Bloom Filters on UPMEM Data Processing Units

Abstract

Motivation. Bloom filters are a well-known probabilistic set data structure often used to compact data. In the context of sequencing, algorithms may rely on them to handle huge volumes of reads and perform computational analyses faster. Optimizing this underlying data structure, which is inherently memory-bound, is thus a major challenge to accelerate further genomics applications.

Results. We design a C++ Bloom filter implementation on the UPMEM Processing-in-memory architecture. Our evaluation shows it provides interesting speedups compared to an efficient Bloom filter implementation running on CPU. For instance, inserting and querying 100 million items in a 1 GiB filter executes 2-3 times faster. We present the implementation challenges we tackled and leverage our experience to discuss whether this type of data structure is a good fit for the UPMEM architecture.

Table of Contents

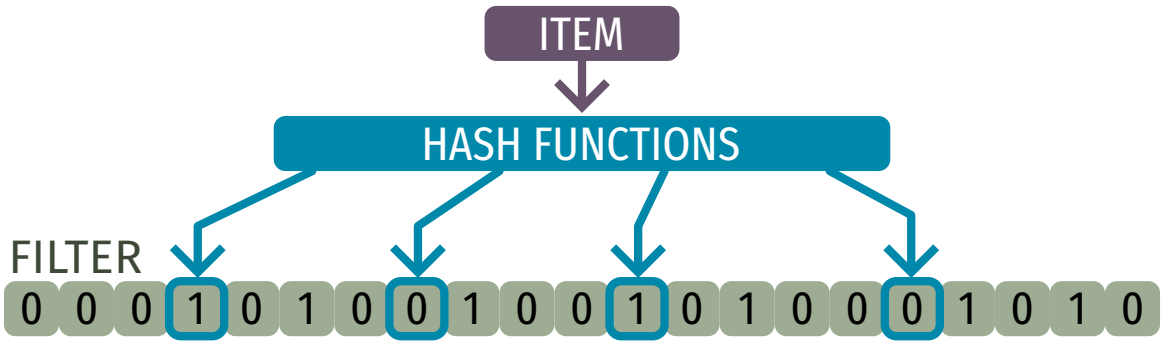


Figure 1: Hashing an item into a list of bit vector indexes

1 Introduction

Genomics applications deal with huge volumes of reads. Therefore, scientists often rely on compression as a preprocessing step. This permits to store data more easily and to run complex computational analyses faster. One way to produce compacted data is to rely on Bloom filters, a well-known probabilistic set data structure.

In this report, we focus on this specific data structure and describe our implementation on the UPMEM Processing-in-memory (PIM) architecture.

1.1 Bloom filters

Bloom filters [1] are a memory-efficient probabilistic set data structure. They support two operations: (i) inserting an element and (ii) querying the presence of an element (respectively Listings 1 and 2). A filter is a vector of bits of size m with all cells initialized to 0. It uses h uniform independent hash functions to hash any element into a list of h indexes as illustrated in Figure 1. The insertion operation sets the bits at the resulting indexes to 1. To query the filter, we look at the indexes and return a positive if all corresponding cells contain a 1. However, the lookup answer should be interpreted as either *No* or *Maybe Yes*. Hashing collisions can indeed lead to believe some elements were inserted in the filter when they were not. The false positive probability is given by the following formula, where n is the total number of elements inserted into the filter:

$$\mathbb{P}[\mathrm{FP}] \sim \left(1 - e^{-\frac{hn}{m}}\right)^h$$

On the other hand, false negative are impossible: $\mathbb{P}[\mathrm{FN}] = 0$.

Listing 1: Basic Bloom filter insertion

```
void insert(const uint64_t item) {
  for (int i = 0; i < h; i++) {
    bloom[hash(item, i)] = 1;
  }
}
```

Listing 2: Basic Bloom filter lookup

```
bool contains(const uint64_t item) {
  for (int i = 0; i < h; i++) {
    if (bloom[hash(item, i)] == 0) { return
        ↪ false; }
  }
  return true;
}
```

A Bloom filter variant, denoted Cache (Listings 3 and 4), uses the idea of blocks to keep the $h-1$ last hash indexes close to the first one as illustrated in Figure 2. Because a block is a lot smaller than the

