DNA Mapping using Processor-in-Memory Architecture

Dominique Lavenier, Jean-Francois Roy, David Furodet

DNA Mapping using Processor-in-Memory Architecture

Dominique Lavenier
IRISA / CNRS
Rennes - France
lavenier@irisa.fr

Jean-Francois Roy, David Furodet
UPMEM
Grenoble - France
jroy@upmem.com

***Abstract* - This paper presents the implementation of a mapping algorithm on a new Processing-in-Memory (PIM) architecture developed by UPMEM Company. UPMEM's solution consists in adding processing units into the DRAM, to minimize data access time and maximize bandwidth, in order to drastically accelerate data-consuming algorithms. The technology developed by UPMEM makes it possible to combine 256 cores with 16 GBytes of DRAM, on a standard DIMM module. An experimentation of DNA Mapping on Human genome dataset shows that a speed-up of 25 can be obtained with UPMEM technology compared to fast mapping software such as BWA, Bowtie2 or NextGenMap running on 16 Intel threads. Experimentation also highlight that data transfer from storage device limits the performances of the implementation. The use of SSD drives can boost the speed-up to 80.**

Keywords - mapping; processing-in-memory; PIM; bioinformatics, genomic; I/O disk bandwidth; hardware accelerator.

I. INTRODUCTION

With the fast evolution of NGS technology (Next Generation Sequencing), mapping DNA sequences to complete genomes is now a daily bioinformatics task. However, it requires important computing power. Each sequencing run generates hundred of millions of short DNA sequences (from 100 to 250 bp length) that are compared with one or several reference genomes to extract new knowledge.

More specifically, the mapping process consists in aligning short fragments of DNA, to large sequences (typically full genomes). Contrary to BLAST-like alignments that locate any portion of similar subsequences, the mapping action performs a complete alignment of the DNA fragments on the target sequence, by adding constraints such as the maximum numbers of substitutions or insertion/deletion errors.

From a computer science point of view, the challenge is to be able to rapidly map hundreds of millions of short DNA fragments to full genomes, such as the Human Genome (3.2 x 10^9 bp). The output of a mapping is a list of coordinates, for each DNA fragments, where matches have been found. As genome structures are highly repetitive structures, a DNA fragments can match at many locations. A mapping quality is thus associated. The quality value depends of the mapping confidence on a specific region. The output is generally encoded using SAM/BAM format [1].

Many mappers are available [3] [4]. They have their own pros and cons depending of several criteria such as speed, memory footprint, multithreading implementation, sensitivity or precision. The BWA mapper [2], based on Burrows-Wheeler Transform, can be considered as a good reference since it is often used in many bioinformatics pipelines. Examples of other mappers are Bowtie [5], NextGenMap [10], SOAP2 [6], BFAST [7] or GASSST [8].

Mapping hundred of millions of short DNA sequences on complex genomes is time-consuming. It may take several hours of computation on a standard multicore processor. Thus, in order to significantly reduce the computation time of such treatment, many hardware implementations on GPU or FPGA accelerators have been proposed.

From the GPU side, a few software such as CUSHAW [12][13], BarraCUDA [14][15], MaxSSmap [16] or SOAP3-dp [17] have been developed (the list is not exhaustive). They provide interesting speed up compared to purely CPU-centric software. We may also cite NextGenMap [10] that can also use GPU resources (if available) to reduce runtime execution by 20-50%. Globally, average speed-up from 5 to 8 can be achieved compared to standard multicore processors. The great advantage of this solution is that GPU boards are cheap and can easily equip any bioinformatics servers.

From the FPGA side, several reconfigurable architectures projects [18][19][20] have proposed interesting approaches. The mapping core engines have high potentiality due to aggressive hardware customization. Unfortunately, the bioinformatics community cannot leverage these developments, because the hardware is not available. That said, the TimeLogic Company commercializes the VelocciMapper, a very fast mapping proprietary solution on their FPGA-DeCypher platform [21] that seems promising.

More recently the Edico-Genome Company has developed a custom VLSI chip, called DRAGEN [22], mainly dedicated to the mapping of DNA sequences, even if it can performs other bioinformatics task. With this chip, standard bioinformatics pipelines that intensively used mapping software (such as GATK [23]) can be highly speed up.

