QUETZAL: Vector Acceleration Framework for Modern Genome Sequence Analysis Algorithms

Julián Pavón*†, Ivan Vargas Valdivieso*†, Carlos Rojas*†, Cesar Hernandez*, Mehmet Aslan§, Roger Figueras*, Yichao Yuan¶, Joël Lindegger‡, Mohammed Alser‡, Francesc Moll†, Santiago Marco-Sola†, Oguz Ergin§
Nishil Talati¶, Onur Mutlu§, Osman Unsal*, Mateo Valero*† and Adrian Cristal*
*Barcelona Supercomputing Center †Universitat Politècnica de Catalunya ‡ETH Zurich
§TOBB ETÜ University of Economics & Technology ¶University of Michigan
Email: julian.pavon@bsc.es

***Abstract*—Genome sequence analysis is fundamental to medical breakthroughs such as developing vaccines, enabling genome editing, and facilitating personalized medicine. The exponentially expanding sequencing datasets and complexity of sequencing algorithms necessitate performance enhancements. While the performance of software solutions is constrained by their underlying hardware platforms, the utility of fixed-function accelerators is restricted to only certain sequencing algorithms.**

This paper presents `QUETZAL`, the first general-purpose vector acceleration framework designed for high efficiency and broad applicability across a diverse set of genomics algorithms. While a commercial CPU's vector datapath is a promising candidate to exploit the data-level parallelism in genomics algorithms, our analysis finds that its performance is often limited due to long-latency scatter/gather memory instructions. `QUETZAL` introduces a hardware-software co-design comprising an accelerator microarchitecture closely integrated with the CPU's vector datapath, alongside novel vector instructions to fully capitalize on the proposed hardware. `QUETZAL` integrates a set of scratchpad-style buffers meticulously designed to minimize latency associated with scatter/gather instructions during the retrieval of input genome sequences data. `QUETZAL` supports both short and long reads, and different types of sequencing data formats. A combination of hardware and software techniques enables `QUETZAL` to reduce the latency of memory instructions, perform complex computation using a single instruction, and transform data representations at runtime, resulting in overall efficiency gain. `QUETZAL` significantly accelerates a vectorized CPU baseline on modern genome sequence analysis algorithms by 5.7×, while incurring a small area overhead of 1.4% post place-and-route at the 7nm technology node compared to an HPC ARM CPU.

I. Introduction

The diminishing cost and improving efficiency of modern genome sequencing technologies, in conjunction with the elucidation of comprehensive genome sequences for both humans and various other species, such as microbes, have inaugurated an era marked by an exponentially expanding array of novel applications and scientific discoveries. These include, for example, personalized medicine [1–7], evolutionary genetics [8–10] and forensics [11–13]. One of the fundamental computational steps in these applications is *genome sequence analysis*, where genome sequences are compared to each other to infer important genetic information. This information includes 1) the number and locations of genomic variations in DNA and RNA sequences for identifying disease causes [14], 2) functional regions that are conserved across evolution among different protein sequences for designing personalized therapeutic treatments [15], and 3) similarity approximation between many sequences for finding highly similar sequences or obtaining phylogenetic trees [16].

Two primary categories of algorithms typically employed in the analysis of genome sequences are: 1) *Sequence alignment* and 2) *edit distance approximation*. The sequence alignment problem is formulated as Approximate String Matching (ASM) [17]. ASM is often solved using Dynamic Programming (DP) algorithms, such as Smith-Waterman-Gotoh (`SWG`) and Needleman-Wunsch (`NW`). DP-based algorithms are computationally expensive with quadratic time and space complexities in sequence length. Edit distance approximation algorithms, such as SneakySnake (`SS`) [18] and Shouji [19], approximate the edit distance (number of edits needed to convert one sequence into the other [20]) that is always less than or equal to a user-defined threshold.

Analyzing the ever-increasing volumes of sequencing data [21] poses significant computational challenges, motivating a large body of research focusing on optimizing genome sequence analysis applications. These include 1) software/algorithmic optimizations and 2) hardware optimizations for both sequence alignment [22–46] and edit distance approximation [18, 19, 36, 47–54].

The majority of software/algorithm proposals fit into one of two categories. First, multiple heuristic algorithms have been proposed to improve the quadratic execution time of DP algorithms by pruning the number of operations required by these algorithms [29, 55–58]. However, these approaches do not ensure an optimal solution, jeopardizing the results of genome sequence analysis. Second, a number of works aim to efficiently rearrange the DP computation while ensuring an optimal solution. Recently among them is a family of promising novel algorithms, called Wavefront Alignment [32, 34]. The implementation of these improved algorithms brings new challenges, such as pointer chasing operations, which reduce the memory-level parallelism in general purpose architectures, thus requiring better and more sophisticated hardware to fully exploit the benefits of these algorithms.

In contrast, existing hardware approaches are subject to three primary limitations. First, some proposals prescribe a limiting hardware architecture tailored to a singular sequencing algorithm [47, 59]. Second, some proposals target only one genome data type (*i.e.,* DNA/RNA) [44] that lack generality. Third, some

approaches exhibit the constraint of exclusively processing short input sequences [44, 59]. Given the ongoing emergence of novel algorithms, such as WFA [32, 33] and BiWFA [34, 35], alongside the introduction of new sequencing data types, exemplified by HiFi from PacBio [60, 61] and Duplex from ONT [62], characterized by longer and more accurate sequences, there exists a need for software developers and hardware architects to devise adaptable, flexible, and scalable systems. These systems should facilitate the flexible integration of new sequence analysis algorithms and data without necessitating the reconstruction of the accelerator. We pose the following question: *how can we design an architecture that not only provides improved efficiency and performance for genome sequence analysis, but is also programmable to accommodate a broad spectrum of emerging genomics algorithms and data?*

The motivation for enhanced performance with programmability finds robust validation within the industrial context as well. NVIDIA's recent integration of Dynamic Programming X (DPX) instructions [63] stands out as a noteworthy instance aimed at bolstering GPU efficiency across various domains such as genomics, proteomics, and robot path planning. This adoption embodies a design philosophy where developers can accelerate existing algorithms and innovate new ones by leveraging DPX instructions, which are ISA extensions to optimize various application domains, thereby capitalizing on the same hardware changes across many domains. Such design philosophy has similarly been used to accelerate workloads in other application domains on GPUs (e.g., tensor programs), using for example tensor cores and tensor memory access units [64]. This paper follows a similar approach to offer a high-performance, programmable solution for a wide range of genome sequence analysis workloads.

Modern ASM algorithms, such as WFA and BiWFA, exhibit notable levels of Data Level Parallelism (DLP), as large amounts of data undergo identical operations. The use of vector architectures in modern CPUs, characterized by Single Instruction Multiple Data (SIMD) execution [65], is a well-known technique for harnessing the available DLP in applications. Vector architectures offer a versatile framework, making them adaptable to a wide range of ASM algorithms. However, these algorithms pose new challenges to efficient execution on vector architectures. Specifically, these algorithms employ memory-indexed instructions, *e.g.,* scatter/gather, which are costly and entail considerable latency for retrieving and storing intermediate data [32, 36].

This paper presents *QUETZAL*, the *first universal* vector acceleration framework for modern ASM algorithms using hardware-software co-design. The primary design goal of QUETZAL is to strike a balance between efficiency and versatility. The proposed design supports both short and long sequences, and different data encoding schemes to efficiently accommodate various alphabets of RNA, DNA, and protein data. In particular, QUETZAL combines a vector accelerator architecture tightly coupled with the general-purpose CPU's pipeline, and supports novel vector instructions to fully unlock the potential of the proposed hardware. QUETZAL accelerator features a pair of scratchpad-style *buffers* connected to the Vector Processing Unit (VPU) in the core's datapath. QUETZAL instructions use these buffers to replace the memory-indexed instructions with direct data requests to the VPU.

We demonstrate the effectiveness of QUETZAL using a full-system cycle-accurate simulator and different use cases of approximate string matching in genome sequence analysis (both sequence alignment and edit distance approximation). Specifically, we use the state-of-the-art ASM algorithms (WFA, BiWFA, and SS) for both short and long sequences for evaluation. Our results show that QUETZAL outperforms baseline vectorized CPU implementations by 6.1×, and 5.2×, on average, for read alignment, and edit distance approximation, respectively. We implement QUETZAL in RTL using a 7nm technology node; the place-and-route results reveal that our design consumes a small $746\mu W$ power and $0.097mm^2$ area (a small overhead of 1.4% compared to a Fujitsu A64FX processor).

The key contributions of this paper are as follows:

- A detailed analysis of challenges involved in accelerating genome sequence analysis algorithms using a CPU vector datapath.
- Design of a cost-effective vector accelerator architecture tightly coupled with a commodity CPU's vector datapath.
- Introduction of vector instructions for unleashing the complete potential of our vector accelerator design.
- *QUETZAL:* a programmable vector framework to accelerate a wide range of genome sequence analysis algorithms that offers an average speedup of 5.7× compared to a baseline CPU architecture with a small 1.4% area overhead.

II. Background and Motivation

This section provides an overview of classical and modern genome sequence analysis algorithms. Subsequently, we analyze the shortcomings of modern genome sequence analysis algorithms to fully exploit the performance and energy benefits of vector architectures.

A. Classical DP sequence alignment algorithms

Sequence alignment is a fundamental method used in genomics to compare and identify similarities between two biological sequences, such as DNA, RNA, or protein sequences. The goal is to assess the similarity between two sequences by introducing the minimum necessary gaps (insertions and deletions) and mismatches to match (or align) one sequence with the other. Fig. 1.a shows an example of the Needleman-Wunsch (NW) table [22] (a classic and extensively employed DP approach) on the sequence pair <ACAG, AAGT>.

In this example, each entry represents the number of edits required to align the prefixes of the two strings up to the current row and column. A new entry (marked in blue in Fig. 1.a) is computed through simple arithmetic from their west, north-west, and north neighbors (marked in green and red in Fig. 1.a). Traceback determines the optimal alignment by re-tracing the origin of the value in the southeast corner. For example, the origin of the blue-marked cell is marked in green in Fig. 1.a, and traceback would record it as an insertion. From there, traceback would determine the origin of the green-marked cell, and so on, until it reaches the north-west corner. The Needleman-Wunsch algorithm has been the cornerstone for many subsequent algorithms and tools [22, 46]. The Needleman-Wunsch algorithm provides an *optimal* solution as it calculates the complete DP table regardless of the sequence length.

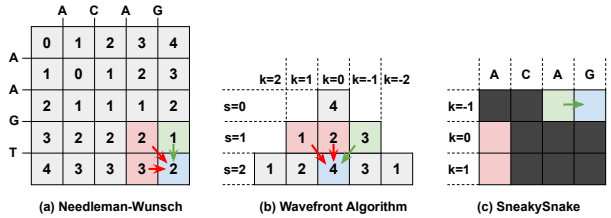

Fig. 1. **Examples for the Needleman-Wunsch [22], Wavefront Algorithm [32], and SneakySnake [18] algorithms for the sequence pair <ACAG,AAGT>.**

Banded alignment: Several software proposals aiming to reduce the $O(n^2)$ runtime complexity of classic DP algorithms have been published [55, 56]. A well-known technique to speed up the alignment process is referred to in the state-of-the-art as *banded alignment* [58, 66]. In traditional DP, the alignment is obtained after computing the value of all cells in the table. With the heuristic banded alignment optimisation, only cells on the main diagonal and close to this diagonal are evaluated. However, if the alignment between two sequences does not fall within the selected band, the algorithm will fail to identify the optimal alignment and the solution will not be *optimal*.

