Enabling Fast and Energy-Efficient FM-index Exact Matching Using Processing-Near-Memory

Jose M. Herruzo · Ivan Fernandez ·
Sonia González-Navarro · Oscar Plata

Received: date / Accepted: date

Abstract Memory bandwidth and latency constitutes a major performance bottleneck for many data-intensive applications. While high-locality access patterns take advantage of the deep cache hierarchies available in modern processors, unpredictable low-locality patterns cause a significant part of the execution time to be wasted waiting for data. An example of those *memory bound* applications is the exact matching algorithm based on FM-index, used in some well-known sequence alignment applications. *Processing-Near-Memory* (PNM) has been proposed as a strategy to overcome the *memory wall* problem, by placing computation close to data, speeding up memory bound workloads by reducing data movements.

This paper presents a performance and energy evaluation of two classes of processor architectures when executing the FM-index exact matching algorithm, as a reference algorithm for exact sequence alignment. One architecture class is processor-centric, based on complex cores and DDR3/4 SDRAM memory technology. The other architecture class is memory-centric, based on simple cores and ultra high-bandwidth Hybrid Memory Cube (HMC) 3D-stacked memory technology. The results show that the PNM solution improves performance between 1.26× and 3.7× and the energy consumption per operation is reduced between 21× and 40×.

In addition, a synthetic benchmark `RANDOM` was developed that mimics the memory access pattern of the FM-index exact matching algorithm, but with

Jose M. Herruzo
E-mail: jmherruzo@uma.es

Ivan Fernandez
E-mail: ivanfv@uma.es

Sonia González-Navarro
E-mail: sonia@ac.uma.es

Oscar Plata
E-mail: oplata@uma.es
Department of Computer Architecture, University of Málaga, 29071 Málaga, Spain

a user configurable operational intensity. This benchmark allows us to extend the evaluation to the class of algorithms with similar memory behaviour but running over a range of operational intensity values.

Keywords FM-index · short-read alignment · random memory access patterns · high-bandwidth memory · processing-near-memory

1 Introduction

In recent years, the computer architecture community has witnessed a growing trend towards the development of applications that process large datasets [1]. These applications emerge in various areas, such as scientific and engineering computing, and big data analytics, often using high performance computing (HPC) techniques to achieve the required performance/power ratios.

In a contemporary computer, processing units and main memory are often located far from each other. As a result, data must traverse substantial hardware logic when moving from memory to the corresponding processing unit to perform an operation on it. This data movement costs time and energy. Large-scale data intensive workloads cause the movement of large amounts of data across the memory channels, resulting in saturation and poor performance due to their limited bandwidth. Moreover, these memory accesses heavily contribute to the energy consumption, since the power consumed by a memory access is higher than an arithmetic operation [2].

The presence of large and deep on-chip cache hierarchies is very helpful in reducing the impact of this problem for many data-intensive applications, specially for those that exhibit enough temporal/spatial locality when accessing data. However, many other computing workloads exhibit memory accesses in a random and unpredictable way, making these cache hierarchies almost useless. As an example, some applications in the bioinformatics field, like sequence alignments based on FM-index, present this unpredictable memory access pattern [3].

Emphasizing this problem, the amount of data intensive applications has grown in almost every area of science and technology, requiring computational resources to be up to the task. Continuing with the previously considered bioinformatics application, a number of high-throughput sequencing systems have appeared in industry over the past years. These systems are able to produce huge amounts of short reads per day of operation. For instance, the Illumina NovaSeq 6000 [4] sequence system is able to produce up to 6 Terabits of data, which need to be processed as fast as possible. Another example is the huge increase in sensor data and the rise of the Big Data [5], which produces huge amounts of unstructured information. This data is often accessed with unpredictable access patterns.

The described behaviour found in many memory intensive applications impacts on performance and energy in a very relevant way. As a consequence, computer architects are encouraged to explore new memory architectures and paradigms able to reduce both the energy and the execution time when processing a specific workload. In this sense, some new trending memory technologies provide the opportunity to explore the *Processing-Near-Data* (PNM) paradigm [6, 7]. PNM promises to address the above problems, effectively improving the performance and energy efficiency of computing systems, being specifically focused on the memory side.

The FM-index exact matching algorithm for sequence alignment, that performs fast exact matches of short reads to large genomes, is taken as a case study in this paper, as its low operational intensity (being memory-bound) and unpredictable memory access pattern make it a good candidate for near-memory processing.

In general, this paper presents a performance and energy evaluation of two classes of processor architectures when executing different implementations of the FM-index exact matching algorithm, as a reference algorithm for exact sequence alignment. One architecture class is processor-centric, based on high-performance out-of-order cores, including a large and deep cache hierarchy and using DDR3/4 SDRAM memory technology. The other architecture class is memory-centric, based on lightweight in-order cores, including a small and simple cache and using ultra high-bandwidth Hybrid Memory Cube (HMC) [8] 3D-stacked memory technology.

Although the focus is on the exact matching algorithm, however it is known that its type of memory access pattern is common in many other sequence alignment applications. For this reason, we developed a new synthetic benchmark (`RANDOM`) that mimics the memory access pattern of the FM-index exact matching algorithm, but with a user configurable amount of computing operations. This benchmark allows us to extend the previously described evaluation to the class of applications with similar memory behaviour but running over a range of operational intensity values.

The main contributions of the paper can be summarized as follows:

- We define an evaluation framework based on representative commodity system architectures and a PNM architecture. The conventional systems are based on powerful cores with a deep and large cache hierarchy and SDRAM memory technology, while the PNM system is based on lightweight cores with a single and small cache and HMC 3D-stacked memory technology.
- We analyze the performance and energy efficiency of the FM-index exact matching algorithm in the defined evaluation framework. We compare typical commodity computing systems against the PNM architecture, observing that the PNM system obtains between 1.26× and 3.7× better performance and between 21× and 40× better power consumption.
- We develop the `RANDOM` benchmark that mimics the memory pattern of the FM-index exact matching algorithm. In addition, this benchmark is able to perform a variable number of computing operations per each data loaded from main memory (i.e., the operational intensity can be configured).
- We analyze the performance and energy efficiency of the `RANDOM` benchmark in the evaluation framework. The aim is to evaluate algorithms with

similar memory patterns as the FM-index exact matching algorithm but for different operational intensities. In this analysis, the `STREAM` benchmark is also considered as a baseline representation of applications with uniform memory access patterns.

2 Sequence Alignment Based on FM-index

It is usual that the first step in genomic sequencing corresponds to sequence alignment, where sequence reads must be aligned to a genomic reference to identify regions of similarity [9].

In the case of large genomes, memory requirements become a big concern. As a result, big efforts were devoted to design sequence alignment algorithms based on FM-index [10], a structure specially suited for fast exact matches of short reads to large genomes keeping a small memory footprint.

The searching process using FM-index exhibits irregular and unpredictable memory access patterns. Each step of the algorithm accesses a section of the index that it is not known in advance, making the cache hierarchy difficult to exploit. Besides, the exact matching algorithm is a memory bound problem due to its low operational intensity.

2.1 FM-index

Let T be a character string of length n drawn from an alphabet defined as $\Sigma = \{A, C, G, T\}$. As a notation, let $T[i]$ and $T[i..r]$ denote the i-th character of T and the substring of T from character i to character r, respectively. The suffix array SA of T is an array of length n containing the starting positions of all suffixes of T in lexicographical order.

The Burrows-Wheeler Transform (BWT) of T is another string of length $n + 1$, denoted by T^{bwt}, obtained as follows: (1) append to the end of T the symbol \$, which is lexicographically smaller than any symbol in Σ; (2) form a conceptual $(n{+}1) \times (n{+}1)$ matrix M whose rows are the cyclic shifts of $T\$$ sorted in lexicographical order; and (3) the last column of the matrix M is T^{bwt} [11].

The FM-index of T is a compressed full-text index based on T^{bwt}. With the help of two auxiliary data structures, the FM-index supports an efficient searching of a pattern Q in T.

The FM-index was designed as a compressed structure such that the index size can be smaller than the original text. However, in the context of sequence alignment, it is usually not compressed in order to achieve better performance [9].

```
Algorithm BS: Backward Search
Input: C[], Occ[], Q query, n=|T|, m=|Q|
Ouput: (sp,ep): Interval pointers of Q in T
1: sp = C[Q[m]], ep = C[Q[m]+1]
3: for i from m-1 to 1 step -1
4:     sp = LF(Q[i],sp)
5:     ep = LF(Q[i],ep)
6: end for
7: return (sp+1,ep)
```

Fig. 1 Backward search algorithm.