This paper presents an alternative solution based on Processing-in-Memory (PIM) concept. The PIM concept is not new. In the past, several research projects have explored the

potentialities of building processing units as close as possible to the data. The Berkley IRAM project [24] probably pioneers this kind of architecture to limit the Von Neumann bottleneck between the memory and the CPU. The PIM project of the University of Notre Dame [25] was also an attempt to solve this problem by combining processors and memories on a single chip.

UPMEM solution solves the same problem by building DIMM modules integrating high density DRAM and RISC processors. The idea is to complement the main memory of a multicore processor with such smart modules. Data within these modules can be processed independently by activating in-memory computing power, releasing the pressure on the CPU-memory transactions.

The DNA mapping task perfectly illustrates how such time-consuming application can benefit from the PIM architecture. DNA sequences have to be compared with thousands of locations within a reference genome. Deporting this activity directly to the PIM-DRAM module, and parallelizing the whole process to hundreds of PIM cores, avoids a lot of CPU-memory transactions compared to a standard multithreaded solution.

The rest of the paper is structured as follows: the next section briefly describes the main features of the UPMEM solution. Section 3 details how the mapping process is implemented on the UPMEM architecture. Section 4 evaluates the performances and section 5 compares with current mapping software. Section 6 concludes the paper.

II. UPMEM Architecture Overview

UPMEM technology is based on the concept of Processing-in-Memory (PIM). The basic idea is to add processing elements next to the data, i.e. in the DRAM, to maximize the bandwidth and minimize the latency. Host processor is acting as an orchestrator: It performs read/write operations to the memories and commands/controls the co-processors embedded in the DRAM. This data-centric model of distributed processing is optimal for data-consuming algorithms.

UPMEM PIM-DRAM solution can be packaged on 16GByes DIMM modules with 256 processors: One processor every 64 MBytes of DRAM. Each processor can run its own independent program. In addition, to hide memory latency, these processors are highly multithreaded (up to 24 threads can be run simultaneously) in such a way that the context is switched at every clock cycle between threads.

The UPMEM processor, called DPU (DRAM Processing Unit), is a triadic RISC processor with 24 32-bits registers per thread. In addition to memory instructions, it comes with built-in atomic instructions and conditional branching bundled with arithmetic and logic operations.

From a programing point of view, two different programs must be specified: (1) the host program that will dispatch the data to the co-processors memory, sends commands, input data, and retrieve the results; (2) the program that will execute the treatment on the data stored in the PIM DRAM. This is often a short program performing basic operations on the data. It is called a *tasklet*. Note however, that the architecture of the UPMEM DPU allows different tasklets to be specified and run concurrently on different blocks of data.

Depending on the server configuration (i.e. the number of 16 GBytes UPMEM PIM-DRAM modules), a large number of DPU can process data in parallel. Each DPU only accesses 64 MBytes and cannot directly communicate with its neighbors. Data exchanges, if needed, must go through the host processor. A DPU has a fast working memory (64 Kbytes) acting as cache/scratchpad memory and shared by all tasklets (threads) running on the same DPU. This memory working space can be used to transfer blocks of data from the DRAM, and can be explicitly managed by the programmer.

To sum up, programing an application consists in writing a main program (run on the host processor) and one or several tasklets that will be executed on the DPUs. The main program has to synchronize the data transfer to/from the DPUs, as well as the tasklet execution. Note that the tasklet execution can be run asynchronously with the host program, allowing host tasks to be overlapped with DPU tasks

Recently, UPMEM conducted a proof of concept project to validate the technical feasibility of the DPU core on a DRAM process. Indeed, DRAM manufacturing processes are optimized in cost and bitcell density, and have never been designed to enable computing logic.

III. Mapping on Upmem

A. Overview

This section presents the mapping strategy elaborated to fully exploit the PIM architecture. The main idea is to distribute an indexing structure (computed from the genome) across the DPU memories. The host processor receives the DNA sequences and, according to specific k-mer features, dispatches them to the DPUs. To globally optimize the treatment, group of DNA sequences are sent to the DPUs before starting the mapping process. Results are sent back to the host processor. DNA sequences that have not been mapped are reallocated to other DPUs for further investigation. A three pass processing allows more than 99% of DNA sequences to be mapped. This strategy supposes first to have downloaded the complete index into the DPU memory.