B. Modern DP sequence alignment algorithms

To reduce the complexity of classic DP-based approaches, modern algorithms take advantage of similarities between the input sequences to safely avoid computing large regions of the DP table. When executing the traceback step, the elements in the computed region belong to the *optimal* solution (i.e., the same result as computing the complete DP table using `NW`). As a result, modern algorithms efficiently reduce the $O(n^2)$ complexity from classic ones, while providing the *optimal* alignment. For example, Wavefront Alignment (`WFA`) [32] and Bidirectional Wavefront Alignment (`BiWFA`) [34] are two recently proposed DP algorithms that run in $O(n * s)$ time, where n is the sequence length and s the error (or score) between the sequences. In contrast, traditional DP algorithms like `SWG` and `NW` have $O(n^2)$ execution time. Since, most of the time, the error is much smaller than the sequence length, `WFA` can generate alignments highly efficiently.

Fig. 1.b shows an example of `WFA`'s computation on the sequence pair <ACAG, AAGT>. The key idea of `WFA` is to compute how far the string pair can be aligned on a given diagonal k with at most s edits. `WFA` starts with only the main diagonal (k=0) and no edits (s=0), and gradually builds the DP table shown in Fig. 1.b from the top down. Entries are computed from their north-west, north, and north-east neighbors (marked in green and red). Similar to `NW`, traceback determines the optimal alignment by re-tracing the origin of the value in the south-center cell. For example, the origin of the blue-marked cell is marked in green, and traceback would record it as an insertion.

C. Edit distance approximation

Edit distance approximation is a method used to quickly approximate the similarity between two sequences. In contrast to conventional sequence alignment algorithms, edit distance approximation does not guarantee an optimal solution. Instead, such methods often guarantee a lower bound on the edit distance.

SneakySnake (`SS`) [18] is a recent such *edit distance approximation* algorithm. Fig. 1.c shows an example of SneakySnake's computation on the sequence pair <ACAG, AAGT>. The key idea of SneakySnake is to (1) build a Boolean table where each cell indicates a single character match (drawn in color or red) or mismatch (drawn in black), and then (2) to greedily follow a series of maximal exact matches (*i.e.,* runs of red and colored cells) between the two strings. For example, SneakySnake reaches the blue cell by following a match from the green cell. Matches are laid out along rows (indexed by k) in SneakySnake's table, *i.e.,* each row corresponds to a column in the `WFA` table (Fig. 1.b), and a diagonal in the `NW` table (Fig. 1.a).

D. Rationale for flexible domain specific accelerators

ASIC-based domain specific accelerators (e.g., [36, 38, 67, 68]) achieve better performance and energy efficiency compared to general-purpose architectures such as CPUs. However, they incur the high cost of developing custom silicon for a specific application, often implementing a certain algorithm directly in hardware. On the other hand, new algorithms for genome sequence analysis and for new sequencing technologies have recently been developed. Therefore, fixed-function accelerators cannot keep up with the growing complexities and demands of modern genome sequence analysis algorithms. For example, Smith-Waterman (SW), the classic ASM algorithm, was optimized from its original version [24] to a banded SW [58], and further to an adaptive banded SW [66]. Alser *et al.* [46] systematically surveyed 107 genome sequence analysis tools (such as minimap2 [69]) and read alignment algorithms since 1988 to 2020. We make three observations from this analysis. (1) New tools, variations of classic algorithms and new DP-based algorithms are published every year aiming to improve the performance of genome sequence analysis [46]. (2) Different sequencing technologies create a wider set of requirements for genome sequence analysis tools, such as working with longer sequences (e.g., PacBio HiFi [60]), dealing with different alphabets (e.g., DNA/RNA and proteins), configurable scoring functions, among others [46, 47]. (3) Recently, emerging tools are incorporating multiple algorithms for the read alignment stage in the genome sequence analysis pipeline, thus requiring hardware capable of switching between and/or combining multiple algorithms at run time [46]. This paper focuses on the need for a flexible and programmable hardware acceleration framework to accommodate the expanding array of genome analysis algorithms.

E. Flexible state-of-the-art platforms

General-purpose CPUs and GPUs provide a flexible framework for genome sequence analysis algorithms. There is a large body of prior work on accelerating genome sequence analysis algorithms using the SIMD and vector support in CPUs (e.g., [30, 52, 70, 71]). However, modern genome sequence analysis algorithms feature non-unit stride memory access patterns which limits the performance benefits of SIMD and vector architectures (we analyze this in detail in Section II-G). On the other hand, the massive parallelism offered by GPUs makes them an attractive hardware platform to accelerate genome sequence analysis algorithms. Multiple GPU-based approaches have been published recently (e.g., [38, 72–75]) showing considerable performance benefits compared to CPUs

when processing short sequences. However, the performance of GPUs does not scale well for long sequences. Active working set (such as the DP table) size increases considerably for long sequences, exceeding the available on-chip memory. This increasing memory footprint constraints the number of GPU workers allocated to process the input sequences [76, 77], thereby reducing the parallelism offered by GPUs for long sequences. As long-read sequencing technologies [7] become increasingly affordable, featuring high throughput and high accuracy, coupled with the growing accessibility of whole-genome datasets, the efficient analysis of long genome sequences is becoming increasingly important in the realm of bioinformatics [7, 46, 47, 78, 79]. Consequently, the development of a versatile system capable of effectively accelerating long-read sequences is paramount.

In this work, we aim to accelerate modern genome sequence analysis algorithms using general-purpose CPUs and vector support in CPUs. This choice has two main reasons: (1) As discussed in Section II-G, the performance of vectorized approaches is constrained by inefficient vector memory instructions and the serialization of memory instructions at runtime. Therefore, optimizing the execution of these memory instructions can yield substantial performance improvements for modern genome sequence analysis algorithms running on CPUs. (2) As discussed in Section II-E and demonstrated in our experimental findings outlined in Section VII-D, the performance advantages of GPU-based approaches are diminished when processing long sequences due to insufficient on-chip memory and slow off-chip memory accesses. This provides an opportunity for CPU-based implementations such as our proposal (QUETZAL) to outperform GPUs through hardware-software co-design tailored for long sequences.

F. Vectorizing the modern genome sequence analysis algorithms

Commodity high-performance CPUs include support for vector hardware composed of a Vector Register File (VRF), where each vector register is an array of elements, and a Vector Processing Unit (VPU), which consists of multiple parallel execution units referred to as lanes [80]. The number of data elements stored in a vector register and processed by the VPU is referred to as vector length (*vlen*), *e.g.,* 16 *int32* elements in the Fujitsu A64FX's VPU [81].

We analyze the performance of vector implementations of two modern genome sequence analysis algorithms, `WFA` and `SS`. Because there is no vectorized implementation available for both algorithms, we implemented an in-house vectorized version for the *extend* function [82] in `WFA`, and the *diagonals comparison* step in `SS` [83]. Both operations are the most time consuming part of each algorithm, taking from 55% to more than 90% of the total respective execution times. We evaluate both vector approaches in Section VII.

The extend function in `WFA` (pseudo-code in Fig. 2.a) calculates the offsets for a specific number of waves. The outer loop traverses all the waves from low to high boundaries (line 2), and each iteration of the outer loop is executed in a different vector lane. The inner loop (lines 8-19) traverses the input sequences and accounts for the consecutive matching elements. In each iteration, any lane with a mismatch is deactivated. The inner loop stops iterating when no active lanes remain (i.e., when a mismatch has been found in every lane).

```
1:  vlen = get_vector_length()
2:  for (k=lo; k<=hi; k+=vlen)
3:    k_v = vindex(k, 1)  // {k, k+1, ....}
4:    mask = pwhileLt(k, hi)  // k < hi
5:    off = vload(mask_v, offsets + k)
6:    h_v = off
7:    v_v = vsub(mask, off, k_v)
8:    do
9:      text = gather(mask, text, h_v)
10:     patt = gather(mask, pattern, v_v)
11:     cmp = vxor(mask, text, patt)
12:     val = vreverseBytes(mask, cmp)
12:     val = vctlz(mask, val)
13:     val = vshiftr(mask, val, 3) // val/8
14:     off = vadd(mask, val, off)
15:     mask = vcmpEq(mask, cmp, 0)
16:     cnt = vpred_cnt(mask, mask)
17:     v_v = vadd(mask, v_v, 8)
18:     h_v = vadd(mask, h_v, 8)
19:   while (cnt > 0)
20:   vstore(offsets + k, off)
```

a) Vector pseudocode for WFA

```
1:  vlen = get_vector_length()
2:  compare(beg, end, n_v, ne_v)
3:    mask = vMaskAllTrue()
4:    count = vcreate(0)
5:    for (n=beg; n<end; n+=8)
6:      patt = gather(mask, pattern, n_v)
7:      text = gather(mask, text, n_e_v)
8:      cmp = vxor(mask, patt, text)
9:      val = vreverseBytes(mask, cmp)
10:     val = vctlz(mask, val)
11:     val = vshiftR(mask, val,  3) // val/8
12:     count = vadd(mask, count, val)
13:     mask = vcmpEq(mask, val, 0)
14:     n_v = vadd(mask, n_v, 8)
15:     ne_v = vadd(mask, count, 8)
16:   return count
17-60:  ....
61: for (e=1; e<editThreshold; e+=vlen)
62:   e_v = vindex(e, 1)  // {e, e+1, ....}
63:   n_v = batch.beg
64:   n_e_v = vsub(n_v, e_v)
65:   cnt1 = compare(batch.beg,
                     batch.end, n_v, n_e_v)
66:   n_e_v = vadd(n_v, e_v)
67:   cnt2 = compare(batch.beg,
                     batch.end, n_v, n_e_v)
68:   global += vreduce(cnt1)
69:   global += vreduce(cnt2)
```

b) Vector pseudocode for SS

Fig. 2. **Vector pseudocode for `WFA` (a) and `SS` (b) respectively.**

The *diagonals comparison* step in `SS` counts the number of maximum exact matches between two input sequences. The algorithm traverses the input sequences in batches and processes as many diagonals as the edit threshold value (line 61). Then, it calculates the offset for each diagonal (lines 62-64,66) and calculates the number of exact matches in the lower (line 65) and upper diagonals (line 67). To calculate the exact matches, the algorithm traverses both inputs and counts the number of consecutive matches (lines 2-16), similarly to the `WFA` algorithm. Finally, it updates a global counter used for the remaining code to filter the input sequences.

Fig. 3 depicts the performance benefits of the vectorization of the `WFA` and `SS` algorithms. On average, vectorized approaches provide 1.3× and 2.5× higher performance for short and long sequences, respectively (see Section V for our methodology). When processing short sequences, the outer loop in `WFA` (line 2) and `SS` (line 61) perform fewer iterations, leading to lower benefits for vectorized code. For long sequences, the number of iterations increases, providing more data parallelism to be exploited by the available vector hardware.

G. Challenges in accelerating modern genome sequence analysis

algorithms on vector architectures

Algorithms in Fig. 2 use scatter/gather memory instructions[1] to traverse the input sequences. These instructions split a vector memory request into multiple scalar memory requests. Although these scalar requests can be pipelined, they take more cycles to process compared to vector memory instructions that have a stride. First, each request has to calculate an associated address independently. Thus, the core requires multiple cycles to send all

[1] scatter/gather memory instructions are also called the memory-indexed instructions. This paper uses these two terms interchangeably.

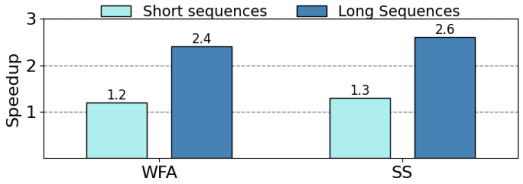

Fig. 3. **Execution speedup for WFA and SS vectorized algorithms. Results are normalized to the baseline implementations [18, 32].**

the scalar requests to the memory subsystem. Second, the load-store queue does not perform any memory coalescing for memory-indexed instructions. This increases the overall latency of the scatter/gather instructions. For example, in Intel and Fujitsu A64FX processors, scatter/gather instruction latency is at least 22 and 19 cycles, respectively, even when all the requested data is already in the L1D cache [81, 84, 85]. To better understand this bottleneck, Fig. 4 depicts the breakdown of the execution time for three vectorized ASM benchmarks, our in-house vectorized `WFA`, `BiWFA` and `SS`, running on an HPC ARM machine [81] with two levels of cache, using the methodology outlined in Section V. The figure shows how cache accesses represent a large fraction (32% to 65%) of the overall execution time in all algorithms. This bottleneck is exacerbated with the longer sequences for two reasons. First, the active working set in these algorithms expands with larger sequence size, surpassing the capacity of the on-device memory. Second, as scatter/gather instructions split one vector memory request into multiple scalar memory requests, they occupy processor pipeline structures such as load/store queues, reservation station, and caches, which slows down the execution of other (memory) operations. Therefore, designing hardware that can efficiently execute scatter/gather instructions can considerably improve the performance of genome sequence analysis algorithms.