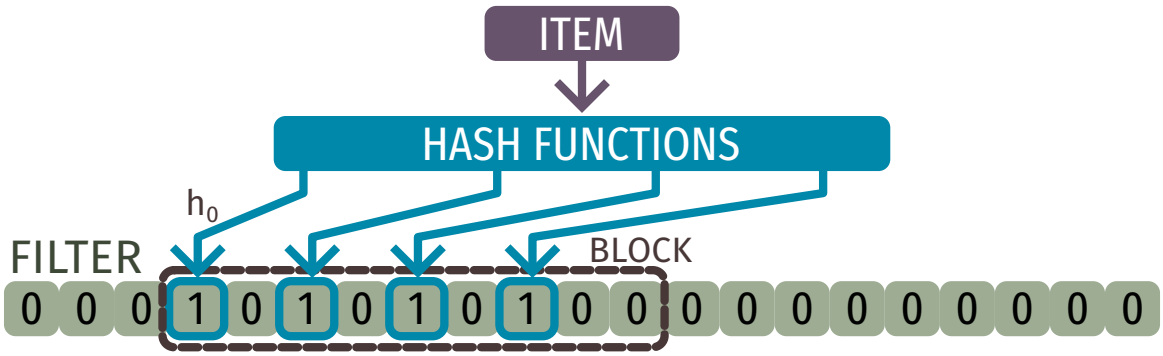


Figure 2: Using a block makes the Bloom filter more cache-friendly

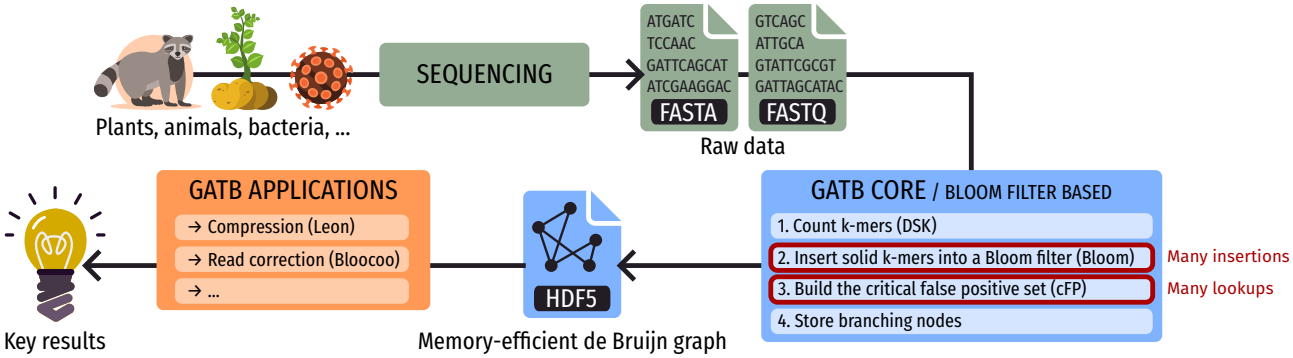


Figure 3: The Genome Analysis Toolbox with de Bruijn graph using Bloom filters

whole filter, simpler hash functions can be used to speed-up the computations. This implementation is more cache-friendly and thus executes faster, at the cost of a slightly increased false positive rate.

Listing 3: Cache Bloom filter insertion

```
void insert(const uint64_t item) {
  h0 = hash(item, 0);
  bloom[h0] = 1;
  for (int i = 1; i < h; i++) {
    h = h0 + (simple_hash(item, i) %
        ↪ BLOCK_SIZE)
    bloom[h] = 1;
  }
}
```

Listing 4: Cache Bloom filter lookup

```
bool contains(const uint64_t item) {
  h0 = hash(item, 0);
  if (bloom[h0] == 0) { return false; }
  for (int i = 1; i < h; i++) {
    h = h0 + (simple_hash(item, i) %
        ↪ BLOCK_SIZE)
    if (bloom[h] == 0) { return false; }
  }
  return true;
}
```

In practice, programs can only manipulate bytes and thus need to rely on bitwise masking to update individual bits. This could lead to race conditions if two threads set different bits within the same byte at the same time. Therefore, we can derive variants of these implementations that rely on an atomic write operation to perform insertions safely in a multithreaded environment.

1.2 Motivation

The Genome Analysis Toolbox with de Bruijn graph (GATB) [3] is an open source C++ library developed by the GenScale group from IRISA/CNRS. It provides a set of very efficient tools to analyze NGS datasets and uses compact data structures to run with a very low memory footprint. The library splits the process into two distinct parts, as illustrated in Figure 3:

1. The core module takes as input raw FASTA/FASTQ data and produces a memory efficient representation of the de-Bruijn graph stored in a HDF5 file. The library provides several representations. In particular, one of them relies on Bloom filters [2], and this is thus where we put our focus in this work.
2. Tools take the graph as input and perform various tasks like compression, assembly, read error correction, etc. The library provides a rich API to develop new tools on top of the core module.

To provide an exact graph representation despite the probabilistic nature of Bloom filters, the GATB library computes an adjacent structure to keep track of critical false positives. More precisely, computing the Bloom-based de-Bruijn graph executes the following steps:

Step 1: Count the abundance of all k-mers (DSK) [4].
Step 2: Insert the solid k-mers[1] into a Bloom filter.
Step 3: Build the critical false positive set.
Step 4: Compute branching information and store complex nodes in a hash table.

