Faster and Cheaper: Pushing the Sequence Alignment Throughput with Commercial CPUs

Zhonghai Zhang
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
zhangzhonghai@ict.ac.cn

Yewen Li*
The Hong Kong University of Science and Technology
Hong Kong SAR, China
yewenli@ust.hk

Ke Meng
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
mengke@ict.ac.cn

Chunming Zhang
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
Phil Rivers Technology Ltd
Beijing, China
zhangchunming@ncic.ac.cn

Guangming Tan*
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
tgm@ict.ac.cn

Abstract

This paper proposes FastAlign, a faster, cheaper, and practical end-to-end solution for sequence alignment using commercial CPUs. It introduces two key innovations: a multi-stage seeding algorithm that improves search performance while maintaining low memory consumption, and an intra-query parallel seed-extension algorithm that eliminates redundancy and increases SIMD utilization. Evaluation results show that FastAlign achieves 2.27× ~ 3.28× throughput speedup and 2.54× ~ 5.65× cost reduction compared to state-of-the-art CPU and GPU baselines while guaranteeing 100% identical output to the de facto software BWA-MEM. FastAlign is open-sourced at https://github.com/zzhofict/BWA-FastAlign.git.

CCS Concepts: • **Applied computing** → **Computational genomics**.

Keywords: Computational Genomics, Next-Generation Sequencing, Sequence Alignment, Parallel System

*Corresponding Authors: Yewen Li and Guangming Tan.

This work is licensed under a Creative Commons Attribution 4.0 International License.

PPoPP '26, Sydney, NSW, Australia
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2310-0/2026/01
https://doi.org/10.1145/3774934.3786421

ACM Reference Format:
Zhonghai Zhang, Yewen Li, Ke Meng, Chunming Zhang, and Guangming Tan. 2026. Faster and Cheaper: Pushing the Sequence Alignment Throughput with Commercial CPUs. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26), January 31 – February 4, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3774934.3786421

1 Introduction

Over the past two decades, genome analysis, an interdisciplinary technology combining biology and computer science, has gradually matured and has been applied in various fields, *e.g.,* genetics [53], epidemiology [39], and public health [37]. Sequencing machines cannot directly generate biologically complete, contiguous sequences of an entire DNA strand. Instead, they randomly sample short fragments from the original DNA molecule and produce sequencing data, *a.k.a.* **reads**. These reads are then mapped to a **reference** genome through the process of sequence alignment. Scientists can identify potential genetic variations by comparing reads that are mapped to the same locations during downstream analysis. Today, genome analysis has expanded from research to personal medical applications, generating exabytes of genome data each year [24]. The pursuit of faster and cheaper solutions for genome analysis is becoming imperative.

While string matching is a classical computational problem [13, 22, 27, 41, 50, 51], the specific properties of genomic data render sequence alignment the primary bottleneck in genomic analysis. Specifically, sequence alignment accepts millions of reads generated by sequencers and a reference

genome as input. The objective is to align these reads to a reference genome by identifying positions yielding the highest matching scores. The output comprises detailed alignment records for each read, including genomic coordinates and base-level matching information. Moreover, the presence of genetic mutations, structural variations, and sequencing errors significantly complicates the alignment task.

For predominant next-generation sequencing technology, BWA-MEM [30] is the de facto standard sequence alignment tool. It follows a seed-and-extend paradigm which has two main phases: the **seeding phase** and the **seed-extension** phase [28–30]. In the seeding phase, the core algorithm is FM-Index search [14], which performs exact matching to produce a set of seeds indicating possible alignment positions of the original read sequence. BWA-MEM2 [52] trades off the base-occurrence counting frequency with more memory consumption. Moreover, several studies [23, 47] introduce new seeding data structures (*i.e.,* hash-based structure and learned index) and employ a significant amount of memory to reduce irregular memory accesses. In the seed-extension phase, the core algorithm is Banded Smith-Waterman (BSW) [9], which extends these seeds into potential alignments. Recently, several studies [40, 52] leverage CPU SIMD or GPU to perform parallel computations for the computation of the compute-bound score matrices.

The development of new hardware and architectures has shifted research focus toward optimizing and accelerating specific computational kernels, state-of-the-art (SOTA) works deliver significant performance improvements in the seeding or seed-extension phase [2, 4, 12, 17–20, 23, 30, 33, 34, 40, 48, 49, 52, 54]. However, they suffer from the following issues from both algorithm and data perspectives: First, the current seeding solutions suffer from **low memory efficiency**. The traditional FM-Index algorithm has a low memory footprint, but its base-by-base search fundamentally limits performance [14, 30]. Several works based on hash indexes and learning indexes enable multi-base searches, yet their memory consumption can reach hundreds of gigabytes [18, 23, 47]. Moreover, search performance does not improve linearly with increasing memory. For instance, ERT-BWA-MEM2 [47] achieves 3× search performance while consuming 6× the memory footprint of BWA-MEM [30]. Second, existing seed-extension designs exhibit **high computation redundancy**. Due to load imbalance caused by substantial uneven sequence characteristics (*e.g.,* query length and bands), the existing CPU and GPU works [21, 23, 52] only perform well when computation workloads of each sequence are close. For instance, the varying sequence lengths resulted in SIMD utilization of only about 16.3% in BWA-MEM2 [52].

Moreover, from the practical perspective, SOTA solutions also lack a systematic approach that **balances performance and cost**, deployable on readily available computing resources. While running carefully designed alignments on high-end GPUs/FPGAs presents a promising direction, the costs of deploying them in real-world production environments can often be daunting. For example, the seed-extension performance on 16 AMD vCPU cores is 54% slower than that on a top-tier V100 GPU, while the CPU platform costs only a tenth of the GPU platform [10, 40, 52]. Those issues raise a natural question: *Can we fully leverage the commercial CPUs provided by cloud vendors to achieve better performance than SOTA solutions with low cost?*

FastAlign. In this paper, we propose FastAlign, a practical, faster, and cheaper end-to-end solution for accelerating next-generation sequencing alignment on CPU servers. FastAlign revitalizes the traditional CPU-based alignment pipeline BWA-MEM [30] by carefully managing memory consumption and resource utilization, fully exploiting the performance potential of low-cost CPU machines, with the following optimizations: (i) For the seeding phase, we propose a multi-stage seeding algorithm, *i.e.,* a hybrid indexing approach that leverages sequence features to enhance both search performance while maintaining low memory consumption. (ii) For the seed-extension phase, we propose an intra-query parallel algorithm to eliminate computation redundancy. Specifically, we propose the dynamic pruning strategy to skip zero alignment and the sliding window mechanism to reduce the costly memory gather operations. In addition, FastAlign ensures 100% identical output to the de facto standard software BWA-MEM, making it a SOTA version of BWA-MEM and BWA-MEM2. More specifically, our specific contributions are listed as follows:

(1) We provide a new open-sourced sequence alignment system to push the SOTA, offering a CPU-based accelerated solution for the end-to-end pipeline.

(2) We propose a novel multi-stage seeding algorithm to leverage the characteristics of the genome data, which achieves an 18.92× improvement in memory efficiency.

(3) We propose an intra-query algorithm to parallelize the seed-extension phase, achieving 3.45× increase in SIMD utilization under diverse query lengths in real-world scenarios.

(4) FastAlign can provide 2.27× ∼ 3.28× throughput speedup over SOTA CPU and GPU baselines with only 19% ∼ 81% of their costs.

2 Background and Motivation

2.1 A Quick Primer on Sequence Alignment

Genome sequence alignment, *a.k.a.* gene sequence mapping, identifies the best matching positions for reads within an extensive reference DNA sequence. In other words, this problem is equivalent to finding the optimal match for a given substring within a long string consisting only of A, C, G, and T. Alignment software typically addresses this problem using a seed-and-extend paradigm. The seeding phase finds exact matches of substrings from a read within the reference

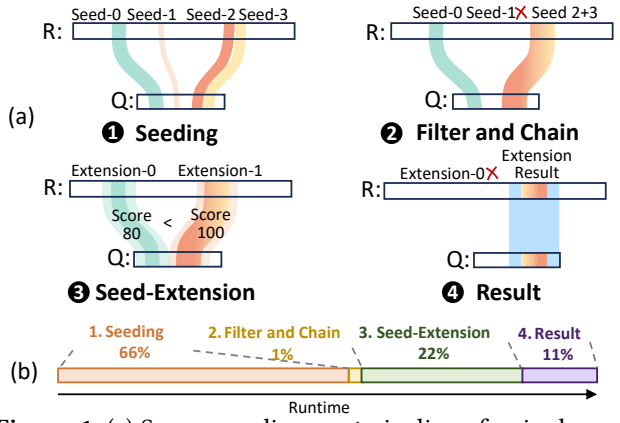

Figure 1. (a) Sequence alignment pipeline of a single read. (b) Execution time breakdown of BWA-MEM [30].

genome. The seed-extension phase extends these seed positions from both ends until the entire read is covered, and the best position is reported as the alignment result.

Execution Pipeline. Figure 1(a) illustrates the sequence alignment pipeline [3, 5, 32, 36, 43]:

(1) *Seeding.* The pipeline accepts a read as input and matches its substring to the reference as long as possible using an exact matching algorithm (*e.g.*, FM-Index search algorithm [14]). We obtain some substrings (*i.e.,* seeds), which are denoted as Seed-0, Seed-1, Seed-2, and Seed-3 in Figure 1-❶.

