

# CoBloom: An FPGA Accelerator System for Bloom Filter Insertion in Genomics Applications

Patrick Hardison

Department of ECE  
Duke University  
Durham, USA  
pgh11@duke.edu

Chris Kjellqvist

Department of Computer Science  
Duke University  
Durham, USA  
cmk91@duke.edu

Ning Liang

Department of Computer Science  
Duke University  
Durham, USA  
ning.liang@duke.edu

Lisa Wu Wills

Department of Computer Science  
Duke University  
Durham, USA  
lisa@cs.duke.edu

**Abstract**—The rapidly expanding volume of genomic datasets causes processing speed to become a significant bottleneck in genomic analysis workflows. FPGA acceleration presents an effective means to optimize computationally intensive workflows. K-mer counting is a widely used operation that involves recording each substring within a nucleotide sequence. This is frequently done with a counting Bloom filter, a data structure that requires multiple hash computations and memory accesses to a large lookup table per K-mer operation. This work presents two primary contributions. First, we profile Bloom filter implementations, examining hash rate performance, memory access patterns with and without prefetching, and the impacts of sorting to identify scaling bottlenecks and optimal resource allocation strategies. Second, we present CoBloom, an FPGA-accelerated counting Bloom filter which employs a hybrid design combining hardware-based hash computation with CPU-managed memory operations. CoBloom’s architecture addresses the identified bottlenecks to create a more efficient K-mer counting pipeline.

**Index Terms**—Field programmable gate array, genomics, hardware acceleration, performance analysis

## I. INTRODUCTION

Computational genomics is an expanding field that increasingly demands high-throughput solutions for massive genomic datasets. For instance, genetic sequencing requires read alignment to reconstruct the full genome, which is commonly solved by Smith-Waterman [14] or Needleman-Wunsch [12] algorithms. Both are dynamic programming problems that are poorly accelerated by vector architectures. Consequently, FPGAs often are good fits for these applications [7], [8], [10].

For our work, we consider K-mer counting, a common operation in genomics workloads. A *K-mer* is a nucleotide sequence  $K$  base-pairs long. By breaking reads into overlapping K-mers, assemblers can efficiently identify shared sequences, build graphs of overlaps, and reconstruct contiguous sequences even from short reads. This approach helps resolve repeats and errors by leveraging the redundancy in the K-mer coverage across the dataset. Geneticists often want to know how many times a given K-mer appears, if at all, within a given genome. To achieve this, the most common approach is to utilize Bloom filters, a probabilistic data structure that checks set membership in constant time, sacrificing perfect accuracy for significant performance gains and a reduced memory footprint.

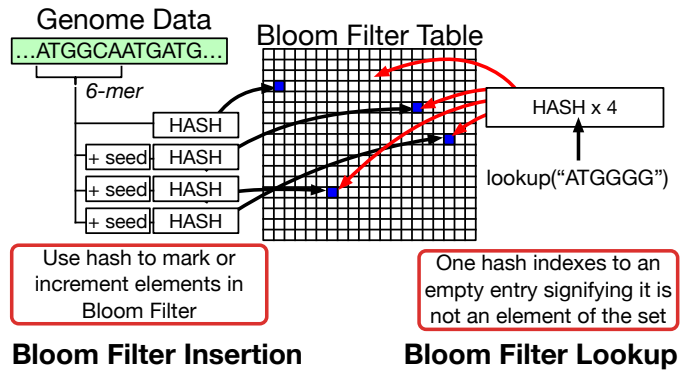


Fig. 1. Bloom Filter access consists of hashing a given input several times and using the hashes as indices into the filter. For insertion, the entries in the table are marked 1 (or incremented for counting Bloom Filters), and for membership, it is only necessary for each entry to be greater than 0. While false positives are possible, they can be made exceedingly rare by increasing the size of the table and increasing the quality of the hash.

### A. Bloom Filters

Bloom filters consist of a large bit array, set initially to zeros. To insert an item into the filter, the item is hashed by multiple independent hash functions, with each resulting hash value serving as an index into the bit array. Each corresponding array entry is set to one. The presence of an item within the filter is signified by all of its hash-indexed positions containing ones. False positives are rare but possible. Given a Bloom filter with  $k$  independent hash functions,  $n$  items added to the filter, and  $m$  addresses in the bit array, its false positive rate  $p$  can be represented by:

$$p = (1 - e^{-k \frac{n}{m}})^k$$

This process is shown in Figure 1. This data structure has been accelerated by FPGAs in prior work [8], [10] by using high-performance Hybrid Memory-Cube memories and by grouping nearby table updates to improve memory efficiency.

