Accel-Align-RMI: a genomic sequencer powered by learned indexing.

Ilaria Pilo
Politecnico di Torino
Institut EURECOM
pilo@eurecom.fr

Abstract

This work explores the utilization of learned indices to enhance the performance of the genomic aligner Accel-Align [1]. While integrating the learned index into Accel-Align, it is important to consider that querying the index involves multiple lookups and additional handling of corner cases, which results in a 10x increase in lookup time compared to the original implementation.

However, a notable advantage of the integration is the reduction in index size by 2 GB. This reduction has a positive impact on the overall execution time, leading to a decrease of 15% in the total runtime of the aligner.

1 Introduction

Genomic alignment plays a crucial role in various areas of genomics research, such as variant calling, transcriptomics, and comparative genomics. The accuracy and efficiency of alignment algorithms significantly impact the analysis of large-scale genomic datasets.

In this work, we present the integration of learned indexing into the Accel-Align genomic aligner, aiming to enhance its efficiency and memory usage while maintaining alignment accuracy.

The report is structured as follows: Section 2 briefly describes the theoretical concepts necessary to understand this work. Section 3 presents and motivates the implementation choices we made, as well as the challenges we faced. Section 4 discusses the experimental results. Finally, Section 5 concludes the report, summarizing our main findings.

2 Background

2.1 Learned index

A *learned index* [2] is an innovative approach to indexing that leverages the computation and learning of the empirical Cumulative Distribution Function (CDF) of keys. This learned CDF is then utilized to predict the position of a key within the index, using the following relation

$$h(key) = \lfloor CDF(key) \cdot N \rfloor \tag{1}$$

where N is the length of the index.

One of the key advantages of such a model is its ability to construct a customized index that dynamically adapts to the unique distribution of keys encountered.

However, it is important to note that the use of a learned index requires re-training of the model and rebuilding of the entire index from scratch whenever a new key is inserted. This limitation implies additional computational overhead and necessitates careful consideration of the trade-off between adaptability and the cost of rebuilding the index.

2.2 RMI

The *Recursive Model Index (RMI)* [2, 3] is a hierarchical learned index that utilizes multiple layers of staged models. This hierarchical structure is essential for accurately representing the empirical CDF of keys, even at a micro-level.

The RMI index typically consists of a root model, intermediate layers, and a leaf layer. Each model is responsible for predicting the appropriate model to be used in the next layer, with the leaf model ultimately predicting the actual key.

When defining the RMI index, several choices need to be made:

- The number of stages, which should be a minimum of two (root and leaf).
- The number of models for each stage, also known as the width of the stage.
- The type of model for each stage, such as linear, spline, or cubic.

Figure 1 illustrates a simple example of a two-stage RMI model. In practice, two layers are often sufficient to achieve good performance, as long as the second layer is wide enough [3, 4].

2.3 Accel-Align and genomic aligners

Accel-Align [1, 5] is a fast genomic aligner that utilizes the seed–embed–extend method for rapid alignment of DNA sequences.

In general, a genomic aligner serves as a computational tool for aligning DNA sequences to a reference genome. Its primary objective is to accurately determine the position and orientation of short DNA fragments, commonly referred to as *reads*, within a larger reference genome.

Such a genome is a sequence of nitrogenous bases consisting of 3 billion characters, hence the target to which we aim to apply the RMI.

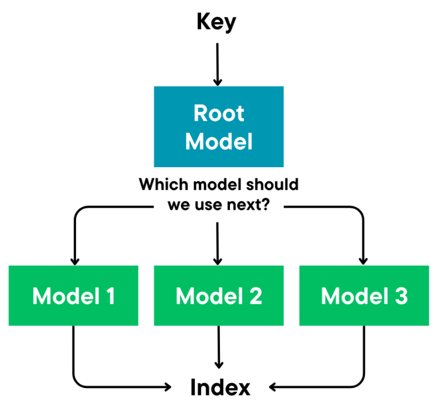

Figure 1: Example of two-stage RMI model.

3 Implementation

Within this section, we present the implementation techniques employed in the development of the index and its integration into Accel-Align. The following subsections detail the specific methodologies utilized throughout the implementation process.

3.1 Key generation

Before constructing the index, the first step involves generating the necessary `(key, value)` pairs from the reference genome. To accomplish this, we divide the genome into all possible subsequences (the so-called *k-mers*) having a length of 16 nucleotides. It is important to note that there are only four distinct nitrogenous bases - A, C, G, and T - which allows us to represent each base using just 2 bits. Consequently, every 16-mer can be represented by a 32-bit value, which becomes the corresponding key. The position of a key indicates the origin of its first element within the reference genome.

Once all keys have been derived, we proceed with the generation of the files `keys_uint32` and `pos_uint32`. These files serve as storage for keys and positions within the index, respectively.

Both files begin with an unsigned 64-bit counter indicating the total number of entries. In `keys_uint32`, each entry consists of a pair of 32-bit unsigned integer values representing the key and the cumulative position[1]. On the other hand, `pos_uint32` contains a simple list of 32-bit unsigned integer positions. The detailed generation process can be found in Algorithm 1. An example of the final structure is displayed in Figure 2.

