Porting and Optimizing BWA-MEM2 Using the Fujitsu A64FX Processor

Rubén Langarita, Adrià Armejach, Pablo Ibáñez, Jesús Alastruey-Benedé, Miquel Moretó

Abstract—Sequence alignment pipelines for human genomes are an emerging workload that will dominate in the precision medicine field. BWA-MEM2 is a tool widely used in the scientific community to perform read mapping studies. In this paper, we port BWA-MEM2 to the AArch64 architecture using the ARMv8-A specification, and we compare the resulting version against an Intel Skylake system both in performance and in energy-to-solution. The porting effort entails numerous code modifications, since BWA-MEM2 implements certain kernels using x86_64 specific intrinsics, e.g., AVX-512. To adapt this code we use the recently introduced Arm's Scalable Vector Extensions (SVE). More specifically, we use Fujitsu's A64FX processor, the first to implement SVE. The A64FX powers the Fugaku Supercomputer that led the Top500 ranking from June 2020 to November 2021. After porting BWA-MEM2 we define and implement a number of optimizations to improve performance in the A64FX target architecture. We show that while the A64FX performance is lower than that of the Skylake system, A64FX delivers 11.6% better energy-to-solution on average. All the code used for this article is available at https://gitlab.bsc.es/rlangari/bwa-a64fx.

Index Terms—sequence alignment, read mapping, genomics, A64FX, SVE, BWA-MEM2, Intel Skylake.

1 Introduction

Precision medicine aims to improve healthcare by exploiting genomic information [1]. In recent years, the sharp reduction in genome sequencing costs has driven a dramatic increase in the amount of data generated for processing, which has posed a significant computational and storage challenge [2]. Sequence alignment, one of the most demanding computational problems addressed in sequencing studies, has numerous applications, including read mapping. The goal of read mapping is to align the reads extracted from the sequencing machines against a reference genome, i.e. for each read the objective is to find the best matching locations when compared to a reference genome [3]. In order to restrict the search space, a common strategy is to use a seed-and-extend approach [4]. Reads are partitioned into small pieces which are searched using exact matching in order to find seeds in the reference genome. Then, a dynamic programming scheme, typically based on the Smith-Waterman algorithm, is used to assign an alignment score for each of the candidates [5, 6].

These workloads are becoming a whole new application domain in High Performance Computing (HPC) [7]. Therefore, porting and optimizing well-known sequence alignment tools like BWA-MEM2, that primarily target x86_64 architectures, to new architectures and emerging technologies is of particular interest. In recent years, Arm-based server (Ampere [8]), cloud (Graviton2 [9]), and HPC (A64FX [10]) solutions are gaining market adoption. Moreover, Amazon AWS EC2 instances that feature in-house chips like the Graviton2 are becoming extremely popular. These trends suggest that sequence alignment tools should be ready to exploit Arm-based hardware.

In this paper we port BWA-MEM2 to the ARMv8-A architecture specification and exploit the newly introduced Arm Scalable Vector Extension (SVE). We provide details of the porting effort required, which mainly involves moving from x86_64 vectorization intrinsics to SVE code. SVE code is vector length agnostic, that is, it can run on any architecture implementing SVE with vector lengths ranging from 128 to 2048 bits. We evaluate our port on Fujitsu's A64FX processor, the first to implement the SVE instruction set (512-bit vectors), and used in the Fugaku Supercomputer that was ranked 1st in the Top500 list from June 2020 to November 2021 [11].

In addition, we propose several optimizations to improve the performance of BWA-MEM2 on the A64FX. Some of this optimizations are generic while others take advantage of the A64FX underlying architecture. Finally, we compare performance and energy-to-solution of optimized implementations running on the A64FX and an Intel x86_64 Skylake architecture that features AVX-512. We show that the A64FX performance is below that of the Skylake system, since sequence alignment applications are heavily constrained by memory latency and the Skylake architecture is better optimized in this regard; whereas the A64FX is optimized to provide high memory bandwidth. However, we also show that the A64FX presents better energy-to-solution results than the Skylake system.

Our main contributions can be summarized as follows:

- Port BWA-MEM2 to the AArch64 architecture and SVE.
- Optimize BWA-MEM2 for the A64FX processor by taking into account its architectural characteristics.

- *R. Langarita is with the Barcelona Supercomputing Center, Barcelona, Spain. E-mail: {ruben.langarita}@bsc.es*
- *A. Armejach and M. Moretó are with the Barcelona Supercomputing Center, Barcelona, Spain and Universitat Politècnica de Catalunya, Barcelona, Spain. E-mail: {adria.armejach, miquel.moreto}@bsc.es*
- *P. Ibañez-Marín and J. Alastruey-Benedé are with Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, María de Luna 1, 50018 Zaragoza, Spain. E-mail: {imarin,jalastru}@unizar.es*

Manuscript received xxx xx, xxxx; revised xxx xx, xxxx.

- Detailed comparison of BWA-MEM2 executions on Fujitsu's A64FX and a well-known Intel Skylake system.

This paper is structured as follows. Section 2 describes the state of the art in relation to genomics algorithms. Section 3 shows the architecture of the A64FX, and compares it against an Intel Skylake system. Section 4 presents BWA-MEM2, the application that concerns us. Section 5 describes the porting process in order to run BWA-MEM2 on a machine with SVE support. Section 6 explains different performance optimizations for the BWA-MEM2 code. Section 7 details the parameters and inputs used for the evaluation. Section 8 presents the results obtained with the A64FX and the Intel system. Finally, Section 9 concludes this work.

2 Background on Sequence Alignment

Many tools have been created in the context of sequence alignment [12, 13, 14, 15, 16, 17]; even outside the genomics context, like the *diff* tool in Linux [18] or *Shazam* [19], the popular application to recognize music. Read mapping tools employ sequence alignment algorithms to find the place in the reference (e.g. the human genome) where a sequence fits better. These input sequences are generated by sequencing machines that output read chunks of introduced samples. These reads can range from hundreds (e.g., in Illumina machines) to thousands of base-pairs (e.g., in PacBio machines) [7]. One of the main objectives of read mapping tools is to reassemble these read chunks to obtain the complete genome. In this process, a reference genome already assembled is used, since the genome of two individuals from the same species differs by approximately 0.1% [20].

Most read mappers are based on two key observations: (i) the reference genome is large, i.e., around 3 giga base-pairs (Gbp) for the human genome, which makes naive approaches unfeasible, and (ii) DNA sequences of the same species are likely to contain short highly matched substrings. Considering these insights, most aligners follow a strategy consisting of two steps: seeding and extending. The seeding step locates the regions within the reference genome where a substring of the short input read is highly matched. This substring is known as a seed. After seeding, in the extension step, the remaining part of the read is aligned to the reference genome around the seed, allowing certain number of differences.

2.1 Finding Seeds

In order to find seeds, most mappers use one of two algorithmic approaches: FM-Index or hash tables. FM-Index can search for a variable size chunk and is typically employed if input read sequences are short, as search time is proportional to the input read length. In contrast, hash tables define a fixed size to be searched for, and are often employed for long reads, as it yields better results [15]. Both algorithms are memory bound due to their irregular memory access patterns, and require a large amount of memory capacity to store the data structures.

There are multiple existing approaches to improve the performance of this step. A common approach is to perform multiple seed searches in parallel for different input reads, as these are independent. Due to the irregular access pattern nature of these algorithms, some approaches colocate data within the same cache line in order to have better locality [21], while other approaches redefine the data structure to compress its memory footprint and enable more aggressive multi-base search steps [22].

2.2 Extend Using Smith-Waterman

For the extend part, the most used algorithm is Smith-Waterman [23, 24] and variants that improve its performance based on observations [5, 6, 25]. The algorithm builds a matrix to score inexact matches of the reference and the input read. To compute the *(i,j)* cell of the matrix, it needs the *(i-1,j)*, *(i,j-1)* and *(i-1,j-1)* cells, which causes a heavily constrained computation due to these dependencies.

The seed-and-extend strategy allows to limit the search space of the Smith-Waterman algorithm by only populating the matrix around the seeds. Moreover, most seeds end up with a low score in the extend process, which adds quite a lot of compute overhead that is later discarded. For this reason, some approaches identify and discard low quality seeds, e.g., FastHASH [26], GateKeeper [27], and Shouji [28].

As in the seeding step, alignments for different seeds are independent and can be efficiently distributed across different threads. In addition, since this is a compute bound algorithm over a matrix, there has been a significant effort to vectorize it. We can differentiate two ways of vectorization: intertask and intratask. Intertask vectorization places a different alignment on each element of the vector. However, this leads to imbalance due to the variable size of the matrices the algorithm needs to compute, and quickly becomes inefficient. In contrast, intratask vectorization tries to apply vectorization to a single alignment. However, this again proves difficult due to the constraints imposed by the algorithm in terms of dependencies. The straightforward implementation vectorizes the anti-diagonal [29], but it delivers limited performance as the access pattern is strided. Some approaches vectorize the columns, ignoring certain constraints [30], or rebuilding vertical constraints in a second run [31].

