Design and Evaluation of a GPU-Accelerated Cuckoo Filter

Tim Dortmann

February 5, 2026

Version: CleanThesis 0.4.1 - JGU patches

Institute of
Computer Science

JG|U

Johannes Gutenberg University Mainz
FB08
Institute of Computer Science
High Performance Computing Group

Bachelor thesis

Design and Evaluation of a GPU-Accelerated Cuckoo Filter

Tim Dortmann

1. Reviewer Prof. Dr. Bertil Schmidt
Institute of Computer Science
High Performance Computing Group
Johannes Gutenberg University Mainz

2. Reviewer Prof. Dr. Felix Schuhknecht
Institute of Computer Science
Information Systems Group
Johannes Gutenberg University Mainz

Supervisors Markus Vieth and Prof. Dr. Bertil Schmidt

February 5, 2026

Tim Dortmann
Design and Evaluation of a GPU-Accelerated Cuckoo Filter
Bachelor thesis, February 5, 2026
Reviewers: Prof. Dr. Bertil Schmidt and Prof. Dr. Felix Schuhknecht
Supervisors: Markus Vieth and Prof. Dr. Bertil Schmidt

Johannes Gutenberg University Mainz
High Performance Computing Group
Institute of Computer Science
FB08
Staudingerweg 9
55128 Mainz

Abstract

Approximate Membership Query (AMQ) data structures are essential for high-throughput systems, yet the industry-standard Bloom filter lacks support for item deletion. This thesis presents the design and implementation of a high-performance, GPU-accelerated Cuckoo filter, delivered as a robust header-only library. By utilizing a parallelized eviction algorithm based on atomic operations, the implementation supports massive parallelism without explicit locking.

A comprehensive evaluation across GDDR7 and HBM3 architectures reveals that the Cuckoo filter significantly outperforms state-of-the-art dynamic alternatives. It achieves insertion and deletion throughputs orders of magnitude higher than the GPU Quotient Filter (GQF) and the Two-Choice Filter (TCF). Crucially, the results demonstrate superior architectural scalability: while TCF and GQF performance stagnates on high-bandwidth hardware due to internal latency bottlenecks, the Cuckoo filter scales linearly with global memory bandwidth. This characteristic positions the Cuckoo filter as the better solution for next-generation, bandwidth-rich GPU architectures.

Abstract (German)

AMQ-Datenstrukturen sind für Hochdurchsatzsysteme essenziell, doch dem verbreiteten Bloom-Filter fehlt die Löschunterstützung. Diese Arbeit präsentiert einen hochperformanten, GPU-beschleunigten Cuckoo-Filter als robuste Header-only Library. Effiziente Algorithmen auf Basis atomarer Operationen ermöglichen dabei massive Parallelität ohne explizites Locking.

Evaluierungen auf GDDR7- und HBM3-Systemen zeigen, dass der Filter dynamische Alternativen deutlich übertrifft. Er erzielt Einfüge- und Löschraten, die um Größenordnungen über denen von GPU Quotient (GQF) und Two-Choice Filtern (TCF) liegen. Entscheidend ist die Skalierbarkeit: Während die Leistung der Konkurrenz auf Hardware mit hoher Bandbreite latenzbedingt stagniert, skaliert der Cuckoo-Filter linear mit der globalen Speicherbandbreite. Dies macht ihn zur führenden Lösung für zukünftige GPU-Architekturen.

Contents

1 Introduction

1.1 Motivation

1.1.1 Problem Statement

In the era of big data, the ability to perform high-speed set membership queries is a fundamental requirement for applications ranging from network traffic analysis [5] to large-scale distributed systems [33]. Determining whether an element belongs to a (often rather large) set is a frequent and performance-critical operation. While exact data structures provide definitive answers, their memory footprint and computational overhead make them impractical for massive datasets. This has led to the widespread adoption of probabilistic data structures, which trade a small, manageable false-positive probability for significant gains in space and time efficiency.

For years, the Bloom filter has been the standard probabilistic data structure for approximate set membership. Its primary limitation, however, is the inability to delete elements, making it unsuitable for dynamic datasets. While there are variations that do support deletion, they incur prohibitive space overheads that often negate their practical viability. Due to this the Cuckoo filter has emerged as a powerful alternative, offering native support for deletions and often superior space efficiency, particularly at low false-positive rates.

Despite these advantages, the performance of Cuckoo filters on traditional CPU architectures can quickly become a bottleneck in high-throughput environments. The sequential nature of the insertion algorithm, which may involve displacing multiple existing items, severely limits scalability on serial processors.

This performance gap motivates the exploration of massively parallel hardware. However, porting a Cuckoo filter to a GPU is not a straightforward translation, as the algorithm's reliance on sequential eviction chains, random memory accesses, and conditional logic conflicts with the GPU's desire for massive parallelism and structured, contiguous memory access patterns. This thesis addresses these architectural mismatches by designing, implementing, and evaluating a GPU-accelerated Cuckoo filter. The core objective is to leverage the massive parallelism of modern GPUs to handle insertions, lookups, and deletions, thereby achieving a significant performance leap over existing CPU-based implementations.

1.1.2 Use Cases

The demand for high-speed, dynamic set membership testing shows up across numerous domains. While CPU-based dynamic filters exist, many modern applications generate data at a rate that creates a performance bottleneck, motivating the need for a massively parallel, GPU-accelerated solution. Some of the areas that could benefit from such a filter are as follows:

- **High-Speed Network Security**: Threat intelligence feeds for malicious IPs, URLs, and malware signatures are updated continuously, requiring a filter that supports both insertions and deletions. In high-speed networks (100Gbps+), a CPU can be overwhelmed by the sheer volume of packets per second. A GPU-accelerated filter can process large batches of packet headers or identifiers in parallel, enabling line-rate inspection against dynamic blacklists in a way that is simply infeasible for CPU-based solutions [29, 15].
- **High-Throughput Caching Systems**: Caches in Content Delivery Networks (CDNs) and HTTP reverse proxies experience constant churn as items are added and evicted. A dynamic filter is essential to prevent expensive disk or network lookups for non-existent objects. When the request rate is in the millions per second, the CPU can offload these cache-presence checks to a GPU, processing them in large batches to free up cycles for handling the actual data I/O [7, 10].
- **Large-Scale Databases and Distributed Systems**: Database systems often use filters to avoid expensive disk lookups for non-existent keys. In a distributed setting, a GPU-accelerated filter could serve as a high-performance shared resource that tracks the existence of records across multiple nodes, reducing network latency and improving overall query performance. [20, 22]
- **Bioinformatics and Computational Biology**: Genomics and proteomics research involves searching for patterns or sequences within massive biological datasets. A GPU-accelerated Cuckoo filter could be used to rapidly pre-screen for the presence of specific k-mers or genetic markers before launching more computationally intensive analyses, significantly speeding up the research pipeline. [13]

1.2 Contributions

This thesis presents a comprehensive study on accelerating probabilistic data structures using GPUs. The main contributions of this work are as follows:

- **High-Performance CUDA Library**: The design and implementation of a parallel Cuckoo filter supporting insertion, lookup, and deletion are presented. The source code is available as an open-source header-only library.[1]

- **Advanced Optimization Techniques**: Several optimization strategies are explored and evaluated to maximize occupancy and memory bandwidth. These include a sorted-insertion algorithm to improve memory locality, a modified eviction strategy designed to reduce the number of random memory accesses at high load factors and the ability to swap out the original XOR-based partial-key cuckoo hashing for alternatives.
- **System-Level Integration Extensions**: Moving beyond a simple filter, two extensions are created to facilitate real-world adoption. First, an Inter-Process Communication (IPC) wrapper is developed to enable zero-copy sharing of the filter between processes. Second, a multi-GPU implementation is provided that transparently partitions data across multiple devices, allowing the filter to scale beyond the memory limits of a single card.
- **Comprehensive Evaluation**: A rigorous analysis is conducted comparing the GPU Cuckoo filter against CPU baselines and other GPU-accelerated filters. The results demonstrate that the implementation achieves competitive throughput with the Blocked Bloom filter while far surpassing all other tested dynamic filters.

1.3 Thesis Structure

Chapter 2
This chapter establishes the foundation for the thesis. It goes over the fundamental concepts of approximate membership query structures, tracing the evolution from the classic Bloom filter to the Cuckoo filter. Additionally, it examines alternative modern data structures, such as the Quotient Filter and the Two-Choice Filter, to contextualize the research landscape. Subsequently, the chapter introduces the architectural principles of GPU computing and the CUDA programming model, highlighting the specific hardware constraints that influence parallel algorithm design.

Chapter 3
This chapter details the design and implementation of the high-performance GPU Cuckoo filter library. It documents the parallel algorithms developed for insertion, lookup, and deletion, explaining how lock-free atomic operations are utilized to manage parallel accesses. Furthermore, it describes some advanced optimizations for memory locality and system-level extensions, including an Inter-Process Communication wrapper and a multi-GPU sharding mechanism.

[1] Available at: `https://github.com/tdortman/cuckoo-filter/`

Chapter 4

This chapter presents a comprehensive empirical analysis of the implemented data structure. The filter is benchmarked against state-of-the-art CPU and GPU baselines across diverse hardware architectures (GDDR7 and HBM3). Beyond raw throughput, the evaluation scrutinizes architectural scaling behaviours, analysing the impact of the memory hierarchy (L2 vs. DRAM), hardware utilization, and cache efficiency. Furthermore, it quantifies the impact of specific algorithmic optimizations, including eviction policies, bucket sizing, and sorted insertion, and validates the system's real-world applicability through multi-GPU scalability tests and a genomic k-mer indexing benchmark.

Chapter 5

The final chapter highlights the key findings of the research, summarizing the contributions made to the field of parallel probabilistic data structures. It critically assesses the limitations of the current implementation and outlines potential avenues for future research and optimization.

Background and Related Work

2

Having established the motivation for designing a GPU-accelerated Cuckoo filter, this chapter provides the necessary background information from its two relevant domains: probabilistic data structures and parallel computing. The discussion first traces the algorithmic lineage of the Cuckoo filter, beginning with the Bloom filter, progressing to the Cuckoo hashing scheme, and ending in the Cuckoo filter itself. Subsequently, the focus shifts to hardware and software with an introduction to the GPU architecture, the CUDA programming model, and the key performance considerations essential for developing efficient parallel algorithms. A firm grasp of both these areas is crucial for understanding the design choices and implementation challenges addressed in the remainder of this thesis.

2.1 Probabilistic Data Structures

At the heart of this thesis is the need for a dynamic and efficient data structure for approximate set membership. This section reviews the key structures that form the basis of the work presented. The analysis begins with the classic Bloom filter and its locality-optimized variant, which serves as an important performance baseline. Following this is an explanation of the mechanics of Cuckoo hashing, the eviction-based strategy that enables the dynamic properties of the target data structure. Finally, the Cuckoo filter is detailed, showing how it combines these concepts to create a powerful and flexible alternative.

2.1.1 Bloom Filter

Invented in 1970, the Bloom filter [4] has long been the dominant probabilistic data structure for approximate membership query (AMQ) problems. Its operation is based on a simple yet effective concept: a bit array of size m and a set of k independent hash functions. To insert an item, the item is hashed k times, and each resulting hash value is used as an index to set a bit in the array to 1. To query for an item's membership, the item is again hashed k times. If all corresponding bits in the array are 1, the item is considered to possibly be in the set. However, if even one bit is 0, the item is definitively not in the set, as illustrated in Figure 2.1.

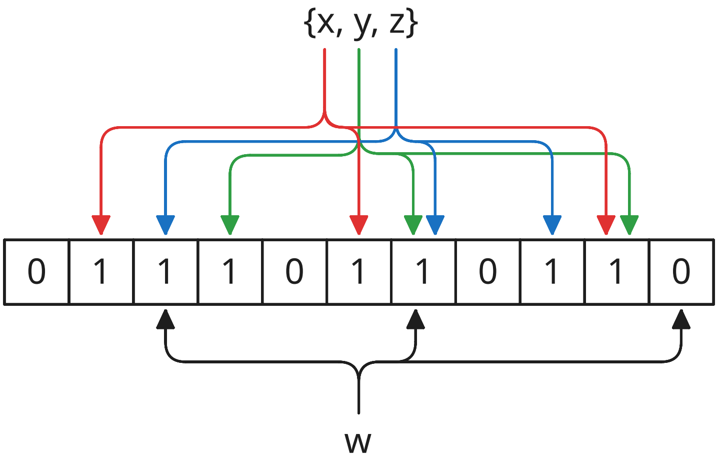


Fig. 2.1: An illustration of a Bloom filter's insertion and lookup mechanism with $m = 11$ slots and $k = 3$ hash functions. The set $\{x, y, z\}$ has been inserted, setting the corresponding bits in the array to 1. A membership query for a new item w returns a definitive "no" (guaranteeing no false negatives) because one of the bits it maps to is 0.

The false-positive rate ϵ of a Bloom filter after inserting n items is approximately:

$$\epsilon \approx \left(1 - e^{-nk/m}\right)^k \tag{2.1}$$

This rate is optimized by choosing the optimal number of hash functions, which for a given n and m is:

$$k = \frac{m}{n} \ln 2 \tag{2.2}$$

Given a target false positive rate ϵ, the number of hash functions used by a space-optimized Bloom filter is given by:

$$k = \log_2(1/\epsilon) \tag{2.3}$$

Such a filter uses $1.44 \log_2(1/\epsilon)$ bits per element.

Despite its widespread use, the classic Bloom filter has several notable disadvantages:

- **No Deletion**: The standard implementation does not support the removal of items, as clearing a bit could inadvertently remove other items that hash to the same location. Variants like the Counting Bloom Filter [10] address this by using counters instead of single bits, but at a significant cost to space efficiency.

- **Linear Complexity**: Lookup and insertion performance is dependent on the number of hash functions, which scales linearly with m. This can lead to performance bottlenecks in high-throughput scenarios.
- **Poor Memory Locality**: The k hash functions produce indices that are typically scattered randomly across the entire bit array. This leads to poor cache performance on CPUs and is particularly detrimental on GPUs, where it prevents efficient memory access.
- **Degrading Performance**: As the filter fills up and more bits are set to 1, the false positive rate steadily increases, eventually converging to a point where all queries yield a positive result.
- **Suboptimal Space Usage**: The optimal space usage for a Bloom filter is approximately 44% higher than the information theoretical lower bound for AMQ data structures. Many modern filters, including the Cuckoo filter, achieve a lower overhead relative to this bound.

2.1.2 Blocked Bloom Filter

To address the issue of poor memory locality, the Blocked Bloom Filter was introduced. As this variant is an important point of comparison in this thesis, the design warrants its own detailed explanation.

Instead of a single bit array, the Blocked Bloom Filter partitions the array into an array of smaller, independent Bloom filters, called blocks. Each block is typically sized to fit within a single CPU cache line (e.g., 64 bytes). The hashing scheme is modified: a single hash function is first used to map an incoming item to a specific block. Then, the standard k hash functions are used to set or check bits only within that selected block, as shown in Figure 2.2.

The primary advantage of this design is a dramatic improvement in memory locality. All k memory accesses for a single item are now confined to a small, contiguous memory region. This is highly beneficial for modern CPU caches and, more importantly for this thesis, is extremely well-suited to the parallel memory access patterns of GPUs. When multiple threads process items that map to the same block, their memory requests can be coalesced into a single transaction, significantly improving memory bandwidth utilization.

However, this performance gain comes at the cost of a higher false positive rate. By partitioning the filter, the Blocked Bloom Filter loses the "averaging" effect of the classic design. If an unlucky distribution of items causes one block to become heavily saturated, its local false positive rate will increase dramatically, and this "hotspot" can dominate the overall false positive rate of the entire structure. Therefore, the Blocked Bloom Filter represents a direct trade-off: sacrificing some statistical efficiency for significantly better performance, making it a highly relevant baseline for evaluating cache-aware and GPU-accelerated data structures.

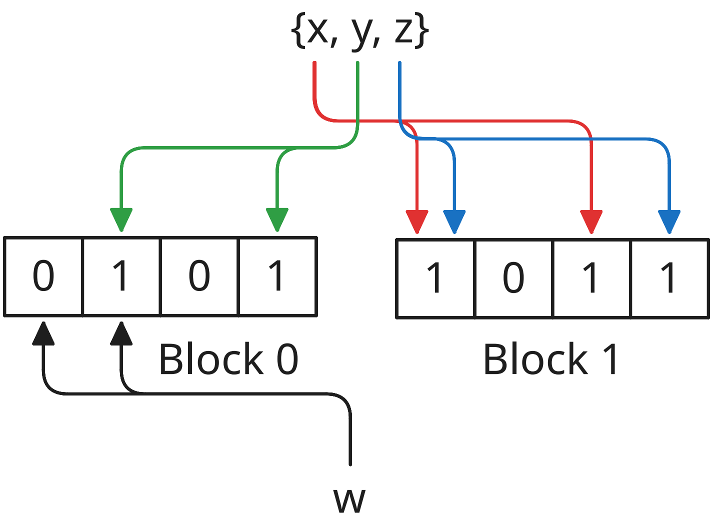


Fig. 2.2: An illustration of a Blocked Bloom Filter. Each item is first mapped to a single block (e.g., item y maps to Block 0, while x and z map to Block 1). The k hash functions then operate only within that selected block, improving memory locality. A query for item w, which maps to Block 0, returns a definitive "no" as one of its target bits is 0.

2.1.3 Cuckoo Hashing

Cuckoo hashing, introduced by Pagh and Rodler in 2004 [27], is a powerful hashing scheme that provides a significant advantage over many others: a worst-case $O(1)$ lookup time. Its name is derived from the cuckoo bird, which is known for laying its eggs in the nests of other birds, often forcing the original occupants out. The scheme's elegance lies in its simplicity compared to previous methods that also offered worst-case constant lookup guarantees [11, 12, 8].

In its original formulation, the scheme uses two independent hash tables, T_1 and T_2, each of size r, and two independent hash functions h_1 and h_2.

The core invariant of Cuckoo hashing is that for any given item x, it is stored in one of two possible locations:

$$T_1[h_1(x) \bmod r] \quad \text{or} \quad T_2[h_2(x) \bmod r]$$

A lookup operation is therefore guaranteed to run in $O(1)$, as it only requires checking these two specific locations.

The insertion process is more involved and follows the distinct "cuckoo" eviction protocol, as illustrated in Figure 2.3. The steps are as follows:

1. For a new item x, its two potential locations $i_1 = h_1(x) \bmod r$ and $i_2 = h_2(x) \bmod r$ are computed.

2. If either $T_1[i_1]$ or $T_2[i_2]$ is empty, the item is placed there, and the insertion is complete.
3. If both slots are occupied, an existing item (say y) is evicted from one of the location (e.g. i_1) and x is placed there.
4. The evicted item y is then reinserted into its alternate location. If y was in T_1, it now attempts to move to $T_2[i_2]$. This move may, in turn, cause another eviction, leading to a chain of evictions.
5. To prevent infinite loops, a maximum number of evictions is set. If this limit is reached, the table is considered full, and all items must be rehashed with a new pair of hash functions, potentially into larger tables.

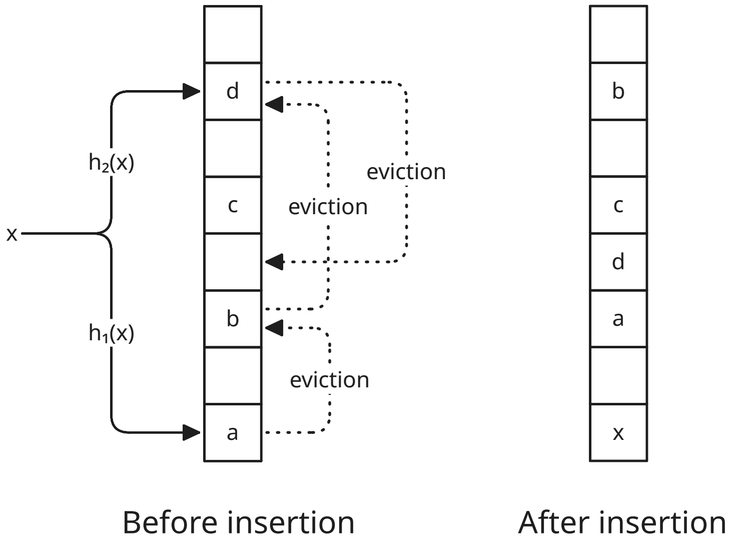


Fig. 2.3: Illustration of a Cuckoo hashing insertion in a single-table variant. The incoming item x finds both of its candidate slots occupied. It evicts item a, triggering a cascading eviction chain until item d is moved to a free slot, resulting in a new stable arrangement of items in the table.

The success of this probabilistic scheme hinges on the load factor, the theory guarantees that if the tables are kept sufficiently sparse. Specifically, if r is greater than $(1+\epsilon)n$, where n is the number of items, the probability of an insertion failing and requiring a rehash is very low [26]. This effectively means the total load factor across both tables should be kept below 50%.

A common and important variant is to use a single table of size $m = 2r$. This is more space-efficient and is the model that more closely resembles a Cuckoo filter's implementation. It also introduces the idea of deriving an alternate location from a current one without storing any extra state.

Pagh and Rodler describe a trick, attributed to John Tromp, that achieves this. The key insight is to redefine the two potential locations for an item x. Instead of simply being $h_1(x) \bmod m$ and $h_2(x) \bmod m$, the locations are defined as:

- $i_1 = h_1(x) \bmod m$
- $i_2 = (h_2(x) - h_1(x)) \bmod m$

With this specific construction, a symmetric function can be used to jump between the two locations. If an item x is currently at a location i, its alternate location i' can be calculated using the mapping

$$i' = (h_2(x) - i) \bmod m \tag{2.4}$$

This works because the transformation is its own inverse for that specific pair of locations. Applying it to i_1 yields i_2, and applying it again to i_2 yields i_1. This concept of a state-free and symmetric mapping is the direct precursor to the partial-key hashing scheme used in Cuckoo filters.

