MiMyCS: A Processing-in-Memory Read Mapper for Compressing Next-Gen Sequencing Datasets

Florestan de Moor, Meven Mognol, Charles Deltel, Erwan Drezen, Julien Legriel, Dominique Lavenier

To cite this version:

Florestan de Moor, Meven Mognol, Charles Deltel, Erwan Drezen, Julien Legriel, et al.. MiMyCS: A Processing-in-Memory Read Mapper for Compressing Next-Gen Sequencing Datasets. BIBM 2024 - IEEE International Conference on Bioinformatics and Biomedicine, Dec 2024, Lisbonne, Portugal. pp.7176. hal-04821180

HAL Id: hal-04821180
https://inria.hal.science/hal-04821180v1

Submitted on 5 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

MiMyCS: A Processing-in-Memory Read Mapper for Compressing Next-Gen Sequencing Datasets

Florestan De Moor
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
florestan.de-moor@irisa.fr

Meven Mognol
UPMEM
Grenoble, France
mmognol@upmem.com

Charles Deltel
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
charles.deltel@inria.fr

Erwan Drezen
Institut Pasteur
Paris, France
erwan.drezen@pasteur.fr

Julien Legriel
UPMEM
Grenoble, France
jlegriel@upmem.com

Dominique Lavenier
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
dominique.lavenier@irisa.fr

***Abstract*—As Next-Gen sequencing (NGS) technologies keep improving their accuracy and get largely deployed in human health care infrastructures, it is critical to design efficient reference-based compressors that fully leverage the capabilities of modern processors and hardware accelerators.**

This work proposes MiMyCS: a C++ software to achieve Mapping in Memory for Compressing Short reads. It performs lossless reference-based compression of NGS datasets such as Illumina reads. To this end, MiMyCS computes a non-exhaustive mapping against a reference genome and accelerates this step with the Processing-in-Memory architecture developed by the UPMEM company. Such architecture extends the computational power of a machine by adding dual in-line memory modules on which each memory bank has its own processing unit that runs up to 16 threads. This creates a massively parallel environment, well-fitted to alleviate memory bottlenecks.

To reduce the overall amount of sequence comparisons and accelerate further the process, MiMyCS also incorporates a Bloom filters-based dispatcher that predicts against which genome parts reads are most likely to be mapped.

We show with real whole human sequencing datasets that MiMyCS is able to achieve a speed-up between 1.2x and 2.7x compared to Genozip, the current leading state-of-the-art compressor, while maintaining a comparable compression ratio and lowering the overall energy consumption. The code of MiMyCS is available at https://gitlab.inria.fr/pim/org.pim.srm.

***Index Terms*—Read mapping, Reference-based compression, Processing-in-Memory, Parallelism, Next-Gen sequencing, Bloom filter**

I. Introduction

Next-Gen sequencing (NGS) is getting increasingly used to offer better health care to the general population, for instance for diagnosing major diseases more rapidly, detecting vulnerable patients, or providing better tailored cures. Medical labs and hospitals are therefore incorporating more human sequencing in their daily operations and facing huge and growing volumes of data. This motivates the search of efficient compression algorithms to reduce the storage footprint of short reads datasets, and help health care systems store these critical data at low cost. Nowadays, data are often compressed with the general library gzip. This has the major advantage of being universally available on most recent systems. Being a general-purpose library, it however fails at leveraging the specificity of sequencing information. Specialized tools such as Genozip [1] or Illumina ORA [2] take advantage of the nature of the data and obtain compression ratios up to 5 times better.

Over the years, NGS has improved a lot, not only in terms of financial cost, but also in terms of sequencing accuracy. With the NovaSeq 6000 technology, Illumina claims that the vast majority of bases have a probability above 99.9% to be correct. Therefore, we can hope to align most reads against a relevant genome reference with a low number of mismatches. The latter are indeed more likely to correspond to individual DNA variants than to sequencing errors. This motivates resorting to reference -based compression for improved ratios.

Processing-in-Memory (PiM) architectures offer a highly-parallelized programming environment where computational units are set up close to the memory. In this work, we consider the production system developed by the UPMEM company [3] and investigate how we can leverage this architecture for performing reference-based compression. We present MiMyCS, a tool to achieve Mapping in Memory for Compressing Short reads. It performs lossless reference-based compression of NGS datasets such as Illumina reads. To this end, MiMyCS computes a non-exhaustive mapping against a reference genome and executes this step with the help of UPMEM DIMMs. Namely, it splits the reference sequence across all processing units, and dispatches reads to attempt to find good mapping positions with up to a few mismatches. To reduce the overall amount of sequence comparisons, MiMyCS incorporates a Bloom filters-based dispatcher that predicts against which genome parts reads are most likely to be mapped. This dispatcher is a core-aspect of our software and contributes significantly to obtaining a fast mapping procedure. We show with real datasets and a PiM server with 4 DIMMs that MiMyCS obtains a speed-up between 1.2x and 2.7x

This paper is funded by the European Union's Horizon Europe Programme for research and innovation under grant 101047160 (BioPIM project), and by the French ANR program ANR-21-CE46-0012 GenoPIM.

