# BWA-MEM-SCALE: Accelerating Genome Sequence Mapping on Commodity Servers

Changdae Kim
cdkim@etri.re.kr
ETRI
Daejeon, Republic of Korea

Kwangwon Koh
kwangwon.koh@etri.re.kr
ETRI
Daejeon, Republic of Korea

Taehoon Kim
taehoon.kim@etri.re.kr
ETRI
Daejeon, Republic of Korea

Daegyu Han
hdg9400@skku.edu
Sungkyunkwan University
Suwon, Republic of Korea

Jiwon Seo*
seojiwon@hanyang.ac.kr
Hanyang University
Seoul, Republic of Korea

## ABSTRACT

As advances in Next-Generation Sequencing have made genome sequence data generation faster and cheaper, the acceleration of genome sequence mapping to the reference genome becomes an increasingly important problem. Much effort has been made to improve the performance of the sequence mapping process.

In this paper, we propose BWA-MEM-SCALE which offers software-based acceleration techniques that fully utilize system resources to speed up genome sequence mapping. BWA-MEM-SCALE has two optimization mechanisms that exploit the system memory resource; *Exact Match Filter (EMF)* finds the input reads that match in full-length to the reference genome so that the expensive mapping process is bypassed for those reads. *FM-index Accelerator (FMA)* skips the prefix of sequences in seed matching with pre-assembled data. Moreover, we fully utilize the CPU cores in the system by carefully pipelining the mapping process and using *in-memory index store*.

We implement the proposed mechanisms on BWA-MEM2 which is the state-of-the-art sequence mapping software. The evaluation shows that BWA-MEM-SCALE achieves substantial speedup compared to BWA-MEM2 when the system has a sufficient amount of resources. For example, with additional 104GB of memory, BWA-MEM-SCALE gives up to 3.32X speedup over BWA-MEM2. Because we support partially deploying the acceleration techniques, BWA-MEM-SCALE speeds up the mapping performance in proportion to the available system resource.

**Source-code:** https://github.com/etri/bwa-mem-scale

*Corresponding author


*ICPP '22, August 29-September 1, 2022, Bordeaux, France*

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545033

## CCS CONCEPTS

• **Applied computing** → **Sequencing and genotyping technologies**; **Bioinformatics**; • **Computing methodologies** → *Massively parallel algorithms.*

## KEYWORDS

NGS, genome sequence alignment, acceleration, memory scalable performance

**ACM Reference Format:**
Changdae Kim, Kwangwon Koh, Taehoon Kim, Daegyu Han, and Jiwon Seo. 2022. BWA-MEM-SCALE: Accelerating Genome Sequence Mapping on Commodity Servers. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3545008.3545033

## 1 INTRODUCTION

Advances in Next-Generation Sequencing made genome sequence data generation faster and cheaper. A single Illumina NovaSeq 6000 machine can generate 20 billion pair-end reads in 44 hours, or 4.5 million reads per hour [13]. Moreover, the cost of genome sequencing for rare disease cases is $10,000 in 2020, and is expected to fall below $1,000 with large-scale sequencing [31].

The availability of faster and cheaper genome sequence generation has made the acceleration of their mappings to the reference sequence an important problem. To obtain a complete genome sequence for a single person, a few tera-bytes of sequencing data need to be processed. Thus, much effort has been made to develop faster genome sequence mapping algorithms, such as using faster indices for seed matching [14–16, 21, 33] or applying pre-alignment filtering [28, 30, 39, 40]. Particularly, architecture and system-level optimizations are reported to improve the performance by much [16, 23, 37]. Recently, specialized accelerators are proposed for genome sequence mapping [1, 2, 5, 8, 9, 12, 17, 28, 30, 36, 42]. While they achieve large performance improvements, their techniques require special hardware, which limits their availability.

This paper proposes BWA-MEM-SCALE that accelerates genome sequence mapping with software-based acceleration techniques. Particularly, BWA-MEM-SCALE takes advantage of available system resources and dynamically applies acceleration mechanisms. Because it does not require any specialized hardware, BWA-MEM-SCALE accelerates the sequence mapping on any commercial off-the-shelf hardware. It fully utilizes the entire system resources,

that is, memory and processors, to improve the sequence mapping performance.

Specifically, BWA-MEM-SCALE optimizes each step of *seed-and-extend* based genome sequence mapping, which is widely used in many sequence alignment algorithms [18, 22, 24–27, 35], by exploiting the properties of the sequence mapping procedure. For example, we propose *Exact Match Filter* to filter the reads that match exactly with the reference genome sequence; 66–76% of input reads can be matched in this way, thus bypassing the expensive mapping computation. In addition, our *FM-index Accelerator* speeds up the performance of the FM-index algorithm; that is, we pre-assemble the prefix of sequences and create efficient indices to reduce random memory accesses in the index lookup step. Moreover, we apply *Enumerated Radix Trees* [33] to accelerate the seeding part.

BWA-MEM-SCALE fully utilizes the system memory by selectively loading the indices. With prioritization of the acceleration mechanisms, it maximizes the performance for the given memory availability. Moreover, BWA-MEM-SCALE exploits the available processing cores with carefully designed pipelining and optimization techniques. The index loading is optimized with a persistent *in-memory index store*, and the latency of read and write stages is also reduced by a read-ahead thread and batched operations.

We implement the proposed techniques in BWA-MEM2-2.0 [29], which is a state-of-the-art genome sequence mapping program with architecture and system-level optimizations. Our experiments with HG001–HG004 datasets [43] show 3.19–3.32× speedup compared to BWA-MEM2-2.0. To achieve the speedup, we use 104GB of additional memory compared to BWA-MEM2-2.0, but we do not require any special hardware components. Moreover, our re-designed pipelining and optimization techniques solely achieve 1.97–2.03× speedup without additional memory requirements.

The contributions of this paper are summarized as follows.

- BWA-MEM-SCALE exploits the available memory resources to accelerate the genome sequence mapping with *Exact Match Filter* and *FM-index Acceleration*. *Exact Match Filter* enables to bypass the expensive mapping computation for the input reads that match exactly to the reference genome, and *FM-index Acceleration* pre-assembles the prefix of sequences for the FM-index algorithm to reduce the amount of random memory access.
- BWA-MEM-SCALE also exploits the available CPU resources to maximize the genome sequence mapping performance. We maximize the pipeline of I/O and computation threads to scale the mapping performance with all the available CPU cores.
- We implement BWA-MEM-SCALE in BWA-MEM2-2.0, which is the state-of-the-art genome mapping system that is already equipped with both architectural and system-level optimizations. We evaluate our implementation with large public datasets, i.e., HG001–HG004, and show that BWA-MEM-SCALE achieves 3.19×–3.32× speedup compared to BWA-MEM2-2.0.

# 2 BACKGROUND AND RELATED WORK

## 2.1 Overview of Genome Sequence Mapping

Genome sequence mapping, also called sequence alignment, is the problem of finding the position of genomic sequence fragments in a large reference genome, such as the human genome. Because the sequence fragments may have low divergence due to natural variation or sequencing error, the aligned sub-sequences in the reference genome are allowed to be slightly different from the fragments. Thus the problem is to find the sub-sequence in the reference genome that most closely matches each sequence fragment, which is equivalent to the problem of finding approximate substring matches in a longer string that consists of alphabet $\Sigma = \{A, C, T, G\}$.

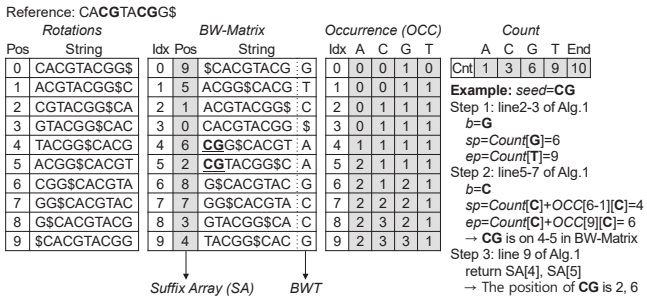


**Figure 1: Example of BWT and FM-index**

To obtain a single complete genome sequence using Next Generation Sequencing (NGS) technology, we need to extract a large number of short sequence fragments and then map the fragments with sequence mapping software. The total size of the input data for a single person reaches a few tera-bytes. To efficiently find the matches from such a large data set, most sequence mapping algorithms employ the *seed and extend* method [18, 22, 24–27, 35]. The method first runs the *seeding phase*, which finds the positions in the reference genome that match small parts (i.e., seeds) of the sequence fragments. Then in the *extension phase*, the seeds are extended along the forward and backward directions to match the entire sequence fragments and the closest matches are reported.