2.2 Exact Matching: Backward Search

Given a pattern Q of length m, $m<n$, the FM-index allows to find all occurrences of Q in the text T [10]. The search process takes two steps: *Count* and *Locate*. The first step is a rank query process that calculates the number of occurrences of Q in T by identifying the first and last rows of matrix M (see sec. 2.1) prefixed by the query Q. The *Locate* step uses the indexes of these rows to access the suffix array, where it finds the position of every occurrence of Q in the text T.

Figure 1 shows the *backward search* (BS) algorithm that implements the *Count* step. Each iteration of the loop 3−6 in BS accesses the query string and makes two calls to the $LF()$ function, one with *sp* (start pointer) and the other with *ep* (end pointer). Note that in every loop iteration, *sp* and *ep* are updated using the value computed in the previous iteration. That constitutes two dependency chains of calls to $LF()$, one for *sp* and the other for *ep*.

The main operation in the BS algorithm is a *Last-to-First Mapping* (LFM), which is performed by calling the function $LF()$, defined as follows:

$$LF(Q[i], p) = C[Q[i]] + Occ[Q[i], p], \tag{1}$$

where i is the index of the loop and p is either *sp* or *ep*. $C[c]$ represents the number of occurrences in T^{bwt} of the symbols alphabetically smaller than c, while the function $Occ(c, p)$ denotes the number of occurrences of symbol c in the prefix $T^{bwt}[1..p]$ (i.e., $Occ()$ is a rank query).

The suffix array is usually a very large compressed data structure. However, its size for the human genome is about 12 GB (3 Gigabases x 4 Bytes), so it can be stored without compression in modern systems. This way, the *Locate* step is very simple, as it only requires an access to the suffix array. For this reason, this paper focuses in the *Count* step.

2.3 Sampled Occ Table

A key aspect in the BS algorithm is the implementation of the $Occ()$ function, which consumes most of the computing time of the algorithm. Other option is to precompute the $Occ()$ function and store the results in a look-up table [10, 12].

As a trade-off between memory space and computing time, only a fraction of the $Occ()$ values are stored in the table [10]. Specifically, the look-up table only stores the $Occ()$ values for pointers divisible by a given factor d. Hence, such sampled table, denoted by $sOcc$, is defined as follows:

$$sOcc[c, p] = Occ(c, 1 + (p-1)\times d) \tag{2}$$

where $p = 1, ..., n + 1$.

The values of the function $Occ()$ not stored in $sOcc$ are computed with the help of T^{bwt} as follows:

$$\begin{aligned} Occ(c, p) = Occ(c, q) + occur(c, T^{bwt}[(q+1)..p]) = \\ sOcc[c, \lfloor(p-1)/d + 1\rfloor] + occur(c, T^{bwt}[(q+1)..p]) \end{aligned} \tag{3}$$

being $q{=}1{+}d\times\lfloor(p-1)/d\rfloor \le p$ and *occur(c,s)* the number of occurrences in the string s of the symbol c.

2.4 Improving Data Locality

To improve data locality, columns of the table $sOcc$ and the string blocks of T^{bwt} required to compute the $Occ()$ function should be placed next in memory. This is accomplished in two steps (Figure 2 (left)). Firstly, rearranging T^{bwt} in an array of substrings of d consecutive symbols, called *buckets* [10]. This table, named bT^{bwt}, is defined as $bT^{bwt}[u, v] = T^{bwt}[d\times(u-1)+v]$, representing the symbol v of the bucket u. Secondly, combining both $sOcc$ and bT^{bwt} tables into a new one denoted by *SFM*. Row j in *SFM* refers to the column j in $sOcc$ and the row j in bT^{bwt}, that is, the bucket required to compute the values of $Occ()$ up to the next $sOcc$ column.

Using these sampled tables, an LFM operation is performed as follows:

$$\begin{aligned} sLF(Q[i], p, d) = C[Q[i]] + sOcc[Q[i], \lfloor(p-1)/d + 1\rfloor] + \\ occur(Q[i], bT^{bwt}[\lfloor(p-1)/d\rfloor + 1, [1..(m \bmod d)]]) \end{aligned} \tag{4}$$

Each LFM uses $sOcc[]$ which is d times smaller than a full-sized look-up table, but at the cost of performing more computation than using a full table.

2.5 Searching k Symbols in a Single Step

Arranging the backward search in steps (loop 3-6 in Figure 1) of k symbols at a time (instead of only one) allows to improve data locality even more and reduce computing cost, but at the cost of increasing slightly the memory footprint for $sOcc$ [12].

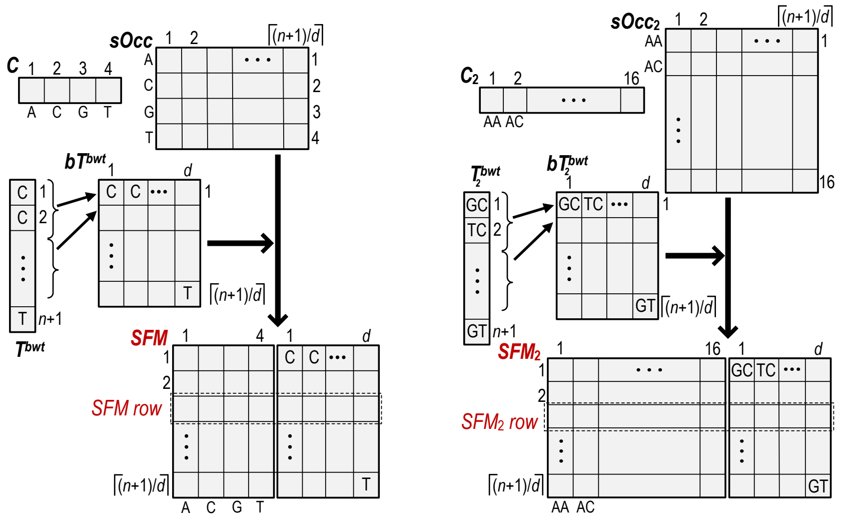

Fig. 2 Sampled (left) and 2-step sampled (right) data structures.

To search k symbols in a single step, the original alphabet (Σ) is replaced by a new one (Σ^k) composed of k-tuples whose elements come from Σ (permutations with repetition). This change in the alphabet implies modifications in the data structures. C is transformed into C_k, which is indexed by k-tuples in Σ^k. $sOcc$ becomes $sOcc_k$, whose first dimension is also indexed by k-tuples. T^{bwt} is transformed into T_k^{bwt}, which is composed of k $(n{+}1)$-symbol strings, namely the last k columns of the matrix M (see Section 2.1). These k strings, however, can be encoded as only one $(n{+}1)$-symbol string, where each symbol is now the concatenation of k symbols. Similarly to bT^{bwt}, T_k^{bwt}, encoded as a single string of k-tuples of symbols, can be blocked into buckets of size d, making up the new table bT_k^{bwt}. As with SFM, the tables $sOcc_k$ and bT_k^{bwt}, are combined into a new one denoted by SFM_k. Figure 2 (right) shows these new tables for $k{=}2$.

The k-step version of the calculation of an LFM (denoted by $sLF_k()$) is an extension of the single-step version, $sLF()$, but using the extended tables. A single $sLF_k()$ call resolves k symbols, being equivalent to k calls to $sLF()$. This arrangement improves data locality. Moreover, the computational cost of $sLF_k()$ is higher than that of $sLF()$, but resulting in a lower computing cost per LFM.

2.6 Overlapping Independent Pattern Searches

BS is a memory latency bound algorithm since many cycles are lost waiting for data due to a chain of dependent data accesses (lines 4-5 in Figure 1, where the pair (sp,ep) updated in a loop iteration is used in the next iteration). As a result, part of the available memory bandwidth is wasted. However, this mem-

```
Algorithm OBS: Query-Overlapped Backward Search
Input: $C[]$, $Occ[]$, $Q[]$ array of queries
Input: $n=|T[]|$, $N_q=|Q[]|$, $m=|Q[k][]|$, $j=1,...,N_q$
Ouput: $(sp[j],ep[j])$: Interval array of pointers of $Q[j]$ in $T$
1:  $sp[j] = C[Q[j][m]]$, $ep[j] = C[Q[k][m]+1]$, $j=1,...,N_q$
2:  for $i$ from $m$-1 to 1 step -1
3:    for $j$ from 1 to $N_q$ step 1
4:      $sp[j] = LF(Q[j][i],sp[j])$
5:      $ep[j] = LF(Q[j][i],ep[j])$
6:      prefetch($Occ[Q[j][i],sp[j]]$)
7:      prefecth($Occ[Q[j][i],ep[j]]$)
8:    end for
9:  end for
10: return $(sp[j]+1,ep[j])$, $j=1,...,N_q$
```

Fig. 3 Overlapped backward search algorithm.

ory latency can be hidden by issuing a given number of different independent searches, overlapping their memory accesses (batch or offline processing). The high number of independent read alignments which are usually involved in solving genome mapping problems makes this approach feasible.