The following algorithm illustrates how the overall mapping process:

1: Distribute the genome index across the DPUs
2: Loop N
3: List L_{IN} ← P x DNA sequences
4: Loop 3
5: Dispatch sequences of list L_{IN} into DPUs
6: Run mapping process
7: Get results → 2 lists: L_{GOOD} & L_{BAD}
8: Output L_{GOOD}
9: L_{IN} ← L_{BAD}

The first loop (line 2) performs N iterations. N is the ratio of the number of DNA sequences to map divided by the number of DNA sequences that is processed in a single iteration. Typically, a single iteration processes 10^6 sequences. The second loop (line 4) dispatches the sequences of the list L_{IN} into the DPUs. In the first iteration, the list L_{IN} contains all the DNA sequences. The mapping (line 6) is run in parallel and provides a mapping score (and coordinates) for all DNA sequences. The results are split into two lists (line 7): a list of sequences with good scores (list L_{GOOD}) and a list with bad scores (list L_{BAD}). Based on new k-mers, the list L_{BAD} is dispatched to the DPUs in the 2nd and 3rd iterations. The following figure illustrates the mapping process:

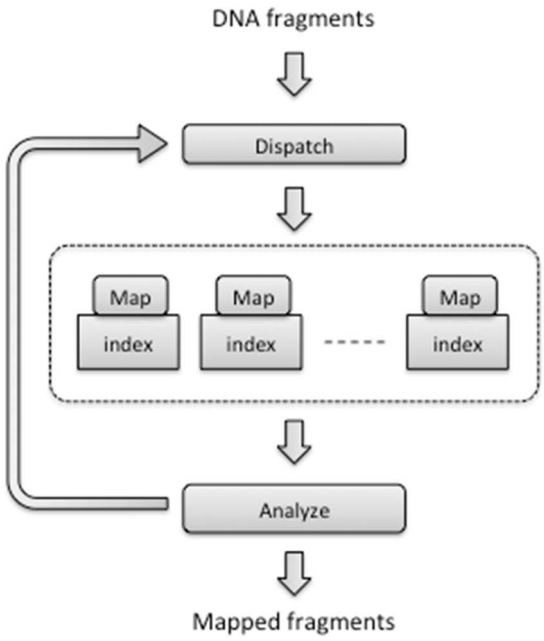

Figure 1: According to their k-mer composition, DNA sequences are dispatched among the UPMEM memories that house a distributed index of the genome. Mapping is run independently on all parts of the index. DNA sequences that have not been mapped are dispatched again to the index according to other k-mer criteria. After 3 rounds, more than 99% of the DNA sequences are mapped.

B. Genome Indexing

To speed up the mapping and to avoid to systematically comparing the DNA sequences with the full text of the genome, the genome is indexed using words of k characters, called k-mers. For each k-mers a list of coordinates specifying its location is attached, typically the chromosome number and the position of the k-mer on that chromosome. Then, the mapping process consists in extracting one or several k-mers from the DNA sequences in order to rapidly locate its position on the genome. The k-mer acts more or less as an anchor from which a complete match can be computed.

The index is composed of a first table of 4^K entries (Index1) that provides for all possible k-mer a list of coordinates where it occurs in the genome. The list of coordinates is stored in a second table (Index2). More specifically, for a specific k-mer, Index1 gives its address in Index2 and its number of occurrences. A line in Index2 indicates the chromosome number and a position on that chromosome.

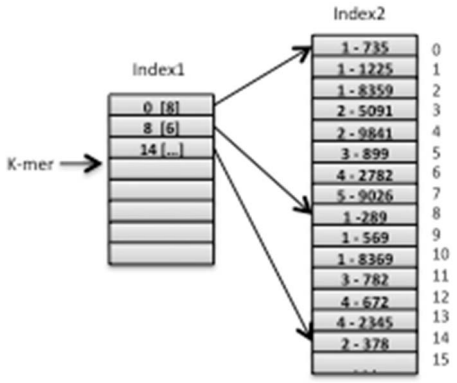

Figure 2: Index1 provides for all possible k-mer its number of occurrences and an entry in Index2. Index2 is a list of coordinates specifying the chromosome number and a position.