However, the standard Bloom filter does come with drawbacks - items cannot be removed from or counted in the set. To avoid these limitations, we use a counting Bloom Filter [6], which, instead of storing a bit at each index, stores a counter. Insert operations increment the counters at all hash-indexed locations, while deletions decrement them. The tradeoff for

these additional properties is the higher memory requirements to store the table. Increasing the distance between all of the table elements can decrease memory performance.

### B. Contributions

We first characterize the workload of bulk Bloom filter insertion on current platforms. Identifying key bottlenecks can determine the applicability of FPGA acceleration. Our analysis reveals that this workload exhibits complementary performance characteristics, favoring a hybrid CPU-FPGA approach. While hash computation benefits from the parallel processing capabilities of FPGAs, the irregular memory access patterns of table updates are better suited to CPU memory hierarchies. Our proposed system, CoBloom, partitions tasks between CPU and FPGA to exploit the strengths of each platform. We deploy our architecture on an Amazon AWS F2 instance [1] and demonstrate a significant speed increase over CPU-only execution.

## II. WORKLOAD ANALYSIS

For this work, we consider two distinct stages of a Bloom filter. First is the hash stage, where each K-mer is converted into multiple hash values using a fixed set of hash functions. Second is the table-update phase, where the table is accessed and incremented using each of the hashes.

### A. Hashing

The goal of the hashing stage is to generate  $k$  unique and uniformly distributed addresses per key. In Bloom filters, the hashing stage must strike a balance between speed and good distribution: while cryptographic hashes offer excellent distribution, they are too computationally expensive to implement. Practical designs utilize lightweight, non-cryptographic hashes with reduced computational overhead. Based on previous research [5], [13], [15], we chose the 32-bit Murmur3 hash [2] for our analysis due to its low collision rate and proven potential for hardware acceleration.

Hash functions achieve a nearly-random distribution of outputs by relying on a combination of bitwise and arithmetic operations. While CPUs have multiple ALUs and SIMD vector units for performing these calculations, they process operands sequentially, which limits their throughput. Conversely, FPGAs are exceptionally well-suited for parallelization and deep compute pipelines, a paradigm that aligns closely with our hash function workload. Dedicated pipelined hashing units can accept new inputs each clock cycle. This allows them to achieve a significantly higher throughput than CPU-based implementations.

We profiled a Bloom filter insertion benchmark and found that computing hashes accounted for 81% of the overall runtime. This benchmark used a typical workload [4]: K-mers of length 8 bytes with four hashes per K-mer. The other 19% was taken up by memory access to update the Bloom filter table. This distribution indicates that computation speed is the limiting factor in this case, suggesting that FPGA acceleration would result in a significant increase in overall throughput.

### B. Memory

The second fundamental problem with accelerating a counting Bloom filter involves memory accesses associated with incrementing table entries. As discussed previously, Murmur3 was chosen in part due to its low chance of collisions. The resulting memory access pattern is extremely sparse, resulting in poor performance of the Bloom filters' average read and write times in most memory systems due to a lack of spatial or temporal locality. While hashing throughput increases linearly with the number of cores, random read-write performance does not. So then, given enough CPU cores dedicated to hashing, this begins to pose as the primary bottleneck of the workload.

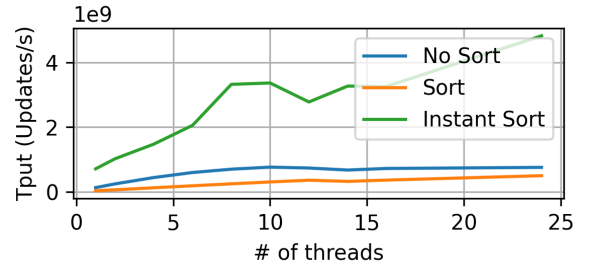


Fig. 2. Sorting the list of hash inserts increases the memory locality of the insertion process. Given a pre-sorted list (Instant Sort), this results in orders-of-magnitude speedup over the baseline random access. However, the overhead of sorting is high, resulting in overall worse performance (Sort).

Previous implementations of Bloom filter accelerators opt to perform both the hash stage and the memory access stage within the FPGA. This approach avoids unnecessary intermediate memory movements, as the hash results can be immediately used to index the table and then discarded without being committed to DRAM. However, the poor cache locality of random table accesses creates memory bandwidth bottlenecks that limit overall system performance.

This is a problem that multiple works have attempted to solve. BunchBloomer [8] performs a hardware-based radix sort on all hash results to gain more spatial locality within table accesses. A second approach utilizes 3D-stacked memory to allow for a much higher bandwidth [10].

