Improving Performance of Genomics Workloads through Software Optimizations and Hardware Acceleration

UPC UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Rubén Langarita Benítez

Directors: Adrià Armejach
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Jesús Alastruey Benedé
Universidad de Zaragoza

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

November 2025

Para mi madre y para mi padre, los principales motivos por los que he llegado hasta aquí.
Espero que os sintáis orgullos@s. Esta tesis es para vosotr@s.

Agradecimientos

A mi madre, por ser mi ejemplo a seguir. Porque todas las cosas buenas que pudiera decir de ti no se podrían expresar con palabras. A mi padre, por enseñarme todos los trucos de la vida, desde arreglar una bicicleta a ser mejor persona. Al próximo almuerzo invito yo, chavalín. A l@s dos, espero que estéis orgullos@s de mi.

A mi hermano, porque a pesar de todas las riñas que hemos tenido, sigues siendo el mejor hermano del mundo. Aunque ahora no nos veamos mucho, te deseo que recorras el camino que más feliz te haga.

A mi abuela Pilar, mientras llegan esos abrazos que te prometí, espero que esta tesis te haga sentir orgullosa. A mi abuelo Dionisio, porque tu forma de entender la vida pasaba por trabajar y querer la tierra hasta el último momento. Gracias por no haberte chivado al papa del día que atasqué la furgoneta en el ribazo. A mi abuelo Manolo, el cual viajó de Granada a Cataluña buscando prosperidad y acabó en Calatorao haciendo el mejor pan de Valdejalón. A mi abuela Benilde, porque siempre estás ahí y por la vida de trabajo duro que has llevado, espero que esta tesis te haga más feliz durante tu merecida jubilación. A tod@s mis abuel@s, gracias por enseñarme el valor del esfuerzo y la constancia.

A mis tí@s Miguel, Dionisio, Azucena y Pilar, aunque no nos veamos mucho, gracias por estar ahí. A mis amigos del pueblo, con los que tan buenas noches (y algunas mañanas) hemos pasado. A Joni, Tomi y Alex. Porque aunque nos separe la distancia, cada vez que nos volvemos a ver, no ha pasado ni un día. Me dais unos raticos memorables. A los colegas de la Universidad de Zaragoza. Espero que no acabe nunca la tradición de juntarnos al menos una vez al año.

A todxs lxs amigxs que he hecho estos años en Barcelona. Porque mi vida sin vosotrxs no tendría sentido. Me dais las fuerzas necesarias para seguir luchando. Gracias de corazón. A Víctor, excompañero de piso y actual compañero de trabajo. Por haber pasado unos de los mejores años de mi vida compartiendo piso contigo. Te deseo la mejor de las suertes en la vida. A toda la gente que está y ha pasado por la oficina, por esos momenticos a la hora del café. Espero seguir viéndonos después de esto. A la gente de la sección sindical y del comité de empresa del BSC, y a todas las personas del centro que luchan por unas condiciones laborales justas. Me hacéis creer en que una ciencia digna es posible.

A todxs lxs profesorxs que he tenido en mi vida, dentro y fuera del aula. Porque sin vosotrxs no estaría hoy aquí. En especial, gracias a María Jesús por apostar por mí cuando nadie más daba ni un duro y por reconducirme hacía una ingeniería en unos tiempos en los que yo no tenía nada claro. Y también, gracias a Don Jesús, el primer profesor con el que se me encendió la curiosidad y la pasión por las matemáticas. A Pablo y Chus, por ser de los mejores profesores de la universidad y por haber seguido conmigo durante el doctorado. A Javier, Miquel y a Santiago, por haberme guiado también en este proceso del doctorado y haber contribuido con grandes ideas. Y a Adrià, por haber sido un gran director de tesis. He aprendido mucho contigo. Ojalá trabajar el resto de mi vida con gente al menos la mitad de competente que tú.

Abstract

Modern multi-core architectures and accelerators have become the cornerstone for accelerating many workloads in scientific computing and engineering. Many efforts have been made to accelerate HPC applications on modern hardware architectures such as CPUs and GPUs, as well as FPGAs and custom accelerators (ASICs) for specific workloads. Hence, HPC platforms are increasingly sought after to handle large-scale workloads that exploit different levels of parallelism available in the accelerators.

However, there is an emergent class of workloads that cannot fully exploit the massively parallel capabilities of mainstream accelerators. Many HPC applications are often bottlenecked by the execution of sequential workflows composed of rather small compute-intensive kernels that implement complex dependency patterns. This is particularly noticeable in life science and healthcare applications, which implement long workflows of data-processing kernels. Often based on stencil and dynamic programming computations, these dependency-bound kernels tend to be moderate in size and implement complex data-dependency patterns that ultimately restrict parallelism exploitation.

Precision medicine aims to improve healthcare by exploiting genomic information. In recent years, the sharp reduction in genome sequencing costs has driven a dramatic increase in the amount of data generated for processing, which has posed a significant computational and storage challenge. Sequence alignment, one of the most demanding computational problems addressed in sequencing studies, has numerous applications, including read mapping. The goal of read mapping is to align the reads extracted from the sequencing systems against a reference genome. A dynamic programming scheme is used to assign an alignment score for each of the candidates, which leads to poor data parallelization due to its dependency-bound patterns.

The main objective of this work is to improve the performance of genomic workloads through software and hardware acceleration. We present four contributions to the field. The first three are software enhancements, including an algorithm proposal, software optimizations, and kernel porting to the ARM architecture. In the last one, we expand our field of study and propose a new hardware accelerator for dependency-bound kernels, which targets dynamic programming algorithms used in genomic pipelines.

Resumen

Las arquitecturas multinúcleo y los aceleradores modernos se han convertido en la piedra angular para acelerar muchas cargas de trabajo en aplicaciones científicas y de ingeniería. Se han realizado muchos esfuerzos para acelerar aplicaciones HPC en arquitecturas hardware modernas como las CPUs y las GPUs, así como las FPGAs y los aceleradores a medida (ASICs) para cargas de trabajo específicas. De ahí que cada vez se busquen más plataformas HPC para manejar cargas de trabajo a gran escala que exploten los distintos niveles de paralelismo disponibles en los aceleradores.

Sin embargo, hay una clase emergente de cargas de trabajo que no pueden explotar plenamente las capacidades de paralelismo masivo de los aceleradores convencionales. Muchas aplicaciones HPC se ven a menudo limitadas por ejecuciones secuenciales compuestas de pequeñas secciones de cálculo intensivo que incluyen patrones complejos de dependencias. Esto es especialmente notable en las aplicaciones de ciencias de la vida y de la salud, que implementan largos flujos de trabajo de procesamiento de datos. A menudo basados en cálculos *stencil* y de programación dinámica, estos algoritmos *dependency-bound* tienden a ser de tamaño moderado e implementan patrones complejos de dependencias de datos que, en última instancia, restringen el aprovechamiento del paralelismo.

La medicina de precisión pretende mejorar la asistencia sanitaria utilizando datos genómicos. En los últimos años, la fuerte reducción de los costes de la secuenciación genómica ha impulsado un aumento espectacular de la cantidad de datos generados para su procesamiento, lo que ha planteado un importante reto computacional y de almacenamiento. La alineación de secuencias, uno de los problemas computacionales más exigentes abordados en los estudios de secuenciación, tiene numerosas aplicaciones, entre ellas el mapeo de *reads*. El objetivo del mapeo de *reads* es alinear las lecturas extraídas de los sistemas de secuenciación contra un genoma de referencia. Se utiliza un esquema de programación dinámica para asignar una puntuación de alineamiento a cada uno de los candidatos, lo que conduce a una paralelización sub-optima de los datos debido a los patrones *dependency-bound*.

El objetivo principal de este trabajo es mejorar el rendimiento de las aplicaciones genómicas mediante la aceleración software y hardware. Presentamos cuatro contribuciones a este campo. Las tres primeras son mejoras software, incluida la propuesta de un algoritmo, optimizaciones software y la adaptación de varios algoritmos a la arquitectura ARM. En la última, ampliamos nuestro campo de estudio y proponemos un nuevo acelerador hardware para algoritmos *dependency-bound*, el cual aborda algoritmos de programación dinámica utilizados en *pipelines* genómicos.

Resum

Les arquitectures multinucli i els acceleradors moderns s'han convertit en la pedra angular per a accelerar moltes càrregues de treball en aplicacions científiques i d'enginyeria. S'han fet molts esforços per a accelerar aplicacions HPC en arquitectures hardware modernes com les CPUs i les GPUs, així com les FPGAs i acceleradors a mida (ASICs) per a càrregues de treball específiques. D'aquí ve que cada vegada es busquin més plataformes HPC per a manejar càrregues de treball a gran escala que explotin els diferents nivells de paral·lelisme disponibles als acceleradors.

No obstant això, hi ha una classe emergent de càrregues de treball que no poden explotar plenament les capacitats de paral·lelisme massiu dels acceleradors convencionals. Moltes aplicacions HPC es veuen sovint limitades per execucions seqüencials compostes de petites seccions de càlcul intensiu que inclouen patrons complexos de dependències. Això és especialment notable en les aplicacions de ciències de la vida i de la salut, que implementen llargs fluxos de treball de processament de dades. Sovint basats en càlculs *stencil* i de programació dinàmica, aquests algorismes *dependency-bound* tendeixen a ser d'una mida moderada i implementen patrons complexos de dependències de dades que, en última instància, restringeixen l'aprofitament del paral·lelisme.

La medicina de precisió pretén millorar l'assistència sanitària utilitzant dades genòmiques. En els últims anys, la forta reducció dels costos de la seqüenciació genòmica ha impulsat un augment espectacular de la quantitat de dades generades per al seu processament, la qual cosa ha plantejat un important repte computacional i d'emmagatzematge. L'alineació de seqüències, un dels problemes computacionals més exigents abordats en els estudis de seqüenciació, té nombroses aplicacions, entre elles el mapatge de *reads*. L'objectiu del mapatge de *reads* és alinear les lectures extretes dels sistemes de seqüenciació contra un genoma de referència. S'utilitza un esquema de programació dinàmica per a assignar una puntuació d'alineament a cadascun dels candidats, la qual cosa condueix a una paral·lelització sub-optima de les dades a causa dels patrons *dependency-bound*.

L'objectiu principal d'aquest treball és millorar el rendiment de les aplicacions genòmiques mitjançant l'acceleració software i hardware. Presentem quatre contribu-

cions a aquest camp. Les tres primeres són millores software, inclosa la proposta d'un algorisme, optimitzacions software i l'adaptació de diversos algorismes a l'arquitectura ARM. En l'última, ampliem el nostre camp d'estudi i proposem un nou accelerador hardware per a algorismes *dependency-bound*, el qual aborda algorismes de programació dinàmica utilitzats en *pipelines* genòmics.

Table of contents

List of figures xix

List of tables xxv

1 Introduction 1

1.1 Objectives and Contributions 2

1.1.1 COFI 2

1.1.2 Porting and Optimizing BWA-MEM2 2

1.1.3 GenArchBench 3

1.1.4 Squire 3

1.2 Thesis Structure 4

1.3 Key Results 4

2 Background 7

2.1 Sequencing Systems 7

2.2 Genome Data Analysis Pipelines and Tools 8

2.2.1 Genome resequencing 9

2.2.2 Genome assembly 10

2.2.3 Metagenomics 11

2.3 Read Mapping Pipelines 11

2.3.1 Seeding stage 12

2.3.2 Filtering: Chain 16

2.3.3 Extend Stage: Align Algorithms 18

2.4 Dependency-Bound Use Cases 19

2.4.1 Data Sorting 20

2.4.2 Signal Processing 20

2.5 Genomic Hardware Accelerators 21

3 Methodology 23
3.1 Read mapping tools 23
3.1.1 BWA-MEM2 23
3.1.2 Minimap2 26
3.2 Target Hardware Platforms 27
3.3 Simulation Platform: Gem5 29
3.4 Inputs: Genomes and Sequences 30

4 COFI: Compressed FM-Index for Large K-steps 31
4.1 State-of-the-Art FM-Index implementations 32
4.1.1 Sampled FM-Index 32
4.1.2 K-step Sampled FM-Index 33
4.1.3 bvSFM: Improving Data Locality 33
4.1.4 State-of-the-Art Summary 35
4.1.5 Memory Footprint Analysis 36
4.2 COFI: Compressed Sparse FM-Index 37
4.2.1 COFI Data Structures: Offsets and Changes 37
4.2.2 Memory Footprint Analysis 38
4.2.3 Backward Search in COFI 39
4.2.4 COFI Example 40
4.2.5 Performance Optimizations 41
4.3 Performance Analysis 42
4.4 Evaluation 45
4.4.1 Inputs Sequences 45
4.4.2 Optimizations 47
4.4.3 Throughput 50
4.4.4 Input Sensitivity 51
4.4.5 Discussion 53

5 Porting and Optimizing BWA-MEM2 using the Fujitsu A64FX Processor 55
5.1 Target Machine: Fujitsu's A64FX 57
5.1.1 Core Out-of-Order Resources 58
5.1.2 The A64FX Memory Hierarchy 58
5.1.3 Memory Hierarchy Latencies 59
5.1.4 Memory Bandwidth 60
5.2 Porting the code to ARM architecture 61
5.2.1 Data Types 61

5.2.2 Porting Effort 62
5.2.3 *add* Translation Example 62
5.2.4 Boolean Translation Example 63
5.2.5 Challenges 64
5.3 Optimizations 66
5.3.1 General Optimizations 66
5.3.2 SMEM Region 67
5.3.3 BSW Region 69
5.4 Experimental Results 70
5.4.1 Input Sequences 70
5.4.2 A64FX Optimizations Evaluation 70
5.4.3 SVE Vector Length Analysis 73
5.4.4 Scalability Analysis 74
5.4.5 Comparison with SKX 74
5.4.6 Discussion 77

6 GenArchBench: A Genomic Benchmark Suite for ARM HPC Processors 79
6.1 GenArchBench Benchmark Suite 80
6.1.1 Adaptive Banded Signal to Event Alignment (ABEA) 81
6.1.2 Bit-Parallel Myers (BPM) 82
6.1.3 Banded Smith-Waterman (BSW) 82
6.1.4 Seed Chaining (CHAIN) 82
6.1.5 SIMD Seed Chaining (FAST-CHAIN) 83
6.1.6 De Bruijn Graph Construction (DBG) 83
6.1.7 FM-Index Search (FMI) 83
6.1.8 K-mer Counting (KMER-CNT) 83
6.1.9 Neural Network-based Base Calling (NN-BASE) 84
6.1.10 Neural Network-based Variant Calling (NN-VARIANT) 84
6.1.11 Pileup Counting (PILEUP) 85
6.1.12 Partial-Order Alignment (POA) 85
6.1.13 Wavefront Alignment (WFA) 85
6.2 ARM Porting of Genomic Kernels 86
6.2.1 BSW 86
6.2.2 FAST-CHAIN 87
6.2.3 FMI 88
6.3 Performance Characterization 88
6.3.1 Single-Thread Performance 89

6.3.2 Parallel Performance 90
6.3.3 Instruction Mix Comparison 92
6.3.4 Microarchitecture Bottleneck Analysis 93
6.3.5 Energy Consumption 95
6.3.6 Evaluation of a Real Genomic Tool 97
6.3.7 Summary 98

7 Squire: A General-Purpose Accelerator to Exploit Fine-Grain Parallelism on Dependency-Bound Kernels 101

7.1 Dependency-Bound Fine-Grain Parallelism 102
7.2 Squire 104
7.2.1 Squire Design 104
7.2.2 Synchronizing Workers 106
7.2.3 Squire API 107
7.3 Using Squire 107
7.3.1 Sorting: Radix Sort 107
7.3.2 1D Dynamic Programming: Chain 108
7.3.3 2D Dynamic Programming: DTW 111
7.3.4 2D Dynamic Programming: WFA 111
7.3.5 Discussion 112
7.4 Evaluation Methodology 113
7.4.1 Architectural Simulation 113
7.4.2 Workloads and Inputs 113
7.4.3 Evaluation of an End-to-End Read-Mapping Application 114
7.5 Evaluation 115
7.5.1 Performance Evaluation 115
7.5.2 Synchronization Module Evaluation 116
7.5.3 End-to-End Application Evaluation 117
7.5.4 Cache Size Exploration 117
7.5.5 Area Overhead 118
7.5.6 Energy Consumption 119
7.6 Comparison with State-of-the-Art 119
7.6.1 Accelerating Smith-Waterman 119
7.6.2 General-Purpose Accelerators 121

8 Conclusions and Future Work 123
8.1 Future Work 125
8.2 Publications 126

References 129

List of figures

2.1 Workflow diagram of common genome analysis pipelines. Going from (1) sequencing, through (2) basecalling, to (3.a) genome resequencing, (3.b) genome assembly, or (3.c) metagenomics. The figure shows the different computational kernels used within each stage or tool. This thesis focuses on the tools and algorithms of the read mapping process (3.a.1). 8

2.2 Genomic pipeline with the throughput of each phase. Image source: [1]. 11

2.3 Read-mapping pipeline. This figure is a zoomed-in view of the region highlighted by a dashed line in Figure 2.1. 12

2.4 Procedure to build the FM-index. (a) Step 1: cyclic rotations to generate matrix M. (b) Steps 2 and 3: sort M alphabetically and extract the last column (BWT) and the suffix array (SA). (c) Step 4: build C and Occ structures. 14

2.5 Backward search example. (a) *sp* and *ep* initialization. (b) LF operation for *sp* and *ep*. (c) Final state of the query. 15

2.6 Chain algorithm dependencies. To calculate the score of anchor 4 ($f(4)$), we add all the previous scores to row 4 and perform a maximum. In the next iteration, $f(4)$ will be added to row 5. 16

2.7 Graphical representation of the anchors generated by an execution. Each point represents an anchor, i.e., a match of k base pairs (15 in this example) between the sequence and the reference. All the anchors form a *chain*. . 17

2.8 Cell computation dependencies in Smith–Waterman algorithm. Image source: [2]. 18

2.9 DTW matrix representation. The samples from signal 1 are set at the top of the matrix, while samples from signal 2 are set at the left. For each cell DTW computes Equation 2.5. Arrows indicate the minimum value from Equation 2.5. 21

3.1 SMEM process scheme. 24

3.2 The 3-stage Minimap2 pipeline. For each stage, the most time-consuming kernel is indicated. 26

4.1 Several FM-Index implementations. (a) Full FM-Index with reference length n. (b) Sampled FM-Index, one out of d counters is stored. (c) 2-step sampled FM-Index, uses an alphabet of 16 symbols. (d) Split bit-vector FM-Index, each bit indicates whether a base is in a given BWT position or not. 33

4.2 *bvSFM* column for the symbol G. Each *bvSFM* entry has a counter (blue boxes, left) and a bit vector (right, 4 bits). $C[G] = 9$ is already added to the counters, so there is no need to add it during the LF operation as in Equation 2.3. 34

4.3 Size, useful, and redundant data of the *bvSFM* structure for $d = 64$ and different values of k. 36

4.4 *COFI* data structures: Offsets and Changes. 37

4.5 *COFI* data structures size for various values of k. 38

4.6 Histogram of column sizes for the human reference GRCh38. Both axis are in logarithmic scale. 38

4.7 Depicts four interleaved sequences to hide memory latency. *OP* denotes the time spent computing the limits of the columns, *MEM* is the time spent waiting for a response from memory, and *BS* stands for the time spent on binary search. *BS* depends on column size, therefore, some sequences finish this phase before others (red boxes). 41

4.8 Search intensity for different column sizes. 43

4.9 Percentage of accesses per column size. 44

4.10 Search intensity for each input. 45

4.11 Average performance across all inputs for different numbers of interleaved sequences. 47

4.12 Speed-up when using software prefetching. 48

4.13 Speed-up when employing conditional moves in the binary search algorithm. 49

4.14 Speed-up of COFI with respect to *bv*SFM for all inputs, two references, and two test machines. 50

4.15 Execution time spent on each column size. 52

4.16 Roofline models for *bvSFM* (input independent) version and three inputs of COFI. 53

5.1 A64FX scheme. We can distinguish the four different core memory groups (NUMA domains), each one with 12 cores, and four HBM2 stacks. Everything is in the same package. 56

5.2 SKX scheme. The system is composed of two Intel Xeon Skylake Platinum 8160 and two DDR4 memories interconnected in the same board. 57

5.3 Memory latencies measured by the *lmbench* benchmark for A64FX and SKX. 59

5.4 Interleaving four SMEM processes for different input reads. 68

5.5 Speed-ups when using large pages with respect to not using them on A64FX with 48 threads. Showing data for D3, D4, and D5 inputs and the three code sections. 71

5.6 Speed-ups for (a) in-line, (b) population count and (c) predication optimizations for D3, D4, and D5 inputs on A64FX with 48 threads. 72

5.7 Average speedups for the interleaved sequences optimization on SMEM for D3, D4, and D5 inputs on A64FX with 48 threads. 72

5.8 A64FX normalized execution time with respect to the unoptimized baseline. Showing executions with 1, 12, 24 and 48 threads using D3, D4, and D5 inputs. 73

5.9 BSW region speedup for 48-thread executions with respect to single-threaded and SVE 128 bits for D3, D4, and D5 inputs. 74

5.10 Per region and total *average* speedups with respect to one thread for different thread counts using inputs D3, D4, and D5. 75

5.11 Average speedups with respect to one thread in A64FX for each region and for the entire execution using 24 and 48 threads using D3, D4, and D5 inputs. 76

5.12 Energy-to-solution comparison of the A64FX and SKX systems for the entire region of interest using D3, D4 and D5 inputs. 77

6.1 Speedup of SIMD kernels over their scalar version on the experimental setup using the large inputs. 87

6.2 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. 89

6.3 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. We also show ideal (linear) parallel speedup (when using *x* threads, ideally we expect a speedup of *x*) in gray. 91

6.4 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. 93

6.5 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. 95

6.6 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. 96

6.7 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). . . 97

6.8 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. We also show ideal (linear) parallel speedup (when using x threads, ideally we expect a speedup of x) in gray. 98

7.1 Examples of coarse-grain tasks with fine-grain parallelism: (a) sorting, (b) dynamic programming matrix, (c) sparse matrix-vector multiplication. Each color in the data structures represents a chunk of fine-grain work. 103

7.2 Squire architectural overview: (a) the simulated multi-core system with a 4x4 NoC and four memory controllers, where each central router holds a core complex and one slice of the L3 cache; (b) a core complex contains one OoO core with private L1 caches, a private L2 cache, and a Squire; (c) Squire contains a set of workers, several control registers, a synchronization module, and an arbiter to communicate with the memory hierarchy (L2). 105

7.3 DTW work distribution among workers. Worker 0 (W_0) computes columns 0 and 1, worker 1 (W_1) columns 2 and 3, worker 2 (W_2) columns 4 and 5, and worker 3 (W_3) columns 6 and 7. The workers compute the cells following the path indicated by the arrows. 111

7.4 Squire evaluation for the five kernels described in Section 7.4.2. We evaluate Squire with 4, 8, 16, and 32 workers. 115

7.5 Squire performance evaluation when using the synchronization module vs the pthread library with 4, 8, and 16 workers. 116

7.6 Squire evaluation for the end-to-end read-mapping application described in Section 7.4.3 with 4, 8, 16, and 32 workers. 117

7.7 Misses per kilo instructions (MPKI) when changing the cache size for the instruction and data caches with 16 workers. 118

7.8 Energy consumption comparison between the baseline and when using Squire with 16 workers for the end-to-end application. 119

8.1 Summary of algorithms, tools, hardware platforms, simulators, and sequencing technologies used in the four contributions of this thesis. . . . 125

List of tables

2.1 Sequencing systems generations. 9

3.1 Target hardware platform specifications. 27
3.2 Simulated architectural parameters. 29
3.3 Details for the reference genomes. 30

4.1 Error rates for Mason inputs. 46
4.2 Number of sequences, length of each sequence and occurrences in GRCh37 and GRCh38 for each input. 46
4.3 Raw throughput in GLFOPs/sec using GRCh38. 51

5.1 *lmbench* bandwidth benchmark results in GB/s for A64FX and SKX. . . 60
5.2 Translations in the `sse2sve.h` file. 63
5.3 Execution time relative standard deviation for 10 runs using D3, D4 and D5 inputs (see Section 3.4) and the three code regions, with and without the thread affinity. 65
5.4 Details for D3, D4 and D5 input datasets. 70
5.5 Billions of committed instructions for a single threaded execution of BSW, and instruction reductions with respect to SVE 128. 74

6.1 Summary of the evaluation. *SIMD speedup w.r.t. scalar* shows the speedup when using the SIMD unit with respect to using the scalar unit (Figure 6.1). The SIMD units are as follows: A64FX has SVE512 (512 bits), Graviton 3 has SVE256 (256 bits), SKX has AVX-512 (512 bits), and Rome has AVX2 (256 bits). *Single thread speedup w.r.t. A64FX* shows the speedup with respect to the A64FX system when using one thread (top of Figure 6.2). *Parallel speedup w.r.t. A64FX* shows the speedup with respect to the A64FX system when using all the threads in the system (bottom of Figure 6.2). A64FX and SKX have 48 threads, while Graviton 3 and Rome have 64 threads. *Parallel efficiency* shows the ratio *S/I*, where *S* is the speedup when using all threads in the system with respect to using only one, and *I* is the ideal (linear) parallel speedup (Figure 6.3). *Most used instruction type* shows the most used instruction type and its percentage (Figure 6.4). *Bottleneck analysis* shows the percentage of each stall type (Figure 6.5). *BE* is back-end stall, *FE* is front-end stall, *BS* is bad speculation stall, and *UW* is useful work. *Norm. energy consum.* shows the energy consumed by each system normalized to the energy consumption of the A64FX (Figure 6.6). 100

7.1 Squire programming interface. For each API call, we provide a brief description and who can use the API call. 107
7.2 Size of the datasets used in the evaluation. 113
7.3 Input sequence datasets. 114
7.4 Squire comparison with other proposals to accelerate Smith-Waterman. The table includes the alignment model, the hardware platform used, gigacells computed per second (GCUPS), GCUPS per area (GCUPS/mm^2), GCUPS per watts (GCUPS/W), and where these results were extracted from. PIM is Processing-in-Memory, DSA is Domain-Specific Accelerators, and GPA is General-Purpose Accelerator. 120
7.5 Qualitative comparison among several general-purpose accelerators. . . 121

Chapter 1

Introduction

Modern multi-core architectures and accelerators have become the cornerstone for accelerating many workloads in scientific computing and engineering [3]. Many efforts have been made to accelerate HPC applications on modern hardware architectures such as CPUs and GPUs, as well as FPGAs and custom accelerators (ASICs) for specific workloads [4]. Hence, HPC platforms are increasingly sought after to handle large-scale workloads that exploit different levels of parallelism available in the accelerators.

However, there is an emergent class of workloads that cannot fully exploit the massively parallel capabilities of mainstream accelerators. Many HPC applications are often bottlenecked by the execution of sequential workflows composed of rather small compute-intensive kernels that implement complex dependency patterns (dependency-bound). This is particularly noticeable in life science and healthcare applications, which implement long workflows of data-processing kernels [5]. Often based on stencil and Dynamic Programming (DP) computations, *dependency-bound* kernels tend to be moderate in size and implement complex data-dependency patterns that ultimately restrict parallelism exploitation.

Precision medicine aims to improve healthcare by exploiting genomic information [6]. In recent years, the sharp reduction in genome sequencing costs has driven a dramatic increase in the amount of data generated for processing, which has posed significant computational and storage challenges [7]. Sequence alignment, one of the most demanding computational problems addressed in sequencing studies, has numerous applications, including read mapping. The goal of read mapping is to align the reads extracted from the sequencing systems against a reference genome, i.e., for each read the objective is to find the best matching locations when compared to a reference genome [8].

The time required for DNA sequencing and data processing remains a critical factor in both research and clinical applications. Depending on the technology, sequencing can take from several hours to multiple days [9, 10]. These delays can hinder timely diagnosis, slow research progress, and limit the responsiveness of forensic or epidemiological investigations. Therefore, optimizing sequencing workflows is essential to reduce processing times without compromising data quality.

In order to restrict the search space, a common strategy is to use a seed-and-extend approach [11]. Reads are partitioned into small pieces that are searched using exact matching in order to find seeds in the reference genome. Then, a dynamic programming scheme, typically based on the Smith-Waterman algorithm, is used to assign an alignment score for each of the candidates [12, 13].

1.1 Objectives and Contributions

The main goal of this work is to improve the performance of genomic workloads through software and hardware acceleration. We present four contributions to the field. The first three are software enhancements, including an algorithm proposal, software optimizations, and kernel porting to the ARM architecture. In the last one, we expand our field of study and propose a new hardware accelerator for dependency-bound kernels, including the dynamic programming algorithms used in genomic pipelines. Each of the four proposals is explained in more detail below.

1.1.1 COFI

COFI is a **CO**mpressed **FM**-**I**ndex for large k-steps. K-steps define how many symbols (base pairs in genomics) can be searched in a single iteration [14]. COFI enables a 15-step FM-index requiring less than 16 GB for a human genome reference of 3 giga base pairs. An algorithm based on this new layout is evaluated on both a Knights Landing (KNL) and a Skylake-based system (SKX). We achieve average speedups of 1.46× and 1.39×, respectively, with respect to a state-of-the-art FM-index implementation that is already well optimized. COFI is available at https://gitlab.bsc.es/rlangari/cofi.

1.1.2 Porting and Optimizing BWA-MEM2

We port BWA-MEM2 [15] to the ARM architecture using the ARMv8-A specification, and we compare the resulting version with an Intel Skylake system in terms of both performance and energy-to-solution. The porting effort entails numerous code modifications, since BWA-MEM2 implements certain kernels using x86_64 specific intrinsics, e.g., AVX-

512. To adapt this code, we use the recently introduced ARM Scalable Vector Extensions (SVE). More specifically, we use Fujitsu's A64FX processor, the first to implement SVE. The A64FX powers the Fugaku Supercomputer, which led the Top500 ranking from June 2020 to November 2021. After porting BWA-MEM2 we define and implement several optimizations to improve performance in the A64FX target architecture. We show that while the A64FX performance is lower than that of the Skylake system, A64FX delivers 11.6% better energy-to-solution on average. All the code used in this contribution is available at https://gitlab.bsc.es/rlangari/bwa-a64fx.

1.1.3 GenArchBench

GenArchBench is a benchmark suite that consists of 13 computationally-demanding CPU kernels from the most widely-used genomic tools. This work introduces code adaptations and optimizations for the genomic kernels targeting ARM HPC CPUs. 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). 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.

This work has been performed in collaboration with a research team coordinated by Lorién López-Villellas. I have contributed to this work by porting and optimizing three kernels. GenArchBench is available at https://github.com/LorienLV/genarchbench/releases/tag/1.0.0.

1.1.4 Squire

Squire is a general-purpose accelerator designed to effectively exploit fine-grain parallelism on dependency-bound kernels. Our proposal incorporates one Squire per core in a typical multi-core system, connecting it to the memory hierarchy to directly access the virtual memory space. Each core controls one Squire, rapidly offloading workloads when necessary. We evaluate Squire on a simulated multicore SoC, obtaining speedups of up to 7.64× in dynamic programming kernels, and an acceleration for an end-to-end application of 3.66×. We also evaluate resource usage, showing that Squire reduces energy consumption by up to 56% with an area overhead of 10.5% per core.

1.2 Thesis Structure

The contents of this thesis are organized as follows:

- Chapter 2 presents the background and state-of-the-art in genomics, such as sequencing systems, algorithms, and hardware.
- Chapter 3 presents the experimental methodology used within the four contributions of this thesis. It includes applications, hardware platforms, and inputs.
- Chapter 4 presents COFI, the first contribution of the thesis. COFI is a new data layout and an alternative algorithm for the FM-Index structure used in exact matching algorithms.
- Chapter 5 presents the second contribution of the thesis, which consists of several software optimizations and the porting of the read mapping application BWA-MEM2.
- Chapter 6 presents the third contribution of the thesis. This work is a project that comprises the porting and the evaluation of several kernels to the ARM architecture. Among all the kernels, I contributed to the porting of three of them.
- Chapter 7 presents Squire, the last contribution of the thesis. Squire is a general-purpose accelerator designed to effectively exploit fine-grain parallelism on dependency-bound kernels.
- Chapter 8 concludes by summarizing the contributions of this thesis, listing the publications resulting from it and considering what future potential research directions it suggests.

1.3 Key Results

In this thesis, we focus on the acceleration of genomic workloads. We did this in a progressive manner. First, we propose a new algorithm (COFI); then, we perform system-level optimizations (BWA-MEM2 on ARM) and benchmarking (GenArchBench); and finally, incorporating all the lessons learned, we design a novel hardware architecture to address the bottlenecks of the genomic tools.

Along this journey, we found that genomic kernels have several distinctive characteristics. First, the large amount of data requires systems with enough memory to process this data. The human genome consists of 3 Gbase-pairs which translates into an ASCII file of 3 GB and other auxiliary structures that can occupy up to 50 GB. In addition to this, the dependency-bound algorithms and the random access patterns make the traditional parallel strategies, such as GPUs and SIMD, inefficient in terms of energy and resource utilization.

We conclude the thesis by proposing Squire, a hardware accelerator that exploits everything learned from the other three contributions. Moreover, we expand the scope of Squire to target other dependency-bound kernels, such as algorithms used in signal processing and sorting.

Chapter 2

Background

This chapter presents the state of the art in the field of genomics. First, sequencing systems are outlined. Then, the execution environment for genomic applications is described, and the main tools are listed, pointing out their main strengths and limitations. Finally, the execution phases of read mapping pipelines are detailed. As we progressed in our research, we found it interesting to expand the use cases beyond genomics. We introduce two additional use cases in our last contribution: data sorting and signal processing. Finally, proposals for hardware acceleration for genomics are presented.

2.1 Sequencing Systems

The first step to obtain the DNA of an individual is to introduce a sample into a sequencing system (Figure 2.1-1). Three generations of sequencing systems are usually distinguished. The first generation uses the Sanger sequencing method, which was used to sequence the first human genome in 2003. These systems are being replaced by the second, third, and fourth generations.

The second generation is able to read fixed length sequences, usually around 100 base-pairs (bps), with a high throughput. The third and fourth generations read variable length sequences, usually with a size in the order of kilobase-pairs or megabase-pairs, however, their throughput is much lower. The quality of the third generation read sequences was extremely low some years ago [16], which has been mitigated by reading the same chunk multiple times, and then going through a consensus process, leading to final read sequences that have higher quality. This technology was named High Fidelity (HiFi). Nowadays, Oxford Nanopore and PacBio have improved their process achieving an accuracy above 99%. Table 2.1 shows the attributes for each sequencer system generation.

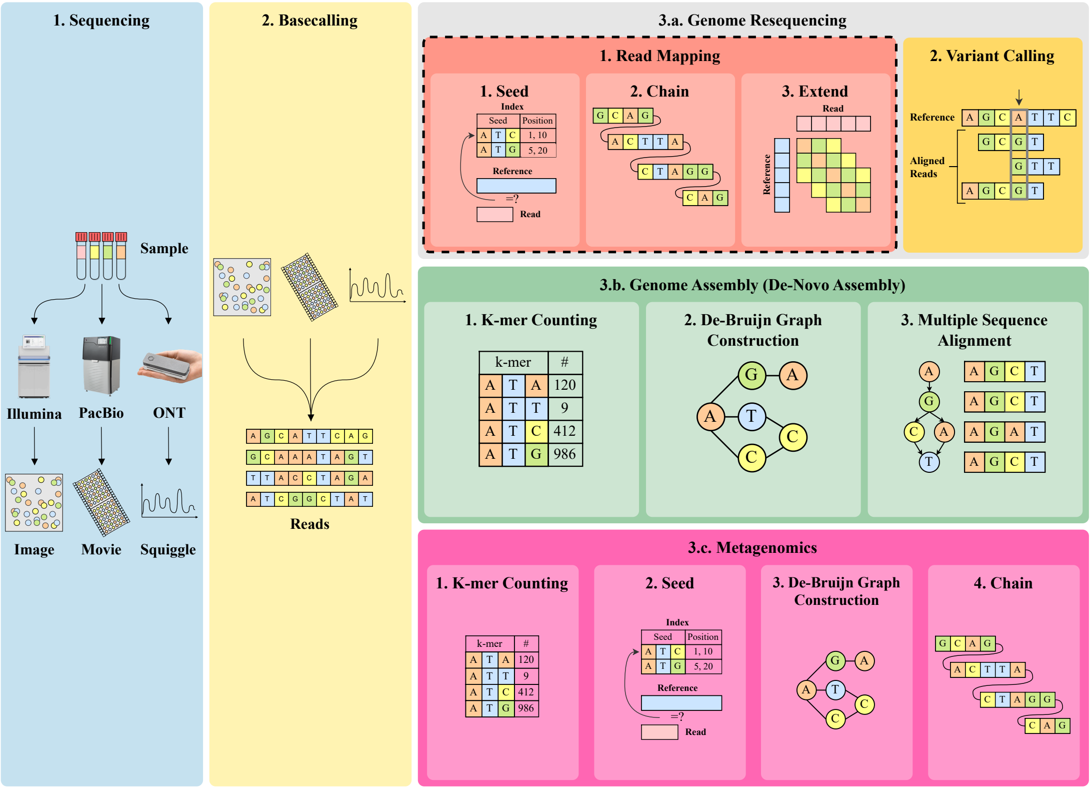


Figure 2.1: Workflow diagram of common genome analysis pipelines. Going from (1) sequencing, through (2) basecalling, to (3.a) genome resequencing, (3.b) genome assembly, or (3.c) metagenomics. The figure shows the different computational kernels used within each stage or tool. This thesis focuses on the tools and algorithms of the read mapping process (3.a.1).

2.2 Genome Data Analysis Pipelines and Tools

Before any processing can be performed, the sequencing systems' raw signals must be transformed into sequences of nucleotides (A, C, G, T). This process is called basecalling (Figure 2.1-2). Typically, a specialized basecalling tool is used to perform this process tailored to each sequencing technology.

For Illumina sequencing, each DNA fragment is replicated 1000 times. Then, two nucleotides are labeled with the same fluorescent dye and added to the fragment. This process is done twice, with green and red fluorescent dyes. For each dye, a photo is taken, and depending on whether light is emitted, the system can interpret which base is being read: adenine, cytosine, guanine, or thymine [17].

For Oxford Nanopore Technologies, each DNA fragment passes through a nanoscale protein pore (nanopore) that has an electrical current passing through it. The machine measures changes in the electrical current. Then, neural network models are used to

Table 2.1: Sequencing systems generations.

Generation	System manufacturers	Read lengths	Read accuracy	Throughput
First	Applied Biosystems	400-900 bps	99.7%	20 bps/sec
Second	Illumina	100-400 bps	99.9%	40 Mbps/sec
Third	Oxford Nanopore	20 kbps - 2 Mbps	98%	1 Mbps/sec
Third/Fourth	PacBio	1-25 kbps	99.9%	1 Mbps/sec

convert these signals into bases. Bonito [18] and Guppy [19] are two of the most widely used tools for basecalling Oxford Nanopore's raw-signal output.

For PacBio sequencing, a movie with fluorescent light pulses is recorded. The PacBio basecalling workflow is called circular consensus sequencing (CCS)[20], which includes KSW2 [21] and Edlib [22] among other steps.

Once the sequences of nucleotides have been 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 (Figure 2.1-3.a) or genome assembly (Figure 2.1-3.b). Both analyses seek to reconstruct the sample's genome by putting together all the sequenced reads.

2.2.1 Genome resequencing

The most common approach to reconstruct a sample's genome is resequencing (Figure 2.1-3.a), which involves reconstructing the sample's genome using a previously known reference genome. To do this, each sequenced read is located and matched to the most likely originating position in the reference genome, allowing small differences such as mismatches, insertions, and deletions (Figure 2.1-3.a.1). Then, a post-processing step determines the variants and mutations between the donor's genome and the reference genome (Figure 2.1-3.a.2).

The process of locating the reads on a reference is called read mapping (Figure 2.1-3.a.1), and it is implemented by many tools, such as: BWA-MEM2 [15, 23], Minimap2 [24], Bowtie2 [25, 26], and GEM [27]. Read mapping is one of the most computationally expensive steps in all genome sequencing pipelines. Consequently, read mapping has been extensively studied and optimized.

Most read mappers are based on the seed-chain-extend technique (Figure 2.1-3.a.1). This technique implements three algorithmic steps to swiftly locate and align a sequence with a reference genome. During the first step, known as seeding (Figure 2.1-3.a.1.1), the mapper searches small subsequences of the reads (seeds) on the reference, leveraging an index structure. The most commonly used indexes for seeding are FM-Index [28] and hash tables [11, 29]. Seeding reduces the potential number of locations in the reference where a sequence can match, decreasing the amount of work performed in subsequent steps. Subsequently, a chaining step (Figure 2.1-3.a.1.2) is performed to further reduce the list of possible matching locations in the reference. During the chaining step, all mapped seeds are processed to find a colinear chain of seeds that can potentially match the input sequence. Finally, during the extension or alignment step (Figure 2.1-3.a.1.3), the input sequence is aligned against the candidate location in the reference genome, identifying the differences between the donor's sequence and the reference genome. Usually, a dynamic programming-based algorithm, such as Needleman-Wunsch [30] or Smith-Waterman-Gotoh [31, 32], is used to compute the alignment.

