



Accelerating minimap2 for long-read sequencing applications on modern CPUs

Saurabh Kalikar¹✉, Chirag Jain²✉, Md Vasimuddin¹✉ and Sanchit Misra¹✉

Long-read sequencing is now routinely used at scale for genomics and transcriptomics applications. Mapping long reads or a draft genome assembly to a reference sequence is often one of the most time-consuming steps in these applications. Here we present techniques to accelerate minimap2, a widely used software for this task. We present multiple optimizations using single-instruction multiple-data parallelization, efficient cache utilization and a learned index data structure to accelerate the three main computational modules of minimap2: seeding, chaining and pairwise sequence alignment. These optimizations result in an up to 1.8-fold reduction of end-to-end mapping time of minimap2 while maintaining identical output.

Long-read or single-molecule sequencing technology from Pacific Biosciences (PacBio) and Oxford Nanopore Technology (ONT) have made substantial leaps in terms of read lengths, sequencing throughput and accuracy since their introduction to the market. Longer read lengths naturally benefit genomics and transcriptomics applications, for example, to detect complex structural variation in case of DNA sequencing¹, or for novel isoform discovery during RNA sequencing². As a result, long-read sequencing is now being adopted in population-scale and biodiversity genome surveys^{3–5}. However, increased sequencing throughput (for example, >1 Tbp per day⁶) also demands faster processing of data to save time and cloud computing costs. Among the many steps performed to analyze a long-read dataset, mapping of long DNA or RNA reads to a reference sequence is usually the first and among the most time-consuming steps in any bioinformatics workflow.

Minimap2 is a widely used sequence-alignment program that supports many use-cases, including mapping long reads or a draft genome assembly to a reference sequence⁷. Although minimap2 uses well-engineered heuristics and software libraries, its performance remains considerably below the peak computing performance of a modern CPU. In minimap2, frequent branching in the code, irregular memory accesses and irregular computation make it challenging to efficiently utilize the available hardware resources. Owing to its complexity, only a few attempts have been made to accelerate minimap2, and they have also been confined to accelerating only one of the three modules within minimap2 (refs. ^{8–10}).

The highest speedup reported so far for minimap2 on multicore CPUs is 1.4 fold, and this was achieved without guaranteeing output identical to the original implementation¹⁰.

The minimap2 algorithm⁷ is based on the standard seed-chain-alignment procedure (Extended Data Fig. 1). The seeding stage identifies short fixed-length exact matches between a read and a reference sequence. Minimap2 makes use of minimizer technique¹¹—a popular *k*-mer sampling method to improve time and space requirements. Before mapping, minimap2 performs offline indexing of the

reference sequence, where it builds a multimap using a hash table with minimizers as keys and minimizer locations as values. This hash table is used during the seeding step when exact matches are collected by searching read minimizers in the reference index. Such matching pairs of minimizers form a set of anchors that are sorted and passed onto the chaining stage. From the complete list of sorted anchors, the chaining stage identifies an ordered subset of anchors that are co-linearly positioned along a diagonal^{12,13}. Minimap2 uses a customized chaining score function to prioritize the highest-scoring chains, which are likely to yield the desired base-to-base alignments of a read. It uses dynamic programming for chaining and has two versions: dynamic programming (DP)-based chaining and range minimum query (RMQ)-based DP chaining. The time complexity of the DP chaining algorithm is $O(n^2)$ in the number of anchors and is used when the number of anchors are expected to be small. The time complexity of the RMQ-based DP chaining algorithm is $O(n \log n)$ in the number of anchors and is used when the number of anchors are expected to be large. The RMQ-based DP chaining is used as a long-join heuristic in minimap2 to chain anchors that are too far from each other in the array; it uses a simplified cost function whereas DP chaining penalizes gaps more effectively. The third and final alignment stage computes base-level alignments for filling the gaps between adjacent anchors in these chains.

In this work we re-engineered the three key computational modules in minimap2: (1) seeding, (2) anchor chaining and (3) pairwise sequence alignment. Optimization of the seeding stage was achieved by replacing the standard hash-table lookup with a machine learning-based lookup using a hardware-efficient implementation of learned index data structures¹⁴. Acceleration of the anchor chaining step was achieved by designing a single-instruction multiple-data (SIMD)-based parallel chaining algorithm, which uses vector processing units (VPUs) available on modern CPUs. To program the VPUs, special SIMD instructions are used that perform the same operation on multiple data items simultaneously, thus enabling parallel computation. The VPUs have evolved through the generations of modern CPUs. Streaming SIMD extensions (SSE) provide a 128-bit SIMD instruction set, whereas the recent CPUs come with Advanced Vector Extensions 2/512 (AVX-2 and AVX-512), which support 256-bit and 512-bit SIMD instructions, respectively. In the final sequence-alignment stage, we reduced runtime by converting 128-bit (SSE) instructions to 256-bit (AVX-2) and 512-bit (AVX-512) instructions. In all of the proposed optimizations, we ensured that the final output remains 100% identical to that of minimap2, which allows users to easily switch to a faster version of minimap2 whenever faster computing throughput is desired.

We compared our optimized minimap2 implementation, mm2-fast, with minimap2 by mapping - real (1) ONT, (2) PacBio continuous long reads (CLRs), (3) PacBio high-fidelity (HiFi)

¹Intel Labs, Bangalore, India. ²Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India. ✉e-mail: saurabh.kalikar@intel.com; chirag@iisc.ac.in; vasimuddin.md@intel.com; sanchit.misra@intel.com

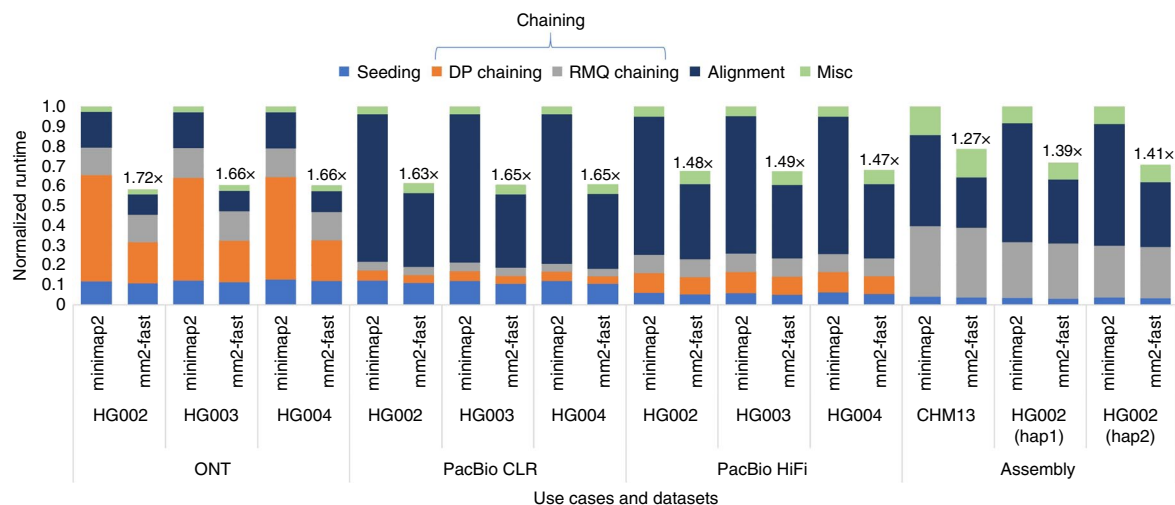


Fig. 1 | Work distribution for three modules. Seeding, chaining (DP chaining and RMQ-based DP chaining) and alignment for minimap2 and mm2-fast across different datasets. Both of the implementations were run using a single thread of a Cascade Lake CPU. The x-axis shows various query datasets, whereas the y-axis is the normalized time with respect to the mapping time consumed by minimap2 corresponding to each dataset. The speedup achieved by mm2-fast over minimap2 for randomly sampled 100,000 reads for ONT, PacBio CLR and PacBio HiFi and assembly contigs are shown on top of the mm2-fast bars.

