to 7.78 GB/s of input queries) at a low false-positive rate, by scaling to use many hash functions. However, they limit the number of insertions to 100K, and their design is only slightly scalable, because the bit-vector size for each hash function is restricted to the length of one half of a URAM block - 144k bits. Moreover, this work neglects to run their design on FPGA, and relies on performance modelling without accounting for the characteristics of off-chip memory access.

In [30], Sateesan et al. focus on implementing high-frequency hashing functions on FPGA to accelerate Bloom filters. They focus on developing a new hardware-friendly hashing algorithm to achieve high frequency, and use their hash design in the Bloom filter architecture from [29]. Like Wada et al., they report good throughput numbers (up to 5.55 GB/s of input queries) at a low false positive rate, by achieving very high frequencies. To do so, they severely limit the number of insertions into the Bloom filter, only permitting 7,260. Their design is unlikely to scale with the size of a Bloom filter, because cascading more SRAM banks to implement a larger bit-vector will cause longer combinational delays, degrading the achievable frequency.

In [23], Kang et al. focus on very large, DRAM-based Bloom filters. Unlike prior studies, this work stores the bit-vector in off-chip DRAM, allowing a significantly larger memory footprint at a cost of decreased throughput. They propose a method to buffer memory accesses, grouping them together to leverage bursting in the DRAM memory access pattern. However, their work is constrained by the DRAM access bottleneck, and they only achieve 0.19 GB/s of query input rate. Furthermore, this design is difficult to scale; their memory-access buffer is designed specifically for the size of the FPGA on which it's implemented, and porting it to a different FPGA requires significant engineering work.

In all of these prior works, the underlying Bloom filter architecture relies on a single-stream design, which is unable to share the bit-vector for high throughput and scalability. BitBlender is the first work to support scalable multi-stream acceleration for Bloom filters.

5.2 Static Scheduling and Dynamic Scheduling

Dynamic scheduling in traditional HLS. Modern high-level synthesis (HLS) compilers generate datapaths by relying on static, compile-time scheduling guarantees. However, many compute patterns require complicated resource sharing schemes. An automated compile-time scheduling cannot find good opportunities for dynamic resource sharing, resulting in performance slowdowns. These opportunities for resource sharing need to be explicitly architected by designers, and

described in the HLS code for the compiler to generate. In [14], Du et al. proposed a dynamic arbitration module in their sparse matrix-vector (SpMV) accelerator, synthesized using statically-scheduled HLS. This work uses dynamic scheduling to balance computational load among processing elements, as needed by the irregular data pattern from the compressed sparse matrix format. For the task of SpMV, their outputs do not need to be in the same order as the inputs, because their data are simply fed into a multiply-accumulator, which is a commutative operation. Therefore, they have no need to reorder the dynamic data pattern, and do not have any danger of deadlocking.

Dynamically-scheduled HLS compilers. Another line of work is the recently developing dynamic HLS compiler toolflow. For example, in [21], Josipović et al. introduced Dynamatic, a dynamically-scheduled HLS compiler, which synthesizes a datapath consisting of elastic components [13] that can tolerate dynamic data rates. Their results showcased a greatly improved cycle-performance for dynamic workloads, at the cost of greatly increased resources and degraded clock frequency. However, this compilation flow inserts another layer of abstraction on top of traditional HLS. In traditional HLS, designers can understand the high-level idea behind their design, and examine the generated instruction-schedule of each module, to understand the behavior. In designs generated by a dynamically-scheduled compiler, the instruction-schedule is dynamically determined, and the behavior is even more opaque to the designer.

In BitBlender, we demonstrate opportunities for implementing advanced dynamic arbitration schemes with our proposed Arbiter and Unshuffle modules, together with the deadlock prevention logic, in statically-scheduled vendor HLS. By explicitly writing the dynamic behaviour, it allows us to understand and predict the throughput to a high degree of accuracy.

6 CONCLUSION

In this paper, we introduce BitBlender, the first high-throughput, 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 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, together with accurate performance and resource estimators, to generate the best-performing BitBlender accelerator on FPGA. Experiments on the Alveo U280 FPGA show that BitBlender achieves a throughput up to 3,213 MQueries/s (i.e., 12.8 GB/s) for a 130Mib bit-vector with 0.005% false-positive rate, which matches the line rate of its 100 Gbps (i.e., 12.5 GB/s) network interface. Compared to a highly-optimized 24-thread CPU benchmark, BitBlender achieves an average speedup of 8.89x, up to a maximum of 10.47x. Compared to the naïvely-duplicated multi-stream FPGA design, it achieves an average speedup of 5.30x, up to a maximum of 7.42x. The source code for BitBlender is open-sourced under the BSD 3-Clause license. It is available here: https://github.com/SFU-HiAccel/BitBlender.

ACKNOWLEDGMENTS

This work is partly supported by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, and CFI John R. Evans Leaders Fund.

REFERENCES

[1] AMD/Xilinx. 2024. Alveo U200 and U250 Data Center Accelerator Cards Data Sheet (DS962). https://docs.amd.com/r/en-US/ds962-u200-u250. https://docs.amd.com/r/en-US/ds962-u200-u250

[2] AMD/Xilinx. 2024. Alveo U280 Data Center Accelerator Card Data Sheet (DS963). https://docs.amd.com/r/en-US/ds963-u280/Summary. https://docs.amd.com/r/en-US/ds963-u280/Summary

[3] AMD/Xilinx. 2024. Alveo U50 Data Center Accelerator Card Data Sheet (DS965). https://docs.amd.com/r/en-US/ds965-u50/Summary. https://docs.amd.com/r/en-US/ds965-u50/Summary

[4] AMD/Xilinx. 2024. AMD UltraScale+ FPGAs Product Selection Guide. https://docs.amd.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide. https://docs.amd.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide

[5] AMD/Xilinx. 2024. Vitis Database Library. https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-database.html Last accessed March 16, 2024.