TABLE I: Reference-based compressors surveyed

Name	Ref	Year	Source code	Multithreaded
LMSRGC	[4]	2023	*Available*	No
RBFQC	[5]	2023	*Available*	No
FQCSpark	[6]	2022	Not available	*Yes*
LMM	[7]	2021	Not available	*Yes*
Genozip	[1]	2021	*Available*	*Yes*
SCCGC	[8]	2019	*Available*	No
UdeACP	[9]	2019	Not available	*Yes*
ParRefCom	[10]	2019	*Available*	*Yes*
LW-FQZip2	[11]	2017	*Available*	*Yes*
HiRGC	[12]	2017	Not available	No
G-FQZip	[13]	2017	Not available	*Yes*
NRGC	[14]	2016	Dead link	No
LW-FQZip	[15]	2015	*Available*	No
HUGO	[16]	2014	*Available*	*Yes*

compared to Genozip [1], the current leading state-of-the-art compressor, while maintaining a comparable compression ratio. We also see that MiMyCS alleviates the CPU usage and reduces the overall energy consumption.

The rest of this paper is organized as follows. First, we present the state-of-the-art of reference-based compression in section II. We explain the architecture and the programming environment of the UPMEM PiM system in section III. We then share the mapping method based on Bloom filters in section IV. Our evaluation setup is specified in section V, while our experimental results are showed in section VI. Finally, we provide some concluding remarks in section VII.

II. Related Work

Reference-based compression has received a lot of attention over the years, as showed in TABLE I, with a major focus on improving the compression ratio rather than compression time. Considerations about the latter become however more present in recent publications that strive to fully leverage the capabilities of modern multithreaded processors and other accelerators. LMSRGC [4] uses GPU acceleration and multiple CPU threads to build the data structures necessary to align reads on the reference, while HiRGC [12] performs GPU-based arithmetic coding. FQCSpark [6] relies on the big data framework Spark to distribute the compression over several worker nodes. LW-FQZip2 [11] spawns multiple threads to both perform the mapping and compression of batches of reads. UdeACP [9] parallelizes the seeding and encoding stages of its compression scheme with multiple threads.

Genozip [1] splits the dataset into blocks and processes them in parallel, with a few extra threads to handle I/O operations. This is a universal compressor that can handle various formats such as FASTQ, FASTA, BAM, SAM or VCF. It performs lossless compression, but also offers lossy options to improve further the compression ratio. It can operate as a reference-free tool, similarly to gzip or its parallel implementation pigz, but can also use a reference to obtain better results. Furthermore, it contains special optimizations for the case of paired-end datasets. To our knowledge, Genozip is currently the most advanced and versatile tool available, with active support and a growing adoption among the genomics community.

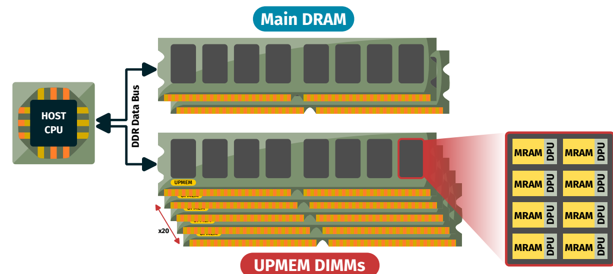

Fig. 1: Server with UPMEM PiM DIMMs

III. UPMEM PiM System

In the von Neumann architecture (i.e. in most modern computers), the central processing unit (CPU) and the memory are two separate components. This design choice becomes a source of bottlenecks for memory-intensive applications. While moving big volumes of data between the memory and the CPU, stalling may appear in the instruction pipeline for various reasons: high cache miss rates, insufficient bandwidth or high latency.

Among manufacturers designing PiM architectures for addressing these issues, the UPMEM company has developed dual in-line memory modules (DIMM) that embed processing units next to the memory [3]. Each unit, called a Data Processing Unit (DPU), can access its own memory bank through a Direct Memory Access engine. This bank contains a 64 MB RAM (called MRAM) which can transfer data to a 64 kB scratchpad (called WRAM). Each DPU constitutes itself a multithreaded environment with 16 threads (called tasklets) executing in parallel.

One UPMEM DIMM is composed of two ranks of 64 DPUs, that operate at a frequency between 350 and 450 MHz, for a total of 8 GB of MRAM memory. These DIMMs can be integrated into an x86 server, next to standard DIMM modules, and therefore extend the machine computational power with additional highly parallelized computing units. Synchronization and coordination of the modules are ensured by the host CPU, since communication between DPUs is not possible. The host can transfer data from the main RAM to the PiM memory banks though a Double Data Rate (DDR) bus, load a compiled program, trigger its execution, wait for its termination and retrieve potential results.

Overall, a server enriched with UPMEM DIMMs (as illustrated in Fig. 1) is a massively parallel environment, with three levels of parallelism: ranks, DPUs, tasklets. Porting an application to such an architecture requires to rewrite the processing into two programs: one for the host and one for the DPUs.