human-sequencing data and (4) human de novo genome assemblies - to the human genome reference using multiple generations of server-grade CPUs. We achieved an up to 1.8-fold speedup compared with minimap2. In the future the mm2-fast code will be maintained as minimap2 further develops.

Results

Experimental set-up. We performed our experiments on four different processor architectures: Intel Xeon Platinum 8180 (Skylake), Intel Xeon Platinum 8280 (Cascade Lake), Intel Xeon Platinum 8380 (Ice Lake) and AMD EPYC 7742 (Rome). Architectural specifications of these systems are listed in Supplementary Table 1. Our implementation, mm2-fast, was built on top of minimap2 v2.22 and therefore all of our benchmarks show a comparison of mm2-fast with v2.22 of minimap2. Our tests involved three types of real human long-read sequencing data (ONT Guppy 3.6.0, PacBio HiFi, PacBio CLR), as well as three human genome assemblies for mapping to the standard reference GRCh38¹⁵. The assemblies were useful to demonstrate the utility of mm2-fast for faster genome-genome comparisons. Long-read sequencing datasets used here were available publicly and derived from human trio benchmark genomes HG002, HG003 and HG004 (Supplementary Table 2). The three human genome assemblies are associated with nearly haploid CHM13¹⁶ and diploid HG002 genomes¹⁷. Each type of dataset was mapped using parameters recommended in minimap2 documentation (Supplementary Table 3).

Minimap2 profile. We profiled a single-threaded execution of minimap2 using datasets listed in Supplementary Table 2, and separately measured time consumed by three key modules (1) seeding, (2) chaining (DP chaining and RMQ-based DP chaining) and (3) alignment. Figure 1 shows the performance comparison and profile of minimap2 with our optimized implementation (mm2-fast). All of the runtime values shown are normalized by the total time consumed by minimap2 corresponding to each dataset. For profiling using a single thread, we used a random subset of 100,000 reads from each of the ONT, PacBio CLR and PacBio HiFi datasets, but no sampling was performed in the case of draft genome assemblies. We observed that the three modules collectively contribute to around 85–97% of the total mapping time across different datasets. The breakdown of time consumption among the modules

was: seeding (3–13%), chaining (9–68%) and alignment (18–76%). Out of the time spent in chaining, 0–54% was spent in DP chaining, whereas RMQ-based DP chaining accounted for 4–36% of the time. Interestingly, the time distribution of the three modules varied across all of the input data types. For instance, the chaining was the most time-consuming step for the ONT and assembly datasets, whereas PacBio CLR and HiFi datasets spent the majority of the time in the alignment phase. We therefore focused on all of the three key modules to achieve better performance.

Summary of optimizations. In mm2-fast, we implemented the following optimizations while ensuring that the mapping output obtained from our optimized minimap2 remains identical to minimap2. The optimization details and the design choices are available in the Methods.

- **Seeding.** We replaced the hash-based minimizer lookup with the learned index-based search over the sorted list of the minimizers in the reference sequence. Internally, learned indexes use machine learning models to predict the positions of the desired minimizers. This resulted in nearly three to fourfold speedup in minimizer lookup and an up to 1.15-fold speedup in the seeding phase.
- **Chaining.** We accelerated DP chaining by vectorizing the traversal over the predecessor anchors using SIMD instructions and 32-bit integer/floating-point representation. Our AVX-512-based vectorized chaining achieved an up to 3.1-fold speedup over the implementation of DP chaining in minimap2.
- **Alignment.** Minimap2 implements base-level alignments using SSE2 instructions with 128-bit vector registers. As AVX-2 and AVX-512 instructions with support of 256-bit and 512-bit vector registers, respectively, are available in majority of modern general-purpose processors, we modified the alignment phase to add AVX-2- and AVX-512-based implementations. Our AVX-512-based version yielded up to 2.2-fold speedup over the SSE2-based implementation in minimap2.

Performance comparison. In Fig. 1, the bars for minimap2 and our optimized implementation (mm2-fast) show relative time consumption of each module across various datasets using a single thread on a Cascade Lake CPU. The speedups achieved for each

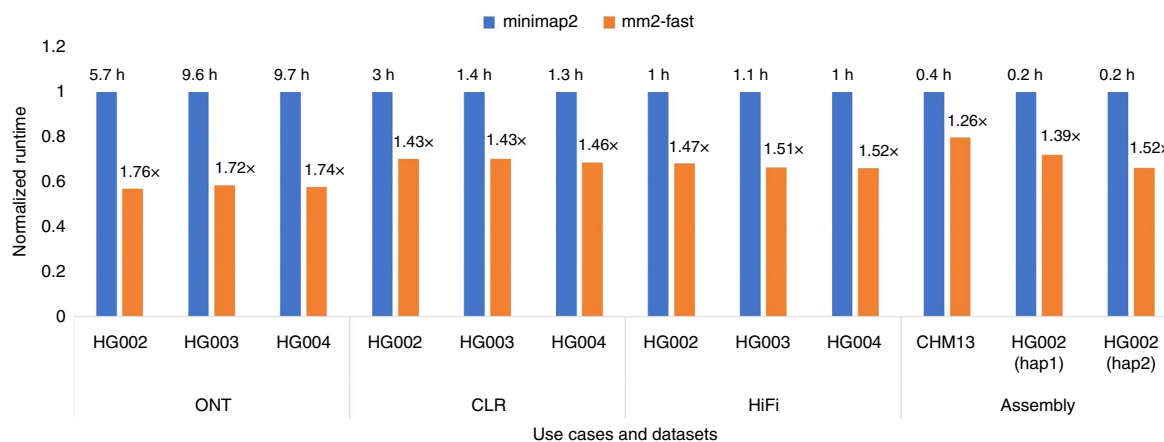


Fig. 2 | Performance comparison of minimap2 and mm2-fast on a single socket Cascade Lake CPU (28 cores) for full datasets. The x-axis shows various query datasets, whereas the y-axis is the normalized time with respect to the mapping time taken by minimap2 corresponding to each dataset. The labels above the bars for minimap2 show the end-to-end mapping time in hours, whereas the labels above the bars of our optimized implementation mm2-fast show the speedup achieved.

dataset are also shown. For single-threaded execution, we achieved up to 1.7-fold speedup compared with minimap2.

