Future Generation Computer Systems 157 (2024) 313–329

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

GenArchBench: A genomics benchmark suite for arm HPC processors

Lorién López-Villellas [d,*,1], Rubén Langarita-Benítez [a], Asaf Badouh [a], Víctor Soria-Pardos [a], Quim Aguado-Puig [b], Guillem López-Paradís [a], Max Doblas [a], Javier Setoain [e], Chulho Kim [f], Makoto Ono [g], Adrià Armejach [a,c], Santiago Marco-Sola [a,c], Jesús Alastruey-Benedé [d], Pablo Ibáñez [d], Miquel Moretó [a,c]

[a] Barcelona Supercomputing Center, Barcelona, Spain
[b] Department d'Arquitectura de Computadors, Universitat Autònoma de Barcelona, Barcelona, Spain
[c] Department d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain
[d] Departamento de Informática e Ingeniería de Sistemas/Aragón Institute for Engineering Research (I3A), Universidad de Zaragoza, Zaragoza, Spain
[e] Arm Research, Cambridge, United Kingdom
[f] Lenovo Research, United States
[g] Lenovo Infrastructure Solutions Group, United States

ARTICLE INFO

Keywords:
Genomics
Arm
High-performance computing
Parallel computing
Vector computing
Performance characterization

ABSTRACT

Arm usage has substantially grown in the High-Performance Computing (HPC) community. Japanese supercomputer Fugaku, powered by Arm-based A64FX processors, held the top position on the Top500 list between June 2020 and June 2022, currently sitting in the fourth position. The recently released 7th generation of Amazon EC2 instances for compute-intensive workloads (C7 g) is also powered by Arm Graviton3 processors. Projects like European Mont-Blanc and U.S. DOE/NNSA Astra are further examples of Arm irruption in HPC. In parallel, over the last decade, the rapid improvement of genomic sequencing technologies and the exponential growth of sequencing data has placed a significant bottleneck on the computational side. While most genomics applications have been thoroughly tested and optimized for x86 systems, just a few are prepared to perform efficiently on Arm machines. Moreover, these applications do not exploit the newly introduced Scalable Vector Extensions (SVE).

This paper presents GenArchBench, the first genome analysis benchmark suite targeting Arm architectures. We have selected computationally demanding kernels from the most widely used tools in genome data analysis and ported them to Arm-based A64FX and Graviton3 processors. Overall, the GenArch benchmark suite comprises 13 multi-core kernels from critical stages of widely-used genome analysis pipelines, including base-calling, read mapping, variant calling, and genome assembly. Our benchmark suite includes different input data sets per kernel (small and large), each with a corresponding regression test to verify the correctness of each execution automatically. Moreover, the porting features the usage of the novel Arm SVE instructions, algorithmic and code optimizations, and the exploitation of Arm-optimized libraries. We present the optimizations implemented in each kernel and a detailed performance evaluation and comparison of their performance on four different HPC machines (i.e., A64FX, Graviton3, Intel Xeon Skylake Platinum, and AMD EPYC Rome). Overall, the experimental evaluation shows that Graviton3 outperforms other machines on average. Moreover, we observed that the performance of the A64FX is significantly constrained by its small memory hierarchy and latencies. Additionally, as proof of concept, we study the performance of a production-ready tool that exploits two of the ported and optimized genomic kernels.

1. Introduction

For many years, Arm processors have dominated the mobile device segment. Their energy efficiency and license-based business model have been the pillars underpinning this success.

In recent years, Arm has burst onto the high-performance computing market with influential companies and consortiums that have become licensees, such as Fujitsu, Amazon, Apple, NVIDIA, Samsung, AMD, Broadcom, HUAWEI, and Qualcomm. Currently, the Arm-based Fujitsu

* Correspondence to: Departamento de Informática e Ingeniería de Sistemas/Aragón Institute for Engineering Research (I3A), Universidad de Zaragoza, Spain.
E-mail address: lorien.lopez@unizar.es (L. López-Villellas).
[1] The corresponding author conducted this work while affiliated with the Barcelona Supercomputing Center.

https://doi.org/10.1016/j.future.2024.03.050
Received 3 November 2023; Received in revised form 25 March 2024; Accepted 31 March 2024
Available online 2 April 2024

A64FX processor powers the Japanese supercomputer Fugaku, which held the top position on the Top500 list between June 2020 and June 2022 and is currently in the fourth position. Moreover, Amazon has been using Arm processors to power its cloud computing platform (AWS), starting in 2018 with the Graviton processor. They followed with the second generation of Graviton in 2019 and the recently released Graviton3.

In the near future, NVIDIA Grace CPUs and Ampere servers will be leading further efforts to breakthrough Arm in HPC. As a result, large-scale computing infrastructures, usually equipped with x86 and IBM Power processors, now have an additional competitive alternative. However, most of the scientific code for HPC is not fully adapted and optimized for Arm architectures.

Over the last decade, genome sequencing has become the cornerstone of genomics and modern precision medicine. Due to the rapid improvement of sequencing technologies, it is currently possible to sequence an individual's genome in less than 24 h. This breakthrough has enabled effective personalized healthcare, allowing the diagnosis and treatment of diseases based on each person's unique genomic disposition [1]. Furthermore, genome sequencing has also been proven crucial in cancer studies [2], drug development [3], or COVID-19 outbreak control [4]. In the past 20 years, genome sequencing costs have dropped dramatically and the amount of sequencing data produced yearly has increased exponentially. More notably, this increase in data production has outperformed the pace of Moore's law. As a result, a significant bottleneck in current genome sequencing analysis is placed on the computational side, executing computational-intensive genomics tools and pipelines.

Genome analysis pipelines have historically been designed to run efficiently on x86 architectures. With the irruption of Arm-based HPC servers, adapting and optimizing genomics tools to exploit HPC Arm architectures effectively has become paramount. For that, we have selected 13 computationally-demanding CPU kernels from the most widely-used genomics tools, and we have included them in a benchmark suite called GenArchBench. All the kernels exploit multi-core parallelism and implement common stages from widely-used genome analysis pipelines such as base-calling, read mapping, variant calling, and de-novo assembly. Additionally, GenArchBench includes input datasets for each kernel (i.e., a small dataset and a large dataset per kernel) and their corresponding outputs to be used as ground truth. The small datasets have been sized to require single-thread execution times no longer than a few minutes (for testing purposes); meanwhile, large datasets require several minutes (for performance evaluation purposes). For convenience, we provide automatic regression tests for all the kernels to verify the correctness of the outputs.

Furthermore, this work introduces code adaptations and optimizations of the genomics kernels targeting Arm HPC CPUs. GenArchBench leverages Arm-specific HPC libraries (carefully optimized for Arm processors) and presents algorithmic and code optimizations to exploit the architecture and resources of Arm HPC machines. Notably, we have optimized some kernels by utilizing the latest Arm Scalable Vector Extensions (SVE) to leverage the potential of the latest Arm HPC processors.

In addition to the benchmark suite porting and optimization, this work presents a performance characterization of GenArchBench on four HPC machines (two Arm-based and two x86-based nodes). The experimental evaluation compares the performance of an A64FX processor, a Graviton3 processor, an Intel Xeon Skylake Platinum 8160 processor, and an AMD EPYC 7742 Rome processor. This characterization includes the kernels' instruction breakdown, single-thread and multi-thread performance evaluations, a microarchitecture bottleneck analysis, and an energy-to-solution study in the different processors. Ultimately, we evaluate the performance impact of these optimizations by integrating two of the accelerated kernels in a production-ready tool used in a myriad of genome analysis pipelines.

In summary, this work makes the following contributions:

- We present GenArchBench, the first benchmark suite targeting Arm HPC architectures for genome analysis pipelines and tools. The benchmark suite is publicly available at https://github.com/LorienLV/genarchbench/releases/tag/1.0.0.
- We propose HPC adaptations and code optimizations applied to GenArchBench's kernels to exploit the potential of Arm HPC processors, leveraging Arm-specific HPC libraries and Arm Scalable Vector Extension (SVE).
- We perform a comprehensive performance characterization of GenArchBench in two HPC Arm processors (i.e., A64FX and Graviton3). We compare the performance of Arm against two reference HPC x86 machines.

2. Background

Genome data analysis pipelines comprise multiple stages and computational tools, from sequencing biological samples to deriving meaningful data analysis results for scientists and healthcare professionals. This section introduces the main sequencing technologies, pipelines, and tools used in common genome analysis (Fig. 1 shows a succinct graphic summary).

2.1. Sequencing technologies

Before any computational analysis can be performed, biological DNA samples must be converted to digital data. This process is performed by the sequencing machines (Fig. 1-1), and, despite the remarkable advances in the last decades, these machines are still unable to read a complete DNA molecule from end to end. Instead, sequencing machines allow reading relatively small chunks of DNA, called reads or fragments, from random locations within the donor's DNA genome. Afterwards, sequenced reads must be jigsaw together to reconstruct or reassemble the original donor's genome.

Sequencing machines are commonly categorized into three generations based on their technological advancements. The first sequencing technologies (Sanger et al. [5] and Maxam et al. [6]) were developed in 1977 and used to sequence the first draft of the human genome in 2000 [7]. Since then, sequencing technologies have evolved quickly, simplifying the sequencing process and increasing the data-production throughput. In the mid-2000s, second-generation technologies [8] were introduced and soon replaced first-generation technologies. Second-generation technology can generate fixed-length sequences of 100–300 bps at a throughput of tens of gigabytes per hour and with a low reading error rate (0.1% of the read length). At present, Illumina dominates the market of second-generation sequencing machines. Recently introduced third-generation technologies, known as long-read sequencing, can read variable-length sequences of considerable length (i.e., tens of kilo base-pairs) at the expense of lower production throughput (less than 10 Gb/hour) and higher reading error rate (0.1%–10% of the read length). Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT) are the most notable manufacturers of third-generation sequencing technologies.

2.2. Genome data analysis pipelines and tools

Before any processing can be performed, the sequencing machines' raw signals must be transformed into sequences of nucleotides (A, C, G, T). This process is called basecalling (Fig. 1-2). Typically, a specialized basecalling tool is used to perform this process tailored to each sequencing technology. For instance, Bonito [9] and Guppy [10] are two of the most widely-used tools for basecalling Oxford Nanopore's raw-signal output.

Once the sequences of nucleotides are decoded, sequenced reads must be processed and analyzed to derive meaningful biological insights. Although many different genome analyses can be performed using sequenced data, most analyses begin with either genome resequencing (1-3.a) or genome assembly (1-3.b). Both analyses seek to reconstruct the sample's genome by putting together all the sequenced reads.

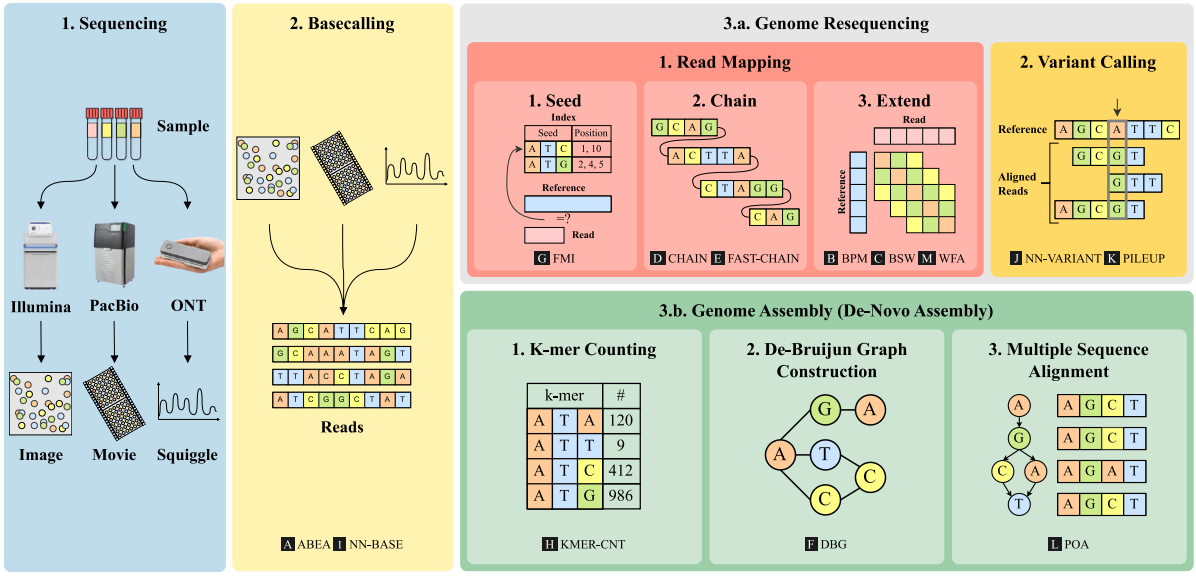

Fig. 1. Workflow diagram of common genome analysis pipelines. Going from (1) sequencing, through (2) basecalling, to (3.a) genome resequencing or (3.b) and genome assembly. The figure shows the different computational kernels used within each stage or tool.

2.2.1. Genome resequencing

The most common approach to reconstructing the sample's genome is by resequencing and involves reconstructing the sample's genome using a previously known reference genome. For that, each sequenced read is located and matched to the most likely originating position in the reference genome, allowing small differences (e.g., mismatches, insertions, and deletions). This processes is called read mapping (Fig. 1-3.a.1) and it is implemented by many tools like BWA-MEM2 [11,12], Minimap2 [13], Bowtie2 [14,15], and GEM [16]. Read mapping is one of the most computationally expensive steps in all genome sequence analyses. Consequently, read mapping has been extensively studied and optimized.

Most sequence mappers are based on the seed-chain-extend technique. This technique implements three algorithmic steps to swiftly locate and align a sequence with a reference genome. During the first step, known as seeding (Fig. 1-3.a.1.1), the mapper searches small subsequences of the reads (seeds) in the reference leveraging an index structure. The most widely-used indexes used for seeding are FM-Index [17] and hash-tables [18,19]. Seeding reduces the potential number of locations in the reference where a sequence can match, decreasing the amount of work performed in subsequent steps. Afterwards, a chaining step (Fig. 1-3.a.1.2) is performed to reduce further the list of possible matching locations in the reference. During the chaining step, all the mapped seeds are processed to find a colinear chain of seeds that can potentially match the input sequence. Finally, during the extensión or alignment step (Fig. 1-3.a.1.3), the input sequence is aligned against the candidate location in the reference genome, discovering the differences between the donor's sequence and the reference genome. Usually, a dynamic programming-based algorithm, such as Needleman–Wunsch [20] or Smith–Waterman–Gotoh [21,22], is used to compute the alignment.