The host program is responsible for allocating the PiM ranks and orchestrating their execution thanks to the UPMEM's SDK. The latter is available for C, C++, Java and Python. The DPUs program is written in C and is built with a dedicated LLVM-based compiler provided by UPMEM. A few debugging tools and libraries are also available, for instance

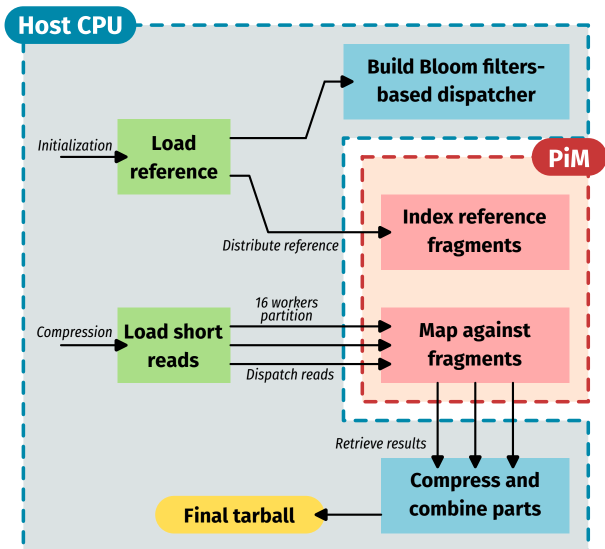

Fig. 2: General overview of MiMyCS: (1) a reference index and a read dispatcher are built, (2) a dataset is compressed with the results of mapping the reads against the reference.

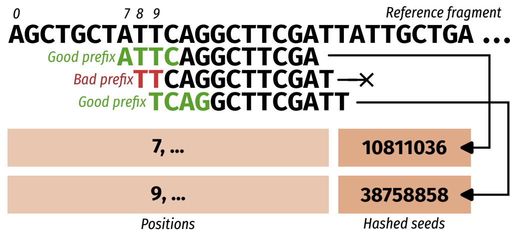

Fig. 3: One DPU indexes its reference fragment by iterating 13-mers (called seeds) and inserting their positions in a table. Some low-interest seeds are filtered out.

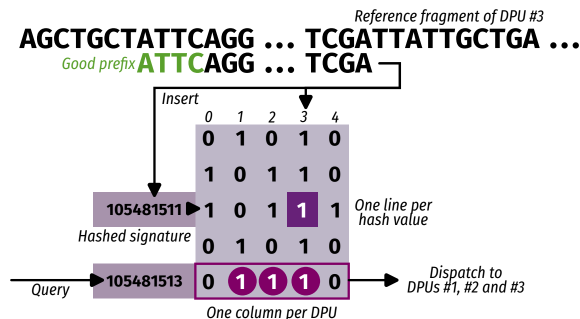

Fig. 4: The dispatcher is a set of Bloom filters, i.e. a matrix of bits with one DPU per column. Querying a signature (i.e. a line) returns the set of DPUs that most likely contain this signature in their reference fragment (false positive are possible).

to deal with tasklets synchronizations, execute direct MRAM accesses, or integrate performance counters.

IV. Compression Method

MiMyCS performs reference-based compression, i.e. relies on mapping reads against a reference genome. For each read, instead of storing the whole DNA sequence in the compressed file, we only need to remember the mapping position and potential mismatches. Given the reference, this is indeed enough information to recreate the original read losslessly. This allows to obtain better compression ratios as we store significantly less information. In order to compete with general-purpose compressors in terms of time performance, the main challenge then lies in designing a fast mapping procedure.

As illustrated in Fig. 2, we distinguish two stages in the overall process of MiMyCS: (1) initialize a few data structures, (2) attempt to map all reads of a dataset, and then output compressed reads in a file by leveraging the results found. We accelerate the mapping with the PiM ranks, but execute the actual compression on the host CPU only.

The main design paradigm of this work is to rely on several heuristics to obtain a non-exhaustive mapping. Our algorithm does not aim to find the best mapping for each read, but rather to get a good-enough alignment as quickly as possible. In particular, we do not attempt to map reads against every part of the reference genome, but rather against specific fragments. We distribute the reference sequence across all the PiM processing units and send each read to a few well-chosen units instead of broadcasting to all. Our dispatching mechanism is based on Bloom filters and constitutes a core contribution of MiMyCS to improving mapping performance.

We now detail more specifically several aspects of the method.

A. Indexing the Reference Genome

Reads can align against the reference sequence either in the direct or reverse complement direction. Instead of spawning two queries from one read, we build the reference sequence by concatenating the input reference genome and a reverse complement copy of it.

We distribute the resulting sequence over the set of PiM processing units. Therefore, each DPU has a fraction of the genome in its MRAM. The fragments overlap slightly to ensure to not miss alignments on the edges.

We then trigger the execution of the PiM ranks, and all genome fragments get indexed in parallel by the DPU they belong to. Each unit iterates the 13-mers of its fragment, which we call seeds, and hashes them to record their sequence position into a table stored in MRAM (Fig. 3). We filter out some low-interest seeds by looking at repetitions in the four initial bases.

Since the programming environment of DPUs has no efficient mechanism for dynamic memory allocation yet, we implement the index with several static arrays to mimic a hash table. Because of this and the limited size of 64 MB of the MRAM, we are unable to consider seeds longer than 13 bases.

B. Designing a Bloom Filters-based Read Dispatcher

The reference genome and index are both distributed over the PiM processing units. Therefore, a naive approach consists

of sending every read to every DPU to attempt to find a mapping against each part of the reference. This however leads to a lot of irrelevant computations. We thus introduce the core heuristic of MiMyCS: a Bloom filters-based read dispatcher. Given a read, this structure is responsible for determining a set of reference genome fragments where a mapping is most likely to appear.