Since vectorizing this algorithm is not straightforward, all implementations resort to hand-written ISA-dependent intrinsics, as the compiler is not able to auto-vectorize it. As a result, most aligners only have these vectorized versions for what has been the dominant ISA in the last decades, i.e., x86_64. Therefore, the optimized versions of the aligners only work out-of-the-box on x86_64 machines, and the Smith-Waterman algorithm needs to be ported in order to make it run with other ISAs.

2.3 Sequencing Technologies

Since the beginning of DNA sequencing, three generations of machines have been created. The first generation uses the Sanger sequencing method, which was used to sequence the first human genome in 2001. These machines are being replaced by the second and third generation.

The second generation is able to read fixed length sequences, usually around 100 bps, with a high throughput. The third generation reads variable length sequences, with a much larger size, usually around 10 Kbps. However, the third generation has less throughput and the quality of the

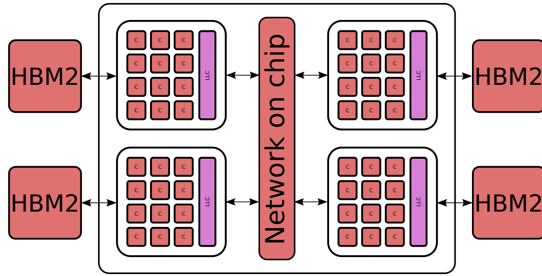


Fig. 1: A64FX scheme. We can distinguish the four different core memory groups (NUMA domains), each one with 12 cores.

TABLE 1: Core out-of-order resources in number of entries.

	A64FX	SKX
Re-order buffer	128	224
Instruction window	20+20+10+10+19	97
Scalar registers	96	180
Vector registers	128	168

read sequences is extremely low [7]. The quality issue can be mitigated by reading the same chunk multiple times, and then going through a consensus process, which leads to final read sequences that have higher quality and longer length than those obtained from second generation machines.

2.4 Sequence Alignment Applications

Most aligners optimized for short reads produced by second generation machines employ the FM-Index algorithm to find seeds. Examples include BWA [12, 13], its newer version BWA-MEM2 [14], Bowtie [32], GEM [17], and RMAP [33]. GEM keeps adding characters to a region until the number of seeds for that region falls bellow a threshold; then, they start a new region, and it repeats the process until the end of the read. Later, GEM verifies the candidate matches using Myers' fast bit-vector algorithm [34]. When reading, the sequencer assigns different qualities to the different base-pairs of a read. RMAP exploits this information by reducing the penalty of mismatches over base-pairs with low quality.

Minimap2 [15] is an example of an optimized aligner for long reads produced by third generation machines. It handles long reads in an efficient way by using hash tables to collect seeds. Due to the fixed size of the seeds, Minimap2 perform an extra step of chaining seeds to reduce the search space. This step looks up for seeds together in the query and in the reference genome to chain several seeds and increase the number of consecutive hits. For the extend part, minimap2 runs Suzuki-Kasahara [35], a modification of the Smith-Waterman algorithm that operates with the deltas among the cells, and allows 8-bit computation regardless of the size of the query. Recently, Kalikar et al. [36] detail a series of optimizations to accelerate the three main computational kernels of the CPU version of minimap2.

2.5 Sequence Alignment Accelerators

Multiple FPGA and ASIC-based accelerators have been proposed for genomics workloads. For instance, Darwin [37] and follow-up work by Olson et al. [38] propose a pipeline that has novel hardware-accelerated algorithms for seeding and alignment. GenAsm speed-ups the process even further at the cost of lower accuracy by leveraging a simpler alignment algorithm [39]. Finally, SeGraM extends the GenAsm work by adapting the pipeline for a sequence-to-graph alignment application [40].

We can also find hardware acceleration for specific kernels. For example, Guo et al. [41] accelerate the chain kernel from the Minimap2 tool; and Diab et al. [42] accelerate sequence alignment using Processing in Memory (PiM) systems from UPMEM. However, the adoption of these proposals is hindered by the different barriers (e.g., monetary, knowledge, deployment) present when adopting non cpu-centric technologies.

TABLE 2: Memory hierarchy overview.

	A64FX	SKX
Private L1	64 KB (4-way)	32 KB (8-way)
Private L2	N/A	1 MB
Shared LLC	8 MB × 4	33 MB × 2
Mem. capacity	8 GB × 4	48 GB × 2
Peak bandwidth	256 GB/s × 4	120 GB/s × 2
Technology	on-package HBM2	off-chip DDR4

3 Target Machine: Fujitsu's A64FX

In this work we make use of the Fujitsu A64FX. Some of the software optimizations we propose in Section 6 take advantage of the A64FX architecture, which differs significantly from conventional x86_64 architectures. Therefore, in this section we describe and characterize the A64FX architecture to better understand its trade-offs and optimize for them.

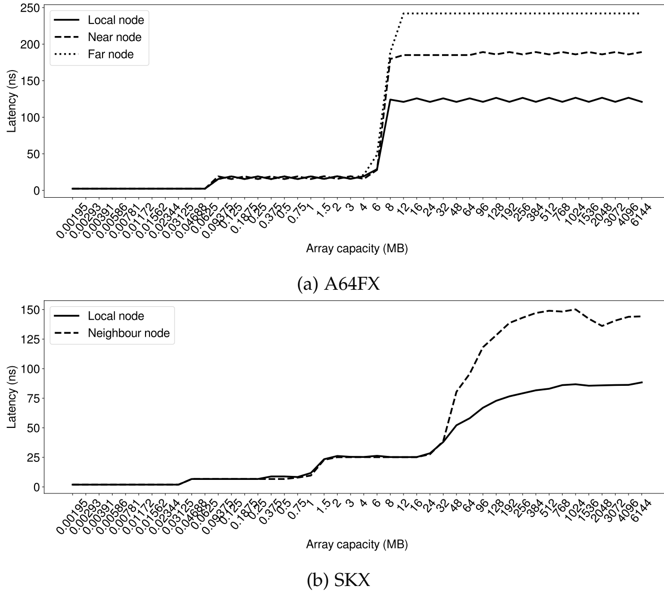
The A64FX was launched in 2019, however, accessibility to the machine was restricted until the second half of 2020. Figure 1 shows an overview of the chip, which features a total of 48 compute cores and four HBM2 interfaces. The A64FX chip has been designed to perform well under HPC workloads with stringent memory bandwidth requirements. Its architecture is based on four Non-Uniform Memory Access (NUMA) domains within the same chip and was the first to implement SVE [43, 44, 45]. In this section, we present an overview of the A64FX and compare its main features against an Intel Xeon Skylake system (SKX), also HPC oriented.

3.1 Core Out-of-Order Resources

The A64FX has moderate out-of-order resources, as we can see in Table 1. The number of entries in the ROB is nearly the double for SKX. A64FX divides its 79-entry instruction window in 5 different reservation stations, one for each execution path: 2 paths for arithmetic operations (20 entries for each path), 2 paths for memory operations (10 entries for each path), and 1 path for branch operations (19 entries). In contrast, SKX has 97 entries in a unified instruction window. In addition, the number of physical registers is lower in the A64FX: 96 vs. 180 physical scalar registers, 128 vs. 168 physical vector registers, for A64FX and SKX, respectively. Both architectures implement 512-bit vector functional units.

3.2 The A64FX Memory Hierarchy

The A64FX memory hierarchy differs significantly from the traditionally employed in x86_64 platforms like SKX. The



(a) A64FX

(b) SKX

Fig. 2: Memory latencies measured by the *lmbench* benchmark for A64FX and SKX.

TABLE 3: Memory latency in nanoseconds obtained with *lmbench* benchmark for A64FX and SKX. This table is a simplification of Figure 2.

	A64FX	**SKX**
Private L1	2.27	1.91
Private L2	N/A	6.68
Shared LLC	17.27	25.14
Local mem.	123.87	86.2
Near mem.	187.58	144
Far mem.	242	N/A

TABLE 4: *lmbench* bandwidth benchmark results in GB/s for A64FX and SKX.

	A64FX		SKX	
	Mean	**Std Dev**	**Mean**	**Std Dev**
Local	228.9	0.028	116.8	0.048
Near	130.0	0.125	34.4	0.029
Far	130.0	0.005		
All	522.4	1.352	124.8	1.476
All-split	915.2	0.141	233.3	0.049

A64FX chip is organized in 4 NUMA domains, also known as core memory groups (CMG), each with 12 cores for general compute. Each domain has its own local memory interface, but it can also use the interfaces of the *near* domain and the two *far* domains (Figure 1). The SKX system has 2 interconnected sockets, which translates into 2 NUMA domains with 24 cores each, also totalling 48 cores.

Table 2 details the memory hierarchy of the two systems we are comparing. Despite the A64FX having a larger L1 cache, we can see that the cache hierarchy of the SKX system has an additional cache level with large private L2 cache of 1 MB per core, totaling 48 MB of cache in this additional level. The A64FX has 8 MB of last level cache (LLC) for each core group (12 cores), a total of 32 MB, while the SKX has 33 MB per socket (NUMA node), a total of 66 MB of LLC. Also, the associativity of the L1 is bigger in SKX, 8 ways versus 4 ways in A64FX.