We evaluated the impact of sorting by isolating the table update components of the Bloom filter and considering the impact of sorting on the input hashes. We compare a baseline table update with no sorting, a second that performs a sort at the beginning of its execution, and a final one that is provided pre-sorted hashes to represent an upper bound on the effectiveness of sorting. The results are shown in Figure 2. Results demonstrate that while sorting substantially improves memory access patterns and theoretical throughput, there is a significant time penalty associated with sorting the input, reducing the throughput below the baseline without sorting.

Bunchbloomer [8] circumvents many of the costs of directly sorting through two methods. First, they pass their hashes directly to the first stage of their radix sorting unit, eliding a memory access. And second, the radix buckets are very wide

8KB windows. The authors bunch together all of the hash accesses that will hit within an 8KB window. For their single-bit Bloom filters, this corresponds to 64K-entry windows that can effectively batch multiple accesses. Although expensive, maintaining this state is feasible within the resources of an FPGA.

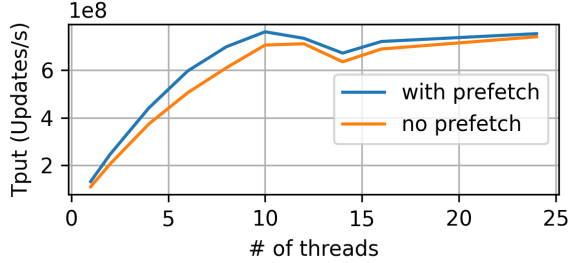


Fig. 3. Because the Bloom Filter table is significantly larger than CPU caches, every access is likely to be a capacity miss. We can hide some of the DRAM access latency then by prefetching the address associated with each of the hashes and actually updating the table in a future iteration. This optimization increases the throughput of table updates by 11% on average.

However, as we are targeting counting Bloom Filters with 1-byte counters, each bucket only corresponds to an 8K entry window. This reduction in window size results in an  $8\times$  decrease in batching effectiveness, limiting its applicability to our workload.

Given the size of a Bloom filter that utilizes the whole 32-bit address space from the hash function, each access is nearly guaranteed to be a capacity miss. CPUs have several qualities that make them easier to optimize for these patterns. CPU architectures operate at significantly higher clock rates, have advanced mechanisms for servicing multiple in-flight DRAM accesses, and can mask DRAM latency through prefetching. As an example of these optimizations, we used prefetching to fetch future hashes and the table entries that they point to. We evaluated the impact of prefetching and found that it improved performance by an average of 11%, shown in Figure 3.

### III. IMPLEMENTATION

The performance characteristics in this analysis motivate our hybrid CPU-FPGA approach. Our proposed design, CoBloom, decouples hashing from table updates, allowing us to benefit from an efficient, high-throughput hashing accelerator while utilizing the high clock rate and efficient out-of-order execution of the CPU to handle long-latency table updates.

#### A. System Design

We used the open-source Beethoven [9] framework and Chisel HDL [3] to design our accelerator system. The Murmur3 accelerator core defines a command interface using Beethoven that the host system can use to communicate with the core. Beethoven transforms this Chisel interface into a C++ library that enables the host system to communicate directly with the accelerator core from the testbench. Our Murmur3 core also declares memory interfaces to DRAM for reading

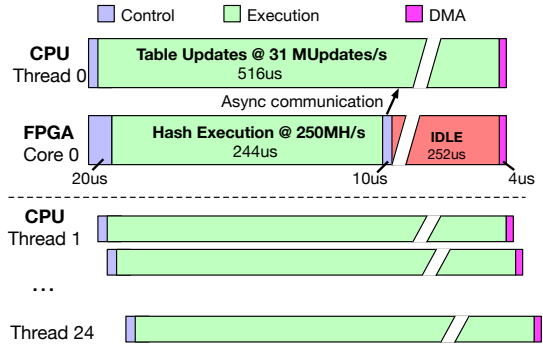


Fig. 4. The FPGA and CPU can work asynchronously from each other to provide the greatest speedup. While the FPGA is hashing, the CPU can perform table updates, overlapping their execution. Because the FPGA is significantly faster than the CPU, there is idle time on the FPGA; however, this allows other FPGA hashing cores to run in parallel without contention. Each CPU thread in our architecture has its own hashing core in the FPGA.

inputs and writing out the hashes. We then use Beethoven to elaborate a many-core system, automatically generating the memory and control interconnects as well as floorplanning.

