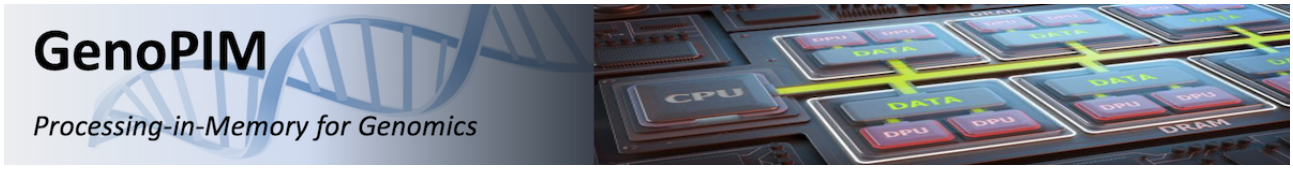

Genome Matching with UPMEM Data Processing Units

Abstract

Motivation. As bacterial databases are growing exponentially, finding genes or mutations require efficient and fast algorithms to perform approximate pattern matching of a query against a set of genomes. It is thus essential to assess the potential of modern processors and hardware accelerators for such task. We focus in particular on the Processing-in-Memory accelerator from the UPMEM company.

Results. We devise a proof-of-concept implementation on the UPMEM PiM architecture and compare it to a CPU-only equivalent that uses SIMD instructions. Our evaluation shows that the PiM program scales better as the database size increases, but suffers from the lack of vectorized instructions and fails to achieve significant speed-ups. Despite promising characteristics, this workload does not appear as a good fit for this architecture.

Table of Contents

1 Introduction ***2***

2 Implementation Details ***3***

2.1 CPU-only Implementation 3

2.2 UPMEM PiM Implementation 4

3 Evaluation ***5***

3.1 System 5

3.2 Datasets 5

3.3 Results 5

4 Discussion and Conclusion ***6***

References ***7***

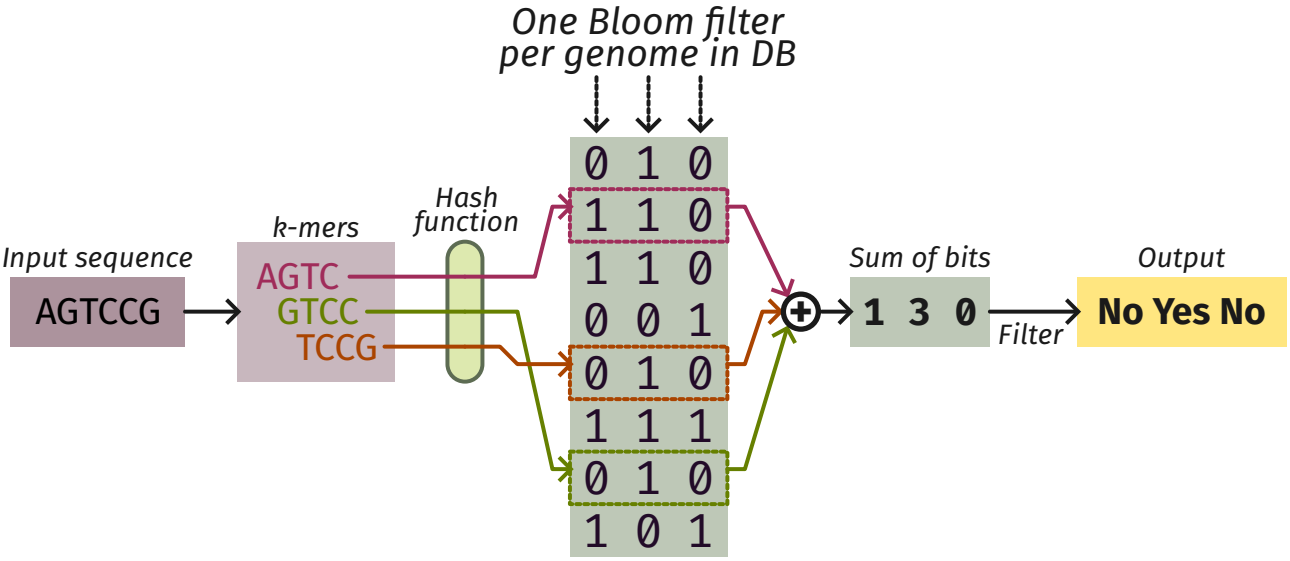

Figure 1: Querying an input sequence against a database of genomes represented as Bloom filters

1 Introduction

As the volume of sequencing data grows exponentially, it is paramount to design fast and efficient algorithms that fully leverage the capabilities of modern processors and hardware accelerators. In this work, we consider the objective of performing approximate pattern matching of a query against a set of genomes. This typically serves the purpose of finding genes or mutations within a database.

