Cuckoo Filter on the GPU

Tim Dortmann

Contents

- Approximated Membership Query Filters
- Why the Cuckoo filter
- How to parallelise to run efficiently on the GPU
- Benchmarks

AMQ Filters

- Imagine you have a (static) set of items
- Want to quickly check if something is part of said set
- Should be fast with minimal memory usage
- Often used in databases/distributed systems
- We’re okay with a few false positive
- Bloom filter as the “standard” solution
- What if we also want to support deletion?

Why the Cuckoo Filter

- Either better FPR at same space or less space for same FPR
- Only stores a fingerprint per item (n lowest bits of hashed item)
- Supports deletion
- Lookup is still guaranteed constant time
- In theory no false negatives
- GPU is interesting to support massive batches

Implemented Variants

- Hybrid Approach (GPU only for lookups, no buckets)
- CPU only (single-threaded)
- GPU only (this is where all the optimisations happened)

How does the filter work

- Most parts behave identically to a normal hash table
- Except we only store a “fingerprint” of the key

  => No rehashing, responsibility falls on the user
- Item can be in one of exactly 2 buckets
- Thanks to XOR you only need to know one bucket index

```cpp
template <typename H>
static __host__ __device__ uint32_t hash(const H& key) {
    auto bytes = reinterpret_cast<const cuda::std::byte*>(&key);
    cuco::xxhash_32<H> hasher;
    return hasher.compute_hash(bytes, sizeof(H));
}

static __host__ __device__ TagType fingerprint(const T& key) {
    return static_cast<TagType>(hash(key) & tagMask) + 1;
}

static __host__ __device__ cuda::std::tuple<size_t, size_t, TagType>
getCandidateBuckets(const T& key, size_t numBuckets) {
    TagType fp = fingerprint(key);
    size_t i1 = hash(key) & (numBuckets - 1);
    size_t i2 = getAlternateBucket(i1, fp, numBuckets);
    return {i1, i2, fp};
}

static __host__ __device__ size_t
getAlternateBucket(size_t bucket, TagType fp, size_t numBuckets) {
    return bucket ^ (hash(fp) & (numBuckets - 1));
}
```

How to parallelise (Querying)

- There is no dependency between threads
- Just find the two potential buckets and search through them
- Use multiple streams to hide latency and work on multiple chunks in parallel

How to parallelise (Insertion)

- The nature of the algorithms means we jump around like crazy (this is bad for obvious reasons)
- Each bucket has a spinlock for threads to acquire
- Use the tag to calculate an initial index in the bucket to improve performance
- As the table fills up the number of collisions grows rapidly

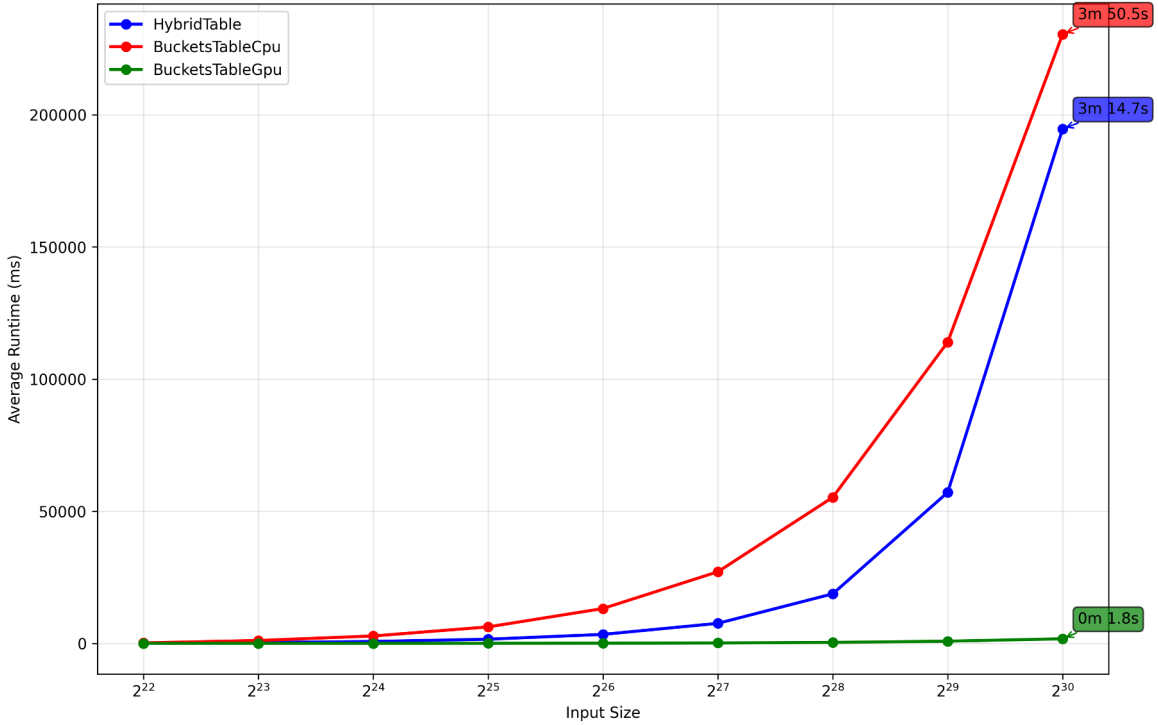
  => Kernel runs slower
- Can deal with this to an extent by again overlapping copy and compute