Each of the 4 NUMA domains of the A64FX are connected to 8 GB of on-package memory implemented using HBM2 technology, leading to a total of 32 GB of main memory that cannot be extended via conventional off-chip DIMMS. In contrast, the SKX system has 48 GB of main memory per NUMA domain, 96 GB in total. In the case of SKX, the amount of memory can be further extended if higher capacity DIMMS are used. This memory capacity limitation is a negative factor for the A64FX, especially for genomics applications, where memory capacity can usually be traded for computation. However, bandwidth is the strong point of the A64FX, as it has a peak bandwidth of 256 GB/s per NUMA domain, reaching 1 TB/s total peak bandwidth. The SKX system has a peak bandwidth of 120 GB/s per NUMA domain, 240 GB/s in total.

3.3 Memory Hierarchy Latencies

To obtain the different memory latencies, we have executed the *lmbench* benchmark [46]. *lmbench* instantiates an array with a fixed capacity and performs random accesses to it. If the array fits in a given cache level and not in lower ones, *lmbench* can obtain the latency for that level, since the random access ensures that the data is not in a lower level.

Figure 2 shows the obtained results, which present a stair shape. Each *step* in the plots corresponds to a different memory level, and the array capacity of each *slope* matches with the capacity described in Table 2. For example, for the A64FX, we can see a *step* between 0.0625 (64 KB) and 8 (8 MB), the capacity of L1 and a slice of LLC respectively, this means that this latency corresponds to the LLC. For the sake of simplicity, Table 3 shows the latencies for each cache level in nanoseconds.

We observe that the LLC latency is 31% lower in A64FX with respect to SKX. However, SKX has 1 MB of L2 cache per core with a compelling latency. When accessing memory, the latency is almost 50% higher in the A64FX for the local domain. In addition, accessing a far NUMA domain in the A64FX is really costly, nearly twice the latency of the local memory. Memory latency has less importance in regular applications that can benefit from spatial or temporal locality, however, genomics applications can suffer a significant penalty due to their irregular access patterns.

3.4 Memory Bandwidth

We also employ *lmbench* to measure bandwidth per NUMA domain. *lmbench* runs a simple STREAM benchmark to obtain the bandwidth. We explore several configurations:

- Local: threads from a NUMA domain request data to their local memory.
- Near: threads from a NUMA domain request data to their neighbour (near) memory.
- Far (only in A64FX): threads of a NUMA domain request data to a far memory.
- All: threads from all domains request data. The memory is interleaved among all NUMA domains.
- All-split: in order to obtain the peak theoretical bandwidth, we instantiate one process on each NUMA domain (four in A64FX and two in SKX) using the *local* configuration.

We can see the results in Table 4, where we perform three executions for each configuration, and we report the

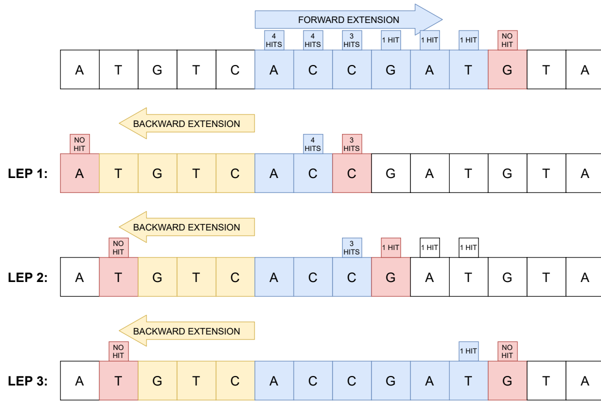


Fig. 3: SMEM process scheme.

average and the standard deviation. We can see that for the *local* configuration, the results are consistent with the peak bandwidth specified for each memory technology on both machines. The *near* and *far* configurations are influenced by the characteristics of the network-on-chip and the socket-to-socket connection in the SKX case; consistently obtaining lower performance. In the *All* configuration, we observe that the measured bandwidth is far from the peak theoretical bandwidth. Since the memory is allocated in an interleaved manner, the requests from different NUMA domains are hindering memory performance. Finally, with the *All-split* configuration, we obtain a bandwidth close to the peak theoretical bandwidth.

4 BWA-MEM2

For this work, our target application is BWA-MEM2. BWA-MEM2 [14] is a popular sequence alignment tool that enhances its previous version, BWA [12, 13]. It contains multiple optimizations that considerably speed-up alignments while maintaining the same output. It supports all kind of reads, but it is optimized for short reads produced by second generation sequencing machines.

BWA-MEM2 follows a seed-and-extend approach. To profile the application, we define three regions of interest:

- **SMEM**: uses an FM-Index structure to find seeds based on exact matching.
- **SAL**: a suffix array translates FM-Index positions to the locations in the reference genome.
- **BSW**: the Banded Smith-Waterman algorithm extends the seeds and scores the alignments.

These three regions account for between 76% and 86% of the total user execution time of BWA-MEM2. The remaining time accounts for thread synchronization, rearranging data between phases (converting structure of arrays to arrays of structures, and vice versa) and allocating data structures.

4.1 The SMEM Region

To obtain the seeds, BWA-MEM2 queries an FM-Index structure to perform exact search on portions of the input reads and the reference genome. This structure occupies nearly 10 GB for the human genome. This process tries to find Super Maximal Exact Matches (SMEM). A MEM is a fragment of the read input that matches with the reference and it can not be further extended. A SMEM is a MEM that is not contained in any other MEM.

Figure 3 shows the process of getting the SMEMs. First, we fix a point in the read input sequence to start from, the left most blue base in the figure. By querying the FM-Index, we keep expanding the seed (forward extension) until we have no hits in the reference, the red base at the end of the forward extension. During this forward extension, we store the positions where the number of hits changes. These points are called left extension points (LEP). In Figure 3, we have three LEPs, when going from four hits to three hits (LEP1), going from three hits to one hit (LEP2) and going from one hit to no hits (LEP3). Then, the algorithm expands backwards the LEPs until there are no hits in the reference genome. Finally, to obtain the SMEMs, we discard the MEMs that overlap with other MEMs. In our example, the MEM produced by LEP2 is discarded since it overlaps with the MEM produced by LEP3. Therefore, 2 SMEMs are found, those originating from LEP1 and LEP3. These two seeds will later be extended during the extend phase implemented in the BSW region.

For each base-pair, the application has to do a random access to the FM-Index structure. The value read determines the row of the next access, which produces a chain of dependent random accesses. Since the size of the FM-Index structure is usually prohibitive, BWA-MEM2 trades computation for memory storage, like many other proposals. This is achieved by only storing 1 out of every 64 entries of the FM-Index. The missing values are restored using an auxiliary data structure (BWT) that is encoded using 2 bits for each base-pair and additional computation. The closest counter for the required FM-Index row is read, and then with the help of the BWT, it counts the base-pairs needed to reach the final row. This process is in the critical path between two dependent random accesses; therefore, performing this additional computation quickly is essential to the overall performance. In BWA-MEM2 this computation consists of a vector comparison and a population count instruction to count the occurrences of a base-pair in the BWT.

The use of the FM-Index produces a memory bound execution, since one iteration depends on the previous one, and each iteration requires a long latency random access to the structure. In this case, the limitation is not memory bandwidth but its latency. To mitigate the effects of this latency, BWA-MEM2 implements software prefetching in order to load the data in advance, thus reducing the time that the CPU pipeline is blocked. Software prefetching is implemented both for the forward and backward extensions. In the case of the backward extension, the code is written in such a way that the different LEPs are independent and can be interleaved. Since the LEPs can be computed independently, the backward extension memory accesses do not block the CPU pipeline.

4.2 The SAL Region

The SMEM region finds the seeds based on exact matching that will be later extended using BSW. However, the position in the FM-Index found in the SMEM phase has to be translated to obtain the position of the seed in the reference. This is done through the Suffix Array (SA). For

the human genome, an uncompressed SA requires 48 GB, a considerable size that will not fit in certain systems. For this reason the SA is also compressed. BWA-MEM2 offers a configurable compression factor x, that samples the SA by 2^x. This means that the SA will store 1 out of 2^x entries, potentially performing 2^x operations to recover the original value. We choose a compression factor of 3, i.e. our SA occupies 6 GB ($48/2^3$) of memory.

4.3 The BSW Region

Finally, for the extend part, BWA-MEM2 uses the Banded Smith-Waterman (BSW) algorithm and the affine-gap penalty model [24]. The alignment is done by populating a matrix with as many columns/rows as the input read. Instead of computing all cells in the matrix, BSW only computes the cells around the main diagonal. The width of this band depends on the score of the previous rows. If the score of one row drops drastically, the width of the band is reduced or even the algorithm can stop. The affine-gap penalty model adds a penalty for opening a gap. This means that two separate gaps produce a worse score than two consecutive gaps.

In contrast with SMEM and SAL, BSW is a compute bound kernel. BWA-MEM2 uses x86_64 intrinsics to implement BSW, and it has 3 versions: SSE, AVX and AVX-512. It uses an inter-task approach, this means that each element of the vector computes one read input sequence.

5 Porting the code to AArch64

The main challenge when porting BWA-MEM2 to AArch64 and the ARMv8-A specification is that the BSW algorithm is implemented exclusively using x86_64 intrinsics, that is: SSE, AVX and AVX-512. Therefore, the main task is to port this algorithm to an ARMv8-A compatible ISA. We undertake this task and port the BSW algorithm to SVE, Arm's recently proposed vector extension that is vector length agnostic (VLA), i.e., one implementation fits all lengths, and supports vector registers of up to 2048 bits.

