Stockholms universitet

From Characters to Digits in Read Mapping

Från tecken till nummer i genomsekvensering

Rebecca Jana Nikkelen

Handledare: Kristoffer Sahlin
Examinator: March Hellmuth
Inlämningsdatum: 2025-08-25

Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

Abstract

The growing size of datasets places significant demands on memory and runtime efficiency in sequence alignment tools. This thesis investigates memory optimization of the genome mapper `strobealign` *by introducing a two-bit encoding scheme for nucleotide sequence storage. The encoding reduces the memory footprint during mapping and alignment, achieving up to a 14% reduction in peak usage compared to the original implementation, while maintaining competitive runtime performance relative to state of the art tools. Although the expected fourfold reduction (compared to the default eight-bit character encoding of nucleotides) was not fully realized due to additional overhead in data structures and file handling, the results demonstrate that sequence level encoding can contribute meaningfully to memory efficiency without sacrificing accuracy. The work highlights opportunities for further optimization, including integration of encoding with other pipeline components and exploration of compact multi sequence storage.*

Sammanfattning

Den snabbt växande storleken på biologiska datamängder ställer höga krav på minnes- och körtidseffektivitet i sekvensjusteringsverktyg. Denna kandidatuppsats undersöker minnesoptimering av programmet `strobealign` *genom att införa ett tvåbitars kodningsschema för lagring av nukleotidsekvenser. Kodningen reducerar minnesanvändningen under mappning av sekvenseringsdata, vilket resulterar i en minskning i minnesanvändning på upp till 14% jämfört med den ursprungliga implementationen, samtidigt som konkurrenskraftig körtidsprestanda bibehålls i förhållande till toppmoderna verktyg. Även om den teoretiskt förväntade fyrfaldiga minskningen inte fullt ut realiserades på grund av ytterligare overhead i datastrukturer, visar resultaten att sekvensnivåkodning kan bidra till minneseffektivitet utan att påverka noggrannheten. Arbetet belyser även möjligheter till vidare optimering, såsom integration av kodningen i fler pipeline-komponenter och utforskning av kompakt lagring av sekvenser.*

Contents

1 Introduction

Bioinformatics is currently a fast evolving field of research that combines mathematics, biology, and computer science into one subject [3]. The evolvement of more accurate sequencing, better performing computers and new methods gives us a better understanding on how DNA and RNA works. It also gives us a chance to make new discoveries. One of the main problems often revolves around project costs, what the research project can afford in terms of materials, technology, and server costs. Often one forgets two important aspects of processing technology, namely RAM and CPU. As computers are evolving, specifications get better.. However, reducing peak memory usage allows more concurrent programs to be executed on the same fixed resources, thus saving project turnover time and compute costs[9].

Currently, the research project associated with this thesis represents nucleotides in a genome as individual characters to symbolize each base. This character-based convention is intuitive and used due to its simplicity and readability. However, it is important to recognize that computers store data in bytes, where each byte consist of eight bits and a bit is a one or a zero. Regardless of the complexity of the character, one character will always occupy one full byte. As DNA sequences are composed of only four bases, for a smaller set of symbols needs to be stored. This small alphabet suggests a more efficient encoding is possible. Since two bits can represent four distinct values, a two-bit encoding scheme can be used to represent DNA sequences compactly. This opportunity for optimization has been well recognized in bioinformatics, especially in areas such as genome assembly and read mapping, where handling large volumes of sequence data efficiently is critical [4][6].

To use this, a proposal of implementing the following function

$$f : x \to y \quad \text{where} \quad x \in \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}, \quad y \in \{00, 01, 10, 11\}$$

where f is bijective. In this mapping, each nucleotide is represented using the two bits instead of the eight. As a result, the memory needs for storing a genomic sequence can be theoretically reduced by a factor of four.

A recently developed read mapper, `strobealign` however, does not use such encoding currently. `Strobealign` has shown impressive speed compared to other read mappers, but it has been observed to consume more memory compared to other mappers [13]. This poses the question of whether integrating a two-bit encoding into `strobealign` could reduce its memory usage, potentially at the cost of changes in runtime efficiency.

The aim of this thesis is to investigate the feasibility and performance tradeoffs of incorporation a two-bit DNA encoding into `strobealign`. This involves implementing the mapping function described above and benchmarking its impact on runtime and memory usage. By exploring this modification, we hope to contribute to the ongoing efforts in optimizing high throughput sequencing tools for both speed and efficiency.