We base our dispatcher on Bloom filters [17], a well-known probabilistic and compact set data structure that supports two operations: (1) inserting an element and (2) querying the presence of an element, both with a constant time complexity. Implementation-wise, a filter is a vector of bits with all cells initialized to 0. It uses a uniform hash function to hash any element into a position in the vector and point to a specific bit . The insertion operation sets the bit to 1, while the lookup returns a positive if the bit equals 1 and a negative otherwise. The probabilistic nature of Bloom filters lies in the fact that a query answer should be interpreted as *No* or *Maybe Yes*. Hashing collisions can indeed lead to believe some elements were inserted in the filter when they were not.

We design our dispatcher as a set of Bloom filters, one per DPU. This can be represented as a matrix of bits, as illustrated in Fig. 4, with hash values for lines and DPUs for columns. To initialize the structure, we iterate the S-mers of each reference fragment, which we call signatures, and insert the hashed signatures into the appropriate filter. Namely, all the signatures of the i-th reference fragment (i.e. the part sent to DPU number i) end up into the i-th column of the matrix. We set empirically $S = 70$ nucleotides, as it sends reads to very few DPUs while maintaining good mapping results.

Once the dispatcher is initialized, we can query a signature (i.e. a line) by performing a lookup operation on each column of the matrix and keep only the positive results. This returns the set of DPUs that contain this signature in their reference fragment, plus a few others as Bloom filters might yield false positives. This querying mechanism is a key component of MiMyCS to steer the mapping attempts towards the most relevant parts of the genome and avoid pointless computations.

In practice, we initialize the dispatcher once and serialize the matrix on disk to quickly reload it for future executions.

C. Mapping Short Reads

1) Host Orchestration: We read the input dataset with 16 workers in parallel, and spawn 2 additional post-process threads. Workers operate independently of one another and maintain their own buffers to communicate with the PiM ranks. We push asynchronous tasks to the PiM ranks in a schedule-and-forget approach.

To handle a read, a worker selects two signatures in the sequence. It queries the dispatcher and determines the DPUs which are returned for both signatures with a set intersection. It inserts the read data in every corresponding communication buffer. If a buffer ends up full after the insertion, the worker triggers the execution of the whole PiM rank the buffer belongs to. This consists of (1) sending the buffers to the DPUs of the rank, (2) starting the execution of the rank, and (3) scheduling a callback to execute once the rank finishes.

The callback function transfers back the mapping results from the DPUs to the host CPU, and pushes a task to the two additional post-process threads. The latter are responsible for reading the DPUs output and committing individual results to the final result structure if they improve the current mapping. For each read, we store the mapping distance, the reference position of the alignment, and the mismatch positions if any.

In case we found no match for a read, we may perform another attempt and query the dispatcher again with different signatures in the sequence. This constitutes a parameter to either prioritize time performance or mapping coverage.

2) Mapping on DPUs: The DPUs receive queries in an array stored in MRAM, and each tasklet handles a slice of them. Given a query, a tasklet selects two seeds in the sequence and retrieves their corresponding positions in the reference index. When the positions of the two seeds are separated by the exact read size, we actually compare the query and the reference sequence. To this end, we perform a bit scan forward / reset procedure written in assembly. Given 32 bits to compare, i.e. 16 base pairs (bp), it returns in one instruction if the mapping is perfect. Otherwise, it takes 7 more instructions per mismatch to complete. In the end, it returns the Hamming distance and the index of all bases giving a mismatch.

The comparison computation directly skips to the next try if it detects too many errors. When it finds a mapping (perfect or with a few errors), it directly skips to the next query without considering other index positions: one result is enough for our compression purposes.

D. Optimizing the Mapping of Paired-end datasets

We introduce an optimization to handle the second part of Illumina paired-end datasets more efficiently. If a read was mapped in the first part, we ignore the dispatcher and directly send the paired read to the DPU containing the reverse complement portion of the aligned reference segment. We indeed make the assumption that it is likely to find a mapping there. This optimization reduces the amount of work the DPUs execute, which accelerates the processing.

E. Compressing the Dataset

To start the actual compression, we read the whole dataset again with the exact same 16-workers split used during the mapping. Each worker considers a batch of reads as one block and encodes the fields in four different streams: names, comments, sequences, and qualities.

1) Names and Comments: We implement a simple approach where we try to split the strings with common separators and check if some fields are numerical. For such fields, we use delta encoding. Otherwise, we append the raw data.

2) DNA Sequences: We use the mapping results previously computed. For mapped reads, we record the reference position, the number of errors, and, if any, the mismatch positions and true bases. If we allow up to 4 mapping errors, this encoding takes at most 9 bytes per read. For unmapped reads, we record

TABLE II: Reference genome and short reads datasets

	Plain file size (GB)	Read count	Read size	Year
Genome GRCh38	3.2	1	3.1 Gbp	2013
Reads H1 (SRR14724532)*				
FASTQ	2x 121.1	356 M	151 bp	2021
FASTA	2x 66.3			
Reads H2 (SRR14724533)†				
FASTQ	2x 122.9	361 M	151 bp	2021
FASTA	2x 67.3			

* https://www.ebi.ac.uk/ena/browser/view/SRR14724532
† https://www.ebi.ac.uk/ena/browser/view/SRR14724533

the full DNA sequence encoded with 2 bits per base, which takes for instance 38 bytes for reads of size 150 bp.