The resulting overlapped BS (OBS) algorithm is shown in Figure 3. OBS executes a total of $2 \times N_q$ LFMs for each step (iteration of the outer loop $2-9$), corresponding to an array of N_q different queries that are searched concurrently. Note that after computing the two LFMs required for a given query, two prefetch operations are issued to retrieve from memory the two Occ values needed for computing the next two LFMs of the same query. The latencies of these memory reads are hidden by computing LFMs from other queries.

2.7 Split Bit-Vector Encoding

A loop iteration in the BS algorithm makes accesses to $sOcc_k$ using the pair (sp,ep) and updates the values of such pair to be used in the next iteration. Due to nature of the input queries and T^{bwt}, the search loop causes an access pattern to $sOcc_k$ not predictable and distributed along the whole table.

However, the two memory patterns, one due to accesses through sp and the other through ep, are partially correlated. After performing some loop iterations in BS, there are usually few matches in the reference text, and the sp and ep pointers may have similar values. In that case, the LFMs executed in a loop iteration likely access $sOcc_k$ entries that are stored in the same cache block. For the 1-step version, we have measured the ratio of these cache block correlations for different query lengths and text sizes assuming 64-byte cache blocks, obtaining a high degree of correlation (between 65% and 92% of the cases the accesses belongs to the same cache block).

We designed the *Split Bit-Vector Encoding* [3] to take advantage of the above memory pattern behavior and thus reduce the memory bandwidth consumption. The upper part of Figure 4 shows a row of the SFM_k data structure

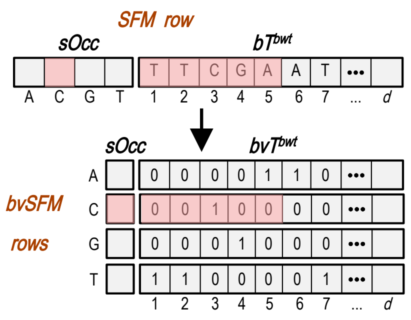

Fig. 4 SFM_k (top) and new $bvSFM_k$ (bottom) tables for k=1 (the accessed data when computing an LFM is shaded in red).

(for k=1). An example of all entries accessed in the computation of an LFM are marked in red. Note that only a single entry in $sOcc$ is accessed together with a separated substring of bT^{bwt}, that may be stored in different cache blocks. As an example, for k=2, the alphabet size is 16 ($|\Sigma|^2$) and thus a single $sOcc_2$ column occupies a complete 64-byte cache block (16 entries $\times$ 4 bytes/entry in $sOcc_2$).

The goal of the *Split Bit-Vector Encoding* of a SFM_k row, denoted by $bvSFM_k$, is that all data needed to perform an LFM is stored in a compact way, occupying a minimum number of cache blocks. The $bvSFM_k$ table is obtained from SFM_k through two transformations: (1) partition each row of SFM_k into $|\Sigma|^k$ rows, where each of them consists of a single $sOcc_k$ entry combined with the complete bucket (from bT_k^{bwt}). Specifically, the row t of SFM_k, that is, $SFM_k[t,*] \equiv sOcc_k[*,t] \mid bT_k^{bwt}[t]$ (a concatenation of the column t of $sOcc_k$ and the bucket t) is transformed into $|\Sigma|^k$ rows of $bvSFM_k$, of the form, $bvSFM_k[(t-1)|\Sigma|^k{+}i,*] \equiv sOcc_k[i,t] \mid bT_k^{bwt}[t]$, for i=1,...,$|\Sigma|^k$; (2) compression of each bucket using a bitmap where each symbol is represented by a single bit. This representation is as follows: given the row $bvSFM_k[(t-1)|\Sigma|^k{+}i,*]$ ($1 \le i \le |\Sigma|^k$), the corresponding bucket ($bT_k^{bwt}[t]$) is replaced by a bitmap of length d, where a symbol in the bucket is represented by a set bit (1) if it is equal to the one associated to the entry $sOcc_k[i,t]$, and by a unset bit (0) otherwise.

The lower part of Figure 4 shows an example of all rows of $bvSFM_k$ (k=1) encoded from the corresponding single SFM_k row. With $bvSFM_k$, all data required to calculate an LFM (in red) are placed together in memory and in a compact way, minimizing memory bandwidth consumption. The transformation also allows the *occur()* function to be simplified, as it simply has to count the number of set bits (1) in the accessed bucket.

According to [3], the new encoding provides a throughput between 60% and 135% better than best previous implementations, being able to reach around 95% of the peak random access bandwidth limit when executed on a Intel Xeon Phi KNL [13].

3 Processing-Near-Memory

Trying to overcome the memory-wall performance and energy problems, new computer architectures and techniques are appearing recently. One of the most promising acceleration paradigms is *Processing-Near-Memory* (PNM). This concept focuses on reducing time and power spent on memory accesses in typical processor-centric systems. For this purpose, they place data closer to the computing units, resulting in data-centric systems.

PNM approach has a relevant impact on memory-intensive applications, specially those ones that fulfill one or more of the following properties: a) low operational intensity, meaning that low computing power is required for each memory access; b) highly parallelizable, because it is usually more efficient to include more small cores than increasing core compute power; or c) not making good use of the deep cache hierarchies (typical in multi-core processors), as when the memory access patterns are irregular.

PNM usually relies on new technologies, specially modern chip manufacturing techniques, which make possible to reduce both the power consumption and area of the computing units. Other hardware techniques include *Through-Silicon Vias* (TSVs), vertical interconnections that enable fast communication inside 3D-stacked memories.

Previous works present different implementations taking advantage of this type of architectures. Some proposals change minimally the classic memory chips, in order to include some computing units inside them, while others focus their research on the Hybrid Memory Cube (HMC) [8] or the High Bandwidth Memory (HBM) [14] systems, two different 3D-stacked memory technologies. In the case of HMC, the specification even includes a lightweight logic layer able to perform simple memory operations. Finally, some implementations design entire new architectures.

In general, there are two different types of PNM architectures. On the one hand, some approaches use general-purpose processors inside the memory units. Those processors can be used for almost any application that require intensive use of memory. However, those PNM-processors can be power-hungry and the area required is relatively high. On the other hand, some architectures include specific-purpose processing units, much more efficient but useful only for some specific computations. Recently, some commercial DRAM-based PIM products (UPMEM) start to appear in the market [15].

4 Evaluation Framework

4.1 System Architectures

We define two representative system architectures with DDR-type SDRAM, one PNM architecture with HMC-type memory and a real commodity architecture (Intel i7-8700 with DDR4 SDRAM). These architectures serve as a framework to evaluate and compare the performance and energy efficiency

Table 1 Summary of hardware architectures.

	DDR Setup 1 (S1)		DDR Setup 2 (S2)		Intel i7-8700	PNM
Cores	64 @ 2.4 GHz		36 @ 3.6 GHz		6 @ 3.2-4.6GHz	64 @ 1.5 GHz
Type	OoO		OoO		OoO	in-Order
Hw Thr.	1		1		2	1
Arch.	x86		x86		x86	ARM-Like
Technol.	22 nm		22 nm		14 nm	28 nm
L1 Cache	32K/32K per core 3-cycle latency		32K/32K per core 3-cycle latency		32K/32K per core 4-cycle latency	8K/8K per core 3-cycle latency
L2 Cache	256K per core 10-cycle latency 8-way set associat.		256K per core 10-cycle latency 8-way set associat.		256K per core 12-cycle latency 4-way set associat.	-
L3 Cache	16M shared 30-cycle latency 16-way set associat. 6 banks H3 Hash		16M shared 30-cycle latency 16-way set associat. 6 banks H3 Hash		12M shared 42-cycle latency 16-way set associat. 6 banks	-
Block Sz.	64B		64B		64B	32/64B
Chann.	4		4		2	-
Memory Freq.	DDR3 1600	DDR4 2400	DDR3 1600	DDR4 2400	DDR4 2666	HMC 2500

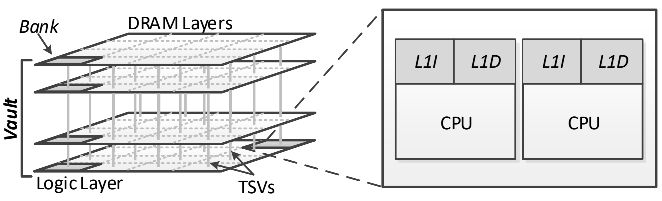

Fig. 5 Evaluated PNM architecture based on a 3D-stacked memory, that includes in-order cores placed on its logic layer.

of the `RANDOM` benchmark and the FM-index exact matching algorithm. Note that the defined general-purpose PNM architecture is not particularly suited for the FM-index exact matching algorithm. In addition, the conclusions of this work are also valid for other applications that present similar memory access patterns.