Each CoBloom accelerator core consists of multiple parallel Murmur3 computation units, each fully pipelined and capable of accepting new K-mer inputs every clock cycle. Beethoven reader and writer streams optimize AXI transactions and act as the interface to read genomic data from DRAM and write hashes back when they are ready. Upon completion of each processing block, the accelerator cores signal the host system, triggering transfer of hash results from FPGA memory to CPU memory for subsequent table update operations.

We deployed a 250MHz FPGA design with 24 Murmur3 hash cores on the VU49P FPGA on AWS EC2 F2 instances. We chose 24 cores because our chosen instance type has 24 CPU cores. The resource utilization numbers are provided in Figure 6, and our design operated at 5W.

We could have fit far more hashing units on the FPGA, but this would be a waste of resources for several reasons. First, the hashing throughput of these cores already fully saturates the device’s DDR bandwidth, resulting in diminishing returns from any additional cores. Additionally, adding more cores increases the static power of the design, reducing efficiency without any tangible benefits.

### IV. RESULTS

We run the Bloom filter insertion on the AWS F2 FPGA using CoBloom<sup>1</sup>. The FPGA features a single channel of DDR4 memory with an ideal bandwidth of 14.9 GB/s. Each Murmur3 FPGA core has a peak memory bandwidth of 3.7 GB/s, indicating that we should achieve peak performance of approximately 3.7 GHash/s. However, because we need additional write bandwidth to manage the DMA, we see diminishing returns as the number of cores grows.

<sup>1</sup>While the F2 instances plan to provide full DMA support, it is currently a work in progress. Therefore, we analyze our work assuming the DMA bandwidth of the prior generation (F1 instances)  $\sim 6$  GB/s and with realistic PCIe communication latencies,  $\sim 2$   $\mu$ s/message.

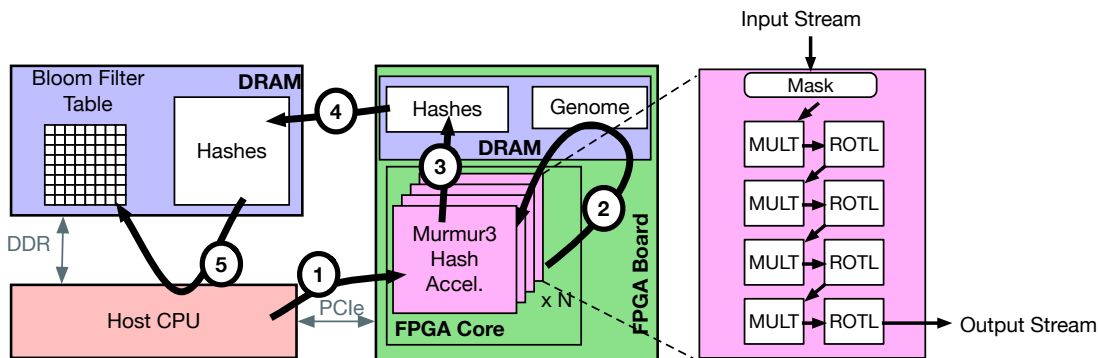


Fig. 5. The block diagram for our BloomFilter System. Genome data is stored inside the FPGA memory, and the Bloom filter table is stored in CPU memory. 1. The host CPU offloads hash function execution to the FPGA. 2. Each accelerator core contains an input and an output stream for inputting the genomic data and outputting the hashes to FPGA-accessible DRAM. The Murmur3 core itself consists of a pipeline of multiplication, left rotation, shift, and XOR operations. 3. As the inputs are hashed, they are streamed out to the FPGA DRAM. 4. After completion, the host CPU DMA's the hashes back to its memory. 5. Finally, the host CPU uses these hashes to update the Bloom Filter Table.

Resource	Total Util.	Per-Core Util.
LUT	231K (17%)	8.4K
Registers	108K (4%)	4.7K
URAM	408 (42.5%)	18.6
DSP	1728 (19%)	72

Fig. 6. Resource utilization of CoBloom. While we did not fully utilize on-chip resources, there is also no need to unnecessarily incur power overheads when the critical path is still random memory access on the host CPU.

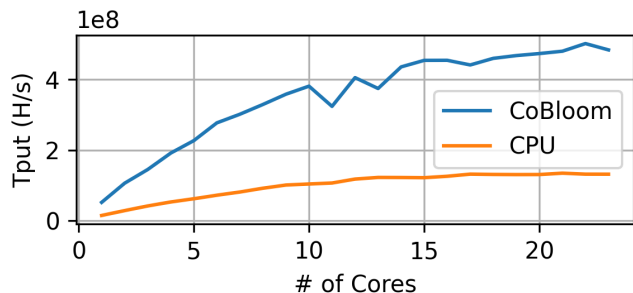


Fig. 7. CoBloom achieves a peak of 500 MH/s using our coprocessing approach, 3.72 $\times$  faster than CPU-only.