(2) *Filter and Chain.* It then filters out short seeds and creates chains by grouping them together. As shown in Figure 1-❷, Seed-1 is filtered out, Seed-2 and Seed-3 are chained to form Seed 2+3.

(3) *Seed-Extension.* Later, it extends all the seeds (*i.e.,* Seed-0, Seed 2+3) from both ends using approximate matching algorithms. After this step, we obtain the alignment candidates denoted as Extension-0 and Extension-1 in Figure 1-❸.

(4) *Result.* As shown in Figure 1-❹, the pipeline finally generates the result with the highest-scoring extension (*i.e.,* Extension-1).

Execution Breakdown. Figure 1(b) illustrates the execution time breakdown of the standard sequence alignment tool, BWA-MEM [30]. We can find that the seeding and seed-extension phase dominate the overall time, accounting for 66% and 22%, respectively. They are also the core of the entire pipeline optimization, and we will subsequently introduce the algorithms of these two phases in detail.

2.2 Algorithm Details

This section introduces the **FM-Index algorithm** and **Smith-Waterman algorithm**, which are used in the seeding and seed-extension phases, respectively. We will first present the details of the algorithms and then analyze them.

FM-Index Algorithm for Seeding. The FM-Index is a highly compressed data structure built from a reference sequence, which enables precise seed searching with $O(n)$

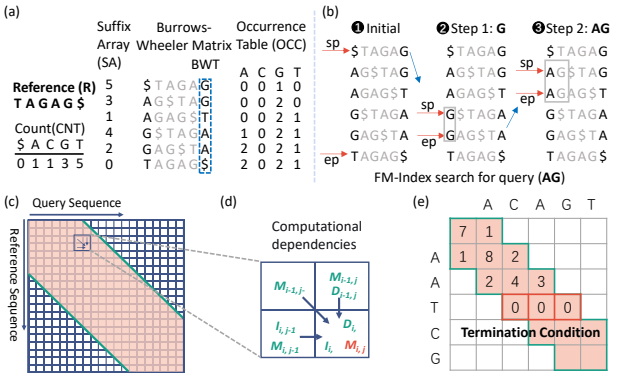

Figure 2. (a) FM-Index data structure for reference R. (b) Searching for query AG using the FM-Index. (c) Execution flow of Banded Smith-Waterman (BSW). (d) Computational dependencies of BSW. (e) Execution flow of BSW with early termination in BWA-MEM [30].

time complexity. As illustrated in Figure 2(a), it consists of four components:

(1) *CNT.* The count table, which indicates the cumulative number of all bases that are alphabetically smaller than a given base.

(2) *SA.* The suffix array, which contains the positions of each suffix in the reference.

(3) *BWT.* The Burrows-Wheeler Transform [6], which is the last column of the Burrows-Wheeler Matrix for the reference.

(4) *OCC.* The occurrence table, which denotes the number of occurrences of a given base in *BWT* before a certain index.

Listing 1. FM-Index based seeding procedure.

```
1  Procedure FM_index_seeding(query):
2      // Input: input sequence query
3      // Output: reference position match_positions
4
5      // 1. Search the matching range in suffix array
6      sp = 0; ep = reference_len
7      while sp <= ep do
8          if reach the end of query then break
9          c = get_next_base(query)
10         sp = CNT[c] + OCC[sp][c]
11         ep = CNT[c] + OCC[ep][c]
12     end
13
14     // 2. Get matching positions via the suffix array
15     for i := sp to ep do
16         match_positions[i] = SA[i]
17     end for
18     return match_positions
19 End Procedure
```

Listing 1 presents the seeding process based on the FM-Index algorithm. The first part (Line 5 ~ Line 12) processes each base to determine the matching range of the query within the suffix array. The second part (Line 15 ~ Line

17) locates all matching positions of the query within the reference genome via the suffix array.

We explain this process using the specific example shown in Figure 2(b). ❶ Set the upper index (*sp*) to the first row in the *BW-Matrix* and the lower index (*ep*) to the last row (*i.e.,* (0, 5)). ❷ Use the *CNT* value for *G* along with the corresponding values for *sp* and *ep* in the *OCC* to narrow down the range (*i.e.,* (3, 4)). ❸ Then extend the search with *A* to obtain the resulting range in the *SA* (*i.e.,* (1, 2)). Finally, we can use the *SA* to find the positions of *AG* in the reference (*i.e.,* 1 and 3). We should notice that the FM-index do not store all the *OCC* values. It stores *OCC* at sampled intervals. Entries between samples are regenerated during query processing through counting the bases (*base_count*) in the *BWT* string.

$$\begin{aligned} I_{i,j} &= max\{M_{i,j-1} - o - e, I_{i,j-1} - e\} \\ D_{i,j} &= max\{M_{i-1,j} - o - e, D_{i-1,j} - e\} \\ M_{i,j} &= max\{0, I_{i,j}, D_{i,j}, M_{i-1,j-1} + s(R[i], Q[j])\} \end{aligned} \quad (1)$$

Smith-Waterman Algorithm for Seed-Extension. The Smith-Waterman algorithm is a dynamic programming algorithm designed to find the optimal match for two strings (*i.e.,* query and reference) by considering edit operations such as mismatching, insertion, and deletion [31, 46]. Figure 2(d) presents the computational dependencies to calculate the score for a specific cell. Equation 1 shows the recursion to compute a $cell_{i,j}$ of score matrix M. Where I means insertion, D means deletion, o means open penalty, e means extending penalty. s means matching score of R at i and Q at j. M is the final score matrix.

In practice, the Banded Smith-Waterman algorithm [9] is adopted to reduce computational complexity. As shown in Figure 2(c), it adds a banded restriction and only computes near-diagonal elements within the band. BWA-MEM employs the Banded Smith-Waterman (BSW) algorithm to reduce computational intensity by incorporating band constraints into the traditional Smith-Waterman calculation area, as shown in Figure 2(c). It leverages the characteristic that the maximum alignment score typically occurs within a diagonal band. Moreover, BWA-MEM further reduces computational overhead by introducing additional termination conditions. Figure 2(e) demonstrates a concrete example of this process: with an initial score set to 8, the calculation terminates immediately when all values in a row drop to zero, ensuring that no zero-value cells occur along the maximum-score path.

2.3 Motivation: Cost-Efficient Solution

Table 1 presents the optimization efforts for the BWA-MEM pipeline. For the seeding phase, BWA-MEM2 [52] is based on the FM-Index algorithm, which reduces base-occurrence counting by increasing the memory footprint of the raw BWT sequence (*e.g.,* 16 GB). ERT-BWA-MEM2 [47] and BWA-MEME [23] propose multi-base lookups based on enumerated radix trees and learned indices to alleviate irregular memory access, respectively. While these methods effectively reduce the number of memory accesses, they have a large memory footprint (*e.g.,* ~128 GB). For the seed-extension phase, BWA-MEM [30] incorporates the BSW algorithm and early termination strategies, yet it lacks support for inter-query or intra-query parallelism. Although BWA-MEM2 [52], ERT-BWA-MEM2 [47], BWA-MEME [23], and BWA-MEM-SCALE [25] introduce inter-query parallelism, they exhibit low SIMD utilization when processing sequences of highly variable lengths. Several GPU-based heuristic systems [1, 7, 38, 40] achieve high performance but can not guarantee 100% identical alignment output with BWA-MEM.

Table 1. Comparison of sequence alignment designs.

Design	Dev	Seed	Ext	Perf	Cost
BWA-MEM2 [52]	CPU	✓	✓	2.95h	~$23
BWA-MEME [23]	CPU	✓	×	2.44h	~$25
ERT-BWA-MEM2 [47]	CPU	✓	×	2.42h	~$25
SeedEx [18]	FPGA	×	✓	1.89h	~$34
BWA-GPU [40]	GPU	×	✓	1.91h	~$36
FastAlign	CPU	✓	✓	0.83h	~$7

Dev: Target device; **Seed**: Optimization for the seeding phase; **Ext**: Optimization for the seed-extension phase; **Perf**: Execution time for NA12878 dataset [15]; **Cost**: Cost for NA12878 dataset. The CPU and GPU instances are detailed in Section 6.1 and the FPGA platform uses the f1.2xlarge instance in the Amazon AWS Hong Kong region [45].

Many works do not pay enough attention to holistic optimization for end-to-end pipelines and prefer higher performance at the cost of more resources. BWA-MEM-SCALE [25] introduces more memory consumption (*e.g.,* ~30% for BWA-MEM-SCALE), and BWA-GPU [40] offloads tasks to the GPU for overlapping computation and read/write. In terms of cost, CPU platforms have a cheap cost but long execution time [23, 30, 47, 52], while GPU [1, 7, 38, 40] and FPGA [18] platforms have shorter execution time but the cost of the platform is high. To be more specific, the cost of previous works ranges from $20 to $40 to process a human genome dataset, hindering the development of genome analysis. We present FastAlign, aiming to provide a faster and cheaper solution for sequence alignment pipelines on CPUs.

2.4 Challenges

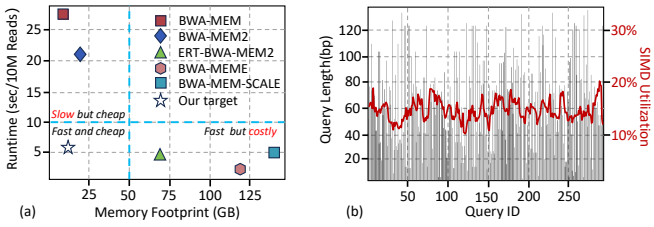

Figure 3. (a) Trade-off between memory consumption and seeding performance. (b) Length distribution of 300 continuous queries v.s. SIMD utilization in BWA-MEM2 [52].