3) Qualities: We use run-length encoding with the modifications described in the Genozip report [1]. We record runs of the dominant quality character separately from the other characters, and we insert a special value to indicate when a character run is not preceded by a dominant character run.

4) Final Tarball: Workers concatenate the four encoded streams and compress blocks with Zstandard into temporary files. Lastly, we regroup the 16 intermediate compressed parts plus a metadata file into a final tarball.

V. Evaluation Setup

We now describe our setup for evaluating the performance of MiMyCS. Hardware-wise, we run all the benchmarks on the same 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). In practice, we run MiMyCS with only 8 ranks. The server operates on Debian 10 and uses version 2023.2.0 of the UPMEM's SDK.

A. Datasets

We consider two real whole genome sequencing datasets of Homo Sapiens obtained with the Illumina NovaSeq 6000 technology: SRR14724532 and SRR14724533. Both are paired-end datasets available for download on the European Nucleotide Archive browser [18]. Since these datasets were produced with recent well-matured next-gen sequencing technology, we expect the number of sequencing errors to be low and to therefore obtain a good mapping coverage against the human genome GRCh38 that we use as reference.

For each dataset, we consider the original FASTQ as well as the equivalent FASTA by stripping the quality information. We refer to the two parts of a paired end dataset with the suffixes /1 and /2. The detailed characteristics of the datasets and reference genome are given in TABLE II.

B. Metrics

1) Compression ratio: We define the compression ratio as the compressed file size divided by the plain file size, expressed as a percentage. Therefore, the smaller, the better.

2) Mapping coverage: The mapping coverage is the percentage of reads for which a mapping was found (either perfect or with a few mismatches). The higher the coverage, the more efficient the compression will be.

3) Execution time and speed: The execution time is the duration reported by the time command to perform a given process (for instance compressing or decompressing a dataset) in seconds. We consider mostly the elapsed time, but also mention the CPU time for further analysis. We also translate execution time into speed by taking into account the plain input dataset size (expressed in MB/s).

4) Energy consumption: To estimate the energy consumption of the host CPU, we rely on the perf command line tool as a wrapper around the Intel® Running Average Power Limit (RAPL) interfaces. These are natively present in Skylake Intel® processors and give measures of the accumulated energy consumption in Joules of the DRAM and Package (PKG) domains. The latter includes the cores, integrated graphics and other uncore components such as the memory controller and last level caches. Benchmarks from [19] show that the RAPL interfaces often offer a good enough approximation of what would be obtained with power meters measures.

Regarding the PiM ranks, we compute an estimation by using the execution time of the different processing steps and the power consumption of the UPMEM DIMMs. We consider a simple approximate model where a DIMM has a power of 23.22 W when actively running and a power of 5 W when idling [20]. The latter is a similar consumption to a standard DIMM being refreshed, while the former is an approximate worst-case estimation, based on the assumptions that (1) the instruction pipeline is full at all time (which is not always the case) and (2) all instructions have the same energy cost (which is false, MRAM reads cost 1000 times more than additions [20]).

C. Baselines

We first compare MiMyCS with a CPU-only implementation (-cpu) to see the benefits of using PiM hardware. This version follows the exact same algorithm, but executes the indexing and mapping immediately in CPU threads instead of dispatching the work asynchronously to DPUs.

Then, we measure how the whole compression pipeline performs against a few state-of-the-art software:

- Genozip [1], in both reference-free and reference-based (-e) configurations. We consider the default mode as well as the fast mode (-fast) that worsens the compression ratio for improved time performance. Finally, we use the default number of threads (1.1 per core, i.e., 35 on our server) or limit the number to 16 (-@16).
- LW-FQZip2 [11], a reference-based software which only handles FASTQ datasets.
- pigz, the multithreaded implementation of the general-purpose gzip library (i.e. a reference-free compressor).

We did not include more tools from TABLE I in our experimental comparison. In many cases, the source code is not publicly available or is lacking instructions to fix compiling

TABLE III: Detailed performance results of MiMyCS for compressing and decompressing short reads datasets

(a) Initialization

Elapsed time (s)	**GRCh38**
Load reference and filters	20
Index reference	21

(b) Processing of short reads datasets

Metrics		**H1/1**		**H2/1**	
		FASTQ	**FASTA**	**FASTQ**	**FASTA**
Compression ratio (%)		5.52	5.04	5.52	5.02
Mapping coverage (%)	Exact	69.58		70.03	
	Up to 4 errors	83.35		83.66	
Elapsed time (s)	Map reads	64	62	73	67
	Compress dataset	68	30	78	34
	Compression total*	188	143	206	154
	Decompression	830	528	849	548
Speed (MB/s)	**Compression**	644	464	597	437
	Decompression	146	126	145	123

* Time for the whole compression pipeline, including initialization and cleaning

issues. For some tools, the use cases are slightly different and prevent a fair comparison of results (e.g., ParRefCom compresses only the DNA sequence and ignores the names and qualities).

VI. Evaluation Results

We now present our experimental results. First, we focus on the performance of our implementation, and then see how it compares with its CPU-only version and the state-of-the-art.

A. *MiMyCS Performance*

We run MiMyCS with 8 PiM ranks, which corresponds to reference sequence fragments of 12.1 Mbp per DPU. We allow up to 4 errors during the mapping stage, and show all results in TABLE III.