After sequence mapping, once the reads are located in the reference genome, a variant calling algorithm (Fig. 1-3.a.2) determines the variants and mutations between the donor's genome and the reference genome. These variations provide crucial insights into the genetic makeup of the sequenced individual, potentially revealing genetic variations that may be associated with diseases and health conditions. Notable examples of widely-used variant callers are GATK Haplotype-Caller [23], Platypus [24], Clair [25,26], DeepVariant [27] and Medaka [28].

2.2.2. Genome assembly

Despite the simplicity and effectiveness of genome resequencing, there is still a lack of high-quality reference genomes for many species. In those situations, genome de-novo assembly (Fig. 1-3.b) is used to reconstruct the donor's genome from scratch jigsawing the sequenced reads together.

Most popular de-novo assembly methods rely on de Bruijn graphs. For a given set of sequences, its corresponding de Bruijn graph contains a node per each sequence's k-mer (i.e., sub-string of length k nucleotides) and an edge that connects adjacent and overlapping k-mers. Before constructing the de Bruijn graph of a set of input sequences, the number of unique k-mers in the reads is counted (Fig. 1-3.b.1) to prune the least frequent ones (likely artifacts of the sequencing process). Afterwards, the de Bruijn graph is constructed (Fig. 1-3.b.2). Then, the consensus sequence is derived using multiple sequence alignment (MSA) algorithms (Fig. 1-3) and the constructed de Bruijn graph. Notable examples of de Bruijn graph based assemblers are Flye [29], Canu [30], and Racon [31].

2.2.3. Metagenomics

Beyond genome resequencing, variant calling, and de-novo assembly, many previously described analysis steps and tools can be found in other genome analysis pipelines. This is the case for many metagenomics analysis pipelines. Metagenomics pipelines seek to analyze genomic information from mixed microbial communities, providing insights into the diversity, interactions and function of microorganisms present in an environmental sample. Metagenomics analyses are performed using tools such as Centrifuge [32], RawMap [33], UNCALLED [34], ReadFish [35], Kraken2 [36] and Clark [37]. These tools employ k-mer counting (Fig. 1-3.b.1) and seeding techniques (Fig. 1-3.a.1.1) for their analysis. Moreover, variant callers like GATK Haplotype Caller [23] and Platypus [24] are used to construct De Bruijn graphs (Fig. 1-3.b.2) and correct artifacts produced during the mapping process (Fig. 1-3.a.1). Furthermore, the chaining process (Fig. 1-3.a.1.2) is also utilized for genome assembly when using alternative approaches based on de Bruijn graphs [38].

3. GenArch benchmark suite

The GenArch benchmark suite comprises 13 multithreaded CPU kernels derived from the most widely used genomics tools and covers

the most important genome sequencing steps. It includes ten kernels from the GenomicsBench [39] benchmark suite and three additional kernels: the Bit-Parallel Myers algorithm [40] (BPM), the Wavefront Alignment algorithm [41] (WFA), and FAST-CHAIN [42]. BPM and WFA complement the sequence alignment kernels of GenomicsBench to better capture contemporary trends. Additionally, FAST-CHAIN [42] is a recent vector-enabled reimplementation of the CHAIN kernel present in GenomicsBench, which allows us to further explore the capabilities of SVE.

Additionally, GenArchBench includes input datasets for each kernel (i.e., a small dataset and a large dataset per kernel) and their corresponding outputs to be used as ground truth. The small datasets have been sized to require single-thread execution times no longer than a few minutes (for testing purposes); meanwhile, large datasets require several minutes (for performance evaluation purposes). For convenience, we provide automatic regression tests for all the kernels to verify the correctness of the outputs.

Although some kernels included in GenArchBench can exploit the capabilities of modern GPUs, this research focuses on porting, accelerating, and evaluating the performance of genomics kernels in Arm processors. Moreover, the Arm-systems evaluated in this work (A64FX and Graviton3) are not equipped with GPUs.

The following text presents GenArchBench's kernels, briefly describing its functionality, which tools use them, and a description of their usage and inputs.

Adaptive Banded Signal to Event Alignment (ABEA): ABEA is a dynamic programming algorithm that compares raw nanopore signals from ONT sequencing machines to a reference genome sequence. ABEA's implementation is based on the Suzuki–Kasahara (SK) [43] algorithm. This step is performed in some tools, such as Nanopolish [44], to correct errors produced in the basecalling process (Fig. 1-2). For GenArchBench, we have used the CPU implementation of f5c [45], a version of ABEA based on Nanopolish's, optimized for both CPU-only and hybrid CPU/GPU executions. This implementation of ABEA exploits coarse-grain multi-threading by dividing the raw signals of the input between the available cores. Since the signals are not of regular size, f5c implements work-stealing to improve load balance. The small and large inputs comprise 1K and 10K raw FAST5 (ONT) reads from chromosome 22 of NA12878 and GRCh38 as the reference genome [46].

Bit-Parallel Myers (BPM): BPM [40] is a dynamic programming algorithm that finds all locations a query string of size m matches a reference string of size n with k or fewer differences (Fig. 1-3.a.1.3). It computes the approximate string matching of two strings in $O(mn/w)$ time, where w is the word size of the machine. BPM is used in read mapping tools, such as GEM-Mapper [16], Edlib [47], GraphAligner [48] or Hobbes [49]. For GenArchBench, we have used an in-house implementation of the algorithm that exploits multi-threading by assigning different pairs of strings to different threads. The small and large inputs comprise 100K and 10M sequence pairs from human sample SRR7733443 downloaded from the sequence read archive [50].

Banded Smith–Waterman (BSW): The Smith–Waterman algorithm [21] is a dynamic programming algorithm that computes the local sequence alignment of two sequences of length m and n, respectively, in $O(mn)$ time and space. A banded version of Smith–Waterman [51] is used to align sequences with a maximum of w insertions/deletions, reducing the time and space complexity to $O(wn)$ (Fig. 1-3.a.1.3). BSW is used in variant discovery tools such as GATK [23], and in sequence alignment software like BWA-MEM [11,12]. For GenArchBench, we have used BWA-MEM2's x86-vectorized implementation of BSW. In order to exploit multi-threading, the set of pairs of strings to align is dynamically divided between processors. The small and large inputs comprise 100K and 10M sequence pairs from human sample SRR7733443 [50].

Seed Chaining (CHAIN): Given the set of seeds from a DNA sequence (read) mapped to another sequence, such as the reference genome, the chaining step (Fig. 1-3.a.1.2) aims to find a chain of colinear seeds. This is a time-consuming step performed by alignment tools, such as Minimap2, and by de-novo assemblers like Flye [29] or Canu [30]. We have used the implementation of CHAIN found in GenomicsBench that extends Minimap2's to exploit inter-task parallelism across reads. The small and large inputs comprise the seeds from 1K, and 10K reads of Pacbio's *Caenorhabditis elegans* worm sequence data [52].

SIMD Seed Chaining (FAST-CHAIN): The previously presented implementation of the CHAIN algorithm utilizes heuristics to stop executing when the result is sufficiently good. This speedups execution at the cost of accuracy, and it hinders the vectorization of the kernel. FAST-CHAIN [42] is an x86-vectorized version of CHAIN that removes the heuristics to exploit SIMD computation. As a result, FAST-CHAIN outputs accurate results and presents performance gains compared to CHAIN. FAST-CHAIN uses the same inputs as CHAIN.

De Bruijn Graph Construction (DBG): The De Bruijn graph (DBG) of an input set of reads is used to represent the overlaps between the sub-strings of length k (k-mers) found in the input (Fig. 1-3.b.2). Each node of the graph represents a k-mer and the edges connect adjacent k-mers in the input set. The construction of these graphs is a time-consuming step in de-novo assemblers like Flye [29], Canu [30] or Racon [31], and in variant callers such as GATK [23] and Platypus [24]. For GenArchBench, we have used the DBG construction of Platypus, which exploits parallelism by assigning different regions of the input to different threads. Both inputs employ chromosome 22 of BWA-MEM aligned records from the Platinum Genomes dataset [53]. The small input uses bases 16M-16.5M, while the large input uses the entire chromosome.

FM-Index Search (FMI): The FM-index is a compressed sub-string index based on the Burrows–Wheeler transform [54]. Given a sub-string s, FM-index can be used to find the location of s in the reference genome in $O(|s|)$ time, where $|s|$ is the length of the sub-string (Fig. 1-3.a.1.1). The FM-index data structure is used in sequence alignment tools such as BWA-MEM [11,12] or Bowtie2 [15], and in metagenomic classification software like Centrifuge [32]. For GenArchBench, we have used the super-maximal exact match kernel of BWA-MEM2, which utilizes the FM-Index structure. This kernel exploits parallelism by dynamically assigning batches of reads among threads. The small and large inputs comprise 1M and 10M pairs of 151 bases from human sample SRR7733443 [50].

K-mer Counting (KMER-CNT): K-mer counting aims to count the number of occurrences of each k-mer in an input sequence (Fig. 1-3.b.1). This task is performed in de-novo assemblers such as Flye [29] or Canu [30] and in metagenomics classification software like Clark [37]. Additionally, note that the functionality of KMER-CNT is very similar to accessing large lookup tables, as done in state-of-the-art mappers like Minimap2 [13]. For GenArchBench, we have used the k-mer counting kernel of Flye. This implementation divides the input-reads between threads and relies on the thread-safe hash-map implementation of Libcuckoo library [55] to concurrently increase the number of individual k-mers shown by each thread. The small and large inputs comprise 1K and 50K *Escherichia coli* Oxford Nanopore reads sequenced by Loman Labs [56].

Neural Network-based Base Calling (NN-BASE): ONT sequencing machines monitor changes in an electrical current as single strands of DNA or RNA pass through a protein nanopore. These changes in the electrical current are then converted to a sequence of nucleotide bases in the basecalling process (Fig. 1-2). The analog signal inevitably contains ambiguities due to noise or measurement errors. Some basecallers, such as Guppy [10] and Bonito [9], rely on neural networks to solve these ambiguities, determining the most likely observed nucleotide in each part of the electrical current. For GenArchBench, we have used Bonito's deep-learning base-caller (NN-BASE), which depends on the PyTorch library [57]. Bonito splits the input signal into smaller chunks of regular size and feeds them to a PyTorch neural network that

Table 1
Characteristics overview of the experimental setup.

	A64FX	Graviton3	SKX	Rome
Cores	4 × 12 (+ 4 assistant)	64	2 × 24	64
SMT	No	No	Disabled	Disabled
Frequency	2.2 GHz (static)	2.6 GHz	1–2.1 GHz (dynamic)	1.5–2.25 GHz (dynamic)
Max. power	120 W	N/A	2 × 150 W	225 W
Mem. capacity	4 × 8 GB	8 × 16 GB	2 × 6 × 8 GB	16 × 64 GB
Mem. technology	on-package HBM2	off-package DDR5 4800 MHz	off-package DDR4 2667 MHz	off-package DDR4 3200 MHz
Peak bandwidth	4 × 256 GB/s	300 GB/s	2 × 120 GB/s	204.8 GB/s
L1i	64 KB (4-way)	64 KB	32 KB (8-way)	32 KB (8-way)
L1d	64 KB (4-way)	64 KB	32 KB (8-way)	32 KB (8-way)
L2	–	1 MB	1 MB (16-way)	512 KB (8-way)
LLC	4 × 8 MB (16-way)	32 MB	2 × 33 MB (11-way)	16 × 16 MB (16-way)
Vector extension	NEON/SVE 512 bits	NEON/SVE 256 bits	SSE/AVX2/AVX512	SSE/AVX2

internally exploits multi-threading. The small and large inputs comprise 1 and 10 raw FAST5 reads from chromosome 20 of NA12878, obtained from the Nanopore WGS Consortium [46].

Neural Network-based Variant Calling (NN-VARIANT): Variant calling is the process of detecting the differences (variants or mutations) between the aligned reads and the reference genome (Fig. 1-3.a.2). This is a costly process performed by statistics-based variant callers, such as GATK HaplotypeCaller [23] or Platypus [24], and deep-learning variant callers, such as Clair [25,26], DeepVariant [27] or Medaka [28]. For GenArchBench, we have used the second generation of Clair variant caller (Clair3), based on the TensorFlow framework [58]. Clair3 exploits parallelism by dividing the input into regular-size chunks, and each of these chunks is processed by one thread using TensorFlow. Our small and large inputs comprise 100K and 10M reference positions, respectively, of chromosome 20 of HG002 from NITS's Genome in a Bottle (GIAB) project [59]. We are using Clair3's ONT pre-trained model `r941_prom_hac_g360+g422` [60].

Pileup Counting (PILEUP): Given the alignment data of a set of aligned reads to a region of a reference genome, usually a SAM or BAM file [61], pileup counting is the process of summarizing the base-pair information at each chromosomal position. This summary, called pileup, is customary the input for long-read neural network variant callers such as Clair [25,26] or Medataka [28] (Fig. 1-3.a.2). For GenArchBench we have used the pileup counting implementation of Medaka, which exploits multi-thread parallelism by distributing 100 kilobase regions of the reference genome between threads. The small input comprises bases 1-1499707 of the *Staphylococcus aureus* genome [10], and the large input comprises bases 1-1412827 of chromosome 20 of sample HG002 [59].

Partial-Order Alignment (POA): The construction of an overlap graph from a set of reads leads to an approximate representation of the original sample's genome. To determine the consensus genome of the sample, the alignment of all the reads against each other is performed in a process called multiple sequence alignment (MSA) (Fig. 1-3.b.3). The Partial Ordered Alignment (POA) algorithm [62] computes the MSA of all sequences by incrementally constructing a partially-order graph aligning new sequences to it using a dynamic programming algorithm such as Smith–Waterman [21] or Needleman–Wunsch [20]. The multiple alignment sequence (consensus sequence) is inferred from the graph by using the Heaviest Bundle algorithm [63]. POA is used in software packages such as Nanopolish [44] or Racon [31]. For GenarchBench we have used the SIMD-optimized version of POA of the SPOA library [64]. SPOA exploits multi-threading by computing the partially-ordered graph of multiple sets of sequences in parallel. The small and large inputs comprise 1K and 6K sets of multiple sequences aligned to a reference genome, each containing between 5 and 115 sequences. This data comes from Minimap2's polishing step of the Flye-assembled *Staphylococcus Aureus* genome [10].