Recently, graph-based mapping techniques are proposed to efficiently find the mappings considering the known variations in the reference genome sequence [10, 19, 32]. The techniques use a graph structure to represent the reference genome instead of a simple sequence structure. In the genome graph, the vertices are the genome bases and the edges are the order of the bases. Then, a position in the reference genome with multiple known variations is naturally represented as multiple paths in the genome graph. Graph-based mapping techniques also employ the *seed and extend* method, where the conventional *seeding phase* is extended to match multiple sequence fragments in the reference genome graph.

## 2.2 Seeding Phase and its Acceleration

Burrows-Wheeler Transform (BWT) [4] and Ferragina Manzini (FM) index [7] are the most widely used *seeding* algorithm. Figure 1 shows an example of BWT strings and FM-index for a short reference genome, which we use to describe the algorithm below.

To generate seed strings of length $N$ in BWT, the character \$ is first appended at the end of the original strings. The character \$ denotes end-of-string and it is defined to be lexicographically smaller than all other alphabets. Then we rotate the string $N$ times to generate $N$ strings, which are shown in the leftmost table in Figure 1. Finally, the rotated strings are lexicographically sorted, and the last characters of the sorted strings constitute the BWT string; for the example in the figure, the BWT string is *GTC$AAGCCG*.

**Algorithm 1** FM-index based Seed Match Algorithm

```
1:  procedure SEED_MATCH(seed[1, M])
2:      b ← seed[M]
3:      sp ← Count[b], ep ← Count[b + 1]
4:      for i ← M − 1 to 1 do
5:          b ← Seed[i]
6:          sp ← Count[b] + OCC[sp − 1][b]
7:          ep ← Count[b] + OCC[ep][b]
8:          // seed[i, M] is the common prefix of [sp, ep − 1] in BW-Matrix
9:      endfor
10:     return SA[sp],...,SA[ep − 1] // the positions of seed in reference
11: end procedure
```

BWT string is the preceding bases of the strings in BW-Matrix (the middle table in the figure).

With the generated BWT string, FM-index performs memory-efficient seed matching. The algorithm uses three tables: *suffix array (SA)*, *occurrence table (OCC)*, and *count table (Count)*. *Suffix array* contains the original positions of BW-Matrix entries. *Occurrence table* has the number of occurrences of each base in the prefix of BWT string. That is, $OCC[9][A]$ is the number of **A**s in $BWT[0:9]$, the first 10 symbols of BWT string (in our example, 2). *Count table* has numbers for each base and they are the cumulative number of bases that are lexicographically smaller than the base. In the figure, $Count[A]$, denoting the occurrences of bases smaller than **A**, has 1 due to the appended \$. The space cost of the FM-index, denoted by the shaded area in the figure, is $O(N)$.

Algorithm 1 shows the procedure of matching a seed on the reference genome using the FM-index structure. The algorithm begins with the last base of the seed, which is matched between $Count[b]$ and $Count[b+1]-1$ in *BW-Matrix* (line 2-3). Figure 1 also presents an example run of the algorithm. Since the seed is **CG**, the matched region with **G** is $[Count[G] : Count[T]-1]$, that is, $[6:8]$. By the definition of *count table*, the region has a common prefix **G**.

Then, it extends the seed with the preceding bases, and narrows the matched region (line 4-8). In other words, it finds a range of BW-Matrix entries that have the extended seed as their common prefix. Since *BWT* indicates the preceding base of the string in BW-Matrix, $OCC[sp-1][b]$ is the number of base $b$ which are followed by the strings lexicographically less than the common prefix in the previous matched region $[sp : ep-1]$. Similarly, $OCC[ep][b]$ is the number of base $b$ which are followed by the strings lexicographically greater than the common prefix of the previous region. Thus, the narrowed region becomes $[Count[b] + OCC[sp-1][b], Count[b] + OCC[ep][b] - 1]$. That is, among the entries starting with base $b$, it excludes the entries which are not followed by the common prefix of the previous region. Step 2 in Figure 1 finds the region matched with **CG**. Among 3 **C** in BWT, the first **C** is excluded since it is not followed by **G**.

The algorithm returns the values of *suffix array* indexed by $[sp : ep-1]$ which are the positions of the seed in the reference. In the example, $SA[4]$ and $SA[5]$ are returned, as **CG** is located at 2 and 6 in the reference.

**Bottleneck:** The algorithm needs to access different parts of SA, OCC, and Count tables. For example, the accesses in the OCC table (line 4–5) is determined by the previous region, and thus they are similar to the pointer-chasing pattern or random walk. Such random access pattern results in a high cache miss rate and thus long memory latency, it is generally identified as the bottleneck of the FM-index algorithm [8, 12, 14, 33, 37, 41].

**Existing Acceleration Techniques:** To address the memory latency issues in FM-index, first, K-mer FM-index has been proposed. K-mer FM-index allows extending multiple consecutive bases at once with few memory accesses, and the number of memory accesses is reduced by $1/k$ [14, 21]. More recently, ERT proposes a variable-length seed matching algorithm and its FPGA accelerator based on a variant of the radix tree [33].

Second, memory accesses optimization for FM-index has been studied. Re-ordering accesses on the occurrence table can improve the memory locality [41], and interleaving the accesses can hide the memory latency [21]. In addition, some studies apply software prefetching techniques for FM-index [14, 21, 37].

Third, several hardware-based accelerators have been proposed to accelerate the compressed FM-index structure [38, 42] and the locality-aware FM-index structure [12, 14]. Segmentation of FM-index is also proposed to fit an FM-index segment in the internal memory of the accelerator [8].

Forth, a learned-index is developed for genome sequence mapping to quickly find the appropriate entries for seeding [14–16].

**Our Proposed Techniques:** We propose *FM-index Accelerator (FMA)* which replace 10s of random walks in line 6-7 of Algorithm 1 with few sequential cache line accesses (Section 3.3). With small tables (1.5GB in total), it improves 2.6% end-to-end performance of BWA-MEM2. Unlike other work, this mechanism does not replace the whole FM-index algorithm and can be easily detached without any correctness problem.

## 2.3 Extension Phase and its Acceleration

Dynamic programming (DP) algorithms for string matching are common approaches for the extension phase. For example, Novoalign uses Needleman-Wunsch alignment algorithm [35], and CUSHAW2 uses Smith-Waterman (SW) algorithm [27]. BWA-MEM and its following implementations use banded Smith-Waterman (BSW) algorithm, an improved version of SW algorithm [24, 37].

**Bottleneck:** The time complexity of DP algorithms after seeding phases is $O(M^2)$ for each seed, where the read length is $M$. This is computationally expensive with many reads [5, 8, 36].

**Existing Acceleration techniques:** First, SIMD optimization is a common approach to accelerate DP algorithms for genome sequence mapping [24, 25]. Recently, AVX512-based 512-bit vectorization is applied [16, 37].

Second, FPGA or ASIC based hardware accelerators have been proposed to improve DP algorithm performance for genome sequence mapping. Darwin-WGA [36] and BioSEAL [17] introduce hardware-accelerator implementations for Smith-Waterman algorithm. GenASM accelerates approximate string matching based on Bitap algorithm [5], and GenAX [8] proposes an automata-based approximate string matching algorithm and its accelerator.

Third, pre-alignment filtering is used to avoid the expensive computation of DP algorithm by early rejections of seeds that likely to have large differences from the reference genome. The seeds can

be rejected when the adjacent seeds are not matched on nearby locations [40], or when quick estimation reports that the seed will have more mismatches than threshold [39]. In addition, FPGA or ASIC based accelerators have been proposed for such estimation [1, 2, 9].

**Our Proposed Techniques:** We propose Exact Match Filter mechanism that filters out the whole-length exactly matched reads to avoid both seeding and extension phases. It requires simple hash table lookups for filtering. Although EMF needs a large memory size (54GB for the human genome), partial loading is possible with a trade-off of performance (Section 3.2).

## 2.4 Other Approaches for Acceleration

To reduce overall computation, multiple-phase algorithms have been proposed. GenCache [30] presents a four-phase algorithm on top of GenAX [8] accelerator. The first phase reveals the exactly matched reads with simple computation, and the following phases align reads with more edit distances with more complex algorithms. GenStore [28] proposes an in-storage processing architecture for genome sequence mapping. It early filters out the exactly matched reads and the definitely non-matched reads by exploiting high in-storage bandwidth. Then, it aligns the remaining reads on CPUs.