The UPMEM implementation split Index2 into N parts, N being the number of available DPUs. As each DPU has a limited memory (64 MBytes), it cannot store the complete genome. Consequently, k-mer positions along the genome are useless inside a DPU without additional information. Thus, in addition to coordinates, portions of genome text corresponding to the neighborhood of the k-mers are memorized. The global indexing scheme is shown below.

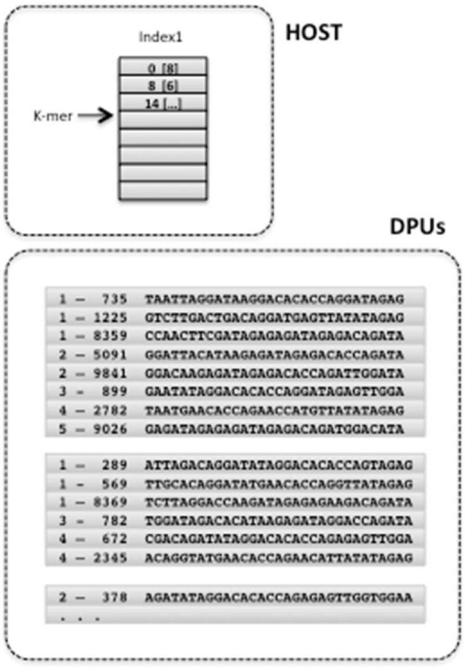

Figure 3: Index1 is stored on the host computer. Index2 is distributed among the DPU memory. Neighborhood information is added to directly performed the mapping analysis.

Thus, for one k-mer, a line of Index2 memorizes the chromosome number (1 Bytes), the position of the k-mer on the chromosome (4 Bytes) and a neighborhood of 180 bp where each nucleotide is 2-bit encoded (45 Bytes). The storage one 1 k-mer requires 50 Bytes. Inside a DPU, 50 MBytes are allocated for the storage of the index or, in other words, the capability to store an equivalent genome of 1 Mbp. The rest of the memory is used for DNA sequences and result transfers.

C. *Mapping algorithm*

The host processor receives a flow of DNA sequences. For each sequence, a k-mer corresponding to the k first characters is extracted. Based on this k-mer, the DNA sequence is dispatched to the corresponding DPU. Every P sequences ($P = 10^6$), the host activates the DPUs to start the mapping process of the DNA sequences stored in each DPU.

More precisely, a specific DPU receive an average of $Q = P/N$ DNA sequences. The mapping consists in comparing these Q sequences with the portions of the genome text stored inside each DPU memory, knowing that the k first characters are identical. The comparison algorithm can be more or less complex depending of the required mapping quality. For stringent mapping allowing only substitution errors, a simple Hamming distance can be computed. For mapping with insertion/deletion errors, banded smith and Waterman algorithm can be performed. A detailed implementation, and the tasklet code, can be found in [28].

However, this strategy doesn't guaranty to find all mapping locations. If an assembly error occurs along the k first characters, the DNA sequence will be dispatched to the wrong DPU and no correct mapping will be detected. Thus, for DNA sequences with a low score, the next k characters are taken into consideration to form a new k-mer allowing a new dispatching. If again, no good score are computed the next k characters are considered. Practically, after 3 iterations, the best matches are systematically found.

D. *Post processing*

As the mapping is fully performed inside the DPUs, no more computation is required. The post processing consists simply in getting the results from the DPUs and formatting the data (BAM/SAM format for example) before writing them to disk

IV. Performance Evaluation

Performances have been evaluated with a DELL server (Xeon Processor E5-2670, 40 cores 2.5 GHz, 64 GBytes RAM) configuration running Linux Fedora 20. In our implementation, I/O transfer has a great impact on the overall performances, and thus, the hard disk read speed is an important parameter. We measure an average bandwidth of 130 MB/s (local disk).