Wavefront Alignment (WFA): The wavefront alignment algorithm (WFA) [41] is a pairwise alignment algorithm (Fig. 1-3.a.1.3) that takes advantage of homologous regions between the sequences to accelerate the alignment process. As opposed to traditional dynamic programming algorithms that run in quadratic time, WFA time complexity is $O(ns)$, proportional to the read length n and the alignment score s, using $O(s^2)$ memory. The wavefront algorithm is used in tools such as wfmash [65], AnchorWave [66] or AncestralClust [67]. GenArchBench uses a custom multi-thread implementation of the algorithm, in which each thread works in the alignment of a pair of strings. The small and large inputs comprise 100K and 1M sequence pairs from human sample SRR7733443 [50].

4. Experimental setup

Our experimental setup consists of two Arm and two x86 HPC systems: a compute node featuring an Arm-A64FX processor (A64FX), a c7 g.16xlarge Amazon-EC2 instance (Graviton3), a system with two x86-64 Intel Xeon Skylake Platinum 8160 (SKX), and a compute node with one x86-64 AMD EPYC 7742 Rome processor (Rome). Table 1 presents an overview of the main characteristics of the four systems.

In terms of computing cores, the A64FX is based on four Non-Uniform Memory Access (NUMA) domains within the chip, also referred to as core memory groups (CMG). Each NUMA domain has 12 cores, plus one assistance core not used for general computing (running daemons, I/O, asynchronous MPI, etc.). In total, the A64FX implements 48 computing cores. Graviton3 implements 64 cores in a single NUMA domain. The AMD Rome CPU comprises 8 core chiplets, known as core cache dies (CCD), and a central I/O die that controls all the I/O and memory functions of the chip. A CCD has two core complex (CCX) clusters, each with 4 cores. Any pair of CCDs can communicate through the I/O die. SKX contains two NUMA chips, each with 24 physical cores.

Regarding operational frequency, Graviton3 presents the highest maximum frequency among the systems with 2.6 GHz. The other three systems' maximum frequency is very similar, ranging between 2.1 and 2.25 GHz. Both x86 systems dynamically adjust their frequency based on their load. Additionally, SKX reduces its frequency when executing AVX/AVX512 instructions. In contrast, the A64FX operates at a fixed frequency set to 2.2 GHz. There is no public information about adaptive frequency operation on Graviton3.

With respect to SIMD extensions, the A64FX is the first CPU to implement the Armv8.2-A Scalable Vector Extension (SVE) [68]. One of SVE's main features is that it is Vector Length Agnostic (VLA); that is, the same binary works on architectures implementing vector registers of different lengths ranging from 128 to 2048 bits. The A64FX implements 512 bits SVE registers. Graviton3 also implements SVE, with a vector length of 256 bits. The A64FX and Graviton3 also support the Arm Neon SIMD extension, a non-VLA SIMD ISA that works with 128-bit vectors. Both x86 systems implement the SSE and AVX2 SIMD extensions, with a vector length of 128 and 256 bits, respectively. The SKX also supports the AVX512 extension, with a vector length of 512 bits. None of the x86 SIMD extensions are VLA.

Concerning main memory, each A64FX's NUMA domain has its own local on-chip 8 GB HBM2 main memory and can access the other three NUMA domains' local memories via a ring bus. Graviton3 is connected to 8 × 16 GB DDR5 channels, for a total of 128 GB of memory. Each

Table 2
Load-to-use memory latencies in nanoseconds of the experimental setup.

	A64FX	Graviton3	SKX	Rome
L1	2.3–5	1.5	1.9	1.8
L2	–	4.6	6.7	3.5
LLC	16.8–21.4	33.1	25.1	13.0
Main Mem. Local	118.2–126.4	153.5	86.2	121.5
Main Mem. Remote	187.7–242.3	–	144.0	–

chip of the SKX is connected to 6 × 8 GB DDR4 local channels and can access the other chip's local memory. The Rome CPU is connected to 16 × 64 GB DDR4 channels, totaling 1 TB of memory.

The cache hierarchy organization of the processors is relatively different. Both Arm machines have two 64 KB private L1 caches per core (instructions and data), while the x86 CPUs feature two 32 KB private L1s per core. Graviton3 and SKX include one private 1MB L2 cache per core, and Rome has one 512 KB private L2 per core. The A64FX has one 8 MB last-level cache (LLC) per NUMA domain, Graviton3 includes one 32 MB LLC, SKX has two 33 MB LLCs (one per NUMA domain), and Rome includes one 16 MB LLC per each 4-core CCX.

Concerning memory bandwidth, the A64FX is designed to achieve good performance executing high memory bandwidth-demanding applications. The peak bandwidth of this chip (4 × 256 GB/s) is nearly 3.5 times higher than the peak bandwidth of Graviton3 (300 GB/s), the second system among the studied in terms of memory throughput. It is followed by SKX, reaching up to 120 GB/s per chip (240 GB/s in total), and Rome holds the last position with a peak bandwidth of 204.8 GB/s.

Table 2 presents the memory access latencies to each level of the memory hierarchy for all machines. All latencies on Rome and Graviton3 and latencies to remote memories on the A64FX have been measured using the LMbench benchmark [69]. Latencies to cache and local memory on the A64FX have been extracted from the micro-architecture manual of the CPU. Latencies on SKX have been measured using Intel Memory Latency Checker. The number of cycles to access the A64FX caches depends on the type of instruction: scalar, floating-point, short SIMD, and large SIMD. The latencies to access the L1 on the systems range from 1.5 ns (Graviton3) to 5 ns (large SIMD access on the A64FX). Even though scalar accesses on the A64FX are faster (2.3 ns), it still presents the highest L1 access latency. As presented previously, the A64FX only implements two levels of caches (L1 and LLC). The L2 access latencies of the other systems range between 3.5 ns (Rome) to 6.7 ns (SKX). Rome presents the fastest access to its LLC (13 ns), closely followed by the A64FX (16.8 ns for scalar access and 21.4 ns for large SIMD access). The LLC access latency on the SKX and Graviton3 is 25.1 and 33.1 ns, respectively. SKX presents the fastest access latency to local main memory (86.2 ns), followed by the A64FX and Rome, with similar latencies (~120 ns). Graviton3 has the highest local memory access latency, as expected from current DDR5 SDRAMs. Accessing remote main memories in the A64FX takes between 187.7 ns (near-remote memory) and 242.3 ns (far-remote memory). Accessing the other chip's main memory on the SKX machine takes 144 ns, 23% faster than A64FX's best case.

The out-of-order resources of the experimental setup are presented in Table 3. We assume that Graviton3 implements the same resources as Neoverse V1 for non-publicly available data (marked with *). Unavailable data for neither Graviton3 nor Neoverse V1 is represented as N/A. The A64FX is tight in out-of-order resources compared with the other three processors. The SKX and Rome have a similar number of physical registers, almost doubling the number of general-purpose registers of the A64FX (180 vs. 96) and implementing 30% more SIMD/FP registers than the A64FX (160 vs. 128). The A64FX can issue up to 7 micro-operations (μOP) per cycle, Graviton3 can issue up to 15, 8 for SKX, and 11 for Rome. The A64FX and SKX are capable of committing 4 micro-operations per cycle. However, SKX can merge two micro-operations into one fused micro-operation, increasing its theoretical commit rate to 8 micro-operations. Rome can commit up to 8 macro-operations (MOP) – i.e., ALU, memory, or merged ALU/memory operation – per cycle. The reorder buffer (ROB) of Graviton3 (256 entries) is twice as big as the A64FX's (128 entries). SKX and Rome have an identical-size ROB (224 entries). The sizes of the load buffers (LB) and store buffers (SB) of the CPUs are relatively different. Graviton3 and SKX implement the largest LB, with 85 and 72 entries, respectively. The LB of the A64FX has 40 entries, and Rome implements a 44-entry LB. Similarly, Graviton3 and SKX have the largest SB (90 and 56 entries, respectively). The A64FX implements a 24-entry SB, half the size of Rome's. Additionally, a store instruction on the A64FX occupies one entry in both the load and the store buffer. While SKX implements a unified reservation station (RS) with 97 entries, both the A64FX and Rome have several smaller RS. The A64FX divides its reservation station into 2 × 20 entries for 2 integer, floating-point, and SIMD pipelines, 2 × 10 entries for 2 address calculation pipelines, and 19 entries for the branch pipeline. Rome's reservation station has 4 × 16 entries for 4 integer pipelines (scalar+SIMD), 28 entries for 3 address calculation pipelines, and 36 entries for 4 floating-point pipelines (scalar + SIMD).

Table 3
Out-of-order resources of the experimental setup.

	A64FX	Graviton3	SKX	Rome
General registers	96	N/A	180	180
SIMD/FP registers	128	N/A	168	160
Issue width	7 (μOP)	15 (μOP)	8 (μOP)	11 (μOP)
Commit width	4 (μOP)	N/A	4-8 (μOP)	8 (MOP)
ROB (entries)	128	256*	224	224
LB (entries)	40	85*	72	44
SB (entries)	24	90*	56	48
RS (entries)	2 × 20 + 2 × 10 + 19	N/A	97	4 × 16 + 28 + 36

*Neoverse V1 CPU defaults.

5. Arm porting of genomics kernels

Most kernels presented in Section 3 target x86 architectures and have not been extensively tested nor optimized for Arm machines. Thus, it was expected that some kernels could run into failures and even generate incorrect results. To verify the execution of the kernels, we used the SKX system to compute the correct output for all kernels and inputs (i.e., ground truth).

For our experiments, we used the GNU compiler (GCC) on Graviton3 (v11.2.0), SKX (v10.1.0), and Rome (v10.2.0). On the A64FX, we used GCC (v10.2.0) and the Fujitsu Compiler (FCC) (v4.2.0b). For most kernels, FCC-compiled binaries exhibited better performance. The Fujitsu Compiler implements two compilation modes: a traditional mode (Trad) based on compilers for earlier systems and a Clang mode based on Clang/LLVM. In all cases, we obtained better execution times when compiling with FCC's Clang mode. We lacked FCC-compiled versions of key-optimized Python libraries. For these reasons, all the results presented in this document for the A64FX have been obtained using the Clang mode of FCC, excluding the two Python kernels (NN-BASE and NN-VARIANT), whose libraries were compiled using GCC.

We compile all kernels with at least `-O2` optimization level and enable CPU-specific optimizations: `-march=armv8-a+sve` on the A64FX, `-mcpu=native` on Graviton3 and `-march=native` on SKX and Rome. Enabling CPU-specific optimizations in ABEA and POA resulted in incorrect executions, probably due to programming errors in the original source code. Therefore, such optimizations are not used for these two kernels.

After performing the appropriate modifications to the kernels so all of them successfully execute on Arm, we applied further optimizations to some kernels to improve the performance obtained in this architecture. Such optimizations are described in the following subsections.

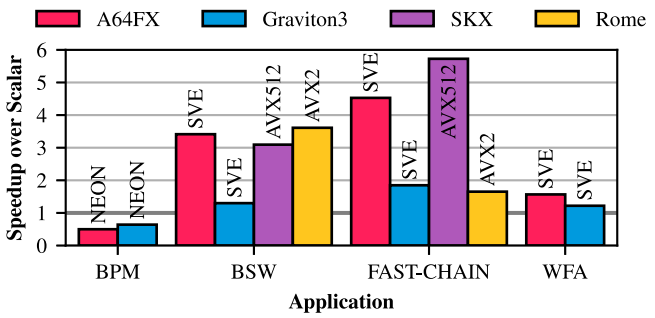

Fig. 2. Speedup of SIMD kernels over their scalar version on the experimental setup using the large inputs.

5.1. Exploiting vectorization

Some kernels implement x86-vectorized versions of their most time-consuming parts. In particular, BSW and FAST-CHAIN include AVX2 and AVX512 versions of their critical functions using intrinsics. Similarly, POA implements SIMD versions of its code using AVX2-intrinsics and SIMD Everywhere (SIMDe). We have implemented SVE-intrinsics versions of FAST-CHAIN, BSW, and WFA and a Neon-intrinsics version of BPM. SIMDe does not fully support SVE yet, so we could not leverage POA's SIMDe version. Fig. 2 shows the speedup of vectorized kernels over their scalar version on the experimental setup using the large input of the kernels. Note that the SVE vector length of Graviton3 (256 bits) is half the A64FX's (512 bits), and therefore the performance speedups of SVE kernels over their scalar versions are more modest in Graviton3.

BPM: The core idea behind vectorizing BPM is to transform the alignment operations used to fill the dynamic programming table into simple machine-word operations. These simple operations are integer additions, bit shifts, and bitwise `OR`s and `AND`s. This way, various dynamic programming cells are bit-packed within a machine word and its dependencies are encoded using bit-wise operations. In packed SIMD, vector operations are performed in independent packets with a maximum width equal to the machine's maximum word width, rather than a whole bit vector (i.e., it is not possible to perform a 128-bit width operation in a 64-bit double word machine). For example, when performing a left-shift operation, the leftmost bit of each word is lost. However, in order to vectorize BPM we would want this bit to be appended to the closest-left word, effectively performing a vector-width left-shift operation. To circumvent this problem, we must perform additional operations to manually carry that bit to the correct position. The number of additional operations required by this approach to work scales with the vector length. Thus, we decided to evaluate the potential of the vector version of BPM using the Neon vector extension (128-bit vectors). The vectorized loop executes 1.7× more instructions than the original but performs 2× fewer iterations.

On the A64FX, SIMD versions of simple instructions, like integer addition, were much more expensive than scalar ones. For example, a simple 64-bit addition takes one cycle, while a vector addition of two 64-bit words takes four cycles. This difference in latencies leads to a slow-down of 2×. Graviton3 has lower SIMD latencies. However, the increase in the number of instructions in the loop leads to a 30% performance loss. Since we did not gain any performance using the Neon version, it was discarded in favor of the original scalar code. We believe that an inter-sequence or coarse-grain approach (i.e., perform the sequence alignment of several sequences simultaneously) will deliver better performance since it simplifies the vectorization.