Such an improvement would not only reduce the computational cost of sequencing larger genomes but would also enable more efficient use of limited computing resources. As most sequence analysis tools scale with input size, optimization at the data representational level becomes critical, especially when working with longer sequences or in memory-constrained environments.

1.1 Aims and purposes

The primary aim of this project is to investigate whether memory usage in the read mapping tool `strobealign` can be reduced by introducing a two-bit encoding scheme for DNA sequences. Currently, `strobealign` stores genome sequences using character based representations, where each nucleotide occupies one byte. This project proposes replacing this with a compact two-bit encoding, which can represent all four nucleotides using only two bits, potentially reducing memory usage by a factor of four.

The central objective is to implement this encoding scheme into `strobealign`'s sequence processing components while preserving the tool's core functionality and alignment accuracy. Consequently, functionality for converting back and forth between bit wise and character encodings are needed. Following implementation, we will conduct a comprehensive benchmarking analysis to evaluate how this change affects both peak memory consumption and runtime performance. Special attention will be given to understanding the trade-offs: while memory usage is expected to decrease, changes in runtime must be carefully measured and interpreted.

If successful, this optimization could allow `strobealign` to handle larger genomes or support more memory-intensive algorithms without requiring proportionally larger hardware resources. More broadly, the findings could contribute to ongoing discussions in computer science and bioinformatics regarding cost-performance tradeoffs in algorithm design and data representation.

In all computer science related fields, balancing runtime and memory usage is a well known and frequently analyzed trade off. As computational demands in genomics continue to grow, these kinds of optimization at the data representation level can have significant impact on scalability and cost-effectiveness.

2 Background

Computers when running programs have access to something called Random Access Memory storage (RAM), RAM works by efficiently labeling and storing the data in so-called bytes. A byte is what we call the collection of eight bits, where a bit is the representation of a zero or a one. This is only meant for short term storing, and is an efficient way of keeping track of data and using it quickly, but storage wise it is quite expensive and it will impact how much computing power the computer has, as it has to use RAM to do calculations and other operations [2].

A quick look on today's computers sold at popular electronic stores, shows that even the cheapest computer has at least four gigabytes (GB), or around $3.2 \cdot 10^{10}$ bits in RAM. Supercomputers on the other hand, the type of computers specifically used for heavy duty calculations which have a need for a lot of floating-point operations per second (also known as FLOPS), can have up to five petabytes of memory. This specific computer can be found in California, in the United States of America, and is used to test nuclear weapons without actually testing them in real life [5]. Supercomputers are increasing in commonality and have a wide variety of uses, but they are still very expensive in both acquiring them and in maintaining them. Meaning that even if theoretically one could end up with abundant of memory, it does not negate the need for optimization.

2.1 DNA and genomes

Deoxyribonuclease acid (DNA) is a long molecule composed of repeating units called nucleotides, each consisting of a phosphate group, a sugar, and a nitrogenous base. There are four different types of bases in DNA: adenine (A), guanine (G), cytosine (C), and thymine (T). These four bases form the fundamental links of DNA sequences. The specific order in which they appear encodes the biological instructions for the development, functioning, and reproduction of living organisms. In particular, sequences of DNA serve as templates for synthesizing new DNA strands or for transcribing into ribonucleic acid (RNA), which is then used in protein production [8]

A chromosome is a tightly packed structure of DNA and associated proteins, organized in such a way that allows for efficient storage and accurate separation during cell division. Each cell contains a full set of chromosomes, and the complete collection of an organism's chromosomes is referred to as its genome.

When working with DNA computationally, such as in read mapping or genome assembly, the DNA sequence is typically represented digitally using the four-character alphabet {A,C,G,T}. This simple and compact domain is well suited for optimization at the computational level. As discussed earlier, a promising approach is to replace the default character based storage, where each character consumes a full byte, with a two-bit encoding scheme. Since a genome can consist of billions of bases, even a small optimization at the base level can lead to substantial computational savings in terms of memory and processing efficiency.

2.2 FASTA files and read mapping

FASTA files are a widely used standard format for storing nucleotide or protein sequences in plain text. Each sequence entry in a FASTA file begins with a definition line, which starts with a greater than sign (>). This is followed by a unique sequence identifier and, optionally, a short description. The sequence itself appears on subsequent lines and is typically wrapped at 80 character for readability. Ambiguous nucleotide positions are represented by the letter "N" [11].

A closely related format, FASTQ, is commonly used for storing raw sequence reads produced by high-throughput sequencing machines. A read in this context references a readout from a genome sequencing machine. Unlike FASTA, FASTQ includes additional information, most notably, a quality score for each base, indicating the confidence of the sequencing instrument in its call. FASTQ files follow a similar record structure, where each read begins with an identifier line, followed by the nucleotide sequence. Due to the sheer number of reads generated in modern sequencing experiments, FASTQ files often reach tens of gigabytes in size[12].