2.1.4 Cuckoo Filter

The Cuckoo filter, introduced by Fan et al. [9], is a probabilistic data structure that adapts the principles of Cuckoo hashing to provide a highly space-efficient and performant alternative to the Bloom filter, most notably by supporting the deletion of items. It improves upon standard Cuckoo hashing in multiple ways.

First, instead of storing entire items, a Cuckoo filter only stores a small fingerprint for each item, which is a fixed-size sequence of bits derived from the item's hash. This significantly reduces the memory footprint. A direct consequence, however, is that traditional rehashing is not an option, as the original items are not available to be re-inserted. Therefore, the filter's size and parameters are typically static once it is created.

Second, it replaces the standard Cuckoo hashing scheme with partial-key cuckoo hashing. This clever technique establishes a symmetric dependency between the two potential bucket locations and the item's fingerprint. A single hash function is used to compute an initial location i_1 and a fingerprint fp. The alternate location i_2 is then derived using the XOR operation:

$$i_2 = i_1 \oplus \text{hash}(fp) \tag{2.5}$$

Thanks to the symmetric nature of XOR, this relationship works both ways, such that $i_1 = i_2 \oplus \text{hash}(fp)$. This allows an evicted fingerprint to calculate its alternate location from its current position and its own fingerprint alone, an important feature since the original item is not stored.

Design and Performance Characteristics

The design of the Cuckoo filter leads to several practical advantages over Bloom filters and their variants.

- **High Space Utilization**: By allowing items to be relocated during insertion, Cuckoo filters can achieve very high load factors. The original paper demonstrates that with a bucket size of 4, the filter can consistently reach an occupancy of over 95% [9]. The GPU implementation in this thesis generally prefers slightly larger bucket sizes for performance reasons, thus consistently reaching a load factor of over 99%.

- **Space Efficiency vs. False Positive Rate**: The false positive rate ϵ of a Cuckoo filter is directly tied to the fingerprint size f and the bucket size b. Because a lookup must check up to $2b$ fingerprints in the worst case, the false positive rate can be approximated by:

  $$\epsilon \approx 2b/2^f \tag{2.6}$$

  The actual amortized space cost per item, C, is the fingerprint size f divided by the filter's load factor α, since the cost of the unoccupied slots must be distributed among the stored items. The minimal fingerprint size required to achieve a target rate ϵ is approximately

  $$f \geq \log_2(2b/\epsilon) = \log_2(1/\epsilon) + \log_2(2b) \tag{2.7}$$

  By substituting this minimal required value for f into the cost definition, an upper bound for the amortized space cost can be established:

  $$C \leq \frac{\log_2(1/\epsilon) + \log_2(2b)}{\alpha} \tag{2.8}$$

  This equation is important as it decomposes the cost into the information-theoretic lower bound ($\log_2(1/\epsilon)$), an overhead term related to the bucket size ($\log_2(2b)$), and an efficiency penalty from the load factor ($1/\alpha$). It clearly reveals the central trade-off: a larger bucket size b improves the load factor α but also increases the overhead term, requiring careful tuning to optimize space. For a target $\epsilon < 3\%$, the authors show that a well-configured Cuckoo filter is more space-efficient than a space-optimized Bloom filter [9].

- **Performance**: A key performance advantage is that any lookup, positive or negative, always reads a fixed number of buckets, resulting in (at most) two cache line misses. This is a significant improvement over standard Bloom filters, where the number of memory probes k scales with the desired false positive rate and can be much larger than two. This theoretical efficiency translates to exceptional practical performance.

As demonstrated later in the evaluation in Section 4.2, the GPU Cuckoo filter effectively matches the throughput of the highly cache-optimized Blocked Bloom Filter in cache-resident scenarios and even surpasses it for positive lookups on high-bandwidth architectures, while simultaneously outperforming all other tested dynamic filters by orders of magnitude.

2.2 GPU Computing

This section introduces the relevant concepts of Graphics Processing Unit (GPU) architectures and the CUDA programming model. An understanding of these principles is essential for contextualizing the design decisions and performance optimizations discussed in the subsequent implementation of the parallel Cuckoo filter. While this thesis targets NVIDIA GPUs using the CUDA framework, the core concepts of massively parallel processing are applicable to other GPU architectures and, to a large extent, modern multicore CPUs as well.

2.2.1 The GPU as a Parallel Processor

At its core, a GPU is a specialized processor designed for massive data parallelism. Originally developed to accelerate the computationally intensive task of rendering graphics, its architecture has evolved to become highly effective for general-purpose computing.

The primary architectural difference between a CPU and a GPU lies in their design philosophies. A CPU is optimized for low-latency execution of a single or a few sequential instruction streams (threads). It dedicates a significant portion of its silicon to sophisticated flow control and large data caches to minimize the execution time of a single task. In contrast, a GPU is designed for high-throughput computing. It makes up for slower single-thread performance by executing thousands of threads in parallel, dedicating far more of its transistors to data processing rather than to data caching and flow control. Figure 2.4 highlights this difference. For example, a modern NVIDIA Blackwell architecture GPU can feature over 24,000 cores. This design results in much higher instruction throughput and memory bandwidth, which is ideal for problems that can be broken down into many independent sub-problems.

It is important to note that not all parts of a program can be effectively parallelized. The portions that can are typically isolated and rewritten as GPU kernels, which are functions compiled separately for the GPU's instruction set. A typical workflow for a GPU-accelerated program looks as follows:

1. Allocate memory on the device (GPU).

2. Copy input data from host memory (CPU's RAM) to device memory.
3. Launch a kernel on the host, which is executed in parallel by many threads on the device.
4. Copy the results from device memory back to host memory.
5. Deallocate memory on the device.

This data transfer overhead means that GPUs are most effective for problems where the computational work significantly outweighs the amount of data that needs to be transferred.

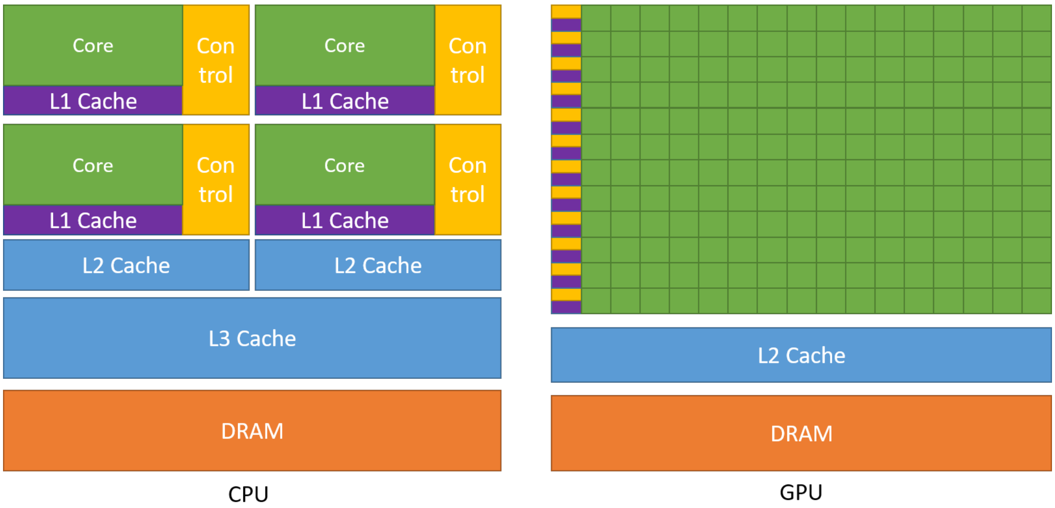


Fig. 2.4: The GPU Devotes More Transistors to Data Processing [@25]

2.2.2 CUDA Programming Model

To manage the massive parallelism of the hardware, NVIDIA developed CUDA (Compute Unified Device Architecture), a parallel computing platform and programming model. CUDA provides a set of extensions to the C++ language that allow developers to write host and device code within the same environment, simplifying the development process. It also abstracts the hardware into a logical hierarchy that is more manageable for the programmer.

The basic unit of execution in CUDA is a thread. When a kernel is launched, it is executed by a vast number of these threads on the GPU. They are organized into a three-dimensional hierarchy:

- **Threads**: The smallest execution unit. Each thread executes the same kernel code but usually operates on different data, identified by its unique coordinates within a block.

- **Blocks**: Each block can contain up to 1024 threads, which can work together by sharing data through fast on-chip shared memory and synchronizing their progress.
- **Grid**: Blocks are organized into a grid. All blocks in a grid execute the same kernel. Blocks are assumed to execute independently and in any order, and there is no guaranteed synchronization mechanism between them during a single kernel launch. [1]

This hierarchical grid structure makes it easy to map threads to data. For example, when processing an image, one might assign a single thread to each pixel, group threads for a tile of the image into a block, and have the entire grid of blocks process the full image.

2.2.3 Hardware Architecture

This logical hierarchy maps onto the physical hardware of the GPU. A GPU is composed of multiple Streaming Multiprocessors (SMs). A global scheduler assigns thread blocks to available SMs for execution, and a key challenge in GPU programming is keeping these SMs saturated with work.

Each SM is a powerful parallel processor in its own right, containing several components (there are more components, but they are omitted here as they are largely irrelevant to the thesis):

- **CUDA Cores**: The basic arithmetic logic units (ALUs) that perform integer and floating-point calculations.
- **Warp Schedulers**: Decides which group of threads gets to execute on each clock cycle.
- **Register File**: The fastest storage on the GPU, holding thread-specific data and intermediate results. While each thread has its own unlimited logical set of registers, the physical register file is a limited resource shared across all active threads on an SM.
- **Shared Memory/L1 Cache**: A low-latency memory space used for user-managed data sharing within a thread block.
- **Load/Store Units**: Manage the movement of data between memory spaces.

Threads are not only grouped into blocks but are also managed by the SM in groups of 32 called *warps*. A warp is the fundamental unit of scheduling on the GPU. All 32 threads in a warp execute in a Single-Instruction-Multiple-Thread (SIMT) fashion, meaning they run in lockstep and execute the same instruction at the same time. This SIMT execution model is the root cause of the most important performance considerations in GPU programming.

[1] Hopper GPUs and newer support Thread Block Clusters, which allows this to an extent

2.2.4 Memory Hierarchy

Performance in GPU programming is directly linked to memory access patterns. A deep understanding of the memory hierarchy is crucial for writing efficient code, as the primary goal of many CUDA optimizations is to maximize the use of fast memory and minimize traffic to slower memory.

- **Registers**: The fastest memory on the GPU. Each thread has its own private registers for its local variables.
- **Shared Memory**: A small, low-latency, user-programmable memory space shared by all threads within a single block. It is essential for intra-block communication and is primarily used as a user-managed cache to avoid redundant reads from the much slower global memory.
- **Global Memory**: The largest memory space on the GPU (the device's VRAM), accessible to every running thread. It has the highest latency and serves as the medium for data transfer between the host and the device. Access to global memory should be minimized and carefully organized whenever possible.

2.2.5 Performance Considerations

The SIMT execution model of warps and the tiered memory hierarchy lead to several critical performance considerations.

Access to global memory is often the most significant performance bottleneck. To mitigate this, the GPU hardware attempts to coalesce memory requests. This is a technique to improve memory bandwidth utilization by servicing multiple logical memory reads from a warp in a single physical memory transaction. This is possible because DRAM technology fetches data in bursts [17]. Modern NVIDIA GPUs have a cache line size of 128 bytes [@25]. If the 32 threads in a warp access 32 consecutive 4-byte words in global memory, this 128-byte request can be satisfied with a single DRAM burst, achieving maximum bandwidth. Conversely, if the memory accesses are scattered and random, the hardware may require up to 32 separate transactions, drastically reducing performance.

A similar principle applies to shared memory, which is organized into 32 memory banks [@25]. Each bank can service one request per cycle. If all 32 threads in a warp access data in different banks, the entire request can be satisfied in a single cycle. However, if multiple threads access the same bank, a bank conflict occurs, and the accesses are serialized, reducing throughput.

At a higher level, a modern GPU can execute multiple operations concurrently, such as copying data from the host while a kernel is running. A key technique for maximizing utilization is to use CUDA streams. A stream is essentially a queue of operations that are executed in order. By using multiple streams, a programmer can enqueue work on the GPU in smaller, independent chunks. This allows the GPU scheduler to more aggressively overlap memory transfers with computation (latency hiding). For example, the GPU can be copying the next chunk of data from the host while simultaneously processing the current chunk, effectively hiding the latency of the PCIe bus transfer and keeping the compute cores busy.

The physical resources on each SM are finite, and their management directly impacts performance. While the programming model exposes what seems like a nearly unlimited number of registers to each thread, in reality, there is a limited, physical register file that all threads active on an SM must share. High register usage by a kernel can have two detrimental effects. Firstly, if a single thread requires more registers than the hardware can allocate, some of its variables will "spill" into global memory, adding massive latency to every access. Secondly, even if there is no spilling, if the total register demand of all threads in a block is too high, the hardware scheduler will be forced to launch fewer concurrent warps on the SM. This reduction in occupancy (the ratio of active warps to the maximum supported warps) hurts the SM's ability to hide memory latency by switching to other warps, leading to lower overall utilization.

Finally, any operation that forces parallel threads to execute sequentially undermines the benefits of parallelism and degrades performance. Branch divergence is a classic example: since all threads in a warp execute the same instruction, if-else statements can cause serialization. If threads in a warp take different paths, the hardware executes each path serially while idling the threads on the other path, negating the benefit of parallelism. A similar and often more severe bottleneck arises from atomic contention. When many threads in a block attempt to perform an atomic operation on the same memory location simultaneously, the hardware is forced to serialize these requests. In cases of extreme contention, the performance can degrade to the point where a traditional lock-based critical section might even be faster. Therefore, minimizing both control-flow divergence and high-contention atomic operations is important for writing efficient, scalable GPU code.

2.3 Related Work

2.3.1 Two-Choice Filter

Recent work by McCoy et al. [21] introduced the Two-Choice Filter (TCF), a data structure designed specifically for high-throughput, parallel execution on GPUs. The TCF shares the same high-level goal as the GPU-accelerated Cuckoo filter presented in this thesis: to provide a deletable, space-efficient filter optimized for the constraints of a GPU.

Structurally, the TCF is similar to a Cuckoo filter. It organizes fingerprints into blocks sized to fit within a GPU cache line (e.g., 128 bytes) to ensure high memory locality. Like the Cuckoo filter, it maps each item to two candidate blocks.

The fundamental difference lies in the insertion strategy. The authors argue that the eviction chains inherent to Cuckoo hashing result in poor memory coherence on GPUs, as a single insertion may trigger a cascade of random memory writes. To avoid this, the TCF uses a strategy derived from the "power-of-two-choices" paradigm [2]. The insertion logic is as follows:

- To insert an item, the TCF inspects both candidate blocks.
- The new fingerprint is placed in the block that is currently less full.
- There are no evictions. If both candidate blocks are full, the insertion into the main table fails.

This approach guarantees a fixed number of memory accesses per insertion. However, because the blocks must be small enough to fit in GPU cache lines, the statistical variance in load distribution increases, leading to premature failures even when the total table is far from full. To address this, the TCF relies on a backing store, a small, secondary hash table to catch these overflows. This hybrid architecture allows the TCF to maintain a high overall occupancy (up to 95%) while keeping the average case insertion logic simple.

For implementation, the TCF leverages CUDA Cooperative Groups to coordinate threads within a warp for lock-free intra-block operations. Additionally, it offers a "Bulk API" that pre-sorts items before insertion, further maximizing memory coalescing. In summary, the TCF prioritizes a non-evicting strategy to maximize memory bandwidth, trading the complexity of eviction logic for the architectural complexity of managing a secondary overflow structure.

2.3.2 Quotient Filter

The Quotient Filter (QF) is a high-performance probabilistic data structure that improves upon the Bloom filter by supporting dynamic deletions and offering superior space efficiency in many configurations [3]. It compactly stores small fingerprints using a scheme based on *Robin Hood hashing* [6]. For a target false positive rate ϵ, a QF uses approximately $1.053(2.125 + \log_2(1/\epsilon))$ bits per item, making it more space-efficient than a space-optimized Bloom filter whenever $\epsilon \leq 1/64$ [21].

The core mechanism relies on splitting a p-bit hash into a q-bit *quotient* and an r-bit *remainder*. The quotient determines an item's "canonical slot" in a table of 2^q slots, while the remainder is the value actually stored. If the canonical slot is occupied, linear probing is used to find the next empty slot. All remainders sharing the same quotient form a contiguous *run*, and sequences of runs form *clusters*. Three metadata bits per entry (`is_occupied`, `is_continuation`, `is_shifted`) encode the structure of these runs, allowing for the reconstruction of the original hash during a lookup.

From a GPU design perspective, the Quotient Filter presents a specific trade-off. Its linear-probing nature results in high cache locality, a desirable property for GPU architectures. However, the insertion process is fundamentally sequential. Inserting a new remainder often requires shifting a sequence of existing remainders to maintain the sorted order required by Robin Hood hashing. This shifting operation is difficult to parallelize efficiently and often results in high thread divergence.

GPU-Based Counting Quotient Filter (GQF)

Early attempts to port the QF to GPUs, such as the work by Geil et al. [14], suffered from significant limitations, including a lack of counting support, high space overhead, and limited scalability (supporting fewer than 2^{26} items). To address these issues, McCoy et al. introduced the GPU-based Counting Quotient Filter (GQF) [21].

The GQF is a highly optimized implementation designed to overcome the shortcomings of previous GPU quotient filters. It supports a comprehensive feature set, including counting, deletions, and resizing. Notably, it also uses an "even-odd" phased approach for bulk insertions to manage concurrency without the complex locking schemes that typically bottleneck concurrent linear probing.

Despite these advancements, the GQF is still bound by the architectural challenge of element shifting. While optimized for bulk updates, concurrent insertions remain difficult to parallelize. The necessity of shifting elements implies that insertion throughput is heavily dependent on the filter's load factor and the distribution of keys, potentially limiting performance compared to bucket-based approaches like the Cuckoo filter which rely on localized atomic swaps.

3 Implementation

This chapter details the complete software implementation of the GPU Cuckoo filter, translating the theoretical concepts from the previous chapter into a concrete, high-performance software artifact. The primary goal of the implementation is to provide a library that is not only fast but also robust, configurable, and scalable.

The chapter begins by outlining the fundamental design of the data structure itself, covering its compile-time configuration options, its memory layout optimized for parallel hardware, and its comprehensive public-facing API. Following this, the chapter dives into the core of the implementation: the parallel algorithms for insertion, lookup, and deletion. Each operation is presented as a CUDA kernel, with detailed explanations of the logic used to handle concurrency, manage the eviction process, and ensure correctness.

Beyond the baseline algorithms, some advanced optimization techniques are explored. These include a sorted insertion strategy to improve memory locality for large filters, an alternative eviction method designed to reduce the amount of necessary evictions at high load factors, the option to load more fingerprints at once when using supported hardware and the usage of bucket placement policies other than the original XOR-based partial-key cuckoo hashing.

Finally, two major extensions are presented that elevate the filter from a single-GPU data structure to a system-level component. An IPC wrapper is detailed, which enables zero-copy sharing of the filter between multiple processes on the same machine. To address datasets larger than a single GPU's memory, the following section then describes a multi-GPU implementation that partitions the workload and data across multiple devices, using high-speed interconnects for efficient communication when possible.

3.1 Challenges

Based on the GPU architecture and programming principles discussed in Chapter 2, translating the Cuckoo filter algorithm into a performant parallel implementation presents several specific technical hurdles. The following challenges directly influenced the design and optimization of the kernels detailed in this chapter:

- **Managing Concurrency and Race Conditions**: A naive parallel implementation where thousands of GPU threads attempt to read and write to the same buckets simultaneously would lead to race conditions and data corruption. Designing an efficient, lock-free mechanism using atomic operations to handle these concurrent memory accesses is one of the primary challenges.
- **Parallelizing the Eviction Path**: During an insertion, if both candidate buckets are full, an existing item must be evicted and reinserted into its alternate location. This eviction process can cascade, leading to multiple evictions. Parallelizing this process is complex, as it requires careful coordination to ensure that threads do not interfere with each other's operations or create deadlocks.
- **Optimizing for Coalesced Memory Access**: The random-access nature of Cuckoo filter operations can lead to scattered, uncoalesced memory accesses, which severely degrade performance. Developing strategies like data sorting to improve memory locality is essential.
- **Balancing Occupancy and Register Pressure**: Achieving a high and balanced occupancy across the filter is key to its space efficiency. However, complex insertion logic can increase the number of registers used per thread. High register usage can limit the number of active warps on an SM, thereby reducing occupancy and the hardware's ability to hide memory latency.

3.2 Data Structure Design

The design of the GPU-accelerated Cuckoo filter emphasizes compile-time configuration and a memory layout optimized for parallel access patterns. This section details these design choices, from the high-level configuration structure down to the public-facing API.

3.2.1 Compile-Time Configuration

To maximize performance by allowing the compiler to generate highly specialized code, the filter's core parameters are defined at compile-time as template parameters. They are consolidated into a single configuration structure, `CuckooConfig`, which provides a clean and explicit way to instantiate the filter. The primary configuration options are:

- `bitsPerTag`: Defines the size of each fingerprint in bits. Has to be 8, 16, or 32.

- `bucketSize`: Specifies the number of fingerprints stored in each bucket. A smaller bucket size generally leads to a lower false positive rate but can negatively impact performance and overall occupancy. A default of 16 was chosen through performance testing (See Section 4.3).
- `maxEvictions`: Sets the maximum number of evictions a single thread is allowed to perform during an insertion attempt before it gives up and reports a failure. The default value of 500 was taken from [9].
- `blockSize`: Configures the thread block size for the internal CUDA kernels. This parameter can be tuned to optimize GPU occupancy for different hardware architectures. A default of 256 was chosen through empirical testing.
- `AltBucketPolicy`: A class which encapsulates the logic for hashing items, calculating the required number of buckets, and deriving an item's alternate bucket. The default implementation uses the standard XOR-based partial-key cuckoo hashing, but alternatives based on the "Additive and Subtractive" Cuckoo filter [16] and an offset-based version [32] have also been implemented to demonstrate this flexibility (See Sections 3.4.3 & 4.12).
- `evictionPolicy`: The policy used to handle insertions into full buckets. The default implementation employs a Breadth-First Search (BFS) strategy. Unlike the traditional greedy (DFS) approach which immediately evicts a random victim, the BFS policy first scans the alternate buckets of the existing items to find an empty slot. As detailed in Section 3.4.2, this strategy minimizes the length of eviction chains and reduces global memory traffic (See Sections 3.4.2 & 4.4).
- `WordType`: Specifies the underlying integer type used for atomic operations and fingerprint storage. The default is `uint64_t`, which is generally optimal for modern GPU architectures. However, this can be reconfigured (e.g., to `uint32_t`) to align with specific hardware characteristics regarding atomic throughput. A benchmarking utility is provided to help users determine the optimal setting, which is primarily useful in cache-resident workloads.