BSW: The SVE version of BSW [70] is a translation to Arm SVE-intrinsics of the x86-vector version found in BWA-MEM2, which groups the sequence alignment of multiple equal-length sequences via SIMD instructions (i.e., inter-sequence vectorization). The x86-intrinsics version of BSW relies on masks and blend operations to select valid entries from the vector registers. The SVE version takes advantage of SVE's predicate instructions to avoid the need for blend operations, effectively reducing the number of total instructions executed. BSW uses integers of 16 bits, allowing to process 32 elements per iteration using SVE-512 (A64FX) and 16 using SVE-256 (Graviton3).

The SVE version of BSW performs 3.4× and 1.3× faster than its scalar version on the A64FX and Graviton3, respectively.

FAST-CHAIN: Our SVE implementation of FAST-CHAIN is a translation to SVE intrinsics of the x86 version. The original x86 implementation of FAST-CHAIN executes its main loop scalar version (i.e., avoids executing the vectorized loop) when the number of iterations to perform is small. Additionally, as usual in x86 vector loops, it implements a loop-tail to process the remaining elements. Since SVE is vector-length agnostic, we could avoid most of the logic of the x86 version, reducing the number of performed instructions.

The x86 vectorized version of FAST-CHAIN uses 32 bits anchors. In some cases, 32-bit anchors are not sufficient, and this kernel generates incorrect results. To solve this, we have implemented 64 and 32 bits SVE versions of FAST-CHAIN. The 64 bits version always outputs correct results, but we have used the 32 bits implementation to compare against the 32 bits x86 implementation.

GenArchBench's SVE version of FAST-CHAIN runs 4.5× and 1.8× faster than its scalar version (CHAIN without heuristics) on the A64FX and Graviton3, respectively. Experimental results show that the performance of FAST-CHAIN compared to regular CHAIN greatly depends on the input used—the usage of heuristics may lead to performance variations based on the characteristics of the input. For instance, using GenArchBench's large input, our SVE version of FAST-CHAIN is 2.2× faster than regular CHAIN on the A64FX, but it presents a 1.4× slowdown on Graviton3.

WFA: The Wavefront Alignment Algorithm consists of two operations: compute the next wavefront (`next` operation) and extend all the farthest-reaching points of a wavefront by exact matching characters from two strings (`extend` operation). The `next` operation can be automatically vectorized by the compiler due to its simple computational pattern. In contrast, the `extend` operator cannot be automatically vectorized as each diagonal requires an irregular amount of computations. To this end, we have vectorized the `extend` operation using a custom implementation relying on SVE intrinsics. Each vector lane extends a different diagonal, comparing four bases per lane until a mismatch is found. Because each diagonal requires a different number of character comparisons, some lanes can require more iterations than others. We tackle this problem by masking the lanes as they finish the extension process. This way, several diagonals are extended in parallel.

The SVE version of WFA delivers a 1.6× and 1.25× speedup over its scalar version on the A64FX and Graviton3, respectively.

5.2. Optimized libraries

Many HPC kernels and tools rely on frequently used libraries. It is common for vendors, such as Arm, Fujitsu, or Intel, to develop optimized versions of widely used functions and libraries targeting their systems and architectures. For genome data analysis, some tools exploit neural networks (NNs) to improve the quality of their analysis and results. For the GenArchBench, we have tested different implementations of the libraries used by the NN-BASE and NN-VARIANT kernels.

NN-BASE: The NN-BASE kernel builds upon the PyTorch library [57]. On the A64FX we have used an optimized version of PyTorch for this specific CPU provided by Fujitsu. On Graviton3, we tried two different PyTorch backends: PyTorch compiled with OpenBLAS (recommended by Arm) and PyTorch compiled with oneDNN optimized with ACL (labeled as experimental). The docker images with the two backends are available in [71]. The oneDNN-ACL backend performed 6.5× better than the OpenBLAS one, and therefore, we used it for our experiments. On SKX, we used an optimized version of PyTorch that exploits the AVX512 vector extension. On Rome, we used an optimized

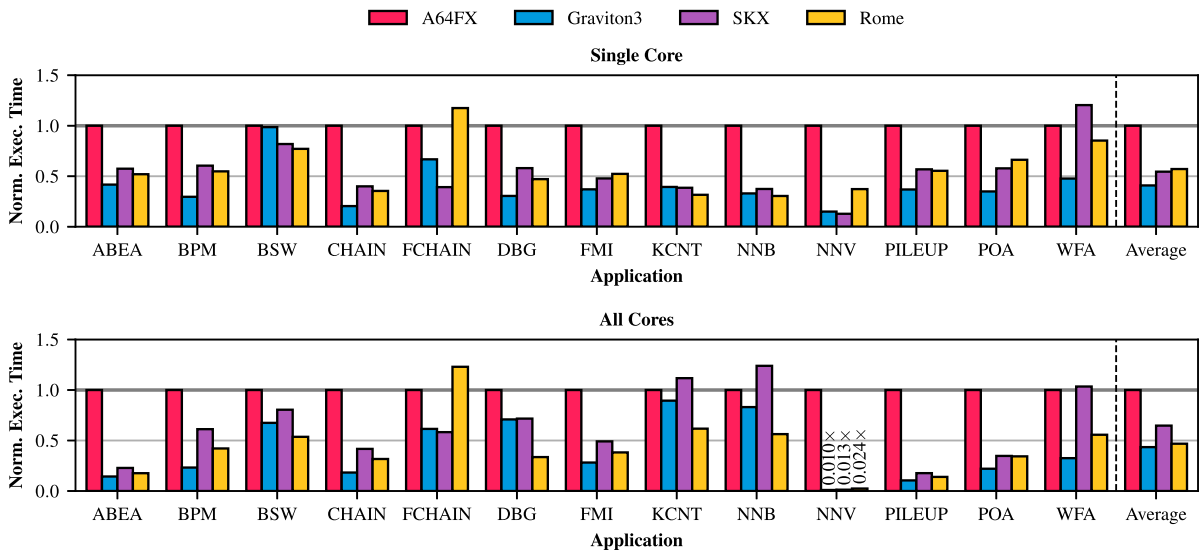

Fig. 3. Single-core (top) and multi-core (bottom) execution time of GenArchBench's kernels on the experimental setup. Multi-core results correspond to executions using all available cores on each machine: 48 threads on the A64FX and SKX and 64 threads on Graviton3 and Rome. The results are normalized to the performance on the A64FX using one core (top) and 48 cores (bottom). FCHAIN, KCNT, NNB and NNV are the abbreviations of FAST-CHAIN, KMER-CNT, NN-BASE and NN-VARIANT, respectively. NN-VARIANT is not taken into consideration for the average in the multi-core plot.

PyTorch version that supports the AVX2 vector extension available on the machine.

NN-VARIANT: The original NN-VARIANT kernel from GenomicsBench is based on Clair [25] variant caller. In turn, this variant caller relies on TensorFlow [58]. Clair uses TensorFlow 1 while Fujitsu provides an optimized version of TensorFlow 2 for the A64FX. For that reason, we decided to use Clair3 [26] instead, an updated version of Clair that relies on TensorFlow 2. To execute using GenArchBench's inputs, we used the Oxford Nanopore `r941_prom_hac_g360+g422` [60] pretrained model from Clair3. On Graviton3, we tested three different TensorFlow backends: TensorFlow compiled with oneDNN optimized with ACL, using TensorFlow's Eigen thread-pool for parallelism (recommended by Arm); TensorFlow compiled with oneDNN optimized with ACL, using ACL's scheduler; and Tensorflow compiled with the Eigen backend. The docker images with the three backends are available in [72]. The Eigen backend performed more than 1.6× better than the other and therefore it was the one used to run our experiments. We used optimized TensorFlow versions on SKX and Rome capable of exploiting the AVX512 and AVX2 vector extensions.

5.3. Algorithmic and code optimizations

This section presents the algorithmic and code optimization we performed to improve the performance of FMI and KMER-CNT.

FMI: GenArchBench's FMI version implements three optimizations proposed by Langarita et al. [70]. One of the most called functions in this kernel is `backwardExt`. To reduce the overhead of the calls, this function is always forced to be in-lined. FMI uses the `builtin_popcount` function. This function counts the number of bits set to one in an integer. None of the tested compilers translates this function to SVE's population count instruction. Instead, they use bitwise operations and masks. To force exploiting SVE capabilities, all calls to `builtin_popcount` are replaced by SVE intrinsics. FMI performance is heavily affected by memory access latencies. To hide these latencies, the optimized version of FMI interleaves the execution of several sequences, effectively performing several memory accesses in parallel. By applying the three presented optimizations, we improved the kernel performance on both Arm machines by roughly 35%.

KMER-CNT: Our experimental evaluation shows that the performance of this kernel is heavily affected by thread migrations. To avoid thread migrations, we ported KMER-CNT from the Pthreads library to OpenMP and set `OMP_PROC_BIND` clause to `true` before executions. This change led to more than 4× speedups on both x86 machines when using all available cores. However, the performance of the Arm machines remained the same.

KMER-CNT relies on two global data structures to store the number of individual k-mers: an array of 4-bit counters and libcuckoo's [55] multi-thread hash-map, which stores 64-bit counters. Each entry of the global array is an 8-bit atomic integer, which is split in half to create two 4-bit counters. The array counters are updated using atomic compare-and-swap operation. Once the 4-bit counter of a k-mer saturates, the following increments are performed in the global hash map, also relying on atomic compare-and-swaps to update its counters.

Even by avoiding thread migrations, the scalability of the original kernel was poor on all the machines. It achieved a maximum of 7× and 5× vs. serial execution on the A64FX (48 threads) and Graviton3 (64 threads), respectively. We decided to implement two new approaches to try to improve parallel performance.

The application divides the input between the available threads. Each thread iterates through the k-mers of its part of the input and increments the counter of the read k-mers in the global array or hash map. Since the input is read sequentially and there is almost no computation to perform, most of the execution time is spent accessing the global counters in mutual exclusion. To reduce contention and improve data locality, our first approach assigns part of the input to each thread, and all threads read the full input but only count part of the k-mers. This way, instead of having a single global array and hash map, each thread can have a smaller instance of the data structures and access them without contention.

The original version of the kernel uses compare-and-swap instead of fetch-and-add to update the counters because each 8-bit entry of the array stores two 4-bit counters. In order to limit memory usage, when the k-mer size is greater than 17, the kernel does not instantiate the global array, and all the counting takes place on the hash-map. For the maximum allowed k-mer size (17), we require 2^{17} 4-bit entries in the global array, resulting in 8 GB of memory. Since we have more than enough memory in all systems, our second approach uses 8-bit instead of 4-bit counters, doubling the memory requirements. This enables fetch-and-add usage and reduces the number of accesses to the hash map.

The single thread execution time of the kernel did not change with any of the new versions. Our first approach (private structures) equally

divides the possible k-mers between threads. However, some k-mers are more common in the input, causing load imbalance between threads deriving in even poorer scalability than the original kernel. The second approach (fetch-and-add) improves the kernel's scalability on all the machines: it runs 2.5×, 1.4×, 3× and 2.3× faster than the original version on the A64FX (48 threads), Graviton3 (64 threads), SKX (48 threads) and Rome (64 threads), respectively. Consequently, we used the fetch-and-add approach for the rest of the experiments.

6. Performance characterization

This section presents a detailed performance characterization of the kernels in our experimental setup. We use the optimized versions of the kernels described in Section 5. For all of the studies presented, we have annotated the code of the kernels to define their region of interest, i.e., we only study the part of the kernels dedicated to meaningful computation. All the results shown in this section have been computed using the large input of each kernel. We noted minimal variation between executions of the kernels, with a maximum relative standard deviation of 5% observed across 10 repetitions of the experiments. Consequently, we showcase the results based on a single execution in the figures. While executing DBG with high thread counts in Graviton3, outlier execution times occurred approximately 10% of the time. In the case of DBG in Graviton3, we selectively present results from an inlier execution.

6.1. Single-thread performance

The top plot of Fig. 3 shows the single-thread execution time of each kernel on the experimental setup. The results are normalized to the performance on the A64FX (see Table 3 of the supplementary material for the execution times of the kernels).

The A64FX features significantly fewer out-of-order resources, a smaller memory hierarchy, and higher memory latencies than the rest of the systems. On average, the former is 2.4×, 1.8×, and 1.7× slower than Graviton3, SKX and Rome on single-threaded executions, respectively. Exploiting the SVE capabilities of the A64FX helps to reduce this slowdown. SVE vectorized kernels (BSW, FAST-CHAIN, and WFA) present better-than-average performance on the A64FX: BSW performance is similar to the exhibited on Graviton3 and only 17% worse than the performance on the x86 machines, FAST-CHAIN performs better than on Rome, and WFA performs better than on SKX. Note that BSW and FAST-CHAIN exploit AVX512 on SKX while they leverage AVX2 on Rome, and that WFA is not vectorized on the x86 machines. The deep-learning kernels (NN-BASE and NN-VARIANT) are the worst-performing on the A64FX.

Graviton3 performs exceptionally well in single-thread executions. On average, it presents 2.44×, 1.33×, and 1.39× performance speedups with respect to the A64FX, SKX and Rome, respectively. FAST-CHAIN performance on Graviton3 is 1.8× better than on Rome (AVX2) but 70% worse than on SKX since it exploits AVX512 (512 bits) on that machine. WFA runs 2.5× and 1.8× faster on Graviton3 than on SKX and Rome, respectively. In contrast to the A64FX, the deep-learning kernels (NN-BASE and NN-VARIANT) deliver good performance on Graviton3, showing speedups of between 3.1–6.2× compared to the A64FX.

6.2. Parallel performance

We evaluate the parallel performance of GenArchBench's kernels using different thread counts: 2, 8, 24, 48, and 64. The A64FX and SKX implement 48 cores. Hence, executions with more than 48 threads have only been performed on Graviton3 and Rome. Controlling thread affinity was mandatory in our experiments to achieve good parallel performance on the machines, especially on the A64FX. For most kernels, all the executions were performed by binding threads to cores.