After sequence mapping, once the reads are located in the reference genome, a variant calling algorithm (Figure 2.1-3.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 [33], Platypus [34], Clair [35, 36], DeepVariant [37] and Medaka [38].

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, de novo genome assembly (Figure 2.1-3.b) is used to reconstruct the donor's genome from scratch by jigsawing the sequenced reads together.

The 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 for each sequence's k-mer (i.e., a substring 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 (Figure 2.1-3.b.1) to prune the least frequent ones (likely artefacts of the sequencing process). Afterwards, the de Bruijn graph is constructed (Figure 2.1-3.b.2). The consensus sequence is then derived using multiple sequence alignment (MSA) algorithms (Figure 2.1-3.b.3) and the constructed de Bruijn graph. Notable examples of de Bruijn graph based assemblers are Flye [39], Canu [40], and Racon [41].

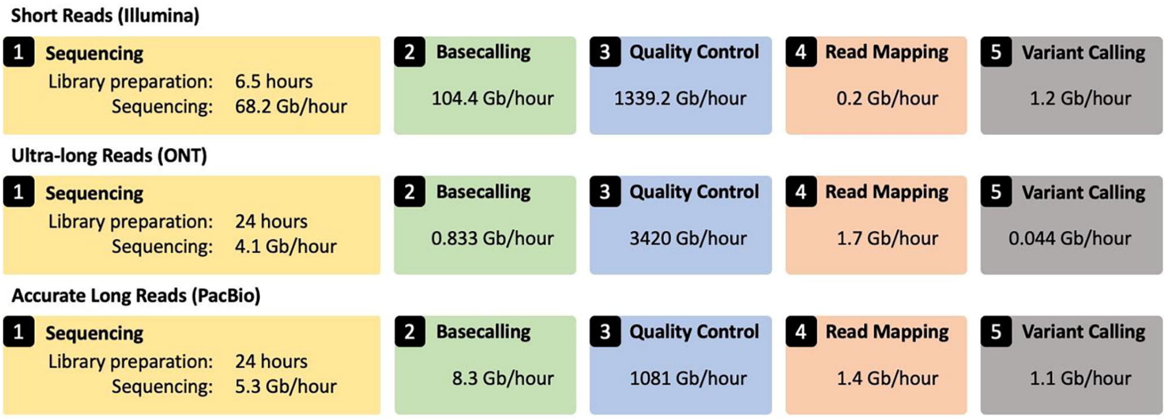


Figure 2.2: Genomic pipeline with the throughput of each phase. Image source: [1].

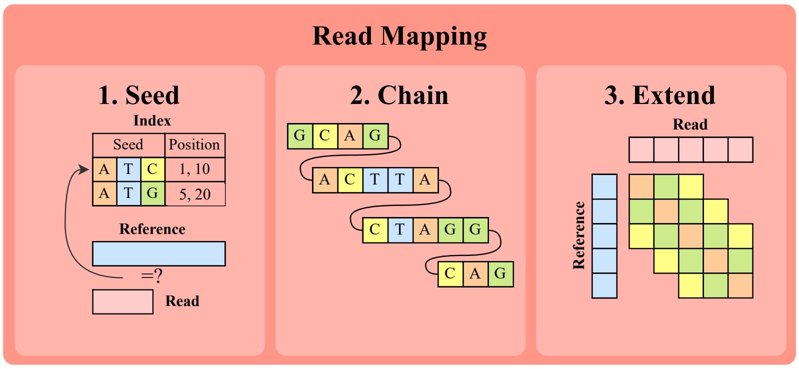
2.2.3 Metagenomics

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 metagenomic analysis pipelines (Figure 2.1-3.c). Metagenomic pipelines seek to analyze genomic information from mixed microbial communities, providing insights into the diversity, interactions, and functions of microorganisms present in an environmental sample. Metagenomic analysis is performed using tools such as Centrifuge [42], RawMap [43], UNCALLED [44], ReadFish [45], Kraken2 [46] and Clark [47]. These tools employ k-mer counting (Figure 2.1-3.c.1) and seeding techniques (Figure 2.1-3.c.2) for their analyses. Moreover, variant callers like GATK Haplotype Caller [33] and Platypus [34] are used to construct de Bruijn graphs (Figure 2.1-3.c.3) and correct artifacts produced during the mapping process (Figure 2.1-3.a.1). Furthermore, the chaining process (Figure 2.1-3.c.4) is also utilized for genome assembly when using alternative approaches based on de Bruijn graphs [48].

2.3 Read Mapping Pipelines

In this thesis, we focus on read mapping, as it is one of the most demanding phases in genomic pipelines. Figure 2.2 shows the throughput of each phase in genomic pipelines. We can observe that read mapping is the main bottleneck in Illumina systems, and the third and second bottlenecks in ONT and PacBio systems, respectively.

Read mapping is highlighted by a dashed line and a red background in Figure 2.1. Figure 2.3 provides a zoomed-in view of the read mapping region. Read mapping tools employ sequence alignment algorithms to find the place in the reference (e.g., the human



Seed	Position
A T C	1, 10
A T G	5, 20

Figure 2.3: Read-mapping pipeline. This figure is a zoomed-in view of the region highlighted by a dashed line in Figure 2.1.

genome) where a sequence fits better. Read mapping has several applications. Among them are finding mutations in a sequenced animal, differences among species, or finding human cancer cells.

Since alignment algorithms are computationally expensive, read mappers usually follow a seed-and-extend approach. During the seeding stage, the tool searches for partial matches between the sequence and the reference. These partial matches are hints (seeds) for the next stages, where dynamic programming algorithms are used to find the best location for each sequence.

2.3.1 Seeding stage

In order to find seeds, most mappers are based on one of these two algorithmic approaches: FM-Index or hash tables (Figure 2.3.1). The FM-Index can search for a variable-sized chunk and is typically employed if input read sequences are short, as search time is proportional to the input read length. In contrast, hash tables define a fixed size to be searched for and are often employed for long reads, as it yields better results [24]. Both algorithms are memory bound due to their irregular memory access patterns and require a large amount of memory capacity to store the data structures.

There are multiple approaches that are used to improve the performance of the seeding stage. A common approach is to perform multiple seed searches in parallel for different input reads, as these are independent. Due to the irregular access pattern nature of these algorithms, some approaches co-locate data within the same cache line in order to have better locality [49], while other approaches redefine the data structure to compress its memory footprint and enable more aggressive multi-base search steps [50].

Hash tables

To find seeds, one option is to build a hash table that contains the positions in the reference for combinations of *k* base pairs (k-mer). When the application indexes the hash table with a k-mer, the hash table returns a list of positions in the reference. Typically, the sequences are split in k-mers following a given algorithm.

The first implementation of a hash table can be found in BLAST [11, 29] tool. It establishes a k-mer size of 11 base-pairs by default. The basic BLAST algorithm has been improved since its creation.

Minimap2 [24] implements a sliding window that scrolls along the sequence and extracts the lowest k-mer in each window. It starts by establishing a window of *w* base-pairs. Then, it extracts the k-mer with the smallest hash value. It shifts the window one position to the right and repeats the process. The hash table is formed by the k-mers as the keys and the positions as the values.

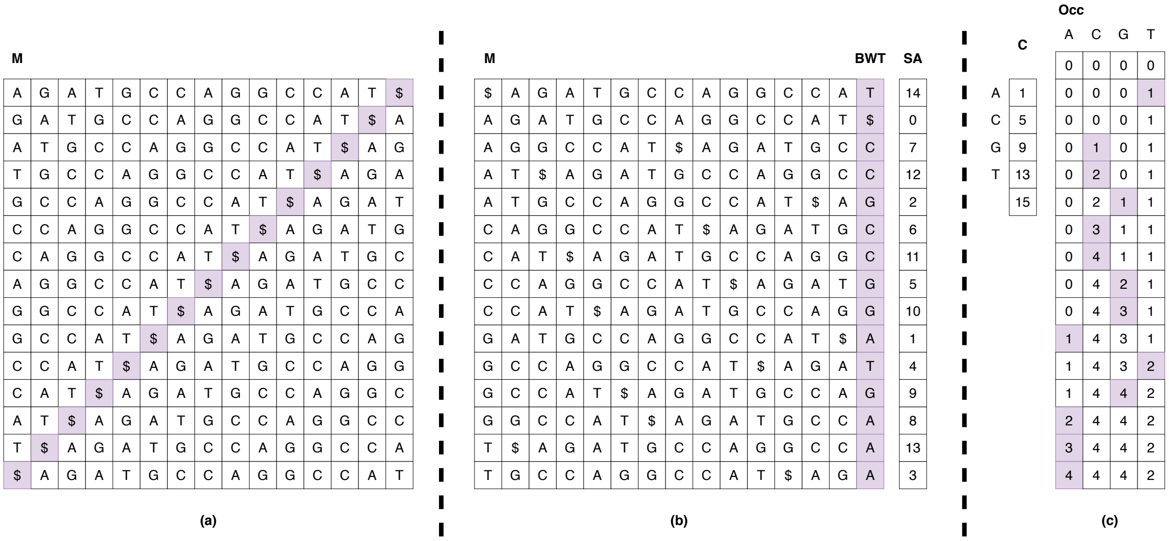
Ma et al. [51] show that k-mers with non-consecutive matches improve sensitivity. Instead of searching for 10 consecutive matches, it is more efficient to search for 10 matches with gaps between them.

These hash tables are considerably large. To cut off the number of entries, k-mers that appear more than a given threshold are removed from the hash table since a repetitive k-mer does not give useful information. To use the hash table, Minimap2 applies the same process to the sequences.

In the case of Minimap2, it establishes the window and extracts the k-mers in the same way as for the reference genome. Minimap2 searches the k-mers in the hash table and creates a set of anchors. An anchor is a tuple of position in the sequence and position in the reference genome. After obtaining all the anchors, a lot of them are part of the same alignment. In order to gather anchors from the same alignment, Minimap2 utilizes the chain algorithm, which is described in Section 2.3.2.

FM-Index

The FM-index is a data structure that allows fast string searches over large texts [28]. It is widely used in multiple genomic pipelines to find exact matches of DNA sequences on a reference genome [52, 53]. In particular, in genome sequence alignment where query sequences typically have a few hundred bases and references can range from a subset of chromosomes to entire genomes. As an indication, the human genome is around 3 gigabase pairs (Gbp). Given a fixed alphabet, the complexity of a sequential search is



M														
A	G	A	T	G	C	C	A	G	G	C	C	A	T	$
G	A	T	G	C	C	A	G	G	C	C	A	T	$	A
A	T	G	C	C	A	G	G	C	C	A	T	$	A	G
T	G	C	C	A	G	G	C	C	A	T	$	A	G	A
G	C	C	A	G	G	C	C	A	T	$	A	G	A	T
C	C	A	G	G	C	C	A	T	$	A	G	A	T	G
C	A	G	G	C	C	A	T	$	A	G	A	T	G	C
A	G	G	C	C	A	T	$	A	G	A	T	G	C	C
G	G	C	C	A	T	$	A	G	A	T	G	C	C	A
G	C	C	A	T	$	A	G	A	T	G	C	C	A	G
C	C	A	T	$	A	G	A	T	G	C	C	A	G	G
C	A	T	$	A	G	A	T	G	C	C	A	G	G	C
A	T	$	A	G	A	T	G	C	C	A	G	G	C	C
T	$	A	G	A	T	G	C	C	A	G	G	C	C	A
$	A	G	A	T	G	C	C	A	G	G	C	C	A	T

(a)

M														BWT	SA
$	A	G	A	T	G	C	C	A	G	G	C	C	A	T	14
A	G	A	T	G	C	C	A	G	G	C	C	A	T	$	0
A	G	G	C	C	A	T	$	A	G	A	T	G	C	C	7
A	T	$	A	G	A	T	G	C	C	A	G	G	C	C	12
A	T	G	C	C	A	G	G	C	C	A	T	$	A	G	2
C	A	G	G	C	C	A	T	$	A	G	A	T	G	C	6
C	A	T	$	A	G	A	T	G	C	C	A	G	G	C	11
C	C	A	G	G	C	C	A	T	$	A	G	A	T	G	5
C	C	A	T	$	A	G	A	T	G	C	C	A	G	G	10
G	A	T	G	C	C	A	G	G	C	C	A	T	$	A	1
G	C	C	A	G	G	C	C	A	T	$	A	G	A	T	4
G	C	C	A	T	$	A	G	A	T	G	C	C	A	G	9
G	G	C	C	A	T	$	A	G	A	T	G	C	C	A	8
T	$	A	G	A	T	G	C	C	A	G	G	C	C	A	13
T	G	C	C	A	G	G	C	C	A	T	$	A	G	A	3

(b)

	C
A	1
C	5
G	9
T	13
	15

Occ

A	C	G	T
0	0	0	0
0	0	0	1
0	0	0	1
0	1	0	1
0	2	0	1
0	2	1	1
0	3	1	1
0	4	1	1
0	4	2	1
0	4	3	1
1	4	3	1
1	4	3	2
1	4	4	2
2	4	4	2
3	4	4	2
4	4	4	2

(c)

Figure 2.4: Procedure to build the FM-index. (a) Step 1: cyclic rotations to generate matrix M. (b) Steps 2 and 3: sort M alphabetically and extract the last column (BWT) and the suffix array (SA). (c) Step 4: build C and Occ structures.

$O(n)$, where n is the length of the reference, while a query based on the FM-index has complexity $O(m)$, where m is the length of the sequence to be searched.

To construct the FM-index of a reference string, its Burrows-Wheeler Transform (BWT) has to be computed [54]. This is achieved by appending a special symbol $, which is lexicographically smaller than all other symbols, and by performing all the circular shifts as shown in Figure 2.4a for the string "AGATGCCAGGCCAT". Then, all the rows of the resulting matrix M are sorted lexicographically, as shown in Figure 2.4b. The last column of M is the BWT of the reference string. For each row of M, its starting position in the original reference is stored in a structure called suffix array (SA) [55], see Figure 2.4b. When a query finishes, the suffix array is looked up to locate the matching positions in the original reference.

The two structures that constitute the FM-index, C and Occ, can be generated from the BWT (see Figure 2.4c). C is an array of size $\sigma + 1$, where σ is the number of symbols in the alphabet. Since DNA has 4 symbols (bases) $\{A, C, G, T\}$, the size of C is 5. $C[s]$ contains the number of bases alphabetically smaller than s in BWT. Occ has a column for each base, and a row for each BWT symbol. Each row corresponds to a BWT position and contains the number of bases of each type in the BWT up to (but not including) that position. Hence, $Occ[p, s]$ contains the number of occurrences of the base s in the BWT range $[0, p)$. For example, in Figure 2.4c, the $Occ[6, 3]$ cell indicates the number of Ts until the 6th position of the BWT, that is, in the range [0-5).

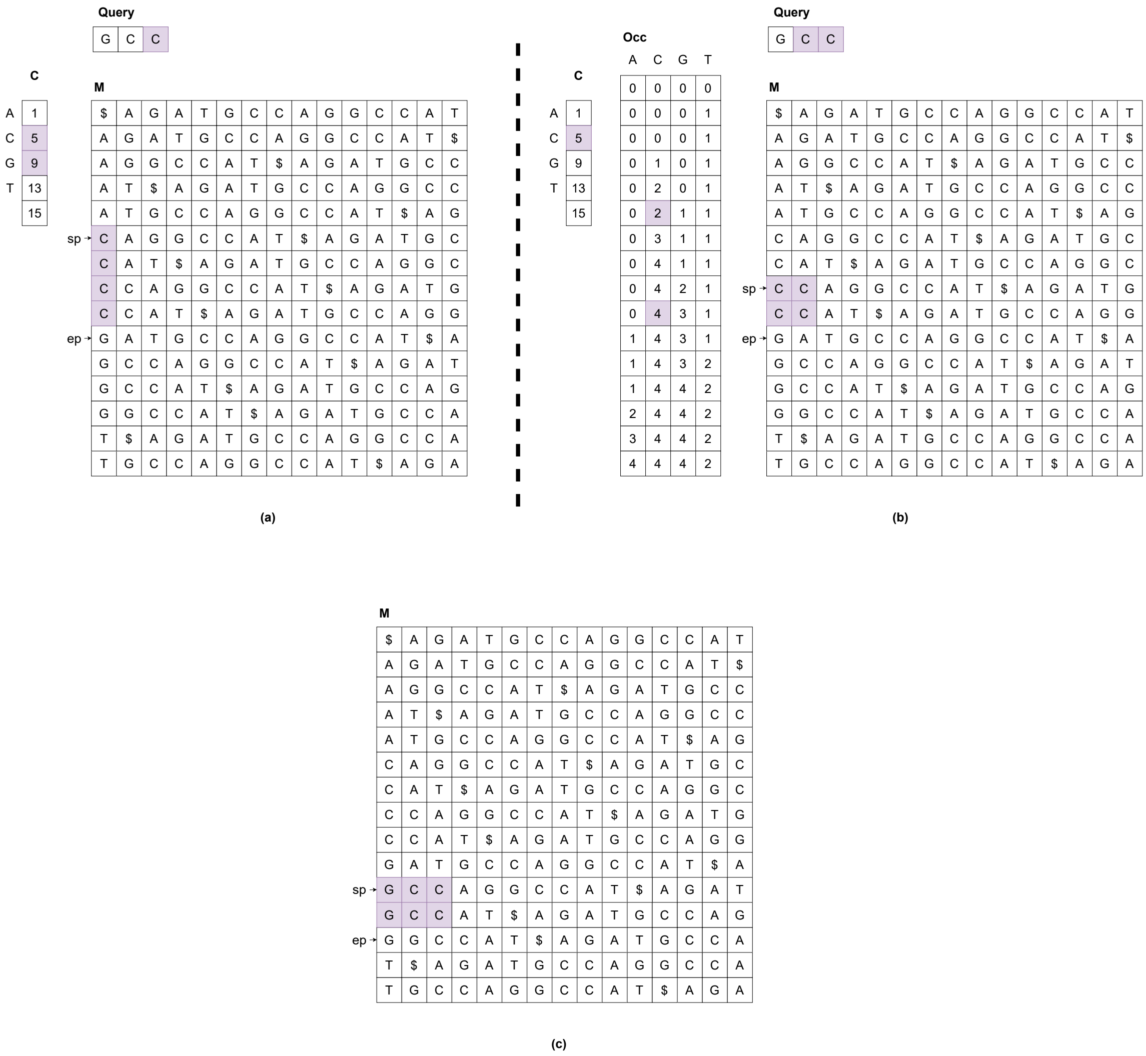


Figure 2.5: Backward search example. (a) *sp* and *ep* initialization. (b) LF operation for *sp* and *ep*. (c) Final state of the query.

The *backward search* (BS) is an algorithm based on the FM-index that performs exact matching of a query $Q[1..n]$. BS defines two pointers (start and end, *sp* and *ep*) that indicate the range where are all the possible candidates of the current search appear in the matrix M. Note that M is not a necessary data structure, we show its instantiation for clarity. BS starts processing the last base in Q, i.e., $s = Q[n-1]$), by initializing the *sp* and *ep* pointers with $C[s]$ and $C[s+1]$, respectively (see Equations 2.1 and 2.2). Figure 2.5a shows an example of this stage for the sequence *GCC*. For each of the remaining bases in the query string, two Last-to-First Mapping operations (LF) are performed, one for each pointer (*sp* and *ep*). LF is defined in Equation 2.3 for a pointer p and a base s [28].

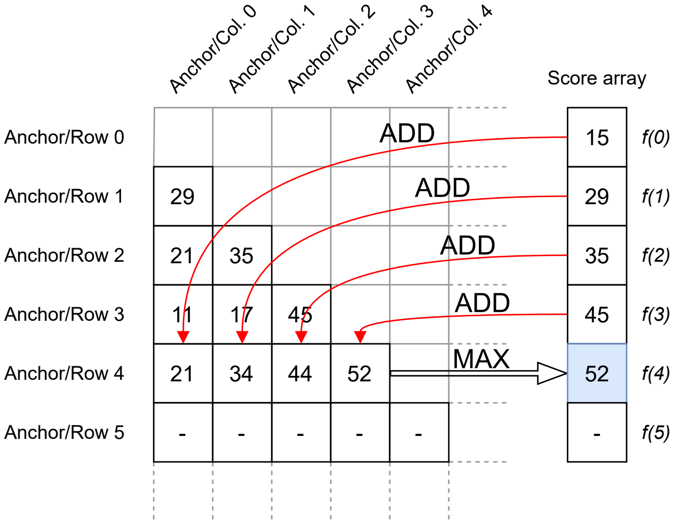


Figure 2.6: Chain algorithm dependencies. To calculate the score of anchor 4 ($f(4)$), we add all the previous scores to row 4 and perform a maximum. In the next iteration, $f(4)$ will be added to row 5.

$$sp = C[s] \tag{2.1}$$

$$ep = C[s+1] \tag{2.2}$$

$$p = LF(p,s) = C[s] + Occ[p,s] \tag{2.3}$$

Following the example, Figure 2.5b shows the effect of the LFs corresponding to the second base $Q[n-2] = C$. The updated pointers indicate the range of rows in M that would contain the partial query we have searched so far. Finally, Figure 2.5c shows the final position of the pointers after performing all the LF operations. The range $[sp, ep)$ indicates the first and last rows of M prefixed by the query $Q[1..n]$.

Each row of M corresponds to one rotation of the original reference. With the SA, we can obtain the starting positions in the original reference of the matches found between sp and ep.

2.3.2 Filtering: Chain

The seeding stage returns a set of seeds to start the alignment stage. However, this set can be pruned even further with other algorithms. In the case of the Minimap2 tool [24], the seeding stage returns anchors. An anchor is a tuple of position in the reference and

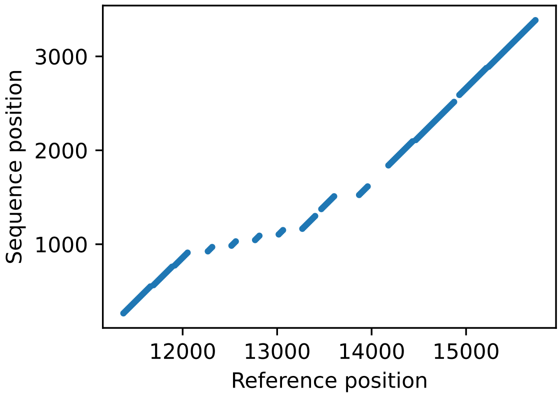


Figure 2.7: Graphical representation of the anchors generated by an execution. Each point represents an anchor, i.e., a match of k base pairs (15 in this example) between the sequence and the reference. All the anchors form a *chain*.

position in the query, which indicates an exact match of k base pairs (15 by default) between the reference and the query.

Chain is a dynamic programming algorithm that combines several anchors to create a more extensive matching region, thus creating a *chain* (Figure 2.3.2). The chain algorithm receives a set of sorted anchors and scores each pair of anchors. The match-up score of the anchor i is calculated with the following formula:

$$f(i) = \max_{(i-T)\leq j<i} \{f(j) + \alpha(i,j) - \beta(i,j)\} \tag{2.4}$$

Where $f(i)$ is the score of the anchor i, $\alpha(i,j)$ is the bonus score between anchors i and j, β is the penalization score between the anchors i and j, and T is the chain iteration threshold. Notice that the $f(i)$ calculation depends on $f(i-1)$. On the other hand, the calculation of α and β can be entirely computed in parallel for all i and j values.

Figure 2.6 represents the dependencies inside the chain kernel. The score array (f in Equation 2.4) is added to the match-up scores (α and β in Equation 2.4). Then, a maximum of the row is calculated, and the score of anchor 4 is obtained. Notice that $f(4)$ will be added to subsequent rows, thus creating a dependency in the outer loop.

In addition to the score array, we compute a predecessor array, where we store the index of the best match-up for each anchor. With both arrays computed, we perform the backtrack. For the best score in the score array, we pick its predecessor recursively. The list of predecessors forms a *chain*. Figure 2.7 shows a graphical representation of an output generated by the *chain* stage where each point represents an anchor.

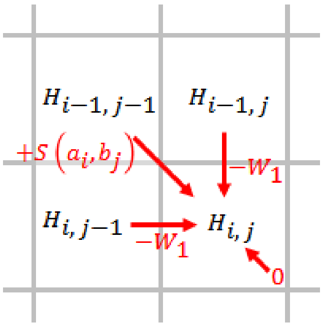


Figure 2.8: Cell computation dependencies in Smith–Waterman algorithm. Image source: [2].

2.3.3 Extend Stage: Align Algorithms

For the extend part, the most popular algorithm is Smith-Waterman [31, 32] and variants that improve its performance based on observations [12, 13, 56] (Figure 2.3.3). The algorithm builds a matrix to score inexact matches of the reference and the input read. To compute the *(i,j)* cell of the matrix, it needs the *(i-1,j)*, *(i,j-1)* and *(i-1,j-1)* cells, which causes a heavily-constrained computation due to these dependencies (see Figure 2.8).

The seed-and-extend strategy allows limiting the search space of the Smith-Waterman algorithm by only populating the matrix around the seeds. Moreover, most seeds end up with a low score in the extend process, which adds significant compute overhead that is later discarded. For this reason, some approaches identify and discard low quality seeds, e.g., FastHASH [57], GateKeeper [58], and Shouji [59].

As in the seeding step, alignments for different seeds are independent and can be efficiently distributed across different threads. In addition, since this is a compute bound algorithm over a matrix, there has been a significant effort to vectorize it. We can differentiate two ways of vectorization: intertask and intratask. Intertask vectorization places a different alignment on each element of the vector. However, this leads to imbalance due to the variable size of the matrices the algorithm needs to compute, and quickly becomes inefficient. In contrast, intratask vectorization tries to apply vectorization to a single alignment. However, this again proves difficult due to the constraints imposed by the algorithm in terms of dependencies. The straightforward implementation vectorizes the antidiagonal [60], but it delivers limited performance as the access pattern is strided. Some approaches vectorize the columns, ignoring certain constraints [61, 62], or rebuilding vertical constraints in a second run [63]. Finally, as proposed by Rognes [64], each lane of the vector could be used for one different sequence. This proposal is the most used in read mapping tools [15, 24], however, it entails problems derived from the load unbalance among the sequences.

Since vectorizing this algorithm is not straightforward, all implementations resort to hand-written ISA-dependent intrinsics, as the compiler is not able to auto-vectorize it. As a result, most aligners only have these vectorized versions for what has been the dominant ISA in the last decades, i.e., x86-64. Therefore, the optimized versions of the aligners only work out-of-the-box on x86-64 machines, and the Smith-Waterman algorithm needs to be ported in order to make it run with other ISAs.

The extend stage takes each one of the produced seeds as input and aligns them with the reference to obtain an *alignment score*. The gap-affine model is used to score an alignment. This model scores matches and mismatches with a given value and the gaps with a linear function, where there is a fixed penalty when the gap starts and an increment for each base pair. The scores are configurable, so biologists can change them to model mutations and events in the genome.

Smith-Waterman (SW) [31, 65] is the algorithm selected by most tools to implement the gap-affine model. It calculates a dynamic programming matrix with a cost of $O(n \times m)$, where n and m are the lengths of the sequences to align. Since this is an expensive computation, most tools implement heuristics to reduce execution time. However, this leads to suboptimal alignments and could harm the result.

Recently, the Wavefront Alignment algorithm (WFA) has been proposed [56]. WFA computes the alignment between two sequences with cost $O(n \times s)$, where n is the sequence length, and s is the alignment error. Hence, if two strings are identical, WFA only computes the main diagonal. It supports the gap-affine scoring scheme, and the result is identical to SW. In addition, WFA also allows optimizations and heuristics.

Other scoring schemes are available in the literature, like the edit distance model, a subset of the gap-affine model, where each error (insert, deletion, or mismatch) has the same penalization. They are not used in modern tools since they are not configurable by biologists. Two examples of algorithms implementing edit distance are Bitap [66, 67] and Bit-Parallel Myers [68].

Edit distance has been chosen for some hardware accelerators. GenASM [69] is a software-hardware co-design based on the Bitap algorithm. However, GenASM does not provide the optimal score (and hence, neither the alignment) since it is an edit distance algorithm.

2.4 Dependency-Bound Use Cases

This thesis focuses mainly on genomics. Among other algorithms, we study and optimize Smith-Waterman and Chain, two dependency-bound kernels. As we progressed in our research, we found it interesting to expand our set of dependency-bound kernels with more use cases. In this section, we discuss two additional use cases utilized in our last contribution: data sorting, and signal processing.

2.4.1 Data Sorting

Sorting [70] is a widely studied problem in computer science, fundamental to various applications such as search engines, data mining, databases, and numerical methods. The importance of sorting spans from energy-efficient devices [71, 72] to GPGPUs and data centers [73–75].

Radix sort [70] is an efficient sorting algorithm with a time complexity of $O(nk)$, where n is the number of elements in the array and k is the length of the key. This makes it particularly effective to sort arrays of 32-bit or 64-bit integers. In the first iteration, the algorithm uses the eight most significant bits to divide the array into 2^8 buckets. This process is repeated recursively for subsequent bits until all bits are processed.

2.4.2 Signal Processing

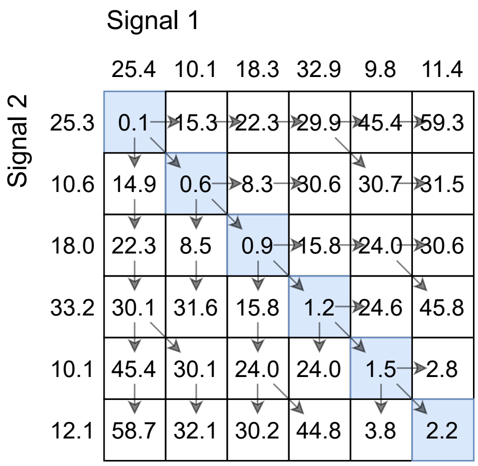
Signal processing focuses on analyzing various types of signals, including sound, images, potential fields, seismic data, altimetry, and scientific measurements. A signal represents a flow of information originating from a source and can take many forms, such as mechanical, optical, magnetic, electrical, or acoustic. Signals can be digital, characterized by discrete values, such as semaphores, Morse code, or the contents of computer memory. Conversely, signals can also be analog, encompassing continuous values like pressure, temperature, or velocity.

Dynamic Time Warping (DTW) is a 2D dynamic programming algorithm designed to align two signals to measure their similarity. It is widely used in applications such as speech recognition, speaker recognition, and music recognition. DTW constructs a dynamic programming matrix with a computational complexity of $O(nm)$, where n and m represent the lengths of the signals being aligned.

Figure 2.9 provides an example of a DTW matrix. Each cell in the matrix is computed using the following equation:

$$M[i,j] = abs(S[i] - R[j]) + min\{M[i-1,j-1], M[i-1,j], M[i,j-1]\} \tag{2.5}$$

Where $M[i,j]$ is the cell in row i and column j, while S and R are the aligned signals. The equation calculates the value of the cell $[i,j]$ as the minimum value of the neighboring



	25.4	10.1	18.3	32.9	9.8	11.4
25.3	0.1	15.3	22.3	29.9	45.4	59.3
10.6	14.9	0.6	8.3	30.6	30.7	31.5
18.0	22.3	8.5	0.9	15.8	24.0	30.6
33.2	30.1	31.6	15.8	1.2	24.6	45.8
10.1	45.4	30.1	24.0	24.0	1.5	2.8
12.1	58.7	32.1	30.2	44.8	3.8	2.2

Figure 2.9: DTW matrix representation. The samples from signal 1 are set at the top of the matrix, while samples from signal 2 are set at the left. For each cell DTW computes Equation 2.5. Arrows indicate the minimum value from Equation 2.5.

left, top, and top-left cells plus the absolute difference between $S[i]$ and $R[j]$. This dependency on the left, top, and top-left cells complicates parallelization.

2.5 Genomic Hardware Accelerators

GPGPUs have been widely used for parallel applications. In the field of sequence alignment, CUDASW++ [76] is a CUDA library for Smith-Waterman. Also, the DPX extension for dynamic-programming kernels has recently been released [77, 78]. In the field of seeding algorithms, Guo et al. provide a detailed analysis of the GPU limitations when accelerating the chain kernel [79]. When programming the GPU version, 16.3% of the instructions in the kernel are control instructions for synchronizing warps. They use an NVIDIA Tesla P100 [80] for the evaluation. The P100 GPU achieves a 3.17× speedup with respect to a 14-core CPU while consuming 300 W and occupying 610 mm^2.

Some seed filters in hardware have been proposed as an intermediate stage between the seeding and alignment phases. The objective is to reduce the computation during the align stage by filtering out the least promising seeds. In this regard, we can highlight Shouji [59], Gatekeeper [58], and FastHASH [57].

Other proposals, like GenASM [69], accelerate the alignment process. The authors of GenASM propose a new hardware-friendly algorithm for edit distance alignments. GMX [81] has been recently proposed as an ISA extension for dynamic programming kernels. And, more generally, GenDP [82] has been proposed as a framework for dynamic-programming algorithms.

There are several proposals for accelerating read mapping pipelines, like GenAx [83], Darwin [84], Olson et. al. [85], and FHAST [86]. It is also worth mentioning SeGraM [87], the first hardware accelerator for sequence-to-graph alignments. Other proposals use in-cache or in-memory operations, like GenCache [88], PARC [89], and GenPIP [90]. We should also mention AIM [91], a PIM framework for sequence alignment that uses UPMEM [92] for the evaluation.

We identify some general-purpose hardware accelerators that inspire this thesis. For example, Meet the Walkers [93], a programmable hardware accelerator for traversing hash tables in a database. UPMEM [94] is the first publicly available general-purpose programmable PIM system. Versa [95], a multiprocessor system whose memories can be reconfigured as a cache, a scratchpad, or to mimic a systolic array.

Chapter 3

Methodology

This chapter describes the methodology used to evaluate the proposals of this thesis. First, the read mapping tools are presented. Then, the characteristics of the systems used for the experimentation are detailed. Next, the simulation environment is presented. Finally, the executed workloads are described.

3.1 Read mapping tools

Here, we present the two read mapping tools we have used in the thesis: BWA-MEM2 [15] and Minimap2 [24].

3.1.1 BWA-MEM2

BWA-MEM2 [15] follows a seed-and-extend approach. To profile the application, we define three regions of interest:

- **Super Maximal Exact Matches (SMEM)**: uses an FM-Index structure to find seeds based on exact matching.
- **Suffix Array Lookup (SAL)**: translates FM-Index positions of the seeds to their locations in the reference genome using a suffix array.
- **Banded Smith-Waterman (BSW)**: extends the seeds and scores the alignments.

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

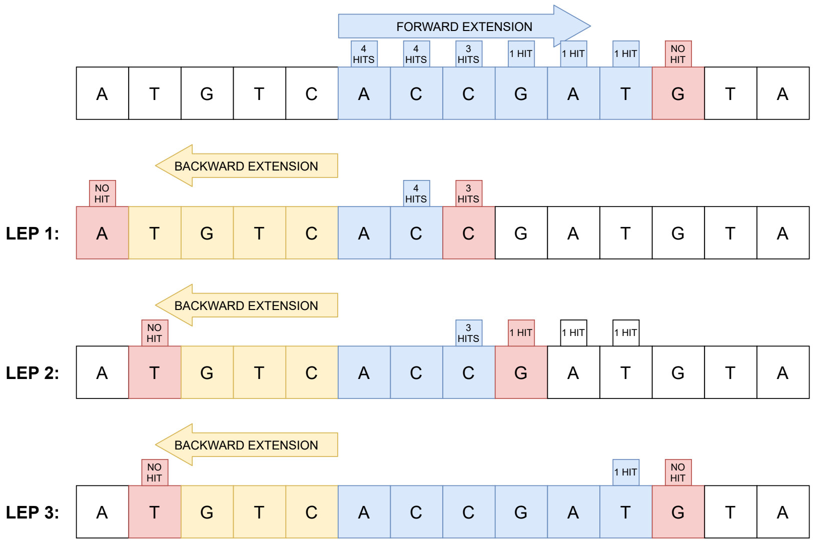


Figure 3.1: SMEM process scheme.

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

Figure 3.1 shows the process of getting the SMEMs. First, we fix a point in the read input sequence to start from, the leftmost blue base in the figure. By querying the FM-Index, we keep expanding the seed (forward extension) until we have no hits in the reference, the red base at the end of the forward extension. During this forward extension, we store the positions where the number of hits changes. These points are called left extension points (LEP). In Figure 3.1, there are three LEPs, when going from four hits to three hits (LEP1), going from three hits to one hit (LEP2) and going from one hit to no hits (LEP3). Then, the algorithm expands backwards the LEPs until there are no hits in the reference genome. Finally, to obtain the SMEMs, we discard the MEMs that overlap with other MEMs. In our example, the MEM produced by LEP2 is discarded since it overlaps with the MEM produced by LEP3. Therefore, two SMEMs are found, those originating from LEP1 and LEP3. These two seeds will later be extended during the extend phase implemented in the BSW region.

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

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

The SMEM region finds the seeds based on exact matching that will be later extended using BSW. However, the position in the FM-Index found in the SMEM phase has to be translated to obtain the position of the seed in the reference. This is done through the Suffix Array (SA). For the human genome, an uncompressed SA requires 48 GB, a considerable size that will not fit in certain systems. For this reason the SA is also compressed. BWA-MEM2 offers a configurable compression factor x, that samples the SA by 2^x. This means that the SA will store one out of 2^x entries, potentially performing 2^x operations to recover the original value. We choose a compression factor of 3, i.e., our SA occupies 6 GB ($48/2^3$) of memory.

Finally, for the extend part, BWA-MEM2 uses the Banded Smith-Waterman (BSW) algorithm and the affine-gap penalty model [32]. The alignment is done by populating a matrix with as many columns/rows as the input read. Instead of computing all cells in the matrix, BSW only computes the cells around the main diagonal, i.e., a band [96]. The width of this band depends on the score of the previous rows. If the score of one row drops drastically, the width of the band is reduced or even the algorithm can stop. The affine-gap penalty model adds a penalty for opening a gap. This means that two separate gaps produce a worse score than two consecutive gaps.

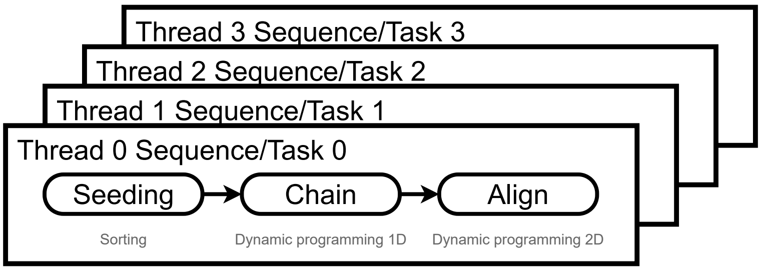


Figure 3.2: The 3-stage Minimap2 pipeline. For each stage, the most time-consuming kernel is indicated.

3.1.2 Minimap2

Minimap2 [24] is a widely used state-of-the-art mapper for long sequences. Figure 3.2 shows how Minimap2 splits work into coarse-grain tasks, where each thread operates on a different sequence and performs a 3-stage pipeline:

- **Seeding:** uses a hash table to find seeds. The bottleneck in this stage is a sorting algorithm to order the seeds.
- **Chain:** combines seeds to create the largest possible matching region. The main kernel comprises a 1D dynamic programming algorithm.
- **Align:** extends the seeds and scores the alignments. The alignment algorithm is a 2D dynamic programming algorithm.

To find seeds, Minimap2 builds a hash table that contains the positions in the reference genome for combinations of *k* base pairs (k-mers). To split the reference genome into k-mers, Minimap2 establishes a sliding window that moves along the reference. Inside this window, Minimap2 selects the k-mer with the smallest hash value in each window. Then, it shifts the window by one position and repeats the process until the entire reference is processed. Each one of these k-mers is called a minimizer. When Minimap2 indexes the hash table with a k-mer, the hash table returns a list of positions in the reference.

When Minimap2 is processing a sequence, it splits the sequence in k-mers in the same way as in the reference with the sliding window. Then, it indexes the hash table with all the minimizers and extracts a list of tuples, called anchors, which consist of the position in the sequence and the position in the reference. Finally, the anchors are sorted by their position in the reference so following stages can easily traverse the list. For this purpose, Minimap2 uses a sorting algorithm, which depends on the input: heap merge for short reads and radix sort for long reads. This sorting process is the most time-consuming step of the entire seeding stage.