In addition, other works present system-level optimizations. BWA-MEM2 re-organizes the workflow of processing a chunk of reads, and applies chunked memory allocations to improve performance [37]. Langmead et al. examine the locking mechanisms and parallelize read and write process to improve scalability [23].

**Our proposed techniques:** Our proposed Exact Match Filter (EMF) is similar to GenStore's first phase. However, the hash table based EMF does not require high bandwidth and it also supports partial loading (Section 3.2). In addition, our implementation on top of BWA-MEM2 carefully re-balances the pipeline with read/write optimizations. Moreover, we implement an in-memory index store that removes index loading time for subsequent executions (Section 3.5).

# 3 DESIGN OF BWA-MEM-SCALE

## 3.1 Overview

BWA-MEM-SCALE is based on BWA-MEM2, a state-of-the-art sequence mapping system with architecture-aware optimizations [37]. BWA-MEM-SCALE further improves the performance with yet another optimization that fully utilizes the commodity system resources without hardware-based accelerators.

Figure 2 shows the process flow diagram of BWA-MEM-SCALE and Table 1 describes the details of the steps. We highlight the modified (or added) steps in BWA-MEM-SCALE in gray color in the diagram; we also italicize them in the table. As shown in the figure, BWA-MEM2 works as follows. After loading FM-index, it reads and parses the input FASTQ file. For each input read, it applies the SMEM, BSW, and SAM steps. The SMEM step executes the seeding phase using the FM-index based algorithm and the BSW step computes the extension phase using Banded-Smith-Waterman algorithm. Then the SAM step transforms the output into SAM format. After this mapping stage is finished, the last step writes the generated SAM string to an output file.

In the sequence mapping process of BWA-MEM2, we make four major improvements. First, to prevent repetitive loading of indices, we implement *in-memory index store* to persist the indices. Second, the read step is accelerated with our read-ahead optimization; we use a dedicated I/O read thread that runs concurrently with the main processing thread. Similarly, the write step is optimized with batch processing to fully utilize disk bandwidth. Third, after reading and parsing the input, *Exact Match Filter (EMF)* filters out the reads that exactly matched the reference genome in full-length, for which we bypass most of the (expensive) computation steps (SMEM, BSW, and SAM); and immediately generate SAM formatted outputs. It requires a large index (54GB), but loading the part of the index also works. Fourth, in the SMEM step, we accelerate the seed matching algorithm using the pre-computed data on *FM-index Accelerator (FMA)*. In addition, if the memory capacity is enough, we instead utilize *ERT* [33] to accelerate the SMEM step. *FMA* requires small size indices but shows limited performance improvement, while *ERT* shows relatively large performance improvement but requires a large index. Therefore, we apply *FMA* when the memory resource is limited, and apply *ERT* otherwise.

Moreover, BWA-MEM-SCALE can adapt itself to the changing resource availability of the underlying system and makes the most of the available resources. That is, if more memory becomes available, BWA-MEM-SCALE utilizes the memory using additional indices for *Exact Match Filtering (EMF)*, *Enumerated Radix Tree (ERT)*, and *FM-index Accelerator (FMA)*. These indices accelerate the computation steps. On the other hand, if additional cores become available, BWA-MEM-SCALE utilizes the cores well with the carefully designed pipelining.

## 3.2 Exact Match Filtering

Genome sequences generally have high similarity, which is the very reason why we can construct a whole genome sequence by mapping many short reads to a reference genome. Prior work reported that 70–80% of reads are exactly mapped in full length to the reference genome [10, 30].

Table 2 shows the amount of input reads that match exactly to *hg38* reference genome for sequencing datasets we use. It shows that 58.9–78.3% of reads in a FASTQ file are mapped to the reference genome exactly. Thus, if we find the exactly matched reads without much computation, we can bypass most of the expensive mapping computations for them and improve the mapping performance.

To take advantage of the similarity, we design *Exact Match Filter (EMF)*. It has a hash table that holds the fragments of the reference genome sequence. When mapping a read, *EMF* first calculates a hash value of the read and retrieves its hash entry. If the read is exactly matched to the entry, we bypass most of the mapping computations for the read and immediately generate its mapping results based on the information from the hash entry.

The hash table is constructed from the reference genome for a given read length. The table contains all *seeds*, that is, genome fragments of the given length, in the reference genome sequence, which are used as the table keys. For example, if the given length is 10, an entry contains $1^{st}$–$10^{th}$ bases of the reference genome as its *seed*, the other entry contains $2^{nd}$–$11^{th}$ bases of the reference genome, and so on in a sliding manner.

Since EMF is designed to quickly find the definitive mapping, it excludes the unknown parts in the reference genome when building

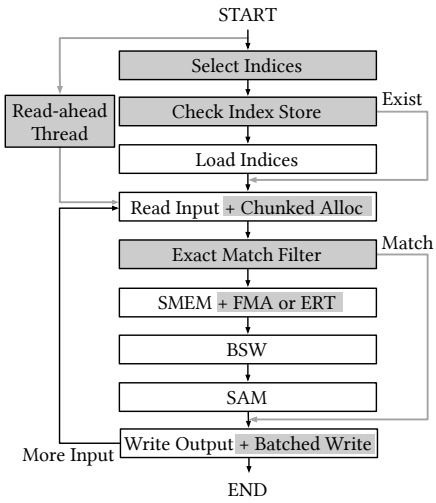


**Figure 2: The process flow diagram of BWA-MEM-SCALE**

**Table 1: Description of the steps in the process flow diagram. The steps that are added in BWA-MEM-SCALE are italicized.**

| Stage | Step | Description |
|---|---|---|
| INDEX | *1) Select Indices* | *For a given memory capacity, select best indices to load (Section 3.4).* |
| | 2) *Check Index Store* | *Check that selected indices exist on the in-memory index store. If exist, skip the step 3 (Section 3.5).* |
| | 3) Load Indices | Load the *selected* indices. |
| READ | 4) Read input *+ Chunked Alloc. + Read-ahead Thread* | Read FASTQ file and parse the content *using chunked allocation for the buffer. A separate read-ahead thread prefetches the FASTQ file from the disk. (Section 3.5)* |
| MAP | 5) *Exact Match Filter* | *Check that a read is exactly matched in its full length to the reference genome. If matched, skip the step 6–8 and directly generate SAM strings (Section 3.2).* |
| | 6) SMEM *+ FMA or ERT* | Find Super-Maximal-Exact-Matching (SMEM) parts using FM-index *or ERT [33]. When using FM-index, FM-index Accelerator (FMA) can be applied (Section 3.3).* |
| | 7) BSW | Score the possible mappings using Banded-Smith-Waterman (BSW) algorithm. |
| | 8) SAM | Generate the result string in SAM file format. |
| WRITE | 9) Write Output *+ Batched Write* | Write SAM formatted strings to disk *in batches (Section 3.5).* |

the hash table. In addition, EMF does not process the reads with unknown bases and the following mapping stage aligns such reads.

To compactly store the genome fragment mappings, we do not store seeds in the table entry. Instead, we store the location of each seed in the reference genome, and *EMF* uses the reference genome in the other index for the hash key comparison. The hash value is calculated from the concatenation of the 2-bit represented sequence using a simplified version of MurmurHash3 [3]. Since the reverse complementary sequence is treated in the same way as the original sequence in the mapping process, the hash value of a sequence should be identical to the hash value of its reverse complementary. Thus, when calculating hash values, we compare the lexicographical order of the sequence and its reverse complementary and use the smaller one to calculate the hash value.

When building *EMF*, seeds from different locations may have the same hash value for the following two reasons. The first is when each of the two seeds is actually identical or the reverse complementary to each other. In this case, the mapping procedure must find all of the locations of the seed. To address this, we construct a separate table that stores the locations of the seeds; we call this table *multi-location table*. When we store the genome fragments (and their locations) that are mapped to multiple locations in the hash table, we also store the index of the *multi-location table* in the hash table entry, so that we can retrieve all the matching locations.

The second is when the two seeds are not identical but have the same hash value. This is a well-known hash collision problem. To efficiently resolve the conflict, we build a binary search tree (BST) for each set of collision entries. That is, for those collision entries, the read with the same hash key is first compared against the root of the BST containing the collision entries. If the read is lexicographically less than the seed, the next candidate is the left child of the root entry; otherwise, we examine the right child. We traverse the hash table entries following the BST in this way and terminate the traversal if the read and the seed are identical or no more children exist.