Throughout the implementation, static assertions are used to enforce important invariants at compile time, such as ensuring that certain parameters are powers of two to allow for efficient bitwise AND operations instead of more costly modulo arithmetic.

3.2.2 Memory Layout and State Management

The filter's primary data storage is a single, contiguous array of fixed-size buckets allocated in the GPU's global memory. To maximize memory bandwidth and avoid misaligned access, the internal layout is carefully structured:

- Each bucket is composed of an array of 64-bit unsigned integer words.
- Fingerprints (tags) are tightly packed within these 64-bit words. For example, a 64-bit word can hold eight 8-bit fingerprints or four 16-bit fingerprints.

While this packed layout necessitates the use of bitwise shift and mask operations to extract individual fingerprints, the additional computational cost of these operations is negligible compared to the latency of memory access, making this a highly beneficial trade-off.

Furthermore, this enables the use of SWAR (SIMD Within A Register) techniques. Bitwise magic numbers can be used to search for empty slots (zero detection) or matching tags (XOR comparison) in parallel within a single register [@1]. This eliminates the need for branching loops when iterating over slots within a word.

The filter's state is maintained by a single global atomic counter that tracks the total number of occupied slots. This counter resides in device memory and is updated atomically by the insertion and deletion kernels. Its value is only lazily copied to the host when an explicit query is made, minimizing host-device communication.

Due to a limitation of CUDA C++ where kernels cannot be class members, the implementation passes a pointer to the filter's class instance to each launched kernel. This allows the threads to access the filter's configuration and state, such as data pointers and capacity, as needed.

3.2.3 Public API

The filter exposes a comprehensive public interface for easy integration. Two sets of APIs are provided: a traditional C-style API that operates on raw device pointers and item counts, and, if the Thrust library is available, a set of convenience wrappers that accept `thrust::device_vector` objects.

`explicit CuckooFilter(size_t capacity)`

The constructor takes a single argument for the desired capacity in terms of the number of items. It guarantees that at least this many items can be stored. Depending on the chosen `AltBucketPolicy`, the actual allocated capacity may be larger. For example, the default XOR-based strategy requires the number of buckets to be a power of two, so the requested capacity is rounded up to the next suitable size.

`size_t insertMany(const T* d_keys, const size_t n)`

This function attempts to insert a batch of `n` items from a buffer in device memory pointed to by `d_keys`. It is the caller's responsibility to ensure the pointer is valid and the buffer is sufficiently large. The function returns the updated total number of occupied slots in the filter after the insertion attempt.

`size_t insertManySorted(const T* d_keys, const size_t n)`

To address the performance penalties of random memory access, this variant introduces a pre-sorting step. The algorithm computes bucket indices and fingerprints, packs them into key-value pairs, and sorts them via CUB's radix sort before insertion begins. This ensures that consecutive threads target contiguous memory regions. While this strategy yields performance gains for massive datasets by minimizing DRAM transaction overhead, the added computational cost of sorting makes it less efficient for smaller filters that already fit within the GPU cache.

`void containsMany(const T* d_keys, const size_t n, bool* d_output)`

Performs a batch lookup for `n` items. The results are written to the `d_output` device buffer, where the boolean at each index corresponds to whether the item at the same index in `d_keys` was found.

`size_t deleteMany(const T* d_keys, const size_t n, bool* d_output = nullptr)`

Performs a batch deletion of `n` items. If the optional `d_output` buffer is provided, the success or failure of each individual deletion is reported in the same manner as the lookup operation.

`void clear()`

Resets the internal state of the filter by setting all buckets and the occupancy counter to zero. This operation does not deallocate any device memory.

`float loadFactor()`

Returns the current load factor of the filter, calculated as the number of occupied slots divided by the total capacity.

`size_t countOccupiedSlots()`

A debugging utility that provides a ground-truth count of occupied slots. It operates by copying the entire filter data to the host and manually counting every non-zero fingerprint. As this is an extremely slow, synchronous operation, it should only be used for verification purposes if the internal atomic counter is suspected to be inaccurate.

3.3 Parallel Algorithms

The core of this thesis is the design of parallel algorithms for the Cuckoo filter's primary operations: insertion, lookup, and deletion. These algorithms are designed to be launched as CUDA kernels, where many threads cooperate to process batches of items simultaneously.

3.3.1 Insertion

The parallel insertion algorithm is designed to handle a large batch of items in parallel, with each CUDA thread being responsible for inserting a single item. The process for each thread is as follows:

1. **Hashing and Key Generation**: Each item is first hashed into a 64-bit value using the xxHash64 algorithm, chosen for its high performance and excellent statistical properties. This hash is split: the upper 32 bits derive the fingerprint, and the lower 32 bits determine the primary bucket index. Distinct hash parts are used to avoid fingerprint clustering. The alternate bucket index is then calculated using the partial-key cuckoo hashing scheme (Section 2.1.4).
2. **Direct Insertion Attempt**: The thread checks the two candidate buckets. To distribute items evenly and reduce contention on the first slots of a bucket, the thread does not start scanning at index 0. Instead, it uses the item's fingerprint to calculate a pseudo-random starting word index. It then iterates through the bucket's 64-bit words, wrapping around to the beginning. For each word, it utilizes a bitwise SWAR algorithm [@1] to generate a mask of empty slots. If a slot is found, an atomic Compare-And-Swap (CAS) attempts to insert the fingerprint.

3. **Eviction Process**: If both candidate buckets are full, the thread initiates the eviction process. It randomly selects one bucket and a random occupied slot within it. It atomically replaces the existing fingerprint (`tag_old`) with its own (`tag_new`). The evicted fingerprint becomes the new item to insert, and the thread calculates its alternate bucket to continue the process.
4. **Termination**: The eviction loop continues until an empty slot is found or a limit on the number of evictions is reached, triggering an insertion failure.

To maintain an accurate count of the total items in the filter without creating a bottleneck on a single atomic counter, a hierarchical reduction is employed. Each thread that successfully inserts an item contributes a +1. These values are first summed efficiently at the warp level using shuffle instructions, then aggregated at the block level using shared memory, and finally, a single atomic addition per block is performed on the global counter in device memory.

Algorithm 1 Parallel Insertion

```
function INSERT(key)
    $h \leftarrow \mathrm{hash}(\text{key})$
    $fp \leftarrow \mathrm{fingerprint}(h)$
    $i_1 \leftarrow \mathrm{primary\_index}(h)$
    $i_2 \leftarrow \mathrm{alternate\_index}(i_1, fp)$
    // Phase 1: Try direct insertion using SWAR
    if TRYINSERT($i_1$, $fp$) or TRYINSERT($i_2$, $fp$) then
        return Success
    // Phase 2: Eviction Chain
    $b \leftarrow$ randomly pick $i_1$ or $i_2$
    tag $\leftarrow fp$
    for $n = 1$ to maxEvictions do
        $s \leftarrow$ random slot index in bucket $b$
        // Atomically swap current tag with new tag
        tag $\leftarrow$ AtomicExchange(bucket$[b]$.slot$[s]$, tag)
        $b \leftarrow$ alternate_index($b$, tag)
        if TRYINSERT(bucket $b$, tag) then
            return Success
    // Table too full
    // Caller will have to rebuild
    return Failure

function TRYINSERT(bucket, tag)
    $start \leftarrow (\text{tag} \bmod \text{bucketSize})/\text{tagsPerWord}$
    for $i = 0$ to wordCount $- 1$ do
        $idx \leftarrow (start + i) \bmod \text{wordCount}$
        word $\leftarrow$ bucket[idx]
        while word has empty slots do
            slot $\leftarrow$ FINDNEXTEMPTY(word)
            if AtomicCAS(word, EMPTY $\rightarrow$ tag at slot) then
                return True
            Reload word
    return False
```

3.3.2 Lookup

The parallel lookup algorithm is a read-only operation optimized for memory access. Each thread calculates the fingerprint and two bucket indices. Similar to insertion, the thread determines a random starting word based on the fingerprint to avoid checking the same memory locations first for every query.

The key optimization is vectorized, non-atomic memory loads combined with SWAR comparisons. Modern GPU hardware can load 128 bits (16 bytes) in a single instruction. The kernel loads two 64-bit words simultaneously starting from the randomized offset. It then broadcasts the query fingerprint and XORs it with the loaded data to check for matches in parallel using constant-time arithmetic, eliminating branching loops.

Algorithm 2 Parallel Lookup

```
function CONTAINS(key)
    $h \leftarrow \text{hash}(\text{key})$
    $fp \leftarrow \text{fingerprint}(h)$
    $i_1 \leftarrow \text{primary\_index}(h)$
    $i_2 \leftarrow \text{alternate\_index}(i_1, fp)$
    // Check both buckets (read-only, no locking needed)
    return FIND($i_1$, $fp$) or FIND($i_2$, $fp$)

function FIND(bucket, tag)
    pattern ← BROADCASTTAG(tag)
    // Random start index aligned to 128-bit boundary
    $start \leftarrow (\text{tag} \bmod \text{bucketSize})/\text{tagsPerWord}$
    $start \leftarrow \text{FloorToEven}(start)$
    for $i = 0$ to wordCount $- 1$ step 2 do
        $idx \leftarrow (start + i) \bmod \text{wordCount}$
        $w_1, w_2 \leftarrow$ LOADWORDS(bucket, $idx$)
        // Use SWAR to check for matches in parallel
        if HASZEROBYTE($w_1 \oplus$pattern) or HASZEROBYTE($w_2 \oplus$pattern) then
            return Success
    return Failure
```

3.3.3 Deletion

The parallel deletion algorithm leverages SWAR to locate and remove items efficiently. Like the other operations, it iterates through the bucket in 64-bit words, starting at a pseudo-random offset derived from the fingerprint.

The thread broadcasts the target tag and uses SWAR to find matches. If a match is found, it attempts an atomic CAS to set the specific slot to EMPTY (zero). If the CAS fails (due to concurrent modification), the thread reloads and retries. This ensures thread safety without locking. The operation continues until the item is removed, or the entire bucket has been scanned.

Algorithm 3 Parallel Deletion

```
function REMOVE(key)
    $h \leftarrow \text{hash}(\text{key})$
    $fp \leftarrow \text{fingerprint}(h)$
    $i_1 \leftarrow \text{primary\_index}(h)$
    $i_2 \leftarrow \text{alternate\_index}(i_1, fp)$
    // Attempt to remove from either valid location
    return REMOVE($i_1$, $fp$) or REMOVE($i_2$, $fp$)

function REMOVE(bucket, targetTag)
    $start \leftarrow (\text{tag} \bmod \text{bucketSize})/\text{tagsPerWord}$
    for $i = 0$ to wordCount $- 1$ do
        $idx \leftarrow (start + i) \bmod \text{wordCount}$
        word ← bucket[idx]
        mask ← SWAR_MATCH(word, targetTag)
        while mask is not 0 do
            slot ← FINDFIRSTSET(mask)
            // Attempt to atomically set specific slot to EMPTY
            if AtomicCAS(word, targetTag → EMPTY at slot) then
                return Success
            // Reload and re-check if CAS failed
            Reload word
            mask ← SWAR_MATCH(word, targetTag)
    return Failure
```

3.4 Optimization Techniques

Beyond the core algorithm design, some noteworthy optimization strategies were explored to further improve insertion performance, particularly in cases where the filter is too large to fit into the device's L2 cache or under high load factors.

3.4.1 Sorted Insertion

One version of the insertion algorithm was implemented to enhance memory locality. Before the main insertion kernel is launched, the items are first packed into a temporary structure where the upper bits represent the primary bucket index and the lower bits contain the fingerprint. This array is then sorted in parallel on the GPU using CUB's high-performance radix sort. Doing this ensures that consecutive threads in the subsequent insertion kernel are likely to be working on items that map to the same or nearby buckets. This approach led to a measurable performance increase once the filter size far exceeded the GPU's L2 cache and memory bandwidth is not abundant, as the random memory accesses of the standard approach were more heavily penalized. However, for filters that fit entirely within the L2 cache (which has a hit rate of over 90% in these cases), the overhead of the initial packing and sorting pass made this version considerably slower and not worth using (see Section 4.5).

3.4.2 Alternative Eviction Strategy

The standard eviction process uses a greedy, depth-first-search (DFS) approach, where a thread immediately follows the eviction chain of a single evicted item. An alternative strategy was implemented to reduce the average length of these chains. When an eviction is necessary, instead of picking one random item to evict, the thread first inspects half the bucket at random. For each candidate, it checks if its alternate bucket has an empty slot. If such a "safe" eviction is found, the swap is performed, and the insertion completes in a single step. If all candidates lead to full alternate buckets, the algorithm falls back to the original DFS-style greedy eviction. This method was found to have a negligible impact when inserting into an empty filter, but it provided a moderate speed-up when inserting into a filter with a high load factor (e.g., 70% or more), where long eviction chains are more common (see Section 4.4).

3.4.3 Flexible Bucket Placement Policies

The standard partial-key cuckoo hashing scheme relies on the XOR operation to determine alternate bucket locations $(i_2 = i_1 \oplus hash(fp))$. For this bijective property to map validly onto the buckets, the number of them must strictly be a power of two. This constraint introduces a significant memory footprint issue: if a dataset requires slightly more capacity than 2^n, the filter must be sized to 2^{n+1}, result in nearly 50% higher memory usage in the worst case.

To mitigate this over-provisioning, two alternative bucket placement policies were implemented. These strategies allow for more granular sizing of the filter, decoupling the capacity from powers of two.

Additive and Subtractive Cuckoo Filter (ASCF)

This policy is based on the work of Huang et al. [16] and relaxes the power-of-two constraint by dividing the filter into two equal-sized blocks (requiring only that the total number of buckets be even).

- **Primary Index**: The primary bucket i_1 is always mapped to Block 0.
- **Alternate Index**: The alternate bucket i_2 is located in Block 1. It is calculated by adding a hash of the fingerprint to i_1 modulo the block size.
- **Inverse Calculation**: To find the alternate bucket for an item currently residing in Block 1, the logic is inverted: the fingerprint hash is subtracted to map back to Block 0.

This arithmetic symmetry replaces the bitwise symmetry of XOR, allowing the filter to grow linearly in steps of two buckets rather than exponentially.

Offset-Based Cuckoo Filter

The second policy, derived from the work of Schmitz et al. [32], uses an asymmetric offset combined with a "choice bit". In this scheme, one bit of the stored fingerprint is reserved to indicate whether the item is currently in its primary or alternate location.

- If the choice bit is 0, the item is in its primary bucket b. The alternate bucket is calculated as:

$$b' = (b + \text{offset}(fp)) \bmod m$$

- If the choice bit is 1, the item is in its alternate bucket b'. The primary bucket is calculated as:

$$b = (b' - \text{offset}(fp)) \bmod m$$

When an item is moved between buckets during an eviction, its choice bit is flipped. This approach supports any number of buckets m, offering maximal space efficiency. The trade-off is a reduction in the effective fingerprint size by one bit (increasing the false positive rate slightly) and the need to update the choice bit during evictions.

Evaluation

These policies trade computational simplicity for memory flexibility. In Section 4.12, these three strategies (XOR, ASCF, and Offset-based) are benchmarked against each other to verify potential performance costs from eliminating the power-of-two constraint.

3.5 IPC Wrapper

To allow the GPU-accelerated Cuckoo filter to be used as a high-performance, system-wide service, an IPC wrapper was developed. This wrapper exposes the filter's functionality through a client-server architecture, with a core focus on enabling true zero-copy data transfer for maximum efficiency.

The architecture is built upon two primary mechanisms: a shared memory queue for communication and CUDA's IPC API for data access. A fixed-size ring buffer, chosen for its superior performance and simplicity, is created in a POSIX shared memory segment to act as a command queue. The key is how input data is handled. Instead of sending buffers through the queue, a client performs the following steps:

1. Allocate memory in its own GPU memory space for the batch of items to be processed.
2. Obtain opaque memory handles to this device memory using `cudaIpcGetMemHandle`.
3. Place these handles inside a request message alongside other metadata like the size of input and output buffers, which is then enqueued into the shared memory ring buffer.

The server runs a single worker thread that continuously dequeues requests from the command queue. For each request, the server's thread uses `cudaIpcOpenMemHandle` on the provided handle. This operation maps the underlying physical memory, originally allocated by the client, into the server's own virtual address space. This yields a valid device pointer that the server can use to launch kernels and read the input keys directly, thereby avoiding the significant overhead of inter-process data copies.

The queue is blocking by design, meaning a client attempting to enqueue a request into a full queue will wait until a slot becomes available. On top of this, the server supports both a graceful shutdown, where it stops accepting new requests but processes all outstanding items, and a forced shutdown that cancels all pending requests.

To simplify its use, a client library abstracts away the low-level IPC details. An optional Thrust wrapper is also provided to allow for seamless interaction with `thrust::device_vector`. This architecture could be enhanced in several ways in the future:

- The blocking communication model could be evolved into a fully asynchronous interface, inspired by modern APIs like Linux's `io_uring`, to maximize throughput in high-concurrency scenarios.
- Support could be added for multiple worker threads managing filters on different GPUs to enable further load balancing.
- The filter could be exposed over a network by leveraging technologies like RDMA (Remote Direct Memory Access) to preserve the zero-copy transfer properties.

3.6 Multi-GPU

To handle datasets that exceed the memory capacity of a single graphics card, a multi-GPU version of the Cuckoo filter was implemented. This version transparently partitions the data and workload across all available devices, presenting a unified interface to the user. This approach, however, necessitates a shift in the public API, as the input and output buffers must now reside in host memory to accommodate their potentially massive size.

3.6.1 Architectural Design

The core design choice was to instantiate a completely independent Cuckoo filter instance on each GPU, each managing its own dedicated device memory. While a single, logically distributed filter was considered, the practical overhead of managing eviction chains across the high-latency interconnect between GPUs was deemed prohibitive, as it would severely compromise insertion performance.

To distribute items across these independent filters, a deterministic partitioning scheme is used. Each key is assigned to a specific GPU based on the result of $\mathrm{hash}(key) \bmod n$, where n is the number of GPUs. While a uniform hash function would ideally distribute keys evenly, minor variances are expected in practice. To account for this and prevent premature insertion failures on one device, each GPU's filter is allocated slightly more capacity (2%) than its proportional share.

3.6.2 Data Distribution and Processing

Distributing the input data from the host to the correct GPU partition is a complex challenge. A naive approach involving a single "primary" GPU that partitions and distributes all data creates a severe load imbalance. Therefore, a fully parallelized, multi-stage workflow was implemented leveraging the Gossip library [18] for optimized communication:

1. **Initial Data Scatter**: The host-side input array is processed in large chunks. Each GPU copies a distinct and proportional sub-chunk from the host into its own device memory in parallel.
2. **Destination Calculation**: Each GPU launches a lightweight kernel to hash its local keys and determine the target GPU index for each item.
3. **Topology-Aware Multisplit and Exchange**: To redistribute the keys to their correct owner GPUs, the implementation utilizes the Gossip library. Unlike a standard all-to-all approach, Gossip performs an efficient *multisplit* operation combined with a topology-aware data exchange. It uses hints for the system's interconnect topology (distinguishing between NVLink and PCIe connections) to generate an optimized transfer plan. This ensures that the data shuffling phase maximizes the available bandwidth and minimizes contention on the interconnect.
4. **Parallel Processing**: Once the exchange is complete, each GPU holds all the keys that belong to its partition. At this point, each GPU proceeds to execute the standard single-GPU insertion, lookup, or deletion kernel on its local data, fully in parallel with the other devices.

3.6.3 Result Consolidation

For lookup and deletion operations that require an output array, the results generated on each GPU must be consolidated and reordered to match the original input order. To solve this efficiently, the Gossip library is again utilized to reverse the exchange process, sending results back to the GPUs that originally held the keys (pre-partitioning). This allows each GPU to perform a parallel scatter operation, writing its portion of the results into the final host output buffer in the correct order.

3.6.4 Overheads

The use of the Gossip library significantly mitigates the overheads associated with manual partitioning. Previous approaches relying on `thrust::sort` and standard NCCL collectives often incurred high temporary memory costs and sorting latency. Gossip's optimized multisplit primitive reduces the auxiliary memory footprint and leverages low-level hardware features for faster data movement. However, the fundamental latency of moving data off-chip remains a bottleneck compared to single-GPU execution. The processable chunk size must still be managed to ensure that the input buffers, the exchange buffers, and the filter itself fit within device memory simultaneously.

3.7 Testing and Verification

To ensure the correctness and reliability of the implementation, a comprehensive verification strategy was employed. This strategy combines fine-grained unit tests to validate individual components with end-to-end empirical tests designed to confirm the functional correctness and theoretical properties of the final software artifacts. The following sections detail the methodologies used for unit testing and the empirical validation of the single-GPU filter, the IPC wrapper, and the multi-GPU implementation.

3.7.1 Unit Testing

The Googletest framework is utilized to implement a comprehensive suite of unit tests. These tests are designed to validate the correctness of individual components in isolation. The test suite covers common use cases and known edge cases (such as empty inputs, full filters, and operations on zero-capacity filters). This low-level validation provides a strong foundation of correctness upon which the larger system is built.

3.7.2 Functional and Empirical Verification

Beyond unit tests, a dedicated test binary was created for each primary output artifact to empirically verify its end-to-end functionality. A standardized test flow was developed to rigorously validate the correctness of the filter implementations. This procedure was then adapted for the specific contexts of the single-GPU, IPC, and multi-GPU versions.