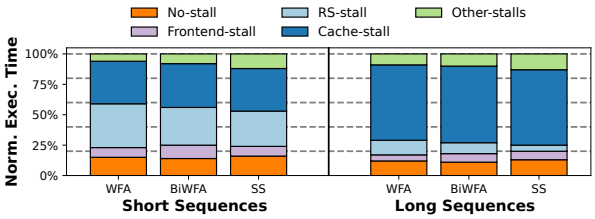

Fig. 4. **Execution breakdown of vectorized WFA, BiWFA and SS algorithms for short and long input sequences, broken down into: no-stall and stalls due to frontend, Reservation Station (RS), cache and others.**

III. QUETZAL: Overview

`QUETZAL` is a vector acceleration framework consisting of two main components: a vector accelerator tightly coupled to the VPU datapath and a set of novel vector instructions that expose the functionality of the accelerator to the programming model. `QUETZAL` design is driven by three main goals: (1) accelerate memory-indexed instructions in modern ASM algorithms, (2) provide a flexible framework applicable to multiple algorithms, and (3) achieve a light-weight hardware implementation that takes advantage of the hardware already available in VPUs such as x86 AVX512 [85] and ARM SVE [81, 86]. First, we analyze how the `QUETZAL` hardware and ISA extensions accelerate memory-indexed instructions (Section III-A), enabling the first goal. Then, we show how to integrate `QUETZAL` into `WFA` and `SS`, enabling the second goal (Section III-C). Finally, we describe `QUETZAL`'s microarchitecture, which implements the `QUETZAL` ISA, achieving a lightweight hardware implementation, enabling the third goal (Section IV).

`QUETZAL` accelerator is composed of four main components, as shown in Fig. 5: (1) Two hardware buffers directly connected to the VPU (Each buffer is refered as `QBUFFER`, Section IV-B) to quickly forward data to the vector ALU without using the cache hierarchy (*e.g.,* the input sequences in `WFA` and `SS`). (2) `data encoder` (Section IV-A) that applies a static bit-encoding to reduce the size of the DNA/RNA input sequences stored in the `QBUFFER`s. (3) `access ctrl` (Section IV-C) logic that processes all the data accesses from the VPU to the `QBUFFER`s and works as the interface between the `QBUFFER`s and the core's VPU components, and (4) `count ALU`s (Section IV-D) that count the number of consecutive elements between two input values.

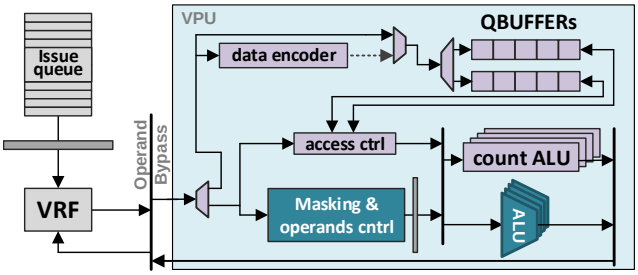

Fig. 5. **Overview of QUETZAL hardware (purple) integrated into the VPU datapath (dark blue).**

A. Accelerating memory-indexed instructions

As mentioned in Section II-G, memory-indexed instructions are split into multiple scalar memory requests and their execution is at best pipelined. This increases their overall latency. Processing all these memory requests concurrently would require substantial changes to the core microarchitecture and cache hierarchy. For example, the number of Address Generation Units (AGUs) in the core and the number of ports in the cache hierarchy must be increased to match the vector length to supply enough bandwidth to the Vector ALU. However, these modifications would considerably increase the total SoC hardware area and energy consumption.

To avoid the area and power overheads of modifying the entire cache hierarchy, `QUETZAL` incorporates two `QBUFFER`s specifically designed to deliver sufficient bandwidth to the VPU for rapid execution of indexed memory instructions. These `QBUFFER`s store frequently used genome sequence analysis data, in particular those that are accessed through indexed memory instructions. Then, the algorithm utilizes `QUETZAL` instructions to access the values previously stored in the `QBUFFER`s. `QUETZAL` does not aim to eliminate scatter/gather support from the cache hierarchy; instead, it aims to work cooperatively with it. `QUETZAL` efficiently facilitates the execution of memory-indexed operations on a fixed number of *hot* values within the active working set. Meanwhile,

the cache hierarchy is used for unit-stride memory operations and to scatter/gather less frequently accessed values from larger data structures, which are larger than the QBUFFERs. As shown in Section VI, QUETZAL takes advantage of the sequence size from established and emerging sequencing technologies [61, 87] to set the size of these QBUFFERs to 16 KB.

QBUFFERs feature three key characteristics that enable them to provide more efficient support for memory-indexed operations. (1) They are direct-mapped. As such, instead of using a memory address, QUETZAL uses an index to access the QBUFFERs, thus requiring simpler control logic compared to caches. (2) QBUFFERs are highly multiported structures, allowing the Vector ALU to access data in just two cycles (Section IV-B), a significant improvement over the 22 or 19 cycles required in Intel and A64FX cores [81, 85], respectively. (3) QBUFFERs support bit-encoded values, i.e., accessing data at sub-byte granularity. For example, genomic sequences use an alphabet of four characters, thus a two-bit encoding is sufficient and much more efficient than a conventional byte-sized encoding (see Section IV-A). The QBUFFERs enable efficient accesses to such unaligned data.

While QUETZAL is primarily designed for accelerating genome sequence analysis algorithms, the proposed microarchitectural enhancements enable QUETZAL to also accelerate applications in other domains (see Sections III-D and III-E).

QUETZAL Instructions. We introduce the instructions used to work with the aforementioned QBUFFERs.

`qzencode(int SEL, vreg VAL, reg Idx)`: This instruction encodes the input sequences in the input vector *VAL* and stores the encoded values in the QBUFFERs. It applies a static bit-encoding to reduce DNA/RNA inputs from an 8-bit representation to a 2-bit representation. *SEL* and *Idx* specify the QBUFFER and position where the encoded data will be stored, respectively.

`qzstore(vreg VAL, vreg IDX, int SEL)`: This instruction stores the values from the input vector *VAL* to the QBUFFERs. *SEL* works the same way as in `qzencode`. Each value in *IDX* is an index to store its corresponding element from *VAL* in the QBUFFER.

`qzload(vreg IDX, int SEL)`: This instruction reads data from the QBUFFERs and outputs a vector register with the read values. *SEL* specifies the read QBUFFER. Each value in *IDX* is an index to read the QBUFFER from QUETZAL.

`qzconf(reg Eb0, reg Eb1, reg Esize)`: It is used to configure the size of the data stored in QUETZAL. Registers Eb0 and Eb1 indicate the number of elements stored in each QBUFFER. Esize indicates the element size (0: 2-bit (encoded), 1: 8-bit (chars) and 2: 64-bit elements).

`qzmhm<OPN>(vreg IDX0, vreg IDX1)`: This instruction reads data stored in the QBUFFERs and computes the operation specified by the opcode *OPN* to the read values (e.g., addition, comparison, etc.). Each element in *IDX0* and *IDX1* is used to read a different position in each of the QBUFFERs respectively. The values read from the QBUFFERs are then processed by *OPN* and the results are packed in a vector register as output.

`qzmm<OPN>(vreg VAL, vreg IDX, int SEL)`: This instruction works similar to `qzmhm`, however it processes both data from the QBUFFERs and the VRF (*VAL*). Each element in *IDX* is an index to read a different position in the QBUFFER specified by *SEL*. The read values are forwarded to *OPN* together with the corresponding element in *VAL*. Finally, this instruction outputs a vector register with all the results.

B. Accelerating counting consecutive matching elements

Counting consecutive matching elements is useful for different applications that calculate maximal exact matches (MEMs) [88] and maximal unique matches (MUMs) [89] including SneakySnake [18], protein multiple sequence alignment [90], read mapping [88], and sequence alignment [91]. QUETZAL features a specialized processing unit capable of efficiently counting the number of consecutive matches between two input sequences. We employ this functional unit together with QBUFFERs to significantly reduce the instruction overhead of modern algorithms when counting consecutive matching elements. The `qzcount` instruction is used to take advantage of this specialized unit.

`qzcount(vreg VAL0, vreg VAL1)`: Both input vectors are split into 64-bit segments; each segment from *VAL0* is processed together with its corresponding segment from *VAL1*. Then, the instruction counts the consecutive matching elements in each 64-bit segment and outputs a vector register with the individual results.

This instruction can be executed standalone or with the `qzmhm` instruction to access the QBUFFERs and count consecutive matches in values previously stored in QBUFFERs.

C. QUETZAL ISA use cases: WFA and SS

Fig. 6 depicts QUETZAL-based implementations for the WFA (a) and SS (b) algorithms. For both algorithms, we first store the input sequences in the QBUFFERs (line 3) and use the `qzcnf` instruction to configure the number of elements and element size (line 4). When using QUETZAL, both algorithms execute using the the `qzmhm<qzcount>` instructions —line 11 and 8 respectively. First, `qzmhm` reads the input sequences stored in the QBUFFERs and executes the `qzcount` functionality to count the number of consecutive matching elements from the read values. By using both instructions (1) memory-indexed instructions are accelerated directly in QUETZAL and (2) the number of instructions in the inner loop of WFA and *compare* function in SS are significantly reduced. As shown in our evaluation in Section VII-A1, QUETZAL significantly outperforms vectorized WFA and SS implementations.

D. QUETZAL on classical DP algorithms

Fig. 7.a depicts the execution flow to process one anti-diagonal in classical DP-based algorithms using commercial vector architectures (❶, ❷) and QUETZAL hardware (❸, ❹). A new anti-diagonal is calculated using the immediate two previous anti-diagonals ❶. The computed diagonal and one vector register from the previous step can be reused to compute the next diagonal ❷; however, new values must be loaded from memory. These new values are composed of elements in the diagonal computed in the previous step and a pre-computed value. In ❸ and ❹ we use QUETZAL to reduce the store-load forwarding from steps ❶ and ❷ by placing one of the input sequences and the pre-computed values in the QBUFFERs. Then, the algorithm reads these values directly from QBUFFERs using `qzload` without using the cache hierarchy.

```
1: vlen = get_vector_length()
2: // The following function uses qzstore
3: store_sequences_in_quetzal()
4: qzconf(pattern.size, text.size,
                          size:2-bit)
5: for (k=lo; k<=hi; k+=vlen)
6:   k_v = vindex(k, 1) // {k, k+1, ....}
7:   mask = pwhileLt(k, hi) // k < hi
8:   off = vload(mask_v, offsets + k)
9:   h_v = off
10:  v_v = vsub(mask, off, k_v)
11:    val=qzmhm<qzcount>(h_v, v_v)
12:    val=qzmhm<qzcount>(h_v, v_v)
13:    off = vadd(mask, val, off)
14:    mask = vcmpNEq(mask, val, 0)
15:    cnt = vpred_cnt(mask, mask)
16:    v_v = vadd(mask, v_v, 8)
17:    h_v = vadd(mask, h_v, 8)
18:  while (cnt > 0)
19:  vstore(offsets + k, off)
```

a) QUETZAL pseudocode for WFA

```
1: vlen = get_vector_length()
2: // The following function uses qzstore
3: store_sequences_in_quetzal()
4: qzconf(pattern.size, text.size,
                          size:2-bit)
5: compare(beg, end, n_v, ne_v)
6:   mask = vMaskAllTrue()
7:   count = vcreate(0)
8:     val=qzmhm<qzcount>(n_v,ne_v)
9:     val=qzmhm<qzcount>(n_v,ne_v)
10:    count = vadd(mask, count, val)
11:    mask = vcmpEq(mask, val, 0)
12:    n_v = vadd(mask, n_v, 8)
13:    ne_v = vadd(mask, count, 8)
14:  return count
15-58: .... // Same than SS lines 17-69
```

b) QUETZAL pseudocode for SS

Fig. 6. **QUETZAL-based pseudocode for WFA and SS algorithms. QUETZAL instructions are highlighted using red background color.**