Since none of the existing implementations uses features present in SVE such as predication or VLA, we selected the SSE implementation to do the porting; as it has a cleaner interface and the code is easier to reason about. Certain intrinsics like arithmetic or load/store operations have a 1-to-1 translation, in these cases we overload the SSE intrinsic name with an inline function that reimplements its functionality using equivalent SVE intrinsics.

For this purpose, we create a header file called `sse2sve.h` that implements all the code related to SVE direct translations. Additional details on how these functions work and an example can be found in Section 5.3. However, complex code regions such as boolean operations or vector-width dependent code are translated in a case-by-case basis, since SVE instructions have a different structure and can benefit from features such as predication. Section 5.4 includes further details.

5.1 Data Types

The first step is to translate the data types between the two architectures. SSE has only one data type `__m128i`, and the data type of the lanes is selected using the intrinsic suffix. For example, it would use `epi16` for signed 16-bit integers, and `epu8` for unsigned 8-bit integers. However, SVE specifies the data type of the lanes but not the vector length. For example, the `svint64_t` data type indicates that the vector lanes will be signed 64-bit integers, but it does not specify how many elements the vector has, i.e., VLA programming. In our translations, we generalize the data types to always be signed 64-bit integers:

```
typedef svint64_t __m128i;
```

Later, we use a *reinterpret* intrinsic in the translation function to convert this generic data type (`svint64_t`) to the data type of the operand in the original instruction being translated. For example, we would use the intrinsic `svreinterpret_s8` to convert the variable to a signed 8-bit integer. Note that while we use multiple intrinsics, the final result of the translation is a single assembly instruction, as we explain in the following example.

TABLE 5: Translations in the `sse2sve.h` file.

SSE intrinsic	SVE translation
`_mm_malloc`	`aligned_alloc`
`_mm_free`	`free`
`__rdtsc`	`cntvct_el0 (hardware counter)`
`_mm_prefetch`	`__builtin_prefetch`
`_mm_setzero_si128`	`svdup_s64(0)`
`_mm_set1_epi{8,16}`	`svdup_s{8,16}`
`_mm_blend_epi{8,16}`	`svsel`
`_mm_add_epi{8,16}`	`svadd_x`
`_mm_adds_epu{8,16}`	`svqadd`
`_mm_sub_epi{8,16}`	`svsub_x`
`_mm_subs_ep{i,u}{8,16}`	`svqsub`
`_mm_max_ep{i,u}{8,16}`	`svmax_x`
`_mm_min_epu{i,u}{8,16}`	`svmin_x`
`_mm_and_si128 (arithmetic)`	`svand_z`
`_mm_and_si128 (predicate)`	`svand_x`
`_mm_or_si128 (arithmetic)`	`svorr_z`
`_mm_or_si128 (predicate)`	`svorr_x`
`_mm_xor_si128 (arithmetic)`	`sveor_x`
`_mm_andnot_si128 (arithmetic)`	`svbic_x`
`_mm_andnot_si128 (predicate)`	`svbic_z`
`_mm_cmpeq_epi{8,16}`	`svcmpeq`
`_mm_cmpgt_epi{8,16}`	`svcmpgt`
`_mm_cmpge_epi16`	`svcmpge`
`_mm_load_si128`	`svld1`
`_mm_store_si128`	`svst1`

5.2 Porting Effort

In Table 5 we show all the SSE intrinsics we have translated in `sse2sve.h` and their corresponding SVE translation. All the functions are marked and forced to be *inline* to avoid function calls and achieve 1-to-1 assembly translations.

5.3 *add* Translation Example

SSE instructions start with the prefix `_mm`, followed by the instruction type, and the lane width. For example, `_mm_add_epi8` indicates an addition instruction with signed 8-bit integers. We translate this intrinsic by declaring a function that overloads its name:

```
inline svint64_t _mm_add_epi8(svint64_t a, svint64_t b) {
        svint8_t a_aux = svreinterpret_s8(a);
        svint8_t b_aux = svreinterpret_s8(b);
        svint8_t r_aux = svadd_x(svptrue_b8(),a_aux,b_aux);
        return svreinterpret_s64(r_aux);
}
```

We reinterpret the source vector operands as signed 8-bit integers, perform the svadd operation, and finally reinterpret the result back to signed 64-bit integers. svptrue_b8 indicates that all lanes are active, since SSE instructions can not be predicated. By doing this, the original SSE code remains unmodified. Even though there are 4 lines of C++ code, this function translates to a single SVE assembly instruction that performs the vector addition:

```
add z7.b, p0/x, z7.b, z26.b
```

SVE architectural registers are {z0..z31} and the .b suffix in a register name indicates that the register is interpreted as a vector of signed 8-bit integers. The p0 register contains the predicate with all lanes active. The compiler sets p0 only once for all SVE instructions within a function block. The end result for all translations performed this way is a 1-to-1 assembly instruction correspondence.

5.4 Boolean Translation Example

BWA-MEM2 uses two ways of storing boolean data types:

- A vector variable with all bits of each lane set to 1 if true, or 0 if false. This is used to mask off elements in vector operations.
- A scalar variable with single bits set to 1 or 0. It uses these scalar variables to create boolean expressions.

SVE has a predicate data type, svbool_t, to store booleans. We take as example the following code snippet:

```
uint32_t cval = _mm_movemask_epi8(cmp1);
if (cval == 0x00) break;
```

cmp1 is a vector variable, previously generated, which acts as a boolean predicate to perform masking operations. _mm_movemask_epi8 moves the least significant bit of each lane to a scalar variable cval. Then, cval is checked as an exit condition for a loop. Therefore, this code requires the creation of the scalar cval mask and then a comparison operation. However, we can translate this code to SVE in a way that the comparison can be done directly using cmp1:

```
if (!svptest_any(svptrue_b8(),cmp1)) break;
```

cmp1 is already of type svbool_t, hence, there is no need of converting it to a scalar data type. We use svptest_any to check if any lane is active, and then, we negate the result. Note that the result of the boolean expression is the same as for the SSE version.

5.5 Challenges

BWA-MEM2 was not ready to execute on any AArch64 machine since certain parts (BSW) are exclusively written using x86_64 intrinsics. Moreover, A64FX was the first to implement SVE and employs a non-conventional memory subsystem. Therefore, we found challenges at the application code, system software, and machine level.

5.5.1 Application Code

BWA-MEM2 is widely used in the genomics community. It was released in 2019 as an upgrade of BWA. Since then, it remains in continuous development. The authors continue solving issues and giving support to the users.

After the start of the port there have been two significant new releases of BWA-MEM2. A stable version 2.0 was released the 9th of July 2020, fixing several bugs, and on the 15th of October 2020 a new feature that allows compressing the Suffix Array (SA) was introduced. Prior to this update we were not able to employ the widely-used human genome, since it did not fit within the A64FX memory capacity. The compression of the SA enables the use of large genomes, including the human genome. Both releases has a small impact on the porting itself as the BSW code remained unchanged.

The main difficulty during the porting was to translate the BSW x86_64 intrinsics to SVE. Most of the code is translated using a header file created for this purpose, which contains one-to-one intrinsics translations to SVE. However, in some cases, we had to rewrite the original code to maintain the semantics, as masks behave differently in SVE (see Section 5.4).

TABLE 6: Execution time relative standard deviation for 10 runs using D3, D4 and D5 inputs (see Section 7.1) and the three code regions, with and without the thread affinity.

	D3			D4			D5		
	SMEM	SAL	BSW	SMEM	SAL	BSW	SMEM	SAL	BSW
w/o affinity (%)	10.2	15.8	8.5	11.3	22.0	8.9	11.2	15.4	10.7
w/ affinity (%)	0.15	0.11	0.42	0.17	0.09	0.65	0.15	0.12	0.38

5.5.2 System Software

Since the A64FX software stack is not mature yet, we have found issues with compilers and libraries.

A64FX was the first processor to implement SVE. Due to the novelty of SVE, at the time of writing this paper, not many compilers support SVE intrinsics. For example, the native Fujitsu compiler (FCC) has many optimizations that target the A64FX but still lacks SVE intrinsics support and we were not able to use it for our experiments. However, FCC has a clang back-end mode that does support intrinsics but at the cost of certain A64FX-specific optimizations. Both GCC11 and Arm HPC compiler also support intrinsics. Therefore, for this work we have employed the FCC compiler with clang backend, as it showed better performance than the rest of the compilers. FCC obtained 3.46%, 4.10%, and 6.07% better performance with respect to GCC11 (the second best compiler) for D3, D4, and D5 inputs (see Section 7.1), respectively.

Large memory pages are essential in applications with irregular access patterns to lower the cost of virtual to physical address translations. To use the larger 2 MB virtual pages, the compiler needs a specific large page library. This library had a bug that lead to performance inefficiencies, and needed additional configuration steps via environment variables that are not trivial. Once the library was fixed and configured properly, the performance of the code compiled with the FCC compiler improved considerably.