Standard Verification Procedure

The core empirical validation follows a standardized, multi-stage procedure using random 64-bit integer keys:

1. **Insertion**: Random keys from the range $[0, 2^{32} - 1]$ are inserted into an empty filter until a specified target load factor is reached.

2. **No-False-Negatives Test**: A query is performed for all the previously inserted keys. The test passes only if every key is successfully found, confirming that no false negatives have occurred.
3. **False-Positive Rate Measurement**: The empirical false-positive rate is measured by querying one million distinct keys known not to be in the filter (drawn from the disjoint range $[2^{32}, 2^{64} - 1]$). The observed rate of positive matches is then calculated and compared against the theoretically expected rate to ensure it falls within an acceptable margin.
4. **Deletion Correctness and Stability Test**: Half of the initially inserted items are deleted. A subsequent query for the entire original set of keys is then conducted to verify three conditions:
   - All non-deleted items must still be found.
   - The vast majority of deleted items must no longer be found
   - The rate of any remaining positive hits for the deleted items must be statistically consistent with the filter's theoretical false-positive rate.

Single-GPU Filter Verification

The core single-GPU filter implementation was subjected to the standard verification procedure outlined above. The test binary for this version operates directly on device-side data buffers, providing a controlled environment to validate the correctness of the CUDA kernels on a single device.

IPC Wrapper Verification

The IPC wrapper is validated using a dedicated test binary designed to operate in one of two modes: as a server or as a client. The procedure is conducted by first manually launching an instance of the binary in server mode. Subsequently, a separate instance is launched in client mode, which then spawns multiple internal threads to concurrently send batches of requests to the running server. This test serves as an empirical proof-of-concept for the IPC mechanism, validating the end-to-end communication, command queuing, and zero-copy data transfer.

Multi-GPU Verification

The multi-GPU implementation was also validated against the standard verification procedure. The test binary for this artifact allows for command-line configuration of the filter's capacity and the number of GPUs to utilize. A key difference from the single-GPU test is that all input and output data buffers reside in host memory. This setup rigorously exercises the entire data pipeline: from the initial scatter from the host, through the inter-GPU data exchange via Gossip, to the final consolidation of results back to the host. The tests are specifically designed to run on datasets that exceed the memory capacity of a single GPU, thereby validating the primary use case of this implementation.

Evaluation 4

The performance of the GPU-accelerated Cuckoo filter is evaluated to demonstrate its throughput, scalability, and resource efficiency. This chapter details the experimental setup, the baselines used for comparison, and the analysis of the results across various metrics.

4.1 Experimental Setup

The performance evaluation was conducted on two distinct hardware configurations to analyse how different memory architectures impact the scalability of the filters. System C is only used for the comparison between sorted and unsorted insertion as well as the comparison of the two eviction policies.

- **System A (GDDR7)**: An AMD EPYC 7713P (64 cores) paired with an NVIDIA RTX PRO 6000 Blackwell GPU featuring 96 GB of GDDR7 memory (1.8 TB/s). The system runs AlmaLinux 10.1 with NVIDIA driver 580.95.05 and CUDA 12.9.86.
- **System B (HBM3)**: An NVIDIA GH200 Grace Hopper system with 72 ARM Neoverse V2 cores and an H100 GPU featuring 96 GB of HBM3 memory (3.4 TB/s). The system runs Ubuntu 24.04.3 with NVIDIA driver 580.105.08 and CUDA 13.0.88.
- **System C (GDDR7)**: An AMD Ryzen 9 5900X (12 cores) paired with an NVIDIA RTX 5070 Ti GPU featuring 16 GB of GDDR7 memory (0.9 TB/s). The system runs NixOS 25.11 with NVIDIA driver 580.119.02 and CUDA 12.9.86.

It is important to note that these systems are not compared head-to-head to determine a single "winner." Rather, they serve as complementary testbeds to isolate architectural bottlenecks. While System B provides significantly higher memory bandwidth (HBM3 vs. GDDR7), System A features approximately 50% more CUDA cores. This disparity is important for the analysis: it allows for the differentiation between compute-bound and memory-bound algorithms. Specifically, performance gains on System A indicate a compute bottleneck, whereas scaling on System B confirms a memory bandwidth bottleneck.

All performance tests use 16-bit fingerprints (with equivalent space allocation for the Blocked Bloom filter) and random 64-bit integers as input keys.

4.1.1 Reference Implementations

To provide a comprehensive analysis, the proposed GPU Cuckoo filter is compared against the following data structures:

- **GPU Blocked Bloom Filter**: Sourced from the *cuCollections* library, serving as a high-performance baseline for an append-only filter. [24]
- **Original CPU Cuckoo Filter**: The reference implementation from the original 2014 paper by Fan et al. [9].
- **Optimized Partitioned CPU Cuckoo Filter**: A variant [31] utilizing multithreading (one thread per partition) to maximize CPU throughput without the overhead of SIMD instructions, which were found to degrade performance in this specific context. Note that this filter was excluded from the System B (ARM-based) benchmarks, as it has a hard dependency on x86-64 intrinsics that prevents compilation.
- **Bulk Two-Choice Filter (TCF)**: A modern, GPU-focused AMQ data structure that serves as a direct competitor. [21]
- **GPU Counting Quotient Filter (GQF)**: A highly space-efficient probabilistic data structure sourced from the same study as the TCF. This implementation supports counting and resizing on top of the other operations. [21]

4.1.2 Evaluation Metrics

The implementations are assessed using the following metrics:

- **Throughput**: Measured for insertions, lookups, and deletions and is evaluated under two distinct conditions (Section 4.2):
  - **L2-Resident**: A filter size small enough to fit entirely within the GPU's L2 cache.
  - **DRAM-Resident**: A larger filter size that necessitates global memory access.
- **False Positive Rate (FPR) and Trade-offs**: The empirical FPR is measured to verify adherence to theoretical bounds. Furthermore, a comparative analysis is performed to determine the maximum achievable throughput for each filter when constrained to specific target false-positive rates (Section 4.7).
- **Cache Efficiency**: L1 and L2 cache hit rates are measured to evaluate memory locality (Section 4.8).

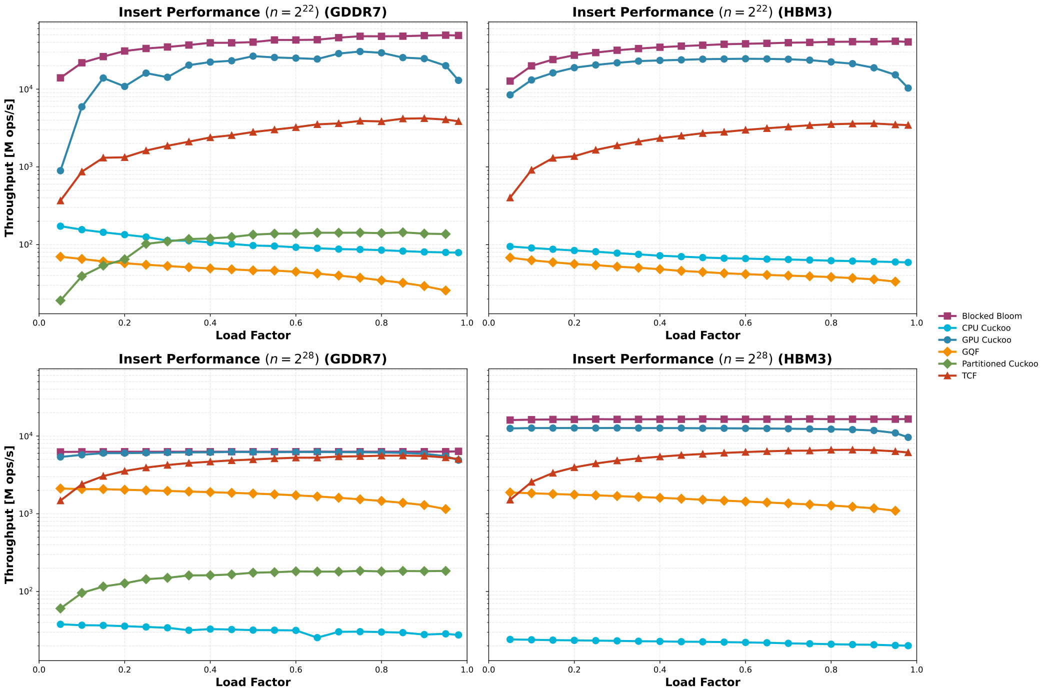
- **Hardware Utilization**: Resource usage is analysed as a percentage of the theoretical peak throughput ("Speed of Light") for compute, DRAM and cache bandwidth (Section 4.9).

4.2 Throughput Analysis

This section analyses the raw throughput of the filters in millions of operations per second (MOPS). To understand the impact of the memory hierarchy, each operation is evaluated under two distinct conditions: a small filter ($n = 2^{22}$, approx. 4.2 million items) that fits entirely within the L2 cache, and a large filter ($n = 2^{28}$, approx. 268 million items) that resides in DRAM.

4.2.1 Insertion Performance

The insertion results for both memory types are presented in Figure 4.1.


Insert Performance ($n = 2^{22}$) (GDDR7)

Insert Performance ($n = 2^{22}$) (HBM3)

Insert Performance ($n = 2^{28}$) (GDDR7)

Insert Performance ($n = 2^{28}$) (HBM3)

Fig. 4.1: Insertion Performance for L2-resident (Top) and DRAM-resident (Bottom) filters. Left side of each plot is System A (GDDR7), Right is System B (HBM3).

The GPU Cuckoo filter demonstrates exceptional competitiveness, effectively bridging the gap between append-only and dynamic data structures. Its insertion logic, relying on random atomic compare-and-swap operations, contrasts with the simpler, linear write patterns of the Blocked Bloom filter.

L2-Resident: When the filter fits in cache, the Cuckoo filter achieves a significant fraction of the Blocked Bloom filter's throughput while far surpassing all other filters.

DRAM-Resident: For large datasets, the Cuckoo filter scales strongly with memory bandwidth. Throughput increases significantly when moving from GDDR7 to HBM3, confirming that the algorithm successfully saturates the memory bus. In contrast, the TCF and GQF show stagnant or regressive performance on the faster HBM3 system. This reveals somewhat of a "scalability wall": these filters are bound by the speed of Shared Memory (SRAM) and intra-warp coordination rather than global DRAM bandwidth. Consequently, the Cuckoo filter remains the fastest dynamic option by a wide margin.

4.2.2 Lookup Performance

Query performance, shown in Figure 4.2, highlights the impact of the Cuckoo filter's bucket layout and short-circuiting capabilities.

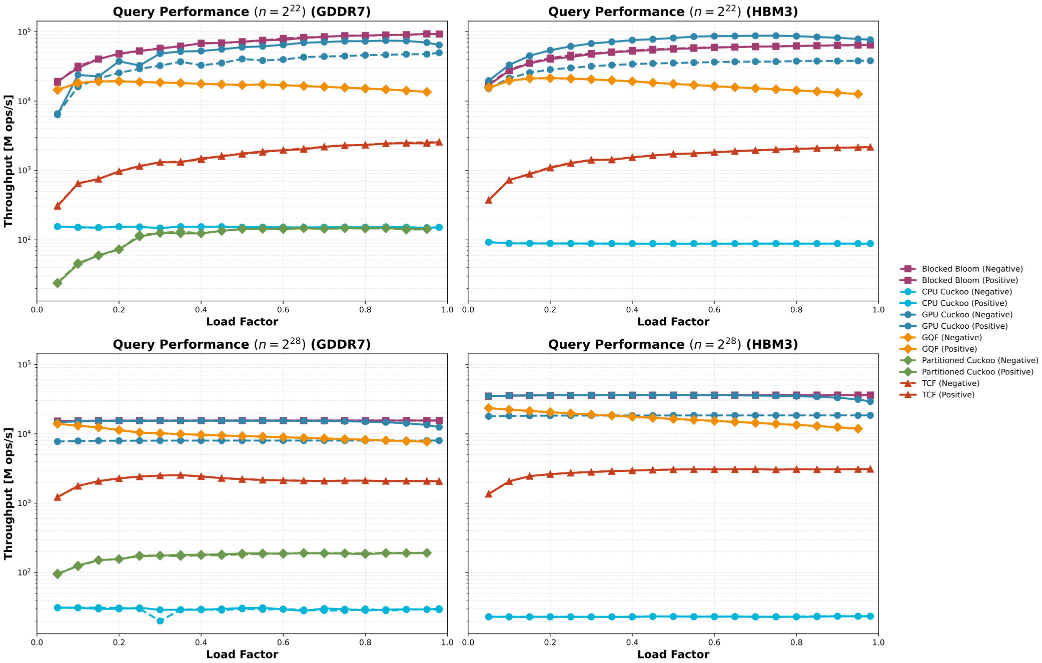


Fig. 4.2: Lookup Performance for L2-resident (Top) and DRAM-resident (Bottom) filters. Left side of each plot is System A (GDDR7), Right is System B (HBM3).

The Cuckoo filter exhibits a distinct performance profile based on the query result. Positive lookups are highly efficient: due to the large bucket size ($b = 16$), most items reside in their primary bucket, allowing the query to complete with a single memory transaction. This allows the Cuckoo filter to match or even outperform the Blocked Bloom filter when data is cached.

For negative lookups, throughput is roughly halved because the algorithm must definitively check both candidate buckets. This behaviour contrasts with the TCF, which utilizes cooperative groups to load both buckets in parallel for every query. While the TCF offers symmetric performance for positive and negative queries, its peak throughput is about an order of magnitude lower than the Cuckoo filter's. The GQF falls somewhat in the middle, benefiting from spatial locality during linear scans but failing to match the low latency of the bucketed approaches.

4.2.3 Deletion Performance

Across all scenarios, the GPU Cuckoo filter maintains a distinct performance lead, though the magnitude of this advantage depends heavily on memory residency (See Figure 4.3).

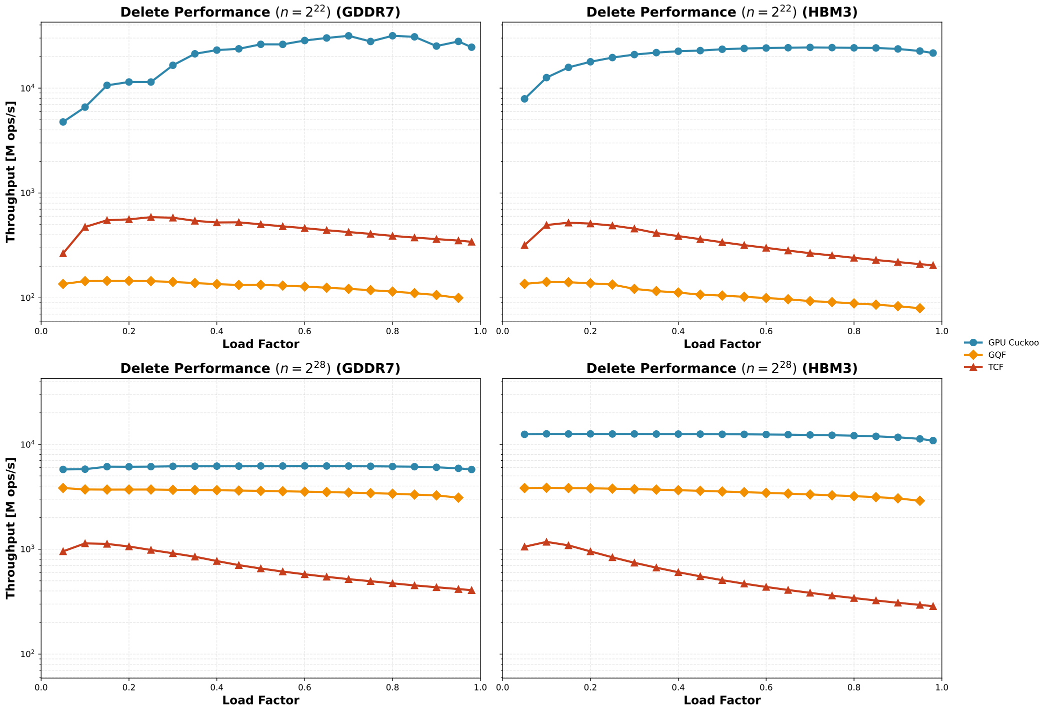


Fig. 4.3: Deletion Performance for L2-resident (Top) and DRAM-resident (Bottom) filters. Left side of each plot is System A (GDDR7), Right is System B (HBM3).

In the L2-resident scenario, the Cuckoo filter is orders of magnitude faster than both the TCF and GQF. The low latency of the L2 cache allows the Cuckoo filter's simple atomic operations to run at a very high speed, while the complex coordination logic of the competitors becomes a significant bottleneck.

However, in the DRAM-resident scenario, this gap narrows as global memory latency becomes the dominant factor for all implementations. On the GDDR7 system, the Cuckoo filter is approximately 67% faster than the GQF, while on the HBM3 system, the lead extends to approximately 3.5×. Despite the narrowing gap, the Cuckoo filter remains the fastest option due to fundamental algorithmic differences:

- **Cuckoo Filter**: Deletion is a simple, localized atomic operation (CAS).
- **GQF**: Requires shifting elements within a run to maintain sorted order. While efficient in bulk, this serial operation limits peak throughput.
- **TCF**: Requires complex coordination within cooperative groups to update block state, which scales poorly compared to independent atomic accesses.

4.3 Bucket Size Impact

The `bucketSize` parameter, defining the number of fingerprints stored in each bucket, plays an important role in the filter's overall performance. Testing demonstrates that performance degrades at both the lower and upper extremes of bucket sizing, necessitating a careful balance between algorithmic overhead and memory access granularity.

4.3.1 Performance Trade-offs

- **Small Buckets**: Configuring the filter with very small buckets negatively impacts performance. With fewer slots per bucket, the likelihood that a primary bucket is full increases significantly. This increases the average number of buckets that must be accessed and loaded from memory to complete an insertion or a lookup, resulting in higher latency.
- **Large Buckets**: Conversely, excessively large buckets introduce hardware-level inefficiencies. If a bucket's size exceeds the GPU's cache line size, fetching a single logical bucket requires multiple physical memory transactions. This significantly increases the memory bandwidth used per operation, as the number of cache lines that must be fetched scales with the size of the bucket.

4.3.2 Optimal Configuration

The experiments in Figure 4.4 identify distinct optimal configurations for insertion and lookup operations depending on the working set size:

- **Insertion**: A bucket size of 16 fingerprints was found to be the fastest configuration in all tested scenarios. This size appears to offer the optimal trade-off, providing enough slots to minimize eviction chains without incurring the bandwidth penalty of multi-cache-line fetches.

- **Lookup (L2-Resident)**: When the filter is sized to fit entirely within the GPU's L2 cache, a bucket size of 8 fingerprints yields the highest throughput. This is driven by instruction-level efficiency. A bucket of this size can be represented as two 64-bit words or a single 128-bit vector. This allows the entire bucket to be loaded into registers via a single vectorized machine instruction, maximizing the throughput of the L1/L2 cache hierarchy.

- **Lookup (DRAM-Resident)**: Once the filter size exceeds the L2 cache capacity, the bottleneck shifts to global memory bandwidth. In this case, a bucket size of 16 becomes favourable again. Since fetching data from DRAM creates a significant latency penalty, it is more efficient to process a larger "middle ground" bucket size that maximizes the utility of the data fetched in a standard 128-byte DRAM burst.

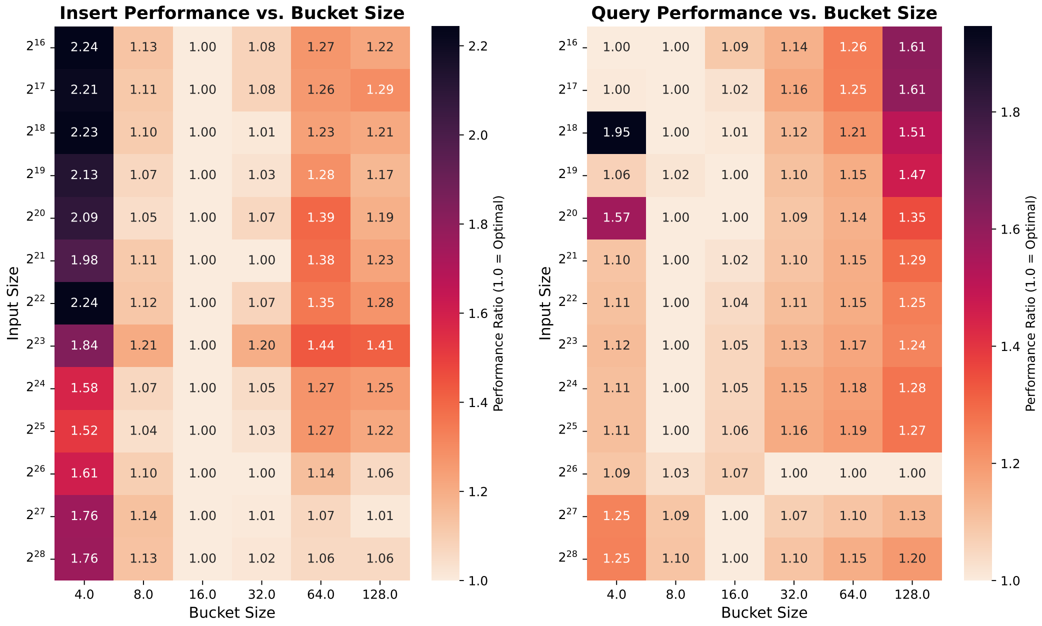


Fig. 4.4: Normalized throughput of the GPU Cuckoo filter for different bucket sizes on System A.

4.4 Eviction Policies

To evaluate the impact of the insertion strategy on performance and stability, a comparative analysis was conducted between the standard DFS eviction policy and the proposed BFS heuristic.