Figure 2 shows a performance comparison of minimap2 and mm2-fast over full datasets listed in Supplementary Table 2 using multithreaded execution on an entire socket of a Cascade Lake CPU. Using multithreaded execution on a single socket, we achieved an up to 1.8-fold speedup compared with minimap2. mm2-fast also scales well with multithreading. On a single socket system with 28 cores, we achieved up to 24.5-fold speedup compared with single-threaded execution (Supplementary Fig. 1). mm2-fast consumes nearly the same amount of memory as minimap2 (Supplementary Table 4). A step by step guide to using mm2-fast and verify the correctness is provided in Supplementary Section 1.

Cross-platform performance and compatibility. To ensure that our optimizations deliver speedups across various architectures, we compared the performances of mm2-fast and minimap2 on three generations of Intel architectures—Skylake, Cascade Lake and Ice Lake—and the recent AMD Rome architecture. The first three support both AVX-2 and AVX-512 vector processing, and thus we used AVX-512 version of mm2-fast on them for these experiments. Rome, however, only supports AVX-2 and hence that version of mm2-fast was used. Supplementary Table 1 provides details on the architectural specifications of these systems. Extended Data Fig. 2 shows the speedups achieved on the four architectures. For each of the query datasets, we consistently achieved high speedups on all four of the processors. Note that these systems with different architectures run on different turbo-frequencies and thus their relative performance is not comparable.

Construction of the learned index. Across the four use-cases mentioned in Supplementary Table 2, the construction of the learned index for mm2-fast takes only 2 min 23 s to 3 min 27 s. Moreover, the construction of the learned index is a one-time activity for any reference sequence and use-case combination; thus, the time spent in the construction of index gets amortized over the multiple samples that are mapped against the index. We therefore did not include it for both mm2-fast and minimap2 during the comparisons.

Discussion

Improving long-read and genome assembly mapping time is important for three reasons: (1) it cuts down waiting time for a general user, (2) it is desirable for population-scale sequencing to achieve better throughput and reduce cloud computing costs, and (3) it

improves the efficiency of real-time sequencing applications—including targeted sequencing^{18–20}—by matching speed of computation with the rate of data generation.

Mapping tools designed for long-read analysis need to account for high sequencing error-rate and, as a result, involve complex heuristics to maintain scalability using large genomes. Three computational modules (that is, seeding, co-linear chaining and alignment) were identified as the most time-consuming steps in minimap2. We focused on accelerating all three of them while developing mm2-fast.

We performed extensive profiling of software performance during the development of mm2-fast to, for example, optimize the count of the instructions executed, the cache-efficiency and other important hardware parameters that dictate CPU performance. mm2-fast is designed to achieve end-to-end hardware-aware acceleration of minimap2 on CPUs while maintaining identical output. Unlike minimap2, mm2-fast implements a learned index to allow faster minimizer lookups, a SIMD-parallel co-linear chaining algorithm and a revised implementation for sequence-alignment. Although the proposed optimizations will generally be useful for any tool which follows seed-chain-align procedure^{21–23}, we chose minimap2 to demonstrate the impact of these optimizations as it is a commonly used read mapper. mm2-fast leverages features available in modern CPUs (for example, wide SIMD instructions) and will work on any modern general-purpose processor; it also offers a similar user-interface to minimap2 for compilation, learned index construction and read mapping.

Using mm2-fast, we achieved variable speedups depending on the type of input data, that is, ONT, PacBio CLR, PacBio HiFi and genome assemblies. This is attributed to the fact that input sequence lengths and error rates change with the type of data, and also minimap2 uses different parameters (for example, the k parameter to set the k -mer size) for each type of data. As output of mm2-fast remains identical to minimap2 (v2.22) in all scenarios, mm2-fast can be directly used as a faster alternative.

There are many interesting directions that can be explored to further optimize minimap2. Here we addressed the read-mapping applications of long reads but do not support all-to-all read overlap computations yet. Computing read overlaps is the most time-consuming step during de novo genome assembly, hence this will be an important module for future acceleration. Second, it may be possible to accelerate dynamic RMQ data structures used in minimap2 for co-linear chaining of minimizer matches. Dynamic RMQ data structures are implemented using segment trees. Here

the irregular organization of data in a tree-like structure makes vectorization challenging. Range minimum query operations may be accelerated using a better cache-efficient design, further speeding up minimap2.

Methods

Seeding using a learned index. A recent work by Kraska and colleagues²⁴ has shown that an index structure can be viewed as a model that maps a key to its position, and therefore can be replaced by machine learning models. For instance, a B-tree can be seen as a model that maps a key to its position in a sorted list. The learned index structures can take advantage of the distribution of the keys and train a machine learning model (for example, a recursive model index or RMI) such that it outperforms traditional B-trees in search time and memory footprint. Following ref.²⁴, several other learned index structures have been proposed^{25–28}. Learned indexes have emerged as a performant alternative to solve problems in various domains including bioinformatics, for example, for genome indexing using FM-index¹⁴ and suffix array²⁹. In this work we design a learned index-based hash table to improve the query time for minimizer lookup during the seeding phase.

Hash table implementation in minimap2. A hash table index is used in minimap2 to store all minimizers of a reference sequence as keys, and their positions in the reference sequence as values. In the seeding phase, a successful hash lookup returns all positions at which a query minimizer occurs in the reference sequence. Typically, such hash table lookups are known to be faster. However, they may incur overheads due to a large number of collisions, and also incur high cache misses as a side-effect of irregular memory accesses during hash table access, including collision resolution. For instance, the hash table in minimap2 uses a traditional quadratic probing technique, which may lead to sub-optimal cache performance. In the following, we present our learned index-based hash table design which outperforms the traditional hash table implementation in minimap2.

Design of a learned hash table. The hash lookup problem can be modeled as a search on a list of key–value pairs that is sorted according to the keys. In our learned hash table design, we maintain two data structures as illustrated in Extended Data Fig. 3. The first is a key-sorted list of key–value pairs where keys are the actual minimizers. The second is a position list containing a concatenated list of lists of positions of all of the minimizers. For each minimizer key, the value part of the first data structure encodes the starting index in the position list and the count of the positions for the minimizer. For instance, in Extended Data Fig. 3, a minimizer entry $mm5 \rightarrow [8, 3]$ shows that $mm5$ appears three times on the reference sequence, at indexes 5, 21 and 57; these three positions are stored at three consecutive locations in the position list, starting from index 8. We use a learned index structure to search for the minimizer entry in the first data structure.

Among the proposed learned index structures, RMI exhibits the best performance/size tradeoff for real-world read-only in-memory dense arrays³⁰. RMI consists of a multilayer tree structure with a model at each tree node. An RMI with more than two layers is almost never needed³¹. Therefore, we train a two-layer RMI model to learn the distribution of the sorted keys and use the trained RMI to search through the sorted list. Extended Data Fig. 4 shows a two-level RMI model and illustrates, with an example, how we perform a lookup operation. While performing a lookup operation for a minimizer, the model at the root layer is used to predict the correct model to use at the leaf layer. The predicted model at the leaf layer is used to predict the position of the key in the sorted list. RMI guarantees that the key is present within a certain range from the predicted location³⁰. If the desired key is not found at the predicted position, the last-mile search is conducted in the provided range to find the key. The last-mile search is typically short because the key is expected to be in proximity of the predicted location. In our experiments, we observed that RMI-based lookup is nearly three to four times faster than the existing hash table implementation in minimap2.