To verify the functionality and performance of CoBloom, we run the full Bloom filter benchmark using an E. Coli genome [11]. CoBloom achieves a 3.72 $\times$  speedup over the software baseline (CPU-only) shown in Figure 7. Given that hashing accounts for 81% of the total runtime on the CPU, if hashing were infinitely fast, our ideal speedup would be closer to 5.2 $\times$ . In practice, we do not achieve this due to the additional overheads of DMA and control latencies between the FPGA and the CPU.

There are two primary previously-proposed FPGA accelerators to compare against. The first is Bunchbloomer [8], which, in addition to hashing, also sorts and buckets the updates. This bucketing places updates with high spatial locality together, allowing the accelerator to update them all together using a

single DRAM transaction. High locality, however, comes at the cost of increased memory pressure due to reading and writing data back and forth from the sorter. The second is a work that optimizes bloom filter operation for utilization of Hybrid Memory Cube (HMC) technology [10]. These works achieve 372 MHash/s and 220 MHash/s, respectively. The first work primarily targets standard bloom filters and notes that for counting bloom filters, it achieves an estimated 250 MHash/s. CoBloom achieves 500 MHash/s, a 2-2.2 $\times$  speedup over prior work.

## V. CONCLUSIONS AND FUTURE WORK

We characterize the Bloom Filter workload for genomics applications and show that CPU-based hashing is a critical performance limiter. Accordingly, we developed CoBloom, a CPU+FPGA coprocessor solution designed to accelerate Bloom Filters. CoBloom accelerates hashing on the FPGA and then uses the high-performance memory architecture already present in modern CPUs to target table lookups. We designed our accelerator and deployed it on AWS F2 instances, demonstrating a 3.72 $\times$  speedup over a CPU-only approach.

While our current design is memory bandwidth throttled on both FPGA and CPU, modern CPUs are equipped with High Bandwidth Memory (HBM), providing 100s+ GB/s of memory bandwidth. Our future work will investigate how to leverage this bandwidth and prior work on memory-centric Bloom Filter optimization to further accelerate Bloom Filters.

## VI. ACKNOWLEDGEMENT

This work is in part funded by the National Science Foundation CAREER award CCF-2045974.

## REFERENCES

- [1] Amazon. Amazon ec2 f2 instances. <https://aws.amazon.com/ec2/instance-types/f2/>.
- [2] Austin Appleby. Smlasher. <https://github.com/aappleby/smlasher>.
- [3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzyniec, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th annual design automation conference*, pages 1216–1225, 2012.

- [4] Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 06 2013.
- [5] Søren Dahlgaard, Mathias Knudsen, and Mikkel Thorup. Practical hash functions for similarity estimation and dimensionality reduction. *Advances in Neural Information Processing Systems*, 30, 2017.
- [6] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.
- [7] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Jae W. Lee, David Bruns-Smith, Brendan Sweeney, Krste Asanovic, Young H. Oh, and Lisa Wu Wills. Accelerating genomic data analytics with composable hardware acceleration framework. *IEEE Micro*, 41(3):42–49, 2021.
- [8] Seongyoung Kang, Tarun Sai Ganesh Nerella, Shashank Uppoor, and Sang-Woo Jun. Bunchbloomer: Cost-effective bloom filter accelerator for genomics applications. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 9–16, 2022.
- [9] Chris Kjellqvist, Brendan Peercy, Alvin R Lebeck, and Lisa Wu Wills. Beethoven: A heterogeneous multi-core accelerator system composer. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 297–308. IEEE, 2025.
- [10] Nathaniel Mcvicar, Chih-Ching Lin, and Scott Hauck. K-mer counting using bloom filters with an fpga-attached hmc. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 203–210, 2017.
- [11] NCBI. Escherichia coli j53 genome assembly, contig unitig\_1\_prcs59. GenBank accession no. CABFNM010000002.1, 2019. Accessed: 2025-08-28.
- [12] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.
- [13] Hüseyin Aydın Seymen and Müştak Erhan Yalçın. Design and implementation of a lightweight bloom filter accelerator for iot applications. In *2023 14th International Conference on Electrical and Electronics Engineering (ELECO)*, pages 1–5, 2023.
- [14] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.
- [15] Fumito Yamaguchi and Hiroaki Nishi. Hardware-based hash functions for network applications. In *2013 19th IEEE International Conference on Networks (ICON)*, pages 1–6, 2013.