4.4.1 Experimental Setup

For this analysis and the subsequent evaluation of sorted insertion (Section 4.5), the experimental setup was expanded with a third hardware configuration to provide finer-grained data on GDDR7 performance scaling:

- **System C (GDDR7)**: An AMD Ryzen 9 5900X (12 cores) paired with an NVIDIA RTX 5070 Ti GPU featuring 16 GB of GDDR7 memory (0.9 TB/s). The system runs NixOS 25.11 with NVIDIA driver 580.119.02 and CUDA 12.9.86.

The tests use a fixed capacity of either 2^{22} (L2-resident) or 2^{28} (DRAM-resident) slots. To accurately measure performance in the critical high-load scenario, the insertion workload is split based on the target load factor α. For each data point, the filter is first pre-filled with 75% of the total items required to reach the target load (i.e., $0.75 \cdot \alpha \cdot capacity$). Subsequently, the remaining 25% of items are inserted to reach the final target load α, and the throughput of this second phase is recorded. This ensures that the measurement captures the performance behaviour specifically as the filter transitions from a moderately full state to the final target occupancy, effectively isolating the impact of the eviction strategy.

4.4.2 Eviction Reduction Analysis

The premise of the BFS eviction policy is that by investing more computational effort to search for a "local" empty slot, the filter can avoid triggering long, expensive eviction chains. To validate this hypothesis, the average number of evictions performed per inserted item was measured.

As shown in Figure 4.5, the BFS policy successfully lowers the eviction rate compared to the greedy DFS approach.

As the filter fills ups, the DFS strategy (which picks a random victim immediately upon collision) sees a quick exponential increase in evictions. In contrast, the BFS strategy delays this spike significantly. By checking up to half the bucket before resorting to an eviction, the BFS approach resolves many collisions locally.

This reduction in evictions directly translates to a reduction in global memory writes, since every eviction saved is an atomic read-modify-write transaction avoided.

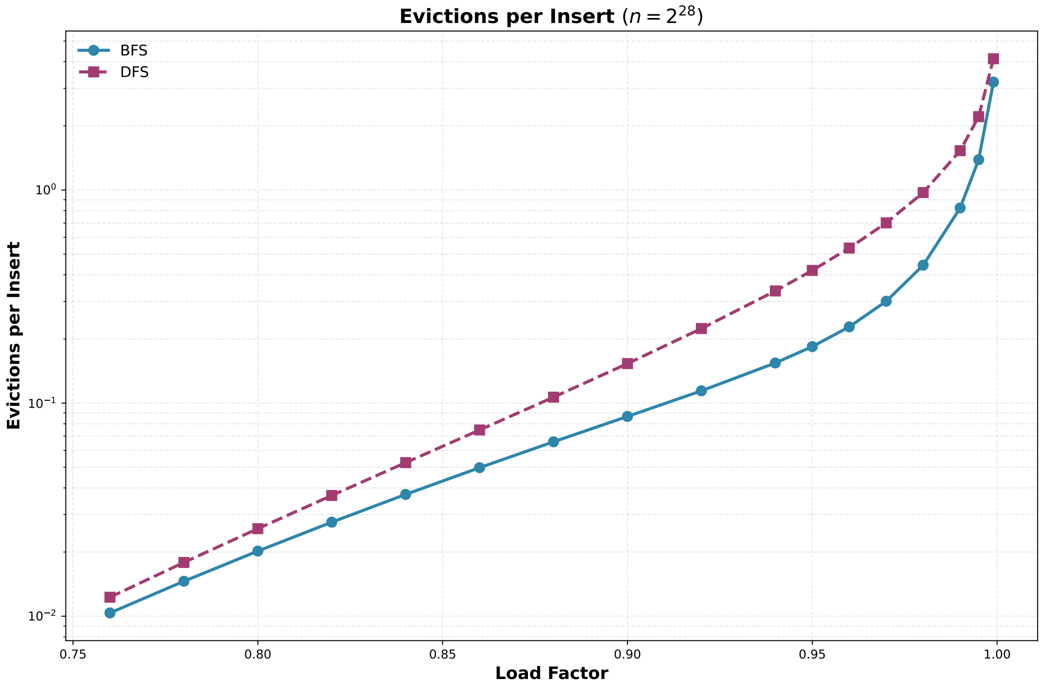


Fig. 4.5: Average number of evictions per insertion on System B.

4.4.3 Performance Analysis

The throughput impact of the BFS policy presents a trade-off between computational complexity and memory bandwidth efficiency (avoiding global memory accesses caused by evictions).

L2-Resident Workloads

In the L2-resident scenario (Figure 4.6), the standard DFS policy consistently outperforms the BFS policy across all systems.

In this scenario, the latency penalty of an eviction (loading a new bucket) is minimal. At the same time, the BFS policy incurs a higher instruction overhead per step because it must perform atomic checks on up to half the slots within the loaded buckets. In this bandwidth-abundant, low-latency environment, the computational cost of these extra checks outweighs the savings from reduced evictions, making the simpler greedy approach faster.

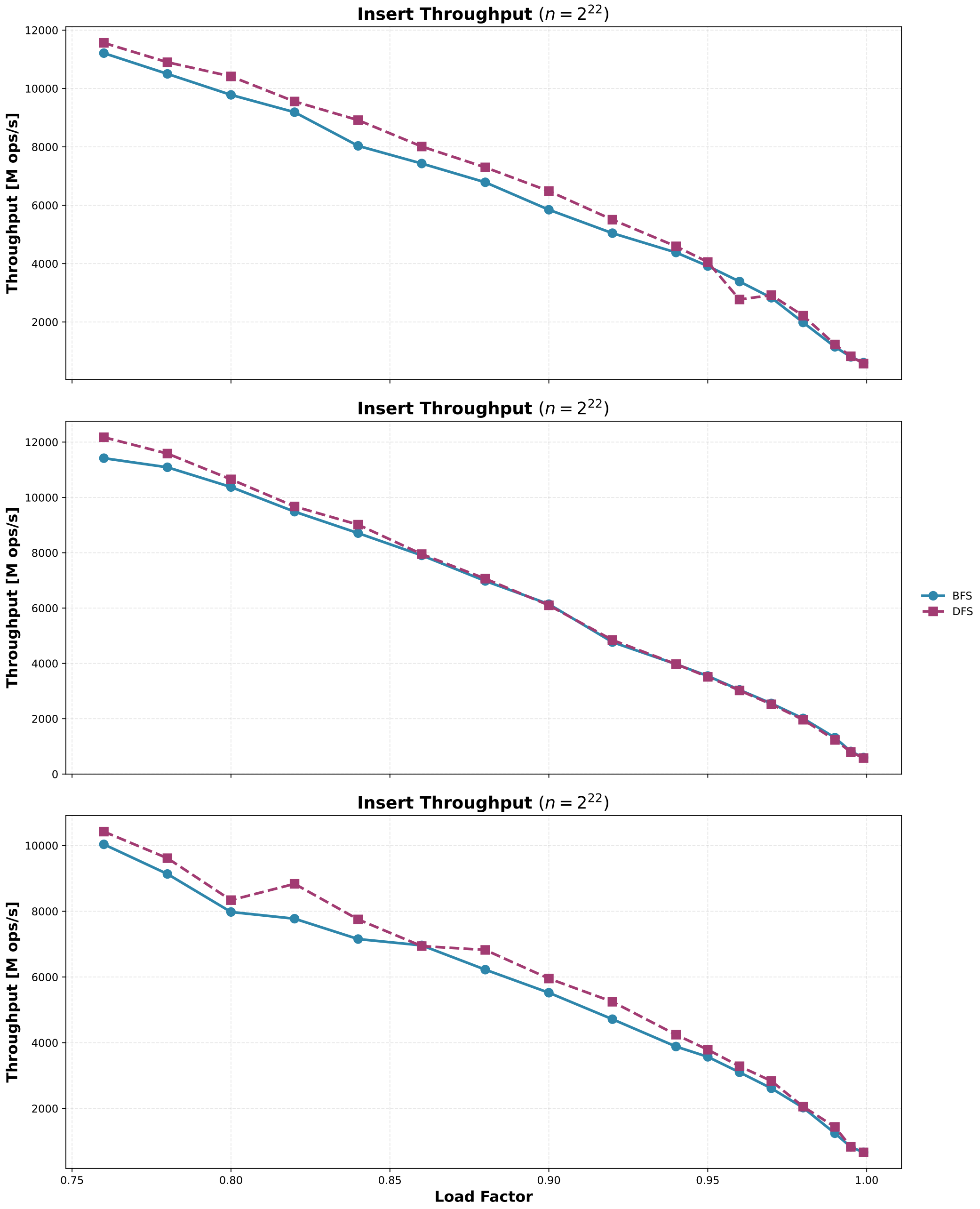


Fig. 4.6: Insert Throughput on System A (Top), System B (Middle) & System C (Bottom) (L2-Resident).

DRAM-Resident Workloads

For large filters that exceed the L2 cache, the performance dynamics become highly dependent on the specific balance between compute resources and memory subsystems. Figure 4.7 illustrates the throughput comparison on all three systems.

- **System A**: On the RTX Pro 6000, the BFS policy offers negligible benefits. This system features the highest core count of all testbeds, and its massive parallelism allows the GPU to effectively hide the latency of the sequential DFS memory accesses by aggressively switching between warps. Because the hardware is already mitigating the penalty of the DFS chains via parallelism, the BFS strategy yields barely any gain.
- **System B**: The HBM3-based GH200 shows the strongest relative improvement from the BFS policy. Despite having massive bandwidth, the standard DFS policy is bottlenecked by the latency of serialized pointer chasing. The BFS policy breaks this dependency chain by resolving collisions locally using the already-loaded cache lines. Since System B has fewer cores than System A to hide this latency, converting the serial memory stalls into local compute work results in a significant throughput increase ($\approx 25\%$).
- **System C**: On the consumer-grade GDDR7 card, the BFS policy is consistently faster. Unlike System A, this GPU lacks the extreme core count required to fully hide DFS latency and unlike System B, it lacks the massive bandwidth to brute force the problem. Here, the BFS policy wins simply by conservation of bandwidth. As shown in Section 4.4.2, BFS significantly reduces the number of evictions. On a bandwidth-constrained device, avoiding these extra memory transactions translates directly to higher throughput.

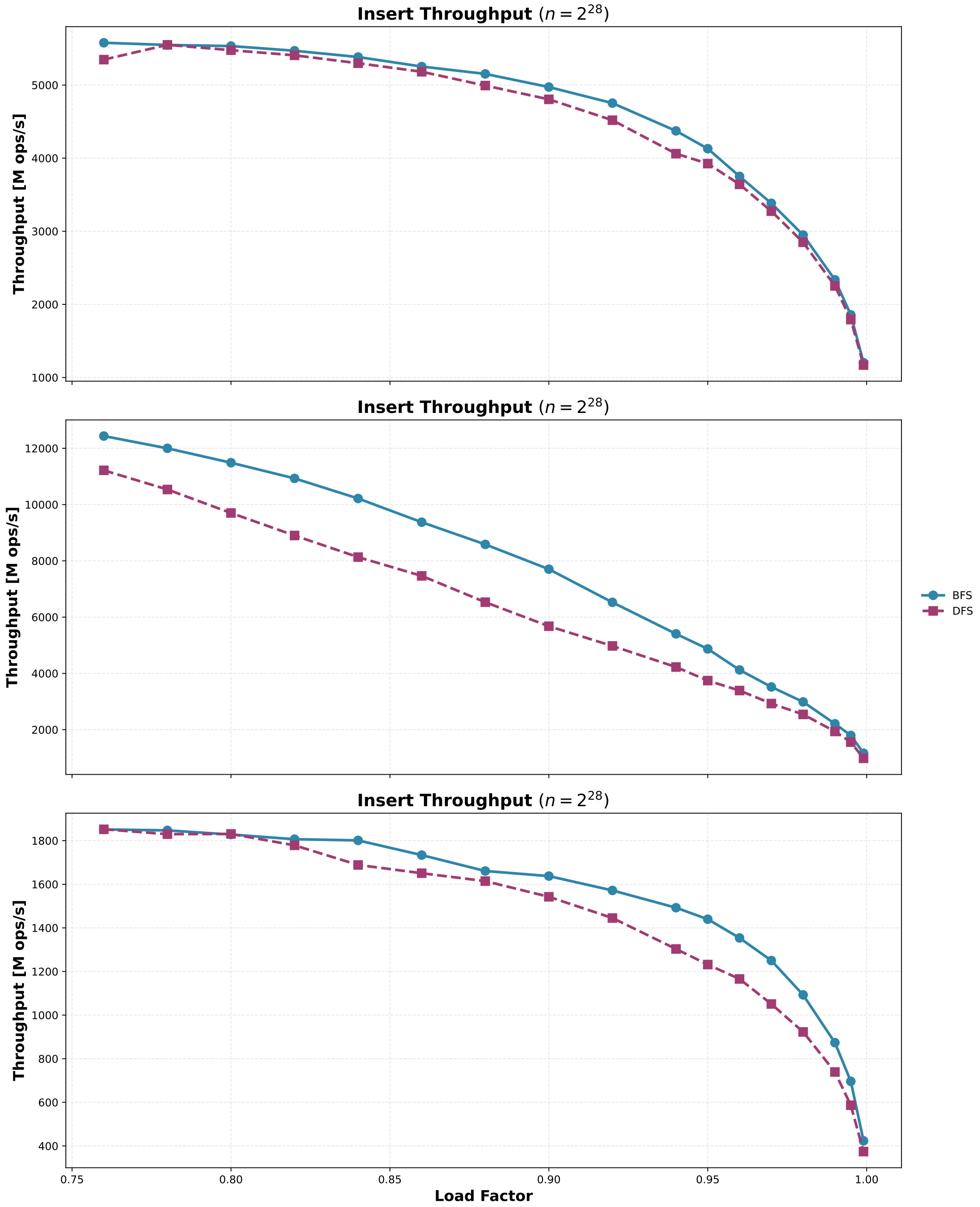


Fig. 4.7: Insert Throughput on System A (Top), System B (Middle) & System C (Bottom) (DRAM-Resident).

4.5 Sorted vs. Unsorted Insertion

To investigate the potential for enhancing insertion throughput by improving memory locality, an alternative insertion strategy was implemented and evaluated. This strategy, detailed in Section 3.4.1, pre-sorts the input keys by their primary bucket index before launching the insertion kernel. The goal is to maximize coalesced memory accesses by ensuring that adjacent threads target the same or nearby buckets.

To strictly isolate the benefits of memory locality from the computational overhead of the sorting algorithm, a *Presorted* metric was also introduced. This metric measures the throughput of the insertion kernel assuming the data has already been sorted "for free", whereas *Sorted* reflects the end-to-end throughput including the radix sort and packing overhead.

4.5.1 Performance Characteristics

The performance impact of sorting was evaluated across all three test systems. Figures 4.8, 4.9, and 4.10 illustrate the throughput comparison between the standard unsorted approach and the sorted variants across a wide range of input batch sizes.

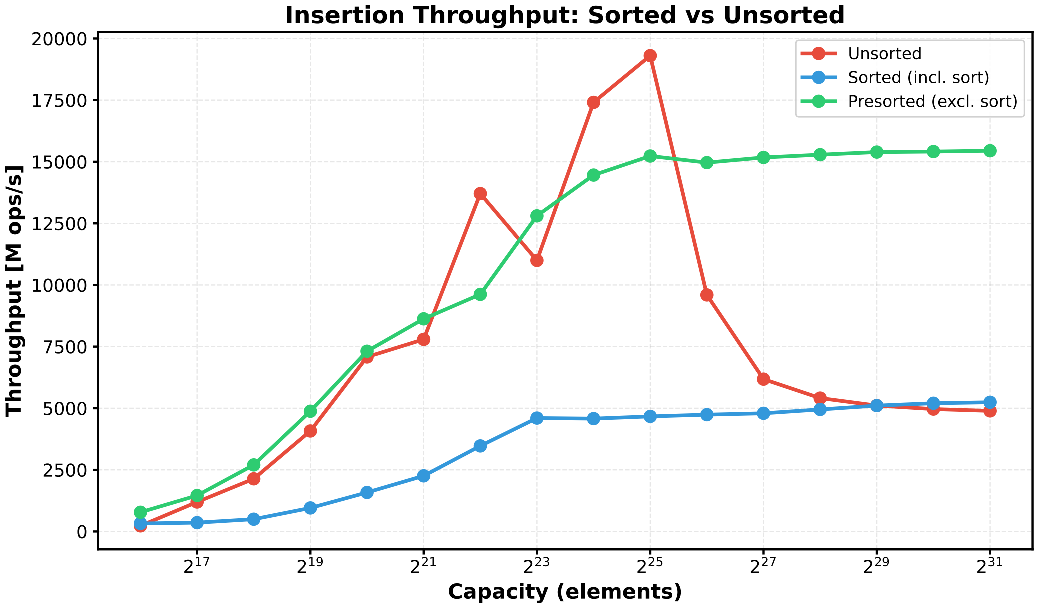


Fig. 4.8: Throughput comparison of sorted vs. unsorted insertion on System A (GDDR7).

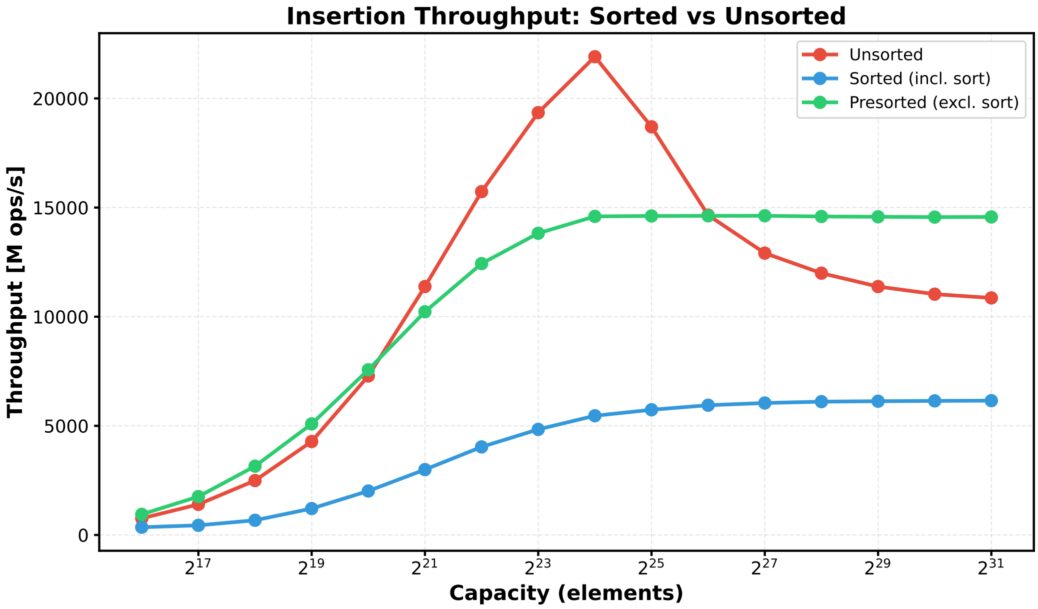


Fig. 4.9: Throughput comparison of sorted vs. unsorted insertion on System B (HBM3).

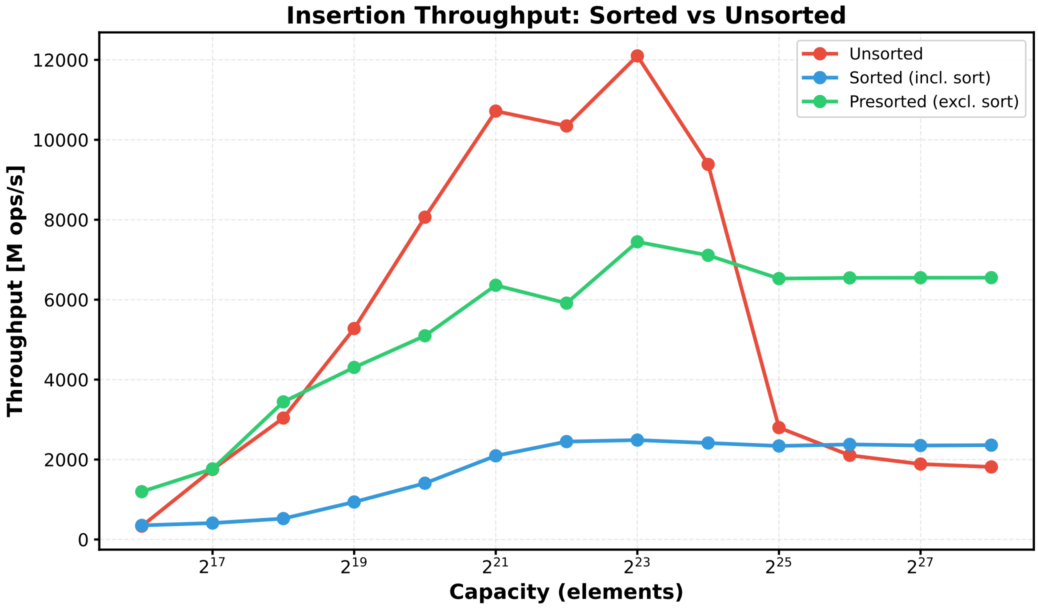


Fig. 4.10: Throughput comparison of sorted vs. unsorted insertion on System C (GDDR7).

Impact on Throughput Curve

The comparison reveals a significant divergence between the cost of sorting and the benefit of locality:

- **Insertion Efficiency (Presorted)**: The *Presorted* case shows that improved memory locality effectively mitigates the performance penalty usually associated with DRAM-resident filters. While the unsorted throughput collapses as soon as the filter exceeds L2 cache capacity, the presorted insertion maintains a significantly higher and more stable throughput profile.
- **The Sorting Tax**: The substantial gap between the *Presorted* and *Sorted* lines highlights the extreme cost of the pre-processing step. For small, L2-resident inputs, this overhead makes the sorted approach non-viable. However, for large DRAM-resident inputs, the cost of sorting can be justified through the avoidance of random memory access stalls

Architecture-Specific Behavior

The speedup provided by pre-sorting depends heavily on the memory technology and bandwidth availability of the GPU.