An example of both formats is shown in Listing 1.

Listing 1: Example of FASTA and FASTQ entries

```
>Chromosome1 len=7
AGGTTTC
>Chromosome2 len=12
AGTCGCTGCGCG

@Chromosome1 len=7
AGGTTTC
+
))caaaf
@Chromosome2 len=12
AGTCGCTGCGCG
+
))caaaf*""###
```

Read mapping is the process of aligning these raw sequence reads back to a reference genome to reconstruct the original genomic sequence or to detect differences between the sample and the reference. In essence, it is like solving a puzzle: the short reads are the pieces, and the reference genome is the guide for placing them correctly. Read mapping is a computationally intensive task, as billions of read may need to be processed, compared, and matched efficiently. As such, the way sequences are stored and represented in memory can have a substantial impact on both runtime and memory usage[14]. There are also many different algorithms dealing with this, however, the one discussed in this thesis deals in hashing.

2.3 Strobealign and hashing

A hash function is a mathematical function that takes data of arbitrary size and maps it to a fixed-size value. In bioinformatics, hash functions are commonly used to index k-mers, which are sub-strings of length k extracted from a longer sequence. A sequence of length L contains $(L - k + 1)$ distinct k-mers, and storing or comparing them directly can be computationally expensive. Instead, each k-mer is transformed into a hash value, which reduces the storage footprint and allows for rapid lookup. Because the probability of two different k-mers producing the same hash value (a collision) is very low when using a good hash function, this approach provides an efficient way to store and retrieve sequence information, thereby speeding up mapping and alignment.

The hash function in `strobealign` already does convert the sequences it uses into bit encoding, which is how it fixes the size of the data, however it does it much later in the process than what is proposed in this thesis. The idea is that instead of doing the conversions when finding the

k-mers, the process is moved forward to minimize the memory allocation as quickly as possible. [13]

2.4 Current practices

Currently, each file being read into the program is represented by a vector in which each element is a string corresponding to a genome sequence. In addition, there are three supporting vectors associated with these sequences: the first stores the sequence identifiers (names), the second stores the sequence lengths, and the third holds the sequences themselves. Listing 2 illustrates this structure with a simplified example containing three sequences. For clarity, the object representing this data is referred to as `References`.

Listing 2: Strobealign's Current Layout

```
Object References:
    vector names = <name1, name2, name3>
    vector sequences = <"ACTG", "AAGC", "GTAC">
    vector lengths = <4,4,4>
```

Currently, `strobealign` stores sequences as character strings in C++ `vector` containers, alongside their names and lengths. While this approach is straightforward and allows efficient random access, each base is stored as an eight bit character, meaning that the short example above already requires 32 bits to represent four bases. In large datasets, this character based storage leads to significant memory usage[16].

The use of `vector` is appropriate for dynamic resizing and sequential storage but the underlying memory cost for character representation remains. This motivates exploring the two-bit encoding for nucleotide sequences, which could reduce the storage requirement by a factor of four while keeping the same access patterns for alignment process.

It should be noted that not all character encodings have the same size. Character encodings depend on the UTF standard used, a character can require anywhere from eight to thirty two bits. In this case, `strobealign` uses single byte characters, but the redundancy remains when compared to the two bit representation.

2.5 Proposed Changes

The main changes revolve around changing the reference class to involve sequences of two bits per base and packed into one byte. The reason why packing the bits is because most machines does not handle data that is smaller than a byte, eight bits, and thus to maximally optimize the memory we pack four bases into a byte. Listing 3 shows how it would look, with the same input as in Listing 2.

Listing 3: Changes to the reference class

```
Object References:
    vector names = <name1, name2, name3>
    vector sequences = <00011110, 00001001 ,10110001>
    vector lengths = <4,4,4>
```

Listing 3's biggest point is that each sequence is only one byte, compared to the four bytes needed for the same input.

The second main change is in addition to how we are converting from characters to bytes and vice versa. These encoding and decoding functions will be made accessible to the rest of the code base, as many units and integration tests operate directly on both the encoded and decoded representation of sequences. The encoding function must also account for sequences that do not have a length divisible by four, in those cases there will be padding added to complete the final byte.

To make the conversion process more concrete, Figure 1 illustrates the transformation between the character based representation and its two-bit encoding equivalent including the padding to it.