5.5.3 Execution Time Variance

We observed a lot of variability when comparing multiple executions using 48 threads. Since the chip has multiple NUMA domains, the custom thread scheduler tries to optimize thread placement, which leads to unexpected thread migrations that were not observed on other machines. These migrations would happen even among different core memory groups, leading to significant performance variation across multiple executions.

We solved this problem by using the pthread affinity functionality, pinning each thread to a core. Table 6 shows the relative execution time standard deviation for 10 different runs, with and without the thread affinity functionality. For OpenMP codes the flag `OMP_PROC_BIND` needs to be used to prevent variability across executions.

6 Optimizations

This section describes several optimizations we have tried over the ported code. First we describe optimizations that apply to the entire application, and then those that target a particular code region. These optimizations try to take advantage of the A64FX architecture, but are likely to perform well on most systems. Some of the optimizations we tried did not yield the expected result, but are explained regardless to provide the lessons learned. Each of the optimizations that had a positive effect is later evaluated in Section 8.

6.1 General Optimizations

Split Input: The A64FX has 4 core memory groups, each of them with 12 cores and 8 GB of shared HBM memory. Each core group sees one group as a near domain, and the other two groups as far domains. Accessing the near domain has a significantly lower latency as shown in Section 3.3 compared to the far domain. In order to avoid accesses to far domains, we have tried to split the input, and execute two different processes, each one on two near domains. By doing this, we would have two processes using 24 cores each (2 CMGs), but each process would require copy of the index and all necessary data structures. After trying multiple combinations and compression schemes, we concluded that this is not a feasible option on the A64FX due to its total memory capacity of 32 GB.

Large Pages: In the A64FX, the default page size is 4 KB. Such small pages are very inefficient on irregular memory accesses since each access will likely go to a different page, requiring a new virtual to physical address translation. Larger 2 MB pages can be enabled when using the FCC with a specific library. The amount of memory covered by these larger pages makes it more common to have temporal page address translation hits, leading to better overall performance. The random accesses into the large FM-Index would benefit from support of even larger page sizes. Most x86_64 HPC systems have support for so-called *huge* pages of 1 GB, but the A64FX does not support pages larger than 2 MB.

6.2 SMEM Region

For the SMEM region, we have tried four optimizations, from which three have lead to positive results.

6.2.1 Aggressive Inlining

We detected a performance issue when calling the function `backwardExt` that performs the backwards extension to find SMEMs. This function performs an access to the FM-Index and then reconstructs the missing entries with a comparison and a population count operation. Besides the negligible overhead of the branch needed to do the function call, this function has large `struct` parameter passed by

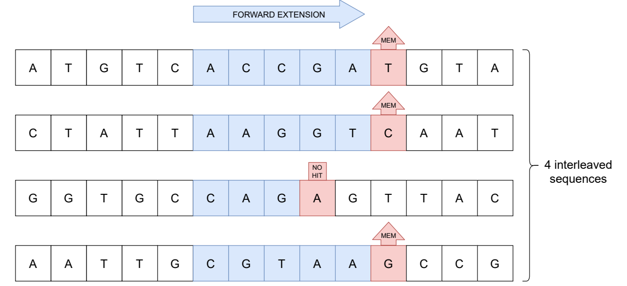


Fig. 4: Interleaving four SMEM processes for different input reads.

value. In addition, the code before and after the function call that manipulates parameters is significant.

While the compiler did not inline this function by using the regular `inline` keyword due to its size, we forced inlining by using the `always_inline` attribute:

```
inline __attribute__((always_inline)) SMEM
          backwardExt(SMEM smem, uint8_t a);
```

By forcing this function to be inlined, the movement of data on the stack is avoided, and the compiler is able to optimize the code beyond the function limits. The code reduction is substantial as the creation of the `SMEM struct` that was passed by value is no longer needed.

6.2.2 Population Count (popcnt)

Counting the number of bits set in a register (the `popcnt` operation) is a critical operation for the FM-Index algorithm as it is on the critical path between two dependent irregular memory accesses. BWA-MEM2 relies on the compiler built-in function `__builtin_popcount`, that on x86_64 translates into a single instruction as the hardware supports this operation. However, on the A64FX the built-in performs the operation using bitwise operations with masks and sums to obtain the result [47]. As opposed to modern x86_64 architectures, the ARMv8.2-A specification does not include an instruction to perform a *popcnt* over a scalar register. However, SVE does have a specific vectorized *popcnt* instruction. Therefore, we rewrote the code using SVE intrinsics by moving the scalar register into a vectorial one, performing the *popcnt* using the `svcnt` intrinsic, and moving back the result to a scalar register again.

6.2.3 Interleaved Sequences

The A64FX has a high memory access latency compared with HPC systems. However, it does have a significant advantage in terms of memory bandwidth. As we explained in Section 4.1, the forward extension used to obtain SMEMs leads to chains of dependent long-latency memory accesses. Therefore, the latencies of each individual access cannot be hidden. To solve this issue and allow hiding these latencies, we propose to interleave multiple accesses to the FM-Index from different input reads, effectively performing several forward extensions in parallel, as shown in the Figure 4. This exposes more memory level parallelism as more accesses are in-flight, hiding their latencies; and the core pipeline has additional work to do with multiple forward extensions running. Prior work showed this is an effective technique when memory bandwidth is abundant [22].

6.2.4 Manual Loop Fission

The A64FX has less out-of-order resources than other HPC processors. Limiting the number of instructions within a loop is very important to save resources, such as reorder buffer storage or physical registers. To enable aggressive loop unrolling the FCC compiler has an automatic loop fission feature that can be configured via a specific `pragma`. By using this technique loops can be split, providing smaller loop bodies, which leads to a better utilization of the out-of-order resources [48].

Unfortunately, support for automatic loop fission via the specific `pragma` using the FCC compiler is not compatible with the use of SVE intrinsics. The compiler cannot determine how to split the loop in the presence of intrinsics. Therefore, we have decided to split manually three loops in the SMEM region. However, this is a difficult process to perform manually and our efforts did not produce satisfactory results. Therefore, we defer this optimization to future work, and will test again this feature once its support is extended to accept SVE intrinsics within the loop body.

6.3 BSW Region

We focused on improving the port by taking advantage of the predication feature present in SVE.

BWA-MEM2 uses a selection instruction for zeroing useless results within a vector. We can eliminate these instructions by predicating the instruction that produces the result. As an example, we will take the following piece of code:

```
__m128i m11 = _mm_add_epi8(h00, sbt11);
__m128i cmp11 = _mm_cmpeq_epi8(h00, zero128);
m11 = _mm_blend_epi8(m11, zero128, cmp11);
```

In this code, after performing the *add* operation, a predicate is built with `_mm_cmpeq_epi8`. `_mm_blend_epi8` selects the lane from `zero128` if `cmp11` has a *1* stored for that lane, or takes the value in `m11` otherwise. `zero128` contains 0s for all lanes. This means that the instruction overrides the result obtained from the *add* with 0s where the predicate `cmp11` is 1. Since SVE has instruction predication support, we do the following translation:

```
svbool_t cmp11 = svcmpne_n_s8(svptrue_b8(), h00, 0);
svint8_t m11 = svadd_z(cmp11, h00, sbt11);
```

First, we build the predicate in a similar fashion and store it in *cmp11*. Then, the *add* operation can be directly predicated, indicating that non-active lanes in the mask need to be zeroed. This is accomplished with the appropriate *_z* suffix. By doing this we save instructions within performance critical tight loops. Note that we can not do this by overloading the functions presented in Section 5.2 because we are combining two instructions into one. We have applied this technique when possible. Since the algorithm is compute bound and we are reducing the compute pressure within tight loops, we are able to gain performance by leveraging SVE's predication.

7 Evaluation Methodology

In this section, we describe the evaluation methodology employed to assess our BWA-MEM2 port on the A64FX.

TABLE 7: Details for D3, D4 and D5 input datasets.

	D3 [49]	D4 [50]	D5 [51]
Organism	Homo Sapiens	Homo Sapiens	Homo Sapiens
Machine	Illumina Genome Analyzer II	Illumina HiSeq 2000	Illumina HiSeq 2000
Seq. length	76	101	101
Num. of seq.	17.8 M	92.4 M	1,436.8 M
Run	SRR043348	SRR622461	SRR622457

TABLE 8: Test machines configuration.

	2 × Intel Xeon Platinum 8160 (SKX)	A64FX
Cores	2 × 24	48
Issue width	4	4
Frequency (GHz)	2.1	2.2
Last-level Cache (MB)	2 × 33	32
Vector extension	AVX-512	SVE 512 bits
Main memory	DDR4, 2×48GB, 2×120GB/s	HBM2, 32GB, 1024GB/s

7.1 Reference and Inputs

We use as reference the human genome (GRCh38), which was built in 2013 and contains 3.05 Gbp [52]. Therefore, we build the FM-Index from this reference. This process typically takes a few hours, however, this time is negligible compared to the time spent processing the alignments, since the index is built only once and can be reused as many times as necessary for multiple alignment experiments. Precomputed indexes for popular alignment tools and reference genomes can also be found online.