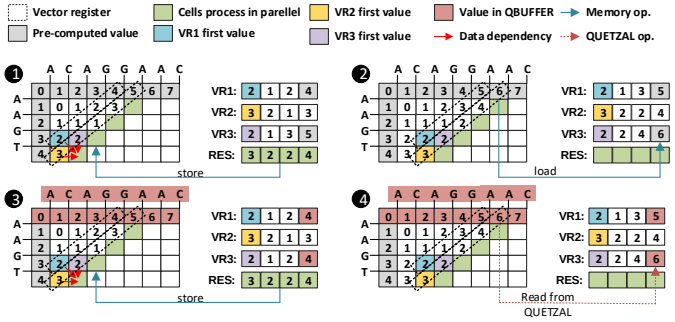

Fig. 7. **Execution flow for a vectorized classic DP-based algorithm using regular vector instructions (1),(2) and QUETZAL instructions (3),(4).**

E. QUETZAL applied to other application domains

QUETZAL has been designed to accelerate memory-indexed instructions with modern genome sequence analysis applications in mind. Nevertheless, these instructions also form bottlenecks in other application domains, where QUETZAL can be directly applied to improve their performance. We use the histogram calculation algorithm [92] as a representative kernel. This algorithm is a key component of database query planning [93, 94] and is heavily used in image processing [95, 96]. Histogram calculation is dominated by pointer chasing operations executed using memory-indexed instructions. Fig. 8 depicts the QUETZAL-based implementation of histogram calculation. The algorithm directly reads and updates the histogram table in the QBUFFERs, thereby reducing the latency due to memory-indexed instructions. We analyze the performance of this algorithm in Section VII-F.

IV. QUETZAL Microarchitecture

This section details the design, implementation, and integration of QUETZAL's hardware components.

A. Data encoder

QUETZAL uses two data encoding schemes, 2-bit and 8-bit encoding, to encode each of the bases of the input sequences. If the two input genomic sequences are RNAs or DNAs, then they can be encoded using a 2-bit unique binary representation since

```
1: vlen = get_vector_length()
2: // Set the histogram size and element size
3: qzconf(histogram.size, 0, size:64-bit)
4: for (i=0; i<input.size; i+=vlen)
5:   input_v = vload(&input[i])
6:   hist_v = qzload(input_v, Buffer0)
7:   hist_v = vadd(hist_v, 1)
8:   hist_v = reduce_conflicts(hist_v)
9:   qzstore(hist_v, input_v Buffer0)
```

Fig. 8. **QUETZAL-based histogram algorithm. QUETZAL instructions are highlighted using red background color.**

both RNA and DNA alphabets have 4 characters (A, C, G, and T for DNA, and A, C, G, and U for RNA). If the two genomic sequences are proteins (or the ambiguous nucleotide, N, which may need to be represented in DNA or RNA), then each base can be encoded into an 8-bit unique binary representation as the protein alphabet has 20 characters. With this, QUETZAL can efficiently handle different types of sequencing data and reduces the size of the QBUFFERs required to store long sequences, such as reads from long-read sequencing technologies (e.g., 10K - 30K base pairs for PacBio [61]). For the DNA/RNA genomic sequences, the data encoder receives a vector of characters from the VRF and it extracts the bits 1 and 2 from each character to generate the 2-bit representation. The extracted bits are packed together in a single vector and forwarded to one of the QBUFFERs. Fig. 9 shows the encoding table (a) and an example encoding for a 512-bit vector (b).

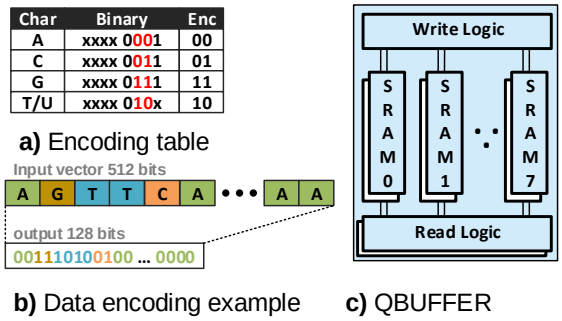

Fig. 9. **QUETZAL bit-encoding logic and QBuffer architecture.**

B. QBUFFER

QBUFFERs are used to reduce the access latency of memory-indexed instructions for algorithms such as WFA and SS. In contrast to other structures in the CPU such as the VRF and caches, the QBUFFER (1) is implemented as a multi-ported structure to provide high bandwidth to the VPU while minimizing its area using multiple low-cost single-port modules, (2) supports bit-encoded data and unaligned data accesses, and (3) works at word granularity. QBUFFER is composed of three hardware blocks: SRAM blocks, write logic, and read logic (Fig. 9.c).

1) SRAM Blocks (SRAMs): These SRAMs store the values used by the VPU. Each SRAM has a 64-bit word length, which matches the bit-width of each VPU lane. To provide enough bandwidth to the qzencode and qzstore instructions when storing consecutive elements, we implement the QBUFFER using a multibanked approach, placing one bank for each of the eight

64-bit VPU lanes. Together, these form a 512-bit vector. The indices to map the SRAMs are interleaved (similar to the VRF). To reduce the latency of multiple concurrent read access to the QBUFFER, we design it as a multi-ported structure. To reduce the area overhead of true multi-ported buffers, we implemented the QBUFFER using data replication [97]. A read port consists of eight SRAMs connected to a single read logic module. A new port can be added by placing a new set of SRAMs and read logic instances in the QBUFFER. To store data in the QBUFFER, the write logic module determines the SRAM column to store the data and writes the value to each SRAM instance, i.e., one copy per read port.

2) Write logic: The write logic stores data into the QBUFFER in two modes: *encoded*- and *direct*-mode. In *encoded*-mode, it receives a 128-bit vector from the data encoder and an *index*. It splits the input vector into two 64-bit segments (segA and segB) and stores them in two consecutive SRAM columns in the position specified by the *index*. A writes in *encoded*-mode is executed in a single cycle. In *direct*-mode, this module receives two input vectors from the VRF: a vector of indices *IDX* and a vector of values *VAL*. First, it splits *IDX* and *VAL* into 64-bit segments. Then, for every pair of segments, it uses the element of *IDX* to select the column and position in the SRAMs to store the corresponding element from *VAL*. The latency of a write in *direct*-mode are depends on the number of concurrent accesses to the same SRAM column. For example, if all the requests go to the same bank, the *direct*-mode write latency will be eight cycles.

3) Read logic: The read logic is used to access the data stored in the SRAMs. QUETZAL supports three different element sizes: 2- and 8-bit encoded data, and 64-bit data. Thus, data access to the QBUFFER might be unaligned with respect to the SRAM word size. To enable unaligned read operations to QUETZAL, we read one word from two consecutive SRAMs and create a single output. Fig. 10 depicts the functionality of the read logic module. The module receives two inputs, an *index* and the *element size*. The *index* and the *element size* are used by the access logic to read the SRAMs, and the *element size* is used by the slicing logic to generate the appropriate output. First, the access logic splits the input index into *set*, *bank*, and *offset* values based on the *element size* ❶. Then, it reads the content from two consecutive SRAMs using the *bank* and *set* values (words *W1* and *W2*) ❷. Next, the slicing logic uses the *offset* value slice the values of *W1* and *W2* ❸. Finally, the module selects the order to pack the sliced values ❹ and generates a single 64-bit output ❺. If the element size if 64 bits, steps ❸ - ❺ only select the corresponding value from one of the two SRAMs.

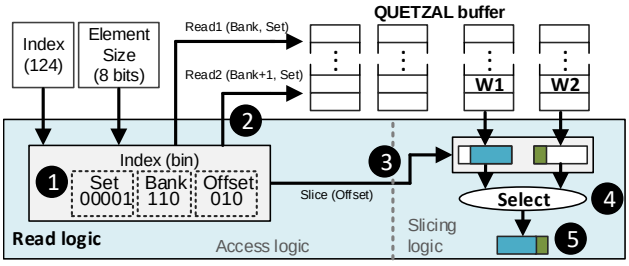

Fig. 10. **Functionality of the `read logic` module.**

C. Access Control

The access ctrl module works as the interface between the VPU micro-architectural components and QBUFFERs. It executes the read/write operations in the QBUFFERs and forwards the read data to the VPU's ALU. It has three registers that are configured with the qzconf instruction. Two registers of them hold the sizes of the input sequences in the QBUFFERs, and the last register specifies the element size of data stored in QUETZAL. This module features two main operations: (1) It controls all the read accesses to QBUFFERs. The module receives one or two vectors of indices (depending on the instruction) and forwards them to QBUFFERs together with the element size. After receiving the response (i.e., data) from QBUFFERs, it forwards the data to the ALUs. (2) It controls the write operation for the qzstore instructions. It receives a vector of indices and a vector of values, using the indices to store the values in the QBUFFERs.

1) Connection with the QBUFFERs: As mentioned in Section IV-B, QBUFFERs are implemented as a single write- and multiple read-port structures, where the number of read ports is an implementation decision. The access ctrl module can fetch eight concurrent read requests from each QBUFFER, i.e., eight 64-bit elements for a 512-bit vector. Thus, the number of cycles required to process all the requests is directly related to the number of ports in QBUFFERs. When the number of requests to a QBUFFER surpasses the number of read ports, some requests are stalled, and the QBUFFER processes them in a round-robin manner. The total latency to process all the read requests requires $8/(num\_ports) + 1$ cycles (the additional cycle is needed for slicing). Section VI analyzes the performance impact of different numbers of ports and read latencies for QBUFFERs.

D. Count ALU

The count ALU module implements the hardware pipeline for the qzcount instruction functionality (Fig. 11). It processes two 64-bit elements. QUETZAL includes as many instances of this module as the number of 64-bit lanes in the VPU. When executing the qzcount instruction, count ALU receives two input values and their element size (*e.g.,* 2-bit). First, it applies a bitwise *xnor* operation to detect matching bits ❶. Then, it implements the logic to count the number of *trailing ones* from the previous operation ❷. The result of this operation determines the number of consecutive matching bits between the two inputs. In the next stage, the number of *trailing ones* is right shifted depending on the element size value ❸, to obtain the number of matching elements. For example, for 2-, 8- and 64-bit elements, the number of *trailing ones* is shifted by one, three, and six, respectively.

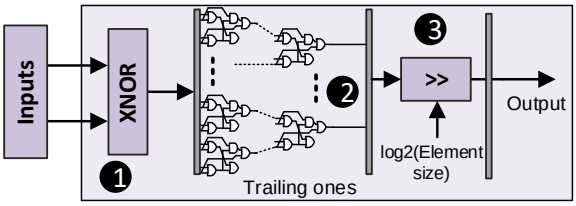

Fig. 11. **Hardware implementation of the qzcount instruction.**

E. Integration with Out-of-Order (OoO) processors

We discuss how to integrate QUETZAL to a commercial OoO processor pipeline.

Qzstore and qzload execute at commit: In our design, QBUFFERs work as direct-mapped structures where the vector ALU directly stores and loads data to/from them. When reading and updating values in the QBUFFERs, there is a risk of overwriting useful data or reading erroneous values in these structures due to speculative execution. To avoid this, we execute the aforementioned instructions as *non-speculative* operations. Thus, these two instructions wait in the issue queues until they are the oldest instructions in the queue. Then, a *ready to execute* signal is sent to the issue logic. For other instructions, algorithms use them only to access data previously stored in the QBUFFERs and are executed speculatively.