Steps 2 and 3 perform respectively a lot of insertions and lookups at once. Improving the elementary operations of Bloom filters when dealing with high volumes of data could thus speed-up significantly the graph construction.

With this concrete application in mind, we detail our goals and design choices in the next section. We then give technical details about our implementation on the UPMEM PIM architecture in Section 3. We review the performances of the library we developed in Section 4. Finally, we summarize our work and conclusions in Section 5. We also share some insights about the general characteristics of the Bloom filters data structure and how our investigations expose them to be well-fitted or not for the UPMEM PIM architecture.

2 Goal and Design Choices

Our goal consists in developing a C++ Bloom filter library on the UPMEM PIM architecture. We target a generic implementation where items are unsigned 64 bits integer. In the context of genomics, this permits to insert k-mers up until $k = 32$.

We focus on optimizing performances for high volumes of data. In particular, we wish to speed-up insertions and queries when dealing with a large input vector of items. The library still supports single insertions and queries through wrappers[2] but we expect them to be slow and irrelevant in the context of PIM architectures. We target applications such as the construction of a Bloom-based de-Bruijn graph presented in Section 1.2.

We report in Listing 5 the API of our library and give additional details about the different methods available:

- **Initialization**. We allow only filters of size being a power of 2. This permits some significant code optimization regarding modulo operations[3]. The number of threads parameter allows to run the host part of the program in multithreaded. We use ranks as the granularity level to manage a set of UPMEM Data Processing Units (DPUs). Ranks commonly contain 64 DPUs.

[1] We call solid the k-mers that appear more than a given threshold, usually 2 or 3. The goal of this process is to eliminate low-abundance k-mers that usually result from sequencing errors.

[2] Single insertions and lookups embed the input item into a vector of size one and call the bulk methods.

[3] If x is a power of 2, then the modulo can be replaced with a bitwise operation: $a \bmod x \equiv a \,\&\, (x-1)$.

- **Insertions**. As previously explained, we put our focus on the bulk operator that takes as input a—preferably large—vector of items.
- **Lookups**. The bulk operator returns a vector of booleans that gives the result of all lookups in the same order as the input vector of items.
- **Weight**. The weight is the proportion of bits set to 1 in the vector. This method is rarely used in practice, but we believe it is interesting to design and implement a version on PIM nonetheless.
- **Serialization**. We provide methods to retrieve and restore the underlying bit vector so that users can save the filter into a file and load it back later.

Listing 5: PIM Bloom filter C++ API

```
// Initialization
PimBloomFilter(size_t size2, size_t nb_hash, size_t nb_threads = 1, size_t nb_ranks = 8)

// Insertions
void insert(const uint64_t& item)
void insert_bulk(const std::vector<uint64_t>& items)

// Lookups
bool contains(const uint64_t& item)
std::vector<bool> contains_bulk(const std::vector<uint64_t>& items)

// Weight
size_t get_weight()

// Serialization
const std::vector<uint8_t>& get_data()
void set_data(const std::vector<uint8_t>& data)
```

3 Implementation Details

We now describe how we implemented Bloom filters on the UPMEM PIM architecture. First, we share an overview of the general process. Then, we will dive into the details of each part: the host on one side, the DPUs on the other side.

Instead of having one big filter in memory like we would have in a CPU implementation, we distribute the data structure and work with a set of many smaller filters. Each DPU stores and manages 16 filters in their MRAM.

When one calls a method of the library, the host is is charge of sending the necessary inputs to the DPUs. These consist of:

- A token. It indicates which method the DPUs will have to execute, for instance an insertion or a query.
- (Optional) An array of items. Since the data structure is split into many filters, we insert items into a specific fragment using a deterministic mapping. Each item is thus sent to only one DPU.
- (Optional) Other required parameters. For instance the size of the item array, or the number of hash functions, the filter size, etc.

All the input arguments are packed into a single array to minimize the number of transfers between the host and the DPUs.