Table 3.1: Target hardware platform specifications.

	SKX	**KNL**	**A64FX**	**Graviton3**	**Rome**
Processor	2 × Intel Xeon Platinum 8160	Intel Xeon Phi 7230	Fujitsu A64FX	AWS Graviton 3	AMD EPYC 7742
Architecture	x86	x86	ARM	ARM	x86
Cores[×Threads]	24×2	64×4	48	64	64
Base frequency	2.1 GHz	1.3 GHz	2.2 GHz	2.6 GHz	2.25 GHz
Vector Extension	AVX-512	AVX-512	SVE 512	SVE 256	AVX2
L1 instr. and data	32 KB (8-way)	32 KB (8-way)	64 KB (4-way)	64 KB	32 KB (8-way)
L2	1 MB (16-way)	-	-	1 MB	512 KB (8-way)
LLC	2×33 MB (11-way)	32 MB	4×8 MB (16-way)	32 MB	16×16 MB (16-way)
Mem. technology	DDR4 2667 MHz	DDR4 2400 MHz | MCDRAM	HBM2	DDR5 4800 MHz	DDR4 3200 MHz
Mem. capacity	2×48 GB	192 GB | 16GB	4×8 GB	8×16 GB	16×64 GB
Mem. bandwidth	2×120 GB/s	90 GB/s | 480 GB/s	4×256 GB/s	300 GB/s	204.8 GB/s

To filter out the repeated seeds, Minimap2 uses the chain algorithm. Chain joins partial exact matches between the sequence and the reference (anchors) to obtain a larger matching region. To *chain* two anchors, Minimap2 uses Equation 2.4 described in Section 2.3.2. A 1D dynamic programming array is used to obtain the optimal *chain* of anchors. The resulting *chain* leaves unaligned gaps between the chained seeds. The alignment algorithm is in charge of aligning these gaps.

As the aligning algorithm, Minimap2 uses KSW2 [21, 24, 97], a modified SW with heuristics to reduce the computation time. That is, KSW2 only computes a certain number of diagonals around the main diagonal, also known as banded SW [96]. In addition, it uses a z-drop heuristic, i.e., it stops computing if the score drops below a given threshold.

3.2 Target Hardware Platforms

We have evaluated our proposals in several systems, two ARM and three x86 HPC systems: a compute node featuring an ARM-A64FX processor (A64FX) [98–100], a c7g.16xlarge Amazon-EC2 instance (Graviton3) [101, 102], a node with two x86-64 Intel Xeon Skylake Platinum 8160 (SKX) [103, 104], a compute node with one x86-64 AMD EPYC 7742 Rome processor (Rome) [105–107], and a compute node with an x86-64 Intel Xeon Phi 7230 (KNL) [108]. Table 3.1 presents an overview of the main characteristics of the five systems.

All the evaluated systems are CPU-based platforms. This choice is motivated by two main reasons. Firstly, the majority of algorithms and tools used in genomics are developed for CPUs, such as BWA-MEM2 [15] and Minimap2 [24]. Secondly, GPU implementations of these algorithms are generally inefficient, as GPUs are not well-suited to the dependency-bound computation patterns typical of dynamic programming kernels [79, 109–112]. Moreover, a significant part of the work presented in this thesis focuses on porting existing tools to the ARM architecture, which is rapidly emerging in HPC and server environments. The previously mentioned A64FX and Graviton3 processors have only recently been released, and they already appear in the Top500 ranking [113] and in Amazon Web Services (AWS) instances. Finally, the systems were selected based on two criteria: (i) they represent state-of-the-art HPC platforms available at the time of each publication, and (ii) they provide sufficient memory capacity to handle genomic workloads.

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 eight 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 four cores. Any pair of CCDs can communicate through the I/O die. SKX contains two NUMA chips, each with 24 physical cores. Each core supports two-way SMT, totaling 96 virtual cores. KNL is composed of 64 physical cores disposed in a mesh with the possibility of activating four-way SMT per core.

Graviton3 presents the highest frequency among the systems with 2.6 GHz. A64FX, Rome, and SKX systems' frequencies are very similar, ranging between 2.1 and 2.25 GHz. While KNL has a frequency of 1.3 GHz.

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 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, totalling 1 TB of memory. KNL has a MCDRAM of 16 GB, which can be used as main memory or as a cache, with an extension of 196 GB of DDR4 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 1 MB 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 and KNL include 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.

Table 3.2: Simulated architectural parameters.

Cores	8 Neoverse-N1-like ARMv8 out-of-order cores 2.4 GHz
Structure entries	ROB: 224 | LD/ST queues: 96/96 | Inst. queue: 120
OoO Private L1 I&D	64 KB, 4-way, 1 cycle data access, 32 MSHRs
Private L2	512 KB, 8-way, 4 cycle data access, 64 MSHRs
Shared L3	Mostly exclusive, 8 slices of 1 MB, 16-way, 10 cycles data access, 128 MSHRs
Coherence protocol	MOESI-like AMBA 5 CHI specification
Network topology	4×4 2D mesh, 1 cycle routers, 1 cycle links (Figure 7.2a)
Memory	1 HBM2 stack, 300 GB/s

Concerning memory bandwidth, the A64FX is designed to achieve good performance executing high memory bandwidth-demanding applications, with a peak bandwidth of 4 × 256 GB/s. The KNL MCDRAM is the second fastest memory with 480 GB/s, while its DDR4 has 90 GB/s. Graviton3, with 300 GB/s, is the third 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.

3.3 Simulation Platform: Gem5

Gem5 [114, 115] is an open source full-system simulator released in 2011. It has the support of big companies, such as Arm, AMD, and Google. With Gem5, we can configure and simulate a full-system SoC. We can tune up the parameters of the CPU, such as the size of the hardware structures or the frequency of the chip. We can combine different caches and memory systems, changing the parameters like the capacity or the technology. Also, we can create new modules and attach them to the system.

For the experiments, we simulate a multicore system consisting of 8 Neoverse-N1-like out-of-order cores, three levels of cache, 4 HBM2e memory channels, and a mesh-based network-on-chip modeling the AMBA 5 CHI protocol, as shown in Figure 7.2a. The modeled system seeks to match the state-of-the-art HPC ARM systems at publication time: the Neoverse-N1 core is present in Graviton3 systems. The simulated system runs Ubuntu 22.04 with Linux kernel 5.4.65. Table 3.2 summarizes the architectural parameters.

Table 3.3: Details for the reference genomes.

	Download link	**Accession number**	**Length**	**Used in sections**
GRCh37 [52]	[116]	GCA_000001405.14	2.86 Gbp	4 and 5
GRCh38 [53]	[117]	GCA_000001405.15	3.05 Gbp	5 and 6
T2T [118]	[119]	GCA_009914755.4	3.16 Gbp	7

3.4 Inputs: Genomes and Sequences

We differentiate two kinds of inputs when dealing with read-mapping tools: a reference genome used to guide the read-mapping process (see Section 2.3) and the sequences to be aligned by the tool.

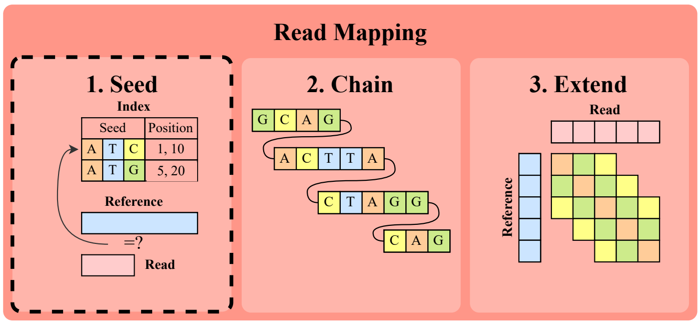
We perform experiments using three human reference genomes: GRCh37 [52], GRCh38 [53], and the human genome T2T [118]. Table 3.3 shows more details about these genomes. Note that the T2T genome is the first human genome completed with no missing parts.

We use the state-of-the-art genome at the time of publication of each article. The first contribution (Section 4) uses GRCh37 and GRCh38 genomes. The second contribution (Section 5) uses the GRCh38 genome. The third contribution (Section 6) use the GRCh38 genome for the evaluation of the end-to-end application. The micro-benchmarks are evaluated using intermediate results extracted from read-mapping tools. The fourth contribution (Section 7) uses the T2T genome.

The input sequences are described individually for each contribution in their correspondent chapters. We observe two types of sequences read: real reads and simulated reads. Real reads come from sequencer systems such as the ones described in Table 2.1. On the other hand, simulated reads are generated with a software such as Mason [120] or PBSIM [121].

Chapter 4

COFI: Compressed FM-Index for Large K-steps



COFI *is a new data layout and algorithm for FM-Index used in the* ***seed stage***

The FM-Index is a data structure used in genomics for exact search of input sequences over large reference genomes. It is widely used by short read aligners such as BWA-MEM2 [15], Bowtie2 [26], and HISAT2 [122]. The FM-Index is more suitable for short reads since it can adjust freely the size of the seed, unlike a hash table, enabling more sensitivity. As we see in Figure 5.8, BWA-MEM2 spends around 60% on the SMEM stage, that utilizes the FM-Index structure. Therefore, optimizing the FM-Index is key for an efficient alignment process.

Algorithms based on the FM-Index show an irregular memory access pattern, resulting in a memory bound problem. We analyze a recent implementation of the FM-Index and highlight existing throughput-memory trade-offs, showing that memory requirements limit implementation of large k-steps. We show that while a linear improvement in computational throughput can be achieved by increasing the number of bases searched per step (k), the memory requirements quickly become prohibitive as they increase exponentially. We propose COFI, a **CO**mpressed **FM-I**ndex for large K-steps. COFI enables a 15-step FM-Index using less than 16 GB for a human genome reference of 3 giga base pairs. We evaluate COFI on both a Knights Landing (KNL) and a Skylake-based system (SKX). We achieve average speedups of 1.46× and 1.39×, respectively, with respect to a state-of-the-art FM-Index implementation that is already well optimized.

COFI is available at https://gitlab.bsc.es/rlangari/cofi.

4.1 State-of-the-Art FM-Index implementations

As we show in Section 2.3.1, FM-Index is a data structure that allows exact searching over a large genome with a computational complexity of $O(m)$, where m is the length of the sequence to be searched. FM-Index is comprised of the matrix *Occ* and the table C, which have been created using the BWT of the reference genome (see Figure 2.4). The algorithm to search a sequence using the index is shown in Figure 2.5 and Equation 2.3.

In this section, we show the state-of-the-art technics used to improve FM-Index based searches, these are sampling and k-steps. Then, we show *bvSFM*, an FM-Index version that combines these techniques and performs a data structure transformation to speed up the reconstruction of the sampled entries. Finally, we perform a memory footprint analysis to find out opportunities of improvement.

4.1.1 Sampled FM-Index

The *Occ* structure requires as many rows as the length of the reference (Figure 4.1a), leading to a large memory footprint. As an example, for the human genome, the *Occ* structure would occupy around 48 GB. To reduce this footprint, a sampling technique that stores one row out of every d rows of *Occ* can be applied [50]. The reduced structure, called *rOcc* (see Figure 4.1b), can be defined by the following expression: $rOcc[p,s] = Occ[p \times d, s]$. This reduction in memory footprint comes at a computational cost. In order to reconstruct the discarded entries, both the reduced *rOcc* and the original BWT are needed. The additional computation reads the BWT to count the bases from the last sampled counter in *rOcc* to the desired position, as in Equation 4.1.

$$Occ[p,s] = rOcc[p/d,s] + occur(s, BWT[(p - p \bmod d)..p]) \tag{4.1}$$

For instance, to compute the number of A symbols up to the tenth row of *Occ*, we add the counter corresponding to the A base found in the second row of the *rOcc*, i.e., 0

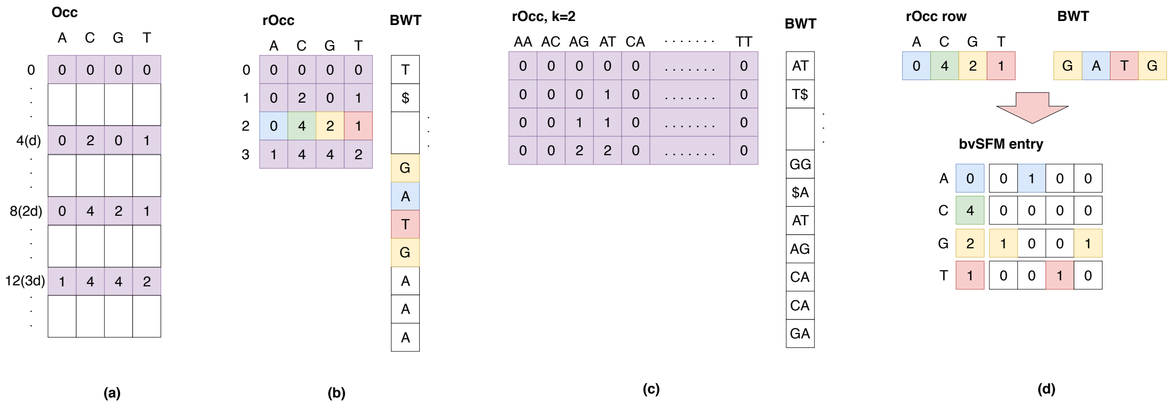


Figure 4.1: Several FM-Index implementations. (a) Full FM-Index with reference length n. (b) Sampled FM-Index, one out of d counters is stored. (c) 2-step sampled FM-Index, uses an alphabet of 16 symbols. (d) Split bit-vector FM-Index, each bit indicates whether a base is in a given BWT position or not.

(see A counter in blue in Figure 4.1b), to the number of occurrences of that base in the first two positions of its corresponding BWT block, i.e., 1 (see coloured BWT-block in Figure 4.1b).

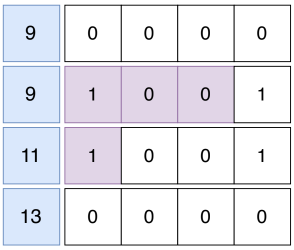
4.1.2 K-step Sampled FM-Index

The *k-step sampled FM-Index* (see Figure 4.1c) searches k bases in a single step [14]. k bases per iteration are processed instead of one, and the rest of the algorithm remains the same.

For example, for $k = 2$ the alphabet changes from *{A,C,G,T}* to *{AA, AC, AG, AT, CA, ..., TT}*. The size of C and *rOcc* structures increases by a factor of four (the size of the original alphabet), every time k is incremented by one. In this case, the trade-off is an exponential memory footprint increase for a linear increase in computation throughput as k increases.

4.1.3 bvSFM: Improving Data Locality

The *split bit-vector sampled FM-Index* (*bvSFM*) speeds up computation by improving data locality [49]. The sampled FM-Index layout is changed in order to store all the data needed to compute an LF in a single cache block. The counter for a base in a sampled *rOcc* row is placed close to a bit-vector that encodes the occurrences of that base in the corresponding BWT block. The length of each bitmap in a *rOcc* row is equal to the sampling rate. A bit set to one indicates that the BWT contains the symbol at that



9	0	0	0	0
9	1	0	0	1
11	1	0	0	1
13	0	0	0	0

Figure 4.2: *bvSFM* column for the symbol G. Each *bvSFM* entry has a counter (blue boxes, left) and a bit vector (right, 4 bits). $C[G] = 9$ is already added to the counters, so there is no need to add it during the LF operation as in Equation 2.3.

position, as shown in Figure 4.1d. Instead of iterating the BWT to count the occurrences of a base, we just need to count the number of bits set in the bit vector. This operation can be efficiently implemented by the *popcount* instruction. We term $bvSFMentry$ the set of sampled counters and their corresponding bitmaps in a *rOcc* row (see Figure 4.1d). As proposed in Chacón et al. [14], a preprocessing step adds the C values to the *rOcc* counters in order to avoid a C read and an addition operation for each LF.

bvSFM example

To illustrate how this approach works, let us consider the FM-Index shown in Figure 4.1b. We take as example the last LF operation of the example shown in Figure 2.5b. The current values of the pointers are $sp = 7$ and $ep = 9$, and the next symbol to be processed by the backward search algorithm is $Q[i] = G$. Figure 4.2 shows the *bvSFM* column corresponding to the symbol G.

The two LF operations, one for *sp* and one for *ep*, required for each symbol *s* in the query are performed as follows:

- Calculate the *bvSFM* entry indexes: p/d ($7/4 = 1$ for *sp*, and $9/4 = 2$ for *ep*).
- Load the *bvSFM* entries, which contain the sampled counters $rOcc[p/d, s]$ ($rOcc[1, G] = 9$ for *sp* and $rOcc[2, G] = 11$ for *ep*), and its corresponding bit-vectors in contiguous memory locations (1001 for both *sp* and *ep*).
- Calculate the bit-vector indexes: $p \bmod d$ ($7 \bmod 4 = 3$ for *sp* and $9 \bmod 4 = 1$ for *ep*).
- Select the bits to perform the $popcount$. These bits, shaded in Figure 4.2, are obtained by performing a bitwise *and* operation between the bit vector and a mask of $< p \bmod d >$ 1s (1110 and 1000 masks for *sp* and *ep*, respectively).

- Perform a *popcount* over the selected bits ($popcount(100_2) = 1$ for *sp* and $popcount(1_2) = 1$ for *ep*).
- The number of occurrences of the symbol G up to each pointer position is computed by adding the sampled counter to the result of the *popcount* operations: $rOcc[p/d, s] + popcount(masked\_bit\_vector)$ $(9 + popcount(100_2) = 9 + 1 = 10$ for *sp*, and $11 + popcount(1_2) = 11 + 1 = 12$ for *ep*).

The values of *sp* and *ep* will be 10 and 12, respectively.

4.1.4 State-of-the-Art Summary

The FM-Index algorithm is widely used in genomic tool-flows, as it enables linear-cost exact matching over large references with manageable memory footprints. The most efficient way to increase computational throughput is by increasing the number of symbols searched per step (k). However, it quickly imposes prohibitive memory consumption. With $k = 2$, the full FM-Index is almost 200 GB for the human genome. Therefore, sampling techniques must be used to lower the memory footprint, trading-off some of the performance gains, as additional computation is necessary. Finally, *bvSFM*, a state-of-the-art technique, proposes an FM-Index layout that uses bitmaps to speedup this additional computation while improving memory locality.

The *bvSFM* structure of an FM-Index with a sampling rate of $d = 64$ and $k = 2$ steps occupies 12 GB. This was found to be the best configuration, as increasing k further led to worse memory management. Moreover, increasing the sampling rate to keep the memory footprint in check requires a chain of dependent *mod* and *popcount* operations that degrade performance.

We selected a CPU-based implementation rather than a GPU-based one because current state-of-the-art FM-index implementations are predominantly CPU-oriented. Widely adopted aligners such as BWA-MEM2 [15], Bowtie2 [26], and HISAT2 [122] all rely on CPU-based FM-index implementations, as these are mature, stable, and fully integrated into complex read-mapping workflows. In practice, FM-index involves highly irregular, pseudo-random memory access patterns, which are difficult to optimize efficiently on GPUs and often negate the expected benefits of massive parallelism. Although GPU implementations such as NVIDIA's NVBIO library [123] and research systems like SOAP3-dp [124] or CUSHAW2-GPU [125] demonstrate promising speedups, they remain relatively specialized and are not yet widely adopted in production pipelines.

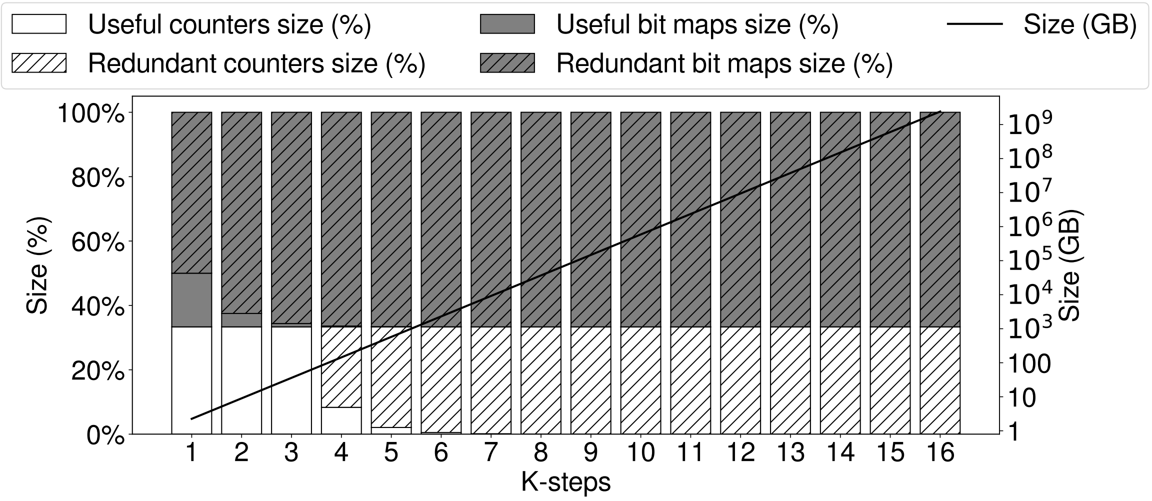


Figure 4.3: Size, useful, and redundant data of the *bvSFM* structure for $d = 64$ and different values of k.

4.1.5 Memory Footprint Analysis

The FM-Index memory usage grows exponentially with k. However, the difference between two consecutive rows of the *Occ* structure is just on one of the counters (see Figure 2.4c). For instance, with $k = 5$, each *Occ* row has 1024 counters (4^k) but only one changes its value from one row to the next. Thus, the fraction of *useful counters* is $1/4^k$, i.e., one per row. With a sampled FM-Index, at most d counter increments occur over consecutive rows of the *rOcc*. Hence, the maximum fraction of *useful counters* is $d/4^k$. A key observation is that the maximum number of *useful counters* in a *rOcc* row is equal to d, regardless of the value of k. Therefore, as k grows, the fraction of *useful counters* decreases exponentially.

For $bvSFM$, the 4^k bitmaps in a $bvSFMentry$ have a constant number of set bits, equal to the sampling factor d (length of each bitmap). That is, if $BWT[i]$ is equal to the symbol s, the bitmap of the symbol s contains a set bit (1) in the i^{th} position, and the rest of the symbols have an unset bit (0) in the i^{th} position of their bitmaps (see Figure 4.1d). The fraction of *useful bits* in the bitmaps is $1/4^k$, and the total number remains constant regardless of the value of k. For instance, with $k = 10$ and $d = 64$, each $bvSFM$ entry contains 64 Mb (64×4^{10} bits), of which only 64 are set.

Figure 4.3 shows how the size of a sampled ($d = 64$) $bvSFM$ structure increases with k (black line, right y-axis), quickly becoming intractable. Stacked bars show the percentage of the $bvSFM$ size dedicated to counters (white background) and bitmaps (gray background). Moreover, we also show the percentage of the $bvSFM$ size allocated

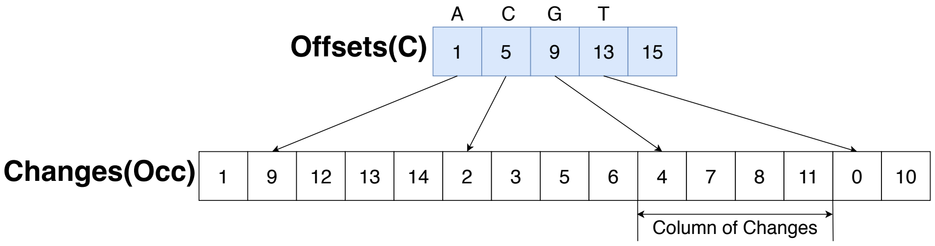


Figure 4.4: *COFI* data structures: Offsets and Changes.

to useful and redundant counters, and to useful and redundant bits in the bitmaps. We can observe that the amount of redundant data quickly becomes dominant. Almost all the data is redundant for $k \geq 5$, because the amount of useful data stays constant while the size of the data structure increases. Therefore, there is an opportunity to devise a new data structure and an accompanying algorithm that can perform large k-steps if we are able to store only useful data.

4.2 COFI: Compressed Sparse FM-Index

A large k leads to better throughput. However, memory usage grows exponentially with k, quickly becoming unmanageable. By leveraging the observation that the amount of useful information stored in the FM-Index remains constant, we propose to use a large *k-step* and compress the sparse FM-Index information while keeping a low overhead to reconstruct the index from the compressed data structures. The following subsections introduce *COFI*, a **CO**mpressed **F**M-**I**ndex for large K-steps.

4.2.1 COFI Data Structures: Offsets and Changes

If we take as a reference the full FM-Index *Occ* structure (see Figure 2.4c), we propose to store only the row indexes where a counter changes for each symbol in the alphabet. In other words, a column would now store the indexes where its corresponding symbol appears in the BWT. We call this new structure *Changes*. The size of *Changes* is constant for any value of k: there are as many elements as in the BWT. The size of the columns is now variable, we store them consecutively as shown in Figure 4.4. In order to find where each column starts and ends, we can reuse the C structure, no modifications are needed. In *COFI*, we call this structure *Offsets* despite being the same as the original C. If we count the occurrences of the symbol s, the limits of its column are *Offsets[s]* and *Offsets[s+1]*. The final layout of *COFI* is shown in Figure 4.4.

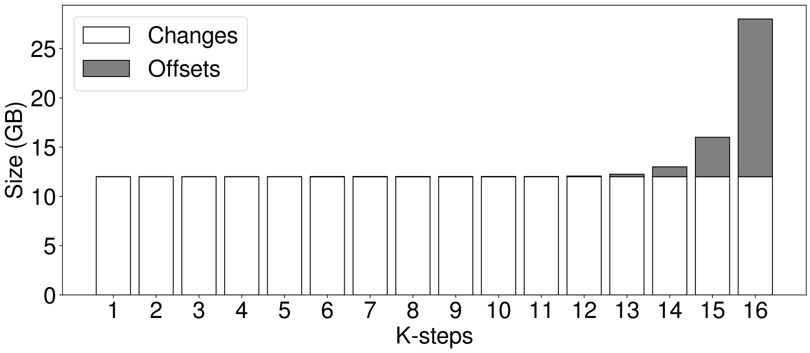


Figure 4.5: *COFI* data structures size for various values of k.

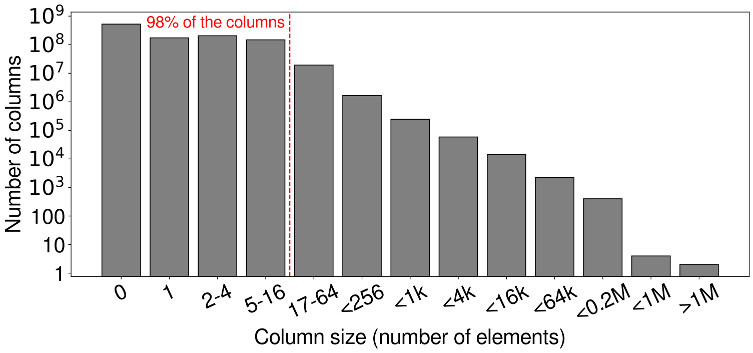


Figure 4.6: Histogram of column sizes for the human reference GRCh38. Both axis are in logarithmic scale.

4.2.2 Memory Footprint Analysis

In previous proposals, the size of the main data structures increases exponentially with k. In *COFI*, the size of *Offsets* also increases exponentially at the same rate. However, it is a small structure, only 4 entries for $k = 1$, or 16 entries for $k = 2$. On the other hand, *Changes* is an array with the same number of elements as the BWT, keeping its size manageable for any value of k. Figure 4.5 shows the size of the proposed data structures for a full human genome reference (GRCh38, 3 giga base pairs) and different values of k. *Changes* occupies a constant amount of 12 GB. For $k = 15$, *Offsets* occupies 4 GB, totaling 16 GB. As a comparison point *bvSFM* requires 12 GB with $k = 2$ and $d = 64$.

An important consideration of COFI is that the columns in the *Changes* structure have different sizes, leading to non-uniform search times for different symbols. Figure 4.6 shows the distribution of the column sizes for the human reference GRCh38 with $k = 15$. The X axis represents the column size in bins that follow a logarithmic scale. The Y axis, also in logarithmic scale, indicates the number of columns within each bin. The size of most columns is smaller than that of a 64-byte cache block: 98% of the columns have 0

Algorithm 1 Backward Search in COFI

```
1: function BACKWARD_SEARCH(Q[N], Changes, Offsets)
2:     sp = Offsets[Q[N-1]]
3:     ep = Offsets[Q[N-1]+1]
4:     for i = N − 2 to 0 do
5:         left = Offsets[Q[i]]
6:         right = Offsets[Q[i]+1]
7:         sp = find_col_index(Changes, left, right, sp)
8:         ep = find_col_index(Changes, left, right, ep)
9:     return $sp, ep$
```

to 16 elements. Nevertheless, large columns can impose a large overhead and are more likely to appear during the search, as we show in our evaluation. Hence, the proposed search algorithm has to take large columns into account, even if they represent a small fraction of the total number of columns.

4.2.3 Backward Search in COFI

To perform the two LF operations for symbol *s* and pointers *sp* and *ep* in *COFI*, the following steps need to be taken:

1. Load *Offsets[s]* and *Offsets[s+1]* to obtain the start and end positions in the *Changes* array of the column corresponding to symbol *s*.

2. Find in *Changes* the first element of the column that is equal or greater than the values of *sp* and *ep*.

3. Return the indexes of the *Changes* array for the found elements.

Algorithm 1 shows the pseudocode to perform a backward search for one sequence and $k = 1$. After initializing *sp* and *ep* in lines 2 and 3, two LF operations are performed, one for *sp* and one for *ep*, for each one of the remaining symbols. First, the start and end indexes of the column are read in lines 5 and 6. Then, the first element greater or equal than *sp* and *ep* is searched. The *find_col_index* function in lines 7 and 8 returns the index of the found element in *Changes*. In the case of *sp*, this index is the starting position of the found sequences so far in matrix *M* (defined in Section 2.3.1), while *ep* indicates the first sequence that is not a match. Therefore, all sequences between *sp* and *ep* in *M* would currently be query matches. Finally, we return the pointers with the indexes that delimit the found sequences in *M* (line 9).

As shown in Figure 4.6, the size of the columns represented in *Changes* can be significantly large. Therefore, in order to perform the search operation quickly, the function *find_col_index* implements a simple binary search algorithm. The current implementation takes advantage of the fact that columns are already sorted in ascending order. The function receives as parameters:

- $array$: array to search in.
- $left$: left limit of the binary search.
- $right$: right limit of the binary search.
- $pointer$: element to search.

As mentioned above, *sp* and *ep* keep track of the found sequences; therefore, after each iteration the distance between both pointers ($d = ep - sp$) will decrease or remain the same. Thus, we can modify the left and right limits in line 7 to perform the search over a subset of the column.

We can illustrate this optimization with the query shown in Figure 2.5. At the beginning of the second loop iteration, the values of *sp* and *ep* are 7 and 9, respectively (see Figure 2.5b). This indicates that $d = 2$ contiguous rows start with CC, a suffix of the pattern being queried. In the last iteration, once the new value of *sp* is calculated in line 6 of the algorithm ($sp = 10$, see Figure 2.5c), the $[left, right] = [9, 13]$ range for the *ep* search can be constrained: (i) we can set the left limit to the recent computed *sp* value, $left = sp = 10$, and (ii) we can set the right limit according to the maximum distance between *sp* and *ep*: $right = sp + d = 10 + 2 = 12$. This is possible because *Changes* is sorted in ascending order and has no repeated elements. Note that this optimization is very effective because the distance between the left and right limits becomes smaller as the query progresses. This optimization can be implemented by changing line 7 of Algorithm 1 to:

$$ep = find\_col\_index(Changes, sp, sp + d, ep) \tag{4.2}$$

where d is calculated as $d = ep - sp$ at the beginning of the *for* loop.

4.2.4 COFI Example

We show how the previous example shown in Figure 4.2 for *bvSFM* takes place in COFI. As a reminder, the initial values of the pointers are $sp = 7$ and $ep = 9$, and the next symbol

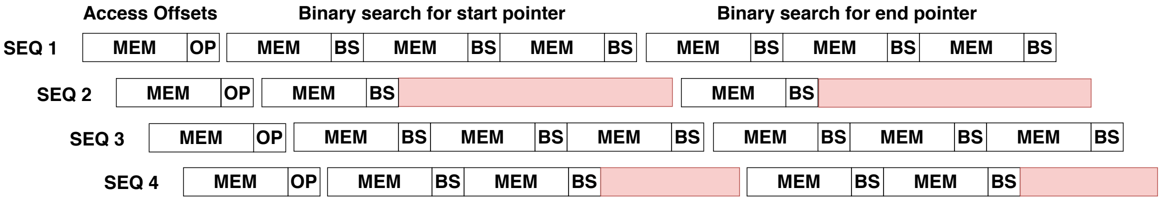

Access Offsets
Binary search for start pointer
Binary search for end pointer
SEQ 1
SEQ 2
SEQ 3
SEQ 4
MEM
OP
BS

Figure 4.7: Depicts four interleaved sequences to hide memory latency. *OP* denotes the time spent computing the limits of the columns, *MEM* is the time spent waiting for a response from memory, and *BS* stands for the time spent on binary search. *BS* depends on column size, therefore, some sequences finish this phase before others (red boxes).

to be processed by the backward search algorithm is $Q[i] = G$. Figure 4.4 shows COFI data structures with the pertinent information for column G. The operations followed in COFI are:

- Read the start and end indexes of the G column in *Changes*: *Offsets[G]* = 9 and *Offsets[G+1]* = 13.
- Apply binary search over the G column to find the elements greater or equal than *sp* and *ep*. The elements found have values 7 and 11 for *sp* and *ep*, respectively.
- Update *sp* and *ep* with the indexes of the found elements, i.e., 10 and 12, respectively.

Although COFI has a higher computational complexity in typical cases, it enables larger *k-steps*. We can increase k to 15 in COFI while keeping a manageable FM-Index size of 16 GB.

4.2.5 Performance Optimizations

The memory access pattern to perform an LF calculation does not present good locality. By inspecting Algorithm 1, we can observe that the LF loop first performs an access to *Offsets*, which is dependent on the current symbol, and then performs binary search over a subset (column) of the *Changes* array. The accessed column is unlikely to be visited in the near future in the current or subsequent searched sequences, precluding temporal locality. In addition, the access pattern in binary search does not present spatial locality. Therefore, each memory access is likely to be long latency as it will miss in the caches.

Sequence interleaving: in order to hide memory latency, as previously implemented in the *bvSFM* proposal, we search multiple sequences at the same time, overlapping long-latency memory requests. Figure 4.7 shows four overlapped searches in COFI, i.e., each iteration of the LF loop operates over four different query sequences. As can be seen in the figure, the size of the *Changes* column determines the number of binary search steps for each pointer. Therefore, overlapped searches can finish at different steps in the binary search algorithm.

Software prefetching: with the same objective to reduce memory access latency, we also employ a simple software prefetching scheme. When processing the current symbol, prefetch operations are issued for the next symbol to retrieve from memory the necessary *Offsets* elements. Additionally, the first pivot point for *sp*'s binary search is prefetched once the column to be accessed is known.

Conditional moves: finally, COFI's LF computation is slightly more complex since it involves executing binary search. Binary search is known to exhibit poor branch prediction performance, as branches are difficult to predict, i.e., there is a 50% chance to take the branch. A branch misprediction flushes the pipeline and causes a significant performance penalty in processors with deep pipelines. Conditional move instructions can be an alternative to branches. These instructions write the contents of one register over another only if the defined predicate value is true, and can be executed speculatively on a processor's pipeline without the associated predication of branches. Therefore, in order to avoid branches, we have implemented our binary search algorithm with conditional moves.

Section 4.4.2 evaluates each optimization separately, and reports their contribution to performance improvements.

4.3 Performance Analysis

A common metric used to measure the theoretical peak performance a workload can exhibit on a particular target machine is the *arithmetic intensity* [126]. This metric defines the number operations, typically floating-point operations, performed per byte brought from off-chip memory. In our study, we will employ *search intensity (SI)* for this purpose. Defined as the number of LFs performed per byte brought from memory [49]. Equation 4.3 computes the SI of a search for a k-step FM-Index:

$$SI = \frac{2 \times k}{\alpha \times B} \tag{4.3}$$

where $2 \times k$ is the number of LFs performed per iteration, α is the average number of cache misses per iteration, which depends on the algorithm, and B is the cache block size (typically of 64 bytes).

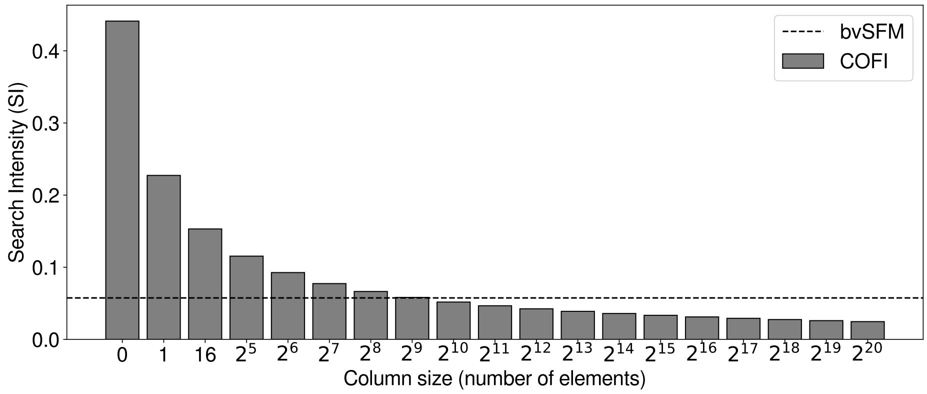


Figure 4.8: Search intensity for different column sizes.

Backward search algorithms are typically memory bound, i.e., little computation is done per byte brought from memory. Therefore, increasing search intensity is paramount in order to increase performance. In the case of $bvSFM$, it has $k = 2$ and computes four LFs per iteration. We need to bring two cache blocks, one for *sp* and another one for *ep*. The *ep* pointer could produce a cache hit if it is close enough to *sp*. For this algorithm, the reported α is 1.088 [49]. Therefore, $bvSFM$ has a search intensity of $\frac{2\times2}{1.088\times64} = 0.057$.

In COFI, SI is input dependent as iterations over large columns result in low SI values due to additional memory accesses. Contrarily, iterations over small columns result in high SI values. To calculate α, we have to take into account the number of memory accesses in *Offsets* and *Changes*.

For *Offsets*, two 4-byte accesses are required, one for the left index and another one for the right index. Since both values are stored in contiguous memory positions, the probability of having a cache miss for the second element is 1/16. Hence, an average of 1.0625 memory operations are performed due to the two *Offsets* accesses. For *Changes*, the number of accesses depends on the number of 4-byte elements in the column, *ne*. When *ne* is 0 or 1, the number of *Changes* accesses is 0 and 1, respectively. When *ne* is 16 or less, the worst case results in two cache misses, and each time we double the size of the column, one extra cache miss has to be added. Hence, when *ne* is larger than 16, the number of cache misses is $\lceil log_2(ne) - 2 \rceil$ for the worst case. Using $k = 15$, SI is $\frac{2\times15}{(1.0625+a)\times64}$, where a is the number of cache misses caused by the *Changes* accesses, and it is defined in Equation 4.4.

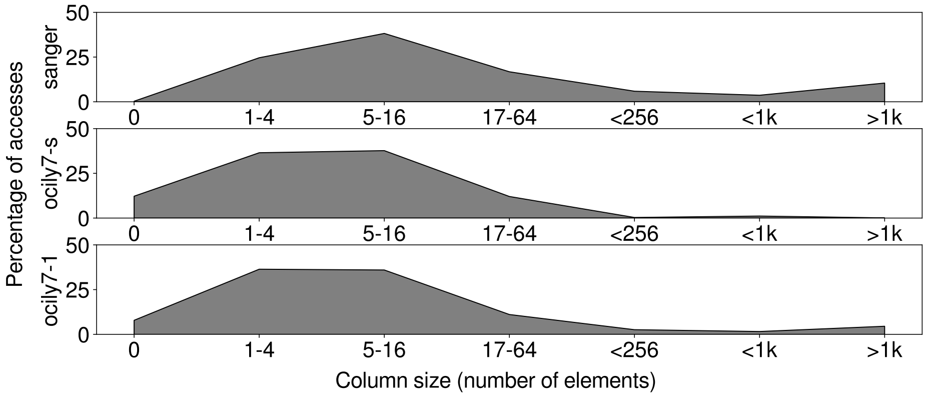


Figure 4.9: Percentage of accesses per column size.

$$a = \begin{cases} ne, & \text{if } ne = 0,1 \\ 2, & \text{if } 2 \leq ne \leq 16 \\ \lceil log_2(ne) - 2 \rceil, & \text{otherwise} \end{cases} \tag{4.4}$$

Figure 4.8 shows the *bvSFM* SI (dotted black line) and COFI's SI (grey bars) for different column sizes. Note that the SI numbers for COFI represent the worst case behaviour of the binary search algorithm, as the searched element could be found in one of the pivot points. We can see that COFI presents higher SI than *bvSFM* for values of *ne* lower than 1024 elements. This metric is directly correlated with performance.