We consider the software COBS [1] that uses Bloom filters [3] to create a compact inverted index. As illustrated in Figure 1, each genome in the database is represented as a bit set and constitutes a column of a matrix. To query an input sequence, COBS performs lookups in the matrix to count how many k-mers from the input appear in each genome of the reference. The implementation of this computational step benefits from SIMD instructions to unpack the bits of each line and sum counters. Finally, COBS filters the results and outputs the list of genomes from the database that obtain a score above a user-defined threshold.

In its classic mode, COBS uses the same Bloom filter size for every genome in the database. The software also proposes a second mode, called compact, where genomes can be represented as Bloom filters of different sizes, which then requires to reorder the set of columns to organize the database matrix in blocks. This alternative mode leads to reduced memory footprint and faster querying times.

In this work, we consider the classic mode of COBS and investigate a modified querying procedure where we also represent the input sequence as a Bloom filter. This facilitates the computation since the result is equivalent to the multiplication of the input bit set by the database matrix. Our motivation lies in the fact that this makes the memory access patterns more linear and thus reduces the amount of cache misses. We perform two implementations: the first on CPU only, and the second using the Processing-in-Memory (PiM) accelerator of the UPMEM company [4]. Our end goal consists of gaining insights whether this application workload is fitted for the UPMEM PiM accelerator or not.

We start by sharing some implementation details in Section 2. Then, we present our experimental setup and show our results in Section 3. Finally, we provide some concluding remarks in Section 4.

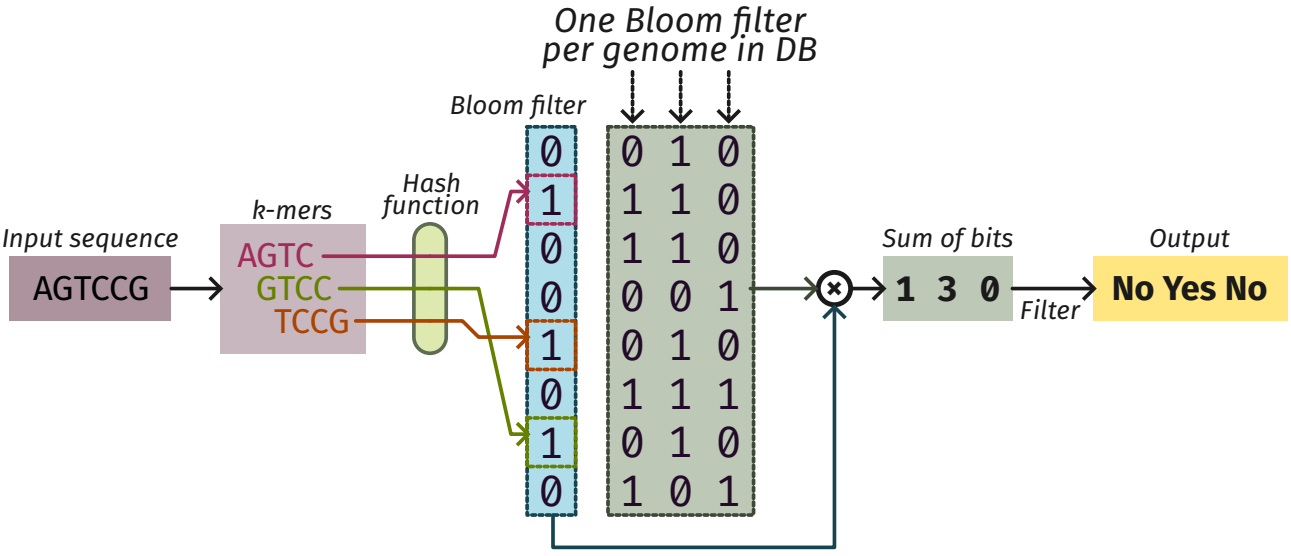

Figure 2: Representing the input sequence as a Bloom filter with k-mers as elements

2 Implementation Details

We make the implementation in C++17. To build Bloom filters, we use only one hash function and rely on the xxHash library[1]. We use the same size of 1 MB for all Bloom filters.