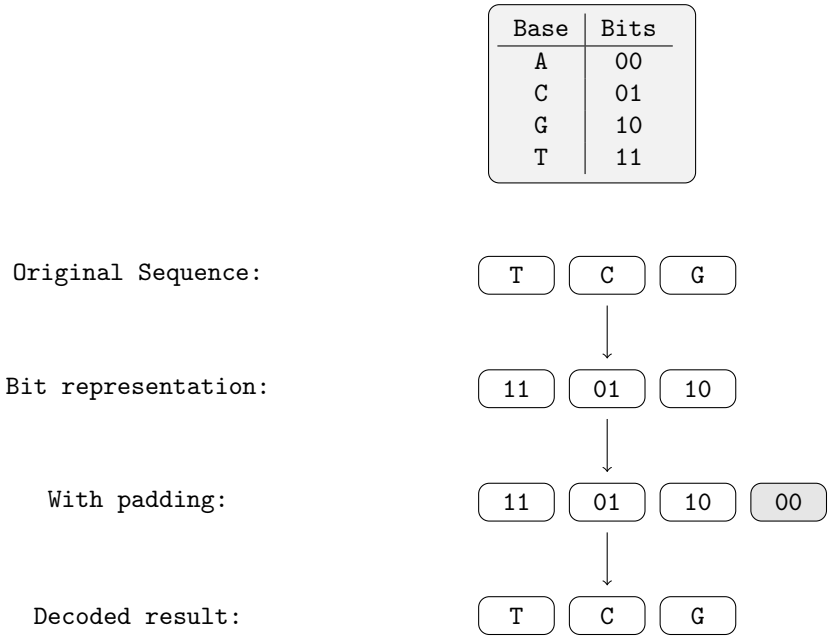

Figure 1: Example of encoding with included padding, and decoding back to the original sequence. Padding bits (gray) are ignored during decoding.

The decoding function on other hand will have to take the padding into consideration and have to know when to stop the decoding such that it does not accidentally add the padded sequences to it.

Another change that will be seen in the entire code base is the changing of data types in any other place where an object reference is made. This is because C++ is a strongly typed language thus each variable has to be declared with a data type. Now, there is always the option to use `auto&` however this has its own disadvantages. One of them being that it is not allowed in all contexts, for example it cannot be a type in a class, or it might actually lead to side effects [15].

2.6 The Code Base and Implementation Details

The `strobealign` code base is found at GitHub at `https://github.com/ksahlin/strobealign`, and the two-bit modified code base can be found at `https://github.com/rebjannik/strobealign`.

The project itself is open source and can be found under the name `strobealign` (as of 2025). The repository is organized into sub folders, separating the testing code from the main implementation. Each C++ class is split into a `.hpp` (header) file and a `.cpp` (implementation) file.

The `.hpp` files contains declarations of variables, functions, and classes. This separation enforces the requirement that the compiler must know about a variable or function before it can be used, thereby minimizing declaration-related errors. The `.cpp` files contain the actual function implementations and logic. When compiling, the compiler automatically incorporates declarations from the header files via the `#include` directive [10].

The primary files modified for this project are `refs.hpp` and `refs.cpp`, which contain the main logic for reading sequences and constructing the corresponding sequence vectors. However, many other classes rely on the reference class, requiring changes in additional. Most notably files requiring changes are `randstrobes.cpp` and `randstrobes.hpp`, both of which underwent significant, though not previously documented, modifications.

A key implementation detail is that both `refs.cpp` and `randstrobes.cpp` contain iterators for traversing sequences to compute hash values. In the two bit representation, these iterators must operate on two bit units rather than bytes or entire vectors, and must avoid iterating over any padding introduced during the encoding phase.

The test suite also required updates to accommodate new data types, while preserving the functional behavior of the existing tests. Many of these tests target edge cases and common scenarios for the respective components under test. One important check ensures that the two iterators produce identical results whether the input is provided as a raw nucleotide string or as a pre-encoded two bit sequence.

3 Methods

3.1 Main issues

The primary objective of this work is to implement and integrate two bit representation of genomic sequences into the existing code base. This encoding aims to reduce memory usage by storing each nucleotide in two bits instead of eight, while ensuring that all program functionality, such as sequence iteration, k-mer construction, and hashing, continues to work correctly with the new format.

3.1.1 Handling of Unknown Nucleotides

Genome sequencing can produce bases marked as "N" to indicated uncertainty. Since the two bit format relies on only four distinct values, representing "N" directly would require reverting to a full byte per base. This would negate any of the memory savings.