In order to calculate SI for each input, we measure the size of the columns that the algorithm accesses during execution. We can see the percentage of accesses per column size in Figure 4.9 for the inputs *sanger*, *ocily7-s* and *ocily7-1* and the reference genome GRCh38. For *sanger*, we can observe that there are no accesses to empty columns, this is because *sanger* was extracted from GRCh38 without introducing errors. The percentage of accesses to columns bigger than 1024 elements is 10.54%, 0.1% and 4.55% for *sanger*, *ocily7-s* and *ocily7-1*, respectively.

By using the SI values from Figure 4.8 and the number of accesses per column type, as shown in Figure 4.9, we plot the calculated SI for each input in Figure 4.10. We can observe that the SI in COFI is significantly higher than that of *bvSFM*, up to 3.28× for *ocily7-s* input. This is likely to lead to better performance as the workload is memory bandwidth bound.

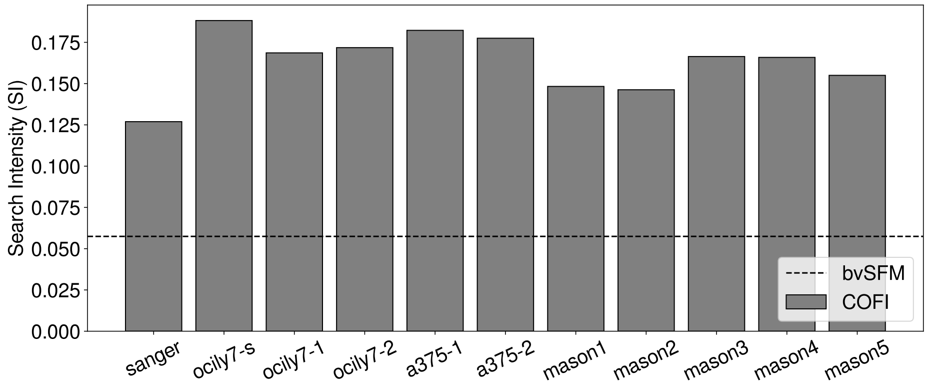


Figure 4.10: Search intensity for each input.

4.4 Evaluation

For the evaluation of COFI, we use two hardware platforms: SKX and KNL (see Table 3.1). It is worth mentioning that KNL has 16 GB of high bandwidth MCDRAM, which is ideal for fitting a 15-step COFI structure of a human genome. To see these systems in more details, see Section 3.2.

First, we present the inputs used for the evaluation. Then, we evaluate the performance impact for each of the optimizations described in Section 4.2.5. We compare COFI against a state-of-the-art proposal, *bv*SFM [49]. *bv*SFM is a *2-step 64 sampled FM-Index* that uses a bitmap to recover the original *Occ* values (see Section 4.1.3). We discuss how inputs affect performance by relating time spent in large columns and SI with performance. Finally, we describe other experiments we performed and our findings.

4.4.1 Inputs Sequences

We use 11 representative sets of inputs sequences selected from different use cases; including reads from particular cell lines, as well as sequences generated using Mason [120], a read simulator.

- *sanger*: it has been extracted from the GRCh38 reference using Mason, simulating the Sanger process [127] without errors. This input has been also used to evaluate *bv*SFM [49].

- Five inputs coming from real reads made by an Illumina HiSeq 2000 system, which outputs sequences with a length of 101 bases.

  - *ocily7-s*: reads from the cell line OCI-LY7 over RNAs smaller than 200 nucleotides [128].

Table 4.1: Error rates for Mason inputs.

Inputs	**Modifications**	**Insertions**	**Deletions**
mason1	3%	0%	0%
mason2	1%	1%	1%
mason3	6%	0%	0%
mason4	0%	6%	0%
mason5	0%	0%	6%

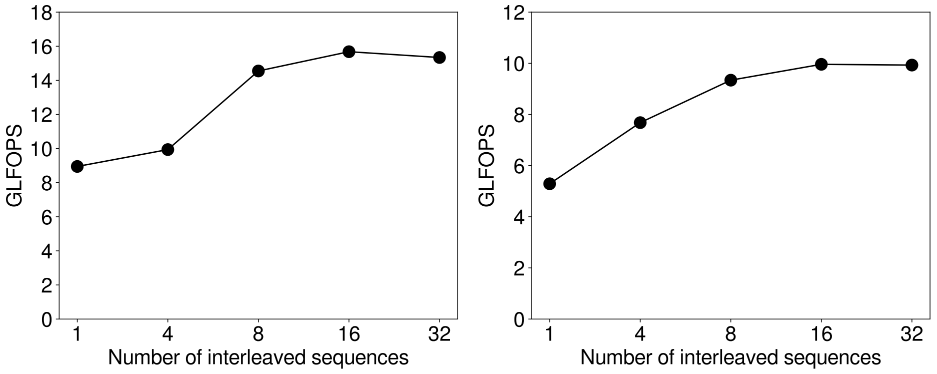
Table 4.2: Number of sequences, length of each sequence and occurrences in GRCh37 and GRCh38 for each input.

Input	**Nº seqs**	**Length**	**Occu. GRCh37**	**Occu. GRCh38**
sanger	20M	200	15.49M	46.71M
ocily7-s	35.21M	101	13.09M	26.41M
ocily7-1	70.38M	101	46.04M	54.45M
ocily7-2	70.89M	101	34.27M	39.21M
a375-1	115.32M	101	61.54M	79.81M
a375-2	115.27M	101	57.11M	106.96M
mason1	10M	150	103,432	384,139
mason2	10M	150	109,470	385,207
mason3	10M	150	898	3,431
mason4	10M	150	758	3,402
mason5	10M	150	1,005	3,407

  - *ocily7-1* and *ocily7-2*: reads from the same cell line OCI-LY7, but over RNAs greater than 200 nucleotides [129].
  - *a375-1* and *a375-2*: reads from the cell line A375 over RNAs greater than 200 nucleotides [130].

- *mason{1..5}*: we use Mason over GRCh38, simulating an Illumina system to generate five inputs. We replicate the simulations described on the appendices of Alser et al. [58] with adjusted error rates. Table 4.1 shows the error introduced for each simulation.

Table 4.2 shows the total number of sequences, their length, and the occurrences on each reference for all the inputs. The number of occurrences is higher for GRCh38, as it contains around two million bases more than GRCh37, and all inputs generated using Mason employ GRCh38 as input. In addition, we note that the number of occurrences is



(a) KNL

(b) SKX

Figure 4.11: Average performance across all inputs for different numbers of interleaved sequences.

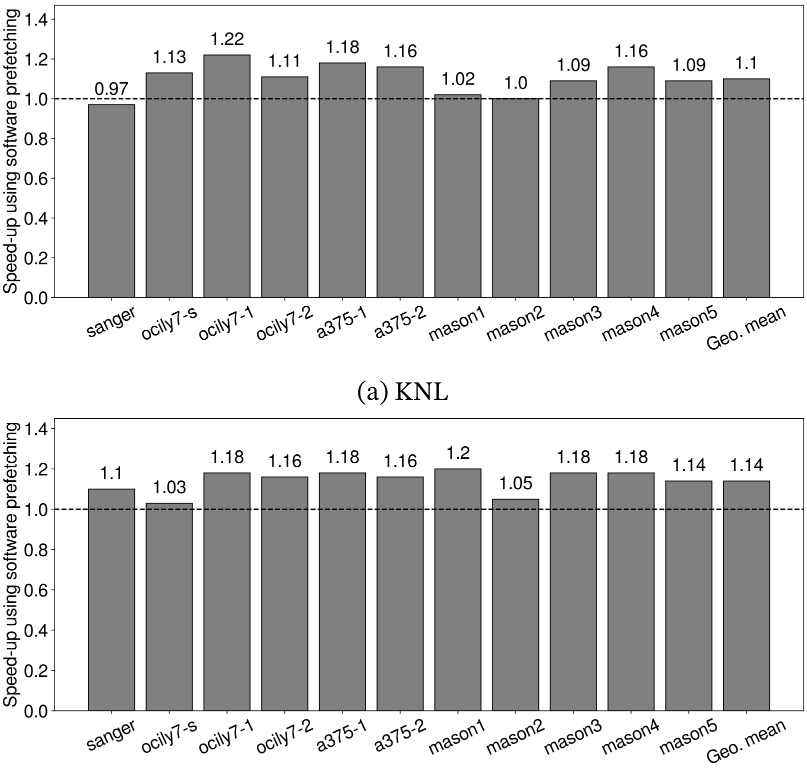
higher for *mason1-2* than for *mason3-5* because the error introduced is lower, 3% and 6% respectively (see Table 4.1).

Our inputs are based on short reads because FM-Index methods dominate in this scenario. However, COFI would perform similarly for any read length, as the algorithm and data structures are read length agnostic, and the performance characteristics would not change.

For each experiment, we perform 128 executions. For each execution, we search all sequences on the target reference. To measure the throughput, we use the number of LF operations performed per second (LFOPs/s). We discard the first execution in order to avoid cold start effects of hardware structures. We calculate the arithmetic mean with the rest of the executions.

4.4.2 Optimizations

We evaluate the performance impact when applying each of the three optimizations described in Section 4.2.5. Therefore, we analyze the impact on the number of interleaved sequences, the benefits of the software prefetching scheme, and the impact of conditional moves in binary search. We use software prefetching and conditional moves in these experiments except when evaluating their own impact. For this set of results, we just show numbers with the GRCh38 reference, since results with GRCh37 are similar and lead to the same conclusions.



(a) KNL

(b) SKX

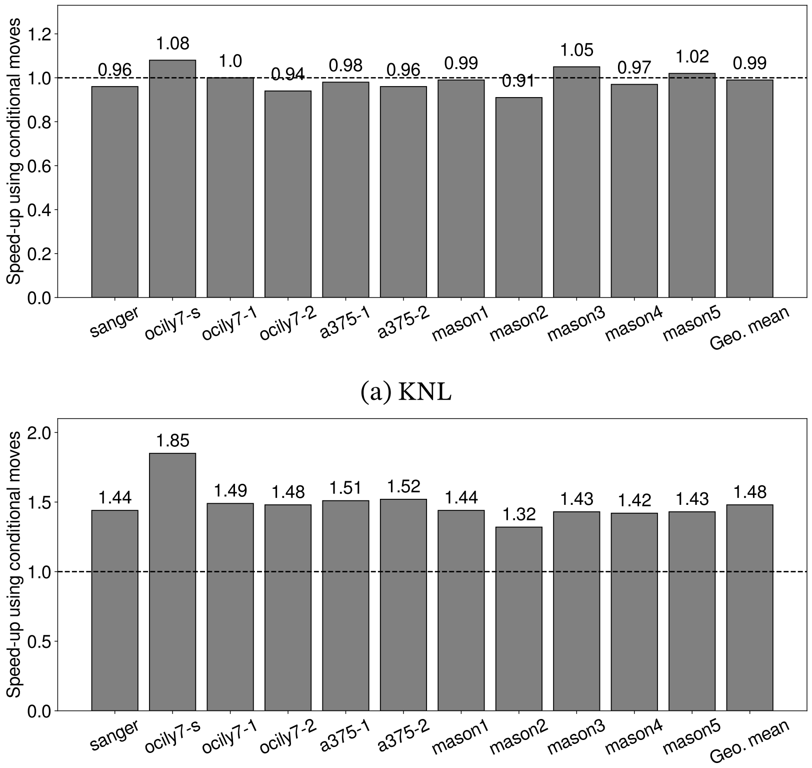
Figure 4.12: Speed-up when using software prefetching.

Number of Interleaved Sequences

Figure 4.11 shows average performance across all inputs when changing the number of interleaved sequences from 1 to 32. In both test machines, KNL and SKX, we obtain the best average performance with 16 interleaved sequences. After 16 sequences, performance remains stable because the memory subsystem is already saturated. A few inputs behave slightly better with 8 or 32 interleaved sequences; however, to be consistent we employ 16 interleaved sequences for all inputs from now on.

Software Prefetching

Figure 4.12 shows the speedup when using the described software prefetching scheme. As previously specified, we are prefetching accesses to the *Offsets* vector and the first pivot of the binary search, but not the remaining pivots. On average, we obtain a speedup of 10% and 14% for KNL and SKX, respectively. Therefore, we consider this optimization useful and we will apply it throughout the rest of the evaluation.



(a) KNL

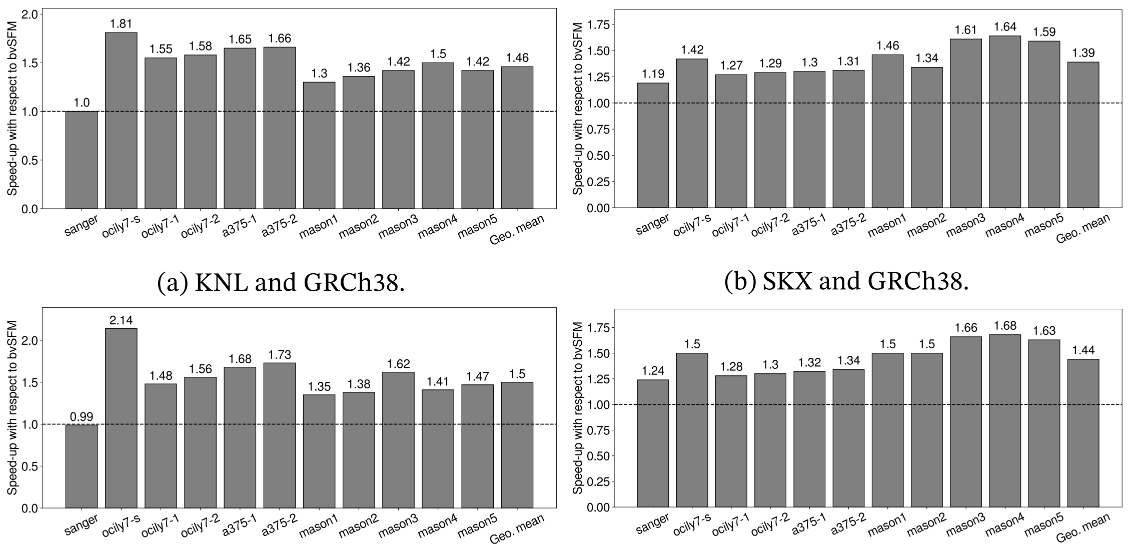
(b) SKX

Figure 4.13: Speed-up when employing conditional moves in the binary search algorithm.

Conditional Moves

Figure 4.13 shows the speedup of using conditional moves in the binary search algorithm. We can observe a significantly different impact on each of the test machines. For SKX, we obtain speedups of up to 1.85× (1.48× on average). This is because the Skylake core is heavily penalized by branch mispredictions due to its aggressive speculative execution on predicted branches, which in this case have a high probability of being mispredicted. These mispredictions are likely to perform additional memory accesses on the wrong execution path, further hindering performance. Prior work already showed that conditional moves are an effective way to speedup binary search algorithms for the mentioned reasons [131].

For KNL, however, performance is not altered significantly, leading to a marginal 1% slowdown on average. Branch mispredictions do not impose such a large penalty in KNL due to a shallower pipeline and less aggressive speculative execution. Since this optimization fails to deliver performance improvements in KNL we chose not to enable it from now on in KNL experiments. It will be used only on SKX experiments.



(a) KNL and GRCh38.

(b) SKX and GRCh38.

(c) KNL and GRCh37.

(d) SKX and GRCh37.

Figure 4.14: Speed-up of COFI with respect to *bv*SFM for all inputs, two references, and two test machines.

4.4.3 Throughput

Figure 4.14 shows the speedup of COFI with respect to the best performing *bv*SFM (*2-step 64-sampled*), for references GRCh37 and GRCh38. For KNL, we observe speedups of up to 1.81× (1.46× on average) when using the GRCh38 reference. Performance differences across inputs come from the different column access profiles in the *Changes* data structure (see Figure 4.9 for examples). The obtained results correlate well with these profiles. For example, *ocily7-s* presents the lowest amount of accesses to large columns and attains the best performance. On the other hand, *sanger* has more accesses to larger columns and COFI has the same performance as *bvSFM*. We present further details on input sensitivity in Section 4.4.4. The results for the GRCh37 reference show the same trends, with an average performance improvement of 1.5×.

For SKX, we obtain speedups of up to 1.64× (1.39× on average) when using the GRCh38 reference. Here we also find similar trends for the different inputs. However, since the binary search component of the algorithm performs significantly better in SKX due to the conditional move optimization, we can see that performance differences between inputs with different column access profiles (i.e., *sanger* and *ocily7-s*) are much narrower. In fact, COFI is 1.19× faster than *bvSFM* with *sanger* on SKX. Again, similar trends can be seen for the CRCh37 reference, with an average improvement of 1.44×.

Table 4.3 shows the raw throughput measured in GLFOPs/sec for each input and the GRCh38 reference genome. In KNL, the *bvSFM* version maintains a constant per-

Table 4.3: Raw throughput in GLFOPs/sec using GRCh38.

	KNL		SKX	
Input	***bv*SFM**	**COFI**	***bv*SFM**	**COFI**
sanger	10.26	10.25	5.65	6.73
ocily7-s	11.69	21.16	12.55	17.81
ocily7-1	11.25	17.45	7.58	9.59
ocily7-2	11.11	17.60	7.56	9.74
a375-1	11.46	18.93	7.97	10.39
a375-2	11.52	19.16	7.98	10.44
mason1	9.61	12.51	5.86	8.57
mason2	10.00	13.59	5.87	7.86
mason3	9.92	14.10	5.83	9.42
mason4	10.05	15.07	5.87	9.65
mason5	10.01	14.25	5.88	9.36
Geometric mean	10.60	15.51	6.94	9.68

formance among the inputs. However, in SKX, the *ocily7-s* input shows a much higher throughput than the others. This is due to two reasons. Firstly, *ocily7-s* contains a high number of repeated sequences. Secondly, SKX has larger last-level cache slices per core and it can exploit data locality for these repeated sequences.

4.4.4 Input Sensitivity

There is a noticeable difference on performance depending on the input. We have previously mentioned that performance is correlated with the SI associated to an input, and consequently, with the number of accesses for each column size.

In Figure 4.15, we can see the percentage of time spent on each column size for *sanger*, *ocily7-s* and *ocily7-1* inputs. In order to obtain representative results of the cost to perform a search for a given column size, we perform this experiment using one thread and one interleaved sequence on the KNL using the GRCh38 reference. Multiple interleaved sequences would slowdown searches over small columns when they interleave with searches over big columns.

As expected, the time spent searching on large columns is directly correlated to the performance differences seen across inputs. That is, inputs that spend more time on larger columns obtain lower performance. *sanger* is the input that spends more time in large columns, 37.5% of the time spent on columns larger than 1024 elements, while *ocily7-s* and *ocily7-1* spend 0.81% and 22.58%, respectively.

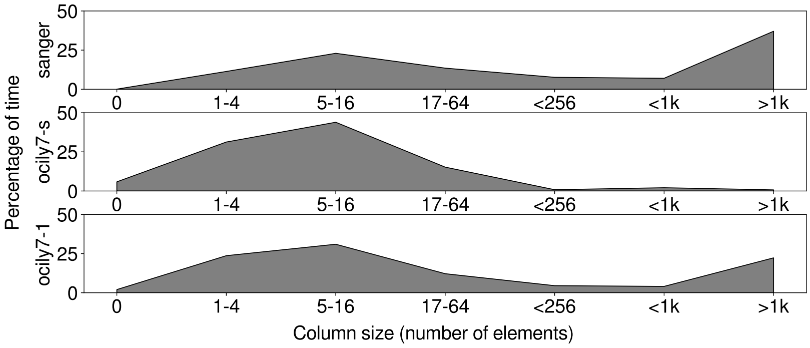


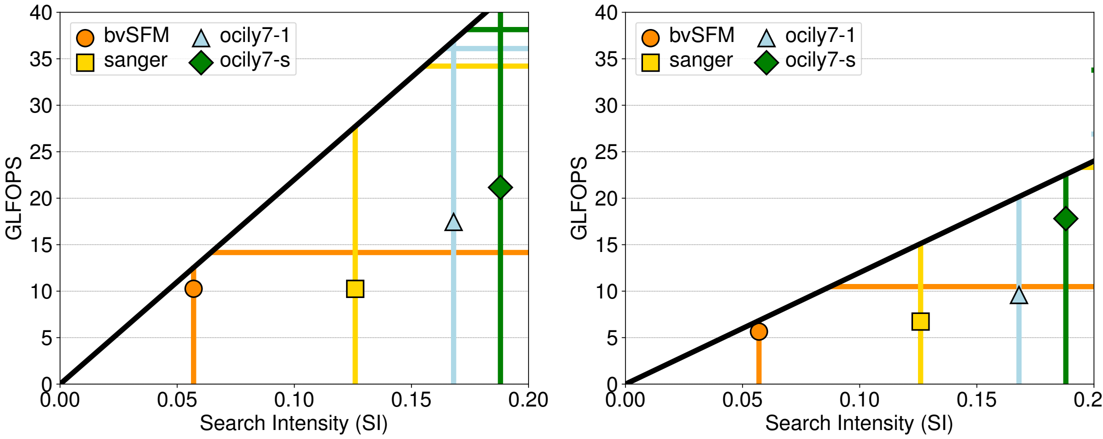
Figure 4.15: Execution time spent on each column size.

An important observation is that some exact search algorithms avoid or even filter-out large columns [57]. The reason is that these columns contain sequences that appear a lot of times, and they provide redundant locations that other columns already find. It is also the reason our inputs based on real read data from cell lines (e.g., *ocily7-s* and *ocily7-1*) access less large columns in percentage than *sanger*, which is produced using a read simulator over the entire genome reference. Therefore, COFI could further benefit from this fact if columns with small SI are filtered-out.

Finally, to illustrate how higher SI improves performance in COFI, we show the roofline models [126] for both test machines in Figure 4.16. Our roofline model ties together throughput, SI, and memory performance in a two-dimensional graph. The Y-axis is GLFOPS per second (throughput) and the X-axis is SI, i.e., LF per byte of off-chip memory traffic. Theoretical ceilings can be derived using hardware specifications. The diagonal black line (memory ceiling) depends on memory bandwidth available and determines the maximum performance achievable for a given SI value. Horizontal lines denote compute ceilings for each input, calculated by dividing the number of executed instructions to perform all LF operations and the number of instructions each machine can retire per cycle. Note that the compute ceilings are input dependent.

We can observe that *bvSFM* performs close to the memory ceiling in both machines, and it is therefore memory bound rather than compute bound. However, COFI manages to increase SI significantly, and for the *ocily7-s* input we are able to break the computational ceiling of *bvSFM* on both machines.

Even though SI has increased, the achieved performance with COFI is far from the theoretical compute ceilings. Therefore, we can conclude that the main performance limitation has now shifted from memory bandwidth to the algorithm itself, due to branches and data dependencies in the code.



(a) KNL roofline.

(b) SKX roofline.

Figure 4.16: Roofline models for *bvSFM* (input independent) version and three inputs of COFI.

4.4.5 Discussion

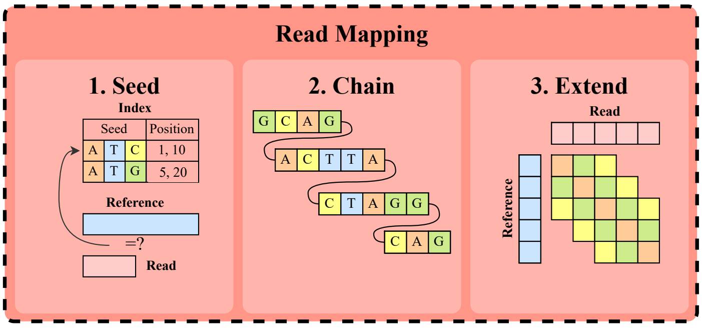
We have tried to apply other ideas to improve performance even further. Among these ideas, we highlight two: (i) to rearrange the columns of *Changes* in order to be cache-friendly, and (ii) the use of larger *k-steps* using SKX.

When performing binary search we can imagine the columns of *Changes* as trees. When the column is large, cache misses arise when traversing the first levels of the tree. In order to avoid this problem, we thought of two approaches. Firstly, to place the children together as in the Eytzinger layout [131]. This would allow us to apply software prefetching to several levels in advance. The main drawbacks that made this implementation not feasible are irregular column sizes, and that the algorithm to recover the original index becomes too costly. Secondly, to group several levels of the tree together [132]. We place n levels of the tree at the center of the column, and perform binary search (in this case n-ary search) for n pivots knowing that they are close in memory. Then, we have to search into one of the 2^n subtrees applying recursion. We have a threshold from which we do not rearrange the column, at which point we use the normal binary search algorithm. The problem is that with this layout we cannot limit the end pointer search, because the columns are not sorted in ascending order. Therefore, we have to repeat the binary search for the whole column for *ep*, precluding any performance improvements.

We also did experiments using $k = 16$ and $k = 17$ for SKX, we already fill up completely the MCDRAM of the KNL with $k = 15$. In both versions, we obtain a small slowdown due to the size of the FM-Index causing a large amount of TLB misses, as the amount TLB entries for huge pages is exceeded.

Chapter 5

Porting and Optimizing BWA-MEM2 using the Fujitsu A64FX Processor



Seed	Position
A T C	1, 10
A T G	5, 20

We port to ARM and optimize ***BWA-MEM2****, a widely used* ***read-mapping tool***

When we began working on this contribution, the A64FX processor [98] had only recently been released. The A64FX was the first processor to implement the Scalable Vector Extension (SVE) instruction set, featuring 512-bit vector units. It was also deployed in the Fugaku Supercomputer, which was ranked first in the Top500 list from June 2020 to November 2021 [113]. At that time, the ARM architecture was starting to gain traction in the HPC and server ecosystems. This trend has continued to strengthen, particularly with the adoption of Amazon's Graviton processors [101, 102] in its cloud servers. Therefore, porting applications to the ARM architecture was an essential step toward supporting and accelerating its adoption.

In this contribution, we port the BWA-MEM2 tool [15] to the ARMv8-A architecture specification and leverage the newly introduced ARM Scalable Vector Extension

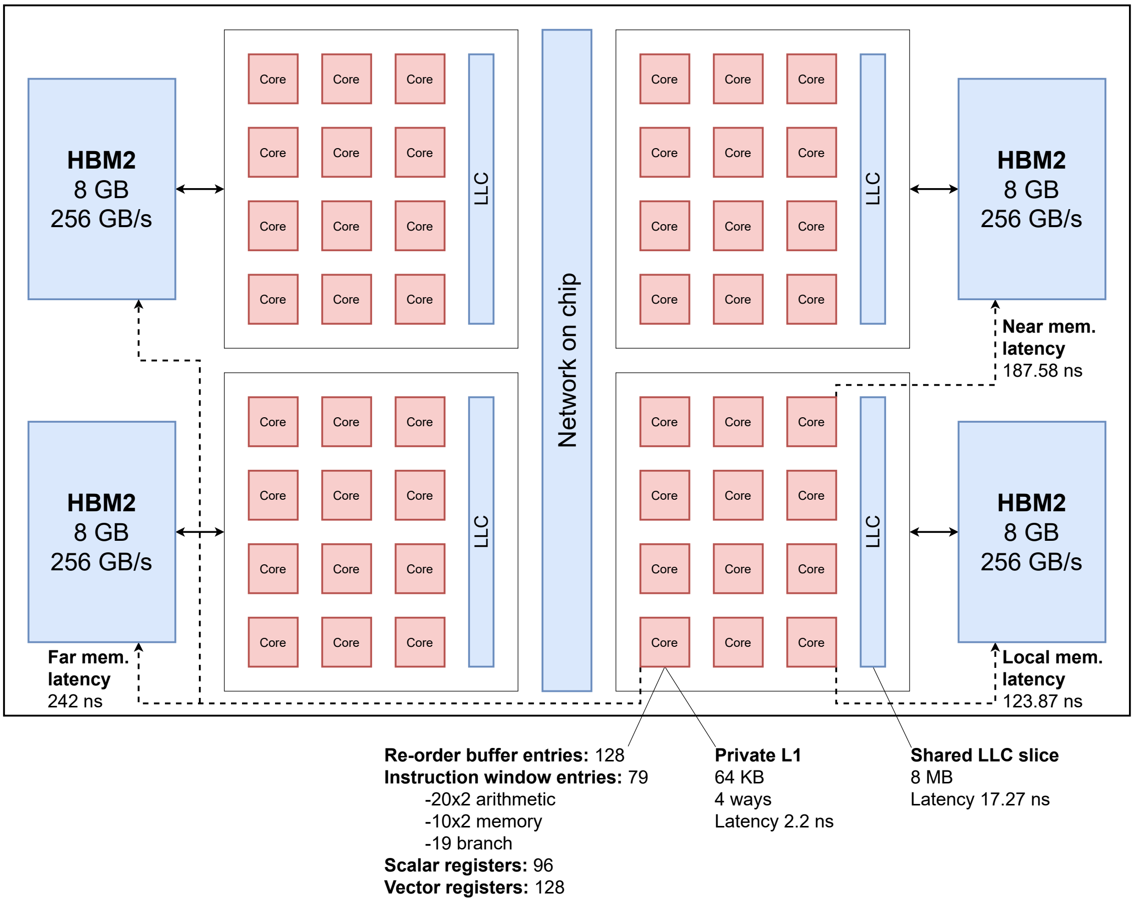


Figure 5.1: A64FX scheme. We can distinguish the four different core memory groups (NUMA domains), each one with 12 cores, and four HBM2 stacks. Everything is in the same package.

(SVE) [133]. We provide details of the porting process, which mainly involves translating x86_64 vectorization intrinsics into SVE-compatible code. SVE code is vector-length agnostic, meaning that it can be executed on any architecture implementing SVE, with vector lengths ranging from 128 to 2048 bits. Finally, we evaluate the ported version on Fujitsu's A64FX processor.

In addition, we propose several optimizations to improve the performance of BWA-MEM2 on the A64FX. Some of these optimizations are generic while others take advantage of the A64FX underlying architecture. Finally, we compare performance and energy-to-solution of optimized implementations running on the A64FX and an Intel x86_64 Skylake architecture (SKX) that features AVX-512 (see Section 3.2). We show that the A64FX performance is below that of the Skylake system, since sequence alignment applications are heavily constrained by memory latency and the Skylake architecture is better optimized in this regard; whereas the A64FX is optimized to provide high memory

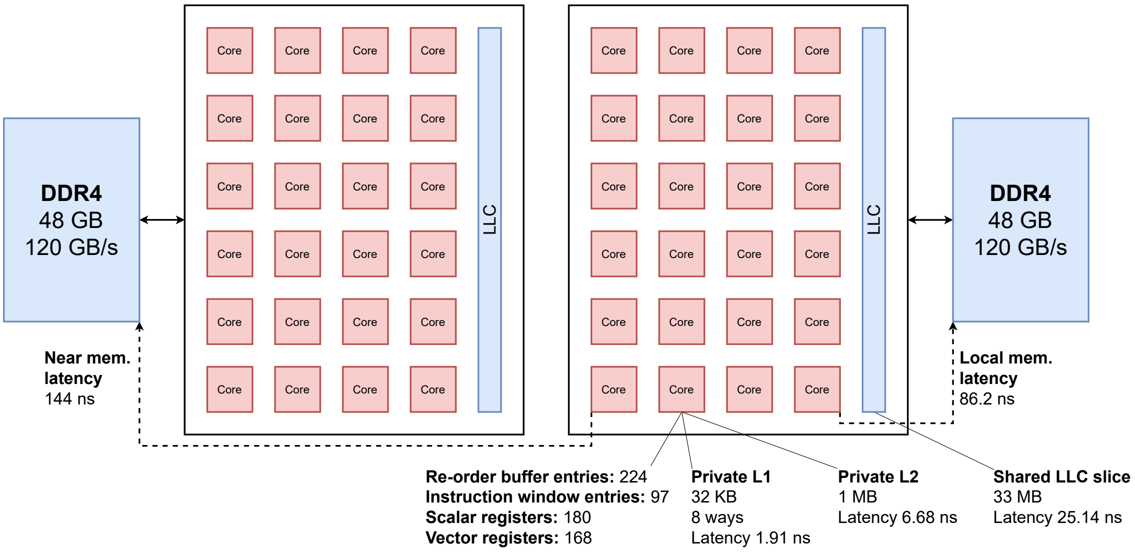


Figure 5.2: SKX scheme. The system is composed of two Intel Xeon Skylake Platinum 8160 and two DDR4 memories interconnected in the same board.

bandwidth. However, we also show that the A64FX presents better energy-to-solution results than the Skylake system.

The code of this contribution is available at https://gitlab.bsc.es/rlangari/bwa-a64fx.

5.1 Target Machine: Fujitsu's A64FX

In this work we make use of the Fujitsu A64FX. Some of the software optimizations we propose in Section 5.3 take advantage of the A64FX architecture, which differs significantly from conventional x86_64 architectures. Therefore, in this section we describe and characterize the A64FX architecture to better understand its trade-offs and optimize for them.

The A64FX was launched in 2019, however, accessibility to the machine was restricted until the second half of 2020. Figure 5.1 shows an overview of the chip, which features a total of 48 compute cores and four HBM2 stacks. The A64FX chip has been designed to perform well under HPC workloads with stringent memory bandwidth requirements. Its architecture is based on four Non-Uniform Memory Access (NUMA) domains within the same chip and was the first to implement SVE [133–135].

In this section, we present an overview of the A64FX and compare its main features against an Intel Xeon Skylake Platinum 8160 (SKX), also HPC oriented. Figures 5.1 and 5.2 show a scheme of both systems with all the specifications.

5.1.1 Core Out-of-Order Resources

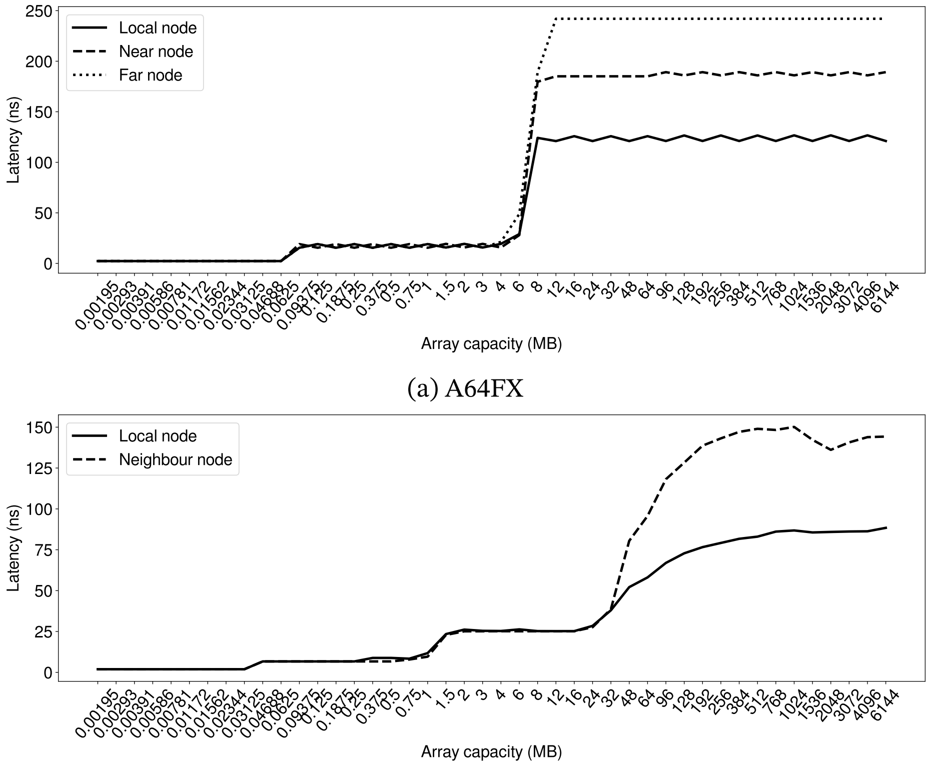
The A64FX has moderate out-of-order resources, as we see in Figures 5.1 and 5.2. The number of entries in the ROB is nearly the double for SKX. A64FX divides its 79-entry instruction window in 5 different reservation stations, one for each execution path: two paths for arithmetic operations (20 entries for each path), two paths for memory operations (10 entries for each path), and one path for branch operations (19 entries). In contrast, SKX has 97 entries in a unified instruction window. In addition, the number of physical registers is lower in the A64FX: 96 vs. 180 physical scalar registers, 128 vs. 168 physical vector registers, for A64FX and SKX, respectively. Both architectures implement 512-bit vector functional units.

5.1.2 The A64FX Memory Hierarchy

The A64FX memory hierarchy differs significantly from the traditionally employed in x86_64 platforms like SKX. The A64FX chip is organized in four NUMA domains, also known as core memory groups (CMG), each with 12 cores for general compute. Each domain has its own local memory interface, but it can also use the interfaces of the *near* domain and the two *far* domains (Figure 5.1). The SKX system has two interconnected sockets, which translates into two NUMA domains with 24 cores each, also totaling 48 cores (Figures 5.2).

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

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



(a) A64FX

(b) SKX

Figure 5.3: Memory latencies measured by the *lmbench* benchmark for A64FX and SKX.

5.1.3 Memory Hierarchy Latencies

To obtain the different memory latencies, we have executed the *lmbench* benchmark [136]. *lmbench* instantiates an array with a fixed capacity and performs random accesses to it. If the array fits in a given cache level and not in lower ones, *lmbench* can obtain the latency for that level, since the random access ensures that the data is not in a lower level.

Figure 5.3 shows the obtained results, which present a stair shape. Each *step* in the plots corresponds to a different memory level, and the array capacity of each *slope* matches with the capacity described in Figures 5.1 and 5.2. For example, for the A64FX, we can see a *step* between 0.0625 (64 KB) and 8 (8 MB), the capacity of L1 and a slice of LLC respectively, which means that this latency corresponds to the LLC.

We observe that the LLC latency is 31% lower in A64FX with respect to SKX. However, SKX has 1 MB of L2 cache per core with a compelling latency. When accessing memory, the latency is almost 50% higher in the A64FX for the local domain. In addition, accessing a far NUMA domain in the A64FX is very costly, nearly twice the latency of the local memory. Memory latency has less importance in regular applications that can benefit

Table 5.1: *lmbench* bandwidth benchmark results in GB/s for A64FX and SKX.

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

from spatial or temporal locality, however, genomic applications can suffer a significant penalty due to their irregular access patterns.

5.1.4 Memory Bandwidth

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

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

We can see the results in Table 5.1, where we perform three executions for each configuration, and we report the average and the standard deviation. We can see that for the *local* configuration, the results are consistent with the peak bandwidth specified for each memory technology on both machines. The *near* and *far* configurations are influenced by the characteristics of the network-on-chip and the socket-to-socket connection in the SKX case; consistently obtaining lower performance. In the *All* configuration, we observe that the measured bandwidth is far from the peak theoretical bandwidth. Since the memory is allocated in an interleaved manner, the requests from different NUMA domains are hindering memory performance. Finally, with the *All-split* configuration, we obtain a bandwidth close to the peak theoretical bandwidth.

5.2 Porting the code to ARM architecture

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

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

For this purpose, we create a header file called `sse2sve.h` that implements all the code related to SVE direct translations. Additional details on how these functions work and an example can be found in Section 5.2.3. However, complex code regions such as boolean operations or vector-width dependent code are translated in a case-by-case basis, since SVE instructions have a different structure and can benefit from features such as predication. Section 5.2.4 includes further details.

5.2.1 Data Types

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

```
typedef svint64_t __m128i;
```

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

5.2.2 Porting Effort

In Table 5.2, we show all the SSE intrinsics we have translated in `sse2sve.h` and their corresponding SVE translation. All the functions are marked and forced to be *inline* to avoid function calls and achieve 1-to-1 assembly translations.

5.2.3 *add* Translation Example

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

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

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

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

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

Table 5.2: Translations in the `sse2sve.h` file.