As the UPMEM memory devices are not yet available, estimation are done with the UPMEM Cycle Accurate Simulator (CAS) developed by the Company. Tasklet programs are written in C and compiled (with a specific compiler) for the DPU target processors. Binaries are directly executed by the CAS simulator.

The CAS is fully consistent with the hardware version as it actually represents the reference design. The CAS is used for intensive testing of the design because it is faster and easier to execute with the test suite. The gap is null between the 2 versions, and this is systematically verified by a specific qualification process. As a consequence, the CAS execution cycle count exactly reflects what the real hardware will produce.

Performances have been evaluated on the following dataset:

- Human Genome (3.2 Gbp)
- DNA sequences: a set of 111 x 106 100 bp sequences (13 GBytes)

To store the index corresponding to the Human genome, the minimum number of DPUs is equal to $3.2 \times 10^9 / 10^6 = 3200$ DPUs (cf. previous section: a DPU store an index that represents the equivalent of only 1Mbp). The UPMEM configuration is thus set to 3328 DPUs (13 DIMM modules). In that situation, the UPMEM memory is equal to 208 GB.

We evaluate the execution time according to the algorithm of section 3:

1. Distribution of the genome index across the DPUs
2. Loop:
    a. Dispatching of the sequence to the DPUs
    b. Mapping
    c. Result analysis

A. *Distribution of the genome index acroos the DPUs*

This step can be divided into the two following actions:

- Download the index from the storage device
- Dispatch the index inside the DPUs

As most of the mappers, the index is pre-computed. In our case, Index1 is fully pre-computed, and the genome is formatted to facilitate its encoding into the DPU memories. The size of Index1 is determined by the k-mer length. Here, the k-mer length is set to 13. The numbers of entries of Index1 is thus equal to $4^{13} = 64$ M entries. One entry stores the number of k-mers (1 integer) and an address in index2 (1 integer). Thus the total size of Index1 is 256 MBytes. This index is stored into the host computer memory and required 2 sec to be downloaded (disk bandwidth = 130 MB/s). The size of the Fasta file containing the genome is equal to the size of the genome (3.2 GBytes). The download time is equal to 25 sec.

Dispatching Index2 across the DPUs consists in writing for each k-mers of the genome 48 bytes in a DPU memory, that is globally $3.2 \times 10^9 \times 48 = 153.6$ GBytes. The bandwidth for transferring data form the Host memory to the DPU memories is estimated to 11.53 GB/s (see [11]). The time for transferring the index is thus equal to $153.6/11.53 = 13.3$ sec. With the associated overhead to format the data, we globally estimate this step to 15 sec.

Actually, downloading the genome and dispatching the index into the DPUs are overlapped and practical time measurements of this initialization step (T_{INIT}) are under 30 sec.

B. Loop execution

The loop performs the following actions:

1. Get block of DNA sequences from disk.
2. Dispatch DNA sequences to DPUs
3. Initialize and start the DPUs
4. Perform the mapping
5. Collect results from DPU
6. Analyze and write results

For each action we detail how the execution time is determined.

1. Get block of DNA sequences from disk

In our implementation, the loop iteration processes 10^6 DNA sequences. These sequences are read from the local disk. One million of DNA sequences of length 100 in Fasta format represent approximately 130 MBytes of data (text sequence + annotation). The time to read this information depends again of the I/O bandwidth of the storage device. With a bandwidth of 130 MB/sec, the time T_1 = is equal to **1 sec**.

2. Dispatch DNA sequences to the DPU:

Dispatching the DNA sequences to the DPUs is fast: it consists in coding the 13 first characters of the sequence and in copying the sequence to the target DPU. Experiments indicates an execution time < 40 ms. Transferring 100 MBytes of data (10^6 sequences of 100bp) to the DPU memory is also very fast. It requires 0.1/11.5 = 8.7 ms. Overall, this step takes a maximum of T_2 = **50 ms**.