- **GDDR7 Systems (A & C)**: On systems relying on GDDR7, the benefits of pre-sorting are massive. In the DRAM-resident scenario ($> 2^{25}$ elements), the pure insertion kernel (excluding sort) achieves approximately 3x higher throughput compared to the unsorted baseline. This confirms that random memory access latency is the primary bottleneck on these architectures. However, when the sorting overhead is included, the net throughput only matches or slightly exceeds the unsorted baseline at maximum capacity.
- **HBM3 System (B)**: On System B, the high-bandwidth memory already absorbs much of the random access penalty inherent to the unsorted approach. Consequently, while pre-sorting still yields a performance improvement, the speedup is less dramatic. Because the speedup factor is smaller, it fails to amortize the cost of the sort, resulting in a net loss in total throughput (including sort) compared to the standard unsorted method.

4.5.2 Trade-offs and Limitations

The breakdown of sorted vs. unsorted performance leads to several conclusions regarding viability:

- **Overhead vs. Gain**: While memory locality theoretically solves the random-access bottleneck, the current cost of the sorting and packing step is too high to realize these gains in an end-to-end pipeline. Unless the sorting cost can be amortized over other operations, the standard unsorted insertion remains superior for general use.

- **Memory Overhead**: The sorting process requires auxiliary buffers for keys and indices, doubling the peak memory consumption during insertion. This effectively halves the maximum batch size.

4.6 Impact of Wider Memory Loads

With the introduction of the NVIDIA Blackwell architecture (Compute Capability 10.0), new PTX instructions allow for wider memory transactions. To evaluate the impact of instruction-level parallelism on query throughput, a special case utilizing the 256-bit non-coherent load instruction was added to the lookup kernel:

```
ld.global.nc.v4.u64 {%0, %1, %2, %3}, [%4];
```

This instruction fetches four `uint64_t` values in a single operation, bypassing the L1 cache to access the L2 cache or global memory directly. This reduces the total number of issued instructions required to fetch bucket data and lowers pressure on the instruction pipeline.

4.6.1 Throughput Analysis

Figure 4.11 compares the query throughput of the standard 128-bit load implementation against the optimized 256-bit variant on System A (Blackwell architecture).

The results highlight two distinct scenarios:

- **L2-Resident Improvement**: When the filter fits into the L2 cache (up until 2^{25} elements), the 256-bit implementation provides a consistent performance uplift of up to 18%. By fetching entire buckets with fewer instructions, the kernel reduces execution overhead, allowing the device to better saturate the L2 bandwidth.
- **DRAM-Resident Convergence**: As the working set spills into global memory, the performance advantage disappears. In this case, the workload becomes strictly bound by DRAM access times. Regardless of whether the data is requested via 128-bit or 256-bit instructions, the memory controller is already saturated, and the instruction issue rate is no longer the bottleneck.

This shows that while the use of 256-bit vector loads offers a "free" performance boost for L2-resident workloads on supported hardware by improving instruction efficiency, it does not fix the memory bandwidth bottleneck of large, DRAM-resident Cuckoo filters.

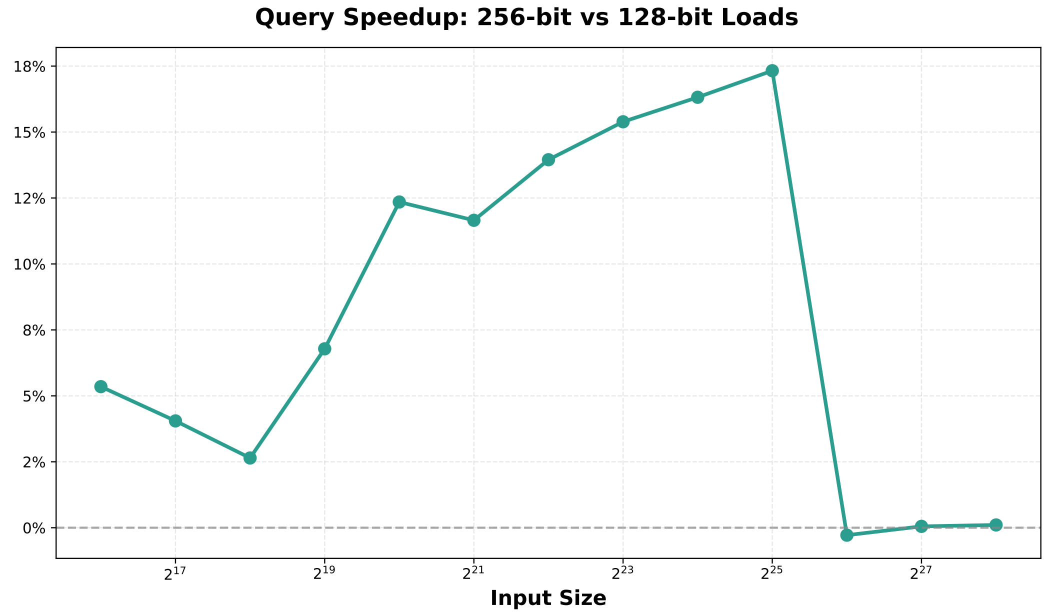


Fig. 4.11: Query speedup going from 128-bit to 256-bit loads on System A.

4.7 False Positive Rate

4.7.1 Empirical Accuracy Analysis

To evaluate the reliability of the implemented filters, the empirical false-positive rate was measured across a range of filter capacities. For each test, the filters were populated to a constant 95% load factor using random keys. The total memory size was varied from 2^{15} to 2^{30} bytes, allowing each implementation to optimize its internal layout within that fixed memory constraint. The results are presented in Figure 4.12:

- **Blocked Bloom Filter**: The Blocked Bloom filter demonstrates the highest false-positive rate among all tested structures, ranging from approximately 0.5% to over 4%. This is a known characteristic of the blocked design: partitioning the bit array into small, fixed-size blocks prevents the "averaging" of hash collisions across the entire filter. Consequently, a few heavily congested blocks can disproportionately skew the overall error rate. It is notably the only filter where the false-positive rate degrades this much as the total memory size increases.

- **Quotient Filter Accuracy**: The GQF exhibits the lowest false-positive rate among all candidates, maintaining an error rate between 0.001% and 0.002%. This confirms the theoretical space efficiency of quotient filters, which handle collisions via Robin Hood hashing and metadata encoding.

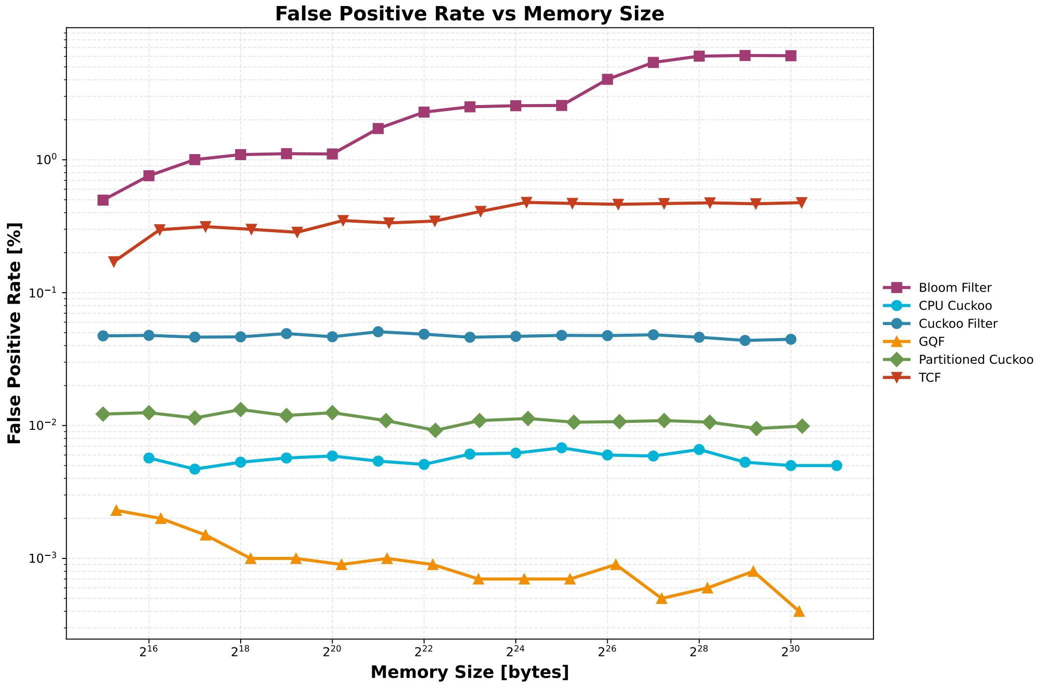


Fig. 4.12: Comparison of False Positive Rates (FPR) versus total memory size for various filter implementations at a 95% load factor.

- **CPU vs. GPU Cuckoo Filters**: A distinction remains visible between the CPU and GPU Cuckoo filter implementations. The CPU version achieves a very low false-positive rate, hovering near 0.005%. The GPU Cuckoo filter, while still highly accurate, exhibits a slightly higher rate of approximately 0.045%. This difference is a direct consequence of the bucket size trade-off discussed in Section 4.3. To maximize parallel throughput, the GPU implementation uses a bucket size of 16, whereas the CPU versions use a standard bucket size of 4. As established in Equation 2.8, a larger bucket size directly increases the collision probability for a fixed fingerprint size. On top of that, the partitioned Cuckoo filter exhibits a similar clustering effect to the Blocked Bloom filter, albeit to a much lesser degree.
- **Comparison with TCF**: The GPU Cuckoo filter significantly outperforms the TCF regarding accuracy. The TCF exhibits an error rate roughly an order of magnitude higher (ranging between 0.2% and 0.5%). While the TCF is more accurate than the Blocked Bloom filter, the Cuckoo filter and GQF designs offer superior accuracy for this workload.

4.7.2 Performance vs. Accuracy Trade-off

In real-world applications, filters are often chosen based on strict False Positive Rate requirements (e.g. $\leq 1\%$ or $\leq 0.1\%$). To evaluate how the filters perform under these constraints, a parameter sweep was conducted to identify the configuration that yields the highest throughput for a given target FPR.

These tests were performed on System B (HBM3) with capacities of 2^{22} (L2-resident) and 2^{28} (DRAM-resident) slots. The CPU implementations are excluded from this comparison as their throughput is simply insufficient. To represent a realistic workload where the presence of items is uncertain, the query throughput is reported as a weighted average of positive and negative lookups, with a 50% hit rate.

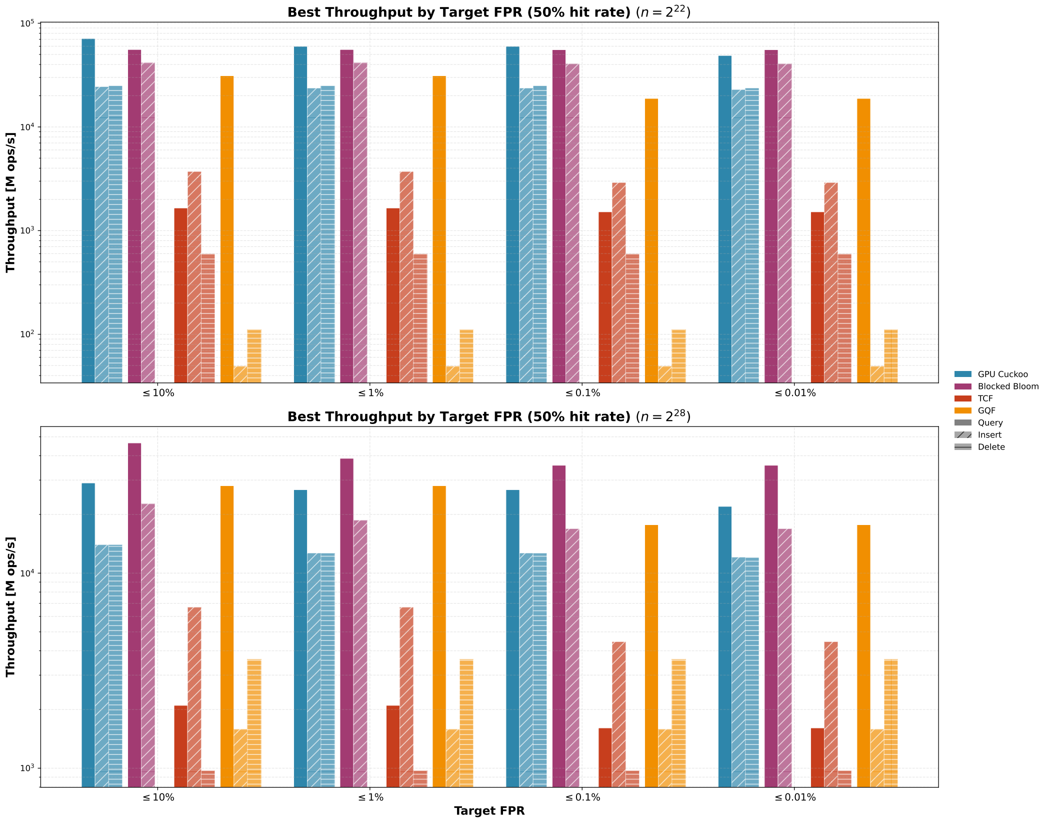


Fig. 4.13: Maximum achievable throughput for Insert, Query, and Delete operations while adhering to strict False Positive Rate targets. Top: L2-resident, Bottom: DRAM-resident. Note the logarithmic scale on the Y-axis.

The results, visualized in Figure 4.13, highlight the strengths of the Cuckoo filter as a general-purpose structure:

- **L2-Resident Superiority**: In the L2-resident scenario, the GPU Cuckoo filter demonstrates exceptional efficiency. For target FPRs of $\leq 10\%$, $\leq 1\%$ and $\leq 0.1\%$, it actually surpasses the Blocked Bloom filter in query throughput. This indicates that when memory latency is neutralized by the cache, the Cuckoo filter's retrieval logic is computationally more efficient than the multiple hashing and bitwise operations required by the Blocked Bloom filter. Furthermore, the performance gap between the Cuckoo filter and the other dynamic filters (TCF, GQF) widens significantly in this scenario, as the latter struggle to fully utilize the L2 bandwidth due to their complex internal synchronization.

- **Baseline Comparison**: As expected for the DRAM-resident case, the Blocked Bloom filter generally offers the highest raw throughput due to its linear memory access patterns. However, its inability to support deletions rules it out for dynamic applications. Among the dynamic filters, the Cuckoo filter remains the closest competitor to the Bloom filter's performance across all sizes.
- **Dynamic Operation Performance**: While the GQF can match the Cuckoo filter in query throughput for specific configurations, it suffers a catastrophic penalty in dynamic operations. Its insertion and deletion throughputs are orders of magnitude lower due to the complex synchronization required to shift elements. Similarly, the TCF, while faster than the GQF for updates, still lags significantly behind the Cuckoo filter in deletion throughput.
- **Conclusion**: The GPU Cuckoo filter emerges as the most robust and well-rounded solution for high-throughput dynamic workloads. It is the only tested data structure capable of maintaining high performance for insertions, lookups, *and* deletions simultaneously, while satisfying strict accuracy constraints.

4.8 Cache Efficiency

To understand how each filter interacts with the GPU's memory hierarchy, the L1 and L2 cache hit rates were measured by profiling the relevant kernels using NVIDIA Nsight Compute (ncu) on System B. While cache hit rate is not a direct proxy for overall throughput, it provides crucial insight into the memory access patterns and architectural bottlenecks of each implementation. The results for Insert, Query, and Delete operations are presented in Figure 4.14.

4.8.1 L1 Cache Analysis

A consistent trend across all operations is the exceptionally high L1 hit rate (near 100%) for both the TCF and the GQF, though they achieve this through different mechanisms.

The TCF relies heavily on shared memory for internal computations. It utilizes cooperative groups to load blocks into shared memory and perform intra-block sorting and coordination. Since shared memory physically occupies the same space as the L1 cache on modern NVIDIA architectures, these accesses are recorded as L1 hits.

In contrast, the GQF does not explicitly utilize shared memory but achieves high L1 efficiency through extreme spatial locality. The implementation assigns individual threads to manage specific, contiguous regions of the filter. As a thread performs operations such as linear probing or shifting elements within a run, it repeatedly accesses the same range of memory addresses. This ensures that the relevant cache lines remain resident in L1, resulting in a near-perfect hit rate despite operating directly on global memory structures.

The Cuckoo filter exhibits a moderate L1 hit rate (typically between 30% and 60%). This reflects its reliance on atomic operations on global memory addresses. While some metadata or repeated accesses may hit in L1, the random nature of hashing means that successive memory requests from a warp rarely map to the same L1 cache line, preventing the high hit rates seen in the locality-optimized implementations.

4.8.2 L2 Cache Analysis

The L2 cache hit rates provide a clear visualization of the transition from a cache-resident workload to a DRAM-resident workload. For the Cuckoo filter and the Blocked Bloom filter, the L2 hit rate remains high (approximately 80-90%) for smaller capacities. However, a sharp decline is observed once the filter size exceeds 2^{24} elements. This inflection point roughly corresponds to the physical L2 cache size of the test GPU. The steep drop-off confirms that beyond this point, every operation effectively incurs a global memory access, explaining the shift in performance scaling discussed in Section 4.2.

The GQF exhibits similar behaviour for its Lookup operation. Because a GQF lookup involves linearly scanning a cluster of slots in memory, it relies heavily on the L2 cache to minimize latency. Once the filter grows too large, these linear scans result in frequent cache misses, aligning its curve with that of the Cuckoo and Bloom filters.

In contrast, the TCF and GQF (specifically for Insertion and Deletion) maintain a consistently high L2 hit rate across all filter sizes. This stability indicates that these algorithms interact with global memory far less frequently than the Cuckoo or Bloom filters. Instead, they perform the vast majority of their work, such as sorting items within a block or managing cooperative groups, using internal registers and shared memory. While this results in high cache statistics, it confirms the architectural analysis from the previous section: these filters are bound by the speed of the GPU's compute units and shared memory (SRAM), preventing them from utilizing the abundant DRAM bandwidth available on modern High-Bandwidth Memory systems.

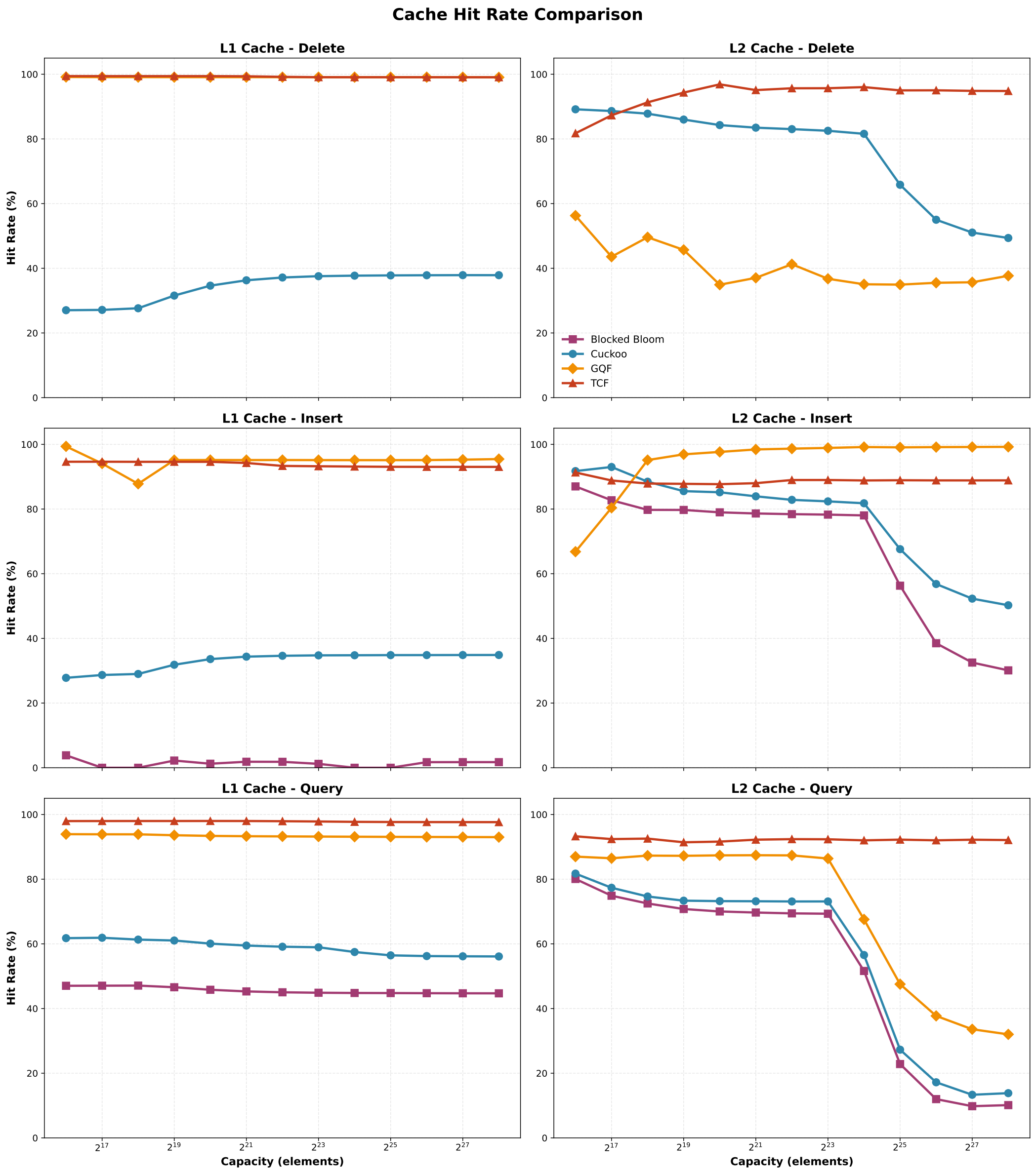


Fig. 4.14: L1 and L2 cache hit rates for the various operations across varying filter capacities.

4.9 Hardware Utilization

To validate the architectural hypotheses regarding memory-bound versus compute-bound behaviour, the resource utilization of each filter was profiled using NVIDIA Nsight Compute just like in Section 4.8. This analysis measures the achieved throughput as a percentage of the GPU's theoretical peak ("Speed of Light") for three critical subsystems: Compute (SM), Cache (L1/L2), and Global Memory (DRAM). The results are presented in Figures 4.15, 4.16, 4.17, and 4.18.