SSE intrinsic	**SVE translation**	**Functionality**
`_mm_malloc`	`aligned_alloc`	Allocate memory
`_mm_free`	`free`	Free memory
`__rdtsc`	`cntvct_el0`	Count cycles
`_mm_prefetch`	`__builtin_prefetch`	Software prefetch
`_mm_setzero_si128`	`svdup_s64(0)`	Set the register to zero
`_mm_set1_epi{8,16}`	`svdup_s{8,16}`	Set the register to a given value
`_mm_blend_epi{8,16}`	`svsel`	Select from two registers with a predicate
`_mm_add_epi{8,16}`	`svadd_x`	Addition
`_mm_adds_epu{8,16}`	`svqadd`	Addition with saturation
`_mm_sub_epi{8,16}`	`svsub_x`	Subtraction
`_mm_subs_ep{i,u}{8,16}`	`svqsub`	Subtraction with saturation
`_mm_max_ep{i,u}{8,16}`	`svmax_x`	Maximum
`_mm_min_epu{i,u}{8,16}`	`svmin_x`	Minimum
`_mm_and_si128 (arithmetic)`	`svand_z`	Bit-wise AND
`_mm_and_si128 (predicate)`	`svand_x`	Logical AND
`_mm_or_si128 (arithmetic)`	`svorr_z`	Bit-wise OR
`_mm_or_si128 (predicate)`	`svorr_x`	Logical OR
`_mm_xor_si128 (arithmetic)`	`sveor_x`	Bit-wise XOR
`_mm_andnot_si128 (arithmetic)`	`svbic_x`	Bit-wise ANDNOT
`_mm_andnot_si128 (predicate)`	`svbic_z`	Logical ANDNOT
`_mm_cmpeq_epi{8,16}`	`svcmpeq`	Equal comparison
`_mm_cmpgt_epi{8,16}`	`svcmpgt`	Greater than comparison
`_mm_cmpge_epi16`	`svcmpge`	Greater or equal comparison
`_mm_load_si128`	`svld1`	Memory load
`_mm_store_si128`	`svst1`	Memory store

5.2.4 Boolean Translation Example

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

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

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

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

`cmp1` is a vector variable, previously generated, which acts as a boolean predicate to perform masking operations. `_mm_movemask_epi8` moves the least significant bit of each lane to a scalar variable `cval`. Then, `cval` is checked as an exit condition for a loop.

Therefore, this code requires the creation of the scalar cval mask and then a comparison operation. However, we can translate this code to SVE in a way that the comparison can be done directly using cmp1:

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

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

5.2.5 Challenges

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

Application Code

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

The main difficulty during the porting was to translate the BSW x86_64 intrinsics to SVE. Most of the code is translated using a header file created for this purpose, which contains one-to-one intrinsics translations to SVE. However, in some cases, we had to rewrite the original code to maintain the semantics, as masks behave differently in SVE (see Section 5.2.4).

System Software

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

A64FX was the first processor to implement SVE. Due to the novelty of SVE at the time this work was carried out, not many compilers supported SVE intrinsics. For example, the native Fujitsu compiler (FCC) has many optimizations that target the A64FX but still lacks SVE intrinsics support and we were not able to use it for our experiments. However, FCC has a clang back-end mode that does support intrinsics but at the cost of certain A64FX-specific optimizations. Both GCC11 and Arm HPC compiler also support intrinsics. Therefore, for this work we have employed the FCC compiler with clang

Table 5.3: Execution time relative standard deviation for 10 runs using D3, D4 and D5 inputs (see Section 3.4) and the three code regions, with and without the thread affinity.

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

backend, as it showed better performance than the rest of the compilers. FCC obtained 3.46%, 4.10%, and 6.07% better performance with respect to GCC11 (the second best compiler) for D3, D4, and D5 inputs (see Section 3.4), respectively.

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

Execution Time Variance

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

We solved this problem by using the *pthread* affinity functionality, pinning each thread to a core. The C code snippet is as follows:

```c
for (int i = 0; i < n_threads; i++) {
    cpu_set_t cpus;
    CPU_ZERO(&cpus);
    CPU_SET(i, &cpus); // Bind this thread to CPU i
    pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t),
                                           &cpus);
    pthread_create(&tid[i], &attr, function, args);
}
```

Table 5.3 shows the relative execution time standard deviation for 10 different runs, with and without the thread affinity functionality.

5.3 Optimizations

This section describes several optimizations we have tried over the ported code. First we describe optimizations that apply to the entire application, and then those that target a particular code region. These optimizations try to take advantage of the A64FX architecture, but are likely to perform well on most systems. Some of the optimizations we tried did not yield the expected result, but are explained regardless to provide the lessons learned. Each of the optimizations that had a positive effect is later evaluated in Section 5.4.

5.3.1 General Optimizations

Split Input

The A64FX has 4 core memory groups, each of them with 12 cores and 8 GB of shared HBM memory. Each core group sees one group as a near domain, and the other two groups as far domains. Accessing the near domain has a significantly lower latency as shown in Section 5.1.3 compared to the far domain. In order to avoid accesses to far domains, we have tried to split the input, and execute two different processes, each one on two near domains. By doing this, we would have two processes using 24 cores each (2 CMGs), but each process would require copy of the index and all necessary data structures. After trying multiple combinations and compression schemes, we concluded that this is not a feasible option on the A64FX due to its total memory capacity of 32 GB.

Large Pages

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

5.3.2 SMEM Region

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

Aggressive Inlining

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

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

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

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

Population Count (popcnt)

Counting the number of bits set in a register (the `popcnt` operation) is a critical operation for the FM-Index algorithm as it is on the critical path between two dependent irregular memory accesses. BWA-MEM2 relies on the compiler built-in function `__builtin_popcount`, that on x86_64 translates into a single instruction as the hardware supports this operation. However, on the A64FX the built-in performs the operation using bitwise operations with masks and sums to obtain the result [137]. As opposed to modern x86_64 architectures, the ARMv8.2-A specification does not include an instruction to perform a *popcnt* over a scalar register. However, SVE does have a specific vectorized *popcnt* instruction. Therefore, we rewrote the code using SVE intrinsics by moving the scalar register into a vectorial one, performing the *popcnt* using the `svcnt` intrinsic, and moving back the result to a scalar register again.

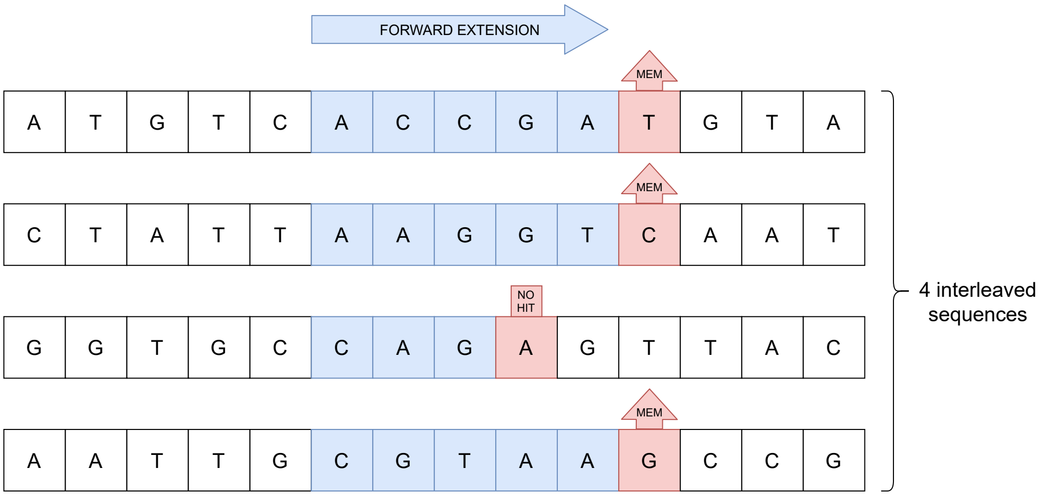

FORWARD EXTENSION
MEM
A T G T C A C C G A T G T A
MEM
C T A T T A A G G T C A A T
NO HIT
G G T G C C A G A G T T A C
MEM
A A T T G C G T A A G C C G
4 interleaved sequences

Figure 5.4: Interleaving four SMEM processes for different input reads.

Interleaved Sequences

The A64FX has a high memory access latency compared with HPC systems. However, it does have a significant advantage in terms of memory bandwidth. As we explained in Section 3.1.1, the forward extension used to obtain SMEMs leads to chains of dependent long-latency memory accesses. Therefore, the latencies of each individual access cannot be hidden. To solve this issue and allow hiding these latencies, we propose to interleave multiple accesses to the FM-Index from different input reads, effectively performing several forward extensions in parallel, as shown in the Figure 5.4. This exposes more memory level parallelism as more accesses are in-flight, hiding their latencies; and the core pipeline has additional work to do with multiple forward extensions running. Prior work showed this is an effective technique when memory bandwidth is abundant (see Section 4.4.2).

Manual Loop Fission

The A64FX has less out-of-order resources than other HPC processors. Limiting the number of instructions within a loop is very important to save resources, such as reorder buffer storage or physical registers. To enable aggressive loop unrolling the FCC compiler has an automatic loop fission feature that can be configured via a specific `pragma`. By using this technique loops can be split, providing smaller loop bodies, which leads to a better utilization of the out-of-order resources [138].

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

5.3.3 BSW Region

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

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

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

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

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

First, we build the predicate in a similar fashion and store it in *cmp11*. Then, the *add* operation can be directly predicated, indicating that non-active lanes in the mask need to be zeroed. This is accomplished with the appropriate *_z* suffix. By doing this we save instructions within performance critical tight loops. Note that we cannot do this by overloading the functions presented in Section 5.2.2 because we are combining two instructions into one. We have applied this technique when possible. Since the algorithm is compute bound and we are reducing the compute pressure within tight loops, we are able to gain performance by leveraging SVE's predication.

Table 5.4: Details for D3, D4 and D5 input datasets.

	D3 [139]	D4 [140]	D5 [141]
Organism	Homo Sapiens	Homo Sapiens	Homo Sapiens
Machine	Illumina Genome Analyzer II	Illumina HiSeq 2000	Illumina HiSeq 2000
Sequence length	76	101	101
Total number of sequences	17.8 M	92.4 M	1,436.8 M
Selected seq. for evaluation	1.25 M	1.25 M	1.25 M
Run	SRR043348	SRR622461	SRR622457

5.4 Experimental Results

In this section, we show the evaluation of the BWA-MEM2 port on Fujitsu's A64FX processor. We present the inputs used for the evaluation. Then, we evaluate the optimizations shown in Section 5.3. We explore how the SVE vector length affects the BWA-MEM2 execution time. Finally, we compare the performance of BWA-MEM2 running on the A64FX and an Intel Skylake (SKX) (see Section 3.2).

5.4.1 Input Sequences

We use three real inputs from the National Center for Biotechnology Information Sequence Read Archive sequenced with an Illumina machine, termed: D3 [139], D4 [140] and D5 [141]. We randomly select 1.25 million sequences from the original files. D3 has sequences of 76 bps long, while D4 and D5 have sequences of 101 bps. These datasets were also used to evaluate BWA-MEM2 when it was released [15]. Table 5.4 shows the details of these datasets.

5.4.2 A64FX Optimizations Evaluation

We use as the initial baseline the BWA-MEM2 code described in Section 5.2. Then, we evaluate the different optimizations described in Section 5.3. We progressively add one optimization at a time to determine its impact on the final performance. When adding an optimization, all previously evaluated optimizations are also present. All the experiments presented in this section employ 48 threads, one per core.

Large Pages

Figure 5.5 shows the speedups when using large pages for each input and region of interest. The average speedup among all the sections and all the inputs is 4.8%. However, the SMEM section is the most affected by this optimization since it is a latency bound

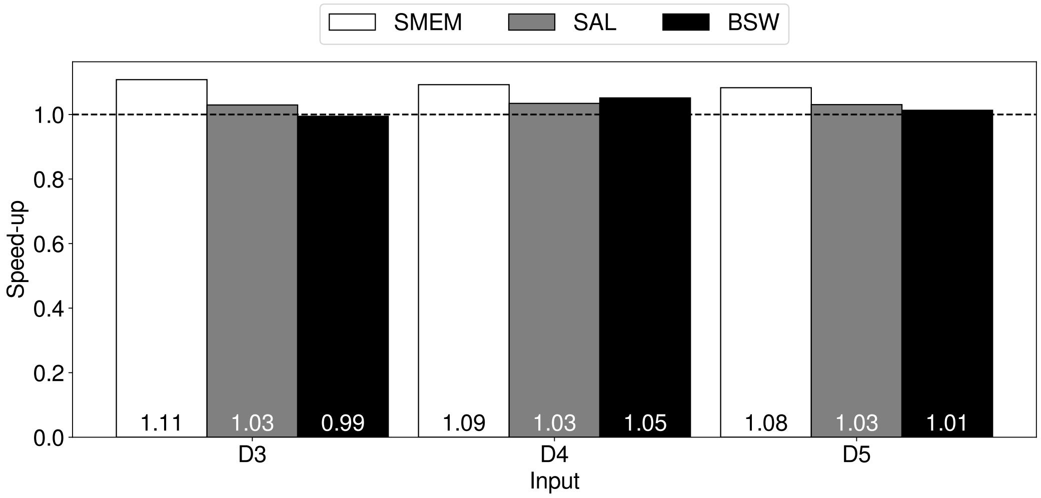


Figure 5.5: Speed-ups when using large pages with respect to not using them on A64FX with 48 threads. Showing data for D3, D4, and D5 inputs and the three code sections.

kernel, and it has a large memory footprint, showing a speedup of 1.11× with the D3 input.

Function Inlining

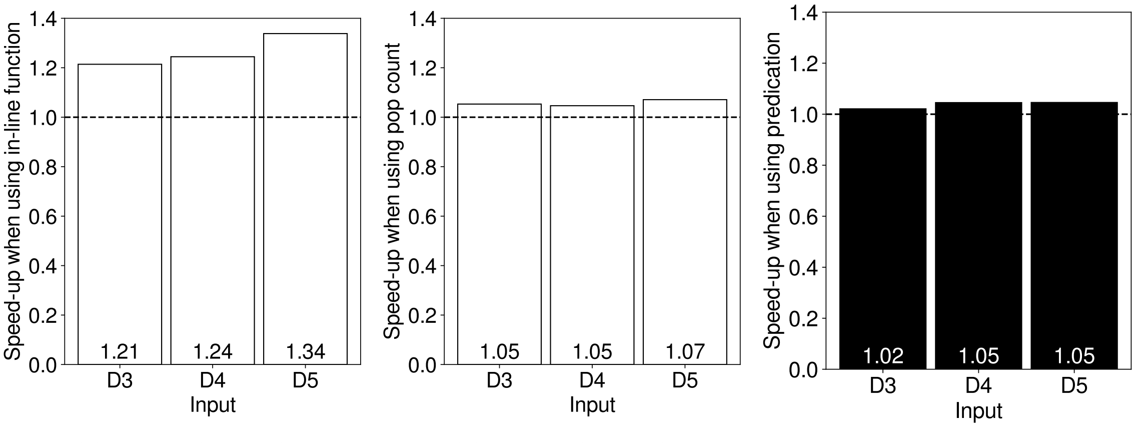
Figure 5.6a shows the obtained speedups when using the *inline* directive in the `backwardExt` function. This optimization only affects the SMEM section, and achieves a 1.27× average speedup in this application phase. Therefore, the function call with parameters passed by value has a significant overhead that can be avoided.

Population Count

Figure 5.6b shows the speedups when adding the `population count (svcnt)` SVE instruction in the SMEM region. By replacing a sequence of arithmetic instructions for a single instruction primitive we achieve an average 5.7% performance improvement.

Interleaved Sequences

Figure 5.7 shows the speedups obtained when interleaving multiple accesses to the FM-Index from different input reads, i.e., performing several forward extensions in parallel. Again, this optimization only affects the SMEM section. We observe that the best configuration is when using four interleaved sequences, which gives an average 8.1% performance improvement with respect to not using interleaved sequences. Therefore, we will use four interleaved sequences from now on.



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

Figure 5.6: Speed-ups for (a) in-line, (b) population count and (c) predication optimizations for D3, D4, and D5 inputs on A64FX with 48 threads.

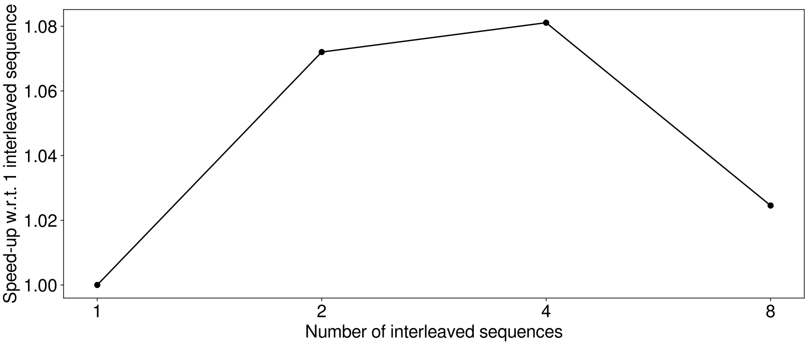


Figure 5.7: Average speedups for the interleaved sequences optimization on SMEM for D3, D4, and D5 inputs on A64FX with 48 threads.

BSW SVE Predication

In Figure 5.6c, we can see the speedups when using SVE predication in the BSW section. Despite we only predicated a few instructions inside the inner loop of the kernel, we achieve an average 3.7% performance improvement in BSW.

Final Version

In total we have performed five optimizations over the baseline implementation. We can see the performance improvements achieved by all the optimizations in Figure 5.8. The figure shows, for multiple thread counts and for each input, the normalized execution time when all optimizations are applied (*Optimized*) with respect to the ported code (*Baseline*). We obtain an execution time improvement of 23.2% on average for 48

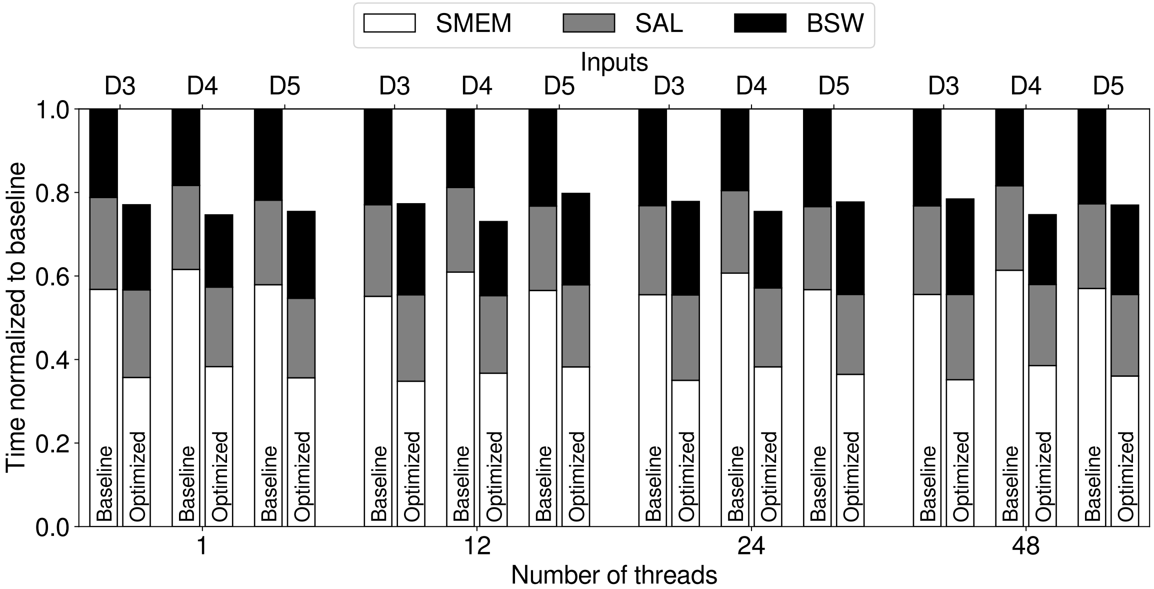


Figure 5.8: A64FX normalized execution time with respect to the unoptimized baseline. Showing executions with 1, 12, 24 and 48 threads using D3, D4, and D5 inputs.

threads. In the figure, we can also distinguish each code region. SMEM is the region that experiences the largest performance gain, 36.9% on average for 48 threads. We also observe that the optimizations have a similar effect regardless of thread count. Therefore, they are not targeting any bottlenecks that appear due to core count scaling. As we show in Section 5.4.5 the application scales almost linearly with thread count.

5.4.3 SVE Vector Length Analysis

In order to test the vector length scalability of our SVE port, we perform experiments with additional vector lengths of 128 and 256 bits. While the A64FX supports up to 512 bit vectors, the hardware can be instructed to use shorter vectors if desired. We execute the same binary for all the vector length, since SVE is a *vector length agnostic* ISA. As expected, we observe that the SMEM and SAL regions are not affected by vector length, as their code is not vectorized. Therefore, we only show the speedups for the BSW region. Figure 5.9 shows the performance improvements when changing the vector length for the three inputs on 48-thread runs, normalized to single-thread SVE 128 bits. We also observe average speedups of 37.6% when going from 128 to 256, and 79.0% when going from 128 to 512 SVE bits. These results correlate with the reduction in terms of committed instructions shown in Table 5.5. This indicates that the performance improvements we

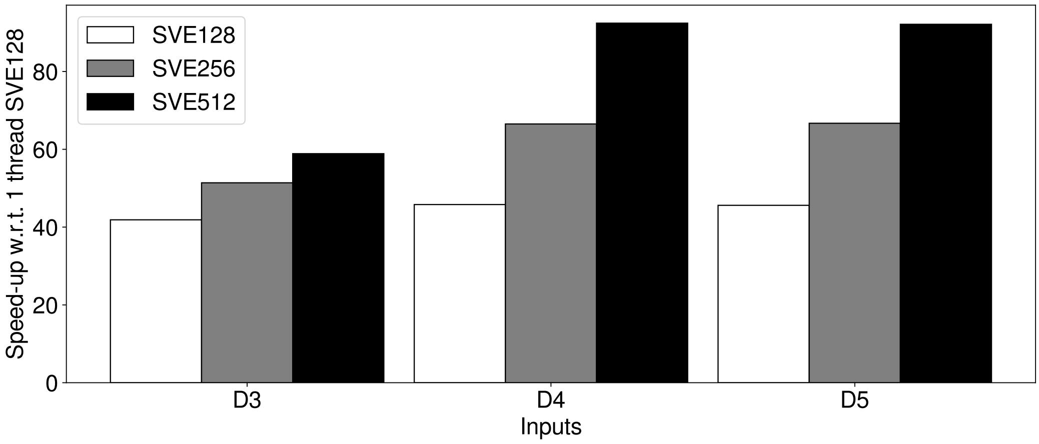


Figure 5.9: BSW region speedup for 48-thread executions with respect to single-threaded and SVE 128 bits for D3, D4, and D5 inputs.

Table 5.5: Billions of committed instructions for a single threaded execution of BSW, and instruction reductions with respect to SVE 128.

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

obtain in terms of SVE scaling are the expected ones, and that memory bandwidth is not limiting the performance of this kernel.

5.4.4 Scalability Analysis

Figure 5.10 shows average performance scaling with 12, 24 and 48 threads for the different regions as well as the entire execution. All the regions attain good scalability for all thread counts, reaching 44.5×, 44.3× and 43.1× for SMEM, SAL and BSW, respectively, with 48 threads. These numbers are close to the theoretical peak (48×), proving that the bandwidth is not limiting performance.

5.4.5 Comparison with SKX

In this section we compare performance and energy-to-solution of the A64FX against the SKX system, both described in Section 5.1 and Table 3.1.

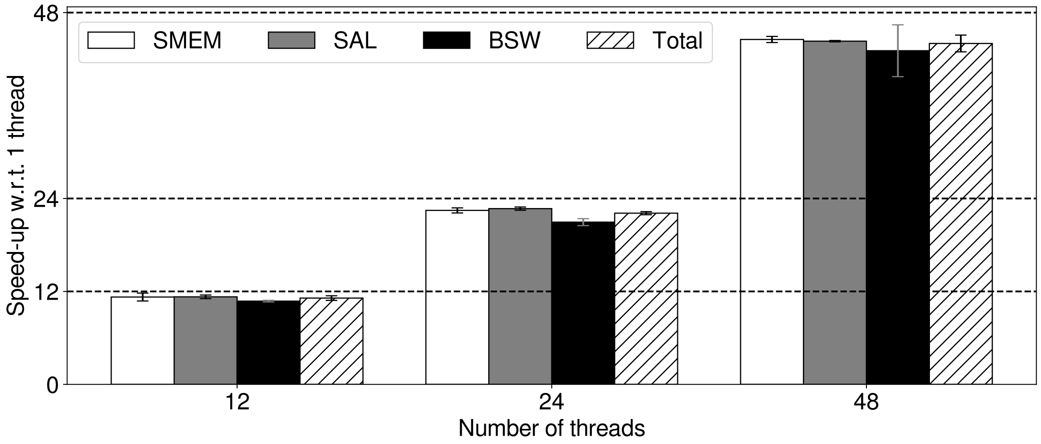


Figure 5.10: Per region and total *average* speedups with respect to one thread for different thread counts using inputs D3, D4, and D5.

Performance Comparison

Figure 5.11 shows performance results for the A64FX and the SKX system for the three regions as well as the entire execution using 24 and 48 threads.

For the SMEM region, SKX clearly outperforms the A64FX on a performance per thread (core) basis, 2.01× with 48 threads. This is due to two factors: (i) the aggressive out-of-order pipeline of the SKX is much more effective at hiding long latency misses, and (ii) the memory access latency of the SKX system is significantly lower (see Figure 5.3). If we compare on a socket-to-socket basis, i.e., 24 threads of SKX versus 48 threads of A64FX, the results are more on par with a 10.6% advantage for SKX on average.

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

The BSW region is much more compute intensive. On a per thread performance basis the SKX system still outperforms the A64FX by 1.47× on average for 48 threads. In this instance the main factors are: (i) the SKX core has more out-of-order resources (see Figures 5.1 and 5.2), and (ii) the memory hierarchy of the SKX system has significantly more cache capacity, with 1 MB of private L2 per core (no private L2 in A64FX) and 33 MB of LLC per socket (32 MB of LLC on A64FX). However, when comparing on a socket-to-socket basis the A64FX outperforms the SKX system by 24.2%.

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

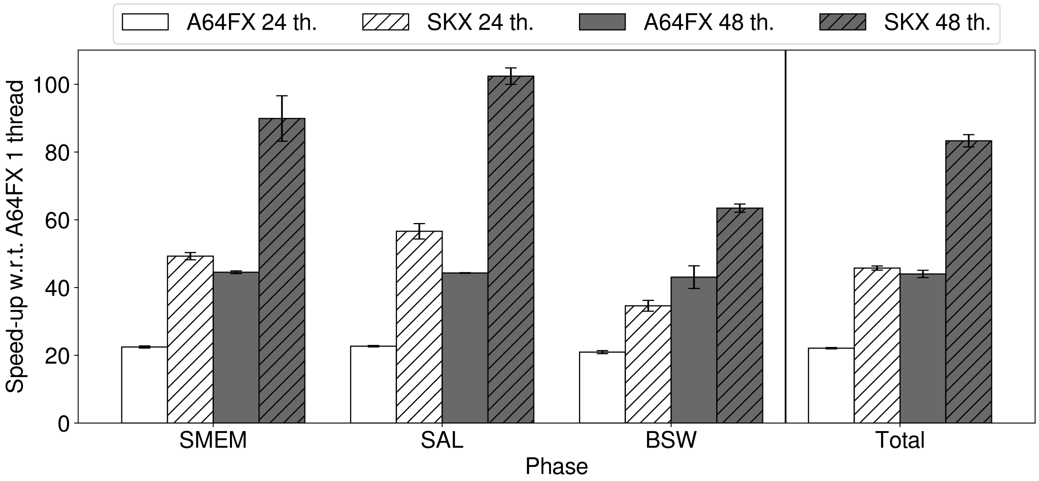


Figure 5.11: Average speedups with respect to one thread in A64FX for each region and for the entire execution using 24 and 48 threads using D3, D4, and D5 inputs.

Energy-to-Solution Comparison

Figure 5.12 shows energy-to-solution for different thread counts on both systems for the entire region of interest; which includes SMEM, SAL and BSW. We can observe that SKX is more energy efficient, 71.1% and 38.9% on average than the A64FX for 1 and 12 threads, respectively. However, for 24 threaded executions the A64FX starts to gain efficiency with respect to the SKX system, and the advantage is reduced significantly to 15.0%. Note that we have to consider that for 24 threads the SKX system has the advantage of only using a single socket, as the other one is in idle state; while the A64FX chip is using half its cores. With 48 threads the SKX system fails to scale well in terms of energy-to-solution (13.2% of improvement over 24 threads) since it has to use the second socket, which increases the power consumption significantly. Therefore, performance gains are offset by the increase in power consumption. However, the A64FX system continues to scale well in terms of energy-to-solution and beats the SKX system by 11.6% on average. If we compare in a socket-to-socket basis the A64FX also has better energy-to-solution by 26.4% on average.

Bottleneck Profiling

In the work presented later in this document (GenArchBench in Section 6), we profile several genomic kernels. Among them, we analyze SMEM and BSW kernels (named FMI and BSW in GenArchBench, respectively).

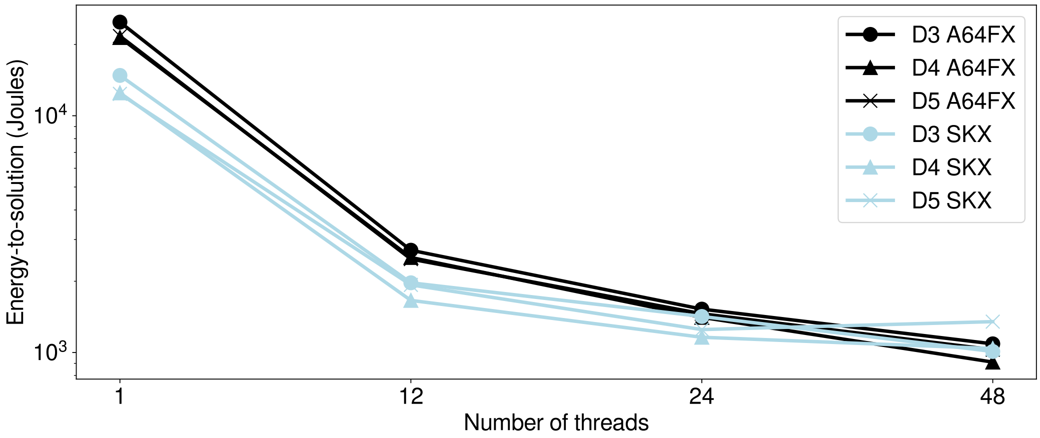


Figure 5.12: Energy-to-solution comparison of the A64FX and SKX systems for the entire region of interest using D3, D4 and D5 inputs.

Figure 6.4 shows the instruction mix of the kernels, while Figure 6.5 shows the microarchitecture bottlenecks of the kernels. For SMEM, we observe a high number of memory operations, which translates into a memory bottleneck, especially for the A64FX due to its high memory latency. On the other hand, BSW is a compute bound kernel dominated by integer and register manipulation operations. In SKX, the execution time is dominated by useful work, while in A64FX, memory stalls are still an important factor.

5.4.6 Discussion

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

Regarding input size sensitivity, larger read sequences could affect data locality, however, BWA-MEM2 is a read mapper specifically designed for short sequences and we did not consider using long reads, as other mappers should be used for those. On the other hand, using a reference genome that fits within the LLC would benefit the

A64FX, as latency would be drastically reduced. A larger reference could make the A64FX unsuitable for the workload; if the reference, the index, and the read sequences exceed the 32 GB of HBM2 memory of the A64FX.

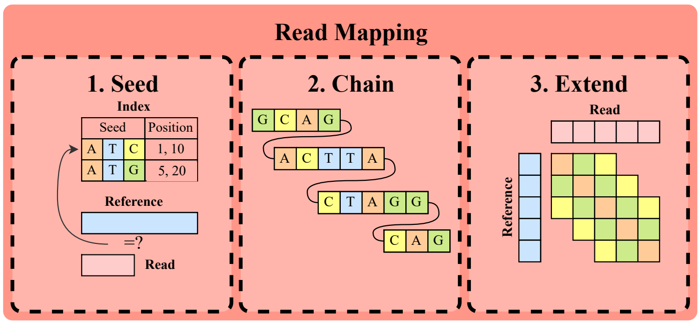
Lessons Learned

From the application side, we have found that working with the BSW code is tedious and error prone. For this kernel, there is one implementation for each vector ISA (i.e., SSE2, AVX2 and AVX-512BW) as it is based on intrinsics. Since each vector ISA has different characteristics (e.g., support for masks) the implementations of the algorithm is also different. While the use of intrinsics may lead to performance gains, the drawbacks in terms of code maintenance, readability, and extensibility to new architectures might offset the benefits. SVE's vector length agnosticism is a step in the right direction, as any machine regardless of the implemented vector length can execute the code.

From the A64FX architecture side, the main take away is that it requires programmers and users to be aware of its memory capacity and NUMA characteristics, which expose certain trade-offs. First, the 32 GB of main memory can be limiting in some scenarios. Second, efficiently use the available bandwidth requires to perform accesses to the local NUMA node, as going to other nodes leads to worse bandwidth utilization (see Table 5.1) and higher latency (see Figure 5.3). Therefore, careful placement of processes, threads and data allocation to minimize accesses to different NUMA nodes is a paramount (see subsection *Execution Time Variance* in Sections 5.2.5).

Chapter 6

GenArchBench: A Genomic Benchmark Suite for ARM HPC Processors



*In **GenArchBench**, we port to ARM and optimize **one kernel for each read-mapping stage***

The ARM architecture has been steadily emerging in the HPC and server ecosystems. The A64FX processor [98] was the first ARM-based processor to power a supercomputer that reached the top position in the Top500 list [113]: the Fugaku Supercomputer, which held this rank from June 2020 to November 2021. More recently, Amazon has adopted the Graviton processor family [101, 102] for its cloud servers, while NVIDIA's Grace processor [142] powers Alps, the Swiss supercomputer that entered the top 10 of the Top500 list in 2024. With the emergence of such a robust ecosystem, the need to develop and enhance a comprehensive software stack for ARM-based HPC systems has become evident. In this context, GenArchBench addresses a key gap by providing a dedicated genomic benchmark suite designed for ARM HPC processors.

In this chapter, we present GenArchBench, a benchmark suite comprising 13 computationally demanding CPU kernels extracted from some of the most widely used genomic tools. All kernels exploit multi-core parallelism and represent common stages in genome analysis pipelines, such as base-calling, read mapping, variant calling, and de novo assembly. Furthermore, this work introduces architecture-specific adaptations and optimizations of these genomic kernels for ARM HPC CPUs. Notably, several kernels have been optimized using the latest ARM Scalable Vector Extensions (SVE) to fully leverage the computational capabilities of modern 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. 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.

This work has been performed in collaboration with a research team. Among them, we find Asaf Badouh, Víctor Soria-Pardos, Quim Aguado-Puig, Guillem López-Paradís, and Max Doblas. Special mention should be made to Lorién López-Villellas, that was coordinating all the work and putting all the pieces together. I have contributed to this work by porting and optimizing BSW, FAST-CHAIN, and FMI kernels.

6.1 GenArchBench Benchmark Suite

The GenArchBench benchmark suite comprises 13 multithreaded CPU kernels derived from the most widely used genomic tools and covers the most important genome sequencing steps. It includes ten kernels from the GenomicsBench [143] benchmark suite and three additional kernels: the Bit-Parallel Myers algorithm [144] (BPM), the Wavefront Alignment algorithm [56] (WFA), and FAST-CHAIN [145]. BPM and WFA complement the sequence alignment kernels of GenomicsBench to better capture contemporary trends. Additionally, FAST-CHAIN 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 provides curated input datasets for each kernel, including small and large datasets, along with their corresponding outputs to be used as ground truth. The small datasets, executed for testing purposes, have been sized to require single-thread execution times no longer than a few minutes; meanwhile, large datasets, run for performance evaluation purposes, require several minutes. 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 genomic kernels in ARM processors. Moreover, the ARM-systems evaluated in this work (A64FX and Graviton3) are not equipped with GPUs.

As we use GenomicsBench as our starting point, GenArchBench adds several improvements to it. First, we port 10 kernels from GenomicsBench from x86 to the ARM architecture and optimize them. In addition, we have added three new kernels (BPM, WFA, and FAST-CHAIN), for which we have performed vectorization from scratch. We also adapt the BSW SIMD version from x86 to the ARM architecture. Finally, we include a comprehensive set of inputs and outputs for testing purposes.

The kernels presented in this section target x86 architectures and have not been extensively tested or 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., the ground truth).

The following subsections present GenArchBench's kernels, briefly describing their functionality, the tools that execute them, and a description of their usage and inputs.

6.1.1 Adaptive Banded Signal to Event Alignment (ABEA)

ABEA is a dynamic programming algorithm that compares raw nanopore signals from ONT sequencing systems to a reference genome sequence. ABEA's implementation is based on the Suzuki-Kasahara algorithm [97]. This step is performed in some tools, such as Nanopolish [146], to correct errors produced in the basecalling process (Figure 2.1-2). For GenArchBench, we have used the CPU implementation of f5c [147], a version of ABEA based on Nanopish'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 [148].

6.1.2 Bit-Parallel Myers (BPM)

BPM [144] 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 (Figure 2.1-3.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 and libraries, such as GEM-Mapper [27], Edlib [22], GraphAligner [149] or Hobbes [150]. 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 [151].

6.1.3 Banded Smith-Waterman (BSW)

The Smith-Waterman algorithm [31] 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 [96] is used to align sequences with a maximum of w insertions/deletions, reducing the time and space complexity to $O(wn)$ (Figure 2.1-3.a.1.3). BSW is used in variant discovery tools such as GATK [33], and in sequence alignment software like BWA-MEM [15, 23]. 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 among core. The small and large inputs comprise 100K and 10M sequence pairs from human sample SRR7733443 [151].

6.1.4 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 (Figure 2.1-3.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 [39] or Canu [40]. We have selected 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 [152].

6.1.5 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 [145] 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.

6.1.6 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 (Figure 2.1-3.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 [39], Canu [40] or Racon [41], and in variant callers such as GATK [33] and Platypus [34]. 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 [153]. The small input uses bases 16M-16.5M, while the large input uses the entire chromosome.

6.1.7 FM-Index Search (FMI)

The FM-Index is a compressed sub-string index based on the Burrows-Wheler transform [154]. 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 (Figure 2.1-3.a.1.1). The FM-Index data structure is used in sequence alignment tools such as BWA-MEM [15, 23] or Bowtie2 [26], and in metagenomic classification software like Centrifuge [42]. 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 [151].

6.1.8 K-mer Counting (KMER-CNT)

K-mer counting aims to count the number of occurrences of each k-mer in an input sequence (Figure 2.1-3.b.1). This task is performed in de-novo assemblers such as Flye [39] or Canu [40] and in metagenomic classification software like Clark [47]. 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 [24]. For GenArchBench, we have used the k-mer counting kernel of Flye. This implementation divides the input-reads among threads and relies on the thread-safe hash-map implementation of Libcuckoo library [155] 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 [156].

6.1.9 Neural Network-based Base Calling (NN-BASE)

ONT sequencing systems 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 (Figure 2.1-2). The analog signal inevitably contains ambiguities due to noise or measurement errors. Some basecallers, such as Guppy [19] and Bonito [18], 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 [157]. Bonito splits the input signal into smaller chunks of regular size and feeds them to a PyTorch neural network that 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 [148].

6.1.10 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 (Figure 2.1-3.a.2). This is a costly process performed by statistics-based variant callers, such as GATK HaplotypeCaller [33] or Platypus [34], and deep-learning variant callers, such as Clair [35, 36], DeepVariant [37] or Medaka [38]. For GenArchBench, we have used the second generation of Clair variant caller (Clair3), based on the TensorFlow framework [158]. 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 [159]. We are using Clair3's ONT pre-trained model `r941_prom_hac_g360+g422` [160].

6.1.11 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 [161], 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 [35, 36] or Medataka [38] (Figure 2.1-3.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 [19], and the large input comprises bases 1-1412827 of chromosome 20 of sample HG002 [159].

6.1.12 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) (Figure 2.1-3.b.3). The Partial Ordered Alignment (POA) algorithm [162] 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 [31] or Needleman-Wunsch [30]. The multiple alignment sequence (consensus sequence) is inferred from the graph by using the Heaviest Bundle algorithm [163]. POA is used in software packages such as Nanopolish [146] or Racon [41]. For GenarchBench we have used the SIMD-optimized version of POA of the SPOA library [164]. 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 [19].

6.1.13 Wavefront Alignment (WFA)

The wavefront alignment algorithm (WFA) [165] is a pairwise alignment algorithm (Figure 2.1-3.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 [166], AnchorWave [167] or AncestralClust [168]. GenArchBench includes a custom multi-thread implementation of WFA, where each thread aligns a pair of strings independently. The small and large inputs comprise 100K and 1M sequence pairs from human sample SRR7733443 [151].

6.2 ARM Porting of Genomic Kernels

Among all the kernels present in GenarchBench, I have contributed to BSW, FAST-CHAIN, and FMI. In this section, the porting and optimization process of these three kernels are described.

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. All the results presented in this chapter 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.