Design choices. We make the following design choices and apply architecture-aware optimizations similar to ref.¹⁴, to achieve maximum speedup using RMI:

- **Number of leaf nodes.** The number of leaf layer models plays a crucial role in the efficiency of the RMI lookup³⁰. Using a large number of leaf layer models delivers better prediction and shorter last-mile searches—at the cost of larger memory consumption. By default, we use $n/32$ leaf layer models where n is the total number of minimizers in the list since empirically that provided a good tradeoff between prediction accuracy and memory consumption.
- **Vectorized last-mile search.** The last-mile search is performed using binary search. Once only a few elements remain for search (say, less than eight 64-bit elements for AVX-512 vector instructions), we compare them simultaneously using SIMD instructions.
- **Batched processing.** Irregular memory accesses while traversing through the RMI tree and the last-mile search lead to higher cache misses, which adversely affect the performance. We use software prefetches to hide the memory latency by processing a batch of lookups at a time. Each individual lookup can be split into a sequence of steps to be performed: (1) visit the RMI root and predict leaf layer model, (2) visit leaf model and predict the location of the key, and

(3) one step for each iteration of the binary search performed around the prediction. During batch processing, we process all of the lookups in a batch in round-robin fashion. Every time a lookup gets a turn, we advance that lookup by one step and use software prefetches to start prefetching the data into the caches for its next step. While the data is getting prefetches, we go over the rest of the lookups in the batch one by one and advance them by one step and start their prefetches. By the time, a lookup gets a turn again, the data it requires for the next step is already expected in cache. If all of the steps of a lookup are done, we replace it with the next unprocessed lookup from outside the batch. We continue this until all of the lookups are done. The batch size should be large enough to hide the memory latency but not so large that data corresponding to all of the lookups in a batch do not fit in cache. We theoretically compute a range of ideal batch sizes and empirically find the best batch size from that range.

SIMD acceleration for co-linear chaining. Chaining in minimap2. The seeding phase outputs a list of all identified anchors sorted according to their position in the reference sequence for further processing in the chaining stage. An anchor is defined as a matching minimizer between a query and a reference sequence. It can be represented as a 3-tuple (r, q, l) , where r and q are the positions of the matching minimizer on the reference and the query sequence, respectively, and l is the length of the minimizer.

Given a list $L: \{a_1, a_2, \dots, a_n\}$ of anchors sorted according to their reference positions, the chaining step identifies ordered subsets of co-linear anchors as chains which achieve the highest chaining scores. Let $S(i)$ be the highest chaining score for a chain that ends at anchor a_i ; $S(i)$ is computed using the following dynamic programming (DP) recursion in minimap2:

$$S(i) = \max \left(\max_{j < i} (S(j) + a_i, l) - \text{gap_cost}(i, j) - \text{overlap}(i, j), a_i, l) \right)$$

where the $\text{gap_cost}(i, j)$ function penalizes the score on the basis of the distance between anchors a_i and a_j , and $\text{overlap}(i, j)$ denotes the count of overlapping bases between a_i and a_j . DP chaining linearly scans the previous $S(j)$ values, thus requiring the $O(n)$ time in computation of every $S(i)$. RMQ-based DP chaining performs an RMQ query over a binary tree to get the best $S(j)$ in $O(\log(n))$ time, but needs to simplify the cost function to enable that. In mm2-fast , we accelerate the former. Extended Data Fig. 5 depicts the chaining of two co-linear anchors and their corresponding gaps and overlap.

DP chaining in minimap2. Supplementary Algorithm 1 presents a DP-based anchor-chaining implementation in minimap2. For each anchor a_i in the list (line 3), the inner loop (line 8) iterates over all of the anchors a_j where $\text{start} \leq j < i$. DP-based execution pattern ensures that the maximal chaining scores till anchor a_{i-1} are already computed before the computation of score for a_i . The expression in line 11 computes the chaining score. The two variables max_score and predecessor_index (lines 12–14) track the maximum scoring chain found so far and the index of the predecessor anchor connected to it. Considering that the gap cost and the overlap can be computed in constant time, the worst-case time complexity for the whole chaining step is $O(n^2)$.

Minimap2 applies a set of heuristics to accelerate the chaining step. The while loop (line 6) decides the range of predecessors based on the distance between the two anchors with respect to the reference sequence (that is, Reference_gap , as illustrated in Extended Data Fig. 5). For instance, if the gap between two anchors is above a user-specified threshold, the gap cost is assumed to be infinity. As the anchors are sorted, the range of anchors with larger distance can trivially be ignored for the score computation. The if condition in the inner loop (line 9) applies another set of filters based on the distances between the anchors with respect to the query sequence (that is, Query_gap , as illustrated in Extended Data Fig. 5). The anchor being considered is ignored if the gap is above a certain threshold or negative. After every iteration of the inner loop, the max_skip condition (line 15) is evaluated. According to the heuristic, if we do not find a better score over the last max_skip attempts, then the inner loop terminates. In minimap2, the default value for max_skip is 25. The sole purpose of this heuristic is to accelerate the chaining step potentially at the cost of chaining accuracy. The heuristic can be disabled by setting max_skip to a large value (say, infinity) to achieve better chaining accuracy. The actual conditional expressions and further implementation details can be found in minimap2 paper⁷.

Our SIMD-based DP chaining. As shown above in the ‘Results’ section, chaining is one of the most compute-intensive steps in the complete pipeline. Despite various heuristics used in the chaining step, the majority of the time is consumed in the execution of the inner loop. We accelerate the chaining step by redesigning the inner loop to exploit SIMD parallelism. Typically, modern compilers provide support for auto-vectorization using compilation flags or annotations which can identify opportunities to convert sequential loops to their SIMD version. However, in practice, a loop with dependencies and branching instructions makes auto-vectorization difficult. We tried auto-vectorizing the inner loop using the latest

versions of gcc and icpc compilers and verified that none of the modern compilers could auto-vectorize the loop. Designing a SIMD-friendly chaining algorithm is non-trivial because the inner loop contains multiple conditional branches, such as `Query_gap` and `max_skip`, and during the computation of `max_score`. In our optimizations, we carefully resolved the branches and inter-loop dependencies and adopted a hybrid (sequential + SIMD) approach to maximize the vectorization gains. We ensured that the sequential and vectorized versions produce identical chaining output.