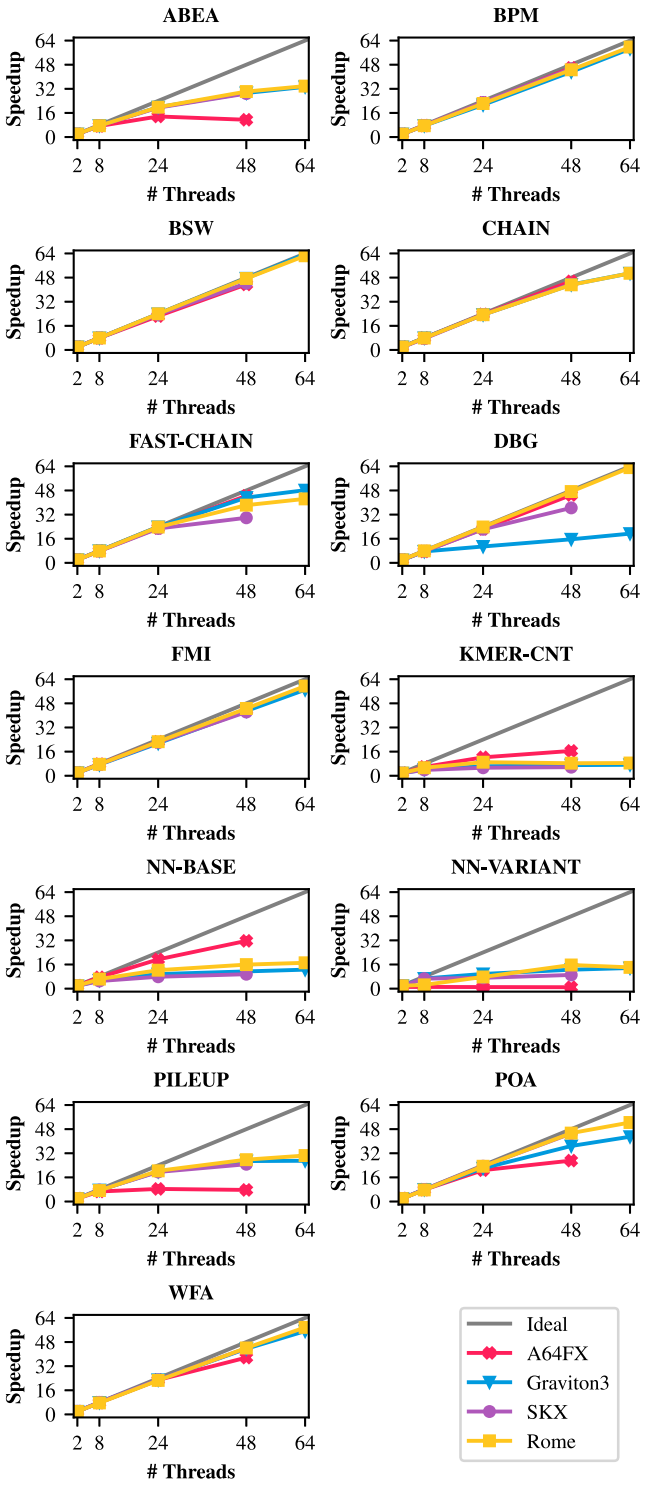

Fig. 4. Speedup over serial execution of GenArchBench's kernels on the experimental setup. We show the achieved speedup using different thread counts: 2, 8, 24, 48, and 64. The A64FX and SKX 64-threads points are not shown in the figure, since those machines only implement 48 cores.

ABEA, NN-BASE, and NN-VARIANT do not allow full thread affinity control. Therefore, thread migrations can occur in these three kernels.

Fig. 4 shows the speedup over serial execution achieved by the kernels on the experimental setup using the previously presented thread counts. Additionally, the bottom plot of Fig. 3 compares the performance obtained using all available cores on each machine: 48 threads

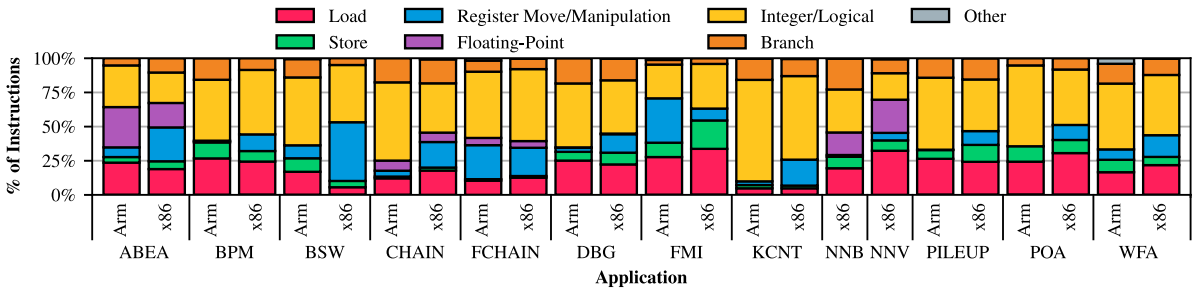

Fig. 5. Instruction mix of GenArchBench's kernels on Arm (A64FX) and x86 (SKX). FCHAIN, KCNT, NNB and NNV are the abbreviations of FAST-CHAIN, KMER-CNT, NN-BASE and NN-VARIANT, respectively.

on the A64FX and SKX and 64 threads on Graviton3 and Rome. For the total execution times of the kernels, refer to Table 3 of the supplementary material. The parallel performance of NN-VARIANT on the A64FX is extremely poor; therefore, it is not considered for the average calculation. The results are normalized to the performance on the A64FX using 48 threads.

All GenArchBench's kernels exploit coarse-grain parallelism. Most of them, except for KMER-CNT, present little to no interaction between threads. It can be seen that some kernels achieve near-perfect scaling on all machines. This is the case for BPM, BSW, CHAIN, FMI, and WFA. For this set of kernels, the normalized plots using 1 thread and all available cores are similar. It is important to note that Graviton3 and Rome show some performance gains compared to the other two machines since the number of available cores is higher.

In ABEA and PILEUP, the primary thread reads the full input and splits it into smaller chunks that are dynamically assigned to idle threads. This is the same scheduling implemented by other kernels, such as BSW. However, the chunks used in ABEA and PILEUP are significantly bigger, leading to load imbalance. For ABEA, the smallest grain size assigned to a thread is a whole read. Therefore, to improve scalability, we would need to change the granularity used by the kernel. PILEUP implements a default chunk size of 100 kbp. Dynamically choosing the chunk size based on the input would reduce load imbalance. The parallel performance of both kernels should also improve by using larger inputs. In both kernels, the A64FX presents poorer scalability than the other three machines, further increasing their performance difference compared to single-thread executions.

DBG also implements dynamic scheduling and shows good scalability and load balance on the A64FX, SKX, and Rome. Runs of DBG on Graviton3 using more than 8 threads present high variability in contrast to the other machines, resulting in poor scalability in most cases. Due to this behavior, DBG performance on Graviton3 using all cores is similar to SKX's.

In KMER-CNT, all threads continuously perform random memory write accesses using atomic operations, resulting in high memory contention and poor scalability. For this kernel, the A64FX presents the best parallel scalability: a maximum speedup of 16× with respect to single-thread executions vs. a maximum of 5× on the other machines. This results in similar performance between the A64FX, Graviton3 and SKX when using all available cores.

NN-BASE does not implement any high-level parallelism. It relies on PyTorch multithreading, which allows using intra-op parallelism (via math libraries like Intel MKL) and inter-op parallelism. This approach works relatively well on the A64FX but offers poor scalability on the rest of the systems.

NN-VARIANT presents significant load imbalance even with low thread counts. In order to improve this, we tried two different scheduling policies: to assign each core a similar-sized chunk of the input and to dynamically assign small chunks to available threads. In both cases, the time needed to process the chunks was unpredictable. Additionally, NN-VARIANT relies on Tensorflow, making it difficult to control the number of threads used. Besides the kernel's high-level parallelism, Tensorflow uses between 1 to 4 threads during the model inference step, degrading parallel performance when NN-VARIANT uses more than 1/4 of the available threads. The performance of NN-VARIANT on the A64FX does not improve with any number of threads, resulting in extremely poor parallel performance compared to the other systems.

6.3. Instruction mix comparison

An application's instruction mix determines which processor pipelines and functional units are the most used during its execution. To obtain it, we used the instruction mix report offered by the Fujitsu Advanced Performance Profiler (FAPP) on Arm (A64FX) and a modified version of DynamoRIO's opcode_mix tool [73] on x86 (SKX) that divides the executed instructions into different categories[2]. The instruction mix offered by both tools is significantly different, so we designed a mapping from FAPP categories to the categories defined in our modified DynamoRIO (see Table 1 of the supplementary material).

Fig. 5 shows the instruction mix of GenArchBench's kernels in Arm (A64FX) and in x86 (SKX). The Fujitsu Advanced Performance Profiler does not allow the creation of child processes. For this reason, we have not been able to compute the Arm instruction mix of the Python kernels (NN-BASE and NN-VARIANT). The `Register Move/Manipulation` category includes any data movement between registers or manipulation of the contents of a register without performing any arithmetic operation (like the `setz` instruction of x86). The `Other` category includes prefetching, cryptographic, string, and special instructions (such as the `DCZVA` and `MOVPRFX` instructions of Arm or the `RDRAND` instruction of x86).

ABEA and the deep-learning kernels (NN-BASE and NN-VARIANT) are the only kernels that perform a significant number of floating-point operations. As explained before, we lack the tools to compute the instruction mix of NN-BASE and NN-VARIANT on Arm, but as for the rest of the kernels, we expect it to be similar on both architectures. CHAIN and FAST-CHAIN also perform floating-point operations but are mainly dominated by integer, logical, and register move/manipulation instructions. FMI mainly performs memory operations and is heavy on register move/manipulation instructions on Arm. On the other hand, KMER-CNT performs nearly no data movements (although memory accesses in this kernel are expensive, as shown in Section 6.4). The rest of the kernels mostly execute integer and logical instructions and between 30% and 40% of data movement operations.

6.4. Microarchitecture bottleneck analysis

We have studied the microarchitecture bottlenecks of each application using FAPP on the A64FX, Perf on Graviton3 and Rome, and Intel VTune Profiler on SKX. We have not been able to compute the

[2] https://github.com/LorienLV/dynamorio

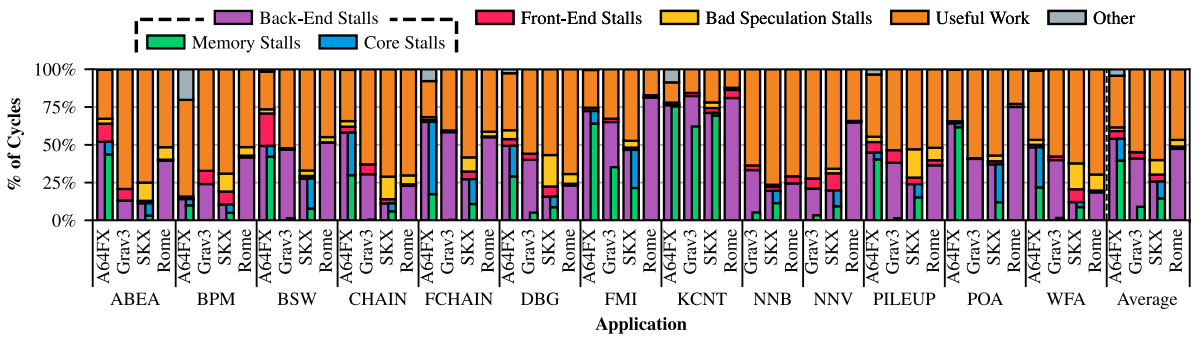

Fig. 6. Microarchitecture bottlenecks of GenArchBench's kernels on the experimental setup. FCHAIN, KCNT, NNB and NNV are the abbreviations of FAST-CHAIN, KMER-CNT, NN-BASE and NN-VARIANT, respectively. Grav3 is the abbreviation of Graviton3.

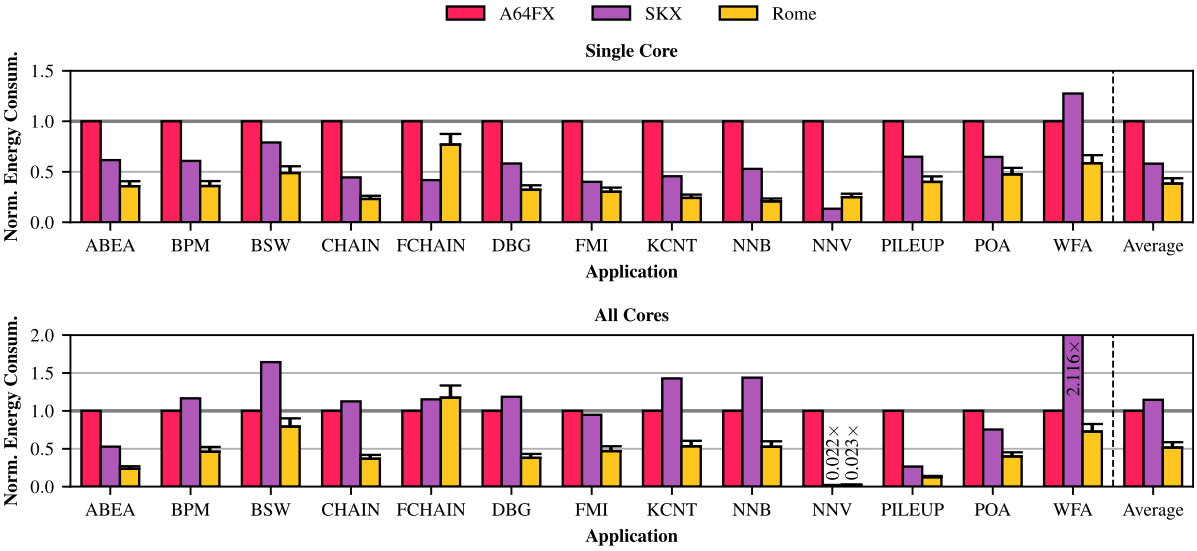

Fig. 7. Single-core (top) and multi-core (bottom) energy-to-solution of GenArchBench's kernels on the experimental setup. Multi-core results correspond to executions using all available cores on each machine: 48 threads on the A64FX and SKX and 64 threads on Rome. The results are normalized to the performance on the A64FX using one core (top) and 48 cores (bottom). FCHAIN, KCNT, NNB and NNV are the abbreviations of FAST-CHAIN, KMER-CNT, NN-BASE and NN-VARIANT, respectively. NN-VARIANT is not taken into consideration for the average in the multi-core plot. Graviton3 is not included in this plot since it does not expose its energy consumption.

microarchitectural bottlenecks of Python kernels (NN-BASE and NN-VARIANT) on the A64FX, as FAPP does not allow the creation of child processes.