To achieve our goals, we face the following challenges in FastAlign's design and implementation:

Challenge-1: Trade-off Between Memory Consumption and Searching Performance. As shown in Figure 3(a),

recent works have proposed new seeding data structures (*e.g.,* enumerated radix trees in ERT-BWA-MEM2 [47] and learned indices in BWA-MEME [23]) for better performance. By building a hash table for the reference genome, these works can achieve multi-base lookup instead of one-base. The drawback of these works is the large memory footprint since the hash table has a minimal memory footprint (about 64 GB) if they want to have performance gains. Thus, a good trade-off between memory consumption and search performance has not been well-established in prior research, and this is the aim of our work.

Challenge-2: Low Parallel Efficiency and Load Imbalance. Figure 3(b) illustrates the length distribution for the continuous queries in the seed-extension phase. We observe that the shortest query length is less than 5 bp (base pairs), while the longest exceeds 100 bp, and significant length variation exists among adjacent queries. Previous works [23, 47, 52] adopt an inter-query algorithm for the BSW kernel in the seed-extension phase. Specifically, each vector lane computes the matching scores for different queries. However, in the same vector lane, the shortest ones need to wait for the longest ones to complete, generating significant load imbalance. This will affect the parallelism efficiency and the seed-extension performance.

3 Seeding Phase

The seeding phase is inherently memory-intensive; our objective is to optimize seeding efficiency while maintaining low memory consumption, thereby improving memory efficiency to reduce server infrastructure costs.

3.1 Opportunity: Hybrid Indexing

As shown in Figure 4(a), we performed statistical analyses on the human genome to characterize the relationship between seed lengths and their hits in the reference genome. The results indicate that the number of hits for a given seed decreases exponentially as the seed length increases. Beyond a critical threshold (*e.g.,*, 29 bp), matches become exceedingly sparse (~2 loci). Current seeding algorithms do not fully exploit this distributional property, highlighting an opportunity to design more efficient indexing strategies informed by genomic sequence statistics.

We evaluated three indexing methodologies, Kmer-Index, FM-Index, and Direct-Index, across a range of seed lengths and match densities using human genome data, with memory efficiency metrics presented in Figure 4(b).

Kmer-Index: It enumerates all possible k-length subsequences composed of A, C, G, T, storing their match positions in a hash table. We can retrieve all the hits in the reference for a seed of length no greater than k with a single memory access. The key advantages include computational efficiency via constant-time lookup complexity, and the primary limitations involve memory requirements that scale

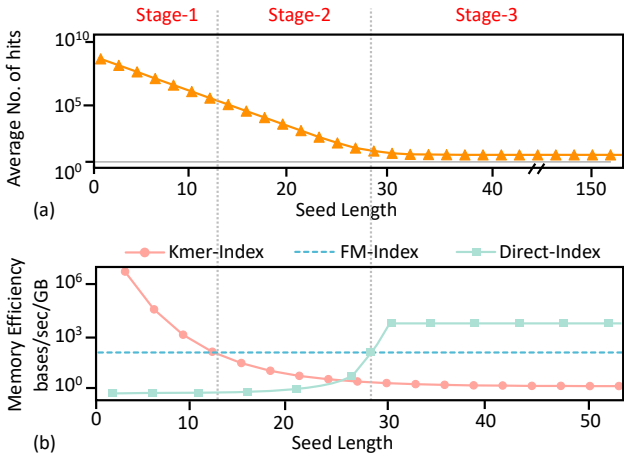

Figure 4. (a) Average number of hits across seed length. (b) Memory efficiency of seeding using various seeding structures across seed length. (Memory efficiency is defined as processed bases per second per GB of memory).

exponentially with k-mer[1]. At shorter seed lengths ($k < 15$ bp), this method empirically demonstrates superior memory efficiency relative to FM-index.

FM-Index: As described in (Section 2), it employs a compressed Burrows-Wheeler Transform (BWT) with fixed memory overhead (dependent solely on reference genome size). Its linear alignment complexity delivers stable memory efficiency across seed lengths.

Direct-Index: It performs base-wise comparisons at each candidate genomic locus until base discordance is encountered, thereby identifying maximal exact matches. Its time complexity scales as $O(hn)$, where h denotes candidate loci count and n represents seed length, while maintaining constant memory overhead proportional to reference genome size. As shown in Figure 4(b), the computational efficiency exhibits an inverse relationship with match density, outperforming the FM-Index when matches are sparse.

Figure 4 illustrates that a hybrid staged indexing framework, dynamically switching between optimal strategies at different phases of seeding, can exploit genomic data characteristics (*e.g.,* locus sparsity at longer seed lengths) to maximize memory efficiency. This adaptive approach tailors computational workflows to the statistical properties of the target genome and motivated us to divide the seeding process into a multi-stage search process, employing different strategies at each stage.

Listing 2 illustrates the details of our multi-stage seeding algorithm. The basic idea is to select the optimal search algorithm at different stages based on data characteristics

[1]Note that k-mer indices grow exponentially if using a dense k-mer index. When k is small(<15), storing all possible k-mers consecutively (with a complexity of $O(4^k)$, *i.e.,* A, C, G, T) results in very few invalid indices. This is because almost all such k-mers have matches in the reference.

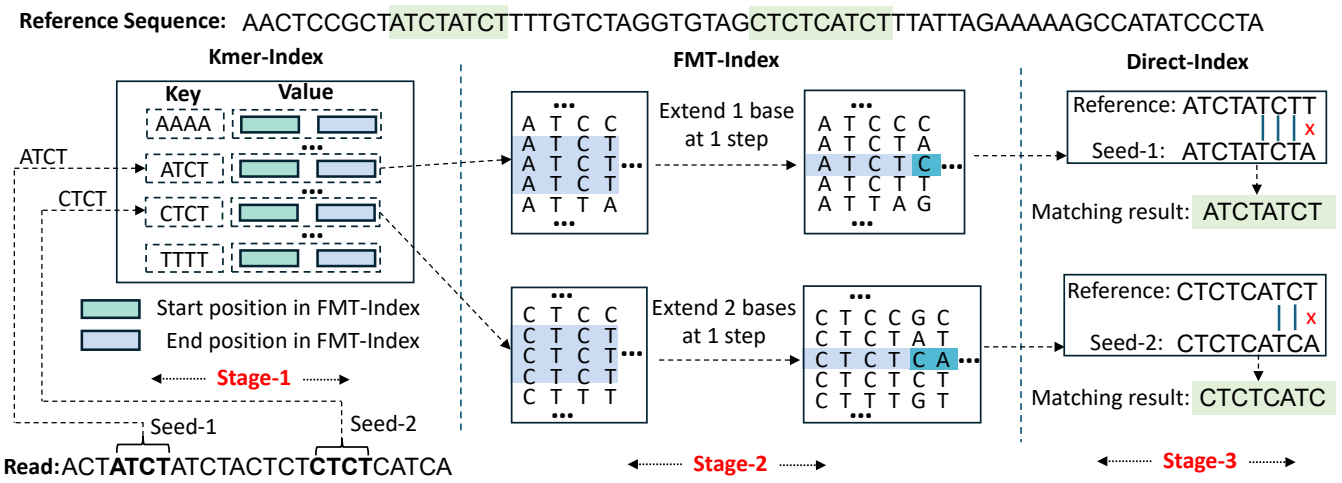

Figure 5. Toy example of the three-stage searching process for identifying seeds of input reads.

and algorithm efficiency. We will describe the entire process of multi-stage seeding in detail with a concrete example in Section 3.2. And we will present the key algorithm *Get_occurrence* for the FMT-Index in Section 3.3, detailing how the FMT-Index reduces non-contiguous memory access and improves computational intensity.

3.2 Multi-Stage Seeding with Hybrid Index

Figure 5 illustrates a concrete implementation of the multi-stage seeding strategy. We partition the seeding process into three discrete phases based on seed length and corresponding match frequency.

Stage-1: Kmer-Index. We pre-compute the Suffix Array (*SA*) intervals for all possible sequences of length k and store the results. When searching for seeds, we query the Kmer-Index to directly retrieve its corresponding *SA* interval for the first k bases. This approach circumvents the FM-Index's requirement for k sequential base extensions, thereby eliminating k random memory accesses and repeated *base_count* operations. As shown in Figure 4(b), storage requirements for full k-length hash indices grow exponentially with increasing k, causing memory efficiency to plummet. We set $k = 14$, an empirically optimized threshold where hash-based indexing consistently outperforms FM-Index in memory efficiency.

Stage-2: Enhanced FM-Index. Building upon the *SA* intervals obtained in Stage-1, seed length now progresses beyond the Kmer-Index's effective range. Given the high match density observed at this stage, FM-Index emerges as the optimal indexing strategy for memory efficiency. To mitigate performance penalties from irregular memory access patterns and high computational intensity, we introduce an enhanced FM-Index variant: the FMT-Index. This novel structure incorporates an n-step prefetching mechanism that maintains a low memory footprint and reduces per-access computational workload. Section 3.3 details the implementation and its performance gains.