For H1/1, each read is sent to 1.96 (resp. 0.46) DPUs on average for the first (resp. second) round, and dataset H2/1 exhibits similar statistics. Compared to an exhaustive broadcast, the Bloom filters dispatcher is thus quite efficient at predicting where an alignment is likely to occur.

To analyze further our results, we took a slice of H1 and computed an alignment with the Burrows-Wheeler Aligner software [21]. It mapped around 90% of reads with up to a few errors. Our mapping coverage remains a few percent below because of the different heuristics we use to accelerate the process. Executing more rounds, as well as decreasing the dispatcher accuracy to obtain more false positives, can improve the coverage and get results closer to the theoretical limit, at the cost of slowing down the application.

The mapping and compression stages of FASTQ datasets take roughly the same time. In the case of several files to compress, it allows setting up a pipeline and compressing a file while mapping another at the same time. This would require reading data from one disk and writing results to another to avoid I/O bandwidth from stalling the whole process.

Next, we evaluate in TABLE IV the performance of our optimization for paired-end datasets. We see that mapping the second part is between twice and three times faster than mapping the first one. This is consistent with the DPUs having less work to execute since many reads are sent to one DPU instead of going through the Bloom filter-based dispatcher. We also report how this optimization affects the mapping coverage, compared to mapping the second part as a single dataset. In both cases, the impact is negligible: a few exact matches are lost, but the overall coverage is slightly better. The optimization thus gives a very interesting speed improvement.

TABLE IV: MiMyCS map paired reads efficiently with no coverage loss

Metrics			**H1**	**H2**
Elapsed time (s)		Map part /1	62	67
		Map part /2	27	29
Mapping coverage of part /2 (%)	**Paired mapping**	Exact	63.19	63.73
		Up to 4 errors	80.55	80.90
	Single mapping	Exact	64.08	64.67
		Up to 4 errors	79.82	80.24

TABLE V: Elapsed time (s) of MiMyCS with PiM DIMMs and an equivalent implementation on CPU-only

Software		**Map H1/1**	
	Index GRCh38	**FASTQ**	**FASTA**
MiMyCS	21	64	62
MiMyCS -cpu	37	134	128

B. *Comparison with a CPU-only Implementation*

We now compare the default implementation of MiMyCS (i.e. relying on UPMEM DIMMs) with a CPU-only equivalent version running on the same server, and report in TABLE V the elapsed time of both the indexing and read mapping steps. We see that using PiM hardware is respectively 1.8 and 2.1 times faster for the indexing and the mapping.

Besides, we measured that the CPU-only implementation takes on average 25.8 ms to map all the reads within a set of rank buffers. On the other hand, a rank of 64 hardware DPUs takes on average 7.6 ms, broken down as follows: 1.3 ms to write data into the MRAMs, 5.3 ms to execute the mapping,

TABLE VI: MiMyCS compresses H1/1 faster than all baselines and obtains a ratio comparable to the fast mode of Genozip

Software	Compression ratio (% of original size)		Compression time (s)	
	FASTQ	FASTA	FASTQ	FASTA
MiMyCS	5.52	🏆 5.04	🏆 188	🏆 143
pigz -6	20.67	27.90	419	328
pigz -9	19.83	26.83	1302	1248
genozip	12.17	18.46	480	676
genozip -e	🏆 4.08	18.46	310	426
genozip -e -fast	5.27	18.49	227	525
genozip -e -@16	4.09	18.56	510	848
genozip -e -@16 -fast	5.32	18.61	362	730
LW-FQZip2	11.15	N/A	18 737	N/A

and 1 ms to transfer back the results to the host CPU. In particular, we note that, data transfers apart, the computational step of mapping reads against a reference fragment is almost five times faster with PiM.

C. Comparison with the State-of-the-art

We compare results of MiMyCS and the state-of-the-art baselines for the dataset H1/1 in TABLE VI. We see that the reference-based mode of Genozip obtains the best compression ratio with the FASTQ file, but fails to deliver similar performances with the FASTA equivalent. It seems indeed to not use the reference available and performs like the reference-free mode. The compression ratio of MiMyCS is 1.5% worse than the default Genozip, but comparable to the fast Genozip and 4 times better than the gzip format.

Regarding execution time, MiMyCS outperforms pigz and LW-FQZip2, but also all Genozip configurations. For the latter, the speed-up ranges from 1.2 to 2.7 times faster for the FASTQ dataset. In TABLE VII, we see that it is also between 2.5 and 4.2 times faster than Genozip in terms of CPU time.

We now compare in TABLE VIII the energy consumption of MiMyCS and several Genozip configurations. We report the RAPL measures of the DRAM and PKG domains. For MiMyCS, we also add an estimation of the consumption of the 8 PiM ranks (i.e. 4 DIMMs). The DPUs are active only during the indexing and mapping steps, and are idling otherwise. While indexing, we consider they are 100% active. During the mapping, we use a tracing tool from UPMEM to monitor the SDK calls and the activity of the DPUs. We measure that they are actively running approximately 40% of the overall time during this step. We thus compute the total durations during which DPUs are active and idle and weight them by the power values to obtain a final energy result of 7.16 kJ. Overall, MiMyCS obtains the lowest energy consumption.