It is worth noting that the content of the generated files remains constant given a specific reference genome. As a result, there is no need to rebuild the RMI index from scratch, eliminating the associated drawbacks and enhancing overall efficiency.

3.2 Dynamic library implementation

`RMI`[2] [6] is the most popular implementation of the Recursive Model Index, written in Rust and fully parallelizable.

[1]The cumulative position is the sum of positions associated with keys lower than the current one.
[2]`github.com/learnedsystems/RMI`

Algorithm 1 Binary files generation

```
sort keys in ascending order
=========== keys_uint32 ===========
write(number_different_keys, 8)
cumulative_pos = 0
prev_key = -1
for each k in keys do
    if k != prev_key then
        write(k, 4)
        write(cumulative_pos, 4)
    end if
    cumulative_pos ++
end for
=========== pos_uint32 ============
write(number_positions, 8)
for each p in positions do
    write(p, 4)
end for
```

The tool can work in two different modes:

- *train*: in the default mode, the tool trains the parameters of a specified RMI architecture. Moreover, it generates C++ code that must be utilized to incorporate the trained architecture in larger projects.
- *optimize*: the tool also offers an "optimize" mode, where it automatically fine-tunes the hyperparameters, returning a concise set of architectures that cover the Pareto front[3].

The original repository implementation suffers from a significant limitation, namely the *code generation* process. Since each index is tied to a specific C++ code, we need to recompile the entire project whenever a new index is employed.

To address this challenge, a novel approach is employed, treating the index as a library. This involves encapsulating the generated code within a shared object. This shared object can be dynamically loaded during runtime, eliminating the need for full project recompilation when switching between different indices.

Furthermore, to simplify the management of the shared object, a generic C++ class wrapper is utilized. This wrapper provides convenient control and interaction with the dynamically loaded index. These modifications are incorporated into my personal fork of the repository[4].

In addition to the aforementioned improvements, the fork also offers several new features, which further enhance the functionality and usability of the `RMI` tool:

1. *Removal of* `<filesystem>` *library requirement*: the generated code eliminates the dependency on the `<filesystem>` library. This adjustment ensures compatibility with older versions of C++ that may not support this library, thereby expanding the range of supported environments.

[3]In multi-objective optimization, the *Pareto front* represents the set of optimal solutions that cannot be improved in one objective without sacrificing performance in another objective.
[4]`github.com/IlariaPilo/RMI`

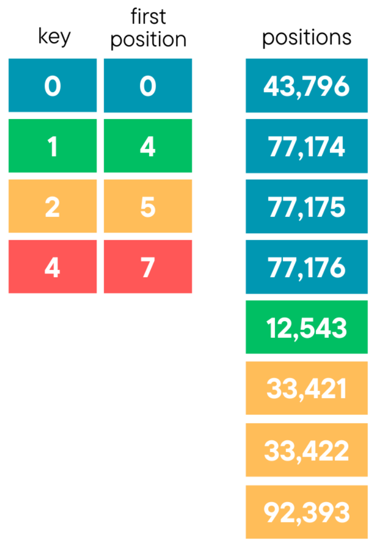

Figure 2: Example of the content of `keys_uint32` (right) and `pos_uint32` (left) files. For the sake of brevity, the number of entries in both files is omitted.

2. *Support for (key, value) structured files*: we introduce support for structured files containing (key, value) pairs, where the value is disregarded during the index training process. This is an important improvement, as these are the files generated from the reference genome in step 3.1.

3. *Enhanced optimization output*: the optimization process provides now improved output, including detailed information such as building time. This enhancement allows users to gain insights into the performance and efficiency of the index-building process.

These additional features contribute to the overall usability and versatility of the tool, making it a valuable and enhanced version of the original repository implementation.

3.3 Accel-Align integration

With all the necessary preparations in place, we can finally integrate the RMI index into Accel-Align.

To begin, we utilize the aforementioned library wrapper class to load the index and establish communication with it. This step enables us to leverage the functionalities of the index within Accel-Align seamlessly.

Next, we modify the existing file loading procedure in Accel-Align to accommodate the new format of the `keys_uint32` and `pos_uint32` files. This adjustment ensures that the index can correctly process and utilize the data contained in these files.

We proceed by implementing a helper function that performs a bounded binary search, using the output of the index lookup function, which includes predicted position and maximum allowable error. The binary search function aids in efficiently locating the desired key within the index structure.

Finally, we replace every direct index access in Accel-Align with a call to the binary search function. This substitution allows us to retrieve the desired information accurately and efficiently. Additionally, we handle any potential corner cases that may arise during the search process, such as cases where the requested value is not found in the index.

	Total time	**Lookup time**
Classic	20:13 min	7.79 s
RMI	17:03	89.2 s

Table 1: Total and lookup times for classic and RMI Accel-Align. Time benchmarks were computed with 12 CPU cores and averaged over 10 runs.