3. Initialize and start the DPU

A DPU runs 10 tasklets. Each tasklet receives two parameters: the number of DNA sequences to process, and the address where these fragments are stored. This represents 2 integers (8 bytes) by tasklet, or 80 bytes per DPU, or an overall transfer of 80x3328 = 266240 bytes. The equivalent time T_3 is: $266240/11.53 \times 10^9$ = **23 µs**. As broadcasting commands to 128 DPU simultaneously is possible, booting the DPU consist in sending 3328/128 = 26 commands. This time is negligible.

4. Mapping

On average, a DPU receive $10^6/3328$ = 300 DNA sequences to process (3328 is the number of available DPUs). The number of occurrences of a k-mer of size 13 is approximately the size of the genome divided by 4^{13}, that is $3.2 \times 10^9/4^{13} = 50$. The number of mappings that must be executed by one DPU is thus equal to 15000 (300 x 50). The simulations executed on the UPMEM Cycle Accurate Simulator range from 10×10^6 to 25×10^6 cycles to perform such a treatment, depending on the DPU load. As a matter of fact, the repartition inside the DPUs is not uniform. It depends of the nature of the DNA sequences. We have to take into account the worst execution time since all DPUs must finish before analyzing all results.

In the second and third round, only a fraction of the DNA sequences that have not matched are sent to other DPUs. It represents less than 10% of the initial number of sequences. The impact on the overall execution time is weak. An upper bound estimation for the 3 loop iteration is 30×10^6 cycles, leading to an execution time T_4 of **40 ms** with a 750 MHz DPU processor frequency.

5. Collect results

For each DNA sequences, the DPU output the following information: genome coordinates and mapping scores (2 integers). There are thus $2 \times 4 \times 10^6$ = 8 M bytes to transfer. The transferring time T_5 = **0.7ms.**

6. Analysis & write results

This step that is run on the host processor evaluates the score of the mapping and selects DNA sequences that have to be analyzed again. It also writes results to the output file. Our experimentation estimates the execution time T_6 to 0.1 sec in the worst case.

Actions 2 to 6 are iterated 3 times. The first time involves all DNA fragments, the second time less than 10% and the third time less than 3%. The cumulated execution time of actions 2 to 6 is thus approximately equal to:

$$T_{2\text{-}6} = 1.13 \times (50 + 40 + 100) = \textbf{190 ms}.$$

Actually, getting the data from the disk (action 1) can be overlapped with the other tasks (actions 2 to 6), leading to an the following T_{LOOP} execution time:

$$T_{LOOP} = \max(T_1, T_{2\text{-}6}) = \textbf{1sec}$$

C. Overall Execution time

The overall execution time **T** for mapping 111×10^6 DNA sequences to the Human genome is approximately given by:

$$T = T_{INIT} + 111 \times T_{LOOP} = 30 + 111 \times 1 = \textbf{141 sec}.$$

The general execution scheme is as follows:

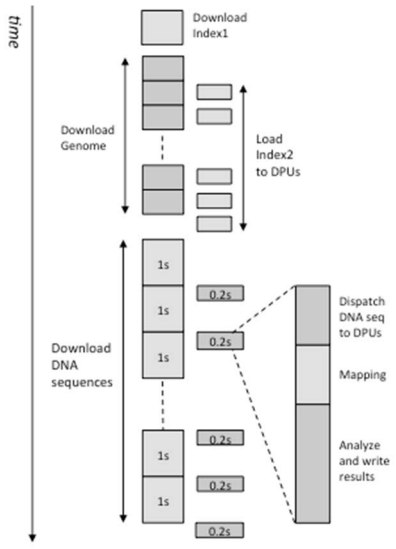

Figure 4: Scheduling of the different tasks. Loading of the data and computation are overlapped. In the implementation, data transfers dominate the overall process.

As we can see, the T_{LOOP} execution time is mainly constrained by the I/O disk bandwidth: loading 10^6 100 bp DNA sequences requires about 1 second while processing these data takes only 0.2 second.

A possible hardware optimization is to use SSD storage devices that have larger throughput. We tested with 512 GB SSD drive present on the server with an average bandwidth of 700 MB/sec. In that case, the time for distributing the index is now dominated by the dispatch index (~15s). For the loop execution time, a good balance is achieved between the time for getting the DNA sequences from SSD (~185 ms) and the time for executing actions 2 to 6 (~200 ms).