As showed in Figure 2, we first convert the input sequence into a Bloom filter. The computation then becomes a matrix multiplication instead of a series of lookups. The final filtering step remains the same, although we need to use a different threshold. In the original procedure, it consists of keeping genomes that match for instance 80% of the input k-mers. With the new input representation, because of collisions, the number of k-mers does not equal the number of ones in the Bloom filter. So we use the latter instead, i.e. the weight of the input Bloom filter, to compute the thresholds for filtering results.

The only parts that differ between the CPU and PiM implementations is how the matrix multiplication is computed and the parallel structure adopted. We now give more details for both implementations.

2.1 CPU-only Implementation

In the CPU version, we use OpenMP to handle input queries in parallel.

To compute the results for one query, we rely on the Intel® Intrinsics. First, we initialize as many counters as genomes in the database, represented by 32-bits integers and packed into vectors of 256 bits. Then, we iterate all the bits of the input Bloom filter and ignore all the zeros. If we find a one, we retrieve the corresponding line in the database and process it as follows for each byte:

1. We use a parallel bit deposit instruction to convert the byte into 8 bytes that end up being either zeros or one.
2. We execute an AVX2 instruction to extend these 8 bytes into 8 32-bits integers, i.e. a vector of 256 bits.
3. Finally, we sum this vector and the corresponding counters with another AVX2 instruction.

[1] https://github.com/Cyan4973/xxHash

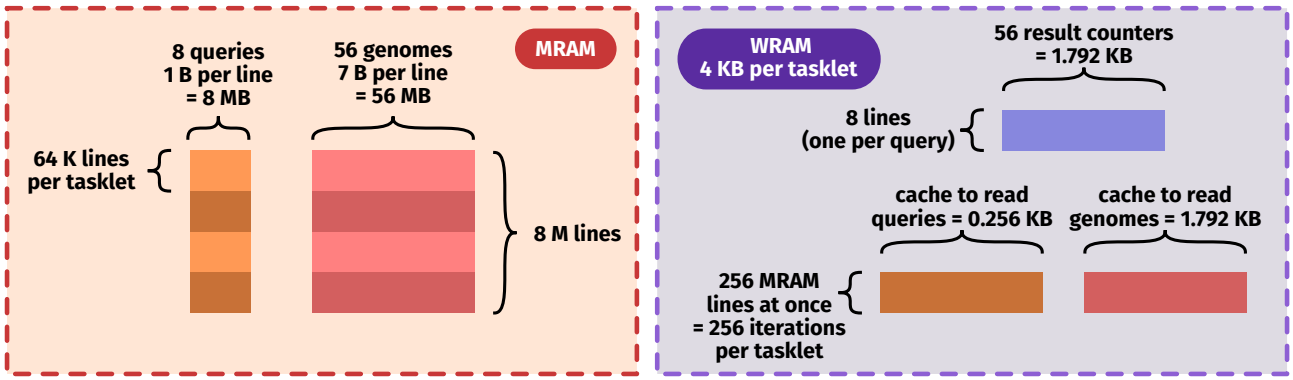

Figure 3: Organizing data in the MRAM and WRAM of DPUs

This process exhibits several characteristics that make it very efficient:

- One byte from the database contains information for 8 genomes, but these are all processed at once thanks to vectorized instructions.
- Memory access patterns are linear and predictable.
- Branches are used only for reading the query, but not for the database, which limits the amount of bad speculation in the micro-instructions pipeline.

2.2 UPMEM PiM Implementation

In the PiM implementation, we adopt the memory organization illustrated in Figure 3. In particular, we split the database and dispatch the data once at the initialization to store 56 genomes per DPU. One rank can thus contain up to 3584 genomes. During the computation, we broadcast the queries to all DPUs, and handle 8 at once. Since each Bloom filter has a size of 1 MB, this structure fully uses the 64 MB memory space of the MRAM. On each DPU, we partition the data and affect different ranges of lines to each of the 16 tasklets running in parallel. The process executes as follows:

1. Each tasklet own result counters in their own WRAM stack and zero-initializes them.
2. Tasklets read their share of the data and perform the corresponding computations to write results into their local counters.
3. Once all the tasklets are done with the computations, results are aggregated in parallel in the result counters of the first tasklet.
4. Finally, the first tasklet write the end results in the MRAM for the host to retrieve them.