One advantage of *EMF* design is that it can find any input reads if their length is longer than the seed length that is used to build EMF. After *EMF* calculates the hash values according to the seed length and retrieves the corresponding table entry, it confirms the exact matches for the read length. Thus, it filters out the reads only if the full length of read is exactly matched in the reference genome. In addition, if the seed location is matched to the read by the length of the seed, it is likely to be exactly matched to the read by the length of the read. Therefore, when a FASTQ file has various lengths of reads (e.g., due to the adaptor trimming and quality-based trimming), we can use the shorter seed length to filter more reads.

Another advantage of *EMF* is that it may be partially deployed. A hash entry in *EMF* covers a seed and its reverse complementary for all their mappings in multiple locations in the reference genome. That is, if we find a matching hash entry for a read, we can find all exactly matched locations for the read in the reference genome, including its reverse complementary. If we cannot find a matching hash entry, whether it's because of a partially loaded table, or because the read cannot be exactly matched, we fall back to the mapping stage, SMEM, BSW, and SAM steps. Since the part of the hash table works, *EMF* can accelerate the mapping even if there is not enough memory to load the entire *EMF* hash table.

**Table 2: Percentages of reads matched exactly to *hg38* reference genome**

| Dataset | Source | #file | BP | #read | Exactly matched (%) | Exactly matched per file (%) | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | Avg | Min | Max |
| D3 [37] | SRX020470 | 6 | 76 | 109M | 67.0% | 65.9% | 58.9% | 68.3% |
| D4 [37] | SRX207170 | 1 | 101 | 610M | 74.5% | - | - | - |
| D5 [37] | SRX206890 | 1 | 101 | 189M | 76.2% | - | - | - |
| HG001 [43] | GIAB | 871 | 148 | 6198M | 76.0% | 76.0% | 65.3% | 78.3% |
| HG002 [43] | GIAB | 935 | 148 | 6.3B | 74.9% | 74.6% | 60.1% | 77.1% |
| HG003 [43] | GIAB | 1005 | 148 | 3.2B | 73.9% | 73.5% | 53.2% | 77.8% |
| HG004 [43] | GIAB | 1024 | 148 | 3.6B | 73.5% | 73.5% | 55.6% | 77.9% |

**Table 3:** *Per-Base FMA* **entry. N is the length of seed prefix.**

| field | size | #elem | content |
|---|---|---|---|
| f_idx | 4B | N-1 | start index of FM-index for forward extension |
| b_idx | 4B | N-1 | start index of FM-index for backward extension |
| range | 4B | N-1 | range size of FM-index |
| #avail | 4B | 1 | number of valid elements of fields above |

A potential issue with *EMF* and similar mechanisms in other work [28, 30] is that it only finds exact mappings; it does not work for inexact, or sub-optimal, mappings. However, as NGS systems typically sequence many reads for a single location, finding the best mapping for an input read is sufficient. In addition, since *EMF* does not include unknown bases in the reference genome, the mappings from the exact match are certainly the best mapping for the reads. Note that *multi-location table* helps to find all exactly matched locations. Thus EMF finds all optimal mappings and only sub-optimal mappings are ignored. We validate the mapping results of BWA-MEM-SCALE in our evaluation by comparing the output against that of BWA-MEM2. The details are in Section 4.6 but when testing with D3–D5 datasets, BWA-MEM-SCALE essentially outputs the same mapping results as that of BWA-MEM2.

## 3.3 FM-index Accelerator

To mitigate the memory latency issue of FM-index, we design *FM-index Accelerator (FMA)*. *FMA* accelerates the SMEM step using our pre-assembled seed matching results of a fixed short length. Specifically, *FMA* indices store the backward extension results, described in Algorithm 1 (line 4–8), for the seeds of pre-defined length. When the SMEM step processes a read, we skip these backward extension computations for the pre-defined length. Instead, we directly use the results from our FMA indices which require only a few sequential memory lookups. After this lookup, subsequent computations find the longer exact mapping for the read in the same way as the conventional the FM-index lookups. With this technique, the number of random memory accesses is largely reduced; for example, *FMA* index with 10bp seeds eliminates 9 random accesses in the loop (line 4–8 in Algorithm 1), by adding 1–2 sequential cache line accesses on the *FMA* index.

Our proposed *FMA* uses two types of indices, namely, *Per-Base FMA index (PB-FMA)* and *Final-Only FMA index (FO-FMA)*. *PB-FMA* stores all per-base extension results, that is, every intermediate results from the iterations of the loop (line 4–8 in Algorithm 1). The intermediate results of *PB-FMA* can be used for further extensions that begin from the middle bases. Table 3 shows the structure of the *PB-FMA* index. The first three fields, two start indices and one range size of FM-index, correspond to the intermediate result values used in the SMEM step in BWA-MEM2. Since the seed matching in BWA-MEM2 concurrently performs the forward extension and the backward extension with reverse complementary, two start indices are used. In addition, as BWA-MEM2 uses the reference genome appended by the reverse complementary of the reference genome, the range size of the forward extension and that of backward extension with reverse complementary are always the same and the bi-directional extensions share a size value. In addition, we do not store the extension for the first base, according to Algorithm 1 line 2–3, since it can be calculated from the small *Count* table. In our experiments, *PB-FMA* covers 11 bases, as the size of an entry with 11 bases fits in 2 cache lines (128 bytes). Since there are 4 alphabets for a base, A, C, G, and T, the total number of entries is $4^{11}$ and the total size of *PB-FMA* is 512MB.

*Final-Only FMA index (FO-FMA)* stores only the final results of the extension. *FO-FMA* is used when the intermediate results are not needed. Its structure is similar to *PB-FMA*, but each field contains only one element (the final result). As the size of an entry is 16 bytes, we set the number of bases for *FO-FMA* as 13. The number of entries is $4^{13}$ and the total size is 1GB. We do not increase the seed length of *FO-FMA* further, since it improves the performance marginally but the index size grows exponentially.

**Algorithm 2** Selective Index Loading of BWA-MEM-SCALE

```
1:  procedure SELECT_INDEX(mem)
2:    INCLUDE_INDEX(FM-index,Reference)
3:    mem ← mem− size(FM-index,Reference)
4:    if mem ≥ size(PB-FMA) then
5:      INCLUDE_INDEX(PB-FMA)
6:      mem ← mem− size(PB-FMA)
7:    end if
8:    if mem ≥ sizeof(FO-FMA) then
9:      INCLUDE_INDEX(FO-FMA)
10:     mem ← mem− sizeof(FO-FMA)
11:   end if
12:   if mem > size(multi_location_table) then
13:     INCLUDE_INDEX(multi_location_table)
14:     mem ← mem− size(multi_location_table)
15:     N ← mem/size(EMF_hash_entry)
16:     INCLUDE_INDEX_PART(EMF_hash_entry, N)
17:     mem ← mem − N×sizeof(EMF_hash_entry)
18:   end if
19:   if mem+size(FM-index,PB-FMA,FO-FMA) ≥ size(ERT-index) then
20:     EXCLUDE_INDEX(FM-index,PB-FMA,FO-FMA)
21:     INCLUDE_INDEX(ERT-index)
22:   end if
23: end procedure
```

## 3.4 Selective Index Loading

The acceleration mechanisms, *EMF*, *FMA*, and *ERT* [33], require 54GB, 1.5GB, and 60GB of memory respectively. Such large memory requirements may negatively affect the performance when a system does not have enough memory or other applications in the sequence processing pipeline also require memory. That is, if the memory capacity of the underlying system is smaller than the required amount of the applications, performance may degrade due to memory swap, and the applications may be suspended in the worst case.

To address the memory capacity problem, BWA-MEM-SCALE proposes a *selective index loading* mechanism. We apply cost-benefit analysis and determine which indices to load and/or which parts of the indices to load in memory. Through our heuristic algorithm, *selective index loading* maximizes performance given the amount of available memory.

For *EMF*, when the available memory is limited, BWA-MEM-SCALE loads only part of its hash table. That is, we load the first

$x$% of the table entries into the memory and the table only serves for the data in the loaded entries. If the hash value in the remaining part of the table (in 100-$x$%) is requested, either directly or indirectly during the collision resolution, we simply fall back to the SMEM step. For *multi-location table*, we do not divide the table and load the whole table whenever part of *EMF* is used, since its size is not large (i.e., about 1GB for the human genome).