More specifically, we made the following improvements to the chaining module

- Inner-loop vectorization.** Supplementary Algorithm 2 shows our vectorized algorithm for anchor chaining using 32-bit number representation. We denote ω as a SIMD width. For instance, using AVX-512-bit instructions and 32-bit numbers, we can process $\omega = \frac{512}{32} = 16$ elements simultaneously. The inner loop at line 8 implements a ω -way vectorized version that computes the maximal chaining scores. Vector instructions perform the same operation over multiple data elements. In Supplementary Algorithm 2, the notation $A[i;j]$ represents a vector operation on a range of data elements $A[i]$ to $A[j]$. $A[i;i]$ broadcasts the value of $A[i]$ to the vector register. Similarly, $[i;j]$ and $[i;i]$ load and broadcast the constant values to the registers, respectively. Conditional branches (if conditions) are vectorized using ω -bit vector masks; if the branch is taken, the respective bit in the vector mask is set to 1 (line 10). We vectorized `gap_cost` and overlap evaluation, and compute the chaining scores for ω predecessor anchors in one vectorized iteration. Once the chaining score is computed, we apply the computed mask `m_continue` (line 12) such that the scores are zeroed out when the respective bit is set in the mask. Similar to the scalar version, the two vector registers `max_score` and `predecessor_index` track the maximum scores across vector lanes. After every loop iteration, `max_score` is compared against the computed chaining scores and masked so that the maximum score is guaranteed to be present in the register. Finally, we extract the maximum score and the predecessor index using a sequential pass over `max_score` (line 16).
- Hybrid vectorized + sequential execution.** In an ideal scenario, loop vectorization delivers maximum benefit when there is no path divergence due to conditional instructions. In the presence of path divergences, vectorized implementation has to compute all control paths and use appropriate masks to obtain the desired outputs. For instance, we need to compute the chaining scores for all vector lanes even though some of the iterations would have continued without computing the chaining scores in the sequential algorithm (Supplementary Algorithm 1, line 10). This results in wasted computation over the vector lanes and leads to sub-optimal performance. When using AVX-512 vectorization, we typically found enough number of vector lanes busy in computing the scores, thus benefiting from vectorization and achieving performance improvement. However, in the case of shorter inner loops (say, <16 iterations), we observed that only a few iterations in the sequential algorithm computed the chaining score. In such a case, due to path divergence, a single vector-iteration might incur more instructions than the sequential executions of few scalar iterations. We mitigated this issue by adopting a hybrid approach: `mm2-fast` falls back to the sequential execution of the inner loop if there are fewer than five iterations; else, it uses the vectorized implementation.
- Disabling `max_skip` heuristic.** As mentioned earlier, the `max_skip` condition in Supplementary Algorithm 1 uses a heuristic parameter `max_skip`, which accelerates chaining in `minimap2` at the cost of chaining accuracy. Moreover, the `max_skip` condition also poses challenges to vectorization as it carries loop dependencies. In our optimizations, we disabled this heuristic by setting `max_skip` to infinity and removed the condition from our vectorized implementation. This benefited us in two ways: (1) we resolved the loop carrying dependency, resulting in less complex SIMD implementation and a better speedup, and (2) we achieved better chaining accuracy. Note that the performance improvement reported in the result section compares the performance of our optimizations with `max_skip = ∞` against `minimap2` with its default setting (`max_skip = 25`). We achieve an up to 3.1-fold speedup in chaining over `minimap2` (default heuristics `max_skip = 25`) and up to 8.4-fold compared with `minimap2` (`max_skip = ∞`). Supplementary Fig. 2 shows the end-to-end performance comparison of `mm2-fast` against `minimap2`, with `max_skip = 25` and `max_skip = ∞`.

SIMD acceleration for pairwise sequence alignment. `Minimap2` computes DP-based global sequence alignment to extend through the gaps between adjacent chained anchors. The presence of long gaps between anchors results in slower DP-based alignment, which can be accelerated using SIMD-based vectorization; however, longer sequences for alignment demand more bits to capture the alignment score; this leads to a lower number of available vector lanes and hence lower parallelism. `Minimap2` adopts the Suzuki–Kasahara formulation for DP-based alignment. Suzuki–Kasahara formulation bounds the number of bits required to capture the score in a DP matrix to the scoring parameters. In practice, these bounds ensure that 8-bits are sufficient for maintaining score values. Therefore, with 128-bit vector processing available with SSE, 16-way parallelism is available.

`Minimap2` applies intra-task parallelism, exploiting parallelism in a single DP matrix. All of the DP matrix cells along an anti-diagonal are independent of each other; thus, `minimap2` applies vectorization with SSE instructions along anti-diagonals to accelerate the alignment. To do so, `minimap2` switches from row-column-based matrix coordinates to diagonal–anti-diagonal-based coordinates. Furthermore, to reduce the computation burden, `minimap2` restricts the DP cell computations to a certain band around the main anti-diagonal. `Minimap2` uses a default band value of 500. Precise details about the alignment computation are available in refs. ^{7,32}

Streaming SIMD extension instructions provide only 16-way parallelism; using AVX-512, the available parallelism increases fourfold to 64-way. Moreover, the default band value in `minimap2` is large enough to keep all of the vector lanes of AVX-512 occupied. In `mm2-fast`, we accelerated the DP-based alignment by utilizing AVX-512 vectorization and used additional logic to maintain the output identical to `minimap2`. To support a wide range of processors, we also developed an AVX-2 version.

`Manymap`¹⁰ also provides an accelerated version of the alignment module of `minimap2` by upgrading the vectorization to AVX-512. `Manymap` also applies optimizations to marginally reduce the instructions in the inner DP loop; however, its alignment output differs from `minimap2`. In `minimap2`, the actual band of DP matrix that is computed could be greater than or equal in size to the given band size; this is because actual anti-diagonal computed in `minimap2` is a multiple of the SIMD width (8 for SSE); SIMD widths change when we switch from SSE to AVX-2/AVX-512. Consequently, the same DP matrix computations with longer SIMD widths can result in computing a bigger band size. Therefore, to maintain the exact same output, we introduced additional instructions in our implementations to ensure the computation of the exact same band as `minimap2`. `Manymap` does not guarantee the same (Supplementary Section 2). Due to this, we skip comparing the performance of `mm2-fast` with `manymap`.

Ideally, moving from SSE to AVX-512 has a potential for fourfold improvement in the runtime; however, in practice, we saw 1.8–2.2-fold speedups due to the following factors: (1) smaller sequences lead to under-utilization of vector lanes; (2) smaller anti-diagonals near the two corners of the DP matrix lead to idle vector lanes; (3) latency and throughput difference between SSE and AVX-512 vector instructions; and (4) use of additional instructions in the AVX-512 version to ensure the exact same output as the SSE version. The memory requirements and access pattern of the AVX-2 and AVX-512 versions in `mm2-fast` remain the same as the SSE version in `minimap2`.

Data availability

Datasets used for benchmarking are publicly available (Supplementary Table 2). Human reference genome is available at https://ftp-trace.ncbi.nlm.nih.gov/ReferenceSamples/giab/release/references/GRCh38/GCA_000001405.15_GRCh38_no_alt_analysis_set.fasta.gz. All ONT and PacBio HiFi datasets (HG002, HG003, HG004) used are available at <https://precision.fda.gov/challenges/10/view>. Datasets for PacBio CLR (HG002, HG003, HG004) are available at https://github.com/genome-in-a-bottle/giab_data_indexes. Genome assemblies are available at: CHM13: NCBI (GCA009914755.3), HG002 (hap1) and HG002 (hap2) are publicly available at ref. ³³. The speedup shown in the paper can also be realized with a smaller subset of the above datasets. Source Data are provided with this paper.