Listing 2. Optimized multi-stage seeding procedure.

```
Procedure Multi_stage_seeding(q, len):
    // Input: query sequnce q and its length len
    // Output: matching results for query q

    // 1. Kmer-Index search
    key = compute_array_offset(q, 14)
    seeds_kmer = Kmer_Index[key]

    // 2. FMT-Index search
    sp = seeds_kmer.occ_idx
    ep = sp + seeds_kmer.occ_range
    for i := 14 to len Step 2 do
        o1, to1 = Get_occurrence(q[i], q[i+1], sp)
        o2, to2 = Get_occurrence(q[i], q[i+1], ep)
        sp = FMT_Index.CNT[q[i]] + o1
        ep = FMT_Index.CNT[q[i]] + o2
        if ep - sp <= 2 then
            query_pos = i; break
        end if
        sp = FMT_Index.CNT[q[i]][q[i+1]] + to1
        ep = FMT_Index.CNT[q[i]][q[i+1]] + to2
        if ep - sp <= 2 then
            query_pos = i + 1; break
        end if
    end for

    // 3. Direct-Index search
    for i := sp to ep do
        ref_pos = FMT_Index.SA[i]
        ref = Direct_Index[ref_pos]
        match_len = match_each_base(q, ref)
        // Store the seeding results
        pack_results(ref_pos, match_len)
    end for
End Procedure
```

Stage-3: Direct-Index. When a seed reaches lengths where its match density drops below critical thresholds, direct base-wise comparison between the seed sequence and corresponding reference genome segments achieves superior memory efficiency. Leveraging the positional catalog of all reference

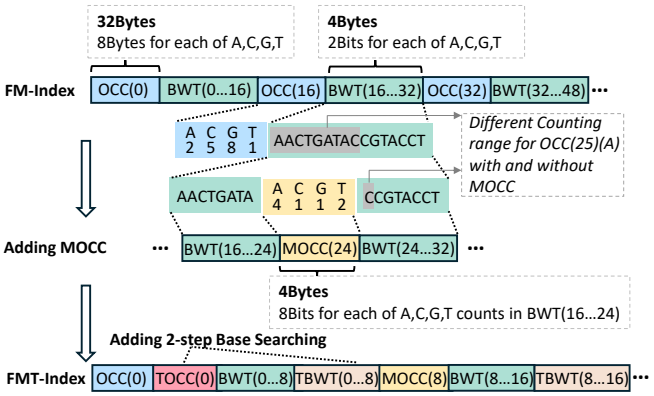

Figure 6. FMT-Index is constructed by incorporating *MOCC* and 2-step base searching into the FM-Index.

matches obtained in Stage-2, the direct index approach processes residual bases through exhaustive base comparison. This eliminates non-contiguous memory access patterns inherent to FM-Index-based methods, therefore achieving better performance.

Listing 3. Getting base occurrence with FMT-Index.

```
1  Procedure Get_occurrence(b1, b2, idx):
2      // Input: base b1, base b2, Occurrence index idx
3      // Output: Match positions occ and tocc
4
5      // 1. Calculate the closest stored OCC index
6      i = idx / OCC_SAMPLING_INTERVAL
7      ai = idx % OCC_SAMPLING_INTERVAL
8
9      // 2. calculate the closest stored MOCC index
10     j = ai / MOCC_SAMPLING_INTERVAL
11     aj = ai % MOCC_SAMPLING_INTERVAL
12
13     // 3. Get b1, b2 count in aj area
14     cnt1 = base_count(TBWT[i][j][0..aj], b1)
15     cnt2 = base_count(TBWT[i][j][0..aj], b1, b2)
16
17     // 4. Calculate OCC and TOCC at idx
18     occ = OCC[i][b1] + MOCC[i][j][b1] + cnt1
19     tocc = TOCC[i][b1][b2] + MOCC[i][j][b1][b2] + cnt2
20
21     return (occ, tocc)
22 End Procedure
```

3.3 Reducing Counting Operations and Irregular Memory Accesses with FMT-Index

Due to the large sampling interval in the FM-Index, reconstructing the *OCC* requires extensive counting operations. However, reducing the sampling interval would exponentially increase memory consumption. To address this issue, we propose a novel data structure, the FMT-Index, integrated with a two-level sampling technique. We will describe the usage process of the FMT-Index in detail with reference to Listing 3 and Figure 6. As shown in Figure 6, we insert the *MOCC* within the *BWT* to record the cumulative counts of bases in the adjacent left-side *BWT* string. Since the length of

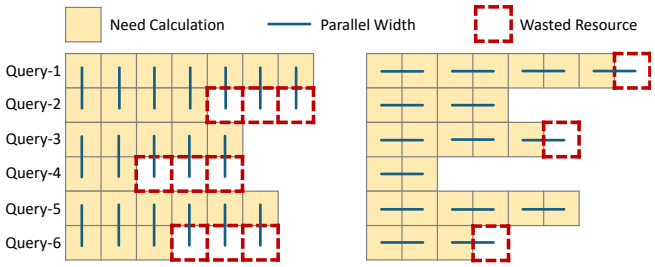

Figure 7. Inter-query and intra-query parallelism.

the *BWT* string between two *OCC* sampling points is within 256, the cumulative count for each base can be represented using 8-bit data. This allows us to achieve the effect of doubling the *OCC* sampling density with only 1/8 additional memory overhead. The only cost is one more addition operation to access the *MOCC*, while reducing the counting range by half. As illustrated in Figure 6, when calculating *OCC*(25)(A), the original FM-Index would need to count the number of 'A's in the *BWT*(16...25) (9 bases). With the *MOCC*, we only need to count the 'A's in *BWT*(24...25) (1 base, Line 14), and can then derive that *OCC*(25)(A) equals 6. Moreover, we introduce enhancements to the *n*-step [8] approach with *n* set to 2, retaining both 1-step data (*OCC*, *BWT*) and 2-step data (*TOCC*, *TBWT*) in contiguous memory space. This allows simultaneous expansion of 1 and 2 bases (see Stage-2 in Figure 5) with a single memory access (Line 18 ~ Line 19), reducing irregular memory access operations by about 50%.

4 Seed-Extension Phase

The seed-extension phase employs the BSW algorithm, which is an inherently compute-bound procedure. Our objective is to provide a new parallelism approach, which eliminates the inherent load imbalance and maximizes computational efficiency within existing infrastructure constraints.

4.1 Opportunity: Intra-Query Parallelism

In the seeding phase, approximately 20-30 seeds are generated for each read. In the seed-extension phase, the sequences (queries) at both ends of these seeds are matched against the reference genome to obtain the highest matching score and the corresponding match positions. We discovered that: (1) these queries show significant variation in length, and (2) the vast majority of queries are mismatched with their corresponding references, resulting in the matching score rapidly dropping to zero. BWA-MEM incorporates an early termination strategy, but it only implements a basic row-wise scalar algorithm, failing to leverage CPU SIMD parallel instructions. BWA-MEM2 employs an inter-query strategy that packs multiple queries into SIMD vectors. As shown in Figure 7, due to the data characteristics described above, this approach introduces substantial load imbalance. We designed a SIMD-accelerated intra-query algorithm that operates along the anti-diagonal direction to address this issue. And based on this, we further proposed a dynamic pruning strategy along the anti-diagonal direction to reduce the search space, while

introducing a sliding window mechanism to eliminate redundant computations, thereby accelerating the seed-extension phase.

Listing 4 depicts the details of our intra-query extension algorithm. The basic idea is to perform computations along the anti-diagonal direction, so that the elements processed simultaneously within SIMD have no data dependencies and can be parallelized. Then, we accelerate this process by reducing unnecessary calculations. We will provide a detailed introduction to the optimization scheme in Section 4.2 and Section 4.3 .

Listing 4. Optimized intra-query extension with SIMD.

```
1  Procedure Intra_query_extension(q, l_q, r, l_r):
2      // Input: query q, query length l_q, r: reference,
         l_r: reference length
3      // Output: match score max_score, match position
         max_position
4
5      // 1. Compute anti-diagonal scores
6      w = SIMD_WIDTH; j_b = 0; j_e = l_q
7      win[0..w] = 0; l_m[0..l_q] = c_m[0..l_q] = 0
8      for i := 0 to l_q + l_r do
9          win[0..l_q] = 0
10         for j := j_b to j_e Step w do
11             ms[j..j+w] = match(q[i..i-w], r[j..j+w])
12             ms[j..j+w] += l_m[j-1..j+w-1]
13             update_insertion(ins[j..j+w], ms[j..j+w])
14             update_deletion(del[j..j+w], ms[j..j+w])
15             c_m[j..j+w] = max(ms[j..j+w],
16                 ins[j..j+w], del[j..j+w], 0)
17             win[0..w] = max(c_m[j..j+w], win[j..j+w])
18         end for
19
20         // 2. Minimize computation with sliding-window
21         cur_max = extract_max_score(win[0..w])
22         if cur_max > max_score then
23             max_position = get_position(c_m, cur_max)
24         end if
25
26         // 3. Skip zero scores with dynamic pruning
27         zero_num = 0
28         while l_m[j_b-1] = 0 and c_m[j_b] = 0 do
29             zero_num += 1
30         end while
31         j_b += zero_num; zero_num = 0
32         while l_m[j_e-1] = 0 and c_m[j_e] = 0 do
33             zero_num += 1
34         end while
35         j_e -= zero_num
36     end for
37     return (max_score, max_position)
38 End Procedure
```

4.2 Skipping Zero Alignment with Dynamic Pruning

The seed-extension algorithm improves upon the basic BSW algorithm by imposing a constraint: the backtracking path from the position with the maximum score must not contain any cell with a zero alignment score. Leveraging this property, we designed a dynamic pruning strategy along the anti-diagonal direction. As shown in the Figure 8, for a cell on the lower-left boundary, if the scores of its upper and upper-left neighboring cells are both zero, this cell can be eliminated in the next computation round. The rationale is that all backtracking paths passing through this cell would inevitably traverse through a zero-score position—either upward or diagonally upward—violating the requirement of the seed-extension algorithm. Similarly, dynamic reduction can be applied to the upper-right boundary. As shown in Listing 4 Line 26 ~ Line 35, by dynamically pruning subsequent computations based on currently computed scores, we effectively reduce the search space. An example is provided in the Figure 8 for illustration.