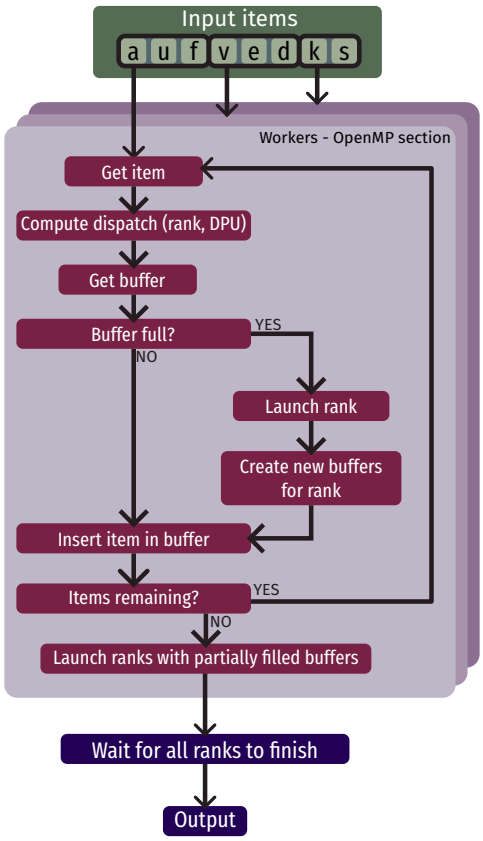


Figure 4: Dispatching the items on the host side

3.1 On the Host side

We use mainly asynchronous calls to manage the set of ranks of DPUs. This allows a better workload partition between the host and the DPUs to keep both busy as much as possible.

We focus on the details of the insertion and query functions only since they are more complex than the others and are the most used in practice. Besides, they required a lot of iterations and optimization to run efficiently. Most of host work consists of dispatching the items correctly to the set of DPUs. In the case of queries, it also handles getting the results back from the DPUs and formatting the output vector of booleans.

The approach we describe in the following sections is illustrated in Figure 4.

3.1.1 Insertions

We configure a set of W workers that execute in parallel on the CPU with an OpenMP section. We partition the input items into W fragments of same size. Each worker reads its own fragment and computes the mapping for each item. The latter is done with a 64 bits hash. We execute a modulo on the upper 32 bits to obtain a specific rank, and then a modulo on the lower 32 bits to target a specific DPU within this rank[4]. Once we know where this item should go, the worker inserts it in a buffer dedicated to this DPU.

[4] In practice, modulo operations are significantly expensive when occurring repeatedly. We use a faster alternative that replaces the modulo by a multiplication followed by a shift.

Buffers have a maximum size because of the limited memory we reserved in MRAM on the DPUs. Therefore, it happens that a worker tries to insert an item into a buffer that is full. In this case, the worker launches the execution of the whole rank with an asynchronous call, and creates a new set of empty buffers for this rank. Several rounds are likely to be necessary to handle the entire input vector of items.

Once the worker finished reading its items, it triggers the launch of all the remaining partially filled buffers. We then wait for all the ranks to finish processing before returning.

3.1.2 Lookups

Querying the filter follows the same approach, but exhibits two main differences.

Because we need to output the lookup results in a specific order, we maintain additional buffers to remember the index of items in the original vector. We could store the item and index at the same place with a tuple but it would waste space since DPUs do not need this information. That is why we keep it in a separate container.

Besides, we schedule an extra callback after the launch of a rank to transfer results back from the DPUs MRAM to the host. The latter then write the results at the right place in the output vector using the index buffers.

3.1.3 Discussion and Challenges

The dispatch approach we presented has two main parameters to tweak:

- The number of workers. The ideal is to have just enough workers to stack enough calls so that DPUs remain busy most of the time. Around 6 workers was a good number in most of our experiments.
- The maximum number of items in a buffer. The bigger, the less rounds are necessary. But the bigger, the longer it takes to fill them and trigger launches. A size of 2^{10} or 2^{11} offered a good balance in most of our experiments.

Regarding the host part, we encountered two main challenges when working on this implementation:

- Filling buffers is a very memory-bound process, as we could see using the VTune Profiler software [5].
- We need to make sure all variables stay alive long enough since the DPUs can still execute calls after the end of the parallel OpenMP section for workers. To this end, we store all buffers in containers. We however need to have generous memory reservations to ensure no reallocation ever happens. Otherwise this could invalidate some references given to the asynchronous DPU callbacks.

3.2 On the DPUs side

We configure the DPUs to run 16 tasklets in parallel. As previously mentioned, each DPU contains 16 filters in its MRAM. Let us now consider what happens when the host supplied the necessary input array and launched the execution. Similarly to the host part, we focus our explanations on the two set operations of a Bloom filter.

3.2.1 Insertions

We partition the input array of items into 16 parts. Each tasklet reads its own part sequentially using an intermediate cache array in WRAM. For each item, it computes a deterministic mapping to know in which of the 16 filters it should go. We use a cheap 16 bits hash to this end. Once the mapping identifier is computed, the tasklet locks the corresponding mutex in a pool of 16 mutexes. It then performs the insertion in a Cache filter manner as detailed in Section 1.1. Because all the bits to set are close to each other, we retrieve a small block of the filter starting at the first hash index in a cache in WRAM. The tasklet performs all the write operations in this cache. It then commits the insertion by writing the cache back in MRAM and releases the lock.