Such a simple partitioning method works well since each part of *EMF* is accessed with almost the same probability. This is because of the following two reasons. First, as we use a uniform hash function in *EMF*, seeds from the reference genome are distributed almost uniformly to the table entries. Second, since sequencing machines for NGS provide randomly fragmented sequences, the positions of the generated reads closely follow the uniform distribution. When we experimented with HG001 dataset, all entries of *EMF* equally contribute to finding the mappings for the reads; for the interest of space, we do not include the result of this experiment.

For the two *FMA indices*, that is, *PB-FMA* and *FO-FMA*, BWA-MEM-SCALE selectively loads the indices. Because the size of the *FMA indices* is small compared to *EMF*, BWA-MEM-SCALE does not apply partial loading for the indices; either it loads and uses the entire table in memory or does not use the table at all.

The detailed procedure of *selective index loading* is described in Algorithm 2. With the amount of available memory (*mem*), BWA-MEM-SCALE applies the techniques to maximize the performance based on the profiling results. If the given memory is smaller than the size of FM-index and reference genome, the program ignores the value and loads the entire FM-index and reference genome because the indices are essential for running the program. Otherwise, we prioritize the optimization techniques and apply a subset of the techniques based on the profiling results.

First, *Per-Base FMA* and *Final-Only FMA* indices are applied as their speedup using the same amount of memory space is the largest; it gives about 2.5% performance improvement for 1.5GB index in total. Second, we deploy *EMF*. Because the whole multi-location table is required for *EMF*, it can be loaded when the remaining memory is larger than the size of the multi-location table. As the hash table of *EMF* can be loaded partially, BWA-MEM-SCALE loads the multi-location table and as many hash entries as possible. Note that *EMF* gives about 1.1% performance improvement per additional 1GB indices. Last, *ERT* mechanism [33] is considered. In our experiments, *ERT* shows about 21% overall performance improvement by using a 60GB size index. *ERT* can completely replace SMEM step, so applying *ERT* eliminates the need for FM-index, *Per-Base FMA*, and *Final-Only FMA*. *ERT* is considered last because *ERT* has the lowest performance gain per memory capacity. However, if the available memory is sufficiently large, we apply *ERT* as it provides a higher performance gain than *FMA indices*.

## 3.5 Pipeline Optimizations

**Read and write stage optimization:** The sequence mapping in BWA-MEM2 consists of three stages, namely, read, map, and write as described in Table 1. In BWA-MEM2, two worker threads alternatively execute the three stages as shown in Figure 3 (a). The mapping stage of BWA-MEM2 takes up the majority of the execution time, and read/write stages do not have much impact on the performance. However, after the map stage of BWA-MEM2 is optimized by *EMF*, *FMA*, and *ERT*, the read and write stages have a much larger impact on the performance as in Figure 3 (b).

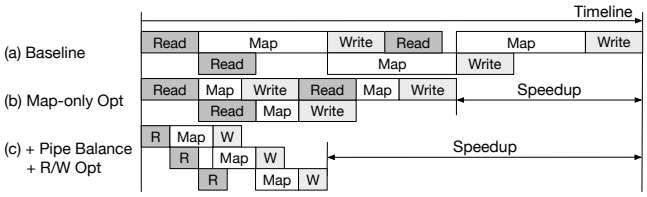


**Figure 3: Effects of read/write optimizations**

Therefore, we implement a few optimizations to shorten the two I/O stages. First, we assign a dedicated read-ahead thread that performs the disk I/O part in the read stage. The read-ahead thread continuously loads the FASTQ file into the memory buffer. Then, the (remaining) read stage parses the contents in the buffer without the disk I/O. Second, we batch the disk I/O for writing the mapping results. Because we process the output of 512 reads in a single batch, the number of system calls is reduced to 1/512. Third, we use chunked memory allocation and deallocation for the read and write stages, which also largely reduce the number of system calls.

**Pipeline balancing:** To further optimize the pipeline performance, we reconfigure the pipeline architecture. That is, we assign three worker threads, instead of two, to match the pipeline depth. In addition, we restrict the amount of CPU usage for the mapping stage, which has task-level parallelism, to ensure every worker thread can always use at least one CPU. Since read and write stages also have some computations, such as parsing the sequence and syscalls for I/O, ensuring at least one CPU helps the read and write stages finished before the other concurrent mapping stage ends.

The effect of read and write stage optimization and pipeline balancing is illustrated in Figure 3 (c). In our experiments, the execution time for the read and write stage is reduced to less than half of the original execution time with our optimizations. The optimized latency of read/write stages and the increased pipeline threads remove the delay of the mapping stage.

**In-memory index store:** BWA-MEM2 loads FM-index every time it is executed. Although FM-index is not large, the portion of the index loading time is 3%–29.3% in our experiments. To optimize the index loading time we implement *in-memory index store*. Instead of loading the indices on application-private memory, we load the indices on process shared memory and use them for the subsequent executions. As it is common to execute the sequence mapping for a series of FASTQ files, this optimization may have a large performance effect. In addition, as the indices are read-only data, *in-memory index store* can be shared by multiple processes. It helps parallel runs of BWA-MEM-SCALE also skip the index loading.

**HugeTLB for indices:** The large indices on memory may increase the TLB misses which degrades the mapping performance [20]. Thus, we apply the HugeTLB mechanism for the indices. We use 1GB pages which reduce both the number of TLB misses and the latency of each TLB miss.

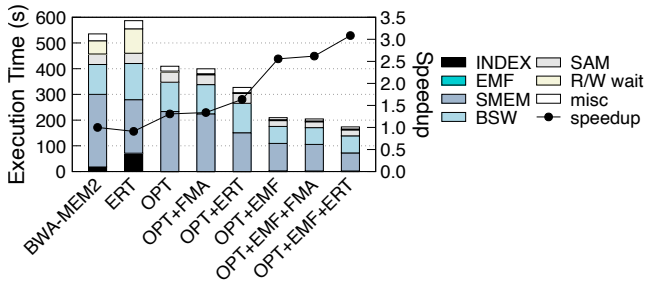


**Figure 4: Execution time breakdown and speedup of acceleration mechanisms with D5 dataset**

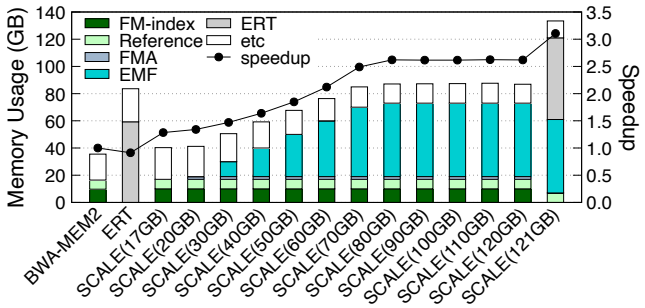


**Figure 5: Memory usage breakdown and speedup of BWA-MEM-SCALE with D5 datasets**

# 4 EVALUATION

## 4.1 Methodology

**System Configuration:** We perform experiments on a two-socket server with two Intel Xeon Gold 6230 CPUs running at 2.10GHz. Each CPU has 20 cores and supports AVX512 instructions aggressively used by BWA-MEM2. Hyper-threading is disabled for reducing the fluctuation of evaluation results. The system has 384GB of DDR4 memory which is enough to load all indices we use. For the storage, we use NVMe SSDs that is attached via PCIe 3.0. To reduce the interference between the read and write accesses, we use three separated disks to store indices, input data (FASTQ files), and output data (SAM files).

**Datasets:** We use datasets listed in Table 2. We use three datasets from the BWA-MEM2 paper [37] for detailed performance and memory usage analysis, and four datasets from the Genome-in-a-Bottle consortium [43] to show the performance of the mapping of a genome sequence of an individual. The read length of the dataset is 76bp, 101bp, or 148bp, and refer to Table 2 for the details. Due to the space limit, the results of D3 and D4 are excluded. For the reference genome, we use the initial version of *hg38*.

**Index building:** BWA-MEM-SCALE requires 4 types of indices, i.e. BWT, ERT, FMA, and EMF index. Building all indices takes 19,797 seconds in our system. Specifically, BWT index takes 2,543 seconds to build, ERT index takes 16,202 seconds, two FMA indices take 41 seconds, and EMF index for one seed length takes 1,011 seconds. Note that the index construction is required once for a reference genome, which is rarely added. For example, *hg19* was released in 2009, and *hg38*, the next version of *hg19*, was released in 2013. Including minor patches, their release interval is usually 1-2 years.