Processor exceptions: When an exception occurs (*e.g.,* a TLB refill exception), the processor will jumps to an Operating System (OS) subroutine that specifically manages the exception. As the qzstore instruction executes at commit, there is no risk of affecting the values in QBUFFERs, and the state of QUETZAL is preserved when the processor resumes execution after resolving the exception.

Architectural state and context switches: QBUFFERs are architectural state, and must be saved across context switches. As is done for vector ALU and FPU state, OS need not save and restore their state on every system call or interrupt, only when the process is descheduled. As context switches occur infrequently, saving QBUFFERs state represents a negligible fraction of the time spent in OS code.

V. Experimental Methodology

A. Simulation Framework

We use the gem5 simulator [98, 99] to evaluate the functionality and performance of QUETZAL. We simulate an ARM 64-bit (aarch64) full-system running an Ubuntu 20.04 with a 4.18.0+ Linux Kernel. We model and validate gem5 against a Fujitsu A64FX-like [81, 100] architecture, which is the processor for the Fugaku, the #2 supercomputer in the Top500 list [101, 102] as of June 2023. Table I summarizes the main simulation parameters. Each simulated core includes a QUETZAL module interconnected to its VPU. We extend the Out-of-Order model of gem5 with the structures detailed in Section IV QUETZAL. We extend the ARM SVE ISA with QUETZAL instructions from Section IV for the software support. As described in Section IV-C1, the latency of accessing data stored in QUETZAL depends on the number of ports in QBUFFER. We model this behavior in detail in gem5 to ensure the accuracy of the performance results.

TABLE I
Simulated System Setup.

CPU: 2.0 GHz, 16-core A64FX-like [81, 100] superscalar OoO **Vector ISA**: ARM SVE ISA - 512-bit Vector Length **Baseline VPU**: Instructions latency from the A64FX Manual
L1-I: 64KB, 8-way assoc., load-to-use = 2 cycles, Stride prefetcher
L1-D: 64KB, 8-way assoc., load-to-use = 4 cycles, Stride prefetcher
L2 Cache: 8MB, shared, 16-way assoc., load-to-use = 37, Stride prefetcher
DRAM: 4-channel HBM2
QUETZAL Configurations
QZ_1P Single-port —QBUFFERs: 8KB each —read latency = 9 cycles **QZ_2P** Dual-port —QBUFFERs: 8KB each —read latency = 5 cycles **QZ_4P** Quad-port —QBUFFERs: 8KB each —read latency = 3 cycles **QZ_8P** Octa-port —QBUFFERs: 8KB each —read latency = 2 cycles

B. Benchmarks

Use case 1: Modern read aligners. For this use case, we evaluate the efficiency of QUETZAL over the WFA and BiWFA algorithms. We use the best-performing configurations reported by Marco-Sola *et al.* [35, 82].

Use case 2: Edit distance approximation. We use the state-of-the-art edit distance approximation technique SneakySnake (SS) [18]. SS filters the input reads to skip the alignment of those inputs that exceed a defined edit distance threshold parameter.

Use case 3: Classical read aligners. We evaluate the applicability and performance benefits of QUETZAL over two classical read alignment algorithms SW (ksw2 [30]) and NW (parasail [70]).

Use case 4: Protein alignment. We evaluate the effectiveness of QUETZAL on datasets with a different and larger alphabet than A, C, G, and T. Protein alignment is required in various proteomics applications, such as in the extension step of protein database searches [103, 104]. We use all the algorithms from use cases 1 and 2 to process protein sequences.

Use case 5: Edit distance approximation+alignment. In the genome sequence analysis pipeline, there are multiple algorithms interconnected, such as filtering and sequence alignment. We demonstrate the flexibility and efficiency of QUETZAL to accelerate multiple stages from the genome sequence analysis pipeline. To this end, we use the SS and WFA algorithms and develop a single implementation combining them. SS filters the input pairs and WFA executes the read alignment on the accepted sequences.

For comparison to baseline techniques, we use the auto-vectorization support from the compiler. For WFA, BiWFA and SS algorithms, we implement an in-house vectorized version using ARM SVE ISA intrinsics. For ksw2 and parasail, we use their open-source vectorized implementations [105, 106] and adapt them to the SVE ISA. We develop a QUETZAL-based implementation for each algorithm evaluated using intrinsics to insert the proposed instructions in the code. We validate the correctness for each QUETZAL implementation by bit-wise comparing their outputs with their corresponding baseline version. For QUETZAL, the execution time reported includes the time the algorithm takes to store the input sequences into the QBUFFERs. We consider the traceback stage execution time in all our experiments.

C. Datasets

For DNA/RNA inputs, the evaluate datasets ranging from 100 base pairs (*bp*) to 30K base pairs. We use two real datasets (*100bp_1*, *250bp_1*) and two simulated datasets (*10Kbp* and *30Kbp*). Table II summarizes the main characteristics of the evaluated datasets. For the *100bp_1*, *250bp_1* and *10Kbp*, we use the datasets available in the SneakySnake repository [83]. We generate the *30Kbp* dataset following the same methodology as SneakySnake [18]. The *100bp_1* and *250bp_1* datasets are representative of the newest short-read technologies available in the market, ranging from the Ilumina

iSeq100 which generates 100bp to the Illumina Next Generation Sequencing (NGS) that generates 300bp [87, 107, 108]. The *10Kbp* and *30Kbp* datasets are representative of long-read technologies such as PacBio that released a new HiFi technology that generates long-read in the range of 10K - 30K base pairs [60, 61, 87]. These sequencing technologies generate Gigabytes of sequences. However, we were compelled to constrain the number of input reads in the datasets to get each experiment simulated in a reasonable time, *e.g.,* days/weeks instead of months. Nevertheless, all the evaluated datasets exceed the LLC capacity significantly. This allows us to represent behaviors indicative of non-cache resident workloads.

For protein alignment, we evaluate the entire BAliBase4 dataset from [109]. For each multiple sequence alignment group in the dataset, we run the pairwise alignment of all possible pairs within the group. For example, for a multiple sequence alignment of 5 sequences, we run 4+3+2+1=10 pairwise sequence alignments.

TABLE II
INPUT DATASET CHARACTERISTICS.

Dataset	Read Length	No. of Pairs (K)	Dataset Size
100bp_1	100	500	95MB
250bp_1	250	500	298MB
10Kbp	10,000	20	381MB
30Kbp	30,000	7	400MB

D. Comparison between QUETZAL and GPU approaches

We compare the performance of the QUETZAL-based implementations of WFA and ksw2 against WFA-GPU [76] and Gasal2 [75] respectively, two GPU approaches using the same algorithms. In these experiments we use a 16-core CPU featuring QUETZAL and an NVIDIA A40 GPU. We use the open-source implementation available for each GPU approach [110, 111].

VI. Design Space Exploration

In this section, we right-size the QUETZAL hardware implementation through a design space exploration. In QUETZAL, QBUFFERs are the critical components with two key parameters (size and number of ports) that directly affect the performance, area, and power consumption of QUETZAL. To size the QBUFFERs, we consider the applicability of QUETZAL on both short- and long-read sequencing technologies. For short reads, we consider the Illumina sequencing technology (100 bp), and for long reads, the HiFi PacBio technology (10K - 30 Kbp). Based on these sequencing technologies, we size the QBUFFERs to 8KB each for the pattern and text buffers (16KB in total). With the `data encoder` module in QUETZAL, each QBUFFER could store up to 32.7Kbp sequences that cover both technology use cases (Illumina and HiFi PacBio).

QUETZAL supports direct hardware acceleration of sequences that are up to 30K base pairs through the QBUFFERs. Some of the modern sequencing frameworks support sequences that are extremely long, an example is Oxford Nanopore which can support up to 2M base pairs [112]. QUETZAL can support such frameworks through software support. For example, it can utilize read mappers such as minimap2 [31], which can generate shorter subsequences from larger sequences where subsequence length is the same as QBUFFER size, mitigating overheads that might stem from performing sequence alignment on the complete, extremely long sequence. Additionally, windowed [36, 42, 48] and tiling [113, 114] software approaches could also be utilized. These divide the input sequence into shorter subsequences, and thus, for independent processing.

TABLE III
AREA AND POWER COMPARISON BETWEEN DIFFERENT QUETZAL CONFIGURATIONS (7NM TECHNOLOGY).

(A) Config.	(B) Area	(C) Leakage Power	(D) A64FX Core	(E) A64FX SoC
QZ_1P	0.013mm^2	98uW	+0.46%	+0.21%
QZ_2P	0.026mm^2	189uW	+0.88%	+0.37%
QZ_4P	0.048mm^2	370uW	+1.75%	+0.69%
QZ_8P	0.097mm^2	746uW	+3.37%	+1.41%

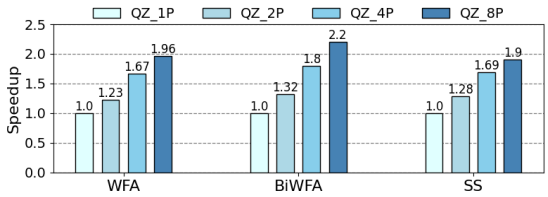

Fig. 12. **Relative performance of QUETZAL configurations with different numbers of ports. Results are normalized to the QZ_1P configuration.**

Number of ports vs performance, area and power consumption: We compare the impact on performance, area and power consumption of different numbers of QBUFFER read ports. To this end, we evaluate the QUETZAL configurations from Table I. The different QUETZAL versions have been physically implemented using Synopsys' ICC2 Place and Route tool [115]. Table III shows the area and power consumption. Columns *D* and *E* in Table III show the percentage of area overhead that each QUETZAL configuration adds to a Fujitsu A64FX core and System on Chip (SoC) respectively, considering a QUETZAL instance integrated into each core of the SoC. We quantify these overheads using the same methodology proposed by Arima et al. [116].

Fig. 12 depicts the performance results for all the QUETZAL configurations evaluated. Increasing the number of ports directly impacts the performance benefits observed in modern algorithms because the latency required to read data in the QBUFFERs is reduced. However, as we use data replication to implement read ports in QBUFFER, area and power significantly increase when a new read port is added. Nevertheless, we implement each bank using a single-ported SRAM to reduce their impact on area. Because of this, even the larger QUETZAL configuration (**QZ_8P**) features a relatively modest area overhead of 1.41% compared to the Fujitsu A64FX SoC, while providing significant performance improvement. Based on this analysis, we set the QUETZAL configuration to **QZ_8P** and use it for the rest of our experiments.

VII. Evaluation

We analyze the performance of QUETZAL for the use cases listed in Section V-B. We evaluate two different QUETZAL approaches, one that only uses the QBUFFERs (referred to as

QUETZAL) and another one that also includes the functionality of the count hardware and qzcount instruction (referred to as QUETZAL+C). Implementations using only SVE intrinsics are referred to as VEC. Performance results are normalized to the baseline version (compiler auto-vectorization) of each algorithm.

A. Single-core Performance Analysis

Fig. 13.a depicts the single-core performance results for all the evaluated algorithms.

1) Modern DP algorithms: For short reads, QUETZAL and QUETZAL+C provide 1.5× and 2.1× higher performance respectively compared to the VEC algorithm; and 5.1× and 5.5× respectively for long reads. Overall, these performance improvements result from (1) QBUFFERs accelerating memory-indexed instructions by reducing the read access latency to only 2 cycles and (2) the count hardware accelerating the process of counting consecutive matching elements with a single instruction.

When processing short reads, modern algorithms are dominated by both reservation station stalls and cache accesses (as shown in Section II-G). As such, QUETZAL+C provides significantly better performance by reducing the number of instructions executed. On the other hand, when processing long reads, these algorithms are dominated by cache accesses. As such, QUETZAL provides significant performance benefits even when using only the QBUFFERs.