When the host has grouped 8 queries into a batch of 8 Bloom filters, it broadcasts the data to all DPUs and launches the execution. While the DPUs are running, the host computes the weights of the 8 Bloom filters to prepare for the filtering step later on. Once all ranks have finished, the host retrieves the results and proceed to the final filtering to get the end results.

To perform the computation on the DPUs, we leverage the sequential nature of memory access patterns to use intermediate caches in WRAM as illustrated in Figure 3. Since we do not have vectorized instructions on the DPUs, we perform the computation differently than the CPU-only implementation. We rely on a bit scan reverse on 32 bits with the `clz` instruction, and then reset the most significant bit set with a mask and a XOR operation. We use this procedure in two nested loops: first for the queries and second for the genomes. Given a query, if `clz` finds a bit set in a database byte, it takes in total 7 instructions to increment the corresponding result counter. We resort to inline assembly to

use the `clz` instruction as efficiently as possible and directly jump to the next iteration if there are no bits set.

3 Evaluation

3.1 System

We execute our experiments on a server with an Intel® Xeon® Silver 4215 CPU @ 2.5 GHz processor (Skylake architecture), 256 GB of DDR4 @ 2.4 GHz RAM, and 20 UPMEM DIMMs @ 350 MHz (which corresponds to 40 ranks, i.e. 2560 DPUs, and a total of 160 GB of MRAM memory). The server operates on Debian 10 and uses version 2023.2.0 of the UPMEM's SDK.

3.2 Datasets

We consider a subset of the 661k collection from [2] that contains bacteria genomes obtained by assembling sequencing data from the European Nucleotide Archive.

In particular, we use 4000 genomes from the *Acinetobacter baumannii* bacteria. We select 3584 genomes to compose the reference database, and we pick a contig from each of the remaining assemblies to compose a dataset of 416 queries. The reads in the latter have a size between 9k and 1.4M nucleotides, with an average of 329k.

Given the way we implemented the matching, the database of 3584 genomes fits perfectly in one rank of DPUs. To simulate a bigger database, we duplicate the same 3584 genomes to fill more ranks, and we measure the database size in terms of number of ranks.

3.3 Results

We execute the matching of the 416 queries with both implementations and report in Figure 4 the elapsed time for a varying number of ranks. Additionally, we plot a linear interpolation to estimate the trend for up to 40 ranks.

We see that the CPU-only implementation, which runs with 32 threads, is very fast but gets significantly slower as the database size increases. This is consistent with each thread getting more computations to execute.

On the other hand, the PiM implementation gets only slightly slower with more ranks. Using more ranks means running more processing units in parallel, so the computational cost remain the same no matter the size of the database. The slight slowdown is thus due to the data transfers between the host and the DPUs that get more numerous, in particular the broadcasting of the queries. This implementation is however a lot slower than the CPU-only version at first because it does not benefit from vectorized instructions. The bit scan procedure we use on DPUs cannot compete with AVX2 instructions.

Besides, with the help of the performance library from the UPMEM's SDK, we measure that the DPUs pipeline is full at 99.2%. There is thus very little stalling on memory accesses, and the execution appears rather compute-bound.

In the end, the interpolation shows the PiM implementation would become faster than its counterpart once the database exceeds a size of 23 ranks. Even with a database of 40 ranks, the maximum that