Code availability

The `mm2-fast` source code is available under the open source MIT license at <https://github.com/bwa-mem2/mm2-fast>. The particular version of `mm2-fast` used in this manuscript is publicly available at ref. ³⁴. The scripts used for the experiments in the manuscript are available at ref. ³⁵.

Optimization Notice: Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to <http://www.intel.com/performance>. Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the US and/or other countries.

Received: 20 July 2021; Accepted: 25 January 2022;

Published online: 28 February 2022

References

- Chaisson, M. J. et al. Multi-platform discovery of haplotype-resolved structural variation in human genomes. *Nat. Commun.* **10**, 1–16 (2019).
- Conesa, A. et al. A survey of best practices for RNA-seq data analysis. *Genome Biol.* **17**, 1–19 (2016).
- Beyter, D. et al. Long-read sequencing of 3,622 Icelanders provides insight into the role of structural variants in human diseases and other traits. *Nat. Genet.* **53**, 779–886 (2021).

4. Rhie, A. et al. Towards complete and error-free genome assemblies of all vertebrate species. *Nature* **592**, 737–746 (2021).
5. De Coster, W., Weissensteiner, M. H. & Sedlazeck, F. J. Towards population-scale long-read sequencing. *Nat. Rev. Genet.* **22**, 572–587 (2021).
6. *PromethION Brochure* (Nanopore Technologies, 2021); <https://nanoporetech.com/sites/default/files/s3/literature/PromethION-brochure.pdf>
7. Li, H. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* **34**, 3094–3100 (2018).
8. Guo, L., Lau, J., Ruan, Z., Wei, P. & Cong, J. Hardware acceleration of long read pairwise overlapping in genome sequencing: a race between FPGA and GPU. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines* 127–135 (IEEE, 2019).
9. Zeni, A. et al. LOGAN: high-performance GPU-based X-drop long-read alignment. In *2020 IEEE International Parallel and Distributed Processing Symposium* 462–471 (IEEE, 2020).
10. Feng, Z., Qiu, S., Wang, L. & Luo, Q. Accelerating long read alignment on three processors. In *Proc. 48th International Conference on Parallel Processing* 1–10 (ACM, 2019).
11. Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M. & Yorke, J. A. Reducing storage requirements for biological sequence comparison. *Bioinformatics* **20**, 3363–3369 (2004).
12. Abouelhoda, M. I. & Ohlebusch, E. Chaining algorithms for multiple genome comparison. *J. Discrete Algorithms* **3**, 321–341 (2005).
13. Jain, C., Gibney, D. & Thankachan, S. V. Co-linear chaining with overlaps and gap costs. Preprint at <https://www.biorxiv.org/content/10.1101/2021.02.03.429492v2> (2021).
14. Ho, D. et al. LISA: learned indexes for DNA sequence analysis. Preprint at <https://arxiv.org/abs/1910.04728> (2020).
15. Schneider, V. A. et al. Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly. *Genome Res.* **27**, 849–864 (2017).
16. Nurk, S., Koren, S., Rhie, A., Rautiainen, M. et al. The complete sequence of a human genome. Preprint at <https://doi.org/10.1101/2021.05.26.445798> (2021).
17. Cheng, H., Concepcion, G. T., Feng, X., Zhang, H. & Li, H. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nat. Methods* **18**, 170–175 (2021).
18. Payne, A. et al. Readfish enables targeted nanopore sequencing of gigabase-sized genomes. *Nat. Biotechnol.* **39**, 442–450 (2021).
19. Kovaka, S., Fan, Y., Ni, B., Timp, W. & Schatz, M. C. Targeted nanopore sequencing by real-time mapping of raw electrical signal with uncalled. *Nat. Biotechnol.* **39**, 431–441 (2021).
20. Zhang, H. et al. Real-time mapping of nanopore raw signals. *Bioinformatics* <https://doi.org/10.1093/bioinformatics/btab264> (2021).
21. Jain, C., Rhie, A., Hansen, N., Koren, S. & Phillippy, A.M. A long read mapping method for highly repetitive reference sequences. Preprint at <https://www.biorxiv.org/content/10.1101/2020.11.01.363887v1.full> (2020).
22. Sedlazeck, F. J. et al. Accurate detection of complex structural variations using single-molecule sequencing. *Nat. Methods* **15**, 461–468 (2018).
23. Ren, J. & Chaisson, M. IRA: the long read aligner for sequences and contigs. Preprint at <https://doi.org/10.1371/journal.pcbi.1009078> (2020).
24. Kraska, T., Beutel, A., Chi, E.H., Dean, J. & Polyzotis, N. The case for learned index structures. In *ACM International Conference on Management of Data* 489–504 (ACM, 2018).
25. Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R. & Kraska, T. FITting-Tree: a data-aware index structure. In *SIGMOD '19: Proceedings of the 2019 International Conference on Management of Data* 1189–1206 (ACM, 2019); <https://doi.org/10.1145/3299869.3319860>
26. Ferragina, P. & Vinciguerra, G. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* **13**, 1162–1175 (2020).
27. Ding, J. et al. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD '20: Proceedings of the 2020 International Conference on Management of Data* 969–984 (ACM, 2020). <https://doi.org/10.1145/3318464.3389711>
28. Wu, Y., Yu, J., Tian, Y., Sidle, R. & Barber, R. Designing succinct secondary indexing mechanism by exploiting column correlations. In *SIGMOD '19: Proceedings of the 2019 International Conference on Management of Data* 1223–1240 (ACM, 2019). <https://doi.org/10.1145/3299869.3319861>
29. Kirsche, M., Das, A. & Schatz, M. C. Sapling: accelerating suffix array queries with learned data models. *Bioinformatics* **37**, 744–749 (2021).
30. Marcus, R. et al. Benchmarking learned indexes. In *PVLDB* Vol. 14, 1–13 (2021).
31. Marcus, R., Zhang, E. & Kraska, T. CDFShop: exploring and optimizing learned index structures. In *SIGMOD '20: Proc. 2020 ACM SIGMOD International Conference on Management of Data* 2789–2792 (ACM, 2020); <https://doi.org/10.1145/3318464.3384706>
32. Suzuki, H. & Kasahara, M. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics* **19**, 33–47 (2018).
33. Cheng, H., Concepcion, G., Feng, X., Zhang, H. & Li, H. *Human Assemblies Evaluated in the Hifiasm Paper* (Zenodo, 2020); <https://doi.org/10.5281/zenodo.4393631>
34. Kalikar, S., Jain, C., Md, V. & Misra, S. *mm2-fast Source Code Used in the Manuscript—Accelerating Minimap2 for Long-Read Sequencing Applications on Modern CPUs* (Zenodo, 2022); <https://doi.org/10.5281/zenodo.5888171>
35. Kalikar, S., Jain, C., Md, V. & Misra, S. *Scripts Used for the Experiments in the Manuscript—Accelerating Minimap2 for Long-Read Sequencing Applications on Modern CPUs* (Zenodo, 2022); <https://doi.org/10.5281/zenodo.5884451>

Acknowledgements

This work is supported in part by the National Supercomputing Mission (NSM) India under DST/NSM/R&D_HPC_Applications to C.J. The authors are grateful to H. Li for guidance and technical discussions on minimap2 and working with us to get our improvements integrated in a branch of minimap2 github repo.