We use three real inputs from the National Center for Biotechnology Information Sequence Read Archive sequenced with an Illumina machine, termed: D3 [49], D4 [50] and D5 [51]. We randomly select 1.25 million sequences from the original files. D3 has sequences of 76 bps long, while D4 and D5 have sequences of 101 bps. These datasets were also used to evaluate BWA-MEM2 when it was released [14]. Table 7 shows the details of these datasets.

7.2 Evaluated Systems and Experiments

We evaluate two compute nodes: the A64FX, and one based on Intel Xeon Platinum 8160 (SKX) that features two sockets. A brief summary of their main characteristics can be found in Table 8. In total, both systems have 48 physical cores.

We perform performance experiments on the A64FX system to test the efficacy of our port and the proposed optimizations. Then, we perform a vector length sensitivity analysis on the A64FX. We evaluate vector lengths of 128, 256, and 512 bits, in order to demonstrate that our vector length agnostic code port scales well. Finally, we compare performance and energy-to-solution for the A64FX and the Skylake system (SKX).

7.2.1 Energy-to-Solution Methodology

We aim to measure energy-to-solution of the three regions of interest, avoiding I/O operations that depend on components other than the processors we are studying, i.e., file systems or disk technology. Therefore, we want to avoid measuring the energy outside the regions of interest, which includes loading the index and the input sequences, as well as writing the final output.

On the SKX system, Slurm [53] allows us to measure the energy of the entire job. Therefore, we perform two executions, the first one just loads the index and exits right before starting the first region of interest. The second run executes the application normally until the end of the last region of interest and exits. To obtain the final result we subtract the energy of the first run from the second one.

On the A64FX we use the power API from Fujitsu [54]. The power API allows us to measure the energy of a specific code region by reading a special register each time we invoke a particular API function. We placed the API calls to avoid the load of the index and the final output writing, measuring the same region as in the SKX machine.

To avoid the load of the input sequences, which happens on a per batch basis, we preload them into memory.

7.3 BWA-MEM2 Configuration

BWA-MEM2 admits multiple configuration parameters. The most relevant define the compression factors of the main data structures. We fix the level of compression for the SA to eight, this means that the SA stores one entry out of eight, potentially performing eight operations to recover a missing value. This compression is enough to fit the human genome index structures in the 32 GB of HBM2 available in the A64FX system. Similarly, the FM-Index compression factor is fixed to 64, this means that it stores 1 out of 64 entries. The missing values are restored with the BWT, that encodes the base-pairs with 2 bits.

BWA-MEM2 employs software prefetching instructions in the SMEM region in order to load data in advance, with the objective to mitigate access latency overheads. We disable the software prefetching temporarily to measure its impact. We observe an average speed-up when using software prefetching of 47.9% and 38.3% for A64FX and SKX respectively in the SMEM region. The A64FX benefits more from software prefetching as it has a higher memory access latency. Since the software prefetching feature is beneficial, and enabled by default in BWA-MEM2, we keep it enabled in all experiments.

We evaluate the application with different thread counts: 1, 12, 24 and 48. 48 threads is the maximum number for both systems, SKX and A64FX. BWA-MEM2 processes the sequences in batches. We set up the number of base-pairs per batch to 10 M per thread. We use the *pthread* affinity functionality in order to pin threads to physical cores, as we explained in Section 5.5.3.

In order to measure performance we count execution cycles. We have used *perf* to extract this information via hardware performance counters [55]. We have placed *perf* probes near the markers of each region of interest in the original code: SMEM, SAL and BSW regions.

For the A64FX and SKX, we use SVE 512 bits and AVX-512 respectively for all experiments unless otherwise stated.

7.4 Porting Correctness

In order to check the correctness of our porting effort, we compared the output files obtained using the SKX system with the unmodified BWA-MEM2 code against those produced by our port using the A64FX. Both binaries generated the exact same alignments for D3, D4, and D5 inputs.

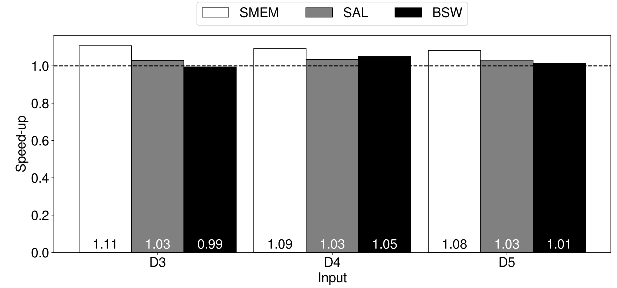
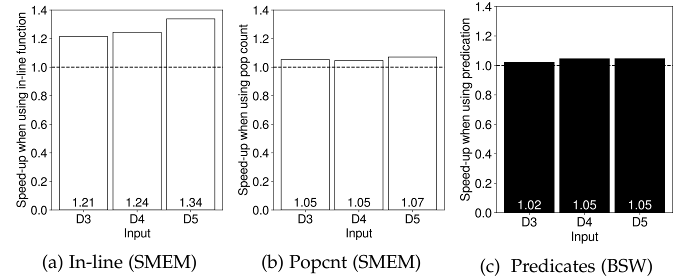


Fig. 5: Speed-ups when using large pages with respect to not using them on A64FX with 48 threads. Showing data for D3, D4, and D5 inputs and the three code sections.



(a) In-line (SMEM) (b) Popcnt (SMEM) (c) Predicates (BSW)

Fig. 6: Speed-ups for (a) in-line, (b) population count and (c) predication optimizations for D3, D4, and D5 inputs on A64FX with 48 threads.

8 Experimental Results

8.1 A64FX Optimizations Evaluation

We use as the initial baseline the BWA-MEM2 code described in Section 5. Then, we evaluate the different optimizations described in Section 6. We progressively add one optimization at a time to determine its impact on the final performance. When adding an optimization, all previously evaluated optimizations are also present. All the experiments presented in this section employ 48 threads, one per core.

8.1.1 Large Pages

Figure 5 shows the speed-ups when using large pages for each input and region of interest. The average speed-up among all the sections and all the inputs is 4.8%. However, the SMEM section is the most affected by this optimization since it is a latency bound kernel, and it has a large memory footprint, showing an speed-up of 1.11× with the D3 input.

8.1.2 Function Inlining

Figure 6a shows the obtained speed-ups when using the *inline* directive in the `backwardExt` function. This optimization only affects the SMEM section, and achieves a 1.27× average speed-up in this application phase. Therefore, the function call with parameters passed by value has a significant overhead that can be avoided.

8.1.3 Population Count

Figure 6b shows the speed-ups when adding the `population count (svcnt)` SVE instruction in the SMEM region. By replacing a sequence of arithmetic instructions for a single instruction primitive we achieve an average 5.7% performance improvement.

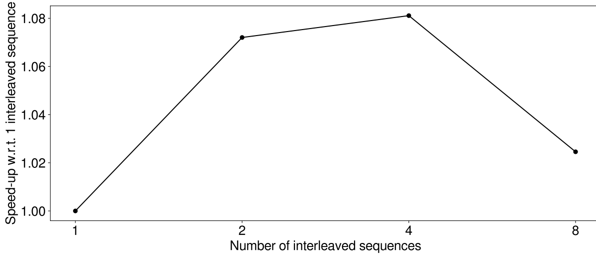


Fig. 7: Average speed-ups for the interleaved sequences optimization on SMEM for D3, D4, and D5 inputs on A64FX with 48 threads.

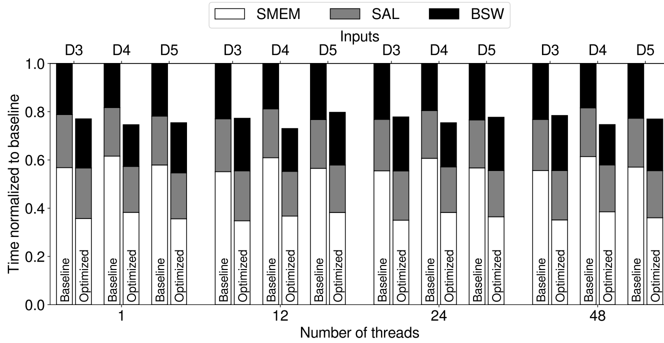


Fig. 8: A64FX normalized execution time with respect to the unoptimized baseline. Showing executions with 1, 12, 24 and 48 threads using D3, D4, and D5 inputs.

8.1.4 Interleaved Sequences

Figure 7 shows the speed-ups obtained when interleaving multiple accesses to the FM-Index from different input reads, i.e., performing several forward extensions in parallel. Again, this optimization only affects the SMEM section. We observe that the best configuration is when using 4 interleaved sequences, which gives an average 8.1% performance improvement with respect to not using interleaved sequences. Therefore, we will use four interleaved sequences from now on.

8.1.5 BSW SVE Predication

In Figure 6c, we can see the speed-ups when using SVE predication in the BSW section. Despite we only predicated a few instructions inside the inner loop of the kernel, we achieve an average 3.7% performance improvement in BSW.

8.1.6 Final Version

In total we have performed five optimizations over the baseline implementation. We can see the performance improvements achieved by all the optimizations in Figure 8. The figure shows, for multiple thread counts and for each input, the normalized execution time when all optimizations are applied (*Optimized*) with respect to the ported code (*Baseline*). We obtain an execution time improvement of 23.2% on average for 48 threads. In the figure, we can also distinguish each code region. SMEM is the region that experiences the largest performance gain, 36.9% on average for 48 threads. We also observe that the optimizations have a similar effect regardless of thread count. Therefore, they are not targetting any bottlenecks that appear due to core count scaling. As we show in Section 8.4 the application scales almost linearly with thread count.