**Baseline:** The baseline of experiments is BWA-MEM2 [29, 37], and AVX512-based optimization is enabled.

## 4.2 Performance and Memory Usage Analysis

Figure 4 shows the performance impact of the combination of acceleration mechanisms. The x-axis represents the various mechanisms applied on the baseline, and the y-axis represents the execution time breakdown and speedup compared to the baseline. In the figure, *OPT* indicates all the optimization techniques in Section 3.5: read/write stage optimization, pipeline balancing, in-memory index store, and hugeTLB (1GB). In addition, *FMA*, *ERT*, and *EMF* indicate *FM-index Accelerator*, *Enumerated Radix Tree*, and *Exact Match Filter*, respectively.

In the results, our pipeline optimizations contribute 30.7%p of performance improvement. The effect of each optimization technique will be analyzed later. In addition, *FMA*, *ERT*, and *EMF* add 3.2%p, 33%p, and 125%p of performance improvement respectively. The improvement of *FMA* and *ERT* is limited as they affect only the SMEM step, while the improvement of *EMF* is large since the filtered reads bypass all computation steps. Note that we use *ERT* without hardware accelerator and the metric is end-to-end execution time. This is why our results are different from the *ERT* paper [33]. *ERT* without optimization techniques show worse performance due to index loading time and unbalanced pipeline (high R/W wait time), but our optimization techniques address such issues and *OPT+ERT* shows performance improvement.

Figure 5 shows performance and memory usage results of BWA-MEM-SCALE with the given memory capacity. The capacity numbers in the x-axis show the given memory capacity. Note that the given memory capacity is for indices, not the entire memory the program uses. The y-axis shows the total memory usage in GBs and the speedup. Memory usage is measured as resident set size (RSS). As at least FM-index and reference genome are required for mapping, the memory capacity for indices starts at 17GB.

The results in the figure show that BWA-MEM-SCALE scales the performance in terms of the given amount of memory. The performance-optimized selection of indices and partial loading of *EMF* make many performance-resource optimization points. Especially, between 30GB and 80GB, *EMF* is used partially and shows linear performance improvement. Note that BWA-MEM-SCALE contains several system-level optimizations. This is the reason that *SCALE(17GB)*, which uses the same indices as *BWA-MEM2*, shows about 30% performance improvement. In addition, when the total memory capacity in the system is over 133GB, BWA-MEM-SCALE provides 3.11× speedup using *ERT* instead of FM-index without any special hardware components.

Figure 6 shows the effect of various optimization techniques on the original BWA-MEM2 and when *EMF* and *ERT* are applied. The left graph in the figure shows that the pipeline balancing improves 10%p of BWA-MEM2 performance as the bottleneck on the read and write stages is alleviated. *In-memory index store* further improves 4.3%p of performance as it removes the index loading time. R/W optimization achieves 5.6%p of additional performance improvement by reducing the latency of the read and write stages. Finally,

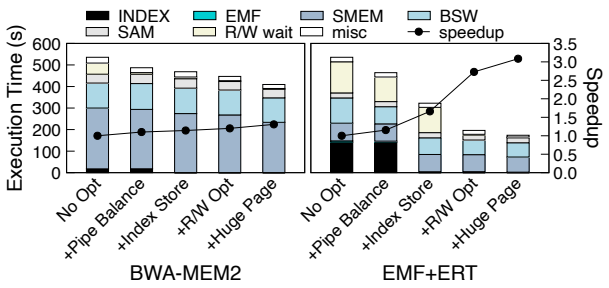


**Figure 6: The effect of optimization techniques**

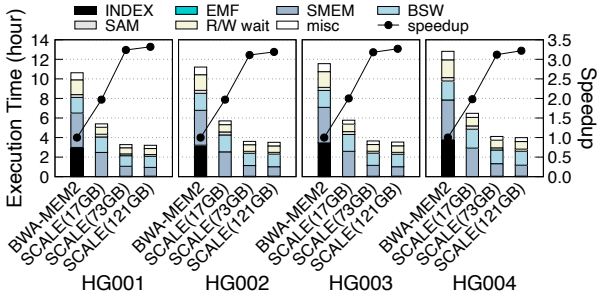


**Figure 7: HG results**

**Table 4: The effect of seed length of Exact Match Filter**

| Seed length | 101bp | 96bp | 91bp | 86bp | 81bp |
|---|---|---|---|---|---|
| Execution time (s) | 209.50 | 209.63 | 209.59 | 209.72 | 209.95 |
| Speedup w.r.t. BWA-MEM2 | 2.56 | 2.55 | 2.55 | 2.55 | 2.55 |

hugeTLB reduces the access latency of indices and the number of TLB misses and creates another 10.9%p of speedup.

The effect of optimization is enlarged when *EMF* and *ERT* are applied. As they accelerate mapping stages a lot, the effect of pipeline balancing and R/W optimization become 15.3%p and 107%p respectively. In addition, the large indices for *EMF* and *ERT* enlarge the effect of *in-memory index store* and hugeTLB. Their performance improvement reaches 50.7%p and 35.7%p respectively.

Table 4 shows the sensitivity study of the seed length of *EMF* and the performance. The results in the table are from the execution with *EMF* and the optimization techniques in Section 3.5, and D5 input is used. We use 81bp–101bp of seed as the read length of D5 input is 101bp. The table shows that even if the seed length is 20bp shorter than the read length, it has a negligible effect on the performance. The main reason is that for the locations where the slightly shorter seed is exactly matched, there is a high probability that the full-length read is exactly matched. In our experiments, 94.5% of the 81bp seed matched locations are matched to the read in 101bp full-length. That is, the little computation is added when the seed length is shorter than the read length. In addition, as the execution time of *EMF* step is a small fraction of the overall execution time, the increased execution time has a small effect on the speedup. Therefore, if a preprocessed FASTQ file is used, including adapter trimming and quality-based trimming [6], we can use the shorter seed length than the original read length. Then, even trimmed reads are accelerated by *EMF*.

## 4.3 Performance of Sequencing of Individual Genome

When performing sequencing of the individual genome, 300–500GB of sequencing data is processed. We use HG001–HG004 datasets to evaluate such scenarios.

Figure 7 shows the results. The x-axis represents the baseline and BWA-MEM-SCALE with various memory capacities for each data set, and the y-axis represents the execution time breakdown and speedup compared to the baseline. Note that results in this section include the time for one-time index loading of BWA-MEM-SCALE to reflect the whole process of genome sequence mapping.

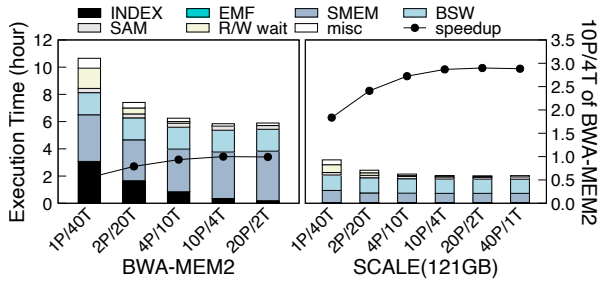


**Figure 8: Performance of parallel execution. 10P/4T means that 10 processes are executed in parallel and each process runs 4 threads for mapping stage.**

In addition, the portion of *misc* is larger than the one in the previous section as it includes the execution time of the script that runs the mapping program for each FASTQ file.

As shown in the figure, the performance increases up to 3.19–3.32× for each dataset. In other words, BWA-MEM-SCALE shortens the time of genome sequence mapping for an individual by 3.32 times with 121GB of indexes. Note that without additional indexes, BWA-MEM-SCALE(17GB) achieves nearly 2× speedup. The main reason is that the length of each FASTQ file is shorter than D5 dataset. Although the mapping finished earlier due to the short FASTQ file, the index loading time does not decrease. This enlarges the effect of *in-memory index store* of BWA-MEM-SCALE.

## 4.4 Performance of Parallel Execution

BWA-MEM-SCALE exploits available memory in the system to accelerate the genome sequence mapping with our proposed indices. Another common approach for utilizing available system memory for sequence mapping programs is to execute the multiple processes of the program in parallel for multiple FASTQ files or shards of FASTQ files. This helps to scale both BWA-MEM2 and BWA-MEM-SCALE because they do not utilize multi-cores for 100%.