We see in TABLE IX that Genozip decompresses twice faster than MiMyCS. This may be explained by its better compression ratio, leading to less disk reading, but mainly by the way it handles and structures data into blocks, which allows for better scaling with any number of threads. On the other hand, MiMyCS is limited to the same number of workers

TABLE VII: MiMyCS compresses H1/1 (FASTQ) with a lower CPU usage compared to Genozip

Software	Compression time (s)	
	Elapsed	CPU
MiMyCS	🏆 188	🏆 2251
genozip -e	310	9546
genozip -e -fast	227	7000
genozip -e -@16	510	8151
genozip -e -@16 -fast	362	5680

TABLE VIII: MiMyCS compresses H1/1 (FASTQ) with a lower energy consumption compared to Genozip

Software	Energy consumption (kJoules)			
	DRAM*	PKG*	DPUs†	Total
MiMyCS	9.12	20.37	7.16	🏆 36.65
genozip -e	15.38	41.66	N/A	57.04
genozip -e -fast	11.52	31.89	N/A	43.41
genozip -e -@16	25.26	61.70	N/A	86.96
genozip -e -@16 -fast	18.52	44.60	N/A	63.12

* Measured with Intel® RAPL
† Estimated with execution time

used during the compression step. Besides, we focused our efforts on accelerating the mapping and compression stages, and dedicated less time to optimizing the decompression implementation. With further work, we believe MiMyCS can get closer to Genozip regarding decompression performance. In the case of a FASTQ dataset, it is already faster than pigz.

VII. Conclusion

We presented MiMyCS, a short reads reference-based compressor that relies on a hardware-software co-design. It uses the processing-in-memory architecture from the UPMEM company and leverages this massively parallel environment by distributing the reference sequence and accelerating the mapping stage. The procedure is coupled to a Bloom filters-based dispatching mechanism to attempt to map reads only against the most relevant parts of the reference. This reduces greatly the overall number of sequence comparisons to compute and ensures a fast mapping search. In terms of compression speed, MiMyCS with 4 PiM DIMMs outperforms all the Genozip configurations we evaluated, even the fast mode with maximum core usage. Moreover, we estimated that our software has lower energy consumption. Measures also show it has lower CPU usage, which leaves room for potential computational work to be executed at the same time. Example applications include, for instance, offloading the CPU of a sequencing server and compressing newly produced datasets.

We envision three main aspects for future work. First, we could investigate how to scale the software with more PiM ranks. Our server is equipped with 20 DIMMs, but we used only 4 in our benchmarks. In our current implementation, distributing the reference sequence to more ranks results in underused MRAM space, more idling time for the DPUs, and thus worse performance. It would be more interesting to keep each MRAM almost full and have duplicate parts. This

TABLE IX: Genozip remains the fastest for decompressing H1/1

Software	Decompression time (s)	
	FASTQ	FASTA
MiMyCS	830	528
pigz	942	276
genozip	687	364
genozip -e	455	380
genozip -e -fast	🏆 415	🏆 189
genozip -e -@16	465	409
genozip -e -@16 -fast	434	356
LW-FQZip2	10,093	N/A

requires adapting the dispatcher to send each read to only one DPU among those with identical reference parts.

Secondly, we would like to investigate more closely the energy savings by measuring real consumption values with power meters. We indeed believe this will help to optimize the hardware-software co-design of MiMyCS for both performance and energy savings.

Finally, we plan to expand our use cases to the metagenomic field, and apply our compressor to human microbiome reads. References for such data, for instance the Unified Human Gastrointestinal Genome [22], require more memory space than the human genome and lead to increased memory-boundedness on a standard CPU architecture. We thus expect the processing-in-memory approach of MiMyCS to appear particularly relevant for such application cases.

References

[1] D. Lan, R. Tobler, Y. Souilmi, and B. Llamas, "Genozip: a universal extensible genomic data compressor," *Bioinformatics*, vol. 37, no. 16, pp. 2225–2230, Aug. 2021. [Online]. Available: https://doi.org/10.1093/bioinformatics/btab102

[2] "Introducing DRAGEN Original Read Archive (ORA)." [Online]. Available: https://developer.illumina.com/news-updates/introducing-dragen-original-read-archive-ora

[3] F. Devaux, "The true Processing In Memory accelerator." IEEE Computer Society, Aug. 2019, pp. 1–24. [Online]. Available: https://www.computer.org/csdl/proceedings-article/hcs/2019/08875680/1ehCtkMGwkU

[4] Z. Lu, L. Guo, J. Chen, and R. Wang, "Reference-based genome compression using the longest matched substrings with parallelization consideration," *BMC Bioinformatics*, vol. 24, no. 1, p. 369, Sep. 2023. [Online]. Available: https://doi.org/10.1186/s12859-023-05500-z

[5] S. Kumar, M. P. Singh, S. R. Nayak, A. U. Khan, A. K. Jain, P. Singh, M. Diwakar, and T. Soujanya, "A new efficient referential genome compression technique for FastQ files," *Functional & Integrative Genomics*, vol. 23, no. 4, p. 333, Nov. 2023. [Online]. Available: https://doi.org/10.1007/s10142-023-01259-x

[6] Y. Ji, H. Fa, H. Yao, S. Shao, M. Wu, H. Fang, Q. Liu, and S. Liu, "FQCSpark: Efficient Spark-based Parallel Compression Algorithm for FASTQ Genome Sequences," in *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, May 2022, pp. 671–676. [Online]. Available: https://ieeexplore.ieee.org/document/9776028