Analogous to the instruction mix, we have designed a mapping from the different profilers and systems metrics to our microarchitecture bottleneck categories: `Back-End stalls`, `Front-End Stalls`, `Bad Speculation Stalls`, `Useful Work` and `Other`. A detailed description of such mappings can be found in Table 2 of the supplementary materials. We further split the category `Back-End stalls` into `Memory Stalls` and `Core stalls`. The `Memory Stalls` category includes stalls due to main memory and caches. Unfortunately, we could not compute `Memory Stalls` nor `Core stalls` in Rome since it does not implement the required performance counters. The same problem occurs on Graviton3, where we could not measure `Core stalls` and the `Memory Stalls` category only includes stalls due to main memory.

Most hardware events in SKX measure slots instead of cycles. Although we have homogenized the formulas used in each machine as much as possible, it is important to note that performance counters are not standardized between machines, let alone architectures, so similar metrics may count moderately different events on different machines. Comparisons between counters of different machines must be seen as rough estimations of reality.

Fig. 6 shows the microarchitecture bottlenecks of GenArchBench's kernels on the experimental setup. On average, there are significantly more memory stalls on the A64FX than on the rest of the machines. While the A64FX has the highest memory bandwidth, it implements a small memory hierarchy and suffers from high memory latencies. On the other hand, the bottlenecks on Graviton3 are more similar to those shown by the x86 machines. ABEA, BSW, DBG, and POA suffer from a high percentage of memory stalls on the A64FX compared to the other machines. On the other hand, FMI and KMER-CNT are mainly memory-bound on all machines. These two kernels mainly perform random memory accesses and do not exploit temporal or spatial locality. Thus, they are highly impacted by memory latencies. Kernels such as DBG, PILEUP, or WFA suffer from a high count of stalls due to bad speculation on x86 systems, especially on SKX, while the bottleneck on Arm is much smaller. Although this can very well be due to how these stalls are counted on different machines, we believe that Arm predicated instructions play an important role in this metric. The number of cycles NN-VARIANT dedicates to useful work on Rome is small compared to the other machines, explaining its poor performance compared to other kernels on this machine. On the contrary, the `Useful Work` metric percentage of NN-BASE is considerable on all machines.

6.5. Energy consumption

Fig. 7 shows the energy-to-solution of GenArchBench's kernels on the A64FX, SKX, and Rome using one (top plot) and all available cores (bottom plot) on each machine. Graviton3 is not included in the

figure as it does not expose its energy consumption. Additionally, we cannot measure the energy consumption of Rome's DRAM. However, since Rome and SKX use the same DRAM technology, we have added a 12% extra energy consumption to Rome's measurements based on the energy consumption of SKX's DRAM (expressed as error bars in Fig. 7). Energy consumption was measured using the Fujitsu power API on the A64FX, and an in-house library[3] based on the Running Average Power Limit (RAPL) on SKX and Rome. Since there is a large difference between machines, the results of NN-VARIANT were not taken into consideration for the average calculation.

The maximum power consumption of the A64FX (120 W) is substantially lower than that of the x86 systems: 2 × 150 W on SKX and 225 W on Rome. However, SKX and Rome are capable of dynamically scaling their frequency depending on the load of the system (CPU throttling), while the A64FX constantly consumes power near its peak, even on low usage.

The results shown in the top plot of Fig. 7 are highly similar to those presented in the top plot of Fig. 3. On average, the A64FX consumes 1.7× more than SKX and 2.6× more than Rome in single-thread executions. Kernels with good single-thread performance, such as BSW or WFA, show better-than-average energy-to-solution results on the A64FX.

The previous picture changes when using all available cores on each machine (bottom plot of Fig. 7). In this scenario, all the machines are near their peak power consumption. The A64FX consumes less energy than SKX in 8 out of 13 kernels (12% less energy consumption on average), while Rome is again the most energy efficient when executing most kernels (1.9× less energy consumption than the A64FX).

6.6. Evaluation of a real genomics tool

Finally, we evaluate the performance of a real genomics tool that uses some of the ported kernels presented in this article. For this matter, we use BWA-MEM2 [12], a read mapping tool (Fig. 1-3.a.1) that employs the FMI kernel for the seed stage (Fig. 1-3.a.1.1) and the BSW kernel for the extend stage (Fig. 1-3.a.1.3). We use the optimized versions of the kernels, which have been presented and evaluated in this work. In our tests, BWA and FMI represent 45% and 34% of the total execution time of BWA-MEM2, respectively.

For our study, we have used three input datasets, each one with 1.25M reads of different lengths obtained from real sequencing machines: D3 [74], D4 [75], and D5 [76]. These reads are aligned against the human genome [77].

Fig. 8 shows the performance of BWA-MEM2 using one (left) and all available cores (right) on the machines of the experimental setup. On single-thread executions, Graviton3, SKX and Rome perform similarly, showing over 2× speedups over the A64FX. When using all available cores, Graviton3 performs 10% better than Rome, and more than 30% better than SKX (as can be expected due to the difference in cores). The A64FX, on the other hand, shows 2× slowdowns compared to SKX. Both single-thread and multi-thread results can be easily correlated with the ones shown in Fig. 3. In the results from GenArchBench, BSW on Graviton3 showed slowdowns with respect to the x86 systems. However, FMI performed better on Graviton3, and the kernel represents a higher percentage of the total execution time of BWA-MEM2. Similarly, the A64FX delivered good performance when executing BSW, but severe slowdowns with respect to the other machines when executing FMI.

We also evaluate the parallel scalability of BWA-MEM2 using 2, 8, 24, 48, and 64 threads. As presented previously (see Fig. 4), BSW and FMI achieved perfect scaling when executed standalone as part of GenArchBench. As we could anticipate, BWA-MEM2 also showed excellent parallel scalability, as presented in Fig. 9.

[3] https://github.com/LorienLV/rapl_stopwatch

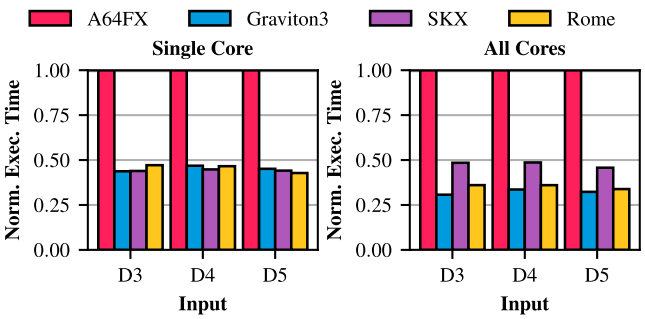

Fig. 8. Execution time of BWA-MEM2 using one core (left) and all available cores in each machine of the experimental setup (right). We show results using three inputs: D3, D4, and D5. The results are normalized to the performance on the A64FX using one core (left) and 48 cores (right).

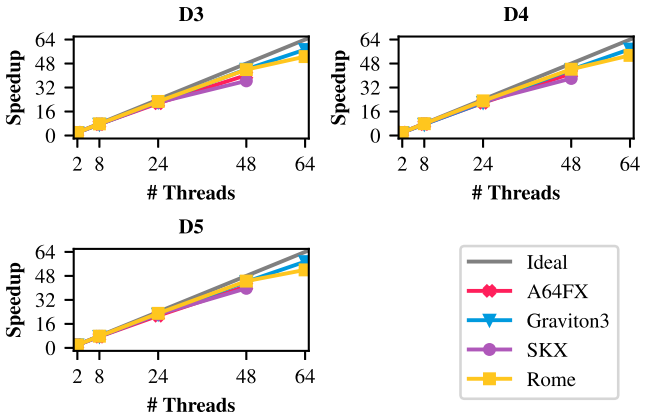

Fig. 9. Speedup over serial execution of BWA-MEM2 on the experimental setup for three inputs: D3, D4 and D5. We show the achieved speedup using different thread counts: 2, 8, 24, 48, and 64. The figure does not show the A64FX and SKX 64-thread points since those machines only implement 48 cores.

7. Discussion

In this section, we discuss the key lessons we have learned through this project.

We have found that working with intrinsics-vectorized kernels tends to be laborious and error-prone. Most of these kernels are not adapted to run on non-x86 architectures, as they often rely on SSE, AVX2, and AVX512 vector intrinsics. To make things worse, some applications do not provide a scalar version that can run on any architecture. The complexity of the studied kernels makes it challenging to auto-vectorize them, requiring the use of intrinsics-vectorized versions to achieve performance improvements. Unfortunately, this approach sacrifices maintainability and readability since it requires multiple versions tailored to different vector extensions, instead of having a single version for testing and maintenance.

Vector agnostic SIMD extensions, such as SVE, enable easier to develop and maintain intrinsics-code that works with different vector lengths. However, we believe that a higher-level solution, in the middle of intrinsics and auto-vectorized code, similar to the SIMDe library, maybe the best way to enable applications to exploit the vector capabilities of current and future vector extensions and architectures.

Measuring and comparing micro-architectural bottlenecks in different systems is extremely challenging. Different CPUs implement completely different hardware counters, and even similar counters may register widely different events. Additionally, some systems expose very few counters, making it challenging to extract meaningful information from them. We found it extremely important for both developers and CPU manufacturers to standardize hardware counters, making it easier

to perform deep analysis of applications' bottlenecks, and enabling straightforward comparison between different systems.

When working with the A64FX, it is crucial to be aware of its micro-architecture in order to achieve optimal performance. The CPU has limited out-of-order resources compared to other HPC processors, which is why it is recommended to use aggressive approaches like loop fissioning to save resources [78] (the Fujitsu Compiler offers hints for this). Moreover, the NUMA configuration and memory characteristics of the A64FX call for minimizing remote memory accesses to reduce latencies and maximize memory bandwidth. We understand that the A64FX is a memory-bandwidth-oriented CPU and that this negatively affects latencies, making it not suitable for all types of applications. However, we expect further iterations of the CPU to implement more OoO resources in order to be competitive with other HPC processors for a wide range of applications, such as genomics. Also, we believe that it will be very beneficial for the energy efficiency of the processor to implement dynamic frequency scaling.

Graviton3 delivers excellent out-of-the-box performance for all of GenArchBench's kernels without requiring the user to extensively know its micro-architecture for fine-tuning. Nonetheless, as of the day we are writing this document, the system is extremely closed, providing scarce hardware counters and not exposing its frequency and power consumption, making it difficult to deeply understand the performance of applications and study the energy efficiency of the system.

8. Related work

Outside the field of genomics and bioinformatics, there are many examples of domain-specific benchmark suites. Some widely known examples are LINPACK Benchmarks [79], for linear algebra; the GAP benchmark suite [80], for graph processing; or BigDataBench [81], for big data.

Focused on genomics, GenomicsBench [39], on which this work is based, is a benchmark suite that includes 12 computationally demanding kernels from common steps in genome data analysis. GenomicsBench includes CPU and GPU kernels, targeting the x86 HPC and NVIDIA GPU ecosystem. GenarchBench includes 10 CPU kernels of this suite and three additional ones. All GenArchBench's kernels have been ported to the Arm architecture and, most of them, implement optimizations targeting Arm. Additionally, GenArchBench includes automatic regression tests to verify the correctness of the execution of its kernels.

The BioPerf [82] benchmark suite, released in 2005, compiles DNA and protein analysis applications, such as Blast [18] or FASTA [83], two of the most widely used aligners at the time. Moreover, it includes three inputs per kernel; pre-compiled binaries for x86, PowerPC, and Alpha; execution scripts; and simulation points (Simpoints) to simulate the execution of the kernels. Similarly, BioBench [84], also released in 2005, offers some of BioPerf's benchmarks and presents a performance characterization. Recently, some of the kernels included in BioBench were updated in BioBench2 [85].

Furthermore, many publications analyze state-of-the-art genomics workloads, algorithms, and tools. Jason et al. [86] present a review of the steps involved in genome assembly. Similarly, Mohammed et al. [87] focus on the genome resequencing pipeline. Most notably, one of the main bottlenecks in genome sequence analysis is read mapping. As a result, there are many works discussing the algorithms used for this process and its acceleration on HPC processors [42,88–90].

Recently, there have been multiple efforts to accelerate widely-used genome analysis kernels exploiting novel hardware solutions [91–96]. Tony et al. [97] provide a comprehensive review of state-of-the-art hardware acceleration techniques for genomics.

9. Conclusions

This paper presents GenArchBench, an Arm-based benchmark composed of 13 computationally demanding kernels for HPC CPUs. GenArchBench's kernels are used within widely used genomics tools and exploit coarse-grain thread-level parallelism. We have optimized these kernels to exploit the capabilities of A64FX and Graviton3 CPUs, including the novel Arm Scalable Vector Extension (SVE). Moreover, we have performed algorithmic and code optimizations and adapted them to exploit Arm-optimized libraries. This work also introduces a detailed performance characterization on four different processors: two Arm CPUs (A64FX and Graviton3) and two x86-64 CPUs (Intel Xeon Skylake Platinum 8160 and AMD EPYC 7742 Rome).

Our results show that Graviton3 performs 1.3× better than the x86 machines on single-thread executions; unlike the A64FX, which shows 2× slowdowns compared to the x86 machines. SVE-enabled kernels perform 1.5–4.5× better on the A64FX and 1.2–1.8× better on Graviton3 compared to their scalar version. On multi-threaded executions, Graviton3 showed performance improvements between 7% and 32% compared to the x86 systems, whereas the A64FX performed 2.3× worse than Graviton3.

Most notably, our results highlight that although the A64FX offers high memory bandwidth, the system implements a modest memory hierarchy and suffers from long-latency memory access. In turn, this is translated into inefficiencies and bottlenecks when executing GenArchBench's kernels. Although only two kernels can be classified as memory bound, more than half of the kernels spend many cycles waiting for memory data on the A64FX.

Regarding power consumption, the peak power consumption of the x86 systems is significantly higher than that of the A64FX. However, x86 systems implement dynamic frequency and voltage scaling based on system load to control energy consumption. In contrast, the A64FX nearly always operates at its maximum power consumption level. Albeit, the A64FX demonstrates similar energy-to-solution results compared to SKX when using all available cores. Unfortunately, Graviton3 is a closed system and does not expose power consumption metrics.