In that situation the new overall execution time is given by:

$$T = 15 + 111 \times 0.2 = \mathbf{37.2\ sec}.$$

V. Comparison with other mappers

To evaluate the speed up brought by the UPMEM technology, we compared the execution time with the following mappers:

- BWA [2]
- Bowtie2 [5]
- NextGenMap [10]

The three software have been run with different number of threads: 8, 16 and 32 and with their default parameters (details of the experimentation can been found in [28]). The following table gives the execution time.

	8 threads	16 threads	32 threads
BWA	5901	3475	2191
Bowtie2	5215	2916	2241
NextGenMap	3485	2104	1552

Table 1: Execution time (in second) of the mapping of 111 millions of 100 bp DNA sequences with the Human Genome. The three software have been run on a DELL server with the following characteristics: Xeon Processor E5-2670, 40 cores 2.5 GHz, 64 GBytes RAM.

The speed-up is calculated as the ratio between the reference software execution time and the estimated UPMEM execution time. It considers both hard and SSD disks.

	8 threads		16 threads		32 threads	
	HARD	SSD	HARD	SSD	HARD	SSD
BWA	41	157	24	93	15	58
Bowtie2	36	140	20	78	16	60
NextGenMap	24	93	15	56	11	41

Table 2: Speed-up of the UPMEM mapping implementation compared to 3 software executions

The three software have been run with and without SSD storage devices. We didn't detect any significant difference in the execution time. We end up with the same conclusion of Lee et al. [29]: mapper software don't benefit of SSD performances.

VI. Conclusion

UPMEM PIM technology is a data-centric hardware accelerator. As opposed to GPU, FPGA and custom VLSI chips that focus on powerful processing units, the computational power of Processing-In-Memory is brought by the large and scalable number of independent processing elements, each one being composed of a processing unit and a DRAM bank. The Van Neumann bottleneck is pushed away, and embarrassingly parallel applications can highly benefit from this architecture by distributing computations across processing elements.

On the genomic side, many other treatments are potential good candidates for an efficient implementation on UPMEM. An implementation study on the well-known Blast software [26] has shown an expected speed-up of 25 compared to a server running 20 Intel cores [27]. There is also a lot of room for implementing many NGS analysis such as short read or long read correction, genome assembly (especially large k-mer counts), GWAS studies, etc. The difficulty is how to split the problem into thousands of tasklets, each of them working independently on a small part of the data.

Performances of hardware accelerators are tightly correlated to their computing infrastructure environment. For the mapping problem where huge volumes of data have to be processed, performances are clearly restrained by data access. In our case, the bandwidth of the hard disk drive is a critical bottleneck. SSD technology can help to increase data transfer. Feeding optimally such accelerators is probably the main problem, especially for large bio-informatics centers where data are stored on large centralized storage devices of several tenths of Tera Bytes. Servers that house hardware accelerators must have a privileged mass-storage connection to keep all their potential computing power.

The SDK of UPMEM DPU has been made available ahead of silicon to enable the porting of applications and anticipate the potential benefit of using such architecture. The SDK comes with a C-compiler, a simulator and the APIs needed to build a full application and will continue to be enhanced. It is widely open to the community. In parallel, UPMEM is partnering with DRAM manufacturers to build silicon chips assembled on DIMM modules, with a prototyping cycle in 2017.

References

[1] The SAM/BAM Format Specification Working Group, Sequence Alignment/Map Format Specification, April 2015, https://samtools.github.io/hts-specs/SAMv1.pdf

[2] Li H. and Durbin R. (2009) Fast and accurate short read alignment with Burrows-Wheeler Transform. Bioinformatics, 25:1754-60.

[3] Jing Shang, Fei Zhu, Wanwipa Vongsangnak, Yifei Tang, Wenyu Zhang, and Bairong Shen, Evaluation and Comparison of Multiple Aligners for Next-Generation Sequencing Data Analysis, BioMed Research International, vol. 2014, Article ID 309650, 16 pages, 2014.

[4] Schbath S, Martin V, Zytnicki M, Fayolle J, Loux V, Gibrat J-F. Mapping Reads on a Genomic Sequence: An Algorithmic Overview and a Practical Comparative Analysis. Journal of Computational Biology. 2012;19(6):796-813. doi:10.1089/cmb.2012.0022.