4.9.1 Compute Utilization

The SM throughput metrics, shown in Figure 4.15, reveal distinct execution characteristics for the different algorithms. The Cuckoo and Blocked Bloom filters exhibit a characteristic "hump" profile. When the filter fits within the L2 cache (up to 2^{24} elements), compute utilization rises steadily as the execution is dominated by hash calculations and bitwise manipulation instructions. However, once the capacity exceeds the L2 limit, compute utilization drops sharply. This decline occurs because the SMs begin to stall while waiting for data from global memory, shifting the primary bottleneck from instruction throughput to memory latency.

In contrast, the TCF shows a steady increase in compute utilization as the filter grows, eventually reaching high levels of SM saturation (up to 80%). This confirms that the TCF is primarily compute-bound, spending the majority of its cycles executing cooperative group logic and sorting operations within shared memory rather than waiting on external memory. The GQF stands out for consistently low compute utilization, particularly for insertion and deletion. This suggests it is bottlenecked by neither pure compute throughput nor memory bandwidth, but likely by serialization latency within threads, such as branch divergence or dependency stalls.

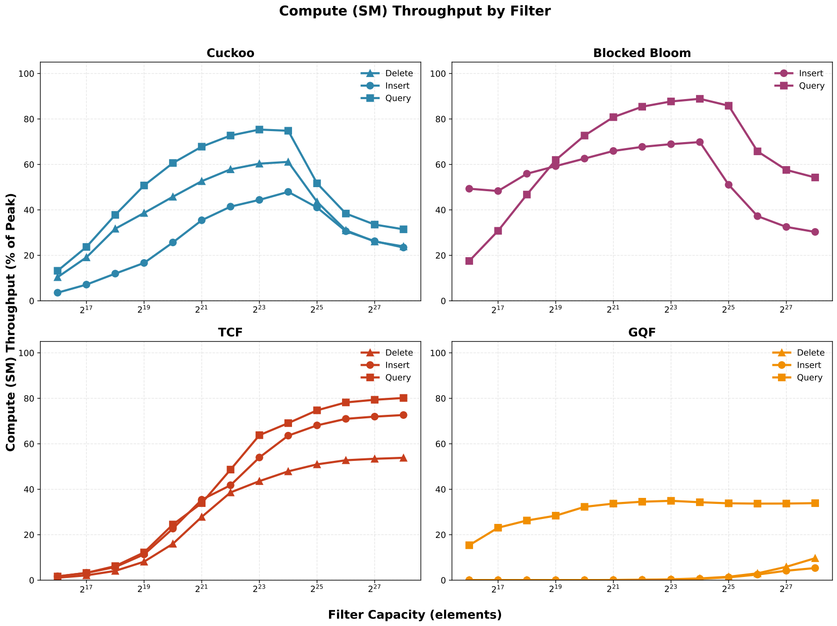


Fig. 4.15: Compute (SM) throughput as a percentage of peak performance on System B.

4.9.2 Cache Throughput

The L1 and L2 cache throughput metrics (Figures 4.16 and 4.17) mirror the cache hit rate findings from Section 4.8.

The Cuckoo and Blocked Bloom filters effectively utilize cache bandwidth up to the L2 capacity limit, after which throughput declines as requests miss to DRAM. In contrast, the TCF and GQF maintain slowly increasing cache throughput regardless of filter size, reinforcing that their working set for active operations remains largely resident in shared memory/L1.

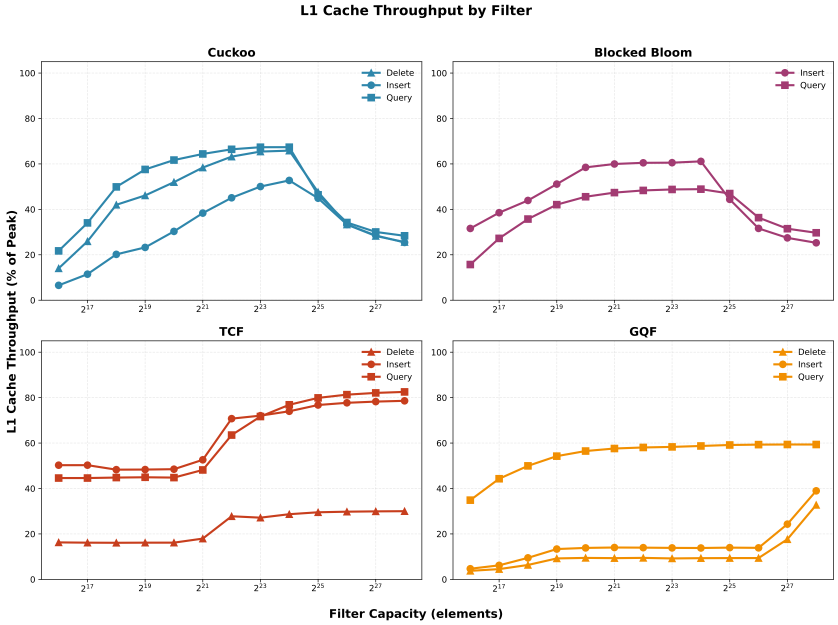


Fig. 4.16: L1 throughput as a percentage of peak performance on System B.

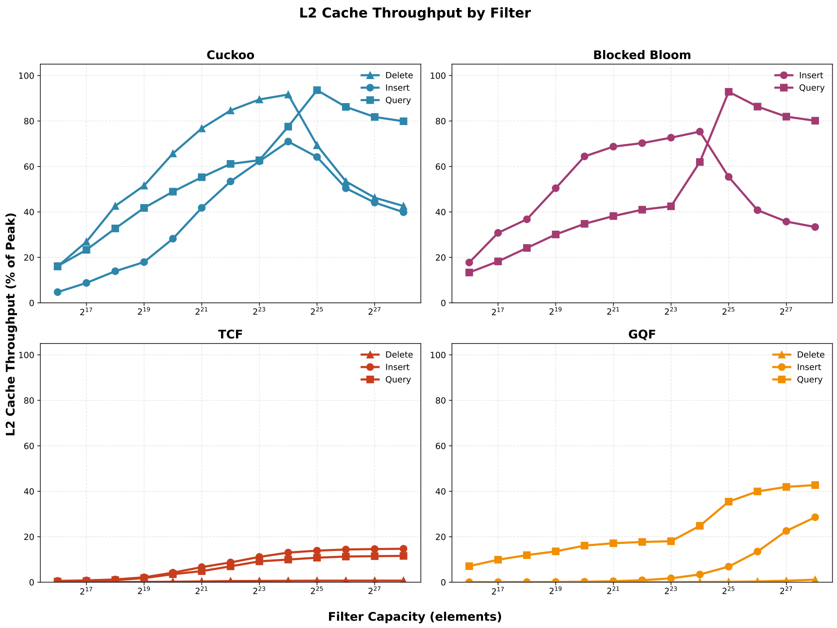


Fig. 4.17: L2 throughput as a percentage of peak performance on System B.

4.9.3 DRAM Throughput

The DRAM throughput results, presented in Figure 4.18, provide definitive confirmation of the scaling characteristics discussed in Section 4.2. For the Cuckoo and Blocked Bloom filters, DRAM utilization jumps significantly once the filter size exceeds the L2 cache limit. Notably, the Cuckoo filter's insert operation utilizes nearly 35% of the peak DRAM bandwidth, while query operations reach over 60%. This confirms that these algorithms are truly memory-bound for large datasets. Consequently, their performance is directly tied to the available memory bandwidth, ensuring they will continue to benefit from future hardware advancements like HBM3e and HBM4.

At the same time, the TCF and GQF show negligible DRAM utilization (near 0%) for insertion and deletion operations, regardless of filter size. This effectively proves that these algorithms are unable to utilize the available global memory bandwidth. Their performance is strictly limited by the speed of the on-chip memory (SRAM). As a result, they are less likely to scale with future improvements in DRAM technology compared to the memory-hungry Cuckoo filter.

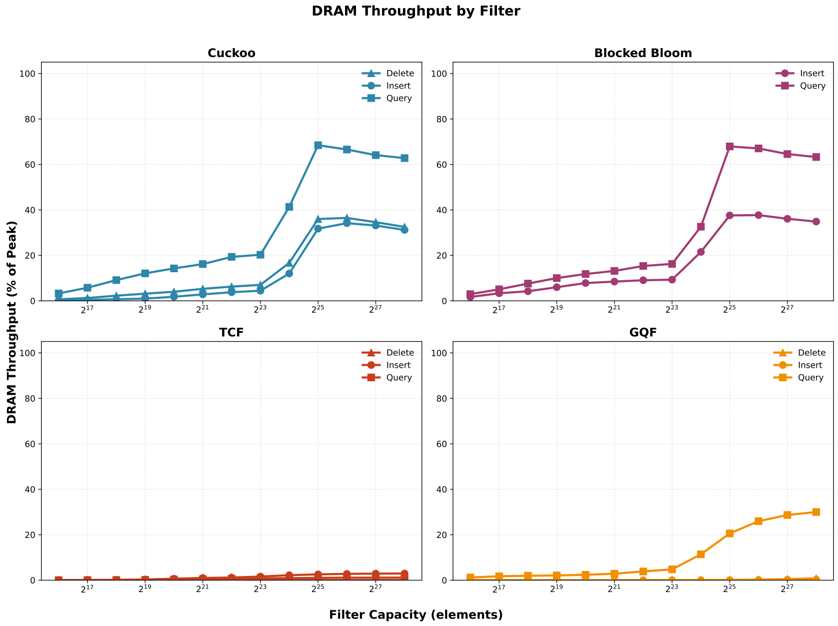


Fig. 4.18: Global Memory (DRAM) throughput as a percentage of peak performance on System B.

4.10 Scalability (Multi-GPU)

To evaluate the filter's ability to scale beyond a single device, a specific set of benchmarks was conducted on a multi-GPU server node. These tests focus on two scalability metrics: Strong Scaling (speeding up a fixed-size problem) and Weak Scaling (maintaining performance as problem size grows with hardware).

4.10.1 Experimental Setup

For the evaluation of multi-device scalability, the experimental setup was expanded with a fourth hardware configuration representing a high-end enterprise cluster node:

- **System D (HBM2e/NVLink)**: An AMD EPYC 7713 (64 cores) paired with $8\times$ NVIDIA A100-SXM4 GPUs, each featuring 80 GB of HBM2e memory ($\approx$ 2 TB/s bandwidth). The GPUs are interconnected via NVLink 3.0, providing a total aggregate bandwidth of 600 GB/s. The system runs AlmaLinux 8.10 with NVIDIA driver 580.95.05 and CUDA 12.8.61.

Unlike the PCIe-based tests performed on Systems A, B, and C, this configuration allows the Gossip library to leverage the high-bandwidth NVLink interconnect for the All-to-All data exchange, significantly reducing the communication latency during the partitioning phase.

4.10.2 Strong Scaling

Strong scaling measures how the execution time changes as the number of GPUs increases while the total problem size remains fixed. Ideally, doubling the number of GPUs should halve the execution time. For this test, the total filter capacity was fixed at 2^{30} slots (approx. 1 billion items), distributed across 2, 4, 6, or 8 GPUs.

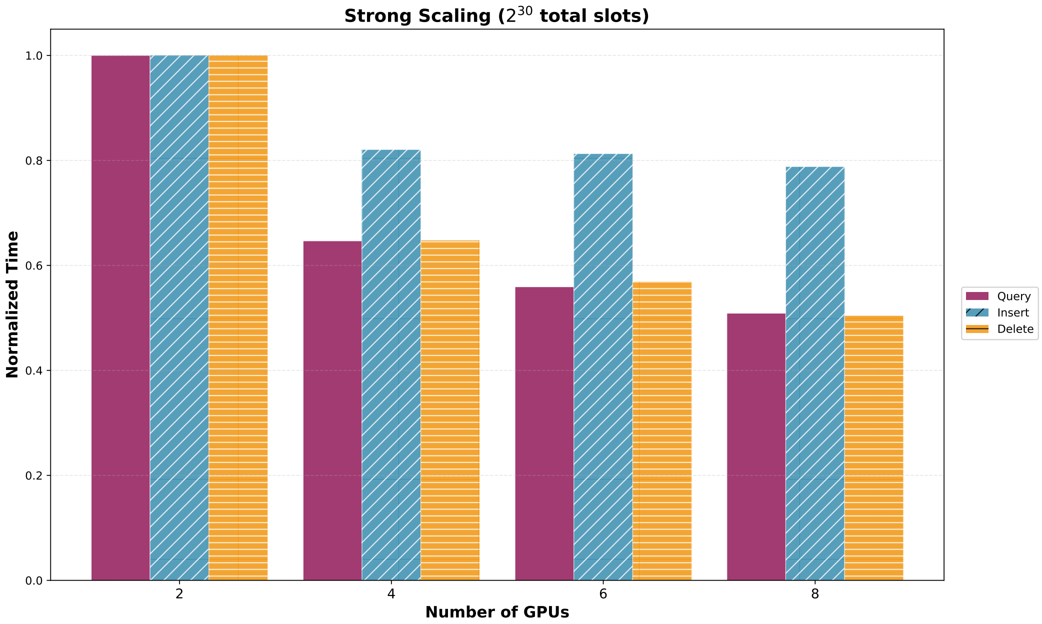


Fig. 4.19: Strong Scaling on System D with a fixed capacity. The Y-axis represents normalized execution time (lower is better), relative to the 2-GPU baseline.

The results in Figure 4.19 reveal distinct behaviours for different operations:

- **Query and Delete**: These operations show decent scaling. Moving from 2 to 8 GPUs reduces the execution time by approximately 50%. While this is a noticeable speedup, it is not linear (ideal scaling would yield a 75% reduction). This indicates that the latency of the All-to-All exchange via NVLink prevents perfect scaling when the work performed by each GPU decreases.

- **Insertion**: Insertion performance is largely flat, showing minimal improvement as GPUs are added. This suggests that for a fixed dataset of this size, the insertion process is dominated by the communication and partitioning phase (hashing keys, calculating destinations, and shuffling data via Gossip) rather than the insertion kernel itself. As more GPUs are added, the complexity of the All-to-All exchange increases, negating the benefit of parallelizing the insertion logic.

4.10.3 Weak Scaling

Weak scaling measures how throughput changes as the number of GPUs increases while the workload per GPU remains fixed. Ideally, the normalized throughput should scale linearly with the number of GPUs (e.g., 8 GPUs should provide $4\times$ the throughput of 2 GPUs). For this test, the capacity was fixed at 2^{30} slots per GPU, meaning the 8-GPU test processed a massive dataset of 2^{33} slots (over 8 billion items).

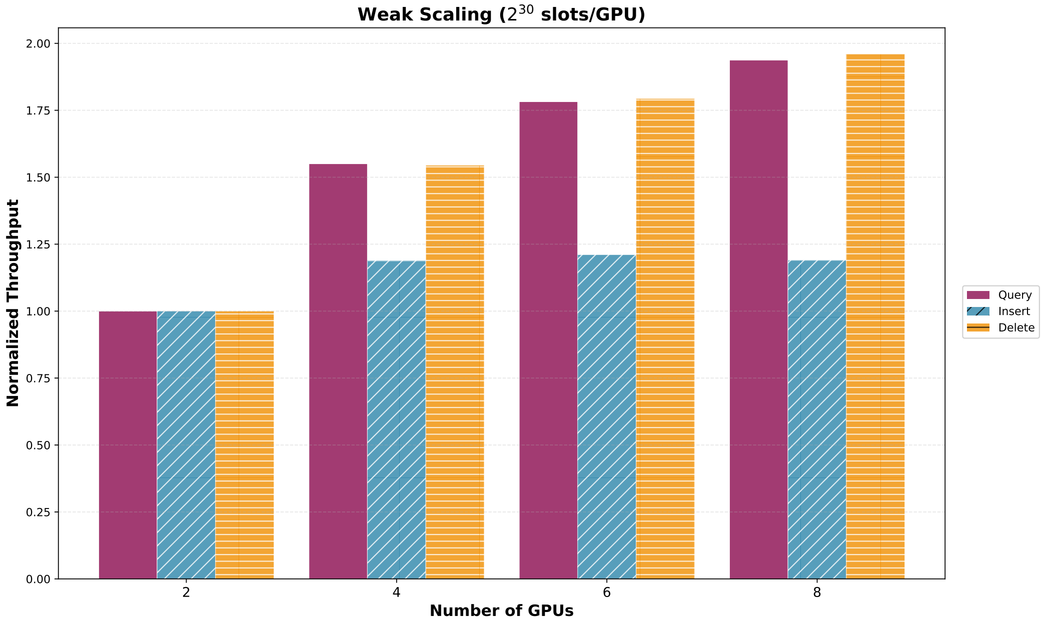


Fig. 4.20: Weak Scaling on System D with a fixed capacity of 2^{30} slots per GPU. The Y-axis represents normalized total throughput relative to the 2-GPU baseline.

Figure 4.20 illustrates the results:

- **Sub-linear Scaling**: While total throughput increases with more GPUs, it does not scale linearly. At 8 GPUs (processing $4\times$ the data of 2 GPUs), the system achieves roughly $1.95\times$ the query/delete throughput and only $1.2\times$ the insertion throughput.
- **Communication Bottleneck**: This behaviour confirms that even with NVLink, the system is bottlenecked by the interconnect. In a distributed Cuckoo filter, every GPU must communicate with every other GPU. As the number of GPUs increases, the density of the All-to-All communication pattern grows, introducing latency that limits the aggregate throughput.

4.10.4 Conclusion

The multi-GPU implementation effectively enables the processing of massive datasets that exceed single-device memory limits (up to 8+ billion items on

8 A100s). However, it is primarily a capacity scaling solution rather than a throughput scaling solution. For datasets that fit within a single GPU, the single-GPU implementation remains far more efficient due to the lack of communication overhead.

4.11 Real-World Benchmark: Genomic K-mer Indexing

While synthetic benchmarks using uniformly distributed integers are useful for showing algorithmic behaviour, real-world data can often present challenges due to skewed distributions. To validate this, a benchmark was conducted on genomic k-mer indexing as a primary use case for approximate membership query structures in bioinformatics.

4.11.1 Background and Experimental Setup

A k-mer is a subsequence of length k derived from a biological sequence, such as DNA. In genomic analysis, counting and filtering k-mers is an important step for tasks like genome assembly and error correction. Since DNA consists of four base pairs (A, C, G, T), the total number of possible k-mers is 4^k, which grows rather quickly.

For this evaluation, the T2T-CHM13v2.0 [30, 23] dataset is used as the first complete sequence of a human genome. The raw FASTA [28] data was preprocessed using KMC 3 [19] to extract all distinct 31-mers (k = 31). To optimize memory usage and processing speed, the text-based k-mers were packed into a 2-bit-per-base binary representation, compressing the dataset roughly by a factor of four and allowing each 31-mer to fit within a single `uint64_t`.

The resulting dataset is approximately 20 GB in size (packed), which is sufficient to effectively saturate the 96 GB of VRAM on System B. All filters were tested for insertion, lookup, and deletion (when supported).

4.11.2 Performance Analysis

The throughput results for the k-mer benchmark are presented in Figure 4.21.

The results confirm that the Cuckoo filter's high performance translates well to real-world workloads:

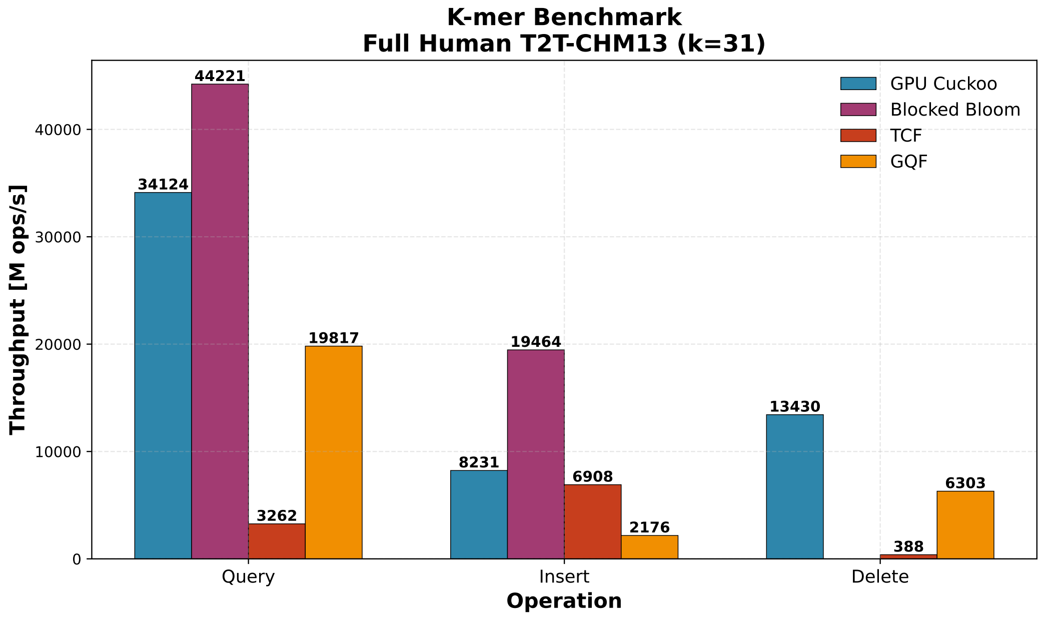


Fig. 4.21: Throughput comparison for inserting, querying, and deleting 31-mers from the full T2T-CHM13 human genome on System B. The Cuckoo filter shows a robust balance of performance across all three operations.