To avoid this, "N" is mapped to one of the four standard bases, using the same approach as in the hashing function via the `seq_nt4_table` lookup (see appendix A). This ensures compatibility without expanding the encoding size.

3.1.2 Iteration and Padding in two bit Sequences

In the original byte-based approach, each base was a separate element. With two-bit encoding each byte contains four bases without padding, or less with padding bits. If not handled carefully, iteration could process these padding bits as valid bases, producing extra k-mers and potentially incorrect results.

This was solved by modifying the iterators to step through sequences two bits at the time and introducing a length tracking variable., The iterator uses this stored length as a stopping condition, ensuring that only genuine bases are processed.

3.2 Main changes

The first big changes were in the constructor of the reference class. This was done by still originally taking a string when making the object, however within initialization the sequence was put through a packing function that would convert it to two bits for each base and pack it into a byte with three other bases.

Listing 4: pseudo code for converting a sequence

```
pack_sequence(str, sequence):
    for character in str:
        val = nt4_table(character)
        byte = bitoperation(val)
        count++

        if (count == 4):
            seq.pushback(byte)
            byte = 0
            count = 0

    if (count >0):
        padding(byte)
        seq.push_back(byte)
```

The pseudo code in Listing 4 can be divided into two main parts. The first is a normal loop that goes through each character within the genome sequence from a file. In the loop, it then uses a table labeled `nt4_table` and finds the byte character that the character should be converted to. The table helps with a fast conversion but also makes sure that it is consequent wherever else we would need to encode or decode a genome sequence. The next step is a bit operation where one first shifts the byte two spaces to the left, making space for the characters and then performing an or operation, or in other words we add the last bit to it. This is because the or operation in binary takes two binary inputs and returns a single binary, it returns a one if at least one of the inputs is one, and a zero if both are zero. Figure 2 shows how the sequence conversion goes from `G` to `GT`, or in other words, we are adding `T` onto the sequence.

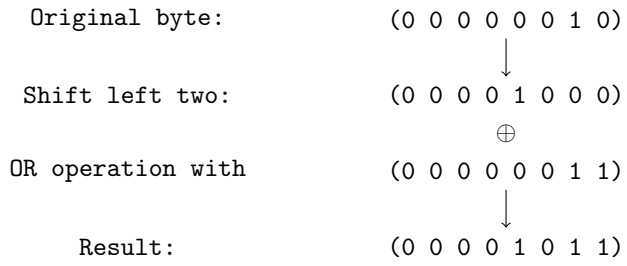

Figure 2: Bit operation from G to GT

The first if statement does the packing of the bytes, making sure that we end up with a byte each for the consequent four bases in a genome. The second part of this function is the padding, this is important for making sure that the last genomes are part of it but also that we can correctly pack it. It does mean that there will be some unnecessary bits involved if the whole sequence is not divisible by four, however, it is a necessary memory allocation to make sure nothing is lost when converting. It will also still be more memory efficient than previous practices, or at least theoretically.

The second main change is the unpacking, or decoding, function that is used to get the sequence in a string format rather than the bit encoding. This function can be seen in Listing 5, important

to note that this function takes both a sequence but also its length. This is to make sure that we does not accidentally convert the padding.

Listing 5: pseudocode for decoding a sequence

```
unpack_sequence(sequence, length):
  idx = 0
  for byte in seq:
    for( i = 0; i < 4 && idx < length; i++, idx++):
        val = extractFirstTwoAndShift(byte)
        result += "ACGT"[val]

  return result
```

.

The idea in the two loops is that the first only handles each byte for itself while the second loop handles the bit operations. The variable `idx` is to count how many bases we have converted from the encoded sequence, and to make sure the padding is not disturbed. The `extractLFirstTwoAndShift` is to use the last two of the bit sequence, `val` then becomes XX000000 where the XX symbolizes which base it is, and the entire byte shifts two to the left.

The method `substr()` was also implemented to make the `aln` class work. This was as the previous version made sure to use the inbuilt function `substring` that is present for strings. Listing 6 shows the implementation, however, it follows closely the logic shown in Listing 5.

Listing 6: pseudocode for substr implementation

```
substr(refID, start, length):
  for(i = start; i<start+length; i++;):

    byte = sequences[refID][i/4]
    shift = 2 * i%4
    val = bitExtraction(byte, shift)

    result += "ACTG"[val]

  return result
```

The biggest difference between the two is that instead of looping it over again, we calculate out the shifting needed first and then do a bit extraction. This is done as we might not start at the beginning of a byte, while in the unpacking we are guaranteed to start and end at a byte. Another important aspect that has not been mentioned thus far is that all sequences come with their own identifications/names, hence the `refID`. This is common practice in fasta files [11], and makes referencing to a specific vector easy, this is also why choosing the data structure vector was so important.