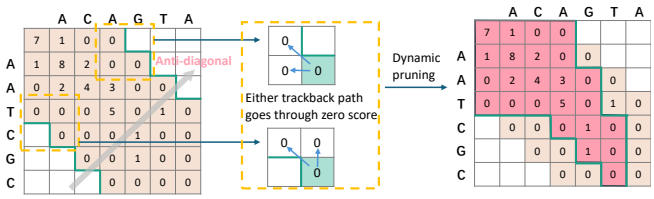

Figure 8. Principles and examples of dynamic pruning strategy along the anti-diagonal.

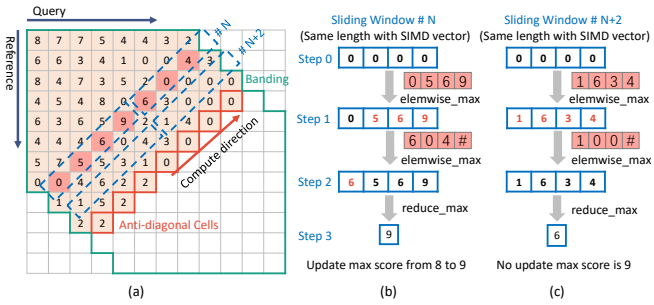

Figure 9. Workflow for updating maximum score and its position using the sliding window. (a) Workflow and update direction of sliding window. (b) Score update steps in #N round. (c) Score update steps in #N+2 round.

4.3 Minimizing Gather Operations with Sliding Window Mechanism

The intra-query parallelism approach introduces additional gather operations when determining the position of the maximum score, which constrains further performance improvement. This is also why this method has rarely been adopted for optimizing seed-extension in previous studies. To address this issue, we incorporate a sliding window algorithm to eliminate unnecessary reduction operations. As shown in Figure 9, in iteration N, we obtain a maximum score of *9*, which is larger than the previous maximum of *8*. At this point, we traverse the anti-diagonal to locate the position of this maximum. However, in iteration $N+2$, the current maximum score is 6—smaller than the previous global maximum of 9—so we skip the computation of the maximum position in this round (Line 20 ~ Line 24 in Listing 4.). As mentioned

in Section 4.1, the vast majority of queries do not yield valid matches; thus, this strategy effectively removes about 80% of unnecessary computation.

5 Implementation

We implemented FastAlign in about 25K lines of C code, using AVX2 instructions to vectorize the intra-query computation. FastAlign has been in stable production use at two companies and three research institutes for about 14 months, processing more than 15 PB of genomic data. FastAlign demonstrates faster processing throughput compared to commercial software solutions such as Sentieon [44]. We also optimize the system from an end-to-end pipeline perspective:

Overlapping IO with Computation. With the optimizations for the seeding and the seed-extension phase, the execution of reading and writing data should be considered. We restructured the original three-stage pipeline into a four-stage pipeline, where the input stage is divided into decompression of input files and data parsing. Data communication between each pipeline stage is facilitated using ping-pong buffers.

On-demand Memory Allocation. During the sequence alignment process, various data blocks (*i.e.,* input reads and output SAMs) introduce frequent memory allocation and release. We introduce an on-demand memory allocation strategy to reduce these frequent memory operations. Specifically, memory is only reallocated locally when the data size exceeds the memory usage from the previous round.

6 Evaluation Results

6.1 Experimental Setup

Datasets. To compare the performance of FastAlign and SOTA baselines, we use the reference human genome assembly [11] and six next-generation sequencing datasets from NCBI [16] to evaluate the performance. Table 2 presents the details of the six human datasets [15], which are adopoted by SOTA baselines [23, 25, 47, 52]. Our dataset includes the widely used human genome, *e.g.,* whole-genome sequencing (WGS) and whole-exome sequencing (WES) data[2].

Table 2. Six experimental human datasets [15].

Label	Type	Len	#Reads	Source
D1	WES	144 bp	1.1×10^8	SRR25735653
D2	WGS	101 bp	1.5×10^9	ERR194147
D3	WGS	150 bp	1.0×10^9	SRR25735658
D4	WES	150 bp	1.2×10^8	SRR25735654
D5	WGS	150 bp	3.0×10^8	ERR9129380
D6	WES	145 bp	1.8×10^8	SRR8381429

Baseline. As shown in Table 3, we use six SOTA baselines to compare their performance with FastAlign. We use GCC 11.4 and the -O3 parameter to compile and run all the CPU baselines. The GPU baselines are compiled with NVCC 12.3.

[2]WGS is employed to determine the complete DNA sequence of an organism's genome, including all coding and non-coding regions. WES focuses specifically on sequencing the exons, which are the parts of the genome that encode proteins. Both WGS and WES sequencing datasets are widely used in genetics research and medical diagnostics [26, 35].

Table 3. Cost evaluation instance that meets the minimal memory requirements of all baselines and FastAlign.

Design	Mem. (GB)	Instance
BWA-MEM [30]	6	ecs.c6a.4xlarge
BWA-MEM2 [52]	16	
FastAlign	16	
ERT-BWA-MEM [47]	64	ecs.r6a.4xlarge
BWA-MEME [23]	128	
BWA-MEM-SCALE [25]	128	
BWA-GPU [40]	10	ecs.gn6v-c8g1.2xlarge

Platform. To evaluate the system throughput, we use Alibaba Cloud [10] CPU platform (ecs.c6a.32xlarge) to evaluate the performance of FastAlign and CPU baselines. The platform is a two-socket server with two AMD EPYC 7H12 CPUs, each CPU having 64 cores and supporting AVX2 instructions. The main memory size is 256 GB, sufficient for all CPU baselines to load their reference indices.

To evaluate the cost, we select the minimum-cost configuration under the pay-as-you-go pricing model that meets both core and memory requirements for each baseline. These instances are provided by Alibaba Cloud [10] and located in the North China region. As shown in Table 3, we choose the ecs.c6a.4xlarge instance (16 AMD EPYC 7H12 vCPU cores, 32 GB main memory, and 2TB SSDs) to evaluate their cost since 32GB memory is sufficient for BWA-MEM [30], BWA-MEM2 [52], and FastAlign. We use the ecs.r6a.4xlarge instance (16 AMD EPYC 7H12 vCPU cores, 128 GB main memory, and 2TB SSDs) to evaluate their cost since more than 64GB of memory is required for ERT-BWA-MEM2 [47], BWA-MEME [23], and BWA-MEM-SCALE [25]. Our experimental GPU platform is ecs.gn6v-c8g1.2xlarge (NVIDIA V100 GPU) to evaluate the cost of GPU baselines. We use two 2 TB SSDs as external storage for data input and output, respectively. The operating system is Ubuntu 22.04.

6.2 Overall Performance

Exp-1 Throughput. We compare FastAlign with six SOTA sequence alignment systems. To evaluate the performance, we set the number of threads to 128 and used the default parameters provided by the baselines for all other settings. The GPU baseline utilized its recommended parameters for performance comparison. Figure 10 shows the throughput for the six datasets. FastAlign achieves a throughput of 357,725 reads/sec to 590,009 reads/sec. We can find that: (i) Compared to the other six baselines, BWA-MEM [30], BWA-MEM2 [52], ERT-BWA-MEM2 [47], BWA-MEME [23], BWA-MEM-SCALE [25], and BWA-GPU [40], FastAlign achieves up to 2.69×, 2.54×, 2.90×, 2.93×, 2.27×, and 3.28× speedup, respectively. The average throughput speedup of FastAlign is

2.85× since it optimizes the seeding phase, the seed-extension phase, and the system level. (ii) FastAlign has stable performance improvements on WGS and WES datasets (2.25× to 2.69× compared to BWA-MEM). In contrast, BWA-MEM2, ERT-BWA-MEM2, and BWA-MEME all perform poorly on the WES datasets. WES datasets generate more queries with length diversity in adjacent regions, exacerbating the redundant computation problem of the inter-query SIMD implementation in the seed-extension phase.

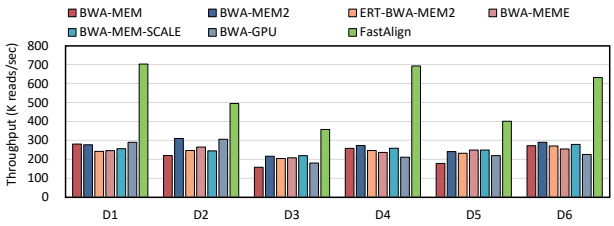

Figure 10. Throughput of BWA-MEM [30], BWA-MEM2 [52], ERT-BWA-MEM2 [47], BWA-MEME [23], BWA-MEM-SCALE [25], BWA-GPU [40], and FastAlign.

Exp-2 Cost. We evaluate the cost using the following metric,

$$Cost = \frac{total_reads}{throughput} \times instance_cost.$$

Figure 11 shows the cost per trillion reads for the six datasets. The cost of FastAlign ranges from 215.99 dollars to 343.78 dollars. We can find that: (i) Compared to the other six baselines, BWA-MEM [30], BWA-MEM2 [52], ERT-BWA-MEM2 [47], BWA-MEME [23], BWA-MEM-SCALE [25], and BWA-GPU [40], FastAlign achieves up to 2.69×, 2.54×, 4.30×, 4.24×, 4.07×, and 5.65× cost reduction, respectively. This emphasizes the cost-effectiveness of FastAlign compared to SOTA baselines.