- **Query Performance**: While the Cuckoo filter does not fully match the Blocked Bloom filter, it maintains a significant lead over other dynamic alternatives. It is approximately 72% faster than the GQF and roughly 10× faster than the TCF. This indicates that the Cuckoo filter's bucketed lookup strategy remains highly efficient even with the specific distribution of genomic data.
- **Insertion Performance**: While the Cuckoo filter trails the append-only Blocked Bloom filter in this high-bandwidth scenario, it remains the fastest among dynamic data structures, outperforming the TCF by 19% and the GQF by 3.8×. This reinforces the observation that the GQF's locking overhead is a severe bottleneck for insertion workloads.
- **Deletion Performance**: In deletion throughput, the Cuckoo filter demonstrates superior performance. It is 2.1× faster than the GQF and nearly 35× faster than the TCF. This highlights the efficiency of the atomic-CAS deletion logic, which scales exceptionally well compared to the other filters' internal logic.

4.11.3 Conclusion

This benchmark demonstrates that the GPU Cuckoo filter is a highly versatile data structure for real-world applications. While the Blocked Bloom filter remains the throughput leader for read-only or append-only tasks, the Cuckoo filter is the only tested solution that delivers a consistent high performance across *all* operations. This makes it particularly valuable for workloads that involve a good mix of insertions, lookups, and deletions.

4.12 Impact of Bucket Policies

As detailed in Section 3.4.3, the standard XOR-based partial-key cuckoo hashing imposes a strict power-of-two constraint on the number of buckets. While this allows for efficient bitwise arithmetic, it can lead to significant memory over-provisioning (up to $2\times$) for datasets that do not align with powers of two.

To evaluate the cost of flexibility, the standard XOR policy was benchmarked against the AddSub (Additive/Subtractive) and Offset (Choice-bit) policies on System B with a fixed load factor of 95%.

The results for both L2-resident and DRAM-resident scenarios are presented in Figure 4.22.

4.12.1 L2-Resident Performance

In the L2-resident scenario, the filters are primarily bound by instruction latency and cache bandwidth. The results show that the XOR policy maintains a notable performance advantage specifically for positive queries, where it achieves approximately 34% higher throughput than the alternatives.

The alternative policies require integer modulo operations to calculate bucket indices, whereas the XOR policy utilizes simple bitwise masking. In this low-latency environment, the computational overhead of these modulo instructions becomes visible. Given that memory capacity is rarely a bottleneck for small, cache-sized filters, the performance penalty of the alternative policies outweighs the benefit of flexible sizing. Therefore, for small datasets, the standard XOR policy remains the optimal choice.

4.12.2 DRAM-Resident Performance

In the DRAM-resident scenario, where performance is primarily dictated by global memory bandwidth, things change:

- **AddSub Performance**: The Additive/Subtractive policy is consistently outperformed by the other policies. It is approximately 20% slower for positive queries and slightly slower for the other operations. This suggests that its specific calculation overhead or register pressure interact poorly with the memory latency hiding mechanisms on the GPU.

- **Offset Policy Viability**: The Offset-based policy matches the performance of the XOR baseline almost perfectly across all operations. Because the workload is memory-bound, the additional compute cycles required for the offset calculation are effectively hidden by the DRAM latency.

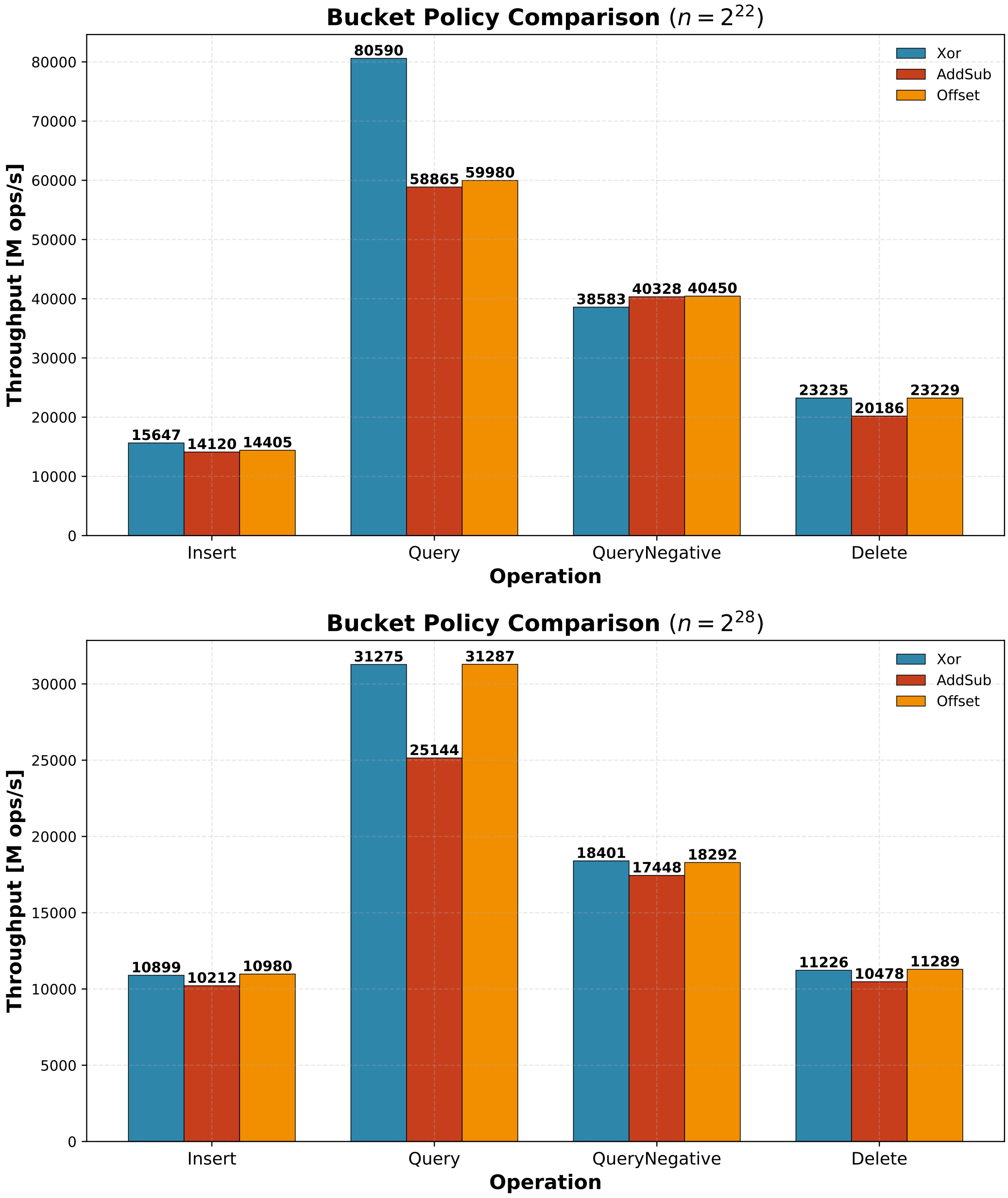


Fig. 4.22: Throughput comparison of the various Bucket Policies on System B.

4.12.3 Conclusion

For massive, memory-bound datasets, the Offset policy is a highly attractive alternative. It delivers throughput equivalent to the XOR baseline while eliminating the power-of-two restriction, allowing users to fit significantly larger datasets into a fixed VRAM budget (e.g., fitting a 5 GB filter into 6 GB of VRAM, which would require 8 GB with the XOR policy). This memory saving comes at the minor cost of a slightly increased false-positive rate (due to the choice bit reducing the effective fingerprint size by 1 bit), a trade-off that is often acceptable for maximizing capacity.

5 Conclusion and Future Work

5.1 Summary of Contributions

This thesis presented the design, implementation, and evaluation of a high-performance Cuckoo filter accelerated by Graphics Processing Units. The primary motivation was to address the performance gap between append-only probabilistic data structures, like the Bloom filter, and dynamic ones needed for high-churn applications.

The key contributions of this work are:

- **High-Performance CUDA Library**: A robust, header-only CUDA library was developed. This library encapsulates the parallel algorithms for insertion, lookup, and deletion, abstracting the complexity of GPU memory management and atomic synchronization. It provides a flexible, template-based C++ interface that allows users to configure critical parameters at compile time for maximum efficiency.
- **Architectural Optimization**: The Cuckoo filter was tuned for GPU hardware, identifying a bucket size of 16 as the optimal trade-off between memory bandwidth utilization and eviction complexity as well as supporting wider memory loads for the latest Blackwell GPUs.
- **System Extensions**: To facilitate real-world adoption, the filter was extended with an IPC wrapper for zero-copy sharing and a multi-GPU sharding mechanism to scale beyond the memory limits of a single device.
- **Rigorous Evaluation**: A comprehensive analysis was conducted across different memory technologies (GDDR7 and HBM3). The results demonstrate that the implementation successfully bridges the gap between append-only and dynamic filters, offering update capabilities orders of magnitude faster than existing alternatives while maintaining query throughput competitive with the Blocked Bloom filter.

5.2 Key Findings

The evaluation yielded a lot of insight regarding the interaction between AMQ data structures and modern GPU architectures:

- **Memory Bandwidth Scalability**: The Cuckoo filter is primarily memory-bound. Unlike the Two-Choice Filter or Quotient Filter, which are limited by shared memory latency and compute overhead, the Cuckoo filter's throughput scales linearly with global memory bandwidth. This makes it uniquely positioned to benefit from hardware advancements like HBM3e and HBM4.
- **The Cost of False Positives**: While the implementation achieves extremely high throughput, this comes at a cost. To maximize insertion speed, the filter utilizes larger buckets, which slightly increases the false-positive rate compared to CPU-based implementations (0.045% vs 0.005%). However, this trade-off allows the filter to handle billions of mutations per second, a feat unreachable by CPU-based alternatives.
- **Performance Parity**: In scenarios where the working set fits within the L2 cache, the GPU Cuckoo filter effectively matches the performance of the append-only Blocked Bloom filter. This invalidates the traditional assumption that one must sacrifice significant read performance to gain deletion capabilities.

5.3 Limitations

Despite the success of the implementation, certain limitations remain:

- **Insertion Performance at Capacity**: As the filter approaches maximum capacity ($> 95\%$), the insertion throughput degrades significantly due to long eviction chains. While the BFS eviction policy helps with this, the fundamental algorithmic bottleneck persists.
- **Insertion Failures**: Unlike a Bloom filter, which always accepts new items (at the cost of a rising false-positive rate), a Cuckoo filter has a non-zero probability of failing to insert an item entirely. This requires the consuming application to implement robust error handling, such as a fallback storage mechanism or a trigger to resize and rebuild the filter.
- **False Positive Rate Flexibility**: The Cuckoo filter's error rate is constrained to discrete steps determined by the available integer widths (8, 16, or 32 bits). This lacks the fine-grained control of a Bloom filter, where the error rate can be adjusted arbitrarily by changing the number of hash functions.

5.4 Future Work

The work presented in this thesis opens several avenues for future research and optimization:

- **Asynchronous APIs**: The current IPC mechanism is blocking. Future iterations could implement an asynchronous command queue, similar to `io_uring`, to allow clients to submit batches of requests without stalling, further maximizing GPU occupancy.
- **Variable-Length Fingerprints**: Investigating methods to support variable-length fingerprints within the fixed-bucket structure could allow users to fine-tune the space-accuracy trade-off without the coarseness of jumping from 16-bit to 32-bit tags.
- **Integration into High-Throughput Systems**: While micro-benchmarks demonstrate raw speed, the ultimate validation would be integrating the library into real-world systems characterized by high churn. Deploying the filter within network intrusion detection systems, streaming analytics pipelines, or GPU-accelerated database engines would provide valuable insight into its impact on end-to-end application latency and throughput.
- **Robustness Against Churn**: Future work could explore hybrid architectures, such as attaching a small "overflow stash" or secondary table, to catch items that fail the eviction chain. This would significantly improve stability and reliability for non-terminating, dynamic workloads.

Bibliography

[2]Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. “Balanced allocations (extended abstract)”. In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*. STOC ’94. Montreal, Quebec, Canada: Association for Computing Machinery, 1994, pp. 593–602 (cit. on p. 17).

[3]Michael A. Bender, Martin Farach-Colton, Rob Johnson, et al. “Don’t thrash: how to cache your hash on flash”. In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1627–1637 (cit. on p. 17).

[4]Burton H. Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Commun. ACM* 13.7 (July 1970), pp. 422–426 (cit. on p. 5).

[5]Andrei Broder and Michael Mitzenmacher. “Network Applications of Bloom Filters: A Survey”. In: *Internet Mathematics* 1.4 (2004), pp. 485–509. eprint: `https://doi.org/10.1080/15427951.2004.10129096` (cit. on p. 1).

[6]Pedro Celis, Per-Ake Larson, and J. Ian Munro. “Robin hood hashing”. In: *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*. SFCS ’85. USA: IEEE Computer Society, 1985, pp. 281–288 (cit. on p. 17).

[7]Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. “Bigtable: A distributed storage system for structured data”. In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26 (cit. on p. 2).

[8]Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. “Extendible hashing—a fast access method for dynamic files”. In: *ACM Transactions on Database Systems (TODS)* 4.3 (1979), pp. 315–344 (cit. on p. 8).

[9]Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. “Cuckoo Filter: Practically Better Than Bloom”. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’14. Sydney, Australia: Association for Computing Machinery, 2014, pp. 75–88 (cit. on pp. 10, 11, 21, 38).

[10]Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. “Summary cache: a scalable wide-area Web cache sharing protocol”. In: *IEEE/ACM Transactions on Networking* 8.3 (2000), pp. 281–293 (cit. on pp. 2, 6).

[11]Edward A Fox, Lenwood S Heath, Qi Fan Chen, and Amjad M Daoud. “Practical minimal perfect hash functions for large databases”. In: *Communications of the ACM* 35.1 (1992), pp. 105–121 (cit. on p. 8).

[12]Michael L. Fredman, János Komlós, and Endre Szemerédi. “Storing a Sparse Table with 0(1) Worst Case Access Time”. In: *J. ACM* 31.3 (June 1984), pp. 538–544 (cit. on p. 8).

[13] Antonio Sérgio Cruz Gaia, Pablo Henrique Caracciolo Gomes de Sá, Mônica Silva de Oliveira, and Adonney Allan de Oliveira Veras. "NGSReadsTreatment–a Cuckoo Filter-based tool for removing duplicate reads in NGS data". In: *Scientific reports* 9.1 (2019), p. 11681 (cit. on p. 2).

[14] Afton Geil, Martin Farach-Colton, and John D. Owens. "Quotient Filters: Approximate Membership Queries on the GPU". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 451–462 (cit. on p. 18).

[15] Jan Grashöfer, Florian Jacob, and Hannes Hartenstein. "Towards application of cuckoo filters in network security monitoring". In: *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE. 2018, pp. 373–377 (cit. on p. 2).

[16] Kun Huang and Tong Yang. "Additive and Subtractive Cuckoo Filters". In: *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. 2020, pp. 1–10 (cit. on pp. 21, 30).

[17] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016 (cit. on p. 15).

[18] Robin Kobus, Daniel Jünger, Christian Hundt, and Bertil Schmidt. "Gossip: Efficient Communication Primitives for Multi-GPU Systems". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP '19. Kyoto, Japan: Association for Computing Machinery, 2019 (cit. on p. 33).

[19] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. "KMC 3: counting and manipulating k-mer statistics". In: *Bioinformatics* 33.17 (May 2017), pp. 2759–2761. eprint: `https://academic.oup.com/bioinformatics/article-pdf/33/17/2759/49040995/bioinformatics_33_17_2759.pdf` (cit. on p. 65).

[20] Zhe Li and Kenneth A Ross. "Perf join: An alternative to two-way semijoin and bloomjoin". In: *Proceedings of the fourth international conference on Information and knowledge management*. 1995, pp. 137–144 (cit. on p. 2).

[21] Hunter McCoy, Steven Hofmeyr, Katherine Yelick, and Prashant Pandey. "High-Performance Filters for GPUs". In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. PPoPP '23. Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 160–173 (cit. on pp. 16–18, 38).

[22] James K. Mullin. "Optimal semijoins for distributed database systems". In: *IEEE Transactions on Software Engineering* 16.5 (2002), pp. 558–560 (cit. on p. 2).

[23] Sergey Nurk, Sergey Koren, Arang Rhie, et al. "The complete sequence of a human genome". In: *Science* 376.6588 (2022), pp. 44–53. eprint: `https://www.science.org/doi/pdf/10.1126/science.abj6987` (cit. on p. 65).

[24] NVIDIA Corporation. *cuCollections*. Version 3b9873a. Oct. 2025 (cit. on p. 38).

[26]Rasmus Pagh. “On the cell probe complexity of membership and perfect hashing”. In: *Proceedings of the thirty-third annual ACM symposium on Theory of computing*. 2001, pp. 425–432 (cit. on p. 9).

[27]Rasmus Pagh and Flemming Friche Rodler. “Cuckoo hashing”. In: *Journal of Algorithms* 51.2 (2004), pp. 122–144 (cit. on p. 8).

[28]William R Pearson and David J Lipman. “Improved tools for biological sequence comparison.” In: *Proceedings of the National Academy of Sciences* 85.8 (1988), pp. 2444–2448 (cit. on p. 65).

[29]Pedro Reviriego, Jorge Martínez, David Larrabeiti, and Salvatore Pontarelli. “Cuckoo Filters and Bloom Filters: Comparison and Application to Packet Classification”. In: *IEEE Transactions on Network and Service Management* 17.4 (2020), pp. 2690–2701 (cit. on p. 2).

[30]Arang Rhie, Sergey Nurk, Monika Cechova, et al. “The complete sequence of a human Y chromosome”. In: *bioRxiv* (2023). eprint: `https://www.biorxiv.org/content/early/2023/07/11/2022.12.01.518724.full.pdf` (cit. on p. 65).

[31]Tobias Schmidt, Maximilian Bandle, and Jana Giceva. “A four-dimensional analysis of partitioned approximate filters”. In: *Proceedings of the VLDB Endowment* 14.11 (2021), pp. 2355–2368 (cit. on p. 38).

[32]Johanna Elena Schmitz, Jens Zentgraf, and Sven Rahmann. “Smaller and More Flexible Cuckoo Filters”. In: *arXiv preprint arXiv:2505.05847* (2025) (cit. on pp. 21, 30).

[33]Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. “Theory and practice of bloom filters for distributed systems”. In: *IEEE Communications Surveys & Tutorials* 14.1 (2011), pp. 131–155 (cit. on p. 1).

Webpages

[@1]Sean Eron Anderson. *Bit Twiddling Hacks*. Accessed: 2025-12-13. 2005. URL: `https://graphics.stanford.edu/~seander/bithacks.html` (cit. on pp. 22, 24).

[@25]NVIDIA Corporation. *CUDA C Programming Guide*. Accessed: 2025-10-20. 2025. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/` (cit. on pp. 13, 15).

List of Figures

Colophon

This thesis was typeset with LaTeX 2$_{\varepsilon}$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Adaptations to the style of the Institute of Computer Science can be found at `https://gitlab.rlp.net/institut-fur-informatik/cleanthesis-jgu`.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe (dazu zählen auch „KI-Werkzeuge“). Sämtliche wörtlichen und sinngemäßen Übernahmen und Zitate sind kenntlich gemacht und nachgewiesen. Auch beim Einsatz von KI-Werkzeugen[1] stellt die Arbeit klar dar, in welchem Umfang diese genutzt wurden und welche Inhalte dadurch erzeugt oder beeinflusst wurden[2]. Ich versichere, dass ich keine Hilfsmittel verwendet habe, deren Nutzung die Prüferinnen und Prüfer explizit ausgeschlossen haben.

Mit Abgabe der vorliegenden Leistung übernehme ich die Verantwortung für das eingereichte Gesamtprodukt. Ich verantworte damit auch alle KI-generierten Inhalte, die ich in meine Arbeit übernommen habe. Die Richtigkeit übernommener (KI-generierter) Aussagen und Inhalte habe ich nach bestem Wissen und Gewissen geprüft.

Ich habe die Arbeit nicht zum Erwerb eines anderen Leistungsnachweises in gleicher oder ähnlicher Form eingereicht. Von der Ordnung zur Sicherung guter wissenschaftlicher Praxis und zum Umgang mit wissenschaftlichem Fehlverhalten habe ich Kenntnis genommen.

Mir ist bekannt, dass ein Verstoß gegen die genannten Punkte prüfungsrechtliche Konsequenzen hat und insbesondere dazu führen kann, dass die Studien- und Prüfungsleistung als mit „nicht bestanden“ bewertet wird. Die Einschreibung kann für bis zu zwei Jahre widerrufen werden, wenn Studierende zweimal oder häufiger bei Prüfungsleistungen täuschen (§ 69 Abs. 4 und 5 HochSchG).

Mainz, February 5, 2026

Tim Dortmann

[1] Mit „KI-Werkzeugen“ werden computergestützte Systeme bezeichnet, die auf Basis maschinellen Lernens neue Inhalte generieren können (z. B. ChatGPT, Gemini, Claude, LLaMA, Midjourney, Stable Diffusion o. ä.).

[2] Die Nutzung von KI-Werkzeugen muss dann kenntlich gemacht werden, wenn Sie den Kern der Aufgabenstellung in von den Prüfern nicht anzunehmender Weise berührt: Sollte nichts anderes explizit mit den Prüferinnen und Prüfern vereinbart worden sein, so ist dies der Fall, sobald wesentliche Teile der eingereichten Arbeit (z. B. ganze Sätze des Textes, ganze Abbildungen, nicht-offensichtliche Teile von Rechnungen oder Beweisen, mehrere Zeilen von Programmcode am Stück) von solchen Werkzeugen generiert wurden und wörtlich oder in abgewandelter Form in die Arbeit übernommen wurden oder wesentliche Konzepte oder Ideen (z. B. eine Gliederung der Literatur oder ein Lösungsansatz für ein Problem) von KI-Werkzeugen generiert wurden.

Hinweis: Es wird im Falle von abweichenden Vereinbarungen mit den Prüferinnen und Prüfern empfohlen, jene im Vorfeld schriftlich festzuhalten und zusätzlich auch noch einmal in der eingereichten Arbeit zu benennen.

Use of AI tools.

AI Tool	Used for	Reason	When
GitHub Copilot	Refactoring	Focus on the important code	Throughout the implementation
Google Gemini	Rewriting notes	Improve readability	Throughout the thesis