6.2.1 BSW

The SVE version of BSW [169] 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

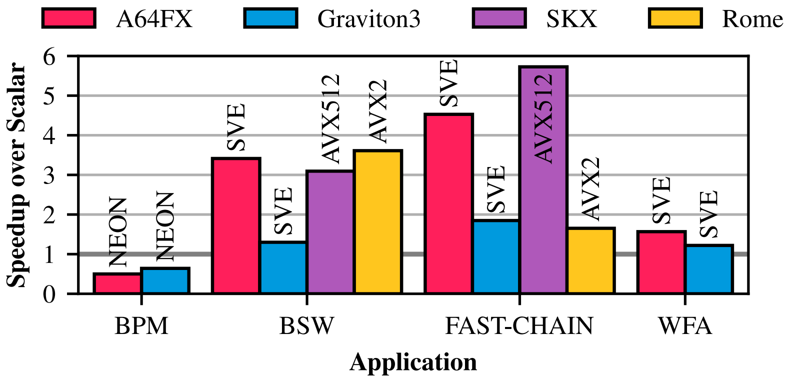


Figure 6.1: Speedup of SIMD kernels over their scalar version on the experimental setup using the large inputs.

avoid the need for blend operations, effectively reducing the number of total instructions executed. BSW operates on 16-bit integers, allowing to process 32 elements per iteration on SVE-512 (A64FX) and 16 on SVE-256 (Graviton3).

As shown in Figure 6.1, the SVE version of BSW performs 3.4× and 1.3× faster than its scalar version on the A64FX and Graviton3, respectively.

6.2.2 FAST-CHAIN

Our SVE implementation of FAST-CHAIN is a translation to SVE intrinsics of the x86 version. We have used the x86 version [145] as a reference, but our FAST-CHAIN code is entirely original. 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-bit anchors. In some cases, 32-bit anchors are not sufficient, and this kernel generates incorrect results (also for the scalar version). To solve this, we have implemented 64-bit and 32-bit SVE versions of FAST-CHAIN. The 64-bit version always outputs correct results, but we have used the 32-bit implementation to compare against the 32 bits x86 implementation.

As shown in Figure 6.1, 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 —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.

6.2.3 FMI

GenArchBench's FMI version implements three optimizations proposed in Chapter 5:

- One of the most called functions in this kernel is `backwardExt`. To reduce the overhead of the calls, this function is 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%.

6.3 Performance Characterization

This section presents a detailed performance characterization of the kernels in our experimental setup. I personally contributed to the benchmark suite by porting three kernels: BSW, FAST-CHAIN, and FMI. However, we show the results for all the kernels in the suite.

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 executions. In the case of DBG in Graviton3, we selectively present results from a representative execution.

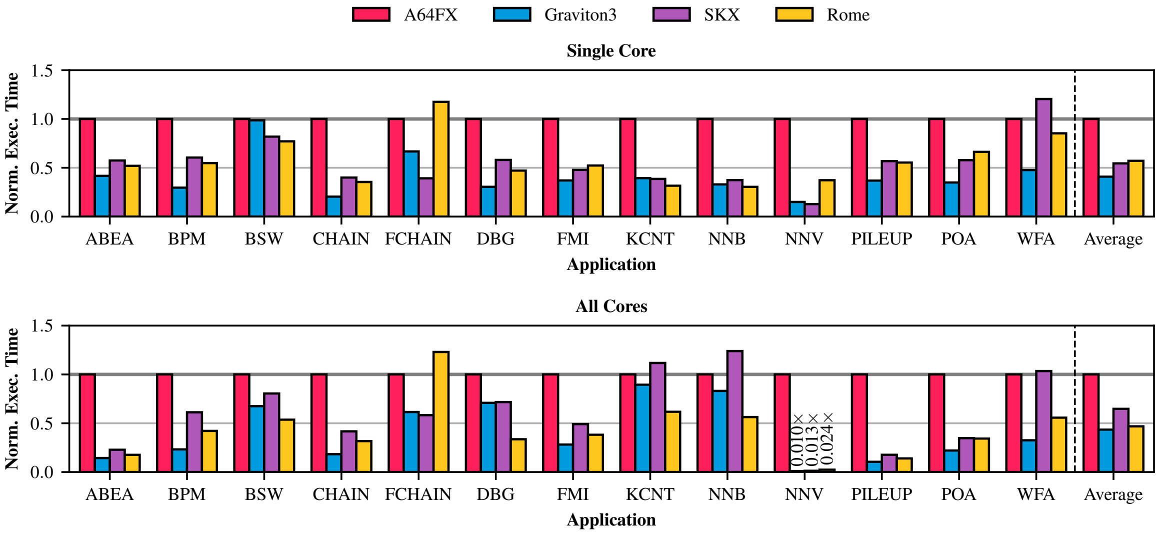


Figure 6.2: Single-core (top) and multi-core (bottom) execution time of GenArchBench's kernels on the experimental setup. Multi-core results correspond to executions using all available cores on each machine: 48 threads on the A64FX and SKX and 64 threads on Graviton3 and Rome. The results are normalized 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.

6.3.1 Single-Thread Performance

The top plot of Figure 6.2 shows the single-thread execution time of each kernel on the experimental setup. The results are normalized to the performance on the A64FX.

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 AVX-512 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. We observe better performance for A64FX in compute-bound kernels such as BSW, with respect to the other systems. The available higher memory bandwidth benefits the A64FX. On the other hand, latency-bound kernels such as FMI hinder A64FX performance since the memory latency is much higher in the A64FX processor (Section 5.1.3).

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 AVX-512 (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.

6.3.2 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. We bind threads to cores to prevent software threads from changing the core on which they execute. When using the *pthreads* library, we use the `pthread_attr_setaffinity_np` C function as shown in Section 5.2.5. For *OpenMP* codes, the flag `OMP_PROC_BIND` needs to be used to bind the threads.

Controlling thread affinity is mandatory in our experiments to achieve good parallel performance on the machines, especially on the A64FX, where migrating a thread mid-execution to other NUMA domains could result in large differences in memory latency and bandwidth (Sections 5.1.3 and 5.1.4). ABEA, NN-BASE, and NN-VARIANT do not allow full thread affinity control. Therefore, thread migrations can occur in these three kernels.

Figure 6.3 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 Figure 6.2 compares the performance obtained using all available cores on each machine: 48 threads on the A64FX and SKX and 64 threads on Graviton3 and Rome. 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, which benefits scalability since the threads can work in their chunks without having to wait for other threads. It can be seen that some kernels achieve near-perfect scaling on all machines. This is the

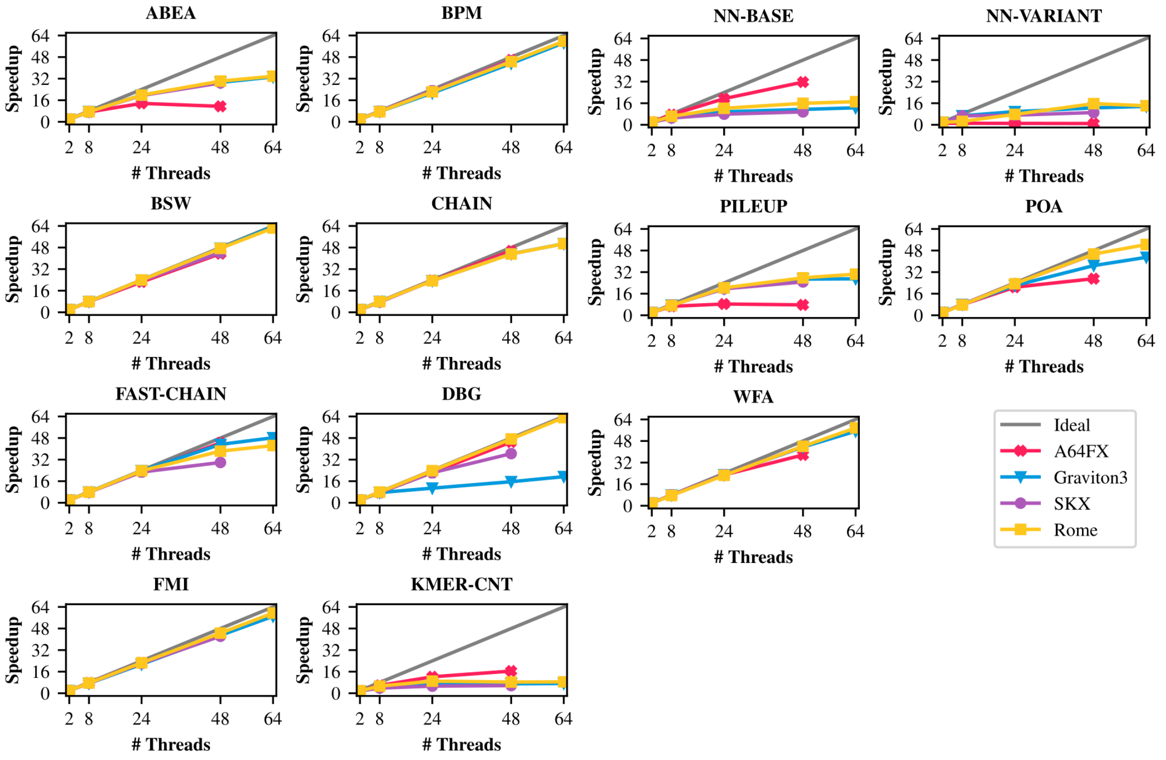


Figure 6.3: Speedup over serial execution of GenArchBench's kernels on the experimental setup. We show the achieved speedup using different thread counts: 2, 8, 24, 48, and 64. The A64FX and SKX 64-threads points are not shown in the figure, since those machines only implement 48 cores. We also show ideal (linear) parallel speedup (when using x threads, ideally we expect a speedup of x) in gray.

case for BPM, BSW, CHAIN, FMI, and WFA. For this set of kernels, the normalized plots using one 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. The parallel performance of both kernels should 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. In contrast, 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 behaviour, 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[170], which allows using intra-op parallelism (via math libraries like Intel MKL[171]) 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 and 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.

6.3.3 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 [172, 173] on x86 (SKX) that divides the executed instructions into different categories [174]. 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.

Figure 6.4 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 manip-

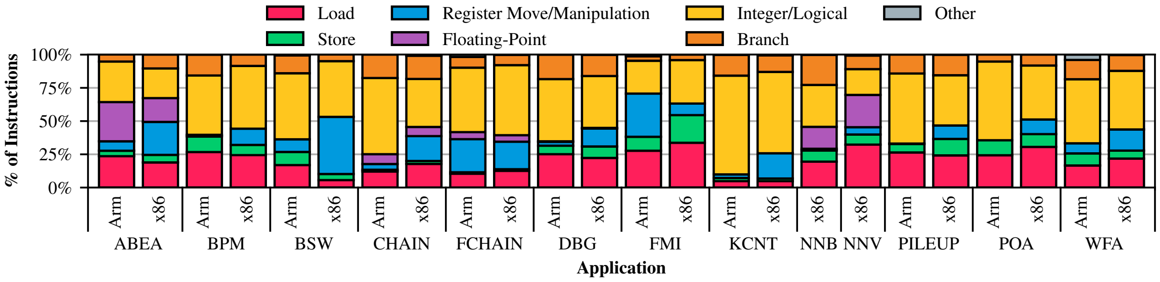


Figure 6.4: Instruction mix of GenArchBench's kernels on ARM (A64FX) and x86 (SKX). FCHAIN, KCNT, NNB and NNV are the abbreviations of FAST-CHAIN, KMER-CNT, NN-BASE and NN-VARIANT, respectively.

ulation 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 6.3.4). The rest of the kernels mostly execute integer and logical instructions and between 30% and 40% of data movement operations.

6.3.4 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 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 system metrics to our microarchitecture bottleneck categories:

- Back-End Stalls: Produced when a back-end stage of the core is stalled. We split this category into:
  - Memory Stalls: Includes stalls due to main memory and caches.
  - Core Stalls: Produced when an arithmetic unit blocks the pipeline.
- Front-End Stalls: Produced when the front-end stage from the core is stalled. This can be the instruction cache or the decode stage.
- Bad Speculation Stalls: Produced by a misprediction from a speculation unit. This includes the branch predictor and the load-store queue.
- Useful Work: Time spent committing instructions.
- Other: Other stalls not taken into account by the other counters.

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* subcategory.

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.

Figure 6.5 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

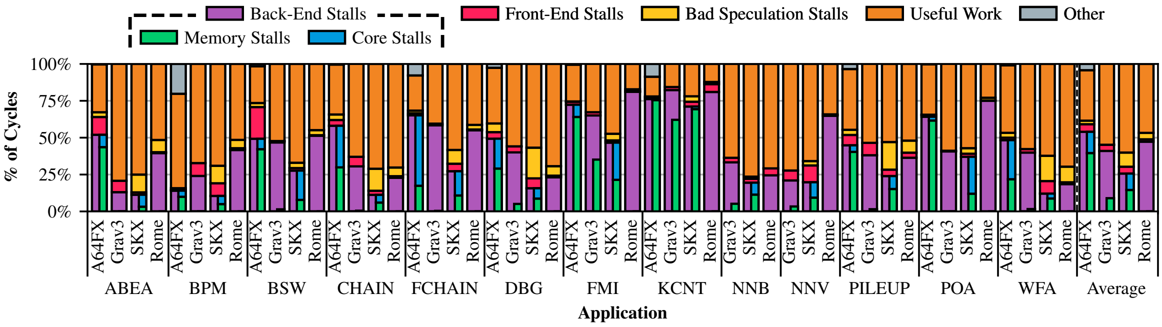


Figure 6.5: Microarchitecture bottlenecks of GenArchBench's kernels on the experimental setup. FCHAIN, KCNT, NNB and NNV are the abbreviations of FAST-CHAIN, KMER-CNT, NN-BASE and NN-VARIANT, respectively. Grav3 is the abbreviation of Graviton3.

bottleneck on ARM is much smaller. Although this may 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, which explains 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.

6.3.5 Energy Consumption

Figure 6.6 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 an estimated 12% extra energy consumption to Rome's measurements based on the energy consumption of SKX's DRAM (expressed as error bars in Figure 6.6). Energy consumption was measured using the Fujitsu power API [175] on the A64FX, and an in-house library [176] based on the Running Average Power Limit (RAPL) [177] on SKX and Rome. Load imbalance makes NN-VARIANT a kernel with poor scalability. Moreover, the A64FX scalability in NN-VARIANT is even lower than in the other systems. Therefore, the results of NN-VARIANT were not taken into consideration for the average calculation, as we consider them to be outliers.

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

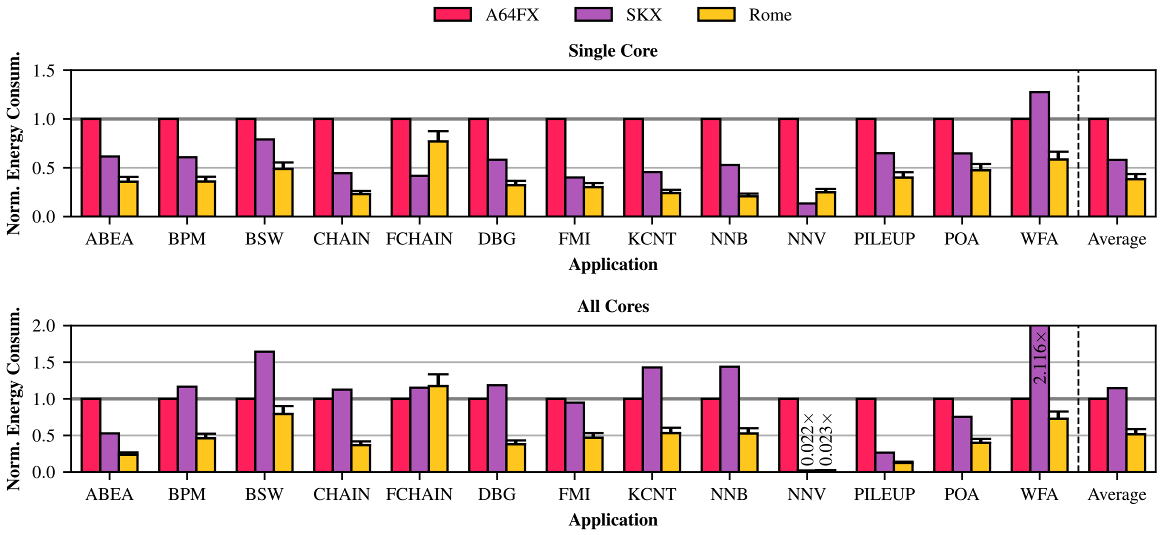


Figure 6.6: Single-core (top) and multi-core (bottom) energy-to-solution of GenArchBench's kernels on the experimental setup. Multi-core results correspond to executions using all available cores on each machine: 48 threads on the A64FX and SKX and 64 threads on Rome. The results are normalized to the 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.

(CPU throttling), while the A64FX constantly consumes power near its peak, even on low usage.

The results shown in the top plot of Figure 6.6 are highly similar to those presented in the top plot of Figure 6.2. 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 Figure 6.6). 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). This change in trend occurs because the SKX system has to use both of its sockets. During single-thread execution, only one socket is active. On the other hand, when all cores are used, both sockets become active, which increases the power consumption by nearly a factor of two.

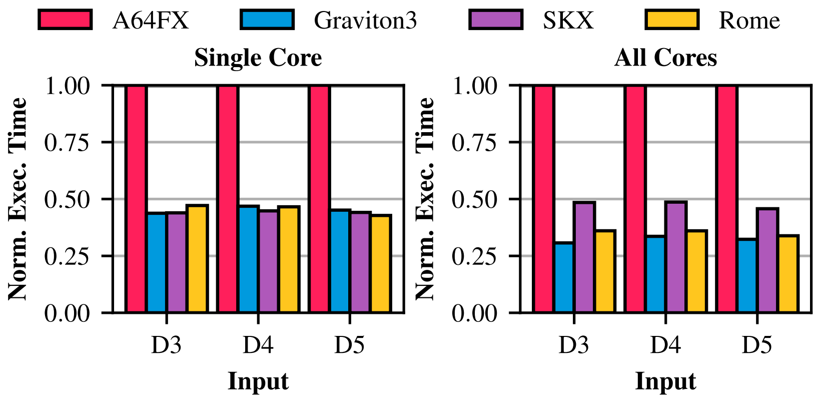


Figure 6.7: Execution time of BWA-MEM2 using one core (left) and all available cores in each machine of the experimental setup (right). We show results using three inputs: D3, D4, and D5. The results are normalized to the performance on the A64FX using one core (left) and 48 cores (right).

6.3.6 Evaluation of a Real Genomic Tool

Finally, we evaluate the performance of a real genomic tool that uses some of the ported kernels presented in this chapter. This evaluation is an extension of the one shown in Section 5.4.5. Specifically, we add two new machines to the analysis: Graviton3 and Rome.

As shown in Section 5.4.5, we use BWA-MEM2 [15], a read mapping tool (Figure 2.1-3.a.1) that employs the FMI kernel for the seed stage (Figure 2.1-3.a.1.1) and the BSW kernel for the extend stage (Figure 2.1-3.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.

We use the inputs already described in Section 5.4.1: the input sequences D3 [139], D4 [140], and D5 [141], and the human genome GRCh38 [53].

Figure 6.7 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 Figure 6.2. In the GenArchBench results, 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

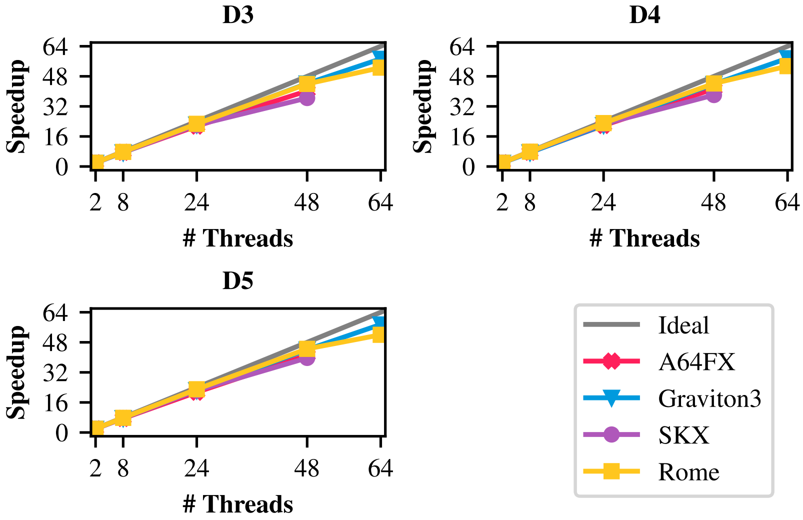


Figure 6.8: Speedup over serial execution of BWA-MEM2 on the experimental setup for three inputs: D3, D4 and D5. We show the achieved speedup using different thread counts: 2, 8, 24, 48, and 64. The figure does not show the A64FX and SKX 64-thread points since those machines only implement 48 cores. We also show ideal (linear) parallel speedup (when using x threads, ideally we expect a speedup of x) in gray.

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 Figure 6.3), 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 Figure 6.8.

6.3.7 Summary

Table 6.1 shows the raw numbers for all the experiments conducted in the evaluation. We can see that A64FX and SKX achieve greater speedups when using SIMD, as they feature wider vector registers (512 bits) than Graviton 3 and Rome (256 bits). For single-thread performance, the FMI kernel is memory-latency-bound, which degrades the performance of the A64FX. On the other hand, BSW is compute-bound, and the higher memory bandwidth of A64FX helps to reduce the performance gap relative to the other systems. FAST-CHAIN is also compute-bound but exhibits more irregular code patterns with a higher number of branches, leading to greater performance variability across systems. When using all cores in the system, we observe a notable increase in speedup for Rome and Graviton 3, as they each have 64 cores compared to 48 in A64FX and SKX.

Communication among threads is minimal for these three kernels, resulting in parallel efficiencies above 88% for BSW and FMI across all systems. However, FAST-CHAIN suffers from load imbalance, which reduces parallel efficiency.

With respect to the most frequently used instruction types, we observe the expected behavior. BSW and FAST-CHAIN show higher usage of integer and register-manipulation instructions, as they are compute-bound kernels. In contrast, FMI exhibits a higher proportion of memory instructions. These observations correlate with the bottleneck analysis, which shows a predominance of back-end stalls across all systems. A64FX is significantly limited by back-end stalls: its memory system is well suited for the BSW and FMI kernels, while the branch-intensive nature of FAST-CHAIN causes core back-end stalls. Graviton 3 slightly mitigates these bottlenecks, although back-end stalls remain dominant. SKX presents the most balanced behavior, with the highest percentage of time spent doing useful work. Rome also suffers from back-end stalls, particularly in FMI (81%), which represents the highest back-end stall percentage in this analysis.

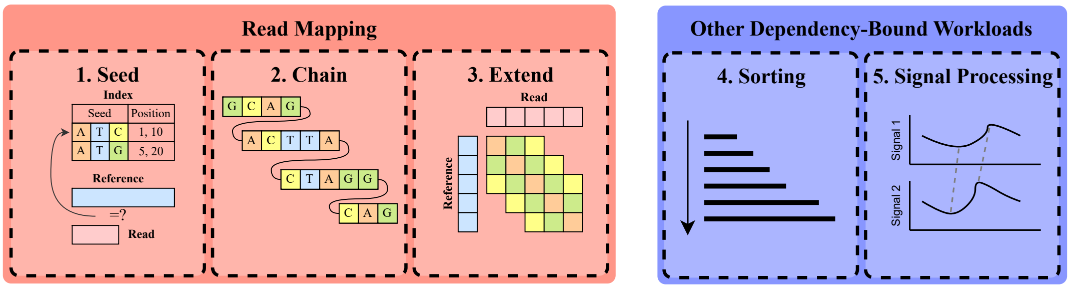
Regarding energy consumption, the SKX system shows higher consumption than A64FX for BSW and FAST-CHAIN, since SKX consists of two chips on the same board, which significantly increases power usage. For FMI, the reduction in execution time on SKX compensates for the higher power draw, making SKX more energy efficient. Rome achieves lower energy consumption than A64FX for BSW and FMI, as the difference in power does not outweigh the performance speedup. However, the performance degradation observed in FAST-CHAIN makes A64FX more energy-efficient for this kernel.

Table 6.1: Summary of the evaluation. *SIMD speedup w.r.t. scalar* shows the speedup when using the SIMD unit with respect to using the scalar unit (Figure 6.1). The SIMD units are as follows: A64FX has SVE512 (512 bits), Graviton 3 has SVE256 (256 bits), SKX has AVX-512 (512 bits), and Rome has AVX2 (256 bits). *Single thread speedup w.r.t. A64FX* shows the speedup with respect to the A64FX system when using one thread (top of Figure 6.2). *Parallel speedup w.r.t. A64FX* shows the speedup with respect to the A64FX system when using all the threads in the system (bottom of Figure 6.2). A64FX and SKX have 48 threads, while Graviton 3 and Rome have 64 threads. *Parallel efficiency* shows the ratio *S/I*, where *S* is the speedup when using all threads in the system with respect to using only one, and *I* is the ideal (linear) parallel speedup (Figure 6.3). *Most used instruction type* shows the most used instruction type and its percentage (Figure 6.4). *Bottleneck analysis* shows the percentage of each stall type (Figure 6.5). *BE* is back-end stall, *FE* is front-end stall, *BS* is bad speculation stall, and *UW* is useful work. *Norm. energy consum.* shows the energy consumed by each system normalized to the energy consumption of the A64FX (Figure 6.6).

		SIMD speedup w.r.t. scalar	**Single thread speedup w.r.t. A64FX**	**Parallel speedup w.r.t. A64FX**	**Parallel efficiency**	**Most used instruction type**	**Bottleneck analysis**				**Norm. energy consum. w.r.t. A64FX**
							BE	**FE**	**BS**	**UW**	
A64FX	**BSW**	3.4×	1×	1×	0.91	Integer (50%)	49%	22%	3%	25%	1
	FAST-CHAIN	4.5×	1×	1×	0.93	Integer (49%)	65%	1%	2%	24%	1
	FMI	-	1×	1×	0.9	Load/Store (38%)	72%	2%	0%	25%	1
Graviton 3	**BSW**	1.3×	1.01×	1.48×	0.99	-	47%	1%	0%	52%	-
	FAST-CHAIN	1.8×	1.50×	1.63×	0.75	-	58%	1%	0%	41%	-
	FMI	-	2.70×	3.55×	0.89	-	65%	2%	0%	33%	-
SKX	**BSW**	3.1×	1.22×	1.24×	0.92	Reg. manipulation (43%)	34%	2%	4%	60%	1.64
	FAST-CHAIN	5.7×	2.55×	1.72×	0.62	Integer (53%)	31%	5%	11%	53%	1.15
	FMI	-	2.09×	2.04×	0.88	Load/Store (55%)	48%	3%	6%	43%	0.95
Rome	**BSW**	3.6×	1.30×	1.86×	0.98	-	51%	0%	4%	45%	0.88
	FAST-CHAIN	1.7×	0.85×	0.81×	0.66	-	55%	1%	3%	41%	1.31
	FMI	-	1.91×	2.62×	0.93	-	81%	0%	2%	17%	0.53

Chapter 7

Squire: A General-Purpose Accelerator to Exploit Fine-Grain Parallelism on Dependency-Bound Kernels



Squire *is a general-purpose hardware accelerator for* ***dependency-bound kernels***

Multithreaded systems use coarse-grained parallelism to distribute tasks among processing units. Within each coarse-grained task, smaller fine-grained parallel tasks can be identified. To exploit this type of parallelism, Single Instruction, Multiple Data (SIMD) techniques are typically used. However, SIMD performs poorly when applied to dependency-bound kernels, where the inner loop contains data dependencies across iterations. GPUs can achieve good performance, but only at the cost of considerable area, power, energy, and financial resources. ASICs also deliver high performance, but their functionality is limited to specific tasks, resulting in a lack of flexibility. These limitations highlight the need to develop a general-purpose accelerator that efficiently targets fine-grained parallelism in dependency-bound kernels.

In this chapter, we present Squire, the fourth and last contribution of the thesis. First, we examine some representative kernels and show how fine-grained parallelism is currently approached. Then, we present Squire, a general-purpose accelerator designed to effectively exploit fine-grain parallelism on dependency-bound kernels. Our proposal incorporates one Squire per core in a typical multi-core system, connecting it to the memory hierarchy to directly access the virtual memory space. Each core controls one Squire, rapidly offloading workloads whenever it needs. We show how we adapt five dependency-bound kernels: Radix Sort [70], Seeding [24], Chain [24], Smith-Waterman [31, 65], and Dynamic Time Warping [178]. Finally, we evaluate Squire on a simulated multicore SoC, obtaining speedups of up to 7.64× in dynamic programming kernels, and an acceleration for an end-to-end application of 3.66×. We also evaluate the usage of resources and show that Squire achieves an energy reduction of up to 56% with an area overhead of 10.5% per core. Finally, we compare Squire against other proposals and accelerators.

7.1 Dependency-Bound Fine-Grain Parallelism

Coarse-grain parallelism is desirable in high-performance computing environments to hide the overheads associated with task management and data movement. This inter-task parallelism means that each processing core operates on an independent task [15, 24, 25, 27, 179–182]. However, this approach often struggles to exploit the fine-grain parallelism inherent in many kernels with complex dependencies.

Intra-task fine-grained parallelism is usually tackled via *SIMD* on general-purpose processors or *SIMT* on GPUs. However, these techniques are inefficient when targeting specific algorithms containing dependencies or sparse patterns. Well-known algorithms that suffer such problems include: Quicksort [183], Dynamic Time Warping [178], and Smith-Waterman [31, 65]. Additionally, data structures with sparse memory patterns, such as FM-Index [28], hash tables [70], and sparse matrix-vector multiplication (SpMV) [184], are often limited by the amount of memory-level parallelism that can be exposed. All these patterns are present in many applications.

Figure 7.1 highlights three kernels that exhibit fine-grain parallelism (shown using different colors) which is challenging to exploit due to existing dependencies. Figures 7.1a and 7.1c show how sorting and SpMV coarse-grain tasks could be further parallelized by processing chunks of the array or independent rows of the matrix in parallel. However, this is not efficient due to data-dependent irregular patterns and the fact that SIMD

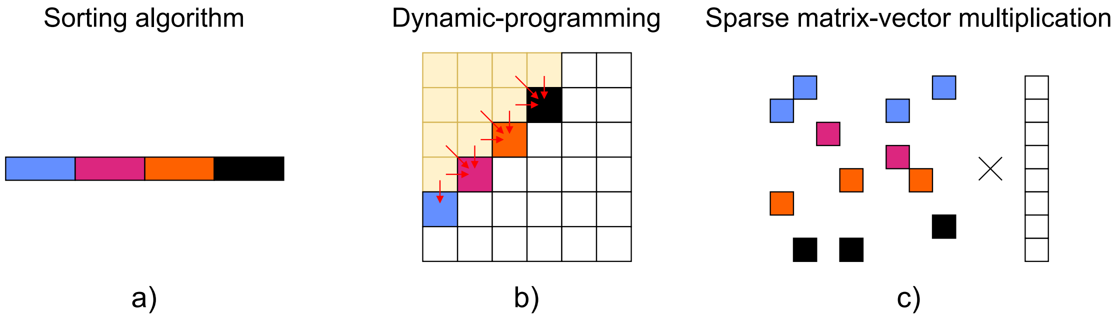


Figure 7.1: Examples of coarse-grain tasks with fine-grain parallelism: (a) sorting, (b) dynamic programming matrix, (c) sparse matrix-vector multiplication. Each color in the data structures represents a chunk of fine-grain work.

gather/scatter memory operations are not efficient [185]. Subfigure 7.1b shows how parallelism is present per cell in a dynamic programming matrix. For some dynamic programming problems, antidiagonal vectorization is the best way to avoid dependencies; however, it requires data rearranging that diminishes potential gains. Support for exploiting this fine-grain parallelism can benefit a large set of workloads.

GPGPUs have been proposed to tackle dynamic programming kernels [186, 187]. However, GPGPUs are designed for massive parallel workloads, and dependencies and sparsity hinder performance. In addition, offloading fine-grain parallel workloads is not recommended due to the high transfer time, and fine-grain synchronization is also challenging. For these reasons, GPGPUs obtain modest speedups when targeting dynamic programming algorithms [79, 109–112]. Finally, custom hardware solves dependency constraints at the cost of fixing the functionality of the proposed components, losing generality [69, 83, 84, 188, 189].

On the other hand, we can find specialized hardware accelerators designed to solve dynamic programming kernels, such as systolic arrays [69, 79, 83, 84] and processing-in-memory (PIM) architectures [88, 90]. However, these solutions rely on hard-wired components with fixed functionality and usually target a specific kernel. Systolic arrays define a fixed data type that cannot be modified; moreover, floating-point arithmetic is often avoided due to its high area cost. PIM systems typically do not support virtual memory, which limits the total memory available to the application. Memory capacity is a critical factor in genomics, where extremely large sequences and genomes are commonly processed.

Chain is a dynamic programming algorithm widely used in the field of genomics [24]. We showed that the SIMD version of chain obtains slowdowns of up to 0.71× with respect to the scalar version with heuristics (see Section 6.2.2). Chain presents dependency-bound patterns in the inner loop, resulting in underutilized vector lanes. Similarly, Guo et al. show that in the GPU version of chain, 16.3% of the kernel instructions are control instructions for synchronizing warps [79]. They use an NVIDIA Tesla P100 for the evaluation [80]. The P100 GPU achieves a 3.17× speedup with respect to a 14-core CPU while consuming 300W and occupying 610 mm^2, resulting in under-utilization of resources.

Because of these constraints, it's clear that we need a more adaptable and efficient way to exploit fine-grained parallelism in dependency-bound workloads. To address this, we plan to integrate our component into a general-purpose CPU that can seamlessly handle a wide range of workloads and data types. We propose a general-purpose accelerator - Squire - to unlock the parallelism potential of a wide range of workloads while maintaining flexibility and low overhead.

7.2 Squire

One of the main goals of Squire is to make it general-purpose, enabling workloads with inherent fine-grain parallelism to benefit from the accelerator. To achieve this, we identify two key design principles: (i) enable low-latency execution of fine-grain tasks and (ii) provide architectural support for fast synchronization between processing units to manage dependencies. Hence, our hardware accelerator must have the following features:

- A set of general-purpose processing units sharing a unified memory view with the host core.
- A synchronization mechanism to enable rapid communication among processing units.

7.2.1 Squire Design

Figure 7.2 shows the architectural overview of Squire, a general-purpose accelerator for dependency-bound fine-grain parallelism. Figure 7.2a illustrates a conventional multi-core SoC with a distributed L3 cache, where each core complex contains two levels of private caches. Figure 7.2b depicts the integration of Squire into the system, where each core complex is augmented with a Squire block interfacing with the private L2 cache. Finally, Figure 7.2c shows that Squire consists of a set of very simple general-

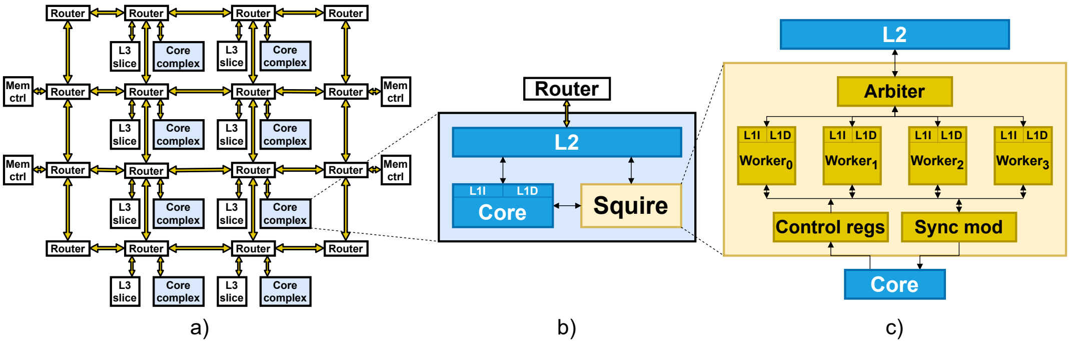


Figure 7.2: Squire architectural overview: (a) the simulated multi-core system with a 4x4 NoC and four memory controllers, where each central router holds a core complex and one slice of the L3 cache; (b) a core complex contains one OoO core with private L1 caches, a private L2 cache, and a Squire; (c) Squire contains a set of workers, several control registers, a synchronization module, and an arbiter to communicate with the memory hierarchy (L2).

purpose in-order cores, termed *workers*. In addition, Squire features control registers and a synchronization module, which are visible to both the host core and the workers.

As discussed in Section 7.1, dependency-bound kernels are often addressed using specialized hard-wired components such as systolic arrays, or through parallel architectures, such as GPGPUs or SIMD unit, which are not inherently optimized for this form of parallelism. In order to increase the flexibility of Squire, we propose employing simple in-order cores with small area and power consumption requirements for each worker. To simplify the design, we assume these cores share the same base ISA as the host core. In addition, each worker has small, private data and instruction caches. We define the size of these caches with a design space study in Section 7.5.4.

A host core can offload computation to the workers via a simple API (see Section 7.2.3) that sets a function's address and the necessary arguments into the control registers. Then, the workers start executing the workload using regular instructions. If the host core has recently accessed the input data, it is likely to still reside in the L2 cache, reducing data transfer latency.

To orchestrate L2 access requests from the worker cores, we employ a shared bus coupled with a centralized arbiter. The arbiter selects one request per cycle from the set of pending L2 accesses issued by the workers. This design enforces a single L2 access per cycle, thereby requiring only a single extra read/write port on the L2 cache, reducing implementation complexity. Cache coherence is maintained through a snoop-based protocol, where all workers monitor the L2 bus for invalidation messages. This is practical given the simplicity of the in-order cores. Moreover, workers are designed to target workloads that maximize L1 data reuse. Empirically, even with 32 workers active, the system sustains an average of no more than one L2 access every two cycles, demonstrating that accessing the L2 cache is not a primary bottleneck.

Tasks are distributed using traditional coarse-grain parallelism, with OpenMP assigning independent workloads to host cores [190]. In read mapping tools, for example, each host core aligns a subset of sequences. These tasks are typically dependency-bound, limiting the effectiveness of SIMD and instruction-level parallelism. Squire addresses this by subdividing tasks into fine-grain sub-tasks, enabling nested parallelism even in dependency-bound kernels.

7.2.2 Synchronizing Workers

The synchronization mechanism is used to coordinate the workers, and it is visible to the host core as well as the workers. We have designed the mechanism to enable modeling dependencies for two distinct common use cases.

On the one hand, our aim is to tackle algorithms that perform computation on 1D data structures. To achieve this, we use a simple mechanism that features a hardware atomic counter, referred to as *global counter*. This will enable handling loops where iteration *i* conditionally consumes the data produced by iteration *i-1*. For this purpose, we require the workers to increment the *global counter* in order, i.e., if worker *x* increments the *global counter* before worker *x-1*, the increment is saved in a structure until worker *x-1* increments the *global counter*. To implement this with a non-blocking scheme, we instantiate one queue per worker and a token. The token indicates which worker is the next to increment the *global counter* and is initialized to zero. If worker *x* wants to increment the *global counter* and the token contains the value *x-1*, an increment request is enqueued in *x*'s queue. When worker *x-1* increments the *global counter*, the queues are searched for pending increments in order, and the token is updated accordingly.

On the other hand, we want to efficiently handle workloads with 2D data structures, such as dynamic programming matrices with vertical and horizontal dependencies. For this reason, we also instantiate an array of hardware atomic counters, with a length equal to the number of workers, referred to as *local counters*. When filling a dynamic programming matrix, a worker increments its *local counter* each time it computes a matrix row. Thus, worker *x* can check *local counter x-1* before starting the next row.

This set of hardware atomic counters is implemented as 64-bit registers that can be accessed in one cycle.

Table 7.1: Squire programming interface. For each API call, we provide a brief description and who can use the API call.

API call	**Description**	**Caller**
`start_squire(f,a)`	Squire executes `f` function with `a` arguments. *Counters* reset to 0.	Core
`stop_worker()`	Suspends the worker execution.	Workers
`id_worker()`	Returns the worker ID.	Workers
`num_workers()`	Returns the total number of workers.	Core/Workers
`inc_lcounter(w)`	Increments the *local counter* `w` by one.	Workers
`inc_gcounter()`	Increments the *global counter* by one.	Workers
`wait_lcounter(w,s)`	Waits until the *local counter* `w` is greater or equal to `s`.	Core/Workers
`wait_gcounter(s)`	Waits until the *global counter* is greater or equal to `s`.	Core/Workers

7.2.3 Squire API

Table 7.1 describes the Squire programming interface. Each functionality in the table defines a new ISA primitive to interact with the accelerator. The table also specifies whether the host core, the workers, or both can invoke the primitives. Section 7.3.1 shows how the Squire API works using radix sort as an example.

Notice that the Squire API is inspired by the *pthreads* API [191] and both programming models are similar.

7.3 Using Squire

This section shows how to use Squire for several kernels. We describe the implementation process for the Radix Sort, Chain, and DTW algorithms in Squire. Finally, we discuss some alternatives considered for certain implementation details.