3.2.2 Lookups

Iterating over items works a bit differently for the lookups. Each tasklet will handle the queries for a specific filter. It reads every input item, computes the deterministic mapping and performs the query only if it corresponds to its identifier. After querying the filter, the tasklet writes the result in a cache of integers common to all tasklets. Since they all update different cells (results are ordered exactly like the input array of items), this does not require synchronization most of the time. We however need a barrier to let tasklet 0 store the results in MRAM and reset the cache once it is full. Before doing so, tasklet 0 compresses the data to store one result per bit. Compared to the sending of the input, this divides the transfer size from the DPUs to the host by 64. This compression significantly improves the overall execution time.

4 Evaluation

We now share how we evaluate the performances of our implementation of Bloom filters on the UPMEM PIM architecture.

4.1 Baseline

We compare our library to the best performing Bloom filter implementation of the GATB library: the Cache filter as presented in Section 1.1. We consider its synchronized version and use it with 8 threads.

Since it only provides single insertions and lookups, we extend the class to add bulk operators. The basic approach consists in looping over the input vector in parallel and applying the single operator on each item. We can however design a better algorithm. By using buckets, we reorder the items to gather those that will end up close to each other in the bit vector. This reordering takes some time but permits to then do the actual insertions or lookups in a much more cache-friendly manner. In our experiments, we notice this approach provides a significant speedup compared to the basic iterative version (around 2 to 3 times faster).

We thus select the bucket-based synchronized Cache filter as our baseline in this report.

4.2 Metrics and Benchmarks

Since our goal consists in accelerating the elementary operations of Bloom filters, the main metric for our evaluation is the elapsed time of each method. We implement a timer decorator to wrap each

function call to the filter interface and measure the elapsed time with the `omp_get_wtime()` method from the OpenMP module.

Additionally, we consider the false positive rate metric. Given N items that we know were not inserted into the filter, we compute the number of positive lookups we obtain by querying all of them. The false positive rate is then:

$$FPR = \frac{\#\text{<Maybe Yes>}}{N}$$

As N increases, the false positive rate can yield a good approximation of the false positive probability.

Listing 6: Bloom filter benchmark

```
std::vector<uint64_t> get_seq_items(const size_t nb, const uint64_t start_offset = 0) {
  std::vector<uint64_t> items(nb);
  for (size_t i = 0; i < nb; i++) { items[i] = i + start_offset); }
  return items;
}

// Creating vectors of items
std::vector<uint64_t> items = get_seq_items(nb_items); // Will be inserted
std::vector<uint64_t> no_items = get_seq_items(nb_no_items, nb_items); // Will not

// Creating filter (PIM or SyncCache)
auto bloom_filter = BloomFilterTimeitDecorator<PimBloomFilter<HashPimItemDispatcher>>(
    ↪ bloom_size2, nb_hash, nb_threads, nb_ranks);
// or
auto bloom_filter = BloomFilterTimeitDecorator<SyncCacheBloomFilter>(bloom_size2, nb_hash,
    ↪ nb_threads);

// Inserting many items
bloom_filter.insert_bulk(items);

// Computing weight
auto weight = bloom_filter.get_weight();

// Querying all inserted items in a random order
auto rng = std::default_random_engine{};
std::shuffle(std::begin(items), std::end(items), rng);
bloom_filter.contains_bulk(items);

// Querying non inserted items and computing false positive rate
auto lookup_result = bloom_filter.contains_bulk(no_items);
double fpr = (double) std::count(lookup_result.begin(), lookup_result.end(), true) / no_items
    ↪ .size();
```

We show the benchmark code we use in Listing 6. We measure the elapsed time of creating a new empty filter, inserting many items, querying all the items inserted and computing the weight of the filter. We then also query 100,000 items not inserted to get the false positive rate.

We perform all our executions with 8 hash functions as it is a quite usual value for this parameter. We consider Bloom filter sizes ranging from 2^{30} (125 MiB) to 2^{33} (1 GiB). Regarding the number of items, we run experiments with 10 million and 100 million, which are realistic numbers of solid k-mers we could find in a set of reads. Finally, regarding the PIM implementation, we consider 6 and 8 ranks (respectively 384 and 512 DPUs).

Hardware-wise, we run all the benchmarks on the same server with an Intel® Xeon® Silver 4215 CPU @ 2.5 GHz processor, 251 GB of DDR4 @ 2.4 GHz RAM, and a total of 40 UPMEM ranks (2560 DPUs) available. The server runs on Debian 10 and uses version 2023.1.0 of the UPMEM SDK.

4.3 Results

In this section, we report the results of our benchmarks. For all cases, we see a common trend that the bigger the filter and the more items we insert / query, the better the speedup of the PIM implementation. This is good news since we designed the latter specifically to deal with huge volumes of data. We now comment the results for each function and metric:

Initialization. We see in Table 1 that the PIM implementation performs significantly faster than the CPU one. Although the initialization is typically done only once, such a speedup can still be interesting when using serialization and reloading previously saved data into a new filter.