To put the performance evaluation results in context, we integrated our optimized versions of FMI and BSW into BWA-MEM2, a production application that uses these two kernels. Then, we evaluated the performance of the application in the experimental setup. BWA-MEM2 performed similarly on Graviton3 and the x86 systems for single-thread executions. However, Graviton3 showed speedups of 1.1–1.5× over the x86 machines when using all available cores. In contrast, BWA-MEM2 performed 2× worse on the A64FX than on the other machines.

Although we have presented many optimizations applied to GenArchBench's kernels, we can foresee room for improvement, not only for Arm architectures but also for x86. Similarly, kernels based on the SIMDe library remain to be vectorized since the library does not support SVE as of today. Moreover, we have observed that deep-learning Python-based kernels deliver extremely poor parallel performance. Due to the increasing importance of these types of applications, we believe that we need to focus on improving their HPC performance.

Looking forward, we envision repeating the evaluation on cutting-edge Arm architectures like Graviton4 and NVIDIA Grace, as well as on the latest x86 microarchitectures such as Intel Sapphire Rapids and AMD Genova. This analysis can provide new valuable insights into the ever-evolving landscape of processor architectures. Furthermore, our research lays a solid foundation for potential porting efforts to other architectures, particularly emphasizing the prospect of adapting our kernels to emerging platforms like RISC-V.

Our results demonstrate that Arm is fully capable of competing with x86 when executing genomics workloads. We believe that GenArchBench lays down the bases for future optimizations of genome analysis tools on Arm. We hope that this benchmark suite can be of help to bioinformatics software developers and computer architects focusing on future HPC Arm architectures.

CRediT authorship contribution statement

Lorién López-Villellas: Writing – original draft, Visualization, Validation, Software, Investigation, Data curation. **Rubén Langarita-Benítez:** Writing – original draft, Visualization, Software, Investigation, Data curation. **Asaf Badouh:** Writing – original draft, Software, Investigation. **Víctor Soria-Pardos:** Writing – original draft, Software, Investigation. **Quim Aguado-Puig:** Writing – original draft, Software. **Guillem López-Paradís:** Software. **Max Doblas:** Software. **Javier Setoain:** Writing – review & editing. **Chulho Kim:** Writing – review & editing. **Makoto Ono:** Writing – review & editing. **Adrià Armejach:** Writing – review & editing. **Santiago Marco-Sola:** Writing – original draft, Supervision, Methodology, Conceptualization. **Jesús Alastruey-Benedé:** Writing – original draft, Supervision, Methodology, Conceptualization. **Pablo Ibáñez:** Writing – original draft, Supervision, Methodology, Conceptualization. **Miquel Moretó:** Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Javier Setoain reports a relationship with Arm Research that includes: employment. Makoto Ono reports a relationship with Lenovo Infrastructure Solutions Group that includes: employment. Chulho Kim reports a relationship with Lenovo Infrastructure Solutions Group that includes: employment. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The link to the repository containing the code and the data is shared in the manuscript.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Science and Innovation MCIN/AEI/10.13039/501100011033 (contracts PID2019-107255GB-C21, PID2019-105660RB-C21, PID2022-136454NB-C22, and TED2021-132634A-I00), by the Generalitat de Catalunya, Spain (contract 2021-SGR-763), by the Gobierno de Aragón (T58_23R research group), by the European Union NextGenerationEU/PRTR, and by Lenovo BSC Contract-Framework Contract (2020).

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.future.2024.03.050.

References

[1] M. Flores, et al., P4 medicine: how systems medicine will transform the healthcare sector and society, Pers. Med. 10 (6) (2013) 565–576, http://dx.doi.org/10.2217/pme.13.57.
[2] L. Chin, et al., Cancer genomics: from discovery science to personalized medicine, Nature Med. 17 (3) (2011) 297–303, http://dx.doi.org/10.1038/nm.2323.
[3] R. Spreafico, et al., Advances in genomics for drug development, Genes 11 (8) (2020) 942, http://dx.doi.org/10.3390/genes11080942.
[4] M.E.K. Niemi, et al., The human genetic epidemiology of COVID-19, Nature Rev. Genet. 23 (9) (2022) 533–546, http://dx.doi.org/10.1038/s41576-022-00478-5.
[5] F. Sanger, et al., DNA sequencing with chain-terminating inhibitors, Proc. Natl. Acad. Sci. 74 (12) (1977) 5463–5467, http://dx.doi.org/10.1073/pnas.74.12.5463.
[6] A.M. Maxam, et al., A new method for sequencing DNA, Proc. Natl. Acad. Sci. 74 (2) (1977) 560–564, http://dx.doi.org/10.1073/pnas.74.2.560.
[7] E.S. Lander, et al., Initial sequencing and analysis of the human genome, Nature 409 (6822) (2001) 860–921, http://dx.doi.org/10.1038/35057062.
[8] J.A. Reuter, et al., High-throughput sequencing technologies, Molecular Cell 58 (4) (2015) 586–597, http://dx.doi.org/10.1016/j.molcel.2015.05.004.
[9] Bonito, 2023, https://github.com/nanoporetech/bonito. (Accessed 7 January 2023).
[10] R.R. Wick, et al., Performance of neural network basecalling tools for oxford nanopore sequencing, Genome Biol. 20 (1) (2019) 1–10, http://dx.doi.org/10.1186/s13059-019-1727-y.
[11] H. Li, Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM, 2013, arXiv preprint arXiv:1303.3997.
[12] M. Vasimuddin, et al., Efficient architecture-aware acceleration of BWA-MEM for multicore systems, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2019, pp. 314–324, http://dx.doi.org/10.1109/IPDPS.2019.00041.
[13] H. Li, Minimap2: pairwise alignment for nucleotide sequences, in: I. Birol (Ed.), Bioinformatics 34 (18) (2018) 3094–3100, http://dx.doi.org/10.1093/bioinformatics/bty191.
[14] B. Langmead, et al., Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Genome Biol. 10 (3) (2009) R25, http://dx.doi.org/10.1186/gb-2009-10-3-r25.
[15] B. Langmead, et al., Fast gapped-read alignment with Bowtie 2, Nature Methods 9 (4) (2012) 357–359, http://dx.doi.org/10.1038/nmeth.1923.
[16] S. Marco-Sola, et al., The GEM mapper: fast, accurate and versatile alignment by filtration, Nature Methods 9 (12) (2012) 1185–1188, http://dx.doi.org/10.1038/nmeth.2221.
[17] P. Ferragina, et al., Opportunistic data structures with applications, in: Proceedings 41st Annual Symposium on Foundations of Computer Science, 2000, pp. 390–398, http://dx.doi.org/10.1109/SFCS.2000.892127.
[18] S.F. Altschul, et al., Basic local alignment search tool, J. Mol. Biol. 215 (3) (1990) 403–410, http://dx.doi.org/10.1016/s0022-2836(05)80360-2.
[19] S. Altschul, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, Nucleic Acids Res. 25 (17) (1997) 3389–3402, http://dx.doi.org/10.1093/nar/25.17.3389.
[20] S.B. Needleman, et al., A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol. 48 (3) (1970) 443–453, http://dx.doi.org/10.1016/0022-2836(70)90057-4.
[21] T. Smith, et al., Identification of common molecular subsequences, J. Mol. Biol. 147 (1) (1981) 195–197, http://dx.doi.org/10.1016/0022-2836(81)90087-5.
[22] O. Gotoh, An improved algorithm for matching biological sequences, J. Mol. Biol. 162 (3) (1982) 705–708, http://dx.doi.org/10.1016/0022-2836(82)90398-9.
[23] A. McKenna, et al., The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data, Genome Res. 20 (9) (2010) 1297–1303, http://dx.doi.org/10.1101/gr.107524.110.
[24] A. Rimmer, et al., Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications, Nature Genet. 46 (8) (2014) 912–918, http://dx.doi.org/10.1038/ng.3036.
[25] R. Luo, et al., Exploring the limit of using a deep neural network on pileup data for germline variant calling, Nat. Mach. Intell. 2 (4) (2020) 220–227, http://dx.doi.org/10.1038/s42256-020-0167-4.
[26] Z. Zheng, et al., Symphonizing pileup and full-alignment for deep learning-based long-read variant calling, Nat. Comput. Sci. 2 (12) (2022) 797–803, http://dx.doi.org/10.1038/s43588-022-00387-x.
[27] R. Poplin, et al., A universal SNP and small-indel variant caller using deep neural networks, Nature Biotechnol. 36 (10) (2018) 983–987, http://dx.doi.org/10.1038/nbt.4235.
[28] Medaka, 2023, https://github.com/nanoporetech/medaka. (Accessed 7 January 2023).
[29] M. Kolmogorov, et al., Assembly of long, error-prone reads using repeat graphs, Nature Biotechnol. 37 (5) (2019) 540–546, http://dx.doi.org/10.1038/s41587-019-0072-8.
[30] S. Koren, et al., Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation, Genome Res. 27 (5) (2017) 722–736, http://dx.doi.org/10.1101/gr.215087.116.
[31] R. Vaser, et al., Fast and accurate de novo genome assembly from long uncorrected reads, Genome Res. 27 (5) (2017) 737–746, http://dx.doi.org/10.1101/gr.214270.116.
[32] D. Kim, et al., Centrifuge: rapid and sensitive classification of metagenomic sequences, Genome Res. 26 (12) (2016) 1721–1729, http://dx.doi.org/10.1101/gr.210641.116.
[33] H. Sadasivan, et al., Rapid real-time squiggle classification for read until using RawMap, Arch. Clin. Biomed. Res. 07 (01) (2023) http://dx.doi.org/10.26502/acbr.50170318.
[34] S. Kovaka, et al., Targeted nanopore sequencing by real-time mapping of raw electrical signal with UNCALLED, Nature Biotechnol. 39 (4) (2020) 431–441, http://dx.doi.org/10.1038/s41587-020-0731-9.

[35] A. Payne, et al., Readfish enables targeted nanopore sequencing of gigabase-sized genomes, Nature Biotechnol. 39 (4) (2020) 442–450, http://dx.doi.org/10.1038/s41587-020-00746-x.

[36] D.E. Wood, et al., Improved metagenomic analysis with Kraken 2, Genome Biol. 20 (1) (2019) http://dx.doi.org/10.1186/s13059-019-1891-0.

[37] R. Ounit, et al., CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers, BMC Genomics 16 (1) (2015) http://dx.doi.org/10.1186/s12864-015-1419-2.

[38] J.T. Simpson, et al., Efficient de novo assembly of large genomes using compressed data structures, Genome Res. 22 (3) (2011) 549–556, http://dx.doi.org/10.1101/gr.126953.111.

[39] A. Subramaniyan, et al., GenomicsBench: A benchmark suite for genomics, in: 2021 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2021, pp. 1–12, http://dx.doi.org/10.1109/ISPASS51385.2021.00012.

[40] G. Myers, A fast bit-vector algorithm for approximate string matching based on dynamic programming, J. ACM 46 (3) (1999) 395–415, http://dx.doi.org/10.1145/316542.316550.

[41] S. Marco-Sola, et al., Fast gap-affine pairwise alignment using the wavefront algorithm, Bioinformatics 37 (4) (2020) 456–463, http://dx.doi.org/10.1093/bioinformatics/btaa777.

[42] S. Kalikar, et al., Accelerating minimap2 for long-read sequencing applications on modern CPUs, Nat. Comput. Sci. 2 (2) (2022) 78–83, http://dx.doi.org/10.1038/s43588-022-00201-8.

[43] H. Suzuki, et al., Introducing difference recurrence relations for faster semi-global alignment of long sequences, BMC Bioinform. 19 (S1) (2018) http://dx.doi.org/10.1186/s12859-018-2014-8.

[44] N.J. Loman, et al., A complete bacterial genome assembled de novo using only nanopore sequencing data, Nature Methods 12 (8) (2015) 733–735, http://dx.doi.org/10.1038/nmeth.3444.

[45] H. Gamaarachchi, et al., GPU accelerated adaptive banded event alignment for rapid comparative nanopore signal analysis, BMC Bioinform. 21 (1) (2020) http://dx.doi.org/10.1186/s12859-020-03697-x.

[46] Nanopore wgs consortium, 2023, https://github.com/nanopore-wgs-consortium/NA12878. (Accessed 7 January 2023).

[47] M. Šošić, et al., Edlib: a C/C ++ library for fast, exact sequence alignment using edit distance, in: J. Hancock (Ed.), Bioinformatics 33 (9) (2017) 1394–1395, http://dx.doi.org/10.1093/bioinformatics/btw753.

[48] M. Rautiainen, et al., GraphAligner: rapid and versatile sequence-to-graph alignment, Genome Biol. 21 (1) (2020) http://dx.doi.org/10.1186/s13059-020-02157-2.

[49] A. Ahmadi, et al., Hobbes: optimized gram-based methods for efficient read alignment, Nucleic Acids Res. 40 (6) (2011) e41, http://dx.doi.org/10.1093/nar/gkr1246.

[50] NCBI sequence read archive, 2023, https://www.ncbi.nlm.nih.gov/sra. (Accessed 7 January 2023).

[51] K.-M. Chao, et al., Aligning two sequences within a specified diagonal band, Bioinformatics 8 (5) (1992) 481–487, http://dx.doi.org/10.1093/bioinformatics/8.5.481.

[52] Pacific biosciences dataset: Caenorhabditis elegans 40x coverage, 2023, http://datasets.pacb.com.s3.amazonaws.com/2014/c_elegans/list.html. (Accessed 7 January 2023).

[53] M.A. Eberle, et al., A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree, Genome Res. 27 (1) (2016) 157–164, http://dx.doi.org/10.1101/gr.210500.116.

[54] M. Burrows, A block-sorting lossless data compression algorithm, 1994, SRC Research Report, 124.

[55] Libcuckoo, 2023, https://github.com/efficient/libcuckoo. (Accessed 7 January 2023).

[56] Loman labs: Escherichia coli, 2023, https://zenodo.org/record/1172816/files/Loman_E.coli_MAP006-1_2D_50x.fasta. (Accessed 7 January 2023).