We present the details of the target architectures in Table 1. As it can be noticed, we compare DDR3 and DDR4 configurations using 64 and 36 out-of-order (OoO) cores at different frequency rates, with a 3D-stacked memory configuration using 64 power-efficient in-order cores at lower frequency.

Applications featuring random memory accesses, as the FM-index exact matching algorithm, usually do not take advantage from typical cache configurations, being even penalized when using deep cache hierarchies. As data is very rarely reused, in many cases a significant part of the cache block is wasted. The key idea behind our proposal is the use of simple cores with only one level of cache, that access memory with low latency and high bandwidth using a PNM approach.

As a result, we define an optimized architecture for random memory accesses using simple ARM-A35 cores, which are 64-bit in-order ARMv8-A processors, very efficient in terms of both power consumption and area. According to the ARM technical reference manual, the smaller version of this architecture is equipped with 8K/8K L1 caches per core and requires an area of less than 0.4 mm^2 using a 28-nm technology. This setup consumes approximately 90 mW at 1-GHz frequency [16]. We provide additional details in terms of area and power consumption in Section 5.5. Using this approach, we are able to reduce area and power consumption, as only one level of private caches (L1) are used for the PNM architecture. We analyze the use of both 64 bytes and 32 bytes for cache block sizes, trying to reduce even more the amount of never-used data brought from main memory to cache.

The described PNM architecture is shown in Figure 5, where the simple general-purpose cores are placed in the logic layer of a 3D-stacked memory cube, taking advantage of the high bandwidth and lower latency compared to a typical processor configuration.

4.2 `RANDOM` Benchmark

The FM-index exact matching (see Figure 1) is a memory-bound algorithm with a random pattern of memory accesses. Although this paper is focused on that algorithm, however it is known that this type of memory pattern is common in many other sequence alignment applications. For this reason, in a first phase, we developed a synthetic benchmark, called `RANDOM`, with the aim of mimicking the memory access pattern of the exact matching algorithm, but with a user configurable amount of computing operations. This benchmark allows us to explore the memory behaviour for these applications running over a range of operational intensity values.

The `RANDOM` benchmark pseudocode is shown in Figure 6. It uses C randomly generated linked lists with no access locality. An array of head pointers, p, is updated a number of times following the linked lists. After each pointer update, the next list element is prefetched. Taking this into account, if C is high enough, the latency of memory accesses will be hidden. A timing diagram of the benchmark is presented in Figure 7.

The `RANDOM` benchmark can be configured in terms of: (1) cache block size, (2) operational intensity, (3) data structure size, (4) number of threads, (5) number of parallel linked lists and (6) datatype for the arithmetic operations. This range of parameters allows to perform a comprehensive evaluation of the performance/energy behaviour of algorithms with this kind of memory access patterns.

In the evaluation we also considered the `STREAM` benchmark [17], which is used as a baseline to compare the `RANDOM` results to those derived from algorithms with uniform (stream) memory access patterns.

```
Benchmark: RANDOM
Input: $N$: Number of random memory accesses
       $C$: Number of linked lists (dependency-chains)
       $OP$: Operations per memory block
       $p$: Array of head pointers to the linked lists
begin
for $i$ from $N$-1 to 1 step $C$
  for $k$ from 0 to $C$ step 1
    $p[k] = p[k] \rightarrow$next
    for j from 0 to OP step 1
      tmp = tmp*p[k]
    end for
    prefetch($p[k]$)
  end for
end for
end
```

Fig. 6 RANDOM benchmark.

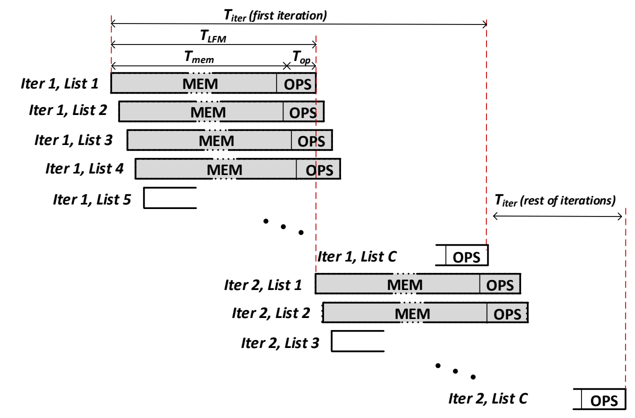

Fig. 7 RANDOM benchmark timing model.

5 Evaluation

5.1 Methodology

5.1.1 ZSim

ZSim is an architectural simulator [18] able to simulate thousand-core systems much faster (between 100-1000×) than other cycle-based simulators. This is possible thanks to novel acceleration techniques used by ZSim. First, it provides a fast sequential simulation using DBT. ZSim uses instrumentation with Intel Pin [19] to perform dynamic binary translation (DBT), eliminating the need for functional modeling of x86 and placing most of the work on the instrumentation phase. Second, it uses parallel simulation for modeling multi-core chips, using an event-driven parallelization technique to improve accuracy.

This simulator is also focused in flexibility and usability, being able to support two main types of core models. A simple core, which is a small core with IPC=1 for all but load/store instructions, and an out-of-order (OoO) core, a modern core with much more functionality present in real-life processors, as branch predictor, complex instruction fetching and detailed arithmetic units. Regarding to memory, ZSim supports a simple memory (fixed latency), a MD1 memory and a DDR (by default, DDR3) memory models. The chosen system configuration (both core and memory models) influences the simulation speed.

In our PNM architecture, we simulate power-efficient, small ARM-like cores using ZSim. However, ZSim does not support the ARM ISA and architecture. Instead of that, we use the simple in-order core model (IPC=1) available in ZSim, configured at low frequency, with a performance comparable to typical ARM-A35 processors.

5.1.2 Ramulator

Ramulator [20] is a fast and cycle-accurate simulation tool for current and future DRAM systems. It is able to accurately provide models for a variety of different memory standards, as for example, DDR3, DDR4, LPDDR, GDDR5, HBM, SALP, HMC and PCM. It can be used in two different ways: *integrated* within an architecture simulator, like gem5 or ZSim, or *standalone*, being fed with a memory trace or an instruction trace. When used as integrated, it provides a simple memory controller which exposes an external API for sending and receiving memory requests. In contrast, when used as standalone, it models a simple CPU that issues the memory requests from the input trace.

We use Ramulator for modeling the memory system, since it is more accurate and extensible than ZSim integrated memory models. Our in-house ZSim and Ramulator integration have been performed by imitating the DRAMSim2 [21] integration with ZSim, where ZSim issues memory requests to Ramulator, waiting for the response in a similar way to a real system.

5.1.3 Simulation Setup and Workloads

The evaluation is conducted on the architectures described in Section 4.1. We use ZSim git public version [22] together with Ramulator-PIM public version, which allows us to simulate accurate memory systems. We validate the obtained results running the DDR3 configurations using both Ramulator and ZSim memory models.

We modify public ZSim version to support direct communication and integration with Ramulator, creating an in-house PNM simulation framework. This integration is implemented imitating the way that ZSim communicates to DRAMSim2 memory simulator (i.e., each memory request is issued and dispatched on demand), avoiding the necessity of using memory traces and allowing running very large simulations.

We use McPAT [23] to obtain power estimations of the evaluated architectures, which is an integrated power, area, and timing modeling framework for multicore and manycore architectures.

We execute our simulations based on ZSim, Ramulator and McPAT in a system with two Intel Xeon Gold 6154 (18 cores, 36 hardware threads) at 3 GHz and 384 GB DDR4 memory running Ubuntu 18.04.1.

Finally, we validate our conclusions running the experiments in a real system based on an Intel i7-8700 Coffee Lake processor with 6 cores (12 hardware threads) at 3.2 GHz each, able to reach 4.6 GHz using turbo-boost. It includes 64 GB of DDR4 memory and runs also Ubuntu 18.04.1.

We compile the benchmarks using GCC with common flags and `-O3` optimization level and configure them according to the following parameters.

– `STREAM` benchmark: it defines an array size of 10,000,000 elements. Reported values are the maximum of all `STREAM` results.
– `RANDOM` benchmark: it uses a 1 GB data structure, with 3,000,000 random accesses.
– FM-index exact matching algorithm: the different versions are based on a 1 GB genome and they search for 500,000 short sequences (approximately, 200 bases each).