Weight. In Table 2, we notice a huge speedup for the PIM implementation regarding the weight function. The underlying algorithm, a bit count reduce, uses a very simple and efficient parallelization scheme that fits the PIM architecture particularly well. Computing the weight of a filter has few interest in practice, but these results provide an idea of the high performances gain we can obtain in these types of situations.

Insertions. We notice in Table 3 a reasonable speedup when inserting items. The performances gain is smaller than for previous methods, but still looks interesting, particularly when the size of the filter increases significantly.

Lookups. Table 4 shows a similar trend for querying the presence of items, although the speedup is smaller. In a typical CPU implementation, lookups are often cheaper than insertions. We however notice the opposite here with the PIM implementation. Two aspects can explain this observation:

- To reorder the booleans and output the query results in the right order, we need to remember the index of each item in the original input vector. In a CPU implementation, we can store the item and the index at the same place with a tuple. But in the PIM implementation, we do not need to send the indexes to the DPUs so we store them in a separate structure. This may be more memory-bound.
- Secondly, we have additional memory transfers compared to insertions since we have to get the results back from the DPUs MRAM. We compressed the data to optimize this phase but it still takes a significant time overall.

Looking at the elapsed time metric, we see that the PIM implementation performs systematically faster with 6 ranks rather than with 8 ranks. We believe this phenomena finds its roots in the way we distribute Bloom filters on the DPUs. Because the hash functions are uniform, all the buckets to transfer items fill at the same rate. With more ranks, there are more buckets to fill and it iterates over more items to trigger launches. DPUs thus spend more time idling and waiting for work to do, which gives an overall worse execution speed.

False positive rate. Table 5 reports similar false positive rates for both the PIM and CPU implementations.

4.4 Traces and Discussion

Using the `dpu-profiling` tool provided by UPMEM, we can monitor the calls to the UPMEM SDK and get a timeline of the application execution. We show the traces we obtain for the insertions and the lookups in Figure 5. The timeline splits the calls in 12 different threads, corresponding to the 6 workers (top lines) and the 6 ranks asynchronous supervisors (bottom lines). Calls in the workers lines are usually quite short since they simply schedule asynchronous transfers, launches and callbacks. The

size2	#items	#ranks	PIM (s)	CPU (s)	PIM Speedup
30	10000000	6	0.148	0.323	2.184
30	10000000	8	0.198	0.323	1.629
30	100000000	6	0.145	0.373	2.580
30	100000000	8	0.187	0.373	1.991
31	10000000	6	0.150	0.667	4.447
31	10000000	8	0.188	0.667	3.548
31	100000000	6	0.144	0.754	5.250
31	100000000	8	0.188	0.754	4.014
32	10000000	6	0.150	1.387	9.234
32	10000000	8	0.187	1.387	7.398
32	100000000	6	0.147	1.528	10.413
32	100000000	8	0.189	1.528	8.068
33	10000000	6	0.156	2.992	19.172
33	10000000	8	0.181	2.992	16.497
33	100000000	6	0.145	3.041	21.039
33	100000000	8	0.185	3.041	16.419

Table 1: Initialization

size2	#items	#ranks	PIM (s)	CPU (s)	PIM Speedup
30	10000000	6	0.022	3.595	166.430
30	10000000	8	0.012	3.595	308.322
30	100000000	6	0.022	3.713	171.696
30	100000000	8	0.011	3.713	326.371
31	10000000	6	0.043	6.764	158.825
31	10000000	8	0.023	6.764	298.625
31	100000000	6	0.044	6.731	153.327
31	100000000	8	0.023	6.731	294.135
32	10000000	6	0.085	12.759	150.920
32	10000000	8	0.044	12.759	290.824
32	100000000	6	0.085	13.462	159.237
32	100000000	8	0.044	13.462	309.406
33	10000000	6	0.169	27.344	162.277
33	10000000	8	0.086	27.344	317.524
33	100000000	6	0.169	26.757	158.614
33	100000000	8	0.085	26.757	313.476

Table 2: Weight

size2	#items	#ranks	PIM (s)	CPU (s)	PIM Speedup
30	10000000	6	0.122	0.228	1.872
30	10000000	8	0.122	0.228	1.868
30	100000000	6	0.759	1.536	2.024
30	100000000	8	0.713	1.536	2.154
31	10000000	6	0.122	0.271	2.231
31	10000000	8	0.118	0.271	2.296
31	100000000	6	0.767	1.582	2.063
31	100000000	8	0.730	1.582	2.168
32	10000000	6	0.123	0.340	2.767
32	10000000	8	0.121	0.340	2.815
32	100000000	6	0.769	1.791	2.328
32	100000000	8	0.724	1.791	2.474
33	10000000	6	0.123	0.399	3.247
33	10000000	8	0.118	0.399	3.388
33	100000000	6	0.781	2.411	3.088
33	100000000	8	0.724	2.411	3.329