7.3.1 Sorting: Radix Sort

The pseudocode for Squire's radix sort implementation is shown in Algorithm 2. The host core executes the `RADIX` function (Line 1), which calls `start_squire` with the function and input data addresses as arguments (Line 3). This call writes the addresses to Squire's control registers, sets the workers' program counters to the function's entry point, and resets internal counters. The workers then execute the `RADIX_Workers` function (Line 8). Each worker retrieves its ID and the total number of workers via the `id_worker` and `num_workers` APIs, using this information to evenly partition the input array (Lines 9–10). Each chunk is sorted using the standard radix sort algorithm (Line 11). Upon completing its chunk, a worker increments the global counter and halts (Lines 12–13). Meanwhile, the host core waits until the global counter matches the number of workers (Line 4).

Algorithm 2 Radix Sort Squire version

```
 1: function RADIX(X[N])
 2:     if N > 10000 then
 3:         start_squire(RADIX_WORKERS, X)
 4:         wait_gcounter(num_workers())
 5:         MERGE_SORTED_ARRAYS(X)
 6:     else
 7:         RADIX_KERNEL(X[0:N])
 8: function RADIX_WORKERS(X[N])
 9:     start = id_worker() × (N / num_workers())
10:     end = (id_worker() + 1) × (N / num_workers())
11:     RADIX_KERNEL(X[start:end])
12:     inc_gcounter()
13:     stop_worker()
```

At this point, the input has been divided into *n* sorted subarrays, which the host core merges using a min-heap (Line 5). Squire may not be beneficial when the workload is too small. To address this, Algorithm 2 includes a check to ensure that at least 10,000 elements are present before activating Squire (Line 2); otherwise, the host core handles sorting directly (Line 7).

7.3.2 1D Dynamic Programming: Chain

We describe the process of integrating the Chain kernel into Squire. First, we show the pseudocode for the baseline version of the Chain kernel. Next, we outline generic software modifications to enable parallelism, and finally, we show the necessary changes to integrate Chain into Squire.

Baseline Chain Kernel

Algorithm 3 shows the pseudocode for the original chain kernel. The function `CHAIN` receives an array of anchors sorted by position in the reference (Line 1). The kernel consists of two nested loops. The outer loop (Line 3) goes through all the anchors sorted by reference position, while the inner loop (Line 4) iterates through the `T` anchors prior to anchor `i` and performs a *match-up* between each of them and anchor `i`. Line 5 corresponds to the calculation of α and β in Equation 2.4, and Line 6 to the addition of $f(j)$ in Equation 2.4. Line 7 performs the maximum, obtaining $f(i)$. Notice that Line 5 can be computed in parallel for all the anchors, while Line 6 must wait for the generation of `F[j]` by Line 7.

Algorithm 3 Chain kernel baseline version

```
1: function CHAIN(A[N])                                   ▷ A: anchors array
2:     T = 5000
3:     for i = 0; i < N; i++ do
4:         for j = i − 1; j ≥ i − T; j−− do
5:             AUX[j] = $\alpha$(A[i], A[j]) − $\beta$(A[i], A[j])
6:             AUX[j] += F[j]                             ▷ Consume F[j]
7:         F[i] = MAX(AUX)                                ▷ Generate F[i]
```

Algorithm 4 Chain kernel Squire version

```
1: function CHAIN_WORKERS(A[N])                           ▷ A: Anchors array
2:     T = 64
3:     for i = id_worker(); i < N; i += num_workers() do
4:         for j = i − T; j ≤ i − 1; j++ do
5:             AUX[j] = $\alpha$(A[i], A[j]) − $\beta$(A[i], A[j])
6:         for j = i − T; j ≤ i − 1; j++ do
7:             if AUX[j] $\neq -\infty$ then
8:                 wait_gcounter(j + 1)
9:                 AUX[j] += F[j]                         ▷ Consume F[j]
10:        F[i] = MAX(AUX)                                ▷ Generate F[i]
11:        inc_gcounter()
12:    stop_worker()
```

Enabling Fine-Grain Parallelism

To delay the consumption of F[i] from Line 7 to Line 6, we alter the order of the inner loop (Line 4). To achieve this, we traverse the anchors in reverse order, i.e., from i-T to i-1. In addition, to isolate dependency-free parallelism from the dependencies imposed by Line 6, we fission the inner loop (Line 4), effectively detaching the computation of α and β in Line 5 from the addition in Line 6.

By default, the chain algorithm has a threshold on the number of anchors it visits backward (T in Lines 2 and 4). However, the best match-up is typically found during the initial iterations. In addition, the chain implements some heuristics to stop the inner loop earlier. For example, if the match-up scores are below a threshold. Therefore, the Chain kernel visits fewer anchors, and typically only the first few are useful. Consequently, we can reduce T with a negligible penalization in accuracy. We observe a misprediction rate lower than 9 per million when setting T to 64. Therefore, we use this value for our final evaluation. Although the overall Minimap2 accuracy remains almost unchanged, limiting T skips some match-ups, and some computation shifts to the align stage.

In the original implementation, after performing the chain stage, there is a second opportunity to rerun the chain algorithm if the area covered by the anchors is not large enough. The chain kernel is executed again with looser parameters and simpler versions of α and β functions. We modify the second chain run to use the same function as in the first chain run while applying the new parameters. This simplifies the implementation process while preserving the output of the original algorithm. As a summary of the software modifications:

- We have reversed the order in which we traverse the inner loop (Line 4).
- We fission the inner loop. Line 5 will be executed in the first loop, and Line 6 in the second one.
- We limit the number of anchors visited backward to 64 (`T` in Lines 2 and 4).
- We reformulate the second chain run to use the same function as in the first run.

Algorithm 4 shows the modified code with all these changes.

Squire Integration

Algorithm 4 shows the modified chain kernel adapted for Squire. The work is divided in a round-robin fashion (Line 3); e.g., with four workers, worker 0 computes the scores of anchors (0, 4, 8, ...), worker 1 computes the scores of anchors (1, 5, 9, ...), and so on.

Note that now the loop in Line 4 can be computed in parallel without dependencies using the workers. Once all the α and β values have been computed, the second loop in Line 6 proceeds to compute the remaining part, which has dependencies across workers (red lines in Figure 2.6). The dependencies are expressed by waiting on the *global counter* until it contains the desired value (Line 8), which means the dependent `F[j]` has been computed. Once the current `F[i]` is computed (Line 10), we increment the *global counter* to notify (Line 11) the consumers of that value.

When β (the penalization score) is high enough, we can stop the computation for that match-up. For these cases, we add a conditional statement (Line 7). Note that bypassing the `wait_gcounter` instruction could cause a race condition. For this purpose, we have implemented the mechanism described in Section 7.2.2, where we enforce the order of the increments in the *global counter*.

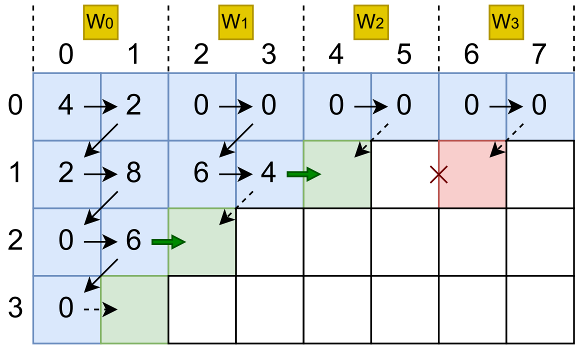


Figure 7.3: DTW work distribution among workers. Worker 0 (W_0) computes columns 0 and 1, worker 1 (W_1) columns 2 and 3, worker 2 (W_2) columns 4 and 5, and worker 3 (W_3) columns 6 and 7. The workers compute the cells following the path indicated by the arrows.

7.3.3 2D Dynamic Programming: DTW

We now detail how to use Squire to exploit fine-grain parallelism in DTW. Other well-known 2D DP kernels (e.g., Smith-Waterman, Needleman-Wunsch, etc.) exhibit the same patterns when computing the DP matrix.

Figure 7.3 shows a graphical scheme of how Squire would compute the DTW matrix. A set of consecutive columns is assigned to each worker; worker 0 (W_0) computes columns 0 and 1, worker 1 (W_1) columns 2 and 3, and so on. Each cell *(i,j)* of the matrix has a dependency with cells *(i-1,j)*, *(i,j-1)* and *(i-1,j-1)*. The workers compute their columns in a row-wise order. Hence, they do not have to worry about the vertical and diagonal dependencies. To solve horizontal dependencies at the boundaries, the *local counters* from the *synchronization module* are used.

Algorithm 5 shows the pseudocode for the Squire version of DTW. First, the work is evenly divided among the workers (Lines 2 and 3). The outer loop iterates through the rows (Line 4), while the inner loop iterates through the assigned columns of the corresponding worker (Line 7). The equations of DTW are implemented in Lines 8, 9, and 10. To synchronize workers at the boundaries, worker *x* increments the *local counter* *x* when it finishes a row (Line 11), so worker *x+1* knows the dependency for that row is solved. Similarly, when worker *x* starts a row, it waits for worker *x-1* to finish its chunk of the row (Line 6). Note that worker 0 has no horizontal dependencies. Therefore, it skips the synchronization (Line 5).

7.3.4 2D Dynamic Programming: WFA

We implemented a Squire version for the Wavefront Alignment algorithm (WFA) [56]. WFA consists of two kernels: extend and compute, which are executed in a ping-pong

Algorithm 5 DTW kernel Squire version

```
function DTW_WORKERS(A[N], B[M])
    start = id_worker() × (M / num_workers())
    end = (id_worker() + 1) × (M / num_workers())
    for i = 0; i < N; i++ do
        if id_worker() ≠ 0 then
            wait_lcounter(id_worker() − 1, i + 1)
        for j = start; j < end; j++ do
            PREV ← MIN(M[i-1,j], M[i,j-1], M[i-1,j-1])
            COST ← COST_FUNC(A[i], B[j])
            M[i,j] ← PREV + COST
        inc_lcounter(id_worker())
    stop_worker()
```

manner: extend, compute, extend, compute, and so on. The core offloads to Squire a combined instance of extend and compute (referred to as a wave). After that, the core synchronizes with all the workers and calculates heuristics and stopping conditions. If necessary, the core offloads a new wave, and this process is repeated until the stopping condition is met.

Despite the total execution time being considerable, the amount of work offloaded to Squire with each wave is minimal, about 2000 cycles. The overhead of moving the data through the caches between Squire and the host core after the computation of each wave is not amortized. As a result, we observed slowdowns of approximately 5%. Therefore, we do not include these results in the evaluation.

7.3.5 Discussion

Throughout the development of Squire, we have examined several ideas regarding certain implementation details.

For communication among the workers, we considered message-passing through a crossbar, a FIFO, or a ring. Finally, we used the shared L2 cache since the worker's messages are part of the output, avoiding the need to write the same data twice.

We also considered other synchronization mechanisms besides the *counters*. Initially, the message-passing mechanism would be used as the synchronization point. We explored expanding the *synchronization module* functionality, allowing subtractions and arbitrary additions over the counter. The current *synchronization module* specifications are sufficient for the algorithms we use, but they could be extended in the future.

Table 7.2: Size of the datasets used in the evaluation.

	RADIX	SEED	CHAIN	SW	DTW
# experiments	15	5	5	5	2
# inputs/exp.	8 arrays	24 seq.	24 arrays	6195 align.	5000 align.
Input avg. size	53536 elems.	23014 bps	53536 anchors	1373 bps	221 samples
Size st. dev.	36886	15075	36886	2950	101
Mem. footprint	837 KB	22.5 KB	837 KB	3.27 KB	1.72 KB

Finally, as we explained in Section 7.2.1, workers must increase the *global counter* in order. We considered solving this problem in software by waiting for the *global counter* to reach its correct value before incrementing it, e.g., in Algorithm 4 adding `wait_gcounter(i)` between Lines 10 and 11. However, this approach would harm available parallelism and performance.

7.4 Evaluation Methodology

7.4.1 Architectural Simulation

We prototype Squire using the Gem5 simulator v23.0 [114, 115] (see Section 3.3). The simulated system is described in Table 3.2. Each host core features a Squire engine that faithfully models the described architecture.

7.4.2 Workloads and Inputs

Table 7.2 details the inputs used for each kernel. All the inputs have been extracted from real genomic and signal-processing datasets. A correctness evaluation has been performed, so the functionality remains the same. All the kernels produce the same result with Squire as with the CPU version, except for CHAIN, where we observe a misprediction rate lower than 9 per million due to the software modifications from Section 7.3.2. To evaluate Squire, we use the five kernels described below.

Radix Sort (RADIX) Radix sort is shown in Section 2.4.1. We did 15 experiments. In each experiment, we sort eight arrays, one for each out-of-order core. Some of the arrays used for radix sort have less than 10,000 elements, thus avoiding offloading work to Squire (see Section 7.3.1). We divide the array into equal chunks and use Squire to sort them (see Section 7.3.1).

Table 7.3: Input sequence datasets.

	Sequencing system	Avg. seq. length	Accuracy
ONT [192]	Oxford Nanopore	17,710	85%
PB CLR [193]	PB Sequel II System	6,739	88%
PB HF 1 [192]	PacBio HiFi	12,858	99.99%
PB HF 2 [192]	PacBio HiFi	15,602	99.99%
PB HF 3 [192]	PacBio HiFi	14,149	99.99%

Seeding (SEED) We evaluate the seeding algorithm from Minimap2 [24] (see Section 2.3.1). We use five input sequences datasets (see Table 7.3). Each one of the datasets has 24 sequences, hence, each out-of-order core performs three seeding processes. The most consuming part of seeding is the final sorting of the seeds. Therefore, we use the Squire version of the radix sort algorithm explained above.

Chain (CHAIN) Chain is a dynamic algorithm used in Minimap2 [24] (see Section 2.3.2). As in seeding kernel, we use five input sequences datasets, where each one has 24 sequences, resulting in three chain processes per out-of-order. The anchors are assigned to the workers in a round robin manner (see Section 7.3.2).

Smith-Waterman (SW) Smith-Waterman is a 2D dynamic programming algorithm used for aligning (see Section 2.3.3). We use the same datasets used in seeding and chain. These datasets produce several alignments that we use as inputs in Smith-Waterman. The work has been distributed using the same approach as for DTW (see Section 7.3.3).

Dynamic Time Warping (DTW) Dynamic Time Warping is a 2D dynamic programming algorithm (explained in Section 2.4.2) used for signal processing. We use two synthetic datasets of 5,000 alignments of floating point numbers. The small dataset has an average alignment size of 133 samples, while the larger one has 380 samples on average. Each worker is the responsible of computing several contiguous columns (see Section 7.3.3).

7.4.3 Evaluation of an End-to-End Read-Mapping Application

With the kernels introduced above, we have built an end-to-end read-mapping tool that receives a set of sequences and produces alignments. We use Minimap2 [24] as the skeleton for our read-mapper since two of the evaluated kernels are extracted from Minimap2 (SEED and CHAIN). We combine SEED, CHAIN, and SW into a single application to set up a read-mapper that serves as a test-bench for evaluating the speedup achieved on an end-to-end application when using Squire.

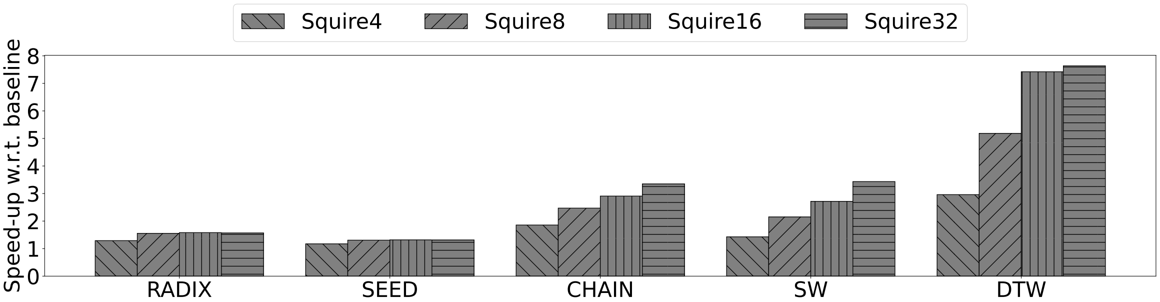


Figure 7.4: Squire evaluation for the five kernels described in Section 7.4.2. We evaluate Squire with 4, 8, 16, and 32 workers.

Table 7.3 shows the inputs used to evaluate the end-to-end application. All these inputs are from sequencing systems that have sequenced the human genome. Note the differences in the accuracy of the inputs, which refer to the errors introduced by the machines during the sequencing (reading) process. ONT and PBCLR have an accuracy of 85% and 88%, respectively, while PBHF inputs have an accuracy of nearly 100% (see Section 2.1). This difference in accuracy is translated into different behavior during the read-mapping process. A higher accuracy implies a lighter volume of work in the align stage when using SW.

We select the 18 most time-consuming sequences from each input set to keep simulation time in Gem5 manageable. This allows us to reduce the execution time while maintaining the application's behavior.

7.5 Evaluation

In this section, first, we show how Squire can speed up the five evaluated kernels. Then, we evaluate the impact the synchronization module has on the design by modifying the implementation to use software mutexes instead of Squire's hardware module. We also evaluate the end-to-end read-mapper to understand how Squire improves a full application. Finally, we perform a design space exploration to justify the size of the caches used by the workers and perform an area and energy consumption study.

7.5.1 Performance Evaluation

Figure 7.4 shows the performance evaluation of Squire for the five kernels described in Section 7.4.2 when changing the number of workers.

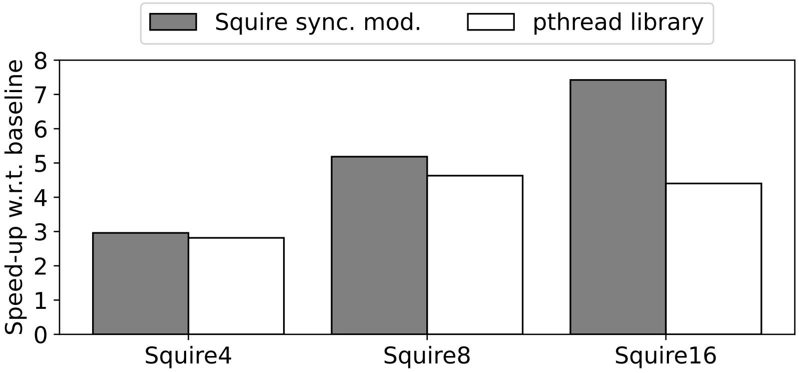


Figure 7.5: Squire performance evaluation when using the synchronization module vs the pthread library with 4, 8, and 16 workers.

For RADIX and SEED, Squire achieves diminishing returns when using from 8 up to 32 workers, due to small input data size. As explained in Section 7.3.1, we stablish a minimum of 10,000 elements to use Squire. Below that, the initialization of Squire becomes the bottleneck in the sorting process. Maximum performance is achieved with 16 workers, reaching 1.58× for RADIX and 1.32× for SEED.

Employing 32 workers for CHAIN and SW leads to noticeable speedups, unlike RADIX and SEED, reaching 3.35× and 3.43× with respect to the base system, respectively. The speedups from 16 to 32 workers are 1.19× and 1.26× for CHAIN and SW.

Finally, Squire obtains remarkable speedups up to 32 workers for DTW, reaching 7.64×. However, we consider 16 workers the optimum point with a speedup of 7.42×.

These results show that Squire can enable fine-grain parallelism on dependency-bond kernels. While Squire scales well with worker count if there is enough work to compute, we advocate that a balanced design should have between 8 and 16 workers. Doubling the number of workers to 32 does not compensate for the cost in the common case.

7.5.2 Synchronization Module Evaluation

Figure 7.5 shows the benefits of using the synchronization module in Squire for DTW kernel. We show the results up to 16 workers since we have considered it the optimum point in Section 7.5.1. We instantiate Squire without the synchronization module and synchronize through the *pthread mutex* library. We use DTW for this experiment since it is one of the kernels (along with SW) that uses the *local counters*.

The synchronization module improves performance for any number of workers, increasing in importance as the number of workers increases. We observe a speedup of up 1.69× when using the synchronization module with 16 workers.

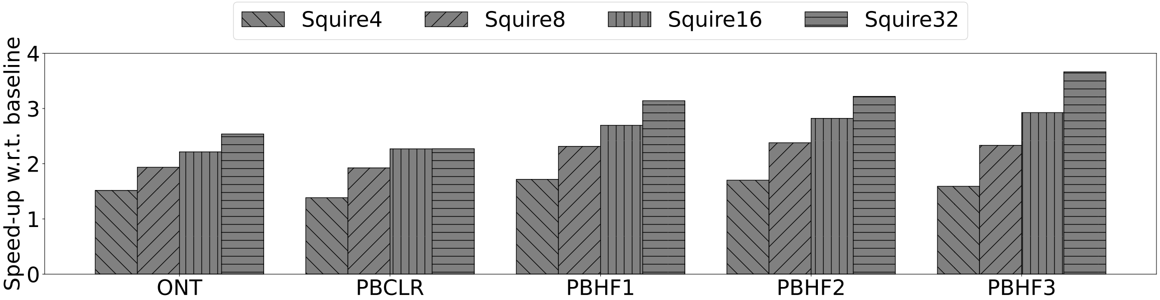


Figure 7.6: Squire evaluation for the end-to-end read-mapping application described in Section 7.4.3 with 4, 8, 16, and 32 workers.

7.5.3 End-to-End Application Evaluation

Figure 7.6 shows the performance evaluation of Squire for an end-to-end application for the five inputs described in Table 7.3. We evaluate Squire with 4, 8, 16, and 32 workers. As stated in Section 7.4.3, the different datasets behave differently during the read mapping process; thus, the align stage has less weight for the PBHF inputs.

When looking at the whole read-mapping end-to-end application, Squire achieves speedups of up to 3.66×. For all the inputs, Squire scales well with worker count and accomplishes its best performance with 32 workers. For ONT and PBCLR inputs, Squire achieves speedups of 2.54× and 2.27×, respectively. For PBHF inputs, Squire achieves speedups higher than 3×. A higher accuracy of the sequencing systems implies more work to process but smaller chunks of work, which favors Squire. As sequencing technologies keep improving, this trend will consolidate and devices like Squire will be more effective.

7.5.4 Cache Size Exploration

Each worker has its own private L1 data and instruction caches, which will largely determine the area that Squire will occupy. For this reason, we perform a design space exploration study to make a judicious choice and minimize the area and power overhead of the design.

To evaluate the cache sizes, we use the end-to-end application and fix the number of workers to 16. We use the ONT input dataset. To evaluate the instruction cache size, we fixed the data cache size to 8 KB and vice versa. To measure performance, we use misses per kilo instructions (MPKI).

Figure 7.7 shows MPKI when varying cache sizes. For the instruction cache, we observe a drastic change when going from 512 B to 1 KB. Beyond that, MPKI remains close to zero. For the data cache, we see consistent improvement up to 8 KB, which we

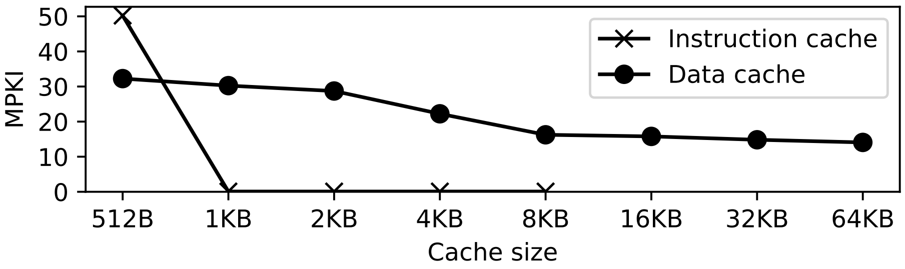


Figure 7.7: Misses per kilo instructions (MPKI) when changing the cache size for the instruction and data caches with 16 workers.

consider the sweet spot. A larger 16 KB data cache improves MPKI marginally at a large cost. Therefore, we have employed 1 KB and 8 KB as instruction and data cache sizes, respectively, for all the experiments in this section.

7.5.5 Area Overhead

We use the Arm Neoverse N1 to model the out-of-order core. Using the public data for an N1 [194] at 7nm, the floor planned area is given as 1.15 mm^2.

The workers we model could be compared to the Arm Cortex M35P microprocessor. This processor is a 4-stage dual-issue in-order core that runs at 2.4 GHz. We select an M35P as an upper bound for modeling the area. The specifications proposed for the workers are much less demanding than the M35P, since the workers do not need any interaction with the operating system, interruptions, or system calls.

Using public data for an M35P at 40LP [195], the floor planned area is given as 0.091 mm^2. This area already includes a 16 KB instruction cache. The instruction cache included in the M35P is larger than the caches we employ since we employ 1 KB for L1I and 8 KB for L1D (see Section 7.5.4). Also, the M35P is a processor capable of booting an operating system, and our workers do not require as many functionalities as the M35P. Therefore, we must consider that we are overestimating the area of the workers.

When employing 16 workers, the total area overhead at 40nm would be 1.456 mm^2. To estimate the area with 7 nm, we scale these numbers, considering fin pitch, gate pitch, and interconnect pitch, using data from several studies [196–201] to arrive at a 12× area reduction when moving from 40 nm to 7 nm. Thus, obtaining an area for a Squire component of 0.121 mm^2.

Therefore, we could place a 16-worker Squire component per core with an area overhead of 10.5%.

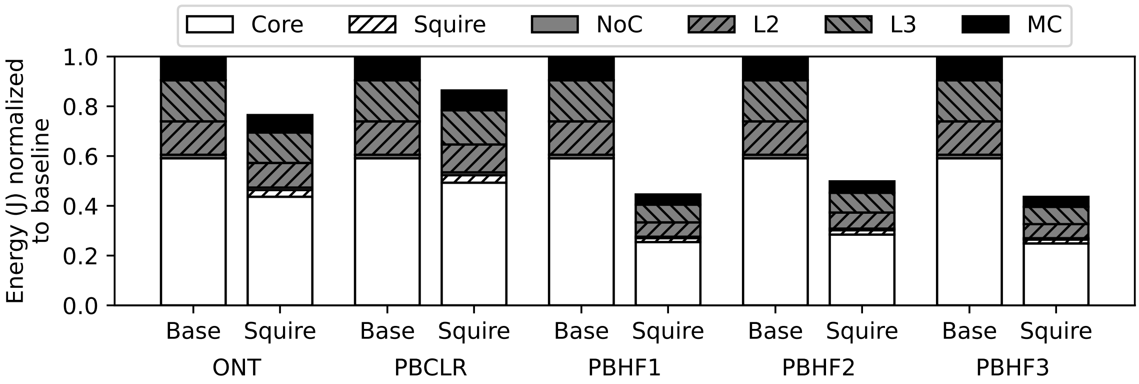


Figure 7.8: Energy consumption comparison between the baseline and when using Squire with 16 workers for the end-to-end application.

7.5.6 Energy Consumption

To estimate the Squire energy consumption, we use McPAT 1.3 [202] with the enhancements proposed by Xi et al. [203]. We performed this estimation using a process technology node of 22 nm, a supply voltage of 0.8 V, and the default clock gating scheme.

Figure 7.8 shows the energy consumption of the baseline when using Squire with 16 workers for the end-to-end application. The executions using Squire achieve significant energy reductions of up to 56% over the baseline system for the PBHF3 input. Similarly, Squire reduces consumption by 55% and 50% for PBHF1 and PBHF2 inputs. The ONT and PBCLR inputs show a more modest energy reduction of 24% and 14%, respectively.

The host cores are the most energy-consuming components, followed by the L2 and L3 caches. The memory controllers and the NoC have a marginal energy consumption. The energy overhead introduced by Squire is small; we observe an energy overhead of around 6% with respect to the host cores, which is largely offset by the reduction in the rest of the components.

7.6 Comparison with State-of-the-Art

7.6.1 Accelerating Smith-Waterman

We compare Squire performance against other proposals that accelerate Smith-Waterman. Among them, we find two CPUs, two GPUs, one Processing-in-Memory (PIM) platform, and two Domain-Specific Accelerators (DSA). We measure the performance using giga-cells updated per second (GCUPS). We also show performance per area (GCUPS/mm^2) and performance per watt (GCUPS/W). We extract these numbers from the article for

Table 7.4: Squire comparison with other proposals to accelerate Smith-Waterman. The table includes the alignment model, the hardware platform used, gigacells computed per second (GCUPS), GCUPS per area (GCUPS/mm^2), GCUPS per watts (GCUPS/W), and where these results were extracted from. PIM is Processing-in-Memory, DSA is Domain-Specific Accelerators, and GPA is General-Purpose Accelerator.

	Reference	Alignment Model	Platform	GCUPS	GCUPS/mm^2	GCUPS/W
CPU	[24]	Gap-affine	2 × Xeon Platinum 8358	40	-	0.08
	[204]	Gap-affine 2 pieces	2 × Xeon Platinum 8358	46	-	0.09
GPU	[76]	Gap-affine	Tesla H100	2,850	3.50	8.14
	[205]	Gap-affine	4 × Tesla H100	11,440	3.51	8.17
PIM	[91]	Gap-affine	UPMEM	6,690	0.02	17.42
DSA	[84]	Gap-affine	GACT	1,024	11.96	232.73
	[69]	Edit distance	GenASM	2,048	201.26	659.79
GPA	[209]	Gap-linear	**Squire**	317	327.48	231.24

each proposal, except for the two CPU implementations [24, 204], which were taken from the GPU implementation by Zeni et al. [205]. We use the area and power that appear in the corresponding articles [69, 84] or on web pages with the systems specifications [206–208]. The chip area of the Intel Xeon Platinum 8358 is not publicly available.

Table 7.4 shows the comparison with other proposals that accelerate Smith-Waterman. We observe that a general purpose platform, such as a CPU, obtains a really low performance when compared with Squire, being almost 8× slower than Squire. When moving to more specific architectures, such as GPUs and PIM platforms, we observe a large improvement in performance, being up to 36× faster than Squire. However, the amount of area and power needed by these platforms is huge. Squire improves performance per area from 93× up to 16374× and performance per power from 13× up to 28×.

We include two accelerators that focus exclusively on Smith-Waterman, GACT accelerator from Darwin [84] and GenASM [69]. To calculate the GCUPS of such proposals, we use the number of processing elements, arrays instantiated, and frequency used in the articles, and assume that all the processing elements from all the arrays are computing cells every cycle. DSAs achieve higher performance than Squire, up to 6.5× higher. However, in terms of GCUPS per area and power, Squire obtains results in the same order of magnitude: 1.6× better performance per area and 2.9× worse performance per power than GenASM. We must take into account that the DSA proposals can only compute Smith-Waterman, while Squire can be used for other algorithms.

Table 7.5: Qualitative comparison among several general-purpose accelerators.

	Walkers[93]	**Transmuter[210]**	**Versa[95]**	**UPMEM[91]**	**Squire**
Programable	✓	✓	✓	✓	✓
Rich ISA support	×	✓	✓	✓	✓
Flexible datapath	×	✓	✓	✓	✓
Virtual memory support	✓	✓	✓	×	✓
Rapid synchronization	×	✓	✓	×	✓
Private accelerator per core	✓	×	×	×	✓

7.6.2 General-Purpose Accelerators

We identify general-purpose hardware accelerators like the Walkers [93], a programmable hardware accelerator for traversing hash tables in a database. Transmuter [210] and Versa [95] propose a matrix of general-purpose processing elements interconnected by a mesh. The accelerator is shared by all the cores of the chip. The system can be reconfigured as a systolic array of processing elements, as a typical memory hierarchy, or as a private scratchpad for each processing element. UPMEM [94] is the first publicly available general-purpose programmable PIM system. AIM [91] is a sequence alignment framework that uses UPMEM for the evaluation.

Table 7.5 shows a qualitative comparison between Squire and the other general-purpose hardware accelerators. The Walkers have a fixed pipeline, forcing the data to traverse it, thus limiting its flexibility. In addition, the Walkers have a very limited ISA support and do not have any method for synchronization. The compute units must execute the code completely in parallel without communicating with the rest. Transmuter and Versa instantiate one accelerator shared for all the cores of a chip. To exploit all the computing resources, the application should be split into two sets of threads, one that is executed on the accelerator and the other on the cores. By contrast, Squire is a simpler private accelerator for each core. The application is divided into as many threads as cores, and then each core performs nested parallelism in its Squire. Moreover, the interconnection networks in Transmuter and Versa (among processing elements and between the accelerator and the host cores) add communication latencies with respect to Squire. UPMEM is a processing in memory component that instantiates several processors per physical memory cell, thus losing the virtual memory capability and limiting the address range the processors can access. Each processor controls a chunk of the memory and cannot access the rest of the system. To communicate with other processors, they must do it through main memory, which hinders performance [92].

A big.LITTLE architecture is composed of several cores with different design targets: computational performance and power efficiency [211]. All are capable of running system code and are visible to the operating system. In contrast, Squire is a set of very simple cores with no system support and is subordinated to a host core. Moreover, Squire is equipped with a synchronization module that allows for fast communication among its workers (see Section 7.5.2).

Chapter 8

Conclusions and Future Work

This thesis has presented several techniques to accelerate genomic pipelines, and more specifically, dynamic programming algorithms used in read mapping tools. We have contributed to the field of genomics with a new data structure for exact matching algorithms, several software optimizations for current kernels and tools, the porting of three kernels and one read mapping application to the ARM architecture, and a general-purpose hardware accelerator to exploit fine-grain parallelism on dependency-bound kernels.

Firstly, we propose COFI, a **CO**mpressed **F**M-**I**ndex for large K-steps. Contrary to prior proposals, COFI's main data structure has constant size with respect to the value of k. This enables large k-step searches that present better trade-offs in terms of throughput per unit of data moved. We show that COFI consistently outperforms a state-of-the-art proposal with improvements of up to 2.14×. On average, COFI obtains 1.46× and 1.39× improvements on KNL and Skylake-based systems, respectively. COFI is available at https://gitlab.bsc.es/rlangari/cofi.

Then, we focus on the optimization and porting of kernels and read mapping tools to the ARM architecture.

We port a well-known genomic application, BWA-MEM2. Our porting effort enables the use of BWA-MEM2 on any ARMv8-A system that supports SVE. We evaluate the optimized version of the port on the A64FX and show almost linear thread-level and the expected data-level (SVE) parallelism. We compare the A64FX system with an established SKX system and show that the SKX system performs better: 1.89× on average for 48 threads and a 4.0% when comparing on a socket-to-socket basis. However, in terms of energy-to-solution, the A64FX presents better results, both when comparing executions with the same thread count using 48 threads, by 11.6%, and when comparing socket-to-socket executions, by 26.4%. We conclude that the strength of the A64FX is to be an energy-efficient CPU with higher memory bandwidth than traditional HPC systems. In contrast, the A64FX has a higher memory access latency than other HPC systems, which negatively affects its performance when executing applications with random memory accesses. The port of BWA-MEM2 is available at https://gitlab.bsc.es/rlangari/bwa-a64fx.

We develop GenArchBench, a benchmark suite that consists of 13 computationally-demanding CPU kernels from the most widely-used genomic tools. 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. Furthermore, this work introduces code adaptations and optimizations of the genomic kernels targeting ARM HPC CPUs. 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). 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. Under the coordination of Lorién López-Villellas, I contributed to GenArchBench by porting and optimizing BSW, FAST-CHAIN, and FMI kernels. GenArchBench is available at https://github.com/LorienLV/genarchbench/releases/tag/1.0.0.

Finally, we propose Squire, a general-purpose accelerator for dependency-bound fine-grain parallelism. Squire consists of a set of simple general-purpose in-order cores, called workers, and a synchronization module for rapid synchronization. Each host core is augmented with a Squire engine to offload fine-grain tasks. We evaluate Squire on a simulated multicore SoC, obtaining speedups of up to 7.64× in dynamic programming kernels, and an acceleration for an end-to-end application of 3.66×. We also evaluate the usage of resources and show that Squire achieves an energy reduction of up to 56% with an area overhead of 10.5% per core.

The results of this thesis have been successfully integrated into several genomic tools and libraries targeting high-performance computing (HPC) servers. We are actively developing, maintaining, and supporting all the open-source code released as part of these contributions, ensuring its long-term sustainability and usability for the research community. Several colleagues and external collaborators have already adopted and extended our code in their own projects, demonstrating its practical impact and versatility.

We opened a pull request and a discussion thread in the central repository of BWA-MEM2 [212] to include our ARM architecture port, thereby expanding its compatibility with emerging HPC systems. In addition, our ARM port of BWA-MEM2 has been proposed as the official replacement for the existing version in the Bioconda framework [213],

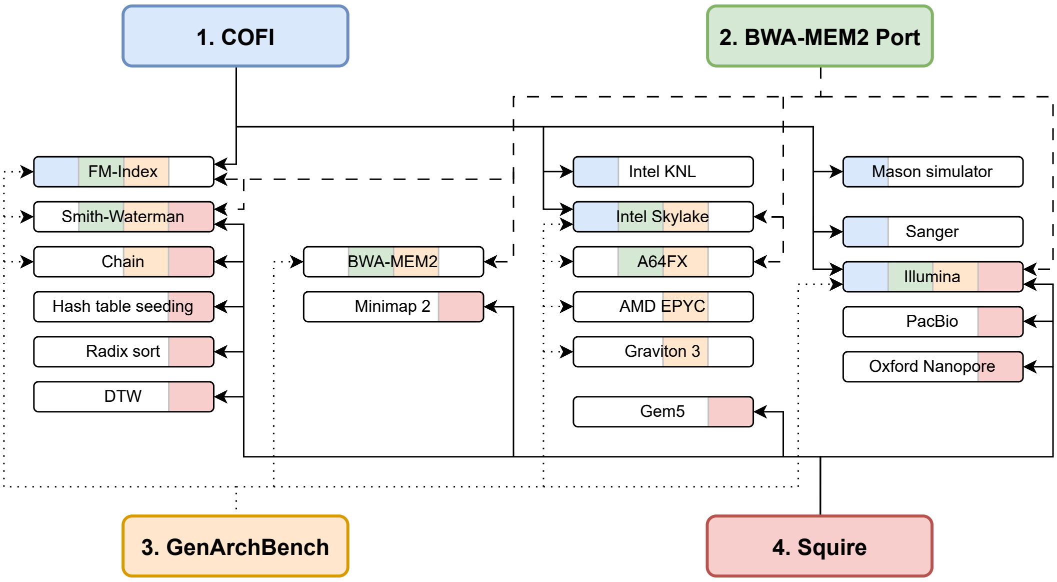


Figure 8.1: Summary of algorithms, tools, hardware platforms, simulators, and sequencing technologies used in the four contributions of this thesis.

broadening its accessibility to bioinformatics users and developers. In parallel, we have contributed to enhancing and optimizing the x86 source code of both Minimap2 [214] and BWA-MEM2 [212], improving their performance and portability across multiple hardware platforms.

Figure 8.1 provides an overview of all the algorithms, tools, hardware platforms, simulators, and sequencing technologies that have benefited from our work and to which we have contributed throughout this thesis.

8.1 Future Work

The work presented in the thesis has led to some projects we are currently working on.

Currently, we are working on the Dynamic Programming Extension (DPE), which includes new vector instructions for dynamic programming kernels. DPE is built on top of ARM vector extension SVE. So far, DPE includes five new instructions to tackle dynamic programming kernels, such as Smith-Waterman, Dynamic Time Warping, Chain and Wavefront. Our most recent results show a reduction in the number of executed instructions of up to 28.5%. We plan to implement DPE in Gem5 and evaluate its performance using several dynamic programming kernels.

In the future, an interesting and promising direction for research is the exploration of custom-precision and mixed-precision arithmetic in genomic kernels. Previous work by Suzuki and Kasahara [97] has demonstrated the feasibility of performing sequence alignment using 8-bit integer representations, significantly reducing data movement and computational overhead. Extending this concept, other performance-critical kernels that currently represent bottlenecks could also benefit from reduced operand precision, provided that accuracy and biological relevance are preserved. Adopting lower-precision arithmetic would decrease the overall memory footprint, enhance data locality, and increase arithmetic intensity, thereby improving both performance and power/energy efficiency on modern high-performance architectures.

8.2 Publications

Here we list the publications resulted from the work of the thesis:

- **Rubén Langarita**, Adrià Armejach, Javier Setoain, Pablo Ibáñez-Marín, Jesús Alastruey-Benedé, and Miquel Moretó (2020). Compressed sparse FM-index: Fast sequence alignment using large K-steps. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 19(1), 355-368.

- **Rubén Langarita**, Adrià Armejach, Pablo Ibáñez-Marín, Jesús Alastruey-Benedé, and Miquel Moretó (2023). Porting and optimizing BWA-MEM2 using the Fujitsu A64FX processor. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 20(5), 3139-3153.