This concludes the main changes of the code base. There where more changes done that were not mentioned above, like fixing compilation errors.

3.3 Validation Process

The evaluation of this thesis focuses on three main criteria: accuracy, memory usage, and runtime performance. These metrics are assessed both for the modified version of `strobealign`, which implements two-bit encoding, and for other widely used read mappers.

Accuracy is measured by aligning reads from a known reference genome and calculating the displacement between predicted and true positions. Two aspects are considered: the accuracy of the alignment itself and the mapping step that places reads within the genome. This ensures that the introduction of two-bit encoding does not compromise the correctness of results compared to the original `strobealign` or alternative mappers.

Memory usage is the primary focus of this work. Peak RAM consumption during the mapping and alignment process is recorded for each tool. The aim is to determine whether the two-bit encoding achieves a substantial reduction in memory, and how this compares

The evaluation of this project focuses on three main criteria: accuracy, memory usage, and runtime performance. Both will be used for the modified version of `strobealign` and the current implementation of `strobealign`, it will also be tested against other widely used read mappers.

Accuracy measures how well the read mapper places sequences at their correct locations in the genome. This is assessed by aligning reads from a known reference and calculating the displacement between the predicted and true positions. This ensures that the two-bit encoding does not affect the core functionality or introduce alignment errors.

Memory usage is the primary metric for this project. Peak RAM consumption is recorded during the alignment process to determine whether the two-bit encoding achieves the intended reduction in memory or not without negative side effects.

Runtime is evaluated using CPU time, which measures only the processor's active work and excludes loading or compilation overhead. Tests are only run once and compared in the general sense

4 Results

The evaluation was performed on two genomic datasets, one the human genome (CHM13, three GB) and rye (seven GB. Each experiment was run across four configurations: the original `strobealign` (green), the modified `strobealign` with the two-bit encoding (pink), `minimap2` (blue) [7], and `BWA-MEM` (orange)[6]. The results are presented in three figures corresponding to the main benchmarking criteria: accuracy, runtime, and memory usage. Solid lines indicate alignment reads, while dashed lines represent mapping reads.

A mapping read refers to the initial placement of a read onto a reference genome without full base-level comparison, primarily used to quickly identify candidate regions for alignment. In contrast, an alignment read includes the full base by base comparison of a read against the reference, producing a detailed alignment that accounts for mismatches, insertions, and deletions[13].

Additionally there is a distinguish between single-end and paired-end reads. Single-end reads, consists of one continuous sequence read from a DNA fragment, whereas paired-end reads consist of two reads from opposite reads of the same fragment[1]. Only single-end reads were used in the evaluation.

Read length refers to the number of base pairs per sequencing read. The experiments include a range of 50 to 500 read lengths to evaluate how each tool performs as read length increases.

4.1 Overview

Overall, the results show that the two-bit encoding reduces memory usage substantially without sacrificing accuracy. Runtime performance remains competitive, although the modified implementation shows a notable increase in alignment time compared to mapping, a difference not observed to the same extent in other tools.

The most relevant trends can be observed by comparing the pink and blue lines: the pink lines show the modified two-bit implementation, while the blue lines show the unmodified `strobealign`. Together, these results demonstrate that two-bit sequence encoding can effectively reduce memory requirements without sacrificing alignment quality, albeit with a modest trade-off in alignment speed.

4.2 Accuracy

Figure 3 shows the accuracy of each tool over varying read lengths. The modified `strobealign` closely follows the original implementation, demonstrating that the introduction of two-bit encoding did not reduce correctness. Both versions of strobealign perform competitively with `minimap2` and `BWA-MEM` across all tested read lengths. Due to the close overlap between mappers, the pink line representing the two-bit implementation may appear brownish in regions where it overlaps with the other mappers.