[5] Langmead B. Trapnell C. Pop M., et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 2009;10:R25.

[6] Li R. Yu C. Li Y., et al. SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics. 2009;25:1966–1967

[7] Homer N. Merriman B. Nelson SF. BFAST: an alignment tool for large scale genome resequencing. PLoS ONE. 2009;4:e7767

[8] Rizk G. Lavenier D. GASSST: global alignment short sequence search tool. Bioinformatics. 2010;26:2534–2540

[9] Ayat Hatem, Doruk Bozdağ, Amanda E Toland, Ümit V Çatalyürek, Benchmarking short sequence mapping tools, BMC Bioinformatics 2013, 14:184

[10] Fritz J. Sedlazeck, Philipp Rescheneder, and Arndt von Haeseler NextGenMap: fast and accurate read mapping in highly polymorphic genomes.Bioinformatics (2013) 29 (21): 2790-2791 first published online August 23, 2013 doi:10.1093/bioinformatics/btt468

[11] UPMEM DPU-Data exchange with main CPU. UPMEM Technical note, version 1.3

[12] Y. Liu, B. Schmidt, D. Maskell: CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform, Bioinformatics, (2012) 28(14): 1830-1837

[13] Y. Liu, B. Schmidt: CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing. IEEE Design & Test of Computers 31(1):31-39, 2014

[14] Klus P, Lam S, Lyberg D, Cheung MS, Pullan G, McFarlane I, Yeo GSH, Lam BY. (2012) BarraCUDA - a fast short read sequence aligner using graphics processing units. BMC Research Notes, 5:27.

[15] Langdon WB, Lam BY, Petke J, Harman M. (2015) Improving CUDA DNA Analysis Software with Genetic Programming. Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation - GECCO '15

[16] Turki Turki and Usman Roshan, MaxSSmap: A GPU program for mapping divergent short reads to genomes with the maximum scoring subsequence, BMC Genomics, 15(1)1:969, 2014

[17] Luo R, Wong T, Zhu J, Liu C-M, Zhu X, et al. (2013) SOAP3-dp: Fast, Accurate and Sensitive GPU-Based Short Read Aligner. PLoS ONE 8(5)

[18] C. B. Olson et al., "Hardware acceleration of short read mapping," Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on IEEE, pp. 161-168, 2012. J.

[19] Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," In Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on, pp. 210-217, IEEE, 2013.

[20] J. Arram, et al. "Leveraging FPGAs for Accelerating Short Read Alignment." IEEE/ACM Transactions on Computational Biology and Bioinformatics. 2016.

[21] http://www.timelogic.com/catalog/799/velocimapper

[22] http://www.edicogenome.com/dragen/

[23] McKenna A, Hanna M, Banks E, Sivachenko A, Cibulskis K, Kernytsky A, Garimella K, Altshuler D, Gabriel S, Daly M, DePristo MA, The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data, 2010 GENOME RESEARCH 20:1297-303

[24] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. (1997). "A Case for Intelligent RAM: IRAM," *IEEE Micro,* **17** (2), pp. 34–44

[25] Kogge, P. M., T. Sunaga and e. a. E. Retter (1995). Combined DRAM and Logic Chip for Massively Parallel Applications. 16th IEEE Conf. on Advanced Research in VLSI, Raleigh, NC

[26] Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W. & Lipman, D.J. (1997) "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." Nucleic Acids Res. 25:3389-3402.

[27] Dominique Lavenier, Charles Deltel, David Furodet, Jean-François Roy. BLAST on UPMEM. [Research Report] RR-8878, INRIA, 2016.

[28] Dominique Lavenier, Charles Deltel, David Furodet, Jean-François Roy. MAPPING on UPMEM. [Research Report] RR-8923, INRIA, 2016

[29] Sungmin Lee, Hyeyoung Min, Sungroh Yoon. Will solid-state drives accelerate your bioinformatics? In-depth profiling, performance analysis and beyond. Briefing in Bioinformatics 2015; 17 (4): 713-727.