Author contributions

S.K. led the software implementation of mm2-fast. All authors contributed to algorithm design, experiments and manuscript preparation, and read and approved the final manuscript.

Competing interests

S.K., V.M. and S.M. are employees of Intel Corporation.

Additional information

Supplementary information The online version contains supplementary material available at <https://doi.org/10.1038/s43588-022-00201-8>.

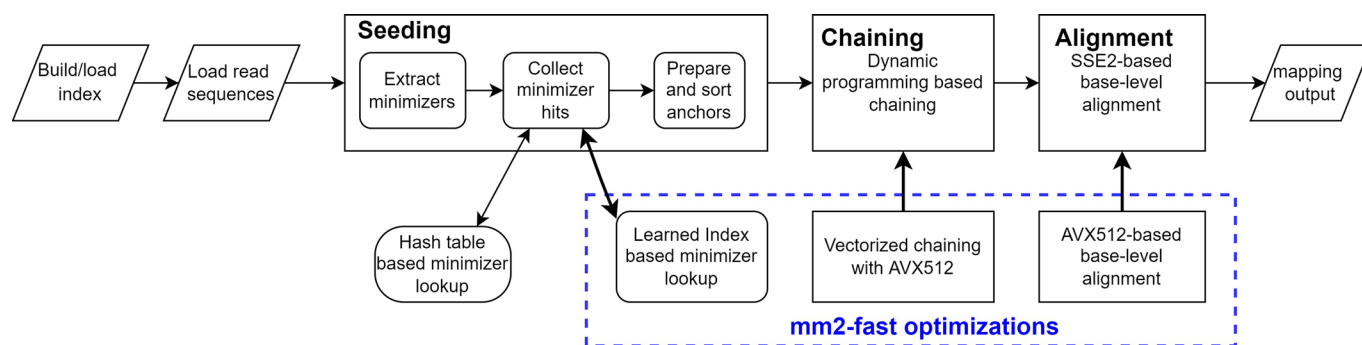
Correspondence and requests for materials should be addressed to Saurabh Kalikar, Chirag Jain, Md Vasimuddin or Sanchit Misra.

Peer review information *Nature Computational Science* thanks Aydin Buluc, Zemin Ning and the other, anonymous, reviewer(s) for their contribution to the peer review of this work. Handling editor: Fernando Chirigati, in collaboration with the *Nature Computational Science* team.

Reprints and permissions information is available at www.nature.com/reprints.

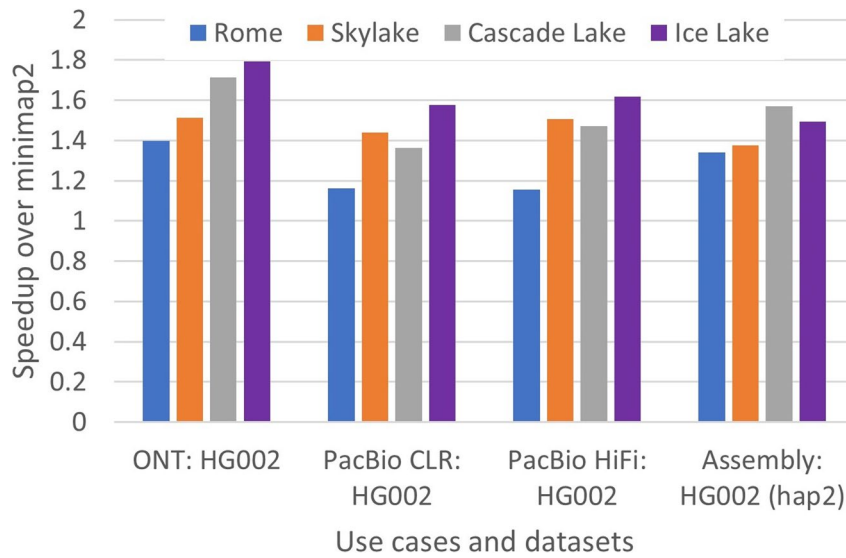
Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© The Author(s), under exclusive licence to Springer Nature America, Inc. 2022



Extended Data Fig. 1 | Minimap2 workflow depicting its three key modules - (i) seeding, (ii) chaining, and (iii) alignment - and mm2-fast optimizations.

The seeding stage identifies short fixed-length exact matches between a read and a reference sequence. Chaining stage selects an ordered subset of these exact matches (anchors) to form a chain. The final alignment stage computes base-level alignments for filling the gaps between adjacent anchors in these chains. Our optimizations to each of the modules are shown in the blue dotted rectangle.

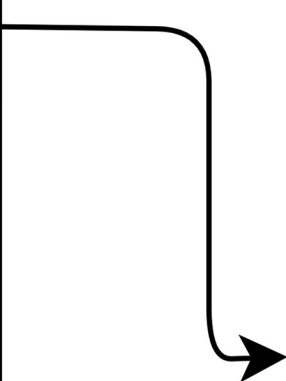


Extended Data Fig. 2 | Cross-platform performance of our optimizations for Rome, Skylake, Cascade Lake and Ice Lake architectures using single socket. X-axis shows various query datasets and y-axis indicates the speedup achieved by mm2-fast over minimap2 - both running on the same CPU.