[57] A. Paszke, et al., PyTorch: An imperative style, high-performance deep learning library, in: H. Wallach, et al. (Eds.), Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.

[58] M. Abadi, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, 2023, Software available from tensorflow.org. (Accessed 18 July 2023).

[59] J.M. Zook, et al., Extensive sequencing of seven human genomes to characterize benchmark reference materials, Sci. Data 3 (1) (2016) http://dx.doi.org/10.1038/sdata.2016.25.

[60] Clair3 ONT model r941_prom_hac_g360+g422, 2023, http://www.bio8.cs.hku.hk/clair3/clair3_models/r941_prom_hac_g360+g422.tar.gz. (Accessed 7 January 2023).

[61] H. Li, et al., The sequence alignment/map format and SAMtools, Bioinformatics 25 (16) (2009) 2078–2079, http://dx.doi.org/10.1093/bioinformatics/btp352.

[62] C. Lee, et al., Multiple sequence alignment using partial order graphs, Bioinformatics 18 (3) (2002) 452–464, http://dx.doi.org/10.1093/bioinformatics/18.3.452.

[63] C. Lee, Generating consensus sequences from partial order multiple sequence alignment graphs, Bioinformatics 19 (8) (2003) 999–1008, http://dx.doi.org/10.1093/bioinformatics/btg109.

[64] Spoa library (SIMD POA), 2023, https://github.com/rvaser/spoa. (Accessed 30 January 2023).

[65] Wfmash: a pangenome-scale aligner, 2023, https://github.com/waveygang/wfmash. (Accessed 30 January 2023).

[66] B. Song, et al., AnchorWave: Sensitive alignment of genomes with high sequence diversity, extensive structural polymorphism, and whole-genome duplication, Proc. Natl. Acad. Sci. 119 (1) (2021) http://dx.doi.org/10.1073/pnas.2113075119.

[67] L. Pipes, et al., AncestralClust: clustering of divergent nucleotide sequences by ancestral sequence reconstruction using phylogenetic trees, in: I. Birol (Ed.), Bioinformatics 38 (3) (2021) 663–670, http://dx.doi.org/10.1093/bioinformatics/btab723.

[68] N. Stephens, et al., The ARM scalable vector extension, IEEE Micro 37 (2) (2017) 26–39, http://dx.doi.org/10.1109/mm.2017.35.

[69] L. McVoy, et al., Lmbench: Portable tools for performance analysis, in: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, ATEC '96, USENIX Association, USA, 1996, p. 23.

[70] R. Langarita, et al., Porting and optimizing BWA-MEM2 using the Fujitsu A64FX processor, IEEE/ACM Trans. Comput. Biol. Bioinform. (2023) 1–14, http://dx.doi.org/10.1109/TCBB.2023.3264514.

[71] PyTorch Arm neoverse, 2023, https://hub.docker.com/r/armswdev/pytorch-arm-neoverse. (Accessed 7 January 2023).

[72] TensorFlow Arm neoverse, 2023, https://hub.docker.com/r/armswdev/tensorflow-arm-neoverse. (Accessed 7 January 2023).

[73] D. Bruening, Efficient, Transparent, and Comprehensive Runtime Code Manipulation (Ph.D. thesis), Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.

[74] SRX020470, 2022, https://www.ncbi.nlm.nih.gov/sra/SRX020470. (Accessed 10 November 2022).

[75] SRX207170, 2022, https://www.ncbi.nlm.nih.gov/sra/SRX207170. (Accessed 10 November 2022).

[76] SRX206890, 2022, https://www.ncbi.nlm.nih.gov/sra/SRX206890. (Accessed 10 November 2022).

[77] Genome reference consortium human reference 38, 2022, http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/. (Accessed 23 July 2022).

[78] T. Odajima, et al., Preliminary performance evaluation of the fujitsu A64FX using HPC applications, in: 2020 IEEE International Conference on Cluster Computing, CLUSTER, 2020, pp. 523–530, http://dx.doi.org/10.1109/CLUSTER49012.2020.00075.

[79] J.J. Dongarra, et al., The LINPACK benchmark: past, present and future, Concurr. Comput.: Pract. Exper. 15 (9) (2003) 803–820, http://dx.doi.org/10.1002/cpe.728.

[80] S. Beamer, et al., The GAP benchmark suite, 2015, arXiv preprint arXiv:1508.03619.

[81] L. Wang, et al., BigDataBench: A big data benchmark suite from internet services, in: 2014 IEEE 20th International Symposium on High Performance Computer Architecture, HPCA, 2014, pp. 488–499, http://dx.doi.org/10.1109/HPCA.2014.6835958.

[82] D. Bader, et al., BioPerf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications, in: IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005, 2005, pp. 163–173, http://dx.doi.org/10.1109/IISWC.2005.1526013.

[83] W.R. Pearson, et al., Improved tools for biological sequence comparison, Proc. Natl. Acad. Sci. 85 (8) (1988) 2444–2448, http://dx.doi.org/10.1073/pnas.85.8.2444.

[84] K. Albayraktaroglu, et al., BioBench: A benchmark suite of bioinformatics applications, in: IEEE International Symposium on Performance Analysis of Systems and Software, 2005, ISPASS 2005, IEEE, 2005, pp. 2–9, http://dx.doi.org/10.1109/ispass.2005.1430554.

[85] BioBench2, 2023, https://github.com/reiverjohn/biobench2. (Accessed 30 January 2023).

[86] J.R. Miller, et al., Assembly algorithms for next-generation sequencing data, Genomics 95 (6) (2010) 315–327, http://dx.doi.org/10.1016/j.ygeno.2010.03.001.

[87] M. Alser, et al., From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures, Comput. Struct. Biotechnol. J. 20 (2022) 4579–4599, http://dx.doi.org/10.1016/j.csbj.2022.08.019.

[88] M. Alser, et al., Accelerating genome analysis: A primer on an ongoing journey, IEEE Micro 40 (5) (2020) 65–75, http://dx.doi.org/10.1109/MM.2020.3013728.

[89] H. Li, et al., A survey of sequence alignment algorithms for next-generation sequencing, Brief. Bioinform. 11 (5) (2010) 473–483, http://dx.doi.org/10.1093/bib/bbq015.

[90] A. Zielezinski, et al., Alignment-free sequence comparison: benefits, applications, and tools, Genome Biol. 18 (1) (2017) http://dx.doi.org/10.1186/s13059-017-1319-7.

[91] Y. Turakhia, et al., Darwin, ACM SIGPLAN Not. 53 (2) (2018) 199–213, http://dx.doi.org/10.1145/3296957.3173193.

[92] A. Nag, et al., Gencache: Leveraging in-cache operators for efficient sequence alignment, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 334–346, http://dx.doi.org/10.1145/3352460.3358308.

[93] D. Fujiki, et al., GenAx: A genome sequencing accelerator, in: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture, ISCA, 2018, pp. 69–82, http://dx.doi.org/10.1109/ISCA.2018.00017.

[94] H. Sadasivan, et al., Accelerated dynamic time warping on GPU for selective nanopore sequencing, J. Biotechnol. Biomed. 07 (01) (2024) http://dx.doi.org/10.26502/jbb.2642-91280134.

[95] T. Dunn, et al., SquiggleFilter: An accelerator for portable virus detection, in: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '21, ACM, 2021, http://dx.doi.org/10.1145/3466752.3480117.

[96] P.J. Shih, et al., Efficient real-time selective genome sequencing on resource-constrained devices, GigaScience 12 (2022) http://dx.doi.org/10.1093/gigascience/giad046.

[97] T. Robinson, et al., Hardware acceleration of genomics data analysis: challenges and opportunities, Bioinformatics (2021) 1–11, http://dx.doi.org/10.1093/bioinformatics/btab017.

Lorién López-Villellas is a Ph.D. student at the University of Zaragoza. His research focuses on exploiting novel and consolidated parallel and vector architectures for scientific applications, such as molecular dynamics and genomics. Prior to starting his Ph.D., he worked as a research engineer at the Barcelona Supercomputing Center for two years. He holds a BSc in computer science from the University of Zaragoza and a MSc in High-Performance Computing from the Universitat Politècnica de Catalunya.

Rubén Langarita-Benítez received his B.S. 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.

Asaf Badouh is a research software engineer with a passion for high-performance computing and machine learning. Over the last 4.5 years, Asaf has been working at the Barcelona Supercomputing Center as a Research Engineer, focusing on genomics and healthcare applications. Prior to that, Asaf worked for 6 years as an intern and engineer in the Intel compiler R&D team in Israel, developing LLVM-based compilers for OpenCL and C/C++. Aside from that, Asaf holds a bachelor's degree in computer science from the Technion, Israel; and a master's degree in Data Science from Universitat Politècnica de Catalunya.

Víctor Soria-Pardos received a B.Sc. in computer science from Universitat de Zaragoza, in 2019. He received an M.Sc. in computer science from Universitat Politècnica de Catalunya (UPC), in 2022. He is currently a second-year Ph.D. in computer architecture with the UPC. He also works as a researcher in the Barcelona Supercomputing Center, within the Center of Excellence partnership with Arm. His research interests include high-performance computing architectures, cache coherence, and multicore architectures.

Quim Aguado-Puig received a B.Sc. degree in computer science in 2019 from the Universitat Autònoma de Barcelona (UAB). He received an MSc at the Universitat Politècnica de Catalunya (UPC) in 2023. He is currently a first-year Ph.D. student at UAB. He has previously worked as a research engineer in the project Designing RISC-V-based Accelerators for next-generation Computers (DRAC) at UAB in collaboration with the Barcelona Supercomputing Center (BSC). His research interests include high-performance computing, massively parallel architectures, and GPU programming; with applications to genomics, computational biology, and sequence alignment.

Guillem López-Paradís received a B.Sc. and M.Sc. from Universitat Politecnica de Catalunya (UPC) in 2017 and 2020, respectively. He is currently a third year Ph.D. student in Computer Architecture at UPC and Barcelona Supercomputing Center (BSC). He actively participates in different European projects, as well as in international collaborations with academia and industry. He has already published some papers in international conferences and participated in different tapeouts designing power-efficient RISC-V processors. His research interests include high-performance computing architectures, scaling RTL Simulations, and domain-specific accelerators, with special emphasis on coherent interconnects between cores and hardware accelerators.

Max Doblas received a B.Sc. in electrical engineering and computer science from Universitat Politècnica de Catalunya (UPC), in 2020. He received an M.Sc. in computer science from UPC, in 2021. He is currently a second-year Ph.D. in computer architecture with the UPC. He also works as a Research Engineer in the project Designing RISC-V-based Accelerators for next-generation Computers (DRAC) at the Barcelona Supercomputing Center (BSC), in which he has designed a power-efficient processor with several extensions for domain-specific applications. His research interests include high-performance computing architectures, and domain-specific accelerators, with applications to genomics, computational biology, and sequence alignment.

Javier Setoain received his Ph.D. in computer science and engineering from the Complutense University of Madrid (UCM), working on workload optimization for GPUs. After working as a post-doc at the Spanish National Centre for Biotechnology (CNB), and a research engineer at Arm Research, he is currently a senior member of technical staff in AMD Research and Advanced Development, working on compiler research for AI accelerators. His research interests center around computer accelerators and specialized architectures, and his current research is focused on automated ML workload optimizations for spatial architectures.

Chulho Kim is Principal Consultant with the Lenovo Infrastructure Solutions Group Services in United States. Chulho has a Bachelor of Science degree in Mathematics of Computation from UCLA in 1989 and joined IBM Kingston. He joined Lenovo US in 2014. He has worked in High Performance Computing (HPC) since 1993. He likes to apply his skills to debug extremely complex issues, ranging from application performance to system and networking performance issues. He is responsible for running Top500/Green500 on customer clusters for Lenovo (#1 Green500 entry since November 2022).

Makoto Ono received the ME in biophysics from the Osaka University in 1985 and joined IBM Tokyo Research lab. He worked on computer graphics research then started system architecture development in IBM System x development. He is currently a Distinguished Engineer at Lenovo Infrastructure Solutions Group and is a lead architect of edge computing. His interests include edge computing, edge AI, and heterogeneous and alternative architecture including non traditional CPU / GPU architecture.

Adrià Armejach is a Lecturer Professor in Computer Architecture at Universitat Politècnica de Catalunya (UPC), and associate researcher at the Barcelona Supercomputing Center (BSC). He received his Ph.D. from UPC in 2014 and then started his research career at BSC, where he lead the technical contributions of multiple FP7 and H2020 projects. His research interest include memory systems, heterogeneous architectures, simulation methodologies and vector architectures. Currently he leads a group that oversees all the architectural simulation efforts at BSC, enabling research on multiple computer architecture topics. He has published more than 30 well-ranked international conference and journal papers.

Santiago Marco-Sola received the M.Sc. and Ph.D. degrees in computer science from the Universitat Politècnica de Catalunya (UPC) in 2012 and 2017, respectively. During his Ph.D., he worked at the Algorithm Development and Bioinformatics Group at Spanish National Centre for Genome Analysis (CNAG) and lectured at the Universitat Autònoma de Barcelona (UAB). He is currently a Senior Researcher at the Barcelona Supercomputing Center (BSC) and a Lecturer at the UPC. His research interests include high-performance computing, heterogeneous architectures, genome-data analysis, and algorithms in bioinformatics and computational biology.

Jesús Alastruey-Benedé received the M.S. degree in Telecommunication and the Ph.D. 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.

Pablo Ibáñez received the M.S. degree in computer science from the Universitat Politécnica de Catalunya, Spain, in 1989, and the Ph.D. degree in computer science from the Universidad de Zaragoza, Spain, in 1998. He is an associate professor with the Computer Science and Systems Engineering Department, University of Zaragoza. His research interests include processor microarchitecture, memory hierarchy, parallel computer architecture, and high performance computing applications.

Miquel Moretó received the B.Sc., M.Sc., and Ph.D. degrees from Universitat Politècnica de Catalunya (UPC), Spain. Currently, he is a Ramón y Cajal Fellow at UPC Barcelona. Prior to joining UPC, he spent 5 years as a Senior Researcher with the Barcelona Supercomputing Center (BSC), Spain, and 15 months as a post-doctoral fellow with the International Computer Science Institute (ICSI), Berkeley. His research interests include high performance computer architectures, domain-specific accelerators and hardware-software co-design for future massively parallel systems.