2) Edit distance approximation: On average, a system with QUETZAL+C shows 2.1× and 5.2× better performance than the VEC algorithm for short and long reads respectively. The most time-consuming operation in SS counts the number of consecutive matching elements in different diagonals in the input sequences, and similar to WFA and BiWFA, the QBUFFERs and count ALU hardware efficiently accelerates this operation.

3) Classical DP algorithms: When executing classic DP approaches, we use QUETZAL to reduce the overhead from store-load forwarding operations by storing and directly reading precomputed values from the QBUFFERs. However, classical algorithms are dominated by long dependence chains which overshadow the latency benefits from QUETZAL. Even for long reads, QUETZAL provides modest performance benefits. On average, QUETZAL outperforms SW (ksw2) and NW (parasail) by 1.3× and 1.4×, respectively.

4) Protein sequence alignment: On average, QUETZAL and QUETZAL+C provide 6.0× and 6.6× higher performance respectively when aligning protein sequences. We make one observation. Because of the larger alphabet required in protein alignment, the overall number of edits required increases significantly. As a result, WFA, BiWFA, and SS algorithms feature longer execution time and require more iterations, increasing the number of operations accelerated by the qzmhm and qzcount instructions. Thanks to this, QUETZAL provides larger performance benefits with proteins compared to DNA/RNA inputs. We conclude that QUETZAL is highly efficient at improving the performance of ASM algorithms independently of the input alphabet.

B. Multicore scalability and cache memory utilization

Fig. 13.b depicts the multicore scalability evaluation of QUETZAL over all the previously evaluated algorithms and datasets using the QUETZAL+C configuration. All QUETZAL-based implementations demonstrate good performance scalability as thread count increases. Nevertheless, performance does not increase linearly with the number of threads. The main reason is related to the memory bandwidth available. For small input sequences, the cache hierarchy is enough to store all the DP matrices, providing linear speedups. However, for large input sequences, a single DP matrix is significantly larger than the LLC size, and off-chip memory requests are necessary to read and update these matrices. In this case, the number of off-chip memory requests increases with the number of cycles, and thus memory bandwidth limits performance scaling.

Fig. 14.a shows the reduction of memory requests issued to the cache hierarchy from QUETZAL compared to the VEC algorithms. All memory operations to access the input sequences are executed directly in QBUFFERs, significantly reducing the number of memory requests. Moreover, the remaining main memory are strided memory instructions that read and update the DP table. These operations are simpler, and other components, such as the cache prefetcher, are capable of accelerating them efficiently [117].

C. Edit distance approximation + alignment

We evaluate the ability of QUETZAL to accelerate two different algorithms (SS+WFA). We compare the VEC and QUETZAL+C implementations of these algorithms. Fig. 14.b shows the performance comparison for all the experiments using a 16-core machine. On average, QUETZAL outperforms the VEC implementation by 1.8×, 2.7×, 3.6× and 3.1× for the 100bp_1, 250bp_1, 10Kbp and 30Kbp datasets respectively. These experiments demonstrate the flexibility and integration of QUETZALto handle multiple algorithms in the genome sequence analysis pipeline, achieving notable performance benefits.

D. Comparison with GPU approaches

We evaluate the performance of QUETZAL compared to the GPU-based approaches listed in Section V-D. In our experiments, we use the entire NVIDIA A40 GPU and a 16-core QUETZAL capable CPU to align all the input datasets listed in Section V-C. We evaluate multiple alignment parameters for the GPU implementations and report the best-performing results.

Fig. 15.a shows the throughput results obtained. We make four observations: (1) When processing short sequences, the parallelism offered by GPUs can outperform VEC and QUETZAL designs. However, the NVIDIA A40 GPU consumes >10× more area compared to QUETZAL. (2) The sequence size limits the parallelism offered by GPUs. With longer sequence lengths, the active working set, encompassing metadata, DP matrix, and other structures, increases significantly. Consequently, the available on-chip memory can serve only a small number of GPU threads, an effect called *low occupancy*, which significantly reduces the performance for long sequences compared to shorter sequences [38, 76, 77]. For example, GPU approaches outperform WFA (VEC) and SW (VEC) by 2.0× and 1.3×, respectively, which represents a performance drop of 40% and 83%, respectively, compared to short sequences. (3) As analyzed in Section II-G, when processing long sequences, the execution time of modern genome sequence analysis algorithms is dominated by memory-indexed instructions. QUETZAL efficiently accelerates

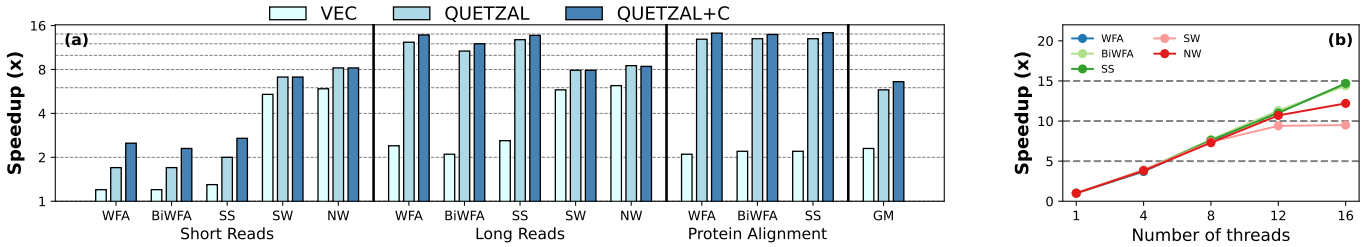

Fig. 13. **(a) Single-core performance results for all evaluated algorithms. (b) multi-core scalability from QUETZAL-based algorithms using the QUETZAL+C version of each algorithm.**

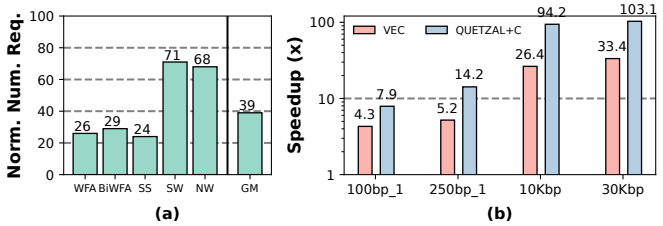

Fig. 14. **(a) Normalized number of cache memory requests of QUETZAL+C algorithms**. (b) Performance results for the SS+WFA implementation, results are normalized to the baseline WFA algorithm.

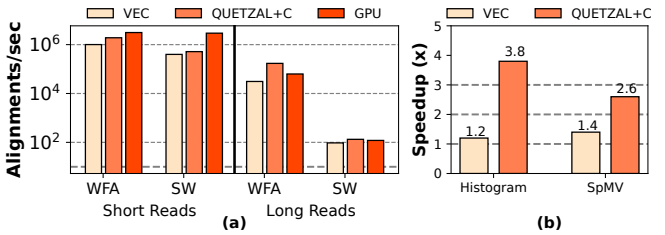

Fig. 15. **(a) Throughput comparison between QUETZAL+C and GPU approaches. Results are reported on a logarithmic scale.** (b) Performance results for Histogram and SpMV applications.

these instructions, providing notable performance benefits. On average, QUETZAL outperforms Gasal2 and WFA-GPU (two open-source GPU implementations) by 1.1× and 2.7×, respectively, for long sequences. (4) With the widespread adoption of long-read sequencing technologies in bioinformatics [7, 46, 47, 78, 79], leveraging QUETZAL allows CPUs to achieve higher throughput compared to GPUs, with small area overheads.

E. Comparison to domain-specific accelerators

Comparing different accelerators is not a trivial task due to the differences in the algorithms, architectures, and target technologies. GCUPS (Giga Cells Updated Per Second) [18, 72] is a commonly used metric to report the maximum DP-elements a solution can process per second. The calculation of GCUPS involves considering two factors: (1) the number of DP-cells calculated per alignment and (2) the average time taken by an algorithm to process an alignment. Table IV presents a comparative analysis of QUETZAL against several genome sequence analysis accelerators, using the PGCUPS (Peak GCUPS) metric, which represents the highest reported throughput achieved by an algorithm in processing an alignment. The area of each evaluated accelerator is scaled [118] to a 7nm technology node.

GenASM [36] and Darwin [119] outperform QUETZAL by 2.7× and 1.2×, but are limited to specific algorithms and alignment parameters. QUETZAL, however, can handle various algorithms and alignment parameters. QUETZAL achieves 0.6× and 4.1× throughput per area compared to WFAsic with and without backtracing respectively. WFAsic lacks hardware for processing the backtracing stage, necessitating data transfer to the host for execution, thereby leading to low performance. GenDP [67] is a domain-specific accelerator designed to accelerate classical DP algorithms, while QUETZAL excels in accelerating modern, more efficient algorithms, resulting in significant performance gains (10.9×) over GenDP.

TABLE IV
PEAK GIGA CELL UPDATES PER SECOND (PGCUPS) PER MM^2

Study	Device	Num. PEs	Area	PGCUPS/mm^2
QUETZAL	CPU	1 PE	0.097mm^2	
Core+QUETZAL	CPU	1 PE	2.9mm^2	554.8
GenASM [36]	ASIC	32 PE	1.37mm^2	1491.8
WFAsic [120] (Without Backtracing)	ASIC	1 PE	0.45mm^2	870.5
WFAsic [120] (With Backtracing)	ASIC	1 PE	0.45mm^2	136.1
GenDP [67]	ASIC	64 PE	5.82mm^2	51.0
Darwin [68]	ASIC	64 PE	5.06mm^2	685.6

While fixed-function accelerators can outperform QUETZAL in some cases, QUETZAL design offers three unique features. First, QUETZAL has the capability to accelerate various algorithms and steps within the genome sequence analysis pipeline, distinguishing itself from certain accelerators that may only support a subset of these steps or algorithms. Second, QUETZAL is cost-effective as it can be integrated within a general-purpose CPU pipeline with a small silicon area overhead. In contrast, the design and introduction of new domain-specific accelerators in the market incur high design and verification costs. Third, instead of designing algorithm-specific hardware, QUETZAL integrates primitive ISA instructions (*e.g.,* memory-indexed instructions and data format transformations) and their hardware acceleration support. These are broadly applicable to workloads even beyond the domain of genome sequence analysis (a design philosophy similar to NVIDIA's DPX instructions [63]) as discussed in Section III-E.

F. Accelerating other application domains with QUETZAL

We designed QUETZAL considering the general performance bottlenecks that prevent efficient vectorization of multiple genome sequence analysis applications. Such bottlenecks arise in many other applications. Therefore, the hardware of QUETZAL can be used to accelerate applications beyond genome sequence analysis. We demonstrate the generality of QUETZAL by accelerating two kernels out of the scope of genome sequence analysis. We evaluate the Sparse Matrix Vector (*SpMV*) multiplication and *histogram* calculation algorithms with the methodology proposed by Pavon et al. [92]. These algorithms target scratchpad-like hardware structures for vector architectures. We modify the aforementioned algorithms to use QUETZAL instructions. For *SpMV*, the algorithm stores segments from the input vector in QBUFFERs. All memory-indexed instructions are executed directly in QUETZAL using the qzmm instruction. We evaluate the *histogram* algorithm from Section III-E. Fig. 15.b depicts the average speedup for both algorithms. QUETZAL outperforms vectorized *SpMV* and *histogram* algorithms by 1.94× and 3.02×, respectively. We conclude that QUETZAL is not only (1) highly effective at improving the performance of multiple genome sequence analysis algorithms but also (2) a general solution capable of accelerating other application domains as well.

VIII. Related Work

We have comprehensively evaluated QUETZAL with several well-known edit distance approximation and alignment algorithms, including SneakySnake [18, 83], Needleman-Wunsch [22], Wavefront Algorithm [32, 33], and Bidirectional WFA [34, 35]. Our experiments demonstrate that QUETZAL significantly improves the performance of these algorithms.