[6] AMD/Xilinx. 2024. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development

[7] Austin Appleby. 2016. smhasher. https://github.com/aappleby/smhasher. https://github.com/aappleby/smhasher [Accessed 15-Feb-2024].

[8] ARM Ltd. 2010. *AMBA AXI-Stream Protocol Specification.* ARM Ltd. https://developer.arm.com/documentation/ihi0051/latest/

[9] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (Jul 1970), 422–426.

[10] Andrei Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1 (Nov 2003). https://doi.org/10.1080/15427951.2004.10129096

[11] Wikipedia Contributors. 2025. Bloom filter — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Bloom_filter#Probability_of_false_positives. https://en.wikipedia.org/wiki/Bloom_filter#Probability_of_false_positives [Accessed 3-March-2025].

[12] Wikipedia Contributors. 2025. Wikimedia Downloads. https://dumps.wikimedia.org/enwiki/20250901/. https://dumps.wikimedia.org/enwiki/20250901/ [Accessed 01-September-2025].

[13] J. Cortadella, M. Kishinevsky, and B. Grundmann. 2006. Synthesis of synchronous elastic architectures. In *2006 43rd ACM/IEEE Design Automation Conference.* 657–662. https://doi.org/10.1145/1146909.1147077

[14] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. 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* (Virtual Event, USA) *(FPGA '22).* Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3490422.3502368

[15] William Feller. 1968. *An Introduction to Probability Theory and Its Applications.* Vol. 1. Wiley. 146–153 pages.

[16] Diego García-Gil, Julián Luengo, Salvador García, and Francisco Herrera. 2019. Enabling Smart Data: Noise filtering in Big Data classification. *Information Sciences* 479 (2019), 135–152. https://doi.org/10.1016/j.ins.2018.12.002

[17] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (Dec 2023), 31 pages.

[18] Jared Harwayne-Gidansky, Deian Stefan, and Ishaan Dalal. 2009. FPGA-based SoC for real-time network intrusion detection using counting Bloom filters. In *IEEE Southeastcon.* 452–458. https://doi.org/10.1109/SECON.2009.5174096

[19] Binhao He, Meiting Xue, Shubiao Liu, and Wei Luo. 2021. Bloom Filter-Based Parallel Architecture for Accelerating Equi-Join Operation on FPGA. *Electronics* 10, 15 (2021). https://doi.org/10.3390/electronics10151778

[20] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. 2008. Mercury BLASTP: Accelerating Protein Sequence Alignment. In *ACM Trans. Reconfigurable Technol. Syst.*, Vol. 1. Association for Computing Machinery, New York, NY, USA, Article 9, 44 pages.

[21] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18).* Association for Computing Machinery, New York, NY, USA, 127–136.

[22] Enio Kaljic, Almir Maric, and Pamela Njemcevic. 2022. Bloom filter based acceleration scheme for flow table lookup in SDN switches. In *2022 XXVIII International Conference on Information, Communication and Automation Technologies (ICAT).* 1–6. https://doi.org/10.1109/ICAT54566.2022.9811185

[23] Seongyoung Kang, Tarun Sai Ganesh Nerella, Shashank Uppoor, and Sang-Woo Jun. 2022. BunchBloomer: Cost-Effective Bloom Filter Accelerator for Genomics Applications. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL).* 9–16. https://doi.org/10.1109/FPL57034.2022.00014

[24] Bruce Maggs and Ramesh Sitaraman. 2015. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review* 45 (Jul 2015), 52–66. https://doi.org/10.1145/2805789.2805800

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

[26] Hunter McCoy, Steven Hofmeyr, Katherine Yelick, and Prashant Pandey. 2023. High-Performance Filters for GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23).* Association for Computing Machinery, New York, NY, USA, 160–173. https://doi.org/10.1145/3572848.3577507

[27] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis* (2nd ed.). Cambridge University Press, USA, 109–111.

[28] Gerrit Pape. 2025. AuroraFlow. https://github.com/pc2/AuroraFlow. https://github.com/pc2/AuroraFlow [Accessed 15-March-2025].

[29] Yan Qiao, Tao Li, and Shigang Chen. 2014. Fast Bloom filters and their generalization. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (2014), 93 – 103. https://doi.org/10.1109/TPDS.2013.46

[30] Arish Sateesan, Jo Vliegen, Joan Daemen, and Nele Mentens. 2022. Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. *Microprocessors and Microsystems* 93 (2022), 104619. https://doi.org/10.1016/j.micpro.2022.104619

[31] Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. 2010. Classification of DNA sequences using Bloom filters. *Bioinformatics* 26, 13 (May 2010), 1595–1600. https://doi.org/10.1093/bioinformatics/btq230 arXiv:https://academic.oup.com/bioinformatics/article-pdf/26/13/1595/48851430/bioinformatics_26_13_1595.pdf

[32] tomtomwombat. 2025. fastbloom: The fastest Bloom filter in Rust. No accuracy compromises. Compatible with any hasher. https://github.com/tomtomwombat/fastbloom/. https://github.com/tomtomwombat/fastbloom/ [Accessed 4-March-2025].

[33] Takuma Wada, Naoki Matsumura, Ryota Yasudo, Koji Nakano, and Yasuaki Ito. 2019. Efficient implementations of Bloom filter using block RAMs and DSP slices on the FPGA. *Concurrency and Computation: Practice and Experience* 33, 12 (2019), e5475. https://doi.org/10.1002/cpe.5475 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5475