- Lorién López-Villellas, **Rubén Langarita**, Asaf Badouh, Víctor Soria-Pardos, Quim Aguado-Puig, Guillem López-Paradís, Max Doblas, Javier Setoain, Chulho Kim, Makoto Ono, Adrià Armejach, Santiago Marco-Sola, Jesús Alastruey-Benedé, Pablo Ibáñez, and Miquel Moretó (2024). GenArchBench: A genomic benchmark suite for arm HPC processors. *Future Generation Computer Systems (FGCS)*, 157, 313-329.

- **Rubén Langarita**, Adrià Armejach, Santiago Marco-Sola, Pablo Ibáñez-Marín, Jesús Alastruey-Benedé, and Miquel Moretó. Squire: A General-Purpose Accelerator to Exploit Fine-Grain Parallelism on Dependency-Bound Kernels. *Parallel Architectures and Compilation Techniques (PACT)*, 2025.

Despite their recent publication, these works have already been cited in studies proposing advances in genome analysis through novel algorithmic approaches [1, 215, 216], software-based techniques [217], and hardware acceleration [218, 219]. They have also been cited in articles that explore the use of vector extensions in HPC [220–222] Finally, they also appear in two state-of-the-art reviews [223, 224].

References

[1] M. Alser, J. Lindegger, C. Firtina, N. Almadhoun, H. Mao, G. Singh, J. Gomez-Luna, and O. Mutlu, "From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures," *Computational and Structural Biotechnology Journal*, vol. 20, pp. 4579–4599, 2022.

[2] "Smith-Waterman Wikipedia." https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm. Accessed: 2025-05-21.

[3] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Communications surveys & tutorials*, vol. 18, no. 1, pp. 732–794, 2015.

[4] Y. Hu, Y. Liu, and Z. Liu, "A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC," in *2022 14th International Conference on Computer Research and Development (ICCRD)*, IEEE, Jan. 2022.

[5] A. Conesa, P. Madrigal, S. Tarazona, D. Gomez-Cabrero, A. Cervera, A. McPherson, M. W. Szcześniak, D. J. Gaffney, L. L. Elo, X. Zhang, and A. Mortazavi, "A survey of best practices for RNA-seq data analysis," *Genome Biology*, vol. 17, Jan. 2016.

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

[7] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big Data: Astronomical or Genomical?," *PLOS Biology*, vol. 13, p. e1002195, July 2015.

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

[9] "BASECLEAR: Sanger sequencing." https://www.baseclear.com/genomics/sanger-sequencing. Accessed: 2025-10-16.

[10] "MD Anderson Cancer Center: Sanger Sequencing." https://www.mdanderson.org/research/research-resources/core-facilities/advanced-technology-genomics-core/services-and-fees/sanger-sequencing.html. Accessed: 2025-10-16.

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

[12] E. Ukkonen, "Finding approximate patterns in strings," *Journal of Algorithms*, vol. 6, p. 132–137, Mar. 1985.

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

[14] A. Chacón, J. C. Moure, A. Espinosa, and P. Hernández, "n-step FM-index for faster pattern matching," *Procedia Computer Science*, vol. 18, pp. 70–79, 2013.

[15] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 314–324, IEEE, 2019.

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

[17] "Illumina sequencing technology." https://emea.illumina.com/science/technology/next-generation-sequencing/sequencing-technology/2-channel-sbs.html. Accessed: 2025-10-28.

[18] "Bonito." https://github.com/nanoporetech/bonito. Accessed: 7 January 2023.

[19] R. R. Wick, L. M. Judd, and K. E. Holt, "Performance of neural network basecalling tools for Oxford Nanopore sequencing," *Genome Biology*, vol. 20, June 2019.

[20] "How does CCS work." https://ccs.how/how-does-ccs-work.html. Accessed: 2025-10-28.

[21] H. Li, "KSW2 repository." https://github.com/lh3/ksw2, 2018. Accessed: 2024-07-26.

[22] M. Šošić and M. Šikić, "Edlib: a C/C++ library for fast, exact sequence alignment using edit distance," *Bioinformatics*, vol. 33, p. 1394–1395, Jan. 2017.

[23] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *arXiv preprint arXiv:1303.3997*, 2013.

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

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

[26] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, p. 357–359, Mar. 2012.

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

[28] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pp. 390–398, IEEE, 2000.

[29] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs,” *Nucleic Acids Research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[30] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *Journal of Molecular Biology*, vol. 48, p. 443–453, Mar. 1970.

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

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

[33] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, “The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data,” *Genome Research*, vol. 20, p. 1297–1303, July 2010.

[34] A. Rimmer, H. Phan, I. Mathieson, Z. Iqbal, S. R. F. Twigg, A. O. M. Wilkie, G. McVean, and G. Lunter, “Integrating mapping-, assembly- and haplotype-based approaches for calling variants in clinical sequencing applications,” *Nature Genetics*, vol. 46, p. 912–918, July 2014.

[35] R. Luo, C.-L. Wong, Y.-S. Wong, C.-I. Tang, C.-M. Liu, C.-M. Leung, and T.-W. Lam, “Exploring the limit of using a deep neural network on pileup data for germline variant calling,” *Nature Machine Intelligence*, vol. 2, p. 220–227, Apr. 2020.

[36] Z. Zheng, S. Li, J. Su, A. W.-S. Leung, T.-W. Lam, and R. Luo, “Symphonizing pileup and full-alignment for deep learning-based long-read variant calling,” *Nature Computational Science*, vol. 2, p. 797–803, Dec. 2022.

[37] R. Poplin, P.-C. Chang, D. Alexander, S. Schwartz, T. Colthurst, A. Ku, D. Newburger, J. Dijamco, N. Nguyen, P. T. Afshar, S. S. Gross, L. Dorfman, C. Y. McLean, and M. A. DePristo, “A universal SNP and small-indel variant caller using deep neural networks,” *Nature Biotechnology*, vol. 36, p. 983–987, Sept. 2018.

[38] “Medaka.” https://github.com/nanoporetech/medaka. Accessed: 7 January 2023.

[39] M. Kolmogorov, J. Yuan, Y. Lin, and P. A. Pevzner, “Assembly of long, error-prone reads using repeat graphs,” *Nature Biotechnology*, vol. 37, p. 540–546, Apr. 2019.

[40] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, “Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation,” *Genome Research*, vol. 27, p. 722–736, Mar. 2017.

[41] R. Vaser, I. Sović, N. Nagarajan, and M. Šikić, “Fast and accurate de novo genome assembly from long uncorrected reads,” *Genome Research*, vol. 27, p. 737–746, Jan. 2017.

[42] D. Kim, L. Song, F. P. Breitwieser, and S. L. Salzberg, “Centrifuge: rapid and sensitive classification of metagenomic sequences,” *Genome Research*, vol. 26, p. 1721–1729, Oct. 2016.

[43] H. Sadasivan, J. Wadden, K. Goliya, P. Ranjan, R. P. Dickson, D. Blaauw, R. Das, and S. Narayanasamy, “Rapid Real-time Squiggle Classification for Read until using RawMap,” *Archives of Clinical and Biomedical Research*, vol. 07, no. 01, 2023.

[44] S. Kovaka, Y. Fan, B. Ni, W. Timp, and M. C. Schatz, “Targeted nanopore sequencing by real-time mapping of raw electrical signal with UNCALLED,” *Nature Biotechnology*, vol. 39, p. 431–441, Nov. 2020.

[45] A. Payne, N. Holmes, T. Clarke, R. Munro, B. J. Debebe, and M. Loose, “Readfish enables targeted nanopore sequencing of gigabase-sized genomes,” *Nature Biotechnology*, vol. 39, p. 442–450, Nov. 2020.

[46] D. E. Wood, J. Lu, and B. Langmead, “Improved metagenomic analysis with Kraken 2,” *Genome Biology*, vol. 20, Nov. 2019.

[47] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi, “CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers,” *BMC Genomics*, vol. 16, Mar. 2015.

[48] J. T. Simpson and R. Durbin, “Efficient de novo assembly of large genomes using compressed data structures,” *Genome Research*, vol. 22, p. 549–556, Dec. 2011.

[49] J. M. Herruzo, S. G. Navarro, P. Ibáñez, V. Viñals-Yufera, J. Alastruey, and O. Plata, “Accelerating Sequence Alignments Based on FM-Index Using the Intel KNL Processor,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2018.

[50] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, “FM-index on GPU: a cooperative scheme to reduce memory footprint,” in *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 1–9, IEEE, 2014.

[51] B. Ma, J. Tromp, and M. Li, “PatternHunter: faster and more sensitive homology search,” *Bioinformatics*, vol. 18, no. 3, pp. 440–445, 2002.

[52] “Genome Reference Consortium Human Reference 37.” http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/. Accessed: 2019-07-23.

[53] “Genome Reference Consortium Human Reference 38.” http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/. Accessed: 2019-07-23.

[54] M. Burrows and D. J. Wheeler, “A Block-Sorting Lossless Data Compression Algorithm,” Tech. Rep. 124, Digital Equipment Corporation, 1994.

[55] U. Manber and G. Myers, “Suffix arrays: a new method for on-line string searches,” *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

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

[57] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, “Accelerating read mapping with FastHASH,” *BMC Genomics*, vol. 14, Jan. 2013.

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

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

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

[61] T. Rognes and E. Seeberg, “Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors,” *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.

[62] M. Farrar, “Striped Smith–Waterman speeds database searches six times over other SIMD implementations,” *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2006.

[63] D.-H. Park, J. Beaumont, and T. Mudge, “Accelerating Smith-Waterman Alignment Workload with Scalable Vector Computing,” in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 661–668, IEEE, 2017.

[64] T. Rognes, “Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation,” *BMC bioinformatics*, vol. 12, no. 1, p. 221, 2011.

[65] T. Smith and M. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.

[66] R. Baeza-Yates and G. H. Gonnet, “A new approach to text searching,” *Communications of the ACM*, vol. 35, no. 10, pp. 74–82, 1992.

[67] S. Wu and U. Manber, “Fast text searching: allowing errors,” *Communications of the ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[68] G. Myers, “A fast bit-vector algorithm for approximate string matching based on dynamic programming,” *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[69] D. S. Cali, G. S. Kalsi, Z. Bingol, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, “GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020.

[70] D. Knuth, *The Art of Computer Programming*, vol. 3. Addison-Wesley, 1973.

[71] M. Rashid, L. Ardito, and M. Torchiano, “Energy consumption analysis of algorithms implementations,” in *2015 ACM/IEEE International Symposium on Empirical software engineering and measurement (ESEM)*, pp. 1–4, IEEE, 2015.

[72] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, “Choosing the” best” sorting algorithm for optimal energy consumption,” in *International Conference on Software and Data Technologies*, vol. 1, pp. 199–206, SCITEPRESS, 2009.

[73] D. P. Singh, I. Joshi, and J. Choudhary, “Survey of GPU based sorting algorithms,” *International Journal of Parallel Programming*, vol. 46, pp. 1017–1034, 2018.

[74] T. B. Chandra, V. Patle, and S. Kumar, “New horizon of energy efficiency in sorting algorithms: green computing,” in *Proceedings of National Conference on Recent Trends in Green Computing. School of Studies in Computer in Computer Science & IT, Pt. Ravishankar Shukla University, Raipur, India*, pp. 24–26, 2013.

[75] N. Schmitt, S. Kamthania, N. Rawtani, L. Mendoza, K.-D. Lange, and S. Kounev, “Energy-Efficiency Comparison of Common Sorting Algorithms,” in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, 2021.

[76] B. Schmidt, F. Kallenborn, A. Chacon, and C. Hundt, “CUDASW++ 4.0: ultra-fast GPU-based Smith–Waterman protein sequence database search,” *BMC bioinformatics*, vol. 25, no. 1, p. 342, 2024.

[77] A. Roca Serrano, “Enhancing data center performance with GPU-accelerated dynamic programming algorithms,” bachelor’s thesis, Universitat Autònoma de Barcelona, 2024.

[78] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu, “Benchmarking and dissecting the nvidia hopper gpu architecture,” in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 656–667, IEEE, 2024.

[79] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, “Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 127–135, IEEE, 2019.

[80] NVIDIA, “NVIDIA Tesla P100.” https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016. Accessed: 2024-07-31.

[81] M. Doblas, O. Lostes-Cazorla, Q. Aguado-Puig, N. Cebry, P. Fontova-Musté, C. F. Batten, S. Marco-Sola, and M. Moretó, “GMX: Instruction Set Extensions for Fast, Scalable, and Efficient Genome Sequence Alignment,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1466–1480, 2023.

[82] Y. Gu, A. Subramaniyan, T. Dunn, A. Khadem, K.-Y. Chen, S. Paul, M. Vasimuddin, S. Misra, D. Blaauw, S. Narayanasamy, and R. Das, "GenDP: A Framework of Dynamic Programming Acceleration for Genome Sequencing Analysis," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[83] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: a genome sequencing accelerator," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 69–82, IEEE Press, 2018.

[84] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A Genomics Co-processor Provides up to 15, 000X Acceleration on Long Read Assembly," *ACM SIGPLAN Notices*, vol. 53, p. 199–213, Mar. 2018.

[85] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 161–168, IEEE, 2012.

[86] E. B. Fernandez, J. Villarreal, S. Lonardi, and W. A. Najjar, "FHAST: FPGA-based acceleration of Bowtie in hardware," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 12, no. 5, pp. 973–981, 2015.

[87] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. Kim, N. M. Ghiasi, G. Singh, J. Gómez-Luna, N. A. Alserr, M. Alser, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "SeGraM ; a universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ACM, 2022.

[88] A. Nag, C. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 334–346, ACM, 2019.

[89] F. Chen, L. Song, H. Li, and Y. Chen, "PARC: A Processing-in-CAM Architecture for Genomic Long Read Pairwise Alignment using ReRAM," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, p. 175–180, IEEE, Jan. 2020.

[90] H. Mao, M. Alser, M. Sadrosadati, C. Firtina, A. Baranwal, D. S. Cali, A. Manglik, N. A. Alserr, and O. Mutlu, "Genpip: In-memory acceleration of genome analysis via tight integration of basecalling and read mapping," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 710–726, IEEE, 2022.

[91] S. Diab, A. Nassereldine, M. Alser, J. Gómez Luna, O. Mutlu, and I. El Hajj, "A framework for high-throughput sequence alignment using real processing-in-memory systems," *Bioinformatics*, vol. 39, Mar. 2023.

[92] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, “Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system,” *IEEE Access*, vol. 10, pp. 52565–52608, 2022.

[93] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, “Meet the Walkers,” *PROC of the 46th MICRO*, pp. 1–12, 2013.

[94] F. Devaux, “The true processing in memory accelerator,” in *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–24, IEEE Computer Society, 2019.

[95] S. Kim, M. Fayazi, A. Daftardar, K.-Y. Chen, J. Tan, S. Pal, T. Ajayi, Y. Xiong, T. Mudge, C. Chakrabarti, D. Blaauw, R. Dreslinski, and H.-S. Kim, “Versa: A 36-Core Systolic Multiprocessor With Dynamically Reconfigurable Interconnect and Memory,” *IEEE Journal of Solid-State Circuits*, vol. 57, no. 4, pp. 986–998, 2022.

[96] K.-M. Chao, W. R. Pearson, and W. Miller, “Aligning two sequences within a specified diagonal band,” *Bioinformatics*, vol. 8, no. 5, p. 481–487, 1992.

[97] H. Suzuki and M. Kasahara, “Introducing difference recurrence relations for faster semi-global alignment of long sequences,” *BMC Bioinformatics*, vol. 19, no. 1, pp. 33–47, 2018.

[98] T. Yoshida, “Fujitsu High Performance CPU for the Post-K Computer,” tech. rep., Fujitsu, 2018.

[99] R. Okazaki, T. Tabata, S. Sakashita, K. Kitamura, N. Takagi, H. Sakata, T. Ishibashi, T. Nakamura, and Y. Ajima, “Supercomputer fugaku CPU A64FX realizing high performance, high-density packaging, and low power consumption,” tech. rep., Fujitsu, 2020.

[100] “A64FX microarchitecture manual.” https://github.com/fujitsu/A64FX/tree/master/doc, 2020. (accessed 30 January 2023).

[101] “Amazon EC2 C7g instances.” https://aws.amazon.com/es/ec2/instance-types/c7g. Accessed: 30 January 2023.

[102] “AWS Graviton technical guide.” https://github.com/aws/aws-graviton-getting-started. Accessed: 30 January 2023.

[103] “Intel Xeon Platinum 8160 Processor.” https://www.intel.com/content/www/us/en/products/sku/120501/intel-xeon-platinum-8160-processor-33m-cache-2-10-ghz/specifications.html. Accessed: 30 January 2023.

[104] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, “Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake,” *IEEE Micro*, vol. 37, p. 52–62, Mar. 2017.

[105] “AMD EPYC 7002 series processors.” https://www.amd.com/en/processors/epyc-7002-series. Accessed: 30 January 2023.

[106] "High Performance Computing: Tuning Guide for AMD EPYC 7002 Series Processors." https://developer.amd.com/wp-content/resources/56827-1-0.pdf, 2020. (accessed 30 January 2023).

[107] D. Suggs, M. Subramony, and D. Bouvier, "The AMD "Zen 2" Processor," *IEEE Micro*, vol. 40, p. 45–52, Mar. 2020.

[108] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, pp. 34–46, Mar 2016.

[109] H. Sadasivan, M. Maric, E. Dawson, V. Iyer, J. Israeli, and S. Narayanasamy, "Accelerating Minimap2 for accurate long read alignment on GPUs," *bioRxiv*, 2022.

[110] K.-E. Berger and F. Galea, "An efficient parallelization strategy for dynamic programming on gpu," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pp. 1797–1806, IEEE, 2013.

[111] P. Steffen, R. Giegerich, and M. Giraud, "GPU parallelization of algebraic dynamic programming," in *Parallel Processing and Applied Mathematics: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009, Revised Selected Papers, Part II 8*, pp. 290–299, Springer, 2010.

[112] V. Boyer, D. El Baz, and M. Elkihel, "Dense dynamic programming on multi GPU," in *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 545–551, IEEE, 2011.

[113] "TOP500 The List." https://www.top500.org/. Accessed: 2021-07-12.

[114] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011. Source: ACM SIGARCH Computer Architecture News ; volume 39, issue 2, page 1-7 ; ISSN 0163-5964.

[115] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 Simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020.

[116] "Genome Reference Consortium Human Reference 37 - Download link." http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/latest/hg19.fa.gz. Accessed: 2025-10-21.

[117] “Genome Reference Consortium Human Reference 38 - Download link.” http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/latest/hg38.fa.gz. Accessed: 2025-10-21.

[118] NCBI, “Human Genome T2T.” https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_009914755.1/, 2022. Accessed: 2024-02-28.

[119] “Human Genome T2T - Download link.” https://api.ncbi.nlm.nih.gov/datasets/v2/genome/download?filename=ncbi_dataset.zip&ncbi_phid=undefined. Accessed: 2025-10-21.

[120] “Mason simulator.” http://www.seqan.de/apps/mason/. Accessed: 2019-07-23.

[121] Y. Ono, K. Asai, and M. Hamada, “PBSIM: PacBio reads simulator—toward accurate genome assembly,” *Bioinformatics*, vol. 29, no. 1, pp. 119–121, 2013.

[122] D. Kim, J. M. Paggi, C. Park, C. Bennett, and S. L. Salzberg, “Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype,” *Nature biotechnology*, vol. 37, no. 8, pp. 907–915, 2019.

[123] “NVBIO.” http://nvlabs.github.io/nvbio/. Accessed: 2019-08-08.

[124] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, H.-F. Ting, S.-M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T.-W. Lam, “SOAP3-dp: Fast, Accurate and Sensitive GPU-Based Short Read Aligner,” *PLOS ONE*, vol. 8, pp. 1–11, 05 2013.

[125] Y. Liu and B. Schmidt, “CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing,” *IEEE Design & Test*, vol. 31, no. 1, pp. 31–39, 2013.

[126] S. Williams, A. Waterman, and D. Patterson, “Roofline: An Insightful Visual Performance Model for Multicore Architectures,” *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[127] F. Sanger, S. Nicklen, and A. R. Coulson, “DNA sequencing with chain-terminating inhibitors,” *Proceedings of the National Academy of Sciences*, vol. 74, no. 12, pp. 5463–5467, 1977.

[128] “Homo sapiens OCI-LY7.” https://www.encodeproject.org/experiments/ENCSR740DKM/. Accessed: 2019-07-23.

[129] “Homo sapiens OCI-LY7.” https://www.encodeproject.org/experiments/ENCSR001HHK/. Accessed: 2019-07-23.

[130] “Homo sapiens A375.” https://www.encodeproject.org/experiments/ENCSR504VXC/. Accessed: 2019-07-23.

[131] P.-V. Khuong and P. Morin, “Array Layouts for Comparison-Based Searching,” *J. Exp. Algorithmics*, vol. 22, pp. 1.3:1–1.3:39, May 2017.

[132] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI99, p. 1–12, ACM, May 1999.

[133] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, p. 26–39, Mar. 2017.

[134] A. Armejach, H. Caminal, J. M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Stencil codes on a vector length agnostic architecture," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pp. 1–12, 2018.

[135] A. Armejach, H. Caminal, J. M. Cebrian, R. Langarita, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó, "Using Arm's scalable vector extension on stencil codes," *The Journal of Supercomputing*, vol. 76, no. 3, pp. 2039–2062, 2020.

[136] C. Staelin, "lmbench: an extensible micro-benchmark suite," *Software: Practice and Experience*, vol. 35, no. 11, p. 1079–1105, 2005.

[137] W. Muła, N. Kurz, and D. Lemire, "Faster population counts using AVX2 instructions," *The Computer Journal*, vol. 61, no. 1, pp. 111–120, 2018.

[138] T. Odajima, Y. Kodama, M. Tsuji, M. Matsuda, Y. Maruyama, and M. Sato, "Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 523–530, 2020.

[139] "SRX020470." https://ddbj.nig.ac.jp/DRASearch/experiment?acc=SRX020470. Accessed: 2020-12-03.

[140] "SRX207170." https://trace.ddbj.nig.ac.jp/DRASearch/experiment?acc=SRX207170. Accessed: 2020-12-03.

[141] "SRX206890." https://trace.ddbj.nig.ac.jp/DRASearch/experiment?acc=SRX206890. Accessed: 2020-12-03.

[142] "CPU NVIDIA Grace." https://www.nvidia.com/es-es/data-center/grace-cpu. Accessed: 7 January 2023.

[143] A. Subramaniyan, Y. Gu, T. Dunn, S. Paul, M. Vasimuddin, S. Misra, D. Blaauw, S. Narayanasamy, and R. Das, "GenomicsBench: A Benchmark Suite for Genomics," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, p. 1–12, IEEE, Mar. 2021.

[144] G. Myers, "A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming," *J. ACM*, vol. 46, pp. 395–415, 05 1999.

[145] S. Kalikar, C. Jain, M. Vasimuddin, and S. Misra, "Accelerating minimap2 for long-read sequencing applications on modern CPUs," *Nature Computational Science*, vol. 2, p. 78–83, Feb. 2022.

[146] N. J. Loman, J. Quick, and J. T. Simpson, "A complete bacterial genome assembled de novo using only nanopore sequencing data," *Nature Methods*, vol. 12, p. 733–735, June 2015.

[147] H. Gamaarachchi, C. W. Lam, G. Jayatilaka, H. Samarakoon, J. T. Simpson, M. A. Smith, and S. Parameswaran, "GPU accelerated adaptive banded event alignment for rapid comparative nanopore signal analysis," *BMC Bioinformatics*, vol. 21, Aug. 2020.

[148] "Nanopore wgs consortium." https://github.com/nanopore-wgs-consortium/NA12878. Accessed: 7 January 2023.

[149] M. Rautiainen and T. Marschall, "GraphAligner: rapid and versatile sequence-to-graph alignment," *Genome Biology*, vol. 21, Sept. 2020.

[150] A. Ahmadi, A. Behm, N. Honnalli, C. Li, L. Weng, and X. Xie, "Hobbes: optimized gram-based methods for efficient read alignment," *Nucleic Acids Research*, vol. 40, p. e41–e41, Dec. 2011.

[151] "NCBI sequence read archive." https://www.ncbi.nlm.nih.gov/sra. Accessed: 7 January 2023.

[152] "Pacific Biosciences Dataset: Caenorhabditis elegans 40x coverage." http://datasets.pacb.com.s3.amazonaws.com/2014/c_elegans/list.html. Accessed: 7 January 2023.

[153] M. A. Eberle, E. Fritzilas, P. Krusche, M. Källberg, B. L. Moore, M. A. Bekritsky, Z. Iqbal, H.-Y. Chuang, S. J. Humphray, A. L. Halpern, S. Kruglyak, E. H. Margulies, G. McVean, and D. R. Bentley, "A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree," *Genome Research*, vol. 27, p. 157–164, Nov. 2016.

[154] M. Burrows and D. J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," Tech. Rep. 124, Digital Equipment Corporation, 1994.

[155] "libcuckoo." https://github.com/efficient/libcuckoo. Accessed: 7 January 2023.

[156] "Loman Labs: Escherichia coli." https://zenodo.org/record/1172816/files/Loman_E.coli_MAP006-1_2D_50x.fasta. Accessed: 7 January 2023.

[157] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[158] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever,

K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” 2016.

[159] J. M. Zook, D. Catoe, J. McDaniel, L. Vang, N. Spies, A. Sidow, Z. Weng, Y. Liu, C. E. Mason, N. Alexander, E. Henaff, A. B. McIntyre, D. Chandramohan, F. Chen, E. Jaeger, A. Moshrefi, K. Pham, W. Stedman, T. Liang, M. Saghbini, Z. Dzakula, A. Hastie, H. Cao, G. Deikus, E. Schadt, R. Sebra, A. Bashir, R. M. Truty, C. C. Chang, N. Gulbahce, K. Zhao, S. Ghosh, F. Hyland, Y. Fu, M. Chaisson, C. Xiao, J. Trow, S. T. Sherry, A. W. Zaranek, M. Ball, J. Bobe, P. Estep, G. M. Church, P. Marks, S. Kyriazopoulou-Panagiotopoulou, G. X. Zheng, M. Schnall-Levin, H. S. Ordonez, P. A. Mudivarti, K. Giorda, Y. Sheng, K. B. Rypdal, and M. Salit, “Extensive sequencing of seven human genomes to characterize benchmark reference materials,” *Scientific Data*, vol. 3, June 2016.

[160] “Clair3 ONT model r941_prom_hac_g360+g422.” http://www.bio8.cs.hku.hk/clair3/clair3_models/r941_prom_hac_g360+g422.tar.gz. Accessed: 7 January 2023.

[161] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, “The Sequence Alignment/Map format and SAMtools,” *Bioinformatics*, vol. 25, p. 2078–2079, June 2009.

[162] C. Lee, C. Grasso, and M. F. Sharlow, “Multiple sequence alignment using partial order graphs,” *Bioinformatics*, vol. 18, p. 452–464, Mar. 2002.

[163] C. Lee, “Generating consensus sequences from partial order multiple sequence alignment graphs,” *Bioinformatics*, vol. 19, pp. 999–1008, May 2003.

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

[165] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, “Fast gap-affine pairwise alignment using the wavefront algorithm,” *Bioinformatics*, vol. 37, p. 456–463, Sept. 2020.

[166] “wfmash: a a pangenome-scale aligner.” https://github.com/waveygang/wfmash. Accessed: 30 January 2023.

[167] B. Song, S. Marco-Sola, M. Moreto, L. Johnson, E. S. Buckler, and M. C. Stitzer, “AnchorWave: Sensitive alignment of genomes with high sequence diversity, extensive structural polymorphism, and whole-genome duplication,” *Proceedings of the National Academy of Sciences*, vol. 119, Dec. 2021.

[168] L. Pipes and R. Nielsen, “AncestralClust: clustering of divergent nucleotide sequences by ancestral sequence reconstruction using phylogenetic trees,” *Bioinformatics*, vol. 38, p. 663–670, Oct. 2021.

[169] R. Langarita, A. Armejach, P. Ibáñez, J. Alastruey-Benedé, and M. Moretó, “Porting and Optimizing BWA-MEM2 Using the Fujitsu A64FX Processor,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 20, p. 3139–3153, Sept. 2023.

[170] "Pytorch cpu threading." https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html. Accessed: 7 January 2023.

[171] "Intel oneAPI math kernel library (MKL)." https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html. Accessed: 7 January 2023.

[172] D. Bruening, *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, 2004.

[173] "DynamoRIO." https://dynamorio.org. Accessed: 7 January 2023.

[174] "DynamoRIO instruction mix tool." https://github.com/LorienLV/dynamorio. Accessed: 7 January 2023.

[175] R. E. Grant, M. Levenhagen, S. L. Olivier, D. DeBonis, K. T. Pedretti, and J. H. Laros III, "Standardizing Power Monitoring and Control at Exascale," *Computer*, vol. 49, p. 38–46, Oct. 2016.

[176] "RAPL stopwatch." https://github.com/LorienLV/rapl_stopwatch. Accessed: 7 January 2023.

[177] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED'10, p. 189–194, ACM, Aug. 2010.

[178] R. Bellman and R. Kalaba, "On adaptive control processes," *IRE Transactions on Automatic Control*, vol. 4, no. 2, pp. 1–9, 1959.

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

[180] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[181] R. Stallman and R. McGrath, *GNU make: A program for directing recompilation : GNU make version 3.79.1.* Free Software Foundation, 2002.

[182] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen, "LAPACK: A portable linear algebra library for high-performance computers," in *Proceedings SUPERCOMPUTING'90*, pp. 2–11, IEEE Computer Society, 1990.

[183] C. A. R. Hoare, "Algorithm 64: quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, 1961.

[184] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," tech. rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[185] D. Habich, J. Pietrzyk, A. Krause, J. Hildebrandt, and W. Lehner, “To use or not to use the SIMD gather instruction?,” in *Proceedings of the 18th International Workshop on Data Management on New Hardware*, pp. 1–5, 2022.

[186] NVIDIA, “Boosting Dynamic Programming Performance Using NVIDIA Hopper GPU DPX Instructions.” https://developer.nvidia.com/blog/boosting-dynamic-programming-performance-using-nvidia-hopper-gpu-dpx-instructions/, 2022. Accessed: 2024-07-15.

[187] B. Plancher and S. Kuindersma, “A performance analysis of parallel differential dynamic programming on a gpu,” in *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*, pp. 656–672, Springer, 2020.

[188] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, and R. Narayanaswami, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, 2017.

[189] Y. Zhang, X. Liao, H. Jin, L. He, B. He, H. Liu, and L. Gu, “DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 371–384, 2021.

[190] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[191] D. Buttlar, J. Farrell, and B. Nichols, *Pthreads programming: A POSIX standard for better multiprocessing*. " O’Reilly Media, Inc.", 1996.

[192] precisionFDA, “PB HiFi/ONT.” https://precision.fda.gov/challenges/10, 2020. Accessed: 2024-02-28.

[193] P. DevNet, “PB CLR.” https://github.com/PacificBiosciences/DevNet/wiki/HG002-Structural-Variant-Analysis-with-CLR-data, 2019. Accessed: 2024-02-28.

[194] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala, J. Jalal, M. Werkheiser, and A. Kona, “The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc,” *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.

[195] Arm, “Arm Cortex-M35P.” https://developer.arm.com/Processors/Cortex-M35P, 2018. Accessed: 2024-03-07.

[196] F. Arnaud, A. Thean, M. Eller, M. Lipinski, Y. Teh, M. Ostermayr, K. Kang, N. Kim, K. Ohuchi, J.-P. Han, D. Nair, J. Lian, S. Uchimura, S. Kohler, S. Miyaki, P. Ferreira, J.-H. Park, M. Hamaguchi, K. Miyashita, R. Augur, Q. Zhang, K. Strahrenberg,

S. ElGhouli, J. Bonnouvrier, F. Matsuoka, R. Lindsay, J. Sudijono, F. Johnson, J. Ku, M. Sekine, A. Steegen, and R. Sampson, “Competitive and cost effective high-k based 28nm CMOS technology for low power applications,” in *2009 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2009.

[197] K.-L. Cheng, C. C. Wu, Y. P. Wang, D. W. Lin, C. M. Chu, Y. Y. Tarng, S. Y. Lu, S. J. Yang, M. H. Hsieh, C. M. Liu, S. P. Fu, J. H. Chen, C. T. Lin, W. Y. Lien, H. Y. Huang, P. W. Wang, H. H. Lin, D. Y. Lee, M. J. Huang, C. F. Nieh, L. T. Lin, C. C. Chen, W. Chang, Y. H. Chiu, M. Y. Wang, C. H. Yeh, F. C. Chen, C. M. Wu, Y. H. Chang, S. C. Wang, H. C. Hsieh, M. D. Lei, K. Goto, H. J. Tao, M. Cao, H. C. Tuan, C. H. Diaz, and Y. J. Mii, “A highly scaled, high performance 45 nm bulk logic CMOS technology with 0.242 μm 2 SRAM cell,” in *2007 IEEE International Electron Devices Meeting*, IEEE, 2007.

[198] D. C. Daly, L. C. Fujino, and K. C. Smith, “Through the looking glass-the 2018 edition: trends in solid-state circuits from the 65th ISSCC,” *IEEE Solid-State Circuits Magazine*, vol. 10, no. 1, pp. 30–46, 2018.

[199] S.-Y. Wu, C. Y. Lin, M. C. Chiang, J. J. Liaw, J. Y. Cheng, S. H. Yang, M. Liang, T. Miyashita, C. H. Tsai, B. C. Hsu, H. Y. Chen, T. Yamamoto, S. Y. Chang, V. S. Chang, C. H. Chang, J. H. Chen, H. F. Chen, K. C. Ting, Y. K. Wu, K. H. Pan, R. F. Tsui, C. H. Yao, P. R. Chang, H. M. Lien, T. L. Lee, H. M. Lee, W. Chang, T. Chang, R. Chen, M. Yeh, C. C. Chen, Y. H. Chiu, Y. H. Chen, H. C. Huang, Y. C. Lu, C. W. Chang, M. H. Tsai, C. C. Liu, K. S. Chen, C. C. Kuo, H. T. Lin, S. M. Jang, and Y. Ku, “A 16nm FinFET CMOS technology for mobile SoC and computing applications,” in *2013 IEEE International Electron Devices Meeting*, IEEE, 2013.

[200] S.-Y. Wu, C. Lin, M. Chiang, J. Liaw, J. Cheng, S. Yang, C. Tsai, P. Chen, T. Miyashita, C. Chang, V. Chang, K. Pan, J. Chen, Y. Mor, K. Lai, C. Liang, H. Chen, S. Chang, C. Lin, C. Hsieh, R. Tsui, C. Yao, C. Chen, R. Chen, C. Lee, H. Lin, C. Chang, K. Chen, M. Tsai, K. Chen, Y. Ku, and S. M. Jang, “A 7nm CMOS platform technology featuring 4 th generation FinFET transistors with a 0.027um 2 high density 6-T SRAM cell for mobile SoC applications,” in *2016 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2016.

[201] R. Xie, P. Montanini, K. Akarvardar, N. Tripathi, B. Haran, S. Johnson, T. Hook, B. Hamieh, D. Corliss, J. Wang, X. Miao, J. Sporre, J. Fronheiser, N. Loubet, M. Sung, S. Sieg, S. Mochizuki, C. Prindle, S. Seo, A. Greene, J. Shearer, A. Labonte, S. Fan, L. Liebmann, R. Chao, A. Arceo, K. Chung, K. Cheon, P. Adusumilli, H. Amanapu, Z. Bi, J. Cha, H.-C. Chen, R. Conti, R. Galatage, O. Gluschenkov, V. Kamineni, K. Kim, C. Lee, F. Lie, Z. Liu, S. Mehta, E. Miller, H. Niimi, C. Niu, C. Park, D. Park, M. Raymond, B. Sahu, and M. Sankarapandian, “A 7nm FinFET technology featuring EUV patterning and dual strained high mobility channels,” in *2016 IEEE International Electron Devices Meeting (IEDM)*, IEEE, 2016.

[202] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, pp. 469–480, 2009.

[203] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *2015 IEEE 21st International symposium on high performance computer architecture (HPCA)*, pp. 577–589, IEEE, 2015.

[204] S. Kalikar, C. Jain, V. Md, and S. Misra, "Accelerating long-read analysis on modern CPUs," *bioRxiv*, July 2021.

[205] A. Zeni, S. Onken, M. D. Santambrogio, and M. Samadi, "Leveraging Difference Recurrence Relations for High-Performance GPU Genome Alignment," in *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, pp. 133–143, 2024.

[206] "What is upmem?." https://github.com/BSC-PIM/UPmem-example/wiki/What-is-UPmem%3F. Accessed: 2025-10-29.

[207] "NVIDIA H100 Specifications." https://www.techpowerup.com/gpu-specs/h100-pcie-80-gb.c3899. Accessed: 2025-10-29.

[208] "Intel Xeon Platinum 8358 Specifications." https://www.intel.com/content/www/us/en/products/sku/212282/intel-xeon-platinum-8358-processor-48m-cache-2-60-ghz/specifications.html. Accessed: 2025-10-29.

[209] R. Langarita, J. Alastruey-Benedé, P. Ibáñez-Marín, S. Marco-Sola, M. Moretó, and A. Armejach, "Squire: A General-Purpose Accelerator to Exploit Fine-Grain Parallelism on Dependency-Bound Kernels," 2025.

[210] S. Pal, S. Feng, D.-h. Park, S. Kim, A. Amarnath, C.-S. Yang, X. He, J. Beaumont, K. May, Y. Xiong, K. Kaszyk, J. M. Morton, J. Sun, M. O'Boyle, M. Cole, C. Chakrabarti, D. Blaauw, H.-S. Kim, T. Mudge, and R. Dreslinski, "Transmuter: Bridging the Efficiency Gap using Memory and Dataflow Reconfiguration," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, (New York, NY, USA), p. 175–190, Association for Computing Machinery, 2020.

[211] ARM Limited, "big.LITTLE Technology: The Future of Mobile," tech. rep., ARM Holdings, 2013. White Paper.

[212] "BWA-MEM2 source code repository." https://github.com/bwa-mem2/bwa-mem2. Accessed: 2025-05-18.

[213] "Bioconda source code repository." https://github.com/bioconda/bioconda-recipes. Accessed: 2025-05-18.

[214] "Minimap2 source code repository." https://github.com/lh3/minimap2. Accessed: 2025-05-18.

[215] M. Alser, Z. Bingöl, D. S. Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating genome analysis: A primer on an ongoing journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, 2020.

[216] M. Doblas, O. Lostes-Cazorla, Q. Aguado-Puig, C. Iñiguez, M. Moreto, and S. Marco-Sola, “QuickEd: high-performance exact sequence alignment based on bound-and-align,” *Bioinformatics*, vol. 41, no. 3, p. btaf112, 2025.

[217] C. Kim, K. Koh, T. Kim, D. Han, and J. Seo, “BWA-MEM-SCALE: Accelerating genome sequence mapping on commodity servers,” in *Proceedings of the 51st International Conference on Parallel Processing*, pp. 1–12, 2022.

[218] Y. Huang, L. Kong, D. Chen, Z. Chen, X. Kong, J. Zhu, K. Mamouras, S. Wei, K. Yang, and L. Liu, “Casa: An energy-efficient and high-speed cam-based smem seeding accelerator for genome alignment,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1423–1436, 2023.

[219] K. Koliogeorgi, *Hardware Acceleration Techniques for Computation and Data Intensive Machine Learning and Bioinformatic Applications*. PhD thesis, National Technical University of Athens, 2023.

[220] R. Shi, G. Schieffer, M. Gokhale, P.-H. Lin, H. Patel, and I. Peng, “ARM SVE Unleashed: Performance and Insights Across HPC Applications on Nvidia Grace,” in *European Conference on Parallel Processing*, pp. 33–47, Springer, 2025.

[221] J. Zhao, X. Zhao, J. Pierre-Both, and K. T. Konstantinidis, “Bindash 2.0: new MinHash scheme allows ultra-fast and accurate genome search and comparisons,” *bioRxiv*, pp. 2024–03, 2024.

[222] G. Accordi, J. Domke, T. Pollinger, D. Gadioli, and G. Palermo, “Towards High-Performance and Portable Molecular Docking on CPUs through Vectorization,” *arXiv preprint arXiv:2509.12232*, 2025.

[223] X. Kong, C. Shen, and J. Tang, “A Review of Biosequences Alignment, Matching, and Mining Based on GPU,” *Current Bioinformatics*, 2025.

[224] H. Sadasivan, A. Klauser, J. Hench, Y. Turakhia, G. Singh, A. Zeni, S. Beecroft, S. Narayanasamy, J. Nivala, B. Robey, O. Mutlu, K. Denolf, and S. Sitaraman, “The Genomic Computing Revolution: Defining the Next Decades of Accelerating Genomics,” in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, p. 1–9, IEEE, Sept. 2024.