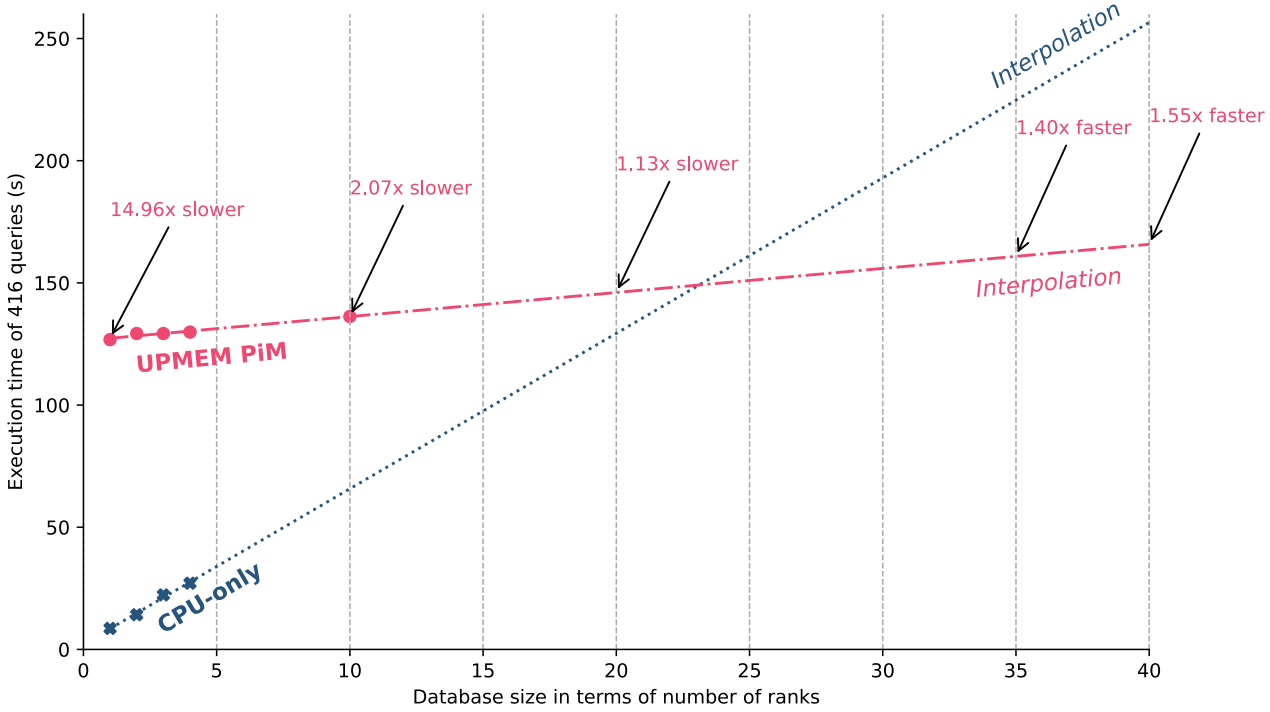

Figure 4: Performances of the CPU-only and UPMEM PiM implementations

can fit on the 20 UPMEM DIMMs of our server, the PiM implementation would only be 1.55 times faster.

4 Discussion and Conclusion

In this report, we considered the application of performing approximate pattern matching of a query against a set of genomes. In particular, we focused on the case of the COBS software [1]. We explored an alternate procedure by compacting the input sequence into as a Bloom filter, with the end goal of investigating whether this approach could benefit from the UPMEM PiM technology. This workload exhibits indeed several characteristics that make it a promising candidate for this architecture:

- The computation can be easily fragmented into small independent tasks.
- Memory access patterns are linear, which means we can easily leverage the WRAM to cache data from the MRAM.
- Queries are broadcasted to all ranks, so the orchestration is simple and lead to balanced work across all processing units.

We performed a proof-of-concept implementation on the PiM architecture, and compared it to a CPU-only version that uses SIMD instructions. Results show the PiM implementation scales better with the database size. Allocating more ranks to increase the available memory space indeed shows little impact on the execution time. The computation duration remains the same, and the slight slowdown is due to the additional data transfers between the host and the DPUs. However, the base computational time is much slower than the CPU-only implementation. Through interpolation, we estimated that the

PiM implementation would be faster only when the database size exceeds 23 ranks, with a potential speed-up of 1.55 times at 40 ranks.

These results indicate this workload may not be the best fit for the UPMEM PiM architecture as it seems unlikely to obtain significant speed-ups. The main reason lies in the fact that the DPUs do not have vectorized instructions to unpack bits and sum several values at once. Instead, we have to rely on a bit scan approach that is less efficient. Overall, the workload seem more compute-bound than memory-bound, so it may be a better fit for other hardware accelerators such as GPUs or FPGAs.

In case the UPMEM PiM architecture gets new instructions that help to accelerate the computation, we may revise our conclusions. We indeed believe that the ability to scale with little overhead is promising, and that the potential of the PiM approach for energy savings need to be investigated further.

REFERENCES

[1] Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. COBS: a Compact Bit-Sliced Signature Index, July 2019. arXiv:1905.09624 [cs].

[2] Grace A. Blackwell, Martin Hunt, Kerri M. Malone, Leandro Lima, Gal Horesh, Blaise T. F. Alako, Nicholas R. Thomson, and Zamin Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*, 19(11):e3001421, November 2021. Publisher: Public Library of Science.

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

[4] Fabrice Devaux. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, August 2019. ISSN: 2573-2048.