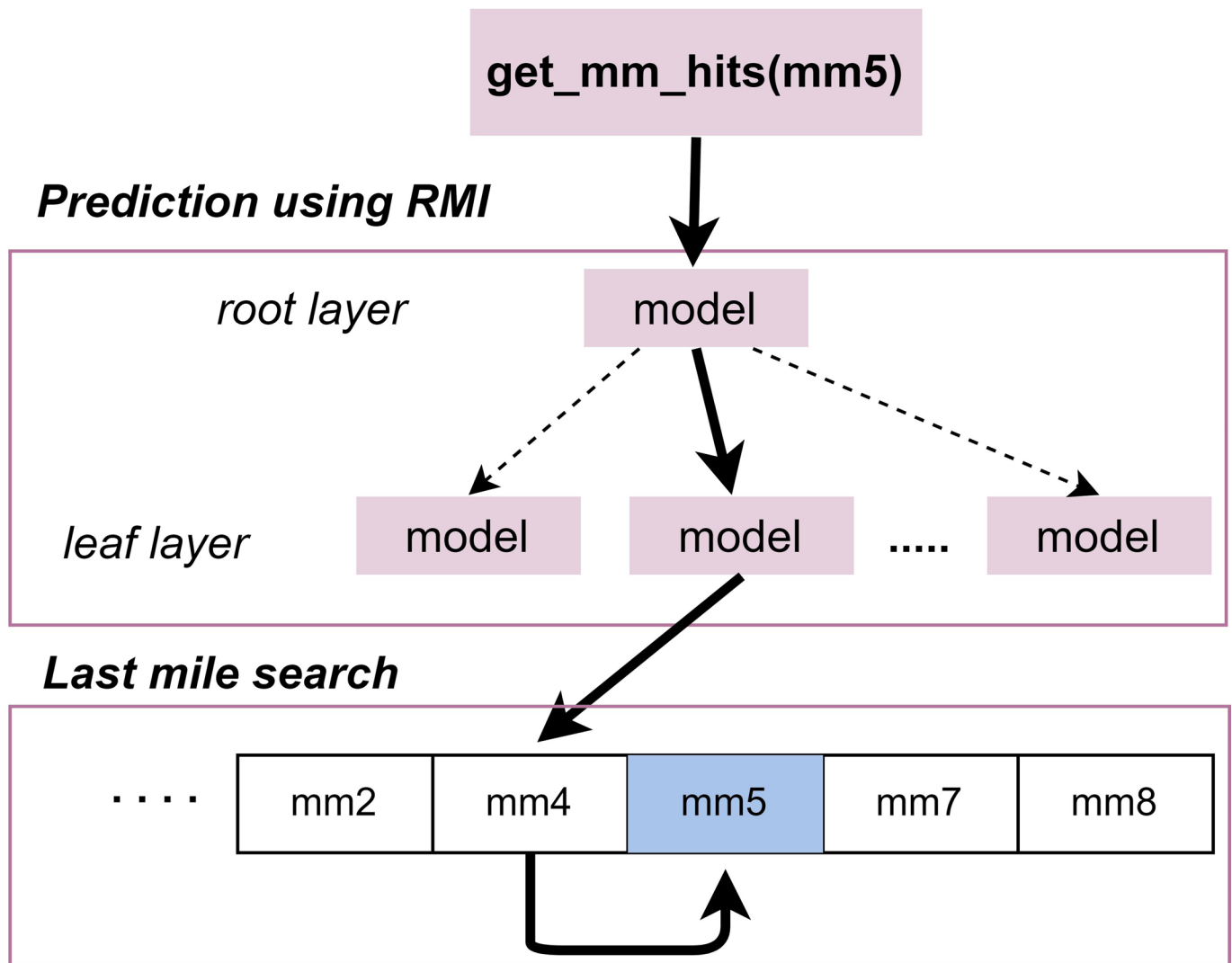
Keys Values

Position list

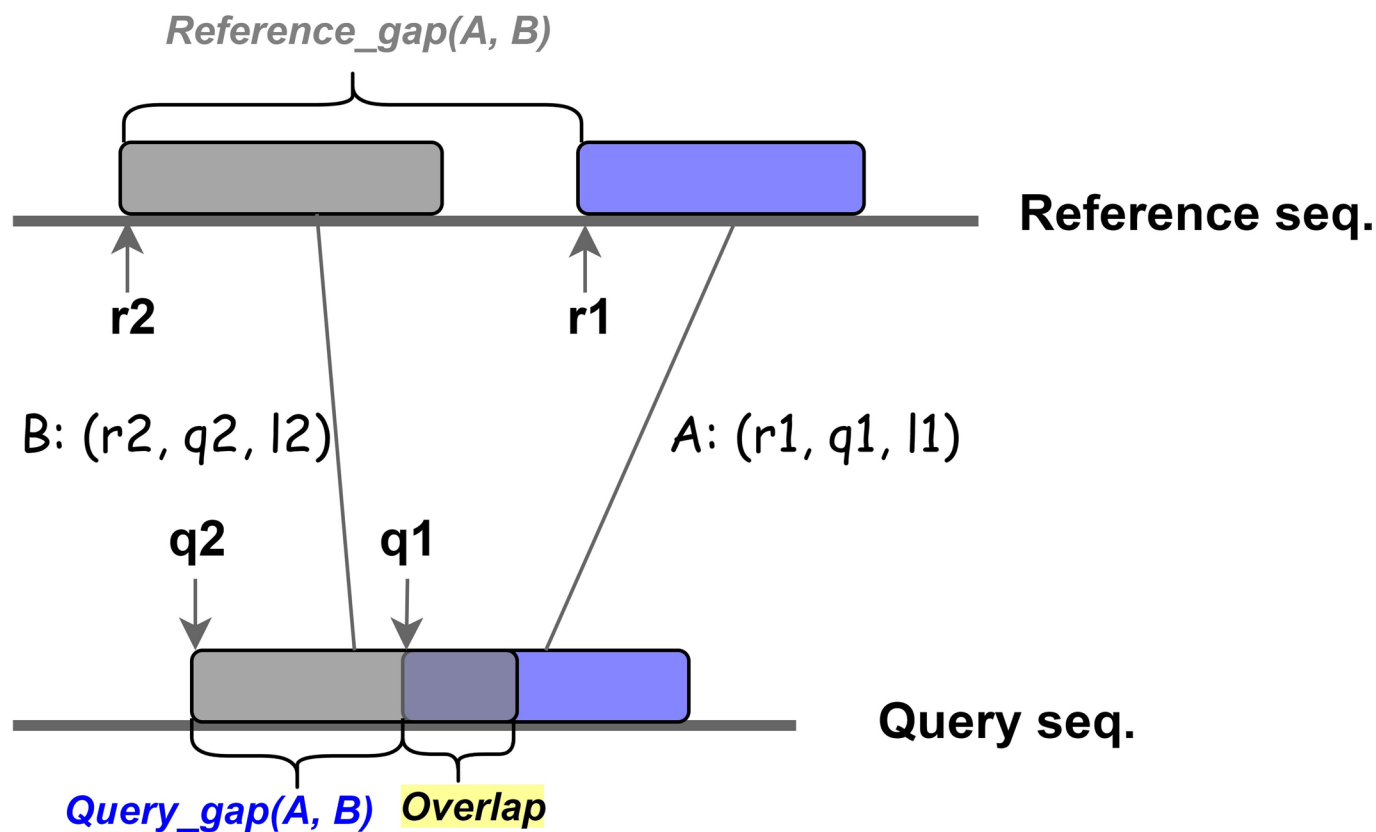
Keys	Values		
mm1	[0, 3]	0	6
mm2	[3, 4]	1	24
mm4	[7, 1]	2	32
mm5	[8, 3]	3	4
mm7	[11, 1]	4	9
mm8	[12, 1]	5	16
mm10	[13, 2]	6	67
mm12	[15, 1]	7	13
		8	5
		9	21
		10	57
		11	45
		12	17
		13	19
		14	33
		15	64



Extended Data Fig. 3 | Data structures used for hash table. Minimizers extracted from the reference sequence are stored in a sorted list as key-value pairs. Position list maintains a separate list of the positions of minimizers on the reference sequence.



Extended Data Fig. 4 | Two-layer RMI. An example minimizer lookup is illustrated - `get_mm_hits(mm5)` calls a lookup for a minimizer `mm5`. The RMI root predicts the leaf layer model which in turn predicts the location of `mm4` in the sorted list. Finally, the last mile search from `mm4` walks to the location of `mm5` and returns its value to the caller.



$$\text{gap_cost} : f(\text{Reference_gap}, \text{Query_gap}, \text{avg_qlen})$$

Extended Data Fig. 5 | Chaining of two co-linear anchors A and B. Here two anchors overlap on the query sequence. Gap cost function in minimap2 is calculated using the reference gap, query gap, and the average length of all anchors *avg_qlen*.