We evaluate the performance of parallel execution of BWA-MEM2 and BWA-MEM-SCALE by varying the number of processes and the number of threads for the mapping stage. Since our system has 40 cores, we use up to 40 processes and set the number of threads for the mapping stage so that the total number of threads does not exceed the number of cores. We use HG001 dataset of 871 files for input, and use all 121GB of indices for BWA-MEM-SCALE. In addition, GNU parallel [34] is used to run the program in parallel with multiple FASTQ files while limiting the maximum number of concurrently running processes. Figure 8 shows the results. In the

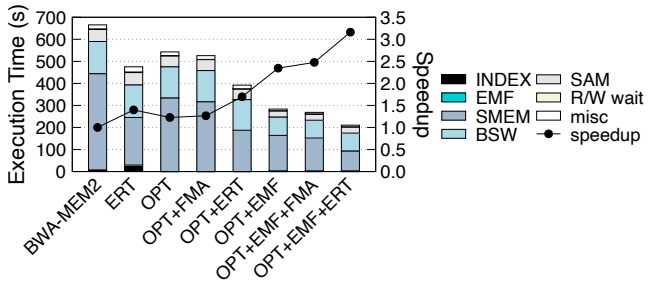


**Figure 9: Execution time breakdown and speedup of acceleration mechanisms with D5 dataset on 16-core workstation**

**Table 5: Cloud cost comparison.**

| | AWS Instance | #cpu | DRAM | $/year | Speedup |
|---|---|---|---|---|---|
| FPGA-ERT [33] | f1.4xlarge (FPGA) | 16 | 244GB | 18,242 | 2.1 [33] |
| BWA-MEM-SCALE | i3.8xlarge | 32 | 244GB | 13,928 | 2.58 |

figure, the x-axis shows the number of processes and threads, and the y-axis shows the execution time in hours. We omit a 40-process case with BWA-MEM2 due to the memory limit of our system.

As shown in the figure, for BWA-MEM2, the parallel execution accelerates the overall performance by up to 1.83×. The best case is the 10 processes with 4 threads for the mapping stage. In addition, BWA-MEM-SCALE further improves the performance by up to 1.59× with the parallel execution. The best case is when 20 processes are used each with 2 threads. The best case setting for BWA-MEM-SCALE is different from that of BWA-MEM2 since their memory usage patterns differ. Note that when running multiple processes in BWA-MEM-SCALE, the processes can share the same copy of our optimized indices in memory. This reduces the overhead from large memory consumption and improves performance. Moreover, the best-to-best comparison shows that the throughput of BWA-MEM-SCALE is 2.89× larger than BWA-MEM2.

We analyze the reasons for the performance improvement brought by parallel execution. First, from the execution time breakdown in Figure 8, we can see that the index loading time of BWA-MEM2 decreases significantly as the number of processes increases. In single process execution, while index loading is performed, other cores remain idle wasting CPU cycles. In parallel execution, however, multiple processes can load the index in parallel and other cores can be utilized at the same time by running, for example, mapping stages in other processes. This has a large performance impact. For BWA-MEM-SCALE, the one-time index loading is sufficient for all consecutive executions by multiple processes, and the serialized index loading has little impact on overall execution time. Second, the parallel execution reduces R/W wait time for both programs. As the number of threads in the mapping stage decreases, the execution time of the mapping stage increases. The increased time hides the time for the read and write stages that are running in a pipeline.

## 4.5 Performance on Workstation and Cloud

The previous experiments are executed on a high-performance machine with a large number of CPU cores. In this section, we evaluate how BWA-MEM-SCALE performs on a machine with less computing power but more storage space. We also compare BWA-MEM-SCALE and FPGA-ERT [33] on public cloud (AWS instances) in terms of performance and cost.

**Workstation:** We run BWA-MEM2 and BWA-MEM-SCALE on a workstation that has a 16-core AMD Ryzen Threadripper PRO 3955WX and 256GB of DDR4 memory. The machine has a sufficient amount of memory to apply all our optimization techniques, but its CPU has a smaller number of cores and also it does not support AVX512 instructions. Instead, AVX2 instructions are supported. The machine has PCIe 4.0 based NVMe SSD. In summary, the workstation has less computing power but faster storage.

Because of the reduced computing power and faster storage system, the (CPU-bound) mapping stage runs longer and the read/write stage and index loading run shorter. Thus the percentage of the execution time of index loading and the read/write stages is reduced.

Figure 9 shows the experimental results of running BWA-MEM-SCALE with the D5 dataset. We observed two noticeable differences compared to the previous results on the high-performance machine. First, *ERT* has 40% performance improvement on the workstation while it degrades the performance on the high-performance server. In the server environment we used, the bottlenecks of *ERT* are the long index loading time and the non-optimized latency of the read and write stage. However, as the execution time (and the ratio) of index loading and the read/write stage is reduced on the workstation, the benefits of *ERT* directly translate to end-to-end performance improvement. Second, the performance impact of the optimization techniques (*OPT* in the figure) is reduced, as our optimization techniques mainly target index loading time (*in-memory index store*) or read and write stage (*pipeline balancing*). Finally, the maximum performance improvement of BWA-MEM-SCALE on the workstation is slightly larger than one on the high-performance server.

**Cloud:** We compare the cost of BWA-MEM-SCALE with that of the FPGA-based genome sequencing accelerator, FPGA-ERT [33] in Table 5. As FPGA-ERT is implemented on AWS F1 FPGA instance (f1.4xlarge), we calculated the cost of using the reserved instance in US East (N. Virginia) for one year. In comparison, to achieve the maximum speedup in BWA-MEM-SCALE, it requires about 140GB of memory. AWS i3.8xlarge instance has a sufficient amount of memory (244GB) to utilize all acceleration mechanisms of BWA-MEM-SCALE. The cost of the reserved i3.8xlarge instance in same region for one year is 24% cheaper than that of f1.4xlarge.

In addition, ERT is reported to achieve 2.1× speedup compared to BWA-MEM2 on f1.4xlarge [33]. In comparison, from our experiments with D5 dataset on i3.8xlarge instance, BWA-MEM-SCALE achieves up to 2.58× speedup compared to BWA-MEM2. In summary, BWA-MEM-SCALE with software-only optimizations achieves higher speedup with a lower price on Amazon Cloud Services compared to FPGA-ERT.

## 4.6 Validation of Mapping Results

Because the mapping results are used in the later part of genome sequence analysis, BWA-MEM-SCALE needs to produce the mapping results that are (close to) identical to that of BWA-MEM2. Therefore, we compare the mapping results of BWA-MEM-SCALE to that of

BWA-MEM2 for D3-D5 datasets (about 360 million reads). When only the pipeline optimizations and *FM-index acceleration (FMA)* are applied, the output files by the two programs are exactly identical. When we applied *Enumerated Radix Tree (ERT)*, the mapping results are identical except for a single case for a particular read that has many unknown bases; BWA-MEM2 failed the mapping for the read, while ERT successfully found a valid mapping.

When we further applied *Exact Match Filter (EMF)*, 3.14% of all mapping results are different between the two programs. For the differences, the mappings by BWA-MEM-SCALE are the aliases of the mapping by BWA-MEM2. That is, although the mapping positions by the two programs differ, the sequences in the mapped regions are actually equivalent. In terms of MAPQ score, our results sometimes give different MAPQ scores compared to that by BWA-MEM2 because calculating MAPQ score requires the suboptimal alignments that EMF does not find. In our experiments, 4.51% of results give different MAPQ scores from BWA-MEM2. Only the results filtered by EMF have different MAPQ score. In addition, EMF generates the different tags for XS tag (the mapping score for the second-best alignment) and XA tag (the locations of suboptimal alignments). Note that they are application-specific tags and not included in the standard tags [11].

Making the SAM string to be identical is our future work. A possible solution is to extend *EMF* index so that it includes MAPQ score and the information for XS and XA tags. Detailed analysis is omitted due to the space limit, but we estimate that an additional 8.45GB of index is sufficient to store MAPQ score, XS tag, and XA tags.

## 5 CONCLUSION

This paper proposes BWA-MEM-SCALE which adapts the changing of resource availability in a system to accelerate genome sequence mapping. We propose two acceleration mechanisms, *Exact Match Filter* and *FM-index Accelerator*, and propose *selective index loading* to adapt the available memory for maximizing mapping performance with the proposed mechanisms and ERT [33]. In addition, to exploit the available CPUs, we carefully re-design the pipeline and implement *in-memory index store*. They maximize the CPU utilization of the mapping itself. The experimental results show that BWA-MEM-SCALE improves the performance of BWA-MEM2 by up to 3.32×. Although such speedup requires 104GB of additional memory than BWA-MEM2, BWA-MEM-SCALE also provides many performance-memory knobs.