Table 3: Insertions

size2	#items	#ranks	PIM (s)	CPU (s)	PIM Speedup
30	10000000	6	0.196	0.238	1.217
30	10000000	8	0.185	0.238	1.291
30	100000000	6	1.483	2.032	1.370
30	100000000	8	1.385	2.032	1.467
31	10000000	6	0.193	0.274	1.420
31	10000000	8	0.172	0.274	1.588
31	100000000	6	1.470	2.265	1.541
31	100000000	8	1.411	2.265	1.606
32	10000000	6	0.195	0.324	1.660
32	10000000	8	0.197	0.324	1.642
32	100000000	6	1.408	2.428	1.724
32	100000000	8	1.392	2.428	1.744
33	10000000	6	0.193	0.515	2.673
33	10000000	8	0.187	0.515	2.755
33	100000000	6	1.421	3.060	2.154
33	100000000	8	1.320	3.060	2.317

Table 4: Lookups

size2	#items	#ranks	PIM	CPU
30	10000000	6	0.0000	0.0000
30	10000000	8	0.0000	0.0000
30	100000000	6	0.0026	0.0134
30	100000000	8	0.0145	0.0134
31	10000000	6	0.0000	0.0000
31	10000000	8	0.0000	0.0000
31	100000000	6	0.0001	0.0008
31	100000000	8	0.0006	0.0008
32	10000000	6	0.0000	0.0000
32	10000000	8	0.0000	0.0000
32	100000000	6	0.0000	0.0001
32	100000000	8	0.0000	0.0001
33	10000000	6	0.0000	0.0000
33	10000000	8	0.0000	0.0000
33	100000000	6	0.0000	0.0000
33	100000000	8	0.0000	0.0000

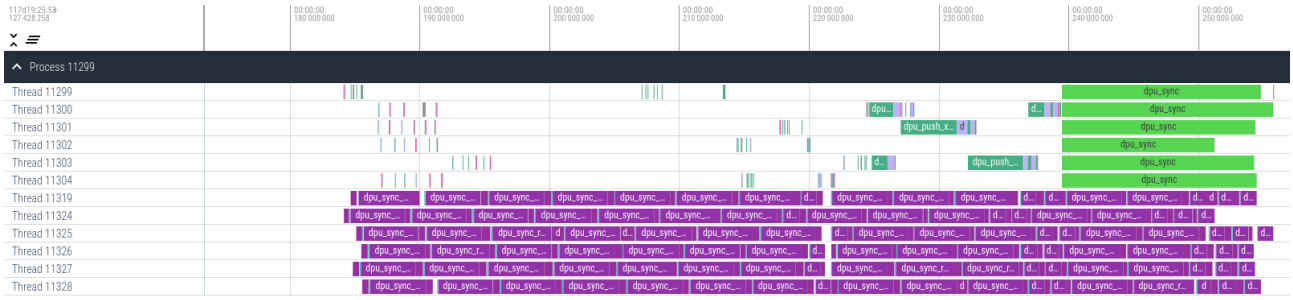
Table 5: False positive rate

actual operations occur in the bottom lines and we can see `dpu_sync` calls performed under the hood that indicate when ranks are executing.

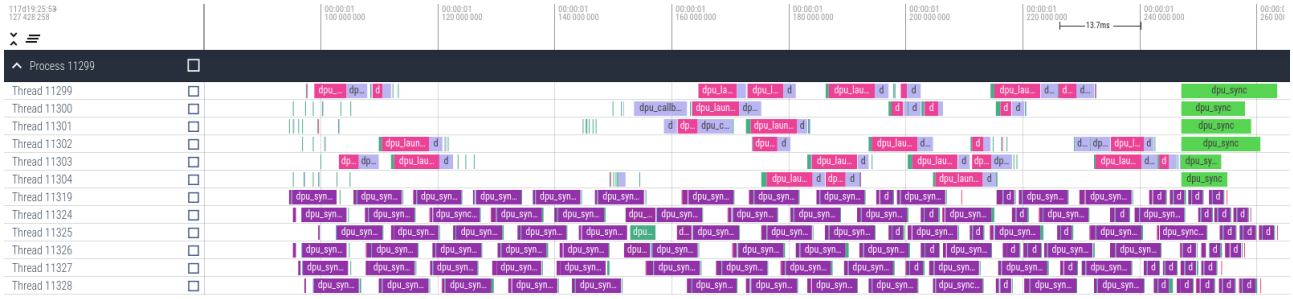
During the insertions, we see that the DPUs are almost always active once the first launch have been triggered. The workers stack enough asynchronous calls to keep them busy and not waste time. This timeline shows that our numerous refinements to the code paid off and that our implementation is quite optimized. We believe accelerating the insertions further could only happen through one the following means:

- Optimize further the DPU program, for instance by tweaking the assembly code. If we increase the idle time of DPUs, we can then send more items in each bucket to fill the idling space and hopefully reduce the total number of rounds.
- Find a way to trigger the first round sooner. But this may only delay the idling of DPUs to later in the timeline and not impact the overall elapsed time.
- Experiment with significantly different ways of distributing the data and items to the DPUs.