[7] S. Bai, J. Chen, Z. Lu, and W. Li, "Reference-Based Compression of FASTQ Data Using Longest Match Model," in *2021 4th International Conference on Information Communication and Signal Processing (ICICSP)*, Sep. 2021, pp. 598–603. [Online]. Available: https://ieeexplore.ieee.org/document/9611973

[8] W. Shi, J. Chen, M. Luo, and M. Chen, "High efficiency referential genome compression algorithm," *Bioinformatics*, vol. 35, no. 12, pp. 2058–2065, Jun. 2019. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty934

[9] A. Guerra, J. Lotero, J. Ã. Aedo, and S. Isaza, "Tackling the Challenges of FASTQ Referential Compression," *Bioinformatics and Biology Insights*, vol. 13, p. 1177932218821373, Jan. 2019, publisher: SAGE Publications Ltd STM. [Online]. Available: https://doi.org/10.1177/1177932218821373

[10] N. Jammula and S. Aluru, "ParRefCom: Parallel Reference-based Compression of Paired-end Genomics Read Datasets," in *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, ser. BCB '19. New York, NY, USA: Association for Computing Machinery, Sep. 2019, pp. 447–456. [Online]. Available: https://dl.acm.org/doi/10.1145/3307339.3342171

[11] Z.-A. Huang, Z. Wen, Q. Deng, Y. Chu, Y. Sun, and Z. Zhu, "LW-FQZip 2: a parallelized reference-based compression of FASTQ files," *BMC Bioinformatics*, vol. 18, no. 1, p. 179, Mar. 2017. [Online]. Available: https://doi.org/10.1186/s12859-017-1588-x

[12] Y. Liu, H. Peng, L. Wong, and J. Li, "High-speed and high-ratio referential genome compression," *Bioinformatics*, vol. 33, no. 21, pp. 3364–3372, Nov. 2017. [Online]. Available: https://doi.org/10.1093/bioinformatics/btx412

[13] C. Peng, Q. Deng, Z. Huang, Y. Sun, and Z. Zhu, "G-FQZip: Lossless Reference-Based Compression of FASTQ Files Using GPUs," in *2017 13th International Conference on Computational Intelligence and Security (CIS)*, Dec. 2017, pp. 553–556. [Online]. Available: https://ieeexplore.ieee.org/document/8288550

[14] S. Saha and S. Rajasekaran, "NRGC: a novel referential genome compression algorithm," *Bioinformatics*, vol. 32, no. 22, pp. 3405–3412, Nov. 2016. [Online]. Available: https://doi.org/10.1093/bioinformatics/btw505

[15] Y. Zhang, L. Li, Y. Yang, X. Yang, S. He, and Z. Zhu, "Light-weight reference-based compression of FASTQ data," *BMC Bioinformatics*, vol. 16, no. 1, p. 188, Jun. 2015. [Online]. Available: https://doi.org/10.1186/s12859-015-0628-7

[16] P. Li, X. Jiang, S. Wang, J. Kim, H. Xiong, and L. Ohno-Machado, "HUGO: Hierarchical mUlti-reference Genome cOmpression for aligned reads," *Journal of the American Medical Informatics Association: JAMIA*, vol. 21, no. 2, pp. 363–373, 2014.

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: https://dl.acm.org/doi/10.1145/362686.362692

[18] R. Leinonen, R. Akhtar, E. Birney, L. Bower, A. Cerdeno-TÃ¡rraga, Y. Cheng, I. Cleland, N. Faruque, N. Goodgame, R. Gibson, G. Hoad, M. Jang, N. Pakseresht, S. Plaister, R. Radhakrishnan, K. Reddy, S. Sobhany, P. Ten Hoopen, R. Vaughan, V. Zalunin, and G. Cochrane, "The European Nucleotide Archive," *Nucleic Acids Research*, vol. 39, no. suppl_1, pp. D28–D31, Jan. 2011. [Online]. Available: https://doi.org/10.1093/nar/gkq967

[19] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "RAPL in Action: Experiences in Using RAPL for Power Measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no. 2, pp. 9:1–9:26, Mar. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3177754

[20] Y. Falevoz and J. Legriel, "Energy Efficiency Impact of Processing in Memory: A Comprehensive Review of Workloads on the UPMEM Architecture," in *Euro-Par 2023: Parallel Processing Workshops*, D. Zeinalipour, D. Blanco Heras, G. Pallis, H. Herodotou, D. Trihinas, D. Balouek, P. Diehl, T. Cojean, K. Furlinger, M. H. Kirkeby, M. Nardelli, and P. Di Sanzo, Eds. Cham: Springer Nature Switzerland, 2024, pp. 155–166.

[21] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics (Oxford, England)*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009.

[22] A. Almeida, S. Nayfach, M. Boland, F. Strozzi, M. Beracochea, Z. J. Shi, K. S. Pollard, E. Sakharova, D. H. Parks, P. Hugenholtz, N. Segata, N. C. Kyrpides, and R. D. Finn, "A unified catalog of 204,938 reference genomes from the human gut microbiome," *Nature Biotechnology*, vol. 39, no. 1, pp. 105–114, Jan. 2021, number: 1 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/s41587-020-0603-3