All benchmarks and applications have been run on all the available threads in each system. Experiments have been run at least three times and we have reported the average value of them. We found very low variance between the obtained values, which means that taking the average value leads to consistent results.

5.2 Benchmark Analysis

We first run experiments using `STREAM` and `RANDOM` benchmarks. Figure 8 shows the maximum memory bandwidth attained from the execution of both benchmarks. As it can be noticed, a significant bandwidth improvement is obtained when using PNM architectures, achieving between 2.7× and 3.4× better performance when compared to the i7-8700 architecture. On the other hand, these PNM systems obtain between 1.4× and 1.9× better performance when compared to the 36-core and 64-core systems.

Figures 9 and 10 show the `RANDOM` benchmark results for different operational intensities. As the FM-index exact matching algorithm uses integer operations, we define the operational intensity as the number of LFM operations (see section 2.2) per accessed cache block from main memory. Given the techniques discussed in section 2 to exploit data locality, the best implementations for the BS algorithm require to access one or two consecutive cache blocks per query step (that is, $2 \times k$ LFM operations). That is the reason to configure the two options in the `RANDOM` benchmark.

As expected, the PNM architecture takes advantage of lower operational intensities where memory bandwidth becomes relevant. However, this architecture provides lower performance results than DDR-based systems for higher operational intensities. This fact is explained because of the lower frequency and lower computing power of the in-order cores in the PNM architecture. There is a trade-off between bandwidth and computing power when comparing Figures 9 and 10, showing the memory bandwidth and the amount of integer operations performed per second, respectively. Additionally, Figure 9 shows that the PNM architecture reaches half of the memory bandwidth when using 32-byte cache blocks instead of the 64-byte ones.

We make three key observations. First, the operational intensity determines if an application is a good fit for the PNM setup or not. Second, for those PNM-friendly applications, the PNM architecture takes advantage of the higher memory bandwidth of the architecture, which leads to better performance. Third, the performance of random-access applications (e.g., `RANDOM`) is similar to the performance of sequential-access applications (e.g., `STREAM`) when using the PNM architecture. In contrast, we find a larger difference when using DDR-based solutions. This fact is explained because of the higher memory latency and the deeper cache hierarchies that are present in the DDR-based systems.

5.3 Roofline Analysis

We build the Roofline models for the experiments conducted in the architectures of Table 1 and present the results in Figures 11, 12, 13 and 14. For this task, we obtain the memory bandwidth usage directly from `STREAM` and `RANDOM` benchmarks.

The horizontal lines represent the computing limit for each architecture, which are measured with our `RANDOM` benchmark. We obtain such limit configuring the benchmark to perform a huge amount of integer operations per each loaded block from memory (65,536). On the other hand, diagonal lines

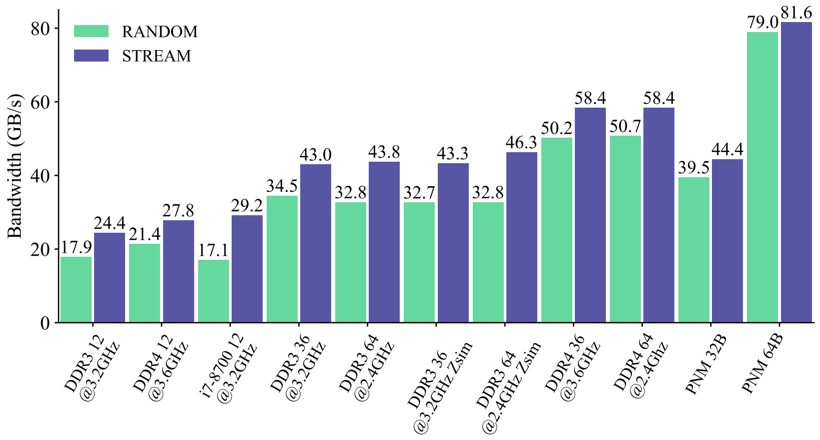

Fig. 8 Memory bandwidth for `STREAM` and `RANDOM` benchmarks in different architectures.

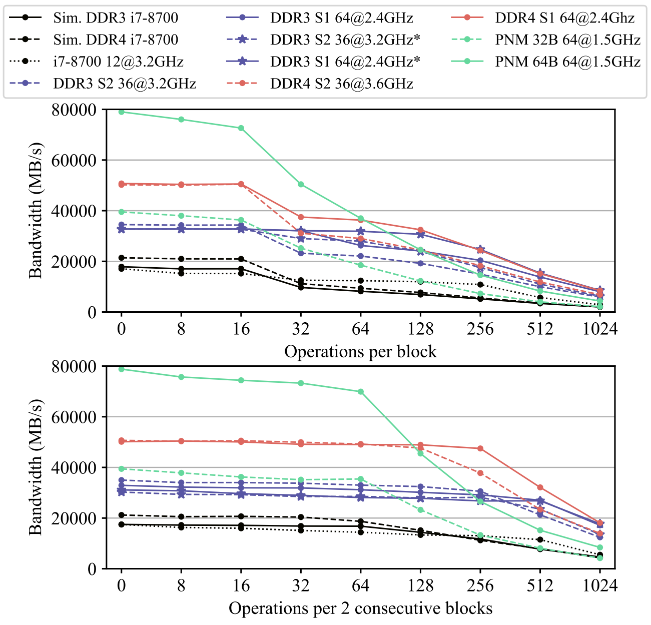

Fig. 9 Memory bandwidth for the `RANDOM` benchmark in MBytes per second using different operational intensities (operations per each accessed cache block or two cache blocks).

represent the memory bandwidth limit, both for uniform accesses (measured with `STREAM`) and for random accesses (measured with `RANDOM`) being configured to perform very few operations per byte. As shown in previous sections, obtained random access bandwidths are always lower than uniform ones, being the difference less significant on the PNM architecture.

Those figures show how the `RANDOM` benchmark fits well to the Roofline model, specially for the Intel i7-8700 architecture. DDR architectures achieve better performance when the operational intensity is high (up to 2× instructions per byte). On the other hand, the PNM architecture provides significant better performance and higher bandwidth usage when the operational intensity is low. This fact can be noticed when checking the bandwidth limits lines, which present a more vertical shape than the DDR ones. Another important observation is that the bandwidth limits for the PNM system are very close when comparing random versus uniform memory accesses (both lines are almost overlapped in Figure 13, a fact that does not occur for the other systems). In other words, when executing applications with random access patterns in the PNM system, the performance is very similar to an application that follows a uniform pattern. This is explained because 1) the smaller cache hierarchy present in the PNM system with respect to the conventional ones, where a cache miss incurs in less penalty, and 2) the reduced distance between compute units and memory. As consequence, the PNM system is a better fit to applications that follow this unpredictable access pattern, as FM-index.

Finally, we observe that the Intel i7-8700 architecture results present certain deviation between the real and simulated ones (Figure 14). This fact is explained because of the higher processor frequency achieved by the real system using turbo boost, which can not be simulated on the current version of ZSim. However, our conclusions are valid as we observe the same behaviour.

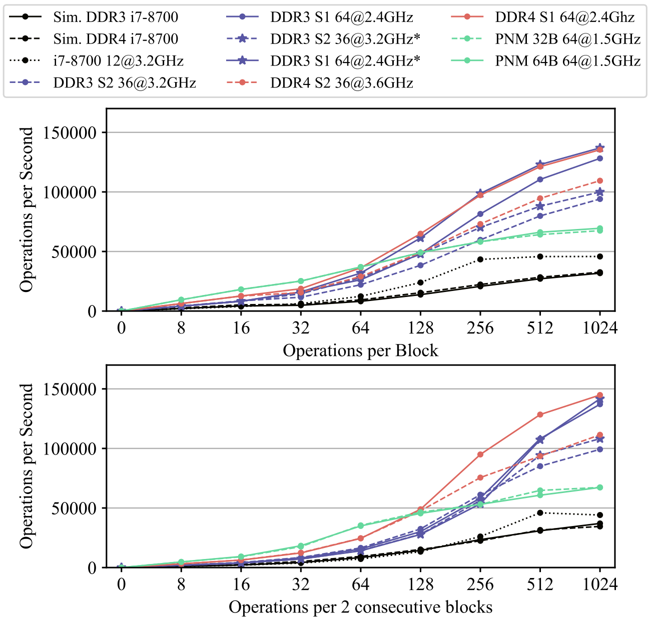

Fig. 10 `RANDOM` benchmark operations per second using different operational intensities.