## ACKNOWLEDGMENTS

This research was supported by Electronics and Telecommunications Research Institute(ETRI) grant funded by the Korean government (22ZS1300), and by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2016M3C4A7952635).

## REFERENCES

[1] Mohammed Alser, Hasan Hassan, Hongyi Xin, Oğuz Ergin, Onur Mutlu, and Can Alkan. 2017. GateKeeper: a new hardware architecture for accelerating pre-alignment in DNA short read mapping. *Bioinformatics* 33, 21 (2017), 3355–3363.

[2] Mohammed Alser, Taha Shahroodi, Juan Gómez-Luna, Can Alkan, and Onur Mutlu. 2020. SneakySnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs. *Bioinformatics* 36, 22-23 (2020), 5282–5290.

[3] Austin Appleby. 2016. MurmurHash3. http://github.com/aappleby/smhasher. (2016). Accessed: Jan. 2022.

[4] Michael Burrows and David J Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report. Systems Research Center of Digital Equipment Corporation.

[5] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. 2020. GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *Proc. International Symposium on Microarchitecture (MICRO)*. 951–966.

[6] Shifu Chen, Yanqing Zhou, Yaru Chen, and Jia Gu. 2018. fastp: an ultra-fast all-in-one FASTQ preprocessor. *Bioinformatics* 34, 17 (2018), i884–i890.

[7] Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic data structures with applications. In *Proc. Symposium on Foundations of Computer Science (FOCS)*. 390–398.

[8] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. GenAx: A Genome Sequencing Accelerator. In *Proc. International Symposium on Computer Architecture (ISCA)*. 69–82.

[9] 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 *Proc. International Symposium on Microarchitecture (MICRO)*. 937–950.

[10] Erik Garrison, Jouni Sirén, Adam M. Novak, Glenn Hickey, Jordan M. Eizenga, Eric T. Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F. Lin, Benedict Paten, and Richard Durbin. 2018. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature biotechnology* 36, 9 (2018), 875–879.

[11] The SAM/BAM Format Specification Working Group. 2021. Sequence Alignment/Map Optional Fields Specification. https://github.com/samtools/hts-specs/blob/master/SAMtags.pdf. (2021). Accessed: Jan. 2022.

[12] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. 2019. MEDAL: Scalable DIMM Based Near Data Processing Accelerator for DNA Seeding Algorithm. In *Proc. International Symposium on Microarchitecture (MICRO)*. 587–599.

[13] Illumina Inc. 2022. NovaSeq 6000 Sequencing System. https://sapac.illumina.com/content/dam/illumina/gcs/assembled-assets/marketing-literature/novaseq-6000-spec-sheet-m-gl-00271/novaseq-6000-spec-sheet-m-gl-00271.pdf. (2022). Accessed: Apr. 2022.

[14] Lei Jiang and Farzaneh Zokaee. 2021. EXMA: A Genomics Accelerator for Exact-Matching. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*. 399–411.

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

[16] Saurabh Kalikar, Chirag Jain, Md Vasimuddin, and Sanchit Misra. 2022. Accelerating minimap2 for long-read sequencing applications on modern CPUs. *Nature Computational Science* 2, 2 (2022), 78–83.

[17] Roman Kaplan, Leonid Yavits, and Ran Ginosasr. 2020. BioSEAL: In-Memory Biological Sequence Alignment Accelerator for Large-Scale Genomic Data. In *Proc. International Systems and Storage Conference (SYSTOR)*. 36–48.

[18] Daehwan Kim, Ben Langmead, and Steven L. Salzberg. 2015. HISAT: a fast spliced aligner with low memory requirements. *Nature methods* 12, 4 (2015), 357–360.

[19] Daehwan Kim, Joseph M. Paggi, Chanhee Park, Christopher Bennett, and Steven L. Salzberg. 2019. Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype. *Nature biotechnology* 37, 8 (2019), 907–915.

[20] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 705–721.

[21] Rubén Langarita, Adrià Armejach, Javier Setoain, Pablo Ibáñez Marín, Jesús Alastruey-Benedé, and Miquel Moretó. 2022. Compressed Sparse FM-Index: Fast Sequence Alignment Using Large K-Steps. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19, 01 (2022), 355–368.

[22] Ben Langmead and Steven L. Salzberg. 2012. Fast gapped-read alignment with Bowtie 2. *Nature methods* 9, 4 (2012), 357.

[23] Ben Langmead, Christopher Wilks, Valentin Antonescu, and Rones Charles. 2019. Scaling read aligners to hundreds of threads on general-purpose processors. *Bioinformatics* 35, 3 (2019), 421–432.

[24] Heng Li. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. (2013). https://arxiv.org/abs/1303.3997

[25] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (05 2018), 3094–3100.

[26] Heng Li and Richard Durbin. 2010. Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform. *Bioinformatics* 26, 5 (2010), 589–595.
[27] Yongchao Liu and Bertil Schmidt. 2012. Long read alignment based on maximal exact match seeds. *Bioinformatics* 28, 18 (2012), i318–i324.
[28] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, et al. 2022. GenStore: a high-performance in-storage processing system for genome sequence analysis. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 635–654.
[29] Vasimuddin Md, Sanchit Misra, and Heng Li. 2019. The next version of bwa-mem. https://github.com/bwa-mem2/bwa-mem2. (2019). Accessed: Jan. 2022.
[30] Anirban Nag, C. N. Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. 2019. GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment. In *Proc. International Symposium on Microarchitecture (MICRO)*. 334–346.
[31] Katharina Schwarze, James Buchanan, Jilles M. Fermont, Helene Dreau, Mark W. Tilley, John M. Taylor, Pavlos Antoniou, Samantha J.L. Knight, Carme Camps, Melissa M. Pentony, Erika M. Kvikstad, Steve Harris, Niko Popitsch, Alistair T. Pagnamenta, Anna Schuh, Jenny C. Taylor, and Sarah Wordsworth. 2020. The complete costs of genome sequencing: a microcosting study in cancer and rare diseases from a single center in the United Kingdom. *Genetics in Medicine* 22, 1 (2020), 85–94.
[32] Jouni Sirén. 2017. Indexing variation graphs. In *Proc. Workshop on algorithm engineering and experiments (ALENEX)*. 13–27.
[33] 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 *Proc. International Symposium on Computer Architecture (ISCA)*. 388–401.
[34] Ole Tange. 2021. GNU Parallel 20211222 ('Støjberg'). https://doi.org/10.5281/zenodo.5797028. (Dec. 2021). Accessed: Apr. 2022.
[35] Novocraft Technologies. 2014. Novoalign & NovoalignCS Reference Manual. https://www.novocraft.com/wp-content/uploads/Novocraft.pdf. (2014). Accessed: Apr. 2022.
[36] 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 *Proc. International Symposium on High Performance Computer Architecture (HPCA)*. 359–372.
[37] Md. Vasimuddin, Sanchit Misra, Heng Li, and Srinivas Aluru. 2019. Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*. 314–324.
[38] Yuanrong Wang, Xueqi Li, Dawei Zang, Guangming Tan, and Ninghui Sun. 2018. Accelerating FM-Index Search for Genomic Data Processing. In *Proc. International Conference on Parallel Processing (ICPP)*. 12.
[39] Hongyi Xin, John Greth, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. 2015. Shifted Hamming Distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping. *Bioinformatics* 31, 10 (01 2015), 1553–1560.
[40] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. 2013. Accelerating read mapping with FastHASH. *BMC genomics* 14, 1 (2013), 1–13.
[41] Jing Zhang, Heshan Lin, Pavan Balaji, and Wu-Chun Feng. 2013. Optimizing Burrows-Wheeler Transform-Based Sequence Alignment on Multicore Architectures. In *Proc. International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*. 377–384.
[42] Farzaneh Zokaee, Mingzhe Zhang, and Lei Jiang. 2019. FindeR: Accelerating FM-Index-Based Exact Pattern Matching in Genomic Sequences through ReRAM Technology. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 284–295.
[43] Justin M. Zook, David Catoe, Jennifer McDaniel, et al. 2016. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific data* 3, 1 (2016), 1–26.