On the software side, a large body of work proposes a wide range of exact and heuristic algorithms for pairwise sequence alignment (e.g., [24, 25, 27–31, 36, 38]). QUETZAL supports both exact and heuristic algorithms. While the performance of prior software approaches is limited by the underlying hardware, QUETZAL presents a hardware-software co-design that significantly improves performance while being programmable to support the emerging landscape of modern genome sequence analysis algorithms.

The use of graphics processing units (GPUs) for pairwise sequence alignment acceleration has gained significant attention in recent years (see, e.g., [38, 72–75, 121–123]). The parallelism and high memory bandwidth of GPUs make them an attractive hardware platform. However, as shown in our experiments, when processing long sequences, the size of the active working set limits the performance of GPUs. QUETZAL provides better performance when processing long sequences thanks to QBUFFERs that are specially designed to efficiently accelerate memory-indexed instructions.

On the hardware side, many domain-specific genome analysis accelerators have been proposed based on FPGA (e.g., [18, 19, 51, 124–127]) and application-specific integrated circuit (ASIC) (e.g., [36, 38, 43–45, 67, 68, 119]) designs. These accelerators exhibit high parallelism, throughput, and energy efficiency. However, they typically lack programmability, *i.e.,* cannot execute new or different algorithms than those targeted in their original design. This makes it inefficient to, for example, alternate between multiple algorithms (*e.g.,* edit distance approximation with SneakySnake and pairwise sequence alignment with WFA). Therefore, alternating between multiple algorithms with specialized accelerators requires constantly moving data between the accelerators and the host machine. In contrast, QUETZAL is highly flexible and does not require additional hardware changes to support a new algorithm, but only coding and recompilation. Moreover, QUETZAL is directly connected to the execution datapath, and thus does not require data offloading to operate.

Recently, Gu *et al.* proposed GenDP [67], an acceleration framework for DP algorithms. Similar to QUETZAL, GenDP aims to provide an efficient hardware accelerator capable of accelerating multiple algorithms. GenDP is a programmable accelerator that incurs high design and verification costs. In contrast, QUETZAL is a solution based on commercially available CPUs, which can be integrated into the core's vector datapath in a cost-effective manner.

IX. Conclusions

We propose QUETZAL, a universal approximate string matching (ASM) vector acceleration framework that supports a large number of state-of-the-art ASM algorithms. We first analyze the limitations of genome sequence alignment algorithms and propose novel vector instructions that address these limitations. We design a novel vector accelerator that implements these instructions. By integrating this cost-effective vector accelerator hardware and instructions with a general-purpose CPU's vector datapath, QUETZAL provides both efficiency and programmability that makes it relevant for speeding up modern and emerging genome sequence analysis workloads. We evaluate the effectiveness of QUETZAL using three use cases: sequence aligners, edit distance approximation, and a combination of both. We demonstrate that QUETZAL is an area- and power-efficient scratchpad-based implementation that can greatly accelerate a large number of ASM algorithms, supporting both short and long reads.

X. Acknowledgements

The authors would like to thank all of our anonymous reviewers for their valuable feedback, meticulous reviews and comments, which have allowed us to improve this work considerably. This work has been partially supported by the Spanish Ministry of Science and Innovation (PID2019-107255GB-C21 / AEI / 10.13039/501100011033). We acknowledge the generous gifts provided by our industrial partners, including IBM, Google, Huawei, Intel, Microsoft, and VMware. This research was partially supported by the EU Horizon project BioPIM (grant agreement 101047160), the AI Chip Center for Emerging Smart Systems Limited (ACCESS), the Swiss National Science Foundation (SNSF), Semiconductor Research Corporation (SRC), and the ETH Future Computing Laboratory (EFCL).

References

[1] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu *et al.*, "Personalized Copy Number and Segmental Duplication Maps Using Next-Generation Sequencing," *Nature Genetics*, 2009.

[2] E. A. Ashley, "Towards Precision Medicine," *Nature Reviews Genetics*, 2016.

[3] L. Chin, J. N. Andersen, and P. A. Futreal, "Cancer Genomics: From Discovery Science to Personalized Medicine," *Nature Medicine*, 2011.

[4] M. Flores, G. Glusman, K. Brogaard, N. D. Price, and L. Hood, "P4 Medicine: How Systems Medicine Will Transform the Healthcare Sector and Society," *Personalized Medicine*, 2013.

[5] G. S. Ginsburg and H. F. Willard, "Genomic and Personalized Medicine: Foundations and Applications," *Translational Research*, 2009.

[6] R. Langreth and M. Waldholz, "New Era of Personalized Medicine: Targeting Drugs for Each Unique Genetic Profile," *The Oncologist*, 1999.

[7] 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," *CSBJ*, 2022.

[8] H. Ellegren, "Genome Sequencing and Population Genomics in Non-Model Organisms," *Trends in Ecology & Evolution*, 2014.

[9] J. Prado-Martinez, P. H. Sudmant, J. M. Kidd, H. Li, J. L. Kelley, B. Lorente-Galdos, K. R. Veeramah, A. E. Woerner, T. D. O'connor, G. Santpere *et al.*, "Great Ape Genetic Diversity and Population History," *Nature*, 2013.

[10] A. Prohaska, F. Racimo, A. J. Schork, M. Sikora, A. J. Stern, M. Ilardo, M. E. Allentoft, L. Folkersen, A. Buil, J. V. Moreno-Mayar *et al.*, "Human Disease Variation in the Light of Population Genomics," *Cell*, 2019.

[11] M. J. Alvarez-Cubero, M. Saiz, B. Martínez-García, S. M. Sayalero, C. Entrala, J. A. Lorente, and L. J. Martinez-Gonzalez, "Next Generation Sequencing: An Application in Forensic Sciences?" *Annals of Human Biology*, 2017.

[12] C. Børsting and N. Morling, "Next Generation Sequencing and Its Applications in Forensic Genetics," *Forensic Science International: Genetics*, 2015.

[13] Y. Yang, B. Xie, and J. Yan, "Application of Next-Generation Sequencing Technology in Forensic Science," *Genomics, Proteomics & Bioinformatics*, 2014.

[14] W.-W. Liao, M. Asri, J. Ebler, D. Doerr, M. Haukness, G. Hickey, S. Lu, J. K. Lucas, J. Monlong, H. J. Abel *et al.*, "A Draft Human Pangenome Reference," *Nature*, 2023.

[15] M. Cagiada, S. Bottaro, S. Lindemose, S. M. Schenstrøm, A. Stein, R. Hartmann-Petersen, and K. Lindorff-Larsen, "Discovering Functionally Important Sites in Proteins," *Nature Communications*, 2023.

[16] K. M. Swenson, M. Marron, J. V. Earnest-DeYoung, and B. M. Moret, "Approximating the True Evolutionary Distance Between Two Genomes," *JEA*, 2008.

[17] G. Navarro, "A Guided Tour to Approximate String Matching," *CSUR*, 2001.

[18] M. Alser, T. Shahroodi, J. Gómez-Luna, C. Alkan, and O. Mutlu, "SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs and FPGAs," *Bioinformatics*, 2020.

[19] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: A Fast and Efficient Pre-Alignment Filter for Sequence Alignment," *Bioinformatics*, 2019.

[20] V. I. Levenshtein *et al.*, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," in *Soviet Physics Doklady*, 1966.

[21] "National Genomic Data Initiatives Review."

[22] S. B. Needleman and C. D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *JMB*, 1970.

[23] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *JACM*, 1974.

[24] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *JMB*, 1981.

[25] O. Gotoh, "An Improved Algorithm for Matching Biological Sequences," *JMB*, 1982.

[26] E. Ukkonen, "Algorithms for Approximate String Matching," *Inf. Control.*, 1985.

[27] R. Baeza-Yates and G. H. Gonnet, "A New Approach to Text Searching," *CACM*, 1992.

[28] S. Wu and U. Manber, "Fast Text Searching: Allowing Errors," *CACM*, 1992.

[29] G. Myers, "A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming," *JACM*, 1999.

[30] H. Suzuki and M. Kasahara, "Introducing Difference Recurrence Relations for Faster Semi-Global Alignment of Long Sequences," *BMC Bioinformatics*, 2018.

[31] H. Li, "Minimap2: Pairwise Alignment for Nucleotide Sequences," *Bioinformatics*, 2018.

[32] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast Gap-Affine Pairwise Alignment Using the Wavefront Algorithm," *Bioinformatics*, 2020.

[33] S. Marco-Sola, "Fast Gap-Affine Pairwise Alignment Using the Wavefront Algorithm," Available at https://github.com/smarco/WFA-paper, accessed: 2023-03-23.

[34] S. Marco-Sola, J. M. Eizenga, A. Guarracino, B. Paten, E. Garrison, and M. Moreto, "Optimal Gap-Affine Alignment in O (S) Space," *Bioinformatics*, 2023.

[35] Marco-Sola, "Optimal Gap-Affine Alignment," Available at https://github.com/smarco/BiWFA-paper, accessed: 2023-04-04.

[36] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand *et al.*, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*, 2020.

[37] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr, R. Ausavarungnirun, N. Vijaykumar, M. Alser, and O. Mutlu, "GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis," in *ASPLOS*, 2022.

[38] J. Lindegger, D. S. Cali, M. Alser, J. Gómez-Luna, N. M. Ghiasi, and O. Mutlu, "Scrooge: A Fast and Memory-Frugal Genomic Sequence Aligner for CPUs, GPUs, and ASICs," *Bioinformatics*, 2023.

[39] R. Rahn, S. Budach, P. Costanza, M. Ehrhardt, J. Hancox, and K. Reinert, "Generic Accelerated Sequence Alignment in SeqAn Using Vectorization and Multi-Threading," *Bioinformatics*, 2018.

[40] Y. Jararweh, M. Al-Ayyoub, M. Fakirah, L. Alawneh, and B. B. Gupta, "Improving the Performance of the Needleman-Wunsch Algorithm Using Parallelization and Vectorization Techniques," *Multim. Tools Appl.*, 2019.

[41] P. D. Vouzis and N. V. Sahinidis, "GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment," *Bioinformatics*, 2011.

[42] D. Senol 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," *ISCA*, 2022.

[43] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "GenAx: A Genome Sequencing Accelerator," *ISCA*, 2018.

[44] D. Fujiki, S. Wu, N. Ozog, K. Goliya, D. Blaauw, S. Narayanasamy, and R. Das, "SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space," *MICRO*, 2020.

[45] 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 *MICRO*, 2019.

[46] M. Alser, J. Rotman, D. Deshpande, K. Taraszka, H. Shi, P. I. Baykal, H. T. Yang, V. Xue, S. Knyazev, B. D. Singer *et al.*, "Technology Dictates Algorithms: Recent Developments in Read Alignment," *Genome Biology*, 2021.

[47] 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*, 2020.

[48] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.

[49] C. Yu, Y. Zhao, C. Zhao, H. Ma, and G. Wang, "DiagAF: A More Accurate and Efficient Pre-Alignment Filter for Sequence Alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2021.

[50] D. Castells-Rufas, S. Marco-Sola, J. C. Moure, Q. Aguado, and A. Espinosa, "FPGA Acceleration of Pre-Alignment Filters for Short Read Mapping With HLS," *IEEE Access*, 2022.

[51] G. Singh, M. Alser, D. S. Cali, D. Diamantopoulos, J. Gómez-Luna, H. Corporaal, and O. Mutlu, "FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications," *IEEE Micro*, 2021.

[52] H. Xin, J. Greth, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Shifted Hamming Distance: A Fast and Accurate SIMD-friendly Filter to Accelerate Alignment Verification in Read Mapping," *Bioinformatics*, 2015.

[53] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating Read Mapping With FastHASH," in *BMC Genomics*, 2013.

[54] 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*, 2017.

[55] Y.-J. Song, D. J. Ji, H. Seo, G. B. Han, and D.-H. Cho, "Pairwise Heuristic Sequence Alignment Algorithm Based on Deep Reinforcement Learning," *IEEE Open Journal of Engineering in Medicine and Biology*, 2021.