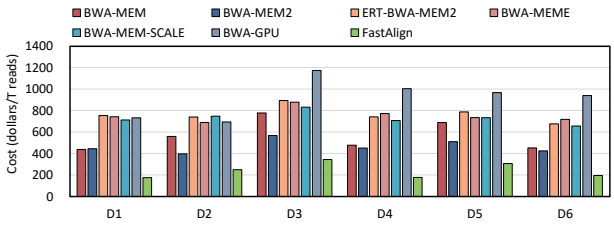

Figure 11. Cost of BWA-MEM [30], BWA-MEM2 [52], ERT-BWA-MEM2 [47], BWA-MEME [23], BWA-MEM-SCALE [25], BWA-GPU [40], and FastAlign in all six datasets.

Exp-3 Accuracy. We evaluate the accuracy of our design and baselines by comparing them with BWA-MEM [30], the standard sequence alignment tool. The evaluation metric is

$$Accuracy = \frac{total_reads - mismatch_reads}{total_reads} \times 100\%.$$

We can find that: (i) The accuracy of FastAlign, BWA-MEM2 [52], ERT-BWA-MEM2 [47], and BWA-MEME [23] is 100%, since we did not eliminate candidate alignments during the seeding and seed-extension phases. For instance, we changed the computation direction in the seed-extension phase, while our intra-query strategy does not ignore dependencies. (ii) The alignment result of BWA-GPU [40] ignores some crucial information (*e.g.,* tag and score), making all the reads mismatched. The accuracy is 0%, which is unacceptable in practical scenarios, and we will not discuss it in the subsequent analysis. (iii) BWA-MEM-SCALE [25] has about 50% tag mismatches in alignment results, which also impact the final analytical results in real-world scenarios.

Exp-4 Parallel Efficiency. We compare the throughput improvement of SOTA baselines and FastAlign as the number of execution threads increased. Figure 12 shows the throughput improvement with the increasing number of execution threads. We can find that: (i) FastAlign outperforms the SOTA baselines in throughput under all thread settings and exhibits good parallel efficiency, thanks to the high SIMD utilization of our intra-query parallelism in the seed-extension phase. (ii) FastAlign has better efficiency than other baselines with a large number of threads (64 threads and 128 threads), as a large number of threads accelerates the mapping part of the pipeline, thus introducing more pipeline stalls. The I/O and pipeline optimizations implemented in our system alleviate this bottleneck.

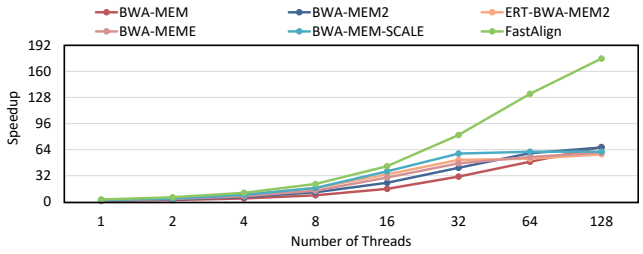

Figure 12. Parallel efficiency with thread number.

6.3 Optimization Analysis

Exp-5 Memory Efficiency. As shown in Figure 13, we evaluate the memory efficiency of FastAlign with five SOTA baselines[3]. We can find that: (i) The memory efficiency of FastAlign ranges from 9.7 million bases to 20.56 million bases. Compared to the other five baselines, FastAlign achieves up to 3.67×, 2.47×, 11.11×, 4.96×, and 18.92× speedup, respectively. We introduce the compressed FMT-Index data structure to reduce memory consumption, and we propose the multi-stage seeding algorithm to eliminate random memory access, which makes FastAlign achieve remarkable performance in the seeding phase while maintaining minimal memory consumption. (ii) Although ERT-BWA-MEM2 and BWA-MEME achieve 1.45× and 1.86× throughput speedup, they consume 4× and 8× memory consumption. In conclusion, FastAlign presents a good trade-off between seeding performance and memory consumption.

[3]We define memory efficiency as the seeding throughput that can be achieved per second for 1 GB memory.

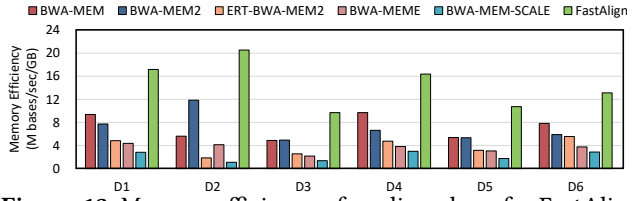

Figure 13. Memory efficiency of seeding phase for FastAlign and baselines in all the six datasets.

Exp-6 Ablation Analysis of Seeding Phase. We perform an ablation study on the seeding phase of FastAlign. Figure 14 shows the throughput of our Hybrid-Index strategies under different seed lengths. We can find that: (i) Our proposed strategies achieve throughput speedup compared to the naive FM-Index across all seed lengths. Compared to the FM-Index, our proposed Hybrid-Index, *i.e.,* FMT-Index, Kmer-Index, and Direct-Index achieve throughput speedups of 1.27×, 1.85×, and 5.75×, respectively. (ii) The Kmer Index achieves the highest throughput improvement over the FM-Index at a seed length of 20 (about 2.64×). The performance improvement diminishes (about 1.49×) as the seed length increases since we fix the hash length as 14, considered for memory efficiency. (iii) The Direct-Index contributes most to the performance when the seed length exceeds 20. It provides 4.31×, 7.58×, and 10.37× throughput speedup over the FM-Index at 50, 100, and 150 seed lengths, respectively.

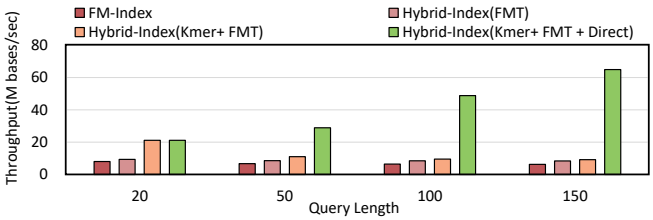

Figure 14. Ablation analysis of the seeding phase of FastAlign on D1 dataset.

Exp-7 SIMD Utilization. Figure 15(a) shows the performance of three types of BSW (Banded Smith-Waterman) implementation in BWA-MEM, BWA-MEM2, and FastAlign[4]. We can find that: (i) The throughput in the seed-extension phase of FastAlign ranges from 434.78 million bases/sec to 1581.82 million bases/sec. Compared to the scalar implementation in BWA-MEM and the inter-query SIMD implementation in BWA-MEM2, FastAlign achieves up to 4.44× and 4.77× throughput speedup, respectively. (ii) As shown in Figure 15(b), the intra-query SIMD implementation of BSW in FastAlign exhibits 56.3% SIMD utilization, while the inter-query SIMD implementation in BWA-MEM2 shows a 16.3% SIMD utilization. Since inter-query SIMD always waits for the longest query to complete, it leads to more redundant computations. (iii) For the D2 dataset, since its queries are relatively short and the length diversity between adjacent

[4]ERT-BWA-MEM2, BWA-MEME, and BWA-MEM-SCALE use the same BSW implementation as BWA-MEM2.

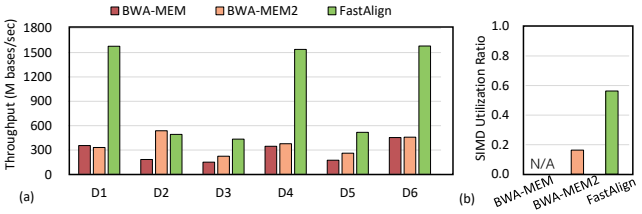

Figure 15. Optimization analysis of the seed-extension phase of FastAlign on all six datasets. (a) Throughput. (b) SIMD utilization ratio.

queries is not significant, the performance of the inter-query surpasses that of the intra-query implementation.

Exp-8 Ablation Analysis of Seed-Extension Phase. We perform an ablation analysis on the seed-extension phase of FastAlign. Figure 16 shows the throughput of our proposed strategies at different query lengths. We can find that: (i) Our seed-extension phase can achieve higher throughput for all query lengths, with an average throughput speedup of 3.65×. (ii) When the query length is shorter than 30, the pruning strategy does not significantly improve performance since the Banded Smith-Waterman has limited computation. Moreover, it even causes a throughput decrease due to the additional computation. (iii) When the query length is longer than 90, longer extension reads quickly decrease their scores to zero due to base mismatches, leading to over 90% redundant computation. In this case, the pruning strategy is more effective compared to the non-pruning version (about 3.5×). (iv) The sliding window mechanism achieves about 1.2× throughput speedup. It eliminates the computation of the maximum score and its position, but the score matrix computation still dominates the seed-extension execution time.

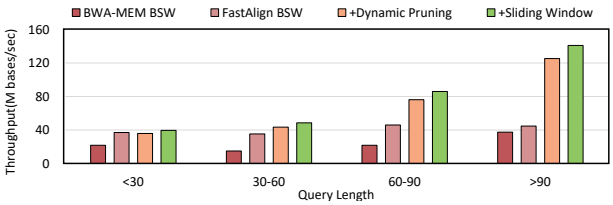

Figure 16. Ablation analysis of the seed-extension phase of FastAlign on D1 dataset.