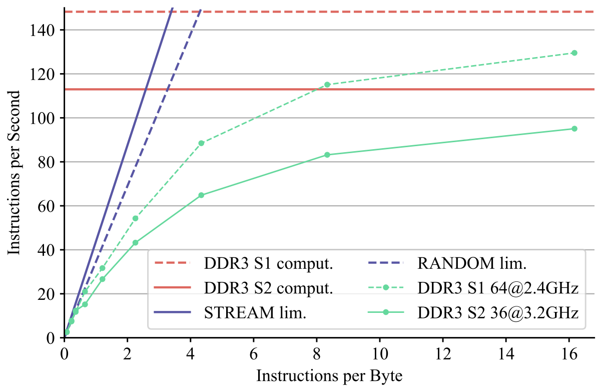

Fig. 11 Roofline model for the DDR3-based systems.

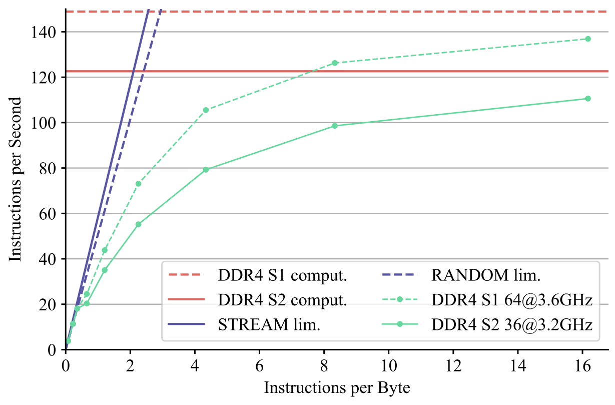

Fig. 12 Roofline model for the DDR4-based systems.

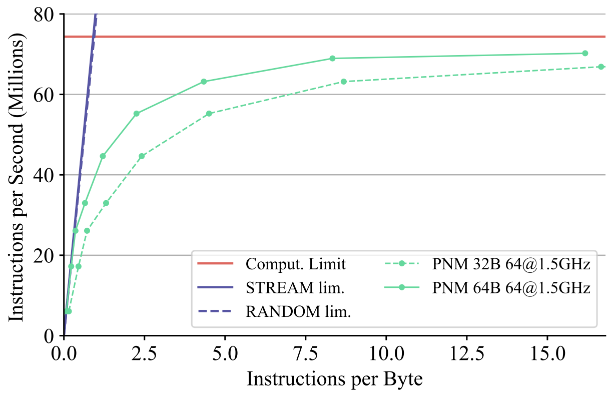

Fig. 13 Roofline model for the PNM system.

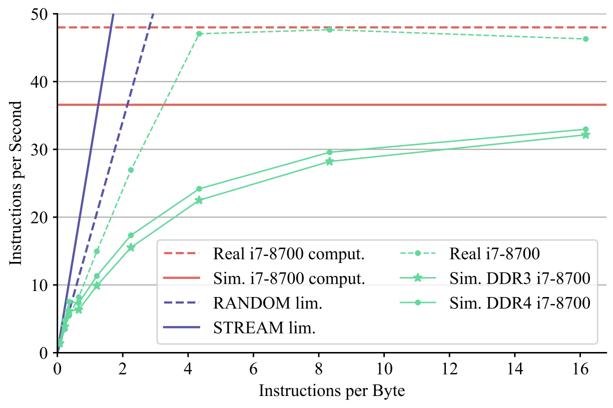

Fig. 14 Roofline model for the Intel Core i7-8700 system.

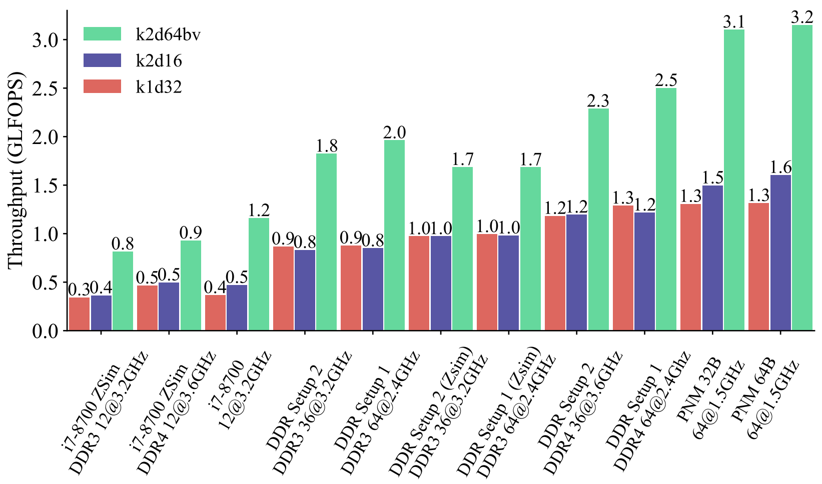

Fig. 15 Performance for three FM-index exact matching versions.

To sum up, the takeaway of Figures 11, 12, 13 and 14 is two fold: 1) random access applications do not take fully advantage of deep cache hierarchies because of the almost absence of data locality, and 2), low operational intensity is bandwidth-hungry and bottlenecked in conventional DDR architectures. FM-index is an example of a random access application with low operational intensity, being a good candidate for acceleration in a PNM system (where we get rid of deep cache hierarchies and provide higher memory bandwidth).

5.4 FM-index Exact Matching Analysis

Once tested the `RANDOM` benchmark, we evaluated the FM-index exact matching algorithm as a real application that presents a similar memory access pattern. Figure 15 shows the obtained performance for three different FM-index implementations: two k-step sampled FM-index versions, with $k = 1$, $d = 32$ (`k1d32`) and $k = 2$, $d = 16$ (`k2d16`), and our split bit-vector FM-index version, with $k = 2$ and $d = 64$ (`k2d64bv`). Performance is measured in giga LFM operations per second (GLFOPS) (see section 2.2).

As it can be noticed, PNM architectures achieve between 2.7×-3.7× better performance than the 12-core ones. We can also observe that the PNM approach provides between 1.26× and 1.87× better performance than 36-core and 64-core architectures with DDR3 and DDR4 memory technologies. On the other hand, DDR4-based architectures provide approximately 25% better performance than DDR3-based ones.

When comparing the FM-index implementations, the split bit-vector 2-step version is able to achieve almost twice the amount of LFM operations per second than the other FM-index versions. Another observation is that the PNM architecture that uses 32-byte cache blocks gets roughly the same performance as the one using 64-byte cache blocks.

5.5 Area and Power Consumption Analysis

We used McPAT to obtain the power consumption and area for the cores used in the evaluated architectures. However, in order to make a more accurate estimation, we also considered the power consumption and area specification from real equivalent processors and we compared them against the estimations.

An in-depth area analysis of the high performance cores were not performed since they are not restricted by the logic layer available in the 3D-stacked memory.

5.5.1 PNM Architecture with Low Power Cores

We first evaluated our PNM architecture with simple in-order cores. According to McPAT, when using a 28-nm technology, each small PNM core that runs at 1.5 GHz requires an area of 2 mm^2 and has a power consumption of 0.5 W. In contrast, when using a 22-nm technology, the required area by one of those cores is 0.61 mm^2 area and the power consumption decreases to 0.2 W.

We designed the cores used in this PNM architecture imitating the smallest A35 cores from ARM, but using a higher frequency. Considering this fact, we can estimate the area and power of the PNM cores based on the ones corresponding to the real ARM cores. We conservatively assumed an area of 0.4 mm^2 and 0.18 W of power consumption per each small core when running at 1.5 GHz (double than the 1 GHz reported power consumption, considering that power consumption does not scale linearly on frequency).

5.5.2 Commodity Architecture with High Performance Cores

For comparison purposes, we consider a dual-socket Intel Xeon Gold 6154 including 36 cores at 3.0-3.7 GHz and an Intel Xeon Phi 7210 [13] with 64 cores at 1.3-1.5 GHz. Intel reports a TDP of 200 W for the Xeon Gold and 215 W for the Xeon Phi.

Our first evaluated architecture (see Table 1) includes 64 cores at 2.4 GHz, which can be considered as a Xeon Phi with a higher frequency. Based on the TDP reported by Intel for those two configurations, we analytically estimate that a modified Xeon Phi at 2.4 GHz should have a TDP between 420 W and 450 W. In contrast, our second architecture is very similar to a dual Xeon Gold setup, which TDP is almost 400 W. These estimations are comparable to those provided by McPAT, which reports a TDP of 450.68 W for the architecture using 64 cores at 2.4 GHz each (Setup 1), and 375.36 W for the setup with 36 cores at 3.6 GHz each (Setup 2). Finally, for the Intel i7-8700, we consider the TDP value of 65 W reported by Intel.