Benchmark Setup

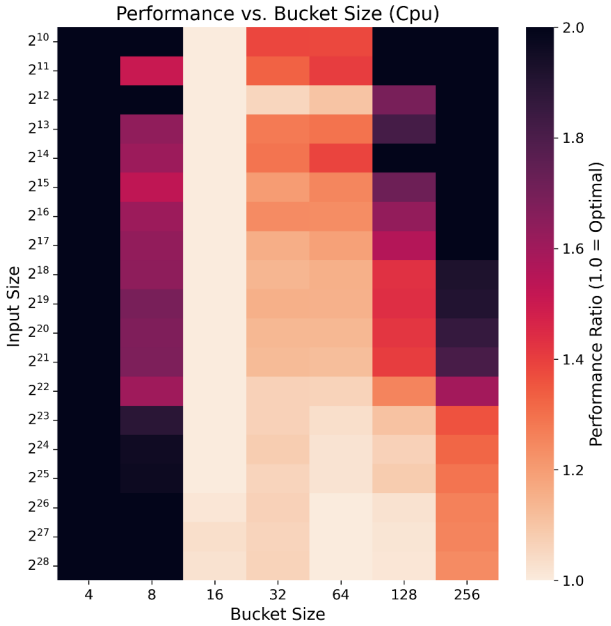
- AMD Ryzen Threadripper 3990X, NVIDIA 4090
- Generate a ton of random numbers on the host
- Insert them all into the table and then query them all
- Assert that results are as expected
- Several rounds of this process and take the average
- Deletion is implemented and works but not part of the benchmark

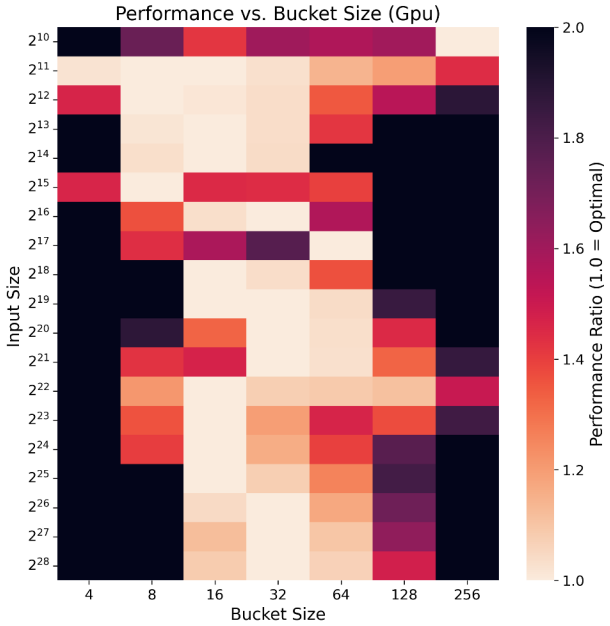
Runtime/Space Comparison

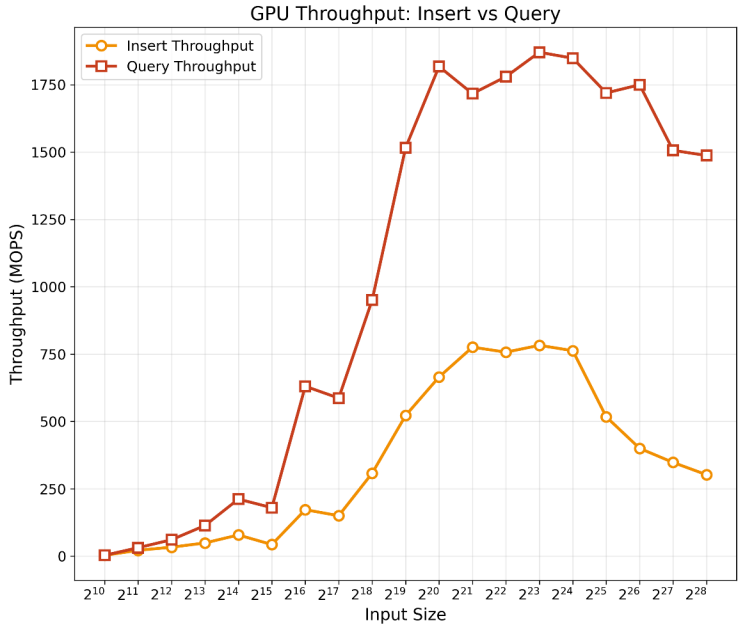
- The GPU only variant is significantly faster (110x and 130x)
- Space wise the hybrid one is by far the least efficient
- CPU variant more efficient space wise (no locks, no extra device view)
- Runtime variance is quite high (presumably due to the randomness)



Performance Analysis







Thank you for your attention!

Any questions?