When querying the filter, we have more idle time for the DPUs because of the additional callback. Once the lookup results have been transferred from the DPUs to the host, the callback still has to perform the reordering of elements in the result vector before returning. This delays the next asynchronous operations and explains why we have some blank spaces in the trace. To optimize the lookups further, we could end the callback right after the transfer and execute the reordering in another thread or some available worker. This is however more complex to implement and would require special care to make sure the data stays alive in the necessary scopes and do not lead to expensive copies. It is unclear whether this would actually yield a significant improvement. In any cases, lookups cannot be faster than insertions, which indicates a lower bound. We can only hope to accelerate the lookups further by at most 35-45% given where the insertions currently stand. But it could still be worth to investigate this path since querying a filter is probably the most used operation in practice.



(a) Insertions



(b) Lookups

Figure 5: Traces of the PIM Bloom filter benchmark

5 Conclusion

In this work, we designed a Bloom filter implementation on the UPMEM PIM architecture. Our evaluation shows it provides an interesting speedup compared to an efficient Bloom filter implementation running on CPU. For a filter of size 2^{33}, our implementation turns out to be around 20, 3, 2, and 300 times faster for respectively an initialization, a bulk of insertions, a bulk of queries, and a weight computation.

However, we obtain the larger performances gain for the less relevant functions of a Bloom filter. In practice, the most used operations are insertions and lookups. This is where we had to proceed with a lot of iterating and optimizing to get decent performances. In the end, these are only a few times faster than our CPU baseline, which is already an interesting improvement although we were hoping for more.

Because of how they work, Bloom filters are inherently very memory-bound. The use of uniform hash functions indeed induce random memory accesses and a lot of cache misses, which harms the performances on CPU. Another goal of this work was to investigate if this property would make them a good fit or not for PIM architectures. Thanks to all the challenges we faced and tackled in this work, we can now provide insights on this topic.

On the UPMEM PIM architecture, we believe the memory-bound property of Bloom filters remains problematic for a few reasons. First, we cannot use an efficient cache in WRAM for the filters stored in MRAM. With the cache filter approach, we can avoid direct MRAM accesses and retrieve a block in a cache. This permits to efficiently set all the bits for the different hash indexes. However, we cannot reuse this cache for the following items as the block will probably be completely elsewhere. This induces a lot of transfers between the MRAM and the WRAM (at least one read transfer for each item handled), just like it would cause a lot of cache misses on CPU. In the event that an efficient sorting algorithm gets designed for the UPMEM PIM architecture in the future, we may be able to

reorder the items and minimize these transfers to accelerate the DPUs part further.

Another bottleneck lies in the distributed nature of our approach. Items are inserted into one of many smaller filter. In order to ensure correctness and forbid false negative, we need to query a specific fragment, i.e. the filter where the item would have been inserted. That is why we use a deterministic mapping to dispatch each item to a specific rank and DPU within this rank. We however believe this data dispatching scheme is not the best fit for the UPMEM PIM architecture. Because data needs to be sent to a specific DPU, this leads to a lot a data copies and buffer preparation which is very memory-bound. It would be much more efficient to send the next raw chunk of data to the first rank available, but we cannot use this approach in the case of Bloom filters.

Finally, let us recall we focused in this work on implementing a rather generic Bloom filter library and used an approach inspired by our experience with the GATB library. Different ways of distributing the data on DPUs could yield better performances gains and would be interesting avenues of research. Furthermore, better speedups could also be obtained by taking a step back and considering a whole application from raw data to some computational result. Executing more processing on the DPUs could make the dispatching and data transfers between the host and the DPUs more cost-efficient. We believe investigating different scenarios that go beyond the sole use of the filters elementary operations could be a very interesting area to explore in future work.

REFERENCES

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[2] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):22, September 2013.

[3] Erwan Drezen, Guillaume Rizk, Rayan Chikhi, Charles Deltel, Claire Lemaitre, Pierre Peterlongo, and Dominique Lavenier. GATB: Genome Assembly & Analysis Tool Box. *Bioinformatics*, 30(20):2959–2961, October 2014.

[4] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652, January 2013.

[5] Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. pages 35–44, March 2014.