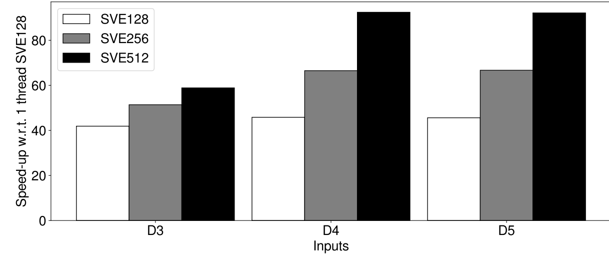


Fig. 9: BSW region speed-up for 48-thread executions with respect to single-threaded and SVE 128 bits for D3, D4, and D5 inputs.

TABLE 9: Billions of committed instructions for a single threaded execution of BSW, and instruction reductions with respect to SVE 128.

	D3		D4		D5	
	total	reduction	total	reduction	total	reduction
SVE 128	197	-	230	-	298	-
SVE 256	151	30.5%	153	50.3%	192	55.2%
SVE 512	124	58.9%	108	113.0%	131	127.5%

8.2 SVE Vector Length Analysis

In order to test the vector length scalability of our SVE port, we perform experiments with additional vector lengths of 128 and 256 bits. While the A64FX supports up to 512 bit vectors, the hardware can be instructed to use shorter vectors if desired. We execute the same binary for all the vector length, since SVE is a *vector length agnostic* ISA. As expected, we observe that the SMEM and SAL regions are not affected by vector length, as their code is not vectorized. Therefore, we only show the speed-ups for the BSW region. Figure 9 shows the performance improvements when changing the vector length for the 3 inputs on 48-thread runs, normalized to single-thread SVE 128 bits. We also observe average speed-ups of 37.6% when going from 128 to 256, and 79.0% when going from 128 to 512 SVE bits. These results correlate with the reduction in terms of committed instructions shown in Table 9. This indicates that the performance improvements we obtain in terms of SVE scaling are the expected ones, and that memory bandwidth is not limiting the performance of this kernel.

8.3 Scalability Analysis

Figure 10 shows average performance scaling with 12, 24 and 48 threads for the different regions as well as the entire execution. All the regions attain good scalability for all thread counts, reaching 44.53×, 44.31× and 43.08× for SMEM, SAL and BSW, respectively, with 48 threads. These numbers are close to the theoretical peak (48×), proving that the bandwidth is not limiting performance.

8.4 Comparison with SKX

In this section we compare performance and energy-to-solution of the A64FX against the SKX system, both described in Table 8.

8.4.1 Performance Comparison

Figure 11 shows performance results for the A64FX and the SKX system for the three regions as well as the entire execution using 24 and 48 threads.

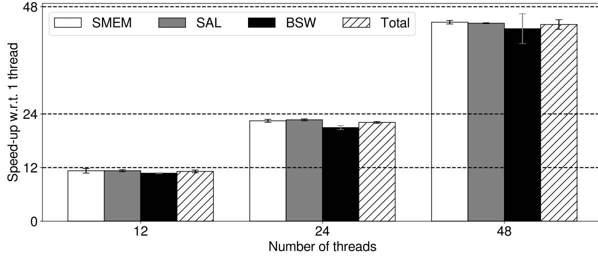


Fig. 10: Per region and total *average* speed-ups with respect to one thread for different thread counts using inputs D3, D4, and D5.

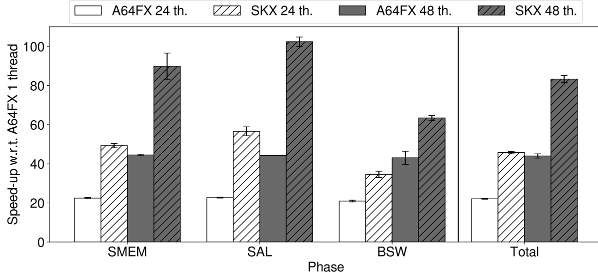


Fig. 11: Average speed-ups with respect to one thread in A64FX for each region and for the entire execution using 24 and 48 threads using D3, D4, and D5 inputs.

For the SMEM region, SKX clearly outperforms the A64FX on a performance per thread (core) basis, 2.01× with 48 threads. This is due to two factors: (i) the aggressive out-of-order pipeline of the SKX is much more effective at hiding long latency misses, and (ii) the memory access latency of the SKX system is significantly lower (see Figure 2). If we compare on a socket-to-socket basis, i.e., 24 threads of SKX versus 48 threads of A64FX, the results are more on par with a 10.6% advantage for SKX on average.

The SAL region presents similar results to SMEM. The workload characteristics are similar and the performance per thread difference in this region for 48 thread runs is of 2.31× on average. In a socket-to-socket basis the SKX systems outperform the A64FX by 27.7% on average.

The BSW region is much more compute intensive. On a per thread performance basis the SKX system still outperforms the A64FX by 1.47× on average for 48 threads. In this instance the main factors are: (i) the SKX core has more out-of-order resources (see Table 1), and (ii) the memory hierarchy of the SKX system has significantly more cache capacity, with 1 MB of private L2 per core (no private L2 in A64FX) and 33 MB of LLC per socket (32 MB of LLC on A64FX). However, when comparing on a socket-to-socket basis the A64FX outperforms the SKX system by 24.2%.

When adding up the three regions together, SKX outperforms the A64FX by 1.89× on average for 48 threads. If we compare on a socket-to-socket basis, the performance gap drops to a 4.0% advantage for SKX on average.

8.4.2 *Energy-to-Solution Comparison*

Figure 12 shows energy-to-solution for different thread counts on both systems for the entire region of interest; which includes SMEM, SAL and BSW. We can observe that SKX is more energy efficient, 71.1% and 38.9% on average than the A64FX for 1 and 12 threads, respectively. However, for 24 threaded executions the A64FX starts to gain efficiency with respect to the SKX system, and the advantage is reduced significantly to 15.0%. Note that we have to consider that for 24 threads the SKX system has the advantage of only using a single socket, as the other one is in idle state; while the A64FX chip is using half its cores. With 48 threads the SKX system fails to scale well in terms of energy-to-solution (13.2% of improvement over 24 threads) since it has to use the second socket, which increases the power consumption significantly. Therefore, performance gains are offset by the increase in power consumption. However, the A64FX system continues to scale well in terms of energy-to-solution and beats the SKX system by 11.6% on average. If we compare in a socket-to-socket basis the A64FX also has better energy-to-solution by 26.4% on average.

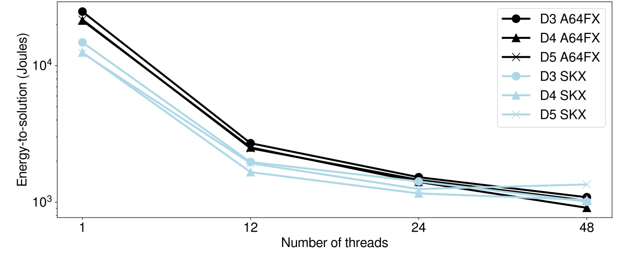


Fig. 12: Energy-to-solution comparison of the A64FX and SKX systems for the entire region of interest using D3, D4 and D5 inputs.

8.5 Discussion

In summary, BWA-MEM2 is a memory latency-bound application that benefits from aggressive out-of-order cores and lower memory access latencies present in the SKX system. In addition, this application is not able to exploit the main advantage of the A64FX, its memory bandwidth. While SKX has a substantial performance advantage on the SMEM and SAL code regions, and also a moderate advantage in performance on BSW; in terms of energy-to-solution, the A64FX system outperforms SKX both at equal thread count and when comparing socket-to-socket. SKX's aggressive out-of-order execution and large caches have an energy cost that is difficult to amortize via performance gains. Overall, the A64FX provides moderate per core performance but a good balance when considering energy footprint, which leads to better energy-to-solution on an application that is not the best fit for the A64FX architecture.

8.5.1 *Lessons Learned*

From the application side, we have found that working with the BSW code is tedious and error prone. For this kernel, there is one implementation for each vector ISA (i.e., SSE2, AVX2 and AVX512BW) as it is based on intrinsics. Since each vector ISA has different characteristics (e.g., support for masks) the implementations of the algorithm is also different. While the use of intrinsics may lead to performance gains, the drawbacks in terms of code maintenance, readability, and extensibility to new architectures might offset the benefits. SVE's vector length agnosticism is a step in the right direction, as any machine regardless of the implemented vector length can execute the code.

From the A64FX architecture side, the main take away is that it requires programmers and users to be aware of its memory capacity and NUMA characteristics, which expose certain trade-offs. First, the 32 GB of main memory can

be limiting in some scenarios. Second, efficiently use the available bandwidth requires to perform accesses to the local NUMA node, as going to other nodes leads to worse bandwidth utilization (see Table 4) and higher latency (see Figure 2). Therefore, careful placement of processes, threads and data allocation to minimize accesses to different NUMA nodes is a paramount.