Exp-9 Execution Time Breakdown. We compare the execution time breakdown between the SOTA baselines and FastAlign. Figure 17 shows the execution time breakdown of each step on the D1 (WES) and D3 (WGS) datasets, respectively. We can find that: (i) Compared with the traditional FM-Index (BWA-MEM [30]) and (BWA-MEM2 [52]), our Hybrid-Index reduces the irregular memory accesses and improves the performance of the seeding phase by 4.7×. Since ERT-BWA-MEM2 [47] and BWA-MEME [23] adopt hash-based indices, irregular memory accesses have been significantly reduced. However, the oversized index data leads to a longer loading time. (ii) For WES data, due to variation in the lengths of adjacent seeds, the inter-query

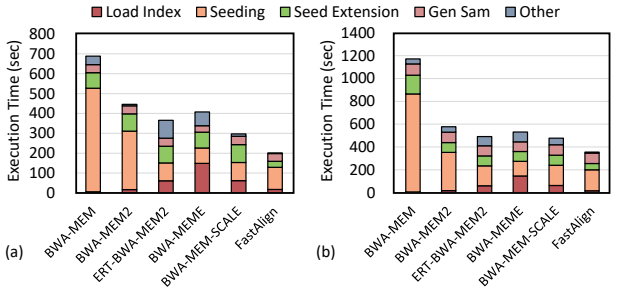

Figure 17. Execution time breakdown for processing (a) WES D1 and (b) WGS D3 datasets of FastAlign and five CPU baselines.

implementation of BWA-MEM2 causes more redundant computation. Its SIMD implementation only achieves as low as 90% throughput compared to BWA-MEM. In contrast, the seed-extension phase of FastAlign has a stable 3.0× speedup for both WES and WGS data. (iii) Compared to BWA-MEM, FastAlign achieves about a 3.3× performance improvement, with its data structure optimization contributing 2.5× and parallelization techniques contributing 1.3× to this gain.

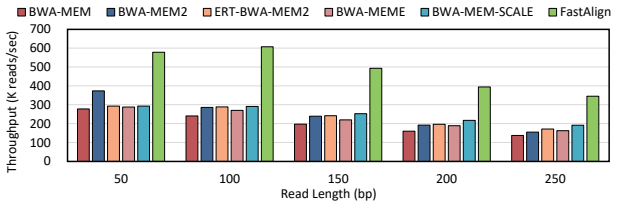

Figure 18. Read length sensitivity analysis of FastAlign.

Exp-10 Read Length. To evaluate the FastAlign's sensitivity, we generated simulated next-generation sequencing data in different lengths using Mason [42]. Figure 18 shows the end-to-end throughput of FastAlign and SOTA baselines. We can find that: (i) FastAlign achieves a throughput of 345,211 reads/sec to 606,469 reads/sec in the five simulated datasets. Compared to the other five baselines BWA-MEM [30], BWA-MEM2 [52], ERT-BWA-MEM2 [47], BWA-MEME [23], and BWA-MEM-SCALE [25], FastAlign achieves up to 2.52×, 2.22×, 2.10×, 2.24×, and 2.08× speedup, respectively. (ii) The average throughput speedup in all five simulated datasets of FastAlign is 2.10×, which demonstrates that FastAlign improves sequence alignment performance across different lengths of next-generation sequencing data.

7 Discussion

Extend FastAlign to FPGA/GPU. Our optimizations are orthogonal to the underlying hardware, thereby enabling their extension to FPGAs and GPUs. Specifically, for the seeding phase, the multi-stage index only uses simple operations, *e.g.,* k-mer lookup, base-level comparison, which has the potential to accelerate with specific logic using FPGAs. For the seed-extension phase, the intra-query and dynamic pruning constitute a foundational optimization that is readily portable to future instruction sets *e.g.,* NVIDIA DPX instructions. Although incorporating acceleration components can enhance performance for certain kernels, the performance gains may not necessarily justify the additional hardware costs. Therefore, after evaluating the overall trade-offs between performance and cost, we have opted to accelerate the entire sequence alignment pipeline on general-purpose CPU platforms.

8 Related Work

Accelerating Seeding Phase. The optimization of the seeding phase is a trade-off between memory and performance. SOTA work often falls into a dilemma: either lightweight data structures but low throughput (*e.g.,* BWA-MEM2 [52]), or high throughput but huge data structures (ERT-BWA-MEM [47], BWA-MEME [23]). Achieving Pareto optimality in the seeding phase remains a highly challenging problem. For the seed-extension phase, previous work has utilized either CPUs or GPUs to implement the parallel computation of DP cells (*e.g.,* BWA-MEM2 [52], BWA-MEM-SCALE [25], BWA-GPU [40]). However, the question of whether the computational resources are efficiently utilized has remained an overlooked problem. This paper proposes multi-stage seeding with Hybrid-Index to achieve a good trade-off between performance and memory footprint.

Accelerating Seed-Extension Phase. BWA-MEM [30] incorporates the BSW algorithm and integrates early termination strategies for the seed-extension phase, yet it lacks support for inter-query or intra-query parallelism. Previous work has utilized either CPUs or GPUs to implement the parallel computation of DP cells (*e.g.,* BWA-MEM2 [52], BWA-MEM-SCALE [25], BWA-GPU [40]). They fail to achieve high SIMD utilization because of the variation in query lengths. The question of whether the computational resources are efficiently utilized has remained an overlooked problem. This paper proposes intra-query parallelism to reduce load imbalance; our design achieves stable performance improvements over all WES and WGS data baselines.

9 Conclusion

This paper proposes FastAlign, a cost-efficient implementation of CPU-based genome sequence alignment tool. FastAlign balances between performance and resource utilization, optimizing for real-world datasets. Specifically, FastAlign includes a multi-stage seeding algorithm and intra-query seed-extension algorithm. It achieves 2.27× ~ 3.28× throughput speedups and 2.54× ~ 5.65× cost reductions compared to SOTA CPU and GPU baselines.

Acknowledgments

The authors would like to thank all anonymous reviewers and our shepherd, Prof. Sreepathi Pai, for their valuable comments and helpful suggestions. This work is supported in part by the National Natural Science Foundation of China, under Grant No. 62032023, T2125013.

References

[1] Quim Aguado-Puig, Max Doblas, Christos Matzoros, Antonio Espinosa, Juan Carlos Moure, Santiago Marco-Sola, and Miquel Moreto. 2023. WFA-GPU: gap-affine pairwise read-alignment using GPUs. *Bioinformatics* 39, 12 (2023), btad701. doi:10.1093/bioinformatics/btad701

[2] Nauman Ahmed, Jonathan Lévy, Shanshan Ren, Hamid Mushtaq, Koen Bertels, and Zaid Al-Ars. 2019. GASAL2: A GPU Accelerated Sequence Alignment Library for High-Throughput NGS Data. *BMC Bioinformatics* 20, 1 (Dec. 2019), 520. doi:10.1186/s12859-019-3086-9

[3] Mohammed Alser, Zulal Bingol, Damla Senol Cali, Jeremie Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. 2020. Accelerating Genome Analysis: A Primer on an Ongoing Journey. 40, 5 (2020), 65–75. doi:10.1109/MM.2020.3013728

[4] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. 2019. Shouji: a fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics* 35, 21 (03 2019), 4255–4263. doi:10.1093/bioinformatics/btz234

[5] Mohammed Alser, Jeremy Rotman, Dhrithi Deshpande, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, Harry Taegyun Yang, Victor Xue, Sergey Knyazev, Benjamin D. Singer, Brunilda Balliu, David Koslicki, Pavel Skums, Alex Zelikovsky, Can Alkan, Onur Mutlu, and Serghei Mangul. 2021. Technology Dictates Algorithms: Recent Developments in Read Alignment. *Genome Biology* 22, 1 (Aug. 2021), 249. doi:10.1186/s13059-021-02443-7

[6] Michael Burrows. 1994. A block-sorting lossless data compression algorithm. *SRS Research Report* 124 (1994). https://api.semanticscholar.org/CorpusID:2167441

[7] Alejandro Chacón, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. 2014. Boosting the FM-index on the GPU: Effective techniques to mitigate random memory access. *IEEE/ACM transactions on computational biology and bioinformatics* 12, 5 (2014), 1048–1059. doi:10.1109/TCBB.2014.2377716

[8] Alejandro Chacón, Juan Carlos Moure, Antonio Espinosa, and Porfidio Hernández. 2013. n-step FM-index for faster pattern matching. *Procedia Computer Science* 18 (2013), 70–79. doi:10.1016/j.procs.2013.05.170

[9] Kun-Mao Chao, William R. Pearson, and Webb Miller. 1992. Aligning two sequences within a specified diagonal band. *Bioinformatics* 8, 5 (1992), 481–487. doi:10.1093/bioinformatics/8.5.481

[10] Alibaba Cloud. 2024. Aliyun ECS. https://www.aliyun.com/product/ecs. (Accessed on 03/31/2024).

[11] Genome Reference Consortium. 2024. Genome Reference Consortium. https://www.ncbi.nlm.nih.gov/grc. (Accessed on 04/01/2024).