Figure 16 shows the estimated and reported power consumption for both real systems and simulated architectures. The grey top section of the bars shows the power consumption of the memory system.

The 3D-stacked memory power consumption is estimated using Micron HMC power consumption calculator tool. DDR power consumption is estimated obtaining a value of 3 W per memory module, using the Micron power calculators as well [24]. This value is considered reasonable for DDR4 DIMM modules [25].

5.6 Power Efficiency of Random Accesses Analysis

Providing McPAT with the power consumption data presented in Section 5.5 and the ZSim output stats, we obtain the efficiency for each architecture measured in LFM operations performed per consumed Joule. Figure 17 shows the efficiency for both our `RANDOM` benchmark and the `k2d64bv` FM-index version.

We can observe a great improvement for the PNM architectures when compared with conventional DDR memory systems, being able to perform 8× more LFM operations and random accesses per joule than i7-8700. Compared to the high-performance systems, with a higher number of cores, the improvement are even greater, using around 21× less energy per LFM operation.

6 Related work

In this section we present the state-of-the art in 1) sequence alignment acceleration techniques and 2) Processing-Near-Memory as an approach to accelerate memory-bound applications.

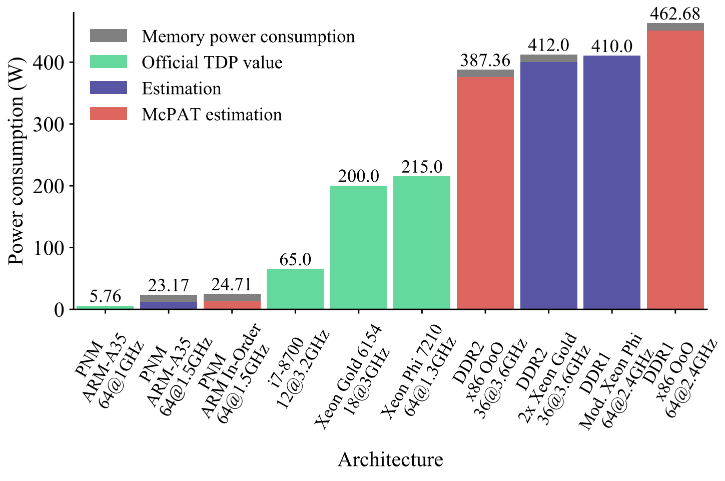

Fig. 16 Instantaneous power consumption for each architecture.

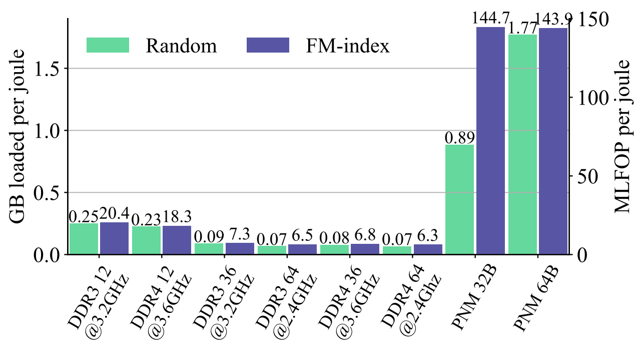

Fig. 17 Memory bandwidth and mega LFM operations per Joule (MLFOP) for the `RANDOM` benchmark and the `k2d64bv` FM-index version.

6.1 Sequence Alignment Optimizations

Biomedical applications, like genome or protein sequencing, are having a great impact in many fields of bioinformatics [26]. To support progress in these applications, fast tools and algorithms have been developed recently for sequence alignment, an usual first step in genome or protein sequencing. Many of the most popular alignment methods are based on some type of index structures, like suffix trees, being FM-index widely used. Examples of sequence aligners based on FM-index are HISAT2 [27] and Bowtie/Bowtie2 [28].

Regarding the acceleration of the above tools on high-performance architectures [26], we can find different proposals in the literature for specific architectures or hardware accelerators, including multi-core processors [29], GPUs (Arioc [30], Clusters (CUSHAW3 [31]), Clouds (BigBWA [32]) and FPGAs (FHAST [33]). Another example is the GenAx accelerator [34], which provides around 30× speedup compared to a 14-core Xeon server. We also find dataflow implementations based on FPGA for Smith-Waterman Matrix-fill and Traceback stages (commonly used in BWA-MEM and Bowtie2) in [35], but the FPGA is only in charge of the Smith-Waterman part (i.e., the rest stages have to run in the CPU). Attached to an IBM POWER8 CPU, the authors also accelerate the Smith-Waterman algorithm using and FPGA as coprocessor, achieving 1.6× speedup compared to the CPU-only execution. The DRAGEN platform [36] is another FPGA-based sequence aligner which operates with a dual Intel Xeon processor in a hardware-software collaboration manner. DRAGEN is a proprietary architecture from Illumina that achieves between 16× and 18× speedup compared to a software-based implementation.

6.2 Processing-Near-Memory

Over last years, new architecture paradigms based on PNM are gaining importance [37,6], in order to solve the problems derived from the memory wall and data-intensive applications.

Consequently, a significant amount of works around these topics have appeared during the last years. Most of them, like ours, based on the Micron HMC architecture, expanding or completely reworking the logic layer. For example, some works based on HMC are: [38], oriented to optimize parallel graph processing; [39], analyzing the performance of Google workloads; [40] and [41], optimizing graph processing applications; [42], presenting a NDP accelerator for basic data analytic operators; [43], including general near-data processors in the logic layer and analyzing their performance for common applications like MapReduce, PageRank and neural networks; [44], focused on neural network acceleration; [45], with some common points with our work, improving bioinformatics applications performance through PNM; and [46], NATSA, a PNM accelerator for time series analysis.

Furthermore, some works are oriented to use different architectures, like [47], working with NDP on GPUs; [48], mixing CPUs and GPUs close to the data; [49] and [50], implementing these techniques with commodity DRAM modules.

In summary, there are some previous related works in both algorithm optimizations and hardware development fields. However, to the best of our knowledge, there are no prior works that propose a method to evaluate unpredictable access pattern applications and take FM-Index as case study.

7 Conclusions

This paper presents an analysis and evaluation of the random memory access pattern of a typical algorithm in the genome sequence alignment arena (exact matching based on FM-index) on both commodity and emerging PNM system architectures. In addition, we developed a `RANDOM` benchmark that mimics the memory access pattern of the above algorithm, but with a configurable amount of computing operations. With this benchmark we explore the behaviour of the class of applications showing this memory pattern over a range of operational intensity values.

Our experiments show that the PNM architecture, using a 3D-stacked memory with in-order power-efficient cores, achieve better performance with a significant lower energy consumption than conventional architectures with conventional memory technology (DDR). More specifically, the PNM architecture achieves more than 3× the performance when running the `RANDOM` benchmark, and between 1.26× and 3.7× when executing the FM-index exact matching algorithm, compared with typical DDR-based systems. Regarding energy efficiency, the PNM architecture shows a reduction in energy consumption between 21× and 40× compared to systems with commodity DDR-based memories and systems with deep and large cache hierarchies.