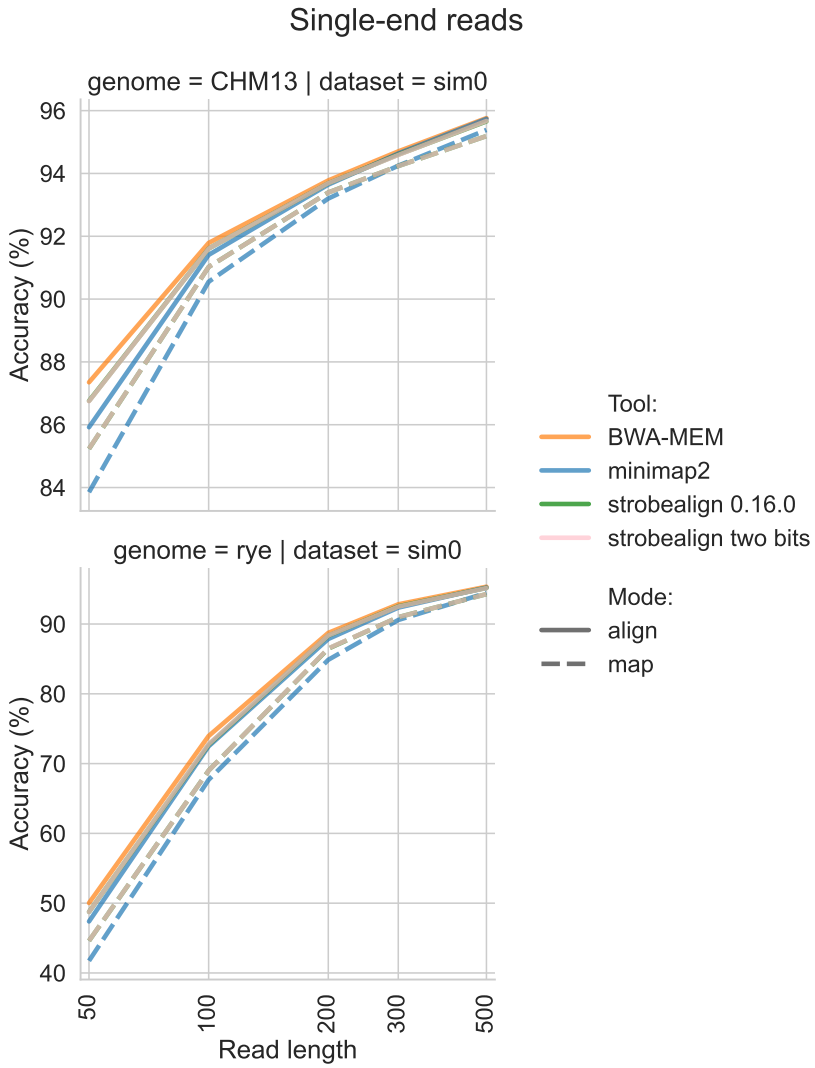

Figure 3: Alignment (solid) and mapping (dashed) accuracy over read length.

4.3 Runtime

Figure 4 reports the runtime required for mapping and alignment. As expected, the original `strobealign` remains among the fastest tools, and the two bit implementation shows comparable performance for mapping. However, alignment shows a noticeable slowdown compared to mapping, which is unusual given that the other tools exhibit smaller differences. `Minimap2` and `BWA-MEM` generally perform more slowly, particularly at longer read lengths.