4 Evaluation

4.1 Hyperparameter optimization

To determine the optimal architecture for the index, we employ the *optimize* mode of the `RMI` tool, which provides us with a selection of the top 10 architectural choices. The optimization results, displayed in Figure 3, offer insights into the various architectures and their performance characteristics.

Observing the results, we notice a trend of decreasing complexity as we progress through the indices. After careful consideration, we select index 4 (`radix22,linear` with branching factor 1048576) as it strikes a balance between complexity and memory size, making it an appealing choice for our implementation within Accel-Align.

4.2 Aligner benchmarking

In this subsection, we conduct a comprehensive comparison between the original Accel-Align and Accel-Align-RMI, focusing on alignment accuracy, memory usage, and time complexity. We perform benchmarking on a dataset consisting of 10 million reads, with k-mer length set to 16. To reproduce these results, please refer to the instructions provided in the repository, as described in Section 6.

Accuracy. As the index does not affect the alignment strategy, we obtain the same accuracy with and without the RMI index (correctly aligned = 97.21%, exactly aligned = 97.13%).

Memory. The *position* portion of the indices depends only on the number of k-mers, therefore it remains unchanged. The *key* portion is compressed with respect to the original one, as we keep only existing k-mers. As a result, we successfully reduce the size of the key portion from 4 GB to 2 GB.

Time. Table 1 provides a comparison of execution times between the original Accel-Align and Accel-Align-RMI. Notably, due to the smaller size of the RMI index, the total execution time is lower (as we need to move less data from disk to memory). However, it is essential to consider that, while the original index utilizes direct access, the RMI index requires an average of 7 steps to locate the correct key, along with additional checks to handle corner cases. Consequently, the RMI approach is approximately 10 times slower than the original index.

```
Models                Branch          AvgLg2          MaxLg2         Size (B)   Build time (ns)
radix,linear        16777216         4.22363         8.00000        402653200      124443169288
radix,linear         8388608         5.02224         9.00000        201326608      112133104229
radix,linear         4194304         5.60928        10.00000        100663312      104003204641
radix22,linear       1048576         7.20767        10.98299         41943040       86061496663
radix,linear          524288         8.43821        12.91700         12582928       89637122826
radix,linear          262144         9.19067        13.94471          6291472       85086825486
radix18,linear         32768        12.08808        15.36150          1835008      107038194455
radix,linear           32768        12.25469        16.53535           786448       92663812165
radix,linear            1024        16.89976        21.13954            24592       93477981081
radix,linear             128        19.98812        23.62716             3088      110412216339
```

Figure 3: Output of the `RMI` optimizer, displaying the 10 most promising architectures of decreasing complexity.

5 Conclusion

In this work, we have presented the integration of the Recursive Model Index into the Accel-Align genomic aligner, along with various enhancements and optimizations. By leveraging the RMI, we achieved notable improvements in memory efficiency with respect to the original Accel-Align implementation. Additionally, we provided insights into the trade-offs between index complexity, memory usage, and execution time.

Through our benchmarking analysis, we observed that while the RMI index resulted in reduced memory requirements and overall execution time, it introduced a performance overhead due to the additional steps required for key lookup. Nonetheless, the benefits of reduced memory consumption and the adaptability of the RMI index make it a valuable addition to the Accel-Align aligner.

6 Source code availability

The source code of this project, as well as the instructions to run the benchmarks, are available at `github.com/IlariaPilo/accel-align-rmi`.

References

[1] Yiqing Yan, Nimisha Chaturvedi, and Raja Appuswamy. "Accel-Align: a fast sequence mapper and aligner based on the seed–embed–extend method". In: *BMC Bioinformatics* 22.1 (2021), p. 257. ISSN: 1471-2105. DOI: `10.1186/s12859-021-04162-z`. URL: `https://doi.org/10.1186/s12859-021-04162-z`.

[2] Tim Kraska et al. *The Case for Learned Index Structures*. 2018. arXiv: `1712.01208 [cs.DB]`.

[3] Marcel Maltry and Jens Dittrich. *A Critical Analysis of Recursive Model Indexes*. 2021. arXiv: `2106.16166 [cs.DB]`.

[4] Ibrahim Sabek et al. "Can Learned Models Replace Hash Functions?" In: *Proc. VLDB Endow.* 16.3 (Nov. 2022), pp. 532–545. ISSN: 2150-8097. DOI: `10.14778/3570690.3570702`. URL: `https://doi.org/10.14778/3570690.3570702`.

[5] Yiqing Yan and Raja Appuswamy. *Accel-Align Source Code*. `https://github.com/yanlolo/accel-align-release`. 2021.

[6] Ryan Marcus, Emily Zhang, and Tim Kraska. "CDFShop: Exploring and Optimizing Learned Index Structures". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2789–2792. ISBN: 9781450367356. DOI: `10.1145/3318464.3384706`. URL: `https://doi.org/10.1145/3318464.3384706`.

Acronyms

CDF Cumulative Distribution Function.

RMI Recursive Model Index.