9 Conclusions

The rapid growth of Arm-based server solutions and the stringent computing needs of genomics workloads demands that widely used genomics applications become ready to execute on ARMv8-A platforms and take advantage of technologies like SVE. In this paper, we have successfully ported a well-known genomics application, BWA-MEM2. Our porting effort enables the use of BWA-MEM2 on any ARMv8-A system that supports SVE. In addition, we have proposed several performance optimizations, some of which target our final evaluation platform, the Fujitsu A64FX processor. We evaluate the optimized version of the port on the A64FX and show almost linear thread-level and the expected data-level (SVE) parallelism.

Finally, we compare the A64FX system with an established SKX system and show that the SKX system performs better: 1.89× on average for 48 threads and a 4.0% when comparing on a socket-to-socket basis. However, in terms of energy-to-solution, the A64FX presents better results, both when comparing executions with the same thread count using 48 threads, by 11.6%, and when comparing socket-to-socket executions, by 26.4%. We conclude that the strength of A64FX is to be an energy efficient CPU with higher memory bandwidth than traditional HPC systems. In contrast, the A64FX has a higher memory access latency than other HPC systems, which negatively affects its performance when executing applications with random memory access patterns.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (contracts PID2019-107255GB-C21 / AEI /10.13039/501100011033 and PID2019-105660RB-C21 / AEI / 10.13039/501100011033), Gobierno de Aragón (T5820R research group), the Generalitat de Catalunya (contracts 2017-SGR-1328 and 2017-SGR-1414), and the European Union's Horizon 2020 research and innovation program (Mont-Blanc 2020 project, grant agreement 779877). Finally, A. Armejach and M. Moretó have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Juan de la Cierva fellowship no. IJCI-2017-33945 and Ramon y Cajal fellowship no. RYC-2016-21104, respectively.

References

[1] I. R. Koenig, O. Fuchs, G. Hansen, E. von Mutius, and M. V. Kopp, "What is precision medicine?" *European respiratory journal*, vol. 50, no. 4, 2017.

[2] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.

[3] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in Bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[5] E. Ukkonen, "Finding approximate patterns in strings," *Journal of algorithms*, vol. 6, no. 1, pp. 132–137, 1985.

[6] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[7] T. Robinson, J. Harkin, and P. Shukla, "Hardware acceleration of genomics data analysis: challenges and opportunities," *Bioinformatics*, pp. 1–11, 2021.

[8] "Ampere altra, the world's first cloud native processor," https://amperecomputing.com/altra/, accessed: 2021-07-12.

[9] "Amazon ec2 c6g instances," https://aws.amazon.com/ec2/instance-types/c6/, accessed: 2021-07-12.

[10] T. Yoshida, "Fujitsu high performance cpu for the post-k computer," 2018.

[11] "Top500 the list," https://www.top500.org/, accessed: 2021-07-12.

[12] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[13] ——, "Fast and accurate long-read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[14] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 314–324.

[15] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[16] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009.

[17] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," *Nature methods*, vol. 9, no. 12, p. 1185, 2012.

[18] "Gnu diffutils - comparing and merging files," https://www.gnu.org/software/diffutils/, accessed: 2021-07-12.

[19] A. Wang *et al.*, "An industrial strength audio search algorithm." in *Ismir*, vol. 2003. Citeseer, 2003, pp. 7–13.

[20] J. F. Crow, "Unequal by nature: A geneticist's perspective on human differences," *Daedalus*, vol. 131, no. 1, pp. 81–88, 2002.

[21] J. M. Herruzo, S. G. Navarro, P. Ibáñez, V. Viñals-Yufera, J. Alastruey, and O. Plata, "Accelerating sequence alignments based on fm-index using the intel knl processor," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2018.

[22] R. Langarita, A. Armejach, J. Setoain, P. Ibáñez, J. Alastruey-Benedé, and M. M. Planas, "Compressed sparse fm-index: Fast sequence alignment using large k-steps," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2020.

[23] M. S. Waterman, T. F. Smith, and W. A. Beyer, "Some biological sequence metrics," *Advances in Mathematics*, vol. 20, no. 3, pp. 367–387, 1976.

[24] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.

[25] S. Marco-Sola, J. C. Moure López, M. Moreto Planas, and A. Espinosa Morales, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, no. btaa777, pp. 1–8, 2020.

[26] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash," in *BMC genomics*, vol. 14, no. 1. BioMed Central, 2013, p. S13.

[27] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 2017.

[28] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: a fast and efficient pre-alignment filter for sequence alignment," *Bioinformatics*, 2019.

[29] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Bioinformatics*, vol. 13, no. 2, pp. 145–150, 1997.

[30] T. Rognes and E. Seeberg, "Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.

[31] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2006.
[32] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome biology*, vol. 10, no. 3, p. R25, 2009.
[33] A. D. Smith, W.-Y. Chung, E. Hodges, J. Kendall, G. Hannon, J. Hicks, Z. Xuan, and M. Q. Zhang, "Updates to the rmap short-read mapping software," *Bioinformatics*, vol. 25, no. 21, pp. 2841–2842, 2009.
[34] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.
[35] H. Suzuki and M. Kasahara, "Introducing difference recurrence relations for faster semi-global alignment of long sequences," *BMC Bioinformatics*, vol. 19, no. 1, pp. 33–47, 2018.
[36] S. Kalikar, C. Jain, M. Vasimuddin, and S. Misra, "Accelerating minimap2 for long-read sequencing applications on modern cpus," *Nature Computational Science*, vol. 2, no. 2, pp. 78–83, 2022.
[37] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 199–213.
[38] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 161–168.
[39] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand *et al.*, "Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020.
[40] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. Kim, N. M. Ghiasi *et al.*, "Segram: a universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," *arXiv preprint arXiv:2205.05883*, 2022.
[41] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 127–135.
[42] S. Diab, A. Nassereldine, M. Alser, J. Gómez-Luna, O. Mutlu, and I. E. Hajj, "A framework for high-throughput sequence alignment using real processing-in-memory systems," *arXiv preprint arXiv:2208.01243*, 2022.
[43] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.
[44] A. Armejach, H. Caminal, J. M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Stencil codes on a vector length agnostic architecture," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–12.
[45] A. Armejach, H. Caminal, J. M. Cebrian, R. Langarita, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Using arm's scalable vector extension on stencil codes," *The Journal of Supercomputing*, vol. 76, no. 3, pp. 2039–2062, 2020.
[46] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.
[47] W. Muła, N. Kurz, and D. Lemire, "Faster population counts using avx2 instructions," *The Computer Journal*, vol. 61, no. 1, pp. 111–120, 2018.
[48] T. Odajima, Y. Kodama, M. Tsuji, M. Matsuda, Y. Maruyama, and M. Sato, "Preliminary performance evaluation of the fujitsu a64fx using hpc applications," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 523–530.
[49] "Srx020470," https://www.ncbi.nlm.nih.gov/sra/SRX020470, accessed: 2022-11-10.
[50] "Srx207170," https://www.ncbi.nlm.nih.gov/sra/SRX207170, accessed: 2022-11-10.
[51] "Srx206890," https://www.ncbi.nlm.nih.gov/sra/SRX206890, accessed: 2022-11-10.
[52] "Genome reference consortium human reference 38," http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/, accessed: 2019-07-23.
[53] "Slurm workload manager," https://slurm.schedmd.com/, accessed: 2021-07-12.
[54] "Powerapi: Sandia national laboratories," https://powerapi.sandia.gov/, accessed: 2021-07-12.
[55] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

Rubén Langarita received his BS degree in computer science from Universidad de Zaragoza in 2018. He spent one academic year as an Erasmus student at the University College Cork. His final degree project was about optimizing molecular dynamics applications. He received MS degree from UPC in January 2021. He is currently working at the BSC as a research student. His research interests include processor microarchitecture and HPC applications.

Adrià Armejach is an associate researcher at the Barcelona Supercomputing Center and a researcher at Universitat Politècnica de Catalunya. His research interests include computer architecture, parallel computing, memory systems, and performance evaluation. He received M.Sc. and Ph.D. degrees from UPC in 2009 and 2014, respectively. He has lead the technical contributions of multiple FP7 and H2020 projects.

Pablo Ibáñez is an associate professor in the Computer Science and Systems Engineering Department at the Universidad de Zaragoza. He received his MS and PhD degrees both in computer science from the UPC in 1989 and from Universidad de Zaragoza in 1998 respectively. His research interests include HPC applications, memory hierarchy, and processor microarchitecture.

Jesús Alastruey-Benedé received the MS degree in Telecommunication and the PhD degree in Computer Science from Universidad de Zaragoza in 1997 and 2009, respectively. He is an associate professor in the Computer Science and Systems Engineering Department (DIIS), Universidad de Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and HPC applications.

Miquel Moretó is a Ramon y Cajal Researcher at the Universitat Politècnica de Catalunya (UPC). Prior to joining UPC, he spent 5 years as a senior researcher at the Barcelona Supercomputing Center and 1.5 years as a post-doctoral fellow at the International Computer Science Institute (ICSI), Berkeley. In 2010, he received the PhD degree from UPC. His research interests include high performance domain-specific architectures and hardware-software co-design for massively parallel systems.