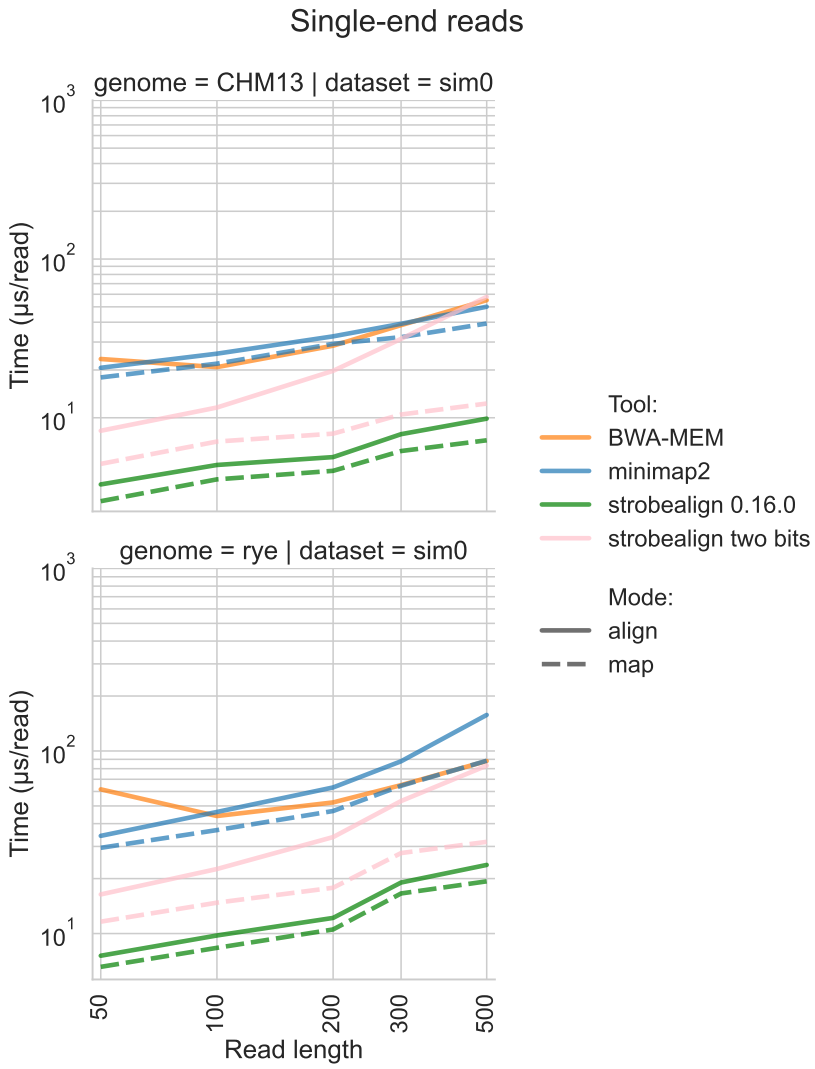

Figure 4: Runtime over read length.

4.4 Memory Usage

Figure 5 compares peak RAM consumption during execution. The modified `strobealign` achieves a substantial reduction in memory usage compared to the original version, confirming the expected benefit of the two-bit encoding. Although it does not outperform `BWA-MEM`, which is the most memory-efficient tool, it consistently uses less memory that `minimap2` at read lengths above 300, while the original `strobealign` only achieves this advantage above a read length of 500 nucleotides.

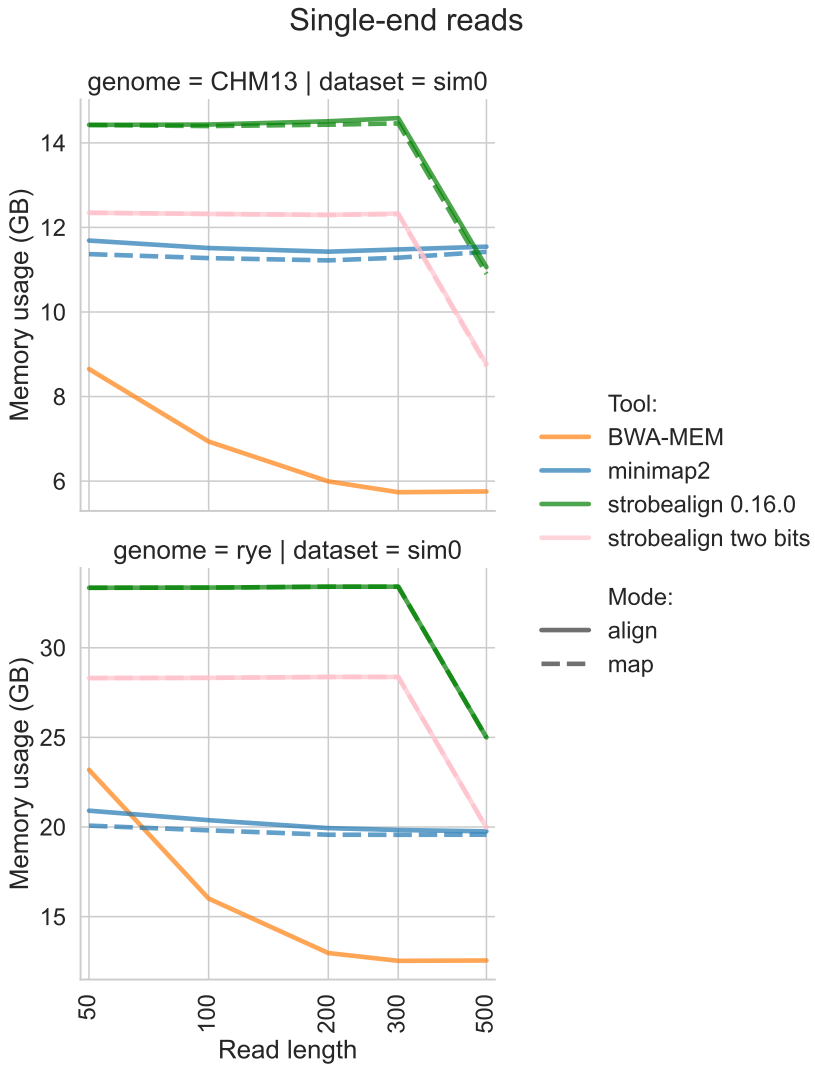

Figure 5: Peak memory usage over