References

1. Chen, C., Zhang, C.Y.: Data-intensive applications, challenges, techniques and technologies: a survey on big data. Information Sciences **275** (2014) 314–347
2. Kestor, G., Gioiosa, R., Kerbyson, D.J., Hoisie, A.: Quantifying the energy cost of data movement in scientific applications. In: 2013 IEEE International Symposium on Workload Characterization (IISWC). (2013) 56–65
3. Herruzo, J., Gonzalez-Navarro, S., Ibañez, P., Viñals, V., Alastruey, J., Plata, O.: Accelerating sequence alignments based on FM-index using the Intel KNL processor. IEEE/ACM Transactions on Computational Biology and Bioinformatics **17**(4) (July 2020) 1093–1104
4. NovaSeq System Specifications The next era of sequencing starts now. https://www.illumina.com/systems/sequencing-platforms/novaseq/specifications.html
5. Chen, M., Mao, S., Liu, Y.: Big data: a survey. Mobile Networks and Applications **19**(2) (Apr 2014) 171–209
6. Mutlu, O., Ghose, S., Gomez-Luna, J., Ausavarungnirun, R.: A modern primer on processing in memory. arXiv preprint arXiv:2012.03112 (2020)
7. Ghose, S., Boroumand, A., Kim, J., Gomez-Luna, J., Mutlu, O.: Processing-in-memory: A workload-driven perspective. IBM Journal of Research and Development **63**(6) (November 2019) 3:1—-3:19
8. Micron Technology, Inc. Hybrid Memory Cube (HMC). https://www.micron.com/products/hybrid-memory-cube
9. Li, H., Homer, N.: A survey of sequence alignment algorithms for next-generation sequencing. Briefings in Bioinformatics **11**(5) (May 2010) 473–483
10. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: 41st Annual Symposium on Foundations of Computer Science. (2000) 390–398
11. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
12. Chacon, A., Moure, J.C., Espinosa, A., Hernandez, P.: n-step FM-index for faster pattern matching. Procedia Computer Science **18** (2013) 70–79
13. Intel Xeon Phi Processor 7210 (16GB, 1.30GHz, 64 core) Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/94033/intel-xeon-phi-processor-7210-16gb-1-30-ghz-64-core.html
14. Lee, D.U., Kim, K.W., Kim, K.W., Kim, H., Kim, J.Y., Park, Y.J., Kim, J.H., Kim, D.S., Park, H.B., Shin, J.W., Cho, J.H., Kwon, K.H., Kim, M.J., Lee, J., Park, K.W., Chung, B., Hong, S.: 25.2 A 1.2v 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In: IEEE International Solid-State Circuits Conference (ISSCC'14). (2014) 432–433
15. Devaux, F.: The true processing in memory accelerator. In: IEEE Hot Chips 31 Symposium (HOTCHIPS 2019). (August 2019)
16. Each milliwatt matters – ultra high efficiency application processors. http://www.armtechforum.com.cn/attached/article/ARM_Each_Milliwatt_Matters20151210111238.pdf
17. McCalpin, J.D.: Stream: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia (1991-2007) A continually updated technical report. http://www.cs.virginia.edu/stream/.
18. Sanchez, D., Kozyrakis, C.: ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In: 40th Annual International Symposium on Computer Architecture (ISCA'13). (June 2013) 475–486
19. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05). (June 2005) 190–200

20. Kim, Y., Yang, W., Mutlu, O.: Ramulator: a fast and extensible DRAM simulator. IEEE Computer Architecture Letters **15**(1) (mar 2015) 45–49
21. Rosenfeld, P., Cooper-Balis, E., Jacob, B.: DRAMSim2: a cycle accurate memory system simulator. IEEE Computer Architecture Letters **10**(1) (mar 2011) 16–19
22. s5z/zsim: a fast and scalable x86-64 multicore simulator. `https://github.com/s5z/zsim`
23. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P.: McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09). (December 2009) 469–480
24. Micron Power Calculators. `www.micron.com/support/tools-and-utilities/power-calc`
25. Crucial (Micron Technology, Inc.) How much power does memory use? `https://www.crucial.com/support/articles-faq-memory/how-much-power-does-memory-use`
26. Schmidt, B., Hildebrandt, A.: Next-Generation Sequencing: big data meets high performance computing. Drug Discovery Today **22**(4) (April 2017) 712–717
27. Kim, D., Paggi, J.M., Park, C., Bennett, C., Salzberg, S.L.: Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype. Nature Biotechnology **37** (August 2019) 907–915
28. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with bowtie2. Nature Methods **9** (March 2012) 357–359
29. Langmead, B., Wilks, C., Antonescu, V., Rone, C.: Scaling read aligners to hundreds of threads on general-purpose processors. Bioinformatics **35**(3) (February 2019) 421–432
30. Wilton, R., Budavari, T., Langmead, B., Wheelan, S.J., Salzberg, S.L., Szalay, A.S.: Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space. PeerJ **3:e808** (2015)
31. Gonzalez-Dominguez, J., Liu, Y., Schmidt, B.: Parallel and scalable short-read alignment on multi-core clusters using UPC++. PLoS One **11**(1) (2016)
32. Abuin, J.M., Pichel, J.C., Pena, T.F., Amigo, J.: BigBWA: Approaching the Burrows-Wheeler aligner to big data technologies. Bioinformatics **31**(24) (2015) 4003–4005
33. Fernandez, E.B., Villarreal, J., Lonardi, S.: FHAST: FPGA-based acceleration of Bowtie in hardware. IEEE/ACM Transactions on Computational Biology and Bioinformatics **12**(5) (2015) 973–981
34. Fujiki, D., Subramaniyan, A., Zhang, T., Zeng, Y., Das, R., Blaauw, D., Narayanasamy, S.: Genax: a genome sequencing accelerator. In: ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18). (2018) 69–82
35. Koliogeorgi, K., Voss, N., Fytraki, S., Xydis, S., Gaydadjiev, G., Soudris, D.: Dataflow acceleration of Smith-Waterman with Traceback for high throughput next generation sequencing. In: 29th International Conference on Field Programmable Logic and Applications (FPL'19). (2019) 74–80
36. Miller, N.A., Farrow, E.G., Gibson, M., Willig, L.K., Twist, G., Yoo, B., Marrs, T., Corder, S., Krivohlavek, L., Walter, A., et al.: A 26-hour system of highly sensitive whole genome sequencing for emergency management of genetic diseases. Genome Medicine **7**(1) (2015) 1–16
37. Ghose, S., Hsieh, K., Boroumand, A., Ausavarungnirun, R., Mutlu, O.: Enabling the adoption of processing-in-memory: challenges, mechanisms, future research directions. arXiv preprint arXiv:1802.00320 (2018)
38. Ahn, J., Hong, S., Yoo, S., Mutlu, O., Choi, K.: A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In: Int'l. Symp. on Computer Architecture (ISCA'15). (2015) 105–117
39. Boroumand, A., Ghose, S., Kim, Y., Ausavarungnirun, R., Shiu, E., Thakur, R., Kim, D., Kuusela, A., Knies, A., Ranganathan, P., Mutlu, O.: Google Workloads for consumer devices: Mitigating data movement bottlenecks. In: ACM 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18). (March 2018) 316–331
40. Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., Kim, H.: GraphPIM: enabling instruction-level PIM offloading in graph computing frameworks. In: 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA'17). (February 2017) 457–468

41. Zhang, M., Zhuo, Y., Wang, C., Gao, M., Wu, Y., Chen, K., Kozyrakis, C., Qian, X.: GraphP: reducing communication for PIM-based graph processing with efficient data partition. In: 24th IEEE International Symposium on High Performance Computer Architecture (HPCA'18). (February 2018) 544–557
42. Drumond Lages De Oliveira, M.P., Daglis, A., Mirzadeh, N., Ustiugov, D., Picorel Obando, J., Falsafi, B., Grot, B., Pnevmatikatos, D.: The Mondrian Data Engine. 44th International Symposium on Computer Architecture (ISCA'17) (June 2017)
43. Gao, M., Ayers, G., Kozyrakis, C.: Practical near-data processing for in-memory analytics frameworks. In: 24th International Conference on Parallel Architectures and Compilation Techniques (PACT'15). (October 2015) 113–124
44. Gao, M., Pu, J., Yang, X., Horowitz, M., Kozyrakis, C.: TETRIS: scalable and efficient neural network acceleration with 3D memory. In: 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17). (April 2017) 751–764
45. Kim, J.S., Cali, D.S., Xin, H., Lee, D., Ghose, S., Alser, M., Hassan, H., Ergin, O., Alkan, C., Mutlu, O.: GRIM-Filter: fast seed location filtering in DNA read mapping using processing-in-memory technologies. BMC Genomics **19**(2) (2018) 23–40
46. Fernandez, I., Quislant, R., Gutierrez, E., Plata, O., Giannoula, C., Alser, M., Gomez-Luna, J., Mutlu, O.: NATSA: a near-data processing accelerator for time series analysis. In: IEEE 38th International Conference on Computer Design (ICCD'20). (2020) 120–129
47. Hsieh, K., Ebrahimi, E., Kim, G., Chatterjee, N., O'Connor, M., Vijaykumar, N., Mutlu, O., Keckler, S.W.: Transparent Offloading and Mapping (TOM): enabling programmer-transparent near-data processing in GPU systems. In: ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16). (June 2016) 204–216
48. Zhang, D., Jayasena, N., Lyashevsky, A., Greathouse, J.L., Xu, L., Ignatowski, M.: TOP-PIM: throughput-oriented programmable processing in memory. In: 23rd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'14). (June 2014) 85–98
49. Farahani, A.F., Ahn, J.H., Morrow, K., Kim, N.S.: NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. 21st IEEE International Symposium on High Performance Computer Architecture (HPCA'15) (February 2015) 283–295
50. Asghari-Moghaddam, H., Son, Y.H., Ahn, J.H., Kim, N.S.: Chameleon: versatile and practical near-DRAM acceleration architecture for large memory systems. In: 49th Annual ACM/IEE International Symposium on Microarchitecture (MICRO'16). (October 2016)