[12] Max Doblas, Oscar Lostes-Cazorla, Quim Aguado-Puig, Nick Cebry, Pau Fontova-Musté, Christopher Frances Batten, Santiago Marco-Sola, and Miquel Moretó. 2023. GMX: Instruction Set Extensions for Fast, Scalable, and Efficient Genome Sequence Alignment. *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2023), 1466–1480. https://api.semanticscholar.org/CorpusID:266085103

[13] Simone Faro and Thierry Lecroq. 2013. The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys (CSUR)* 45, 2 (2013), 1–42. doi:10.1145/2431211.2431212

[14] Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic Data Structures with Applications. In *FOCS*. IEEE, 390–398. doi:10.1109/SFCS.2000.892127

[15] National Center for Biotechnology Information. 2024. Genome - NCBI - NLM. https://www.ncbi.nlm.nih.gov/datasets/genome/. (Accessed on 04/01/2024).

[16] National Center for Biotechnology Information. 2025. National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/. (Accessed on 01/02/2025).

[17] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. GenAx: A genome sequencing accelerator. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 69–82. doi:10.1109/ISCA.2018.00017

[18] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. 2020. SeedEx: A Genome Sequencing Accelerator for Optimal Alignments in Subminimal Space. In *MICRO*. IEEE/ACM, Athens, Greece, 937–950. doi:10.1109/MICRO50266.2020.00080

[19] Sneha D. Goenka, Yatish Turakhia, Benedict Paten, and Mark Horowitz. 2020. SegAlign: A Scalable GPU-Based Whole Genome Aligner. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. doi:10.1109/SC41405.2020.00043

[20] Sree Charan Gundabolu, T. N. Vijaykumar, and Mithuna Thottethodi. 2021. FastZ: accelerating gapped whole genome alignment on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 13 pages. doi:10.1145/3458817.3476202

[21] IntelLabs. 2024. Intel-HLS/GKL: Accelerated kernel library for genomics. https://github.com/IntelLabs/GKL. (Accessed on 04/01/2024).

[22] Petteri Jokinen, Jorma Tarhio, and Esko Ukkonen. 1996. A comparison of approximate string matching algorithms. *Software: Practice and Experience* 26, 12 (1996), 1439–1458. doi:10.1002/(SICI)1097-024X(199612)26:12<1439::AID-SPE71>3.0.CO;2-1

[23] Youngmok Jung and Dongsu Han. 2022. BWA-MEME: BWA-MEM emulated with a machine learning approach. *Bioinformatics* 38, 9 (03 2022), 2404–2413. doi:10.1093/bioinformatics/btac137

[24] Scott D Kahn. 2011. On the future of genomic data. *science* 331, 6018 (2011), 728–729. doi:10.1126/science.1197891

[25] Changdae Kim, Kwangwon Koh, Taehoon Kim, Daegyu Han, and Jiwon Seo. 2023. BWA-MEM-SCALE: Accelerating Genome Sequence Mapping on Commodity Servers. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 83, 12 pages. doi:10.1145/3545008.3545033

[26] Hanns-Georg Klein, Peter Bauer, and Tina Hambuch. 2014. Whole genome sequencing (WGS), whole exome sequencing (WES) and clinical exome sequencing (CES) in patient care. *LaboratoriumsMedizin* 38, 4 (2014), 221–230. doi:10.1515/labmed-2013-0016

[27] Gad M Landau and Uzi Vishkin. 1988. Fast string matching with k differences. *J. Comput. System Sci.* 37, 1 (1988), 63–78. doi:10.1016/0022-0000(88)90045-1

[28] Heng Li. 2013. Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM. *arXiv:1303.3997 [q-bio]* (May 2013). arXiv:1303.3997 [q-bio]

[29] H. Li and R. Durbin. 2009. Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform. *Bioinformatics* 25, 14 (July 2009), 1754–1760. doi:10.1093/bioinformatics/btp324

[30] Heng Li and Richard Durbin. 2010. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 26, 5 (01 2010), 589–595. doi:10.1093/bioinformatics/btp698

[31] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. 2009. The sequence alignment/map format and SAMtools. *bioinformatics* 25, 16 (2009), 2078–2079. doi:10.1093/bioinformatics/btp352

[32] H. Li and N. Homer. 2010. A Survey of Sequence Alignment Algorithms for Next-Generation Sequencing. *Briefings in Bioinformatics* 11, 5 (Sept. 2010), 473–483. doi:10.1093/bib/bbq015

[33] Shuangchen Li, Dimin Niu, Yuhao Wang, Wei Han, Zhe Zhang, Tianchan Guan, Yijin Guan, Heng Liu, Linyong Huang, Zhaoyang Du, Fei Xue, Yuanwei Fang, Hongzhong Zheng, and Yuan Xie. 2022. Hyperscale FPGA-as-a-service architecture for large-scale distributed graph neural network. *Proceedings of the 49th Annual International Symposium on Computer Architecture* (2022). doi:10.1145/3470496.3527439

[34] Yewen Li, Xueqi Li, Ruihao Gao, Wanqi Liu, and Guangming Tan. 2023. NvWa: Enhancing Sequence Alignment Accelerator Throughput via Hardware Scheduling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1236–1248. doi:10.1109/HPCA56546.2023.10070978

[35] Janine Meienberg, Rémy Bruggmann, Konrad Oexle, and Gabor Matyas. 2016. Clinical sequencing: is WGS the better WES? *Human genetics* 135 (2016), 359–362. doi:10.1007/s00439-015-1631-9

[36] Gonzalo Navarro. 2001. A Guided Tour to Approximate String Matching. *ACM Computing Surveys (CSUR)* 33, 1 (2001), 31–88. doi:10.1145/375360.375365

[37] Kartika Padhan, Mohammad K Parvez, and Mohammed S Al-Dosari. 2021. Comparative sequence analysis of SARS-CoV-2 suggests its high transmissibility and pathogenicity. *Future Virology* 16, 3 (2021), 245–254. doi:10.2217/fvl-2020-0204

[38] Seongyeon Park, Junguk Hong, Jaeyong Song, Hajin Kim, Youngsok Kim, and Jinho Lee. 2024. AGAThA: Fast and Efficient GPU Acceleration of Guided Sequence Alignment for Long Read Mapping. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 431–444. doi:10.1145/3627535.3638474

[39] Julian Parkhill and Brendan W Wren. 2011. Bacterial epidemiology and biology-lessons from genome sequencing. *Genome biology* 12 (2011), 1–7. doi:10.1186/gb-2011-12-10-230

[40] Minh Pham, Yicheng Tu, and Xiaoyi Lv. 2023. Accelerating BWA-MEM Read Mapping on GPUs. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 155–166. doi:10.1145/3577193.3593703

[41] Mireille Régnier. 1989. Knuth-Morris-Pratt algorithm: an analysis. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 431–444. doi:10.1007/3-540-51486-4_90

[42] Knut Reinert, Temesgen Hailemariam Dadi, Marcel Ehrhardt, Hannes Hauswedell, Svenja Mehringer, René Rahn, Jongkyu Kim, Christopher Pockrandt, Jörg Winkler, Enrico Siragusa, et al. 2017. The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. *Journal of biotechnology* 261 (2017), 157–168. doi:10.1016/j.jbiotec.2017.07.017

[43] Markus Schmidt, Klaus Heese, and Arne Kutzner. 2019. Accurate High Throughput Alignment via Line Sweep-Based Seed Processing. *Nat Commun* 10, 1 (Dec. 2019), 1939. doi:10.1038/s41467-019-09977-2

[44] Sentieon. 2024. Home - Sentieon | Enable Precision Data for Precision Medicine. https://www.sentieon.com/ [Online; accessed 2025-06-01].

[45] Amazon Web Services. 2025. Amazon EC2 F2 Instance. https://aws.amazon.com/cn/ec2/instance-types/f2/. (Accessed on 08/15/2024).

[46] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197. doi:10.1016/0022-2836(81)90087-5

[47] Arun Subramaniyan, Jack Wadden, Kush Goliya, Nathan Ozog, Xiao Wu, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2021. Accelerated Seeding for Genome Sequence Alignment with Enumerated Radix Trees. In *ISCA*. ACM/IEEE. doi:10.1109/ISCA52012.2021.00038

[48] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly. In *ASPLOS*. ACM, Williamsburg, VA, USA, 199–213. doi:10.1145/3173162.3173193

[49] Yatish Turakhia, Sneha D. Goenka, Gill Bejerano, and WIlliam J. Dally. 2019. Darwin-WGA: A Co-processor Provides Increased Sensitivity in Whole Genome Alignments with High Speedup. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 359–372. doi:10.1109/HPCA.2019.00050

[50] Esko Ukkonen. 1985. Algorithms for approximate string matching. *Information and control* 64, 1-3 (1985), 100–118.

[51] Esko Ukkonen and Derick Wood. 1993. Approximate string matching with suffix automata. *Algorithmica* 10, 5 (1993), 353–364. doi:10.1007/BF01769703

[52] Md. Vasimuddin, Sanchit Misra, Heng Li, and Srinivas Aluru. 2019. Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 314–324. doi:10.1109/IPDPS.2019.00041

[53] Xiaoming Wei, Xiangchun Ju, Xin Yi, Qian Zhu, Ning Qu, Tengfei Liu, Yang Chen, Hui Jiang, Guanghui Yang, Ruan Zhen, et al. 2011. Identification of sequence variants in genetic disease-causing genes using targeted next-generation sequencing. *PloS one* 6, 12 (2011), e29500. doi:10.1371/journal.pone.0029500

[54] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanovic, David A. Patterson, and Anthony D. Joseph. 2019. FPGA Accelerated INDEL Realignment in the Cloud. In *HPCA*. IEEE, Washington, DC, USA, 277–290. doi:10.1109/HPCA.2019.00044

Received 2025-09-01; accepted 2025-11-10