[56] Y. Sun and J. Buhler, "Choosing the Best Heuristic for Seeded Alignment of DNA Sequences," *BMC Bioinformatics*, 2006.

[57] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, "A Greedy Algorithm for Aligning DNA Sequences," *JCB*, 2000.

[58] K.-M. Chao, W. R. Pearson, and W. Miller, "Aligning Two Sequences Within a Specified Diagonal Band," *Bioinformatics*, 1992.

[59] A. Haghi, S. Marco-Sola, L. Alvarez, D. Diamantopoulos, C. Hagleitner, and M. Moreto, "An FPGA Accelerator of the Wavefront Algorithm for Genomics Pairwise Alignment," in *FPL*, 2021.

[60] PacBio, "PacBio," Available at https://www.pacb.com/technology/hifi-sequencing/how-it-works/, accessed: 2023-03-23.

[61] T. Hon, K. Mars, and G. Young, "Highly Accurate Long-Read HiFi Sequencing Data for Five Complex Genomes," *Scientific Data 7*, 2020.

[62] O. N. T. Update, "New Duplex Method for Q30 Nanopore Single Molecule Reads, PromethION 2, and More.(nd)," 2023.

[63] A. C. Elster and T. A. Haugdahl, "Nvidia Hopper Gpu and Grace Cpu Highlights," *Computing in Science & Engineering*, 2022.

[64] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia Tensor Core Programmability, Performance & Precision," in *IPDPSW*, 2018.

[65] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, 1966.

[66] Y.-L. Liao, Y.-C. Li, N.-C. Chen, and Y.-C. Lu, "Adaptively Banded Smith-Waterman Algorithm for Long Reads and Its Hardware Accelerator," in *ASAP*, 2018.

[67] Y. Gu, A. Subramaniyan, T. Dunn, A. Khadem, K. Chen, S. Paul, M. Vasimuddin, S. Misra, D. T. Blaauw, S. Narayanasamy, and R. Das, "GenDP: A Framework of Dynamic Programming Acceleration for Genome Sequencing Analysis," in *ISCA*, 2023.

[68] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally, "Darwin-WGA: A Co-Processor Provides Increased Sensitivity in Whole Genome Alignments With High Speedup," *HPCA*, 2019.

[69] H. Li, "Minimap2," Available at https://github.com/lh3/minimap2, accessed: 2023-04-04.

[70] J. Daily, "Parasail: SIMD C Library for Global, Semi-Global, and Local Pairwise Sequence Alignments," *BMC Bioinformatics*, 2016.

[71] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems," in *IPDPS*, 2019.

[72] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: Accelerating Smith-Waterman Protein Database Search by Coupling CPU and GPU SIMD Instructions," *BMC Bioinformatics*, 2013.

[73] N. Ahmed, T. D. Qiu, K. Bertels, and Z. Al-Ars, "GPU Acceleration of Darwin Read Overlapper for De Novo Assembly of Long DNA Reads," *BMC Bioinformatics*, 2020.

[74] NVIDIA, "NVBIO," https://github.com/NVlabs/nvbio, 2014.

[75] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, "GASAL2: A GPU Accelerated Sequence Alignment Library for High-Throughput NGS Data," *BMC Bioinformatics*, 2019.

[76] Q. Aguado-Puig, S. Marco-Sola, J. C. Moure, C. Matzoros, D. Castells-Rufas, A. Espinosa, and M. Moreto, "WFA-GPU: Gap-Affine Pairwise Alignment Using GPUs," *bioRxiv*, 2022.

[77] S. Park, H. Kim, T. Ahmad, N. Ahmed, Z. Al-Ars, H. P. Hofstee, Y. Kim, and J. Lee, "SALoBa: Maximizing Data Locality and Workload Balance for Fast Sequence Alignment on GPUs," in *IPDPS*, 2022.

[78] S. L. Amarasinghe, S. Su, X. Dong, L. Zappia, M. E. Ritchie, and Q. Gouil, "Opportunities and Challenges in Long-Read Sequencing Data Analysis," *Genome Biology*, 2020.

[79] C. Cheng, Z. Fei, and P. Xiao, "Methods to Improve the Accuracy of Next-Generation Sequencing," *Frontiers in Bioengineering and Biotechnology*, 2023.

[80] F. Minervini, O. Palomar, O. Unsal, E. Reggiani, J. Quiroga, J. Marimon, C. Rojas, R. Figueras, A. Ruiz, A. Gonzalez *et al.*, "Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications," *TACO*, 2023.

[81] Fujitsu, "A64FX Microarchitecture Manual," Available at https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.6.pdf.

[82] S. Marco-Sola, "WFA2-lib," Available at https://github.com/smarco/WFA2-lib, accessed: 2023-03-23.

[83] M. Alser, T. Shahroodi, J. Gómez-Luna, C. Alkan, and O. Mutlu, "SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs and FPGAs," Available at https://github.com/CMU-SAFARI/SneakySnake, accessed: 2023-03-23.

[84] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual." 2019.

[85] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual." 2023.

[86] ARM Ltd, "ARM," https://developer.arm.com/documentation/, accessed: 2022-01-15.

[87] Y. Kurt, Matallana-RamirezLilian, W. Kohlway, R. Whetten, and J. Frampton, "A Fast, Flexible and Inexpensive Protocol for DNA and RNA Extraction for Forest Trees," *Forest Systems*, 2020.

[88] N. Khiste and L. Ilie, "E-MEM: Efficient Computation of Maximal Exact Matches for Very Large Genomes," *Bioinformatics*, 2014.

[89] S. Giuliani, G. Romana, and M. Rossi, "Computing Maximal Unique Matches With the R-Index," *arXiv*, 2022.

[90] M. Chatzou, C. Magis, J.-M. Chang, C. Kemena, G. Bussotti, I. Erb, and C. Notredame, "Multiple Sequence Alignment Modeling: Methods and Applications," *Briefings in Bioinformatics*, 2016.

[91] A. Bayat, B. Gaëta, A. Ignjatovic, and S. Parameswaran, "Pairwise Alignment of Nucleotide Sequences Using Maximal Exact Matches," *BMC Bioinformatics*, 2019.

[92] J. Pavon, I. V. Valdivieso, A. Barredo, J. Marimon, M. Moreto, F. Moll, O. Unsal, M. Valero, and A. Cristal, "Via: A Smart Scratchpad for Vector Units With Application to Sparse Matrix Computations," in *HPCA*, 2021.

[93] Z. Istvan, L. Woods, and G. Alonso, "Histograms as a Side Effect of Data Movement for Big Data," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[94] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero, "VSR Sort: A Novel Vectorised Sorting Algorithm & Architecture Extensions for Future Microprocessors," in *HPCA*, 2015.

[95] S. S. Bagade and V. K. Shandilya, "Use of Histogram Equalization in Image Processing for Image Enhancement," *International Journal of Software Engineering Research & Practices*, 2011.

[96] K. Vij and Y. Singh, "Enhancement of Images Using Histogram Processing Techniques," *Int. J. Comp. Tech. Appl*, 2009.

[97] M. Yabuuchi, Y. Tsukamoto, M. Morimoto, M. Tanaka, and K. Nii, "20nm High-Density Single-Port and Dual-Port SRAMs With Wordline-Voltage-Adjustment System for Read/Write Assists," *IEICE Technical Report; IEICE Tech. Rep.*, 2015.

[98] 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 Bin Altaf, N. Vaish, M. Hill, and D. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, 2011.

[99] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The Gem5 Simulator: Version 20.0+," *arXiv*, 2020.

[100] T. Odajima, Y. Kodama, M. Tsuji, M. Matsuda, Y. Maruyama, and M. Sato, "Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications," in *CLUSTER*, 2020.

[101] E. Strohmaier, "TOP500 Supercomputer," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[102] T. Org, "Top500 The List," Available at https://www.top500.org/lists/top500/, accessed: 2023-03-23.

[103] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *JMB*, 1990.

[104] 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," *NAR*, 1997.

[105] H. Li, "KSW2," Available at https://github.com/lh3/ksw2, accessed: 2023-12-21.

[106] J. Daily, "Parasail: Pairwise Sequence Alignment Library," Available at https://github.com/jeffdaily/parasail, accessed: 2023-03-23.

[107] Ilumina, "Ilumina Sequencing Platforms," Available at https://www.illumina.com/systems/sequencing-platforms.html, accessed: 2023-03-23.
[108] E. Mardis, "DNA Sequencing Technologies: 2006-2016," *Nature Protocols 12*, 2017.
[109] D. of Computer Science at The University of Illinois at Urbana-Champaign, "BAliBase Benchmark Alignment Files," https://publish.illinois.edu/msaevaluation/balibase-benchmark-alignment-files/, accessed: 2023-06-25.
[110] Q. Aguado-Puig, "WFA-GPU," Available at https://github.com/quim0/WFA-GPU, accessed: 2023-12-21.
[111] A. Nauman, "GASAL2," Available at https://github.com/nahmedraja/GASAL2, accessed: 2023-12-21.
[112] D. Senol Cali, J. S. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Nanopore Sequencing Technology and Tools for Genome Assembly: Computational Analysis of the Current State, Bottlenecks and Future Directions," *Briefings in Bioinformatics*, 2019.
[113] S. Walia, C. Ye, A. Bera, D. Lodhavia, and Y. Turakhia, "TALCO: Tiling Genome Sequence Alignment Using Convergence of Traceback Pointers," in *HPCA*, 2024.
[114] H. Hyyrö, "Explaining and Extending the Bit-Parallel Approximate String Matching Algorithm of Myers," Citeseer, Tech. Rep., 2001.
[115] Synopsys, "Synopsys," https://www.synopsys.com/, accessed: 2022-12-21.
[116] E. Arima, Y. Kodama, T. Odajima, M. Tsuji, and M. Sato, "Power/performance/area Evaluations for Next-Generation HPC Processors Using the A64fx Chip," in *COOL CHIPS*, 2021.
[117] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, 1992.
[118] A. Stillmaker and B. Baas, "Scaling Equations for the Accurate Prediction of CMOS Device Performance From 180 Nm to 7 Nm," *Integration*, 2017.
[119] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A Genomics Co-Processor Provides Up to 15,000x Acceleration on Long Read Assembly," *ASPLOS*, 2018.
[120] A. Haghi, L. Alvarez, J. Front, J. M. De Haro Ruiz, R. Figueras, M. Doblas, S. Marco-Sola, and M. Moreto, "WFAsic: A High-Performance ASIC Accelerator for DNA Sequence Alignment on a RISC-V SoC," in *ICPP*, 2023.
[121] E. F. de Oliveira Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A. C. M. Melo, "CUDAlign 4.0: Incremental Speculative Traceback for Exact Chromosome-Wide Alignment in GPU Clusters," *IEEE Trans. Parallel Distrib. Syst.*, 2016.
[122] M. G. Awan, J. Deslippe, A. Buluc, O. Selvitopi, S. Hofmeyr, L. Oliker, and K. Yelick, "ADEPT: A Domain Independent Sequence Alignment Strategy for GPU Architectures," *BMC Bioinformatics*, 2020.
[123] Q. Aguado-Puig, S. Marco-Sola, J. C. Moure, D. Castells-Rufas, L. Alvarez, A. Espinosa, and M. Moreto, "Accelerating Edit-Distance Sequence Alignment on GPU Using the Wavefront Algorithm," *IEEE Access*, 2022.
[124] K. Benkrid, Y. Liu, and A. Benkrid, "A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," *IEEE Transactions on VLSI Systems*, 2009.
[125] J. Hoffmann, D. Zeckzer, and M. Bogdan, "Using FPGAs to Accelerate Myers Bit-Vector Algorithm," *MEDICON*, 2016.
[126] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, "FPGASW: Accelerating Large-Scale Smith–Waterman Sequence Alignment Application With Backtracking on FPGA Linear Systolic Array," *Interdiscip Sci*, 2018.
[127] O. Mutlu and C. Firtina, "Accelerating Genome Analysis via Algorithm-Architecture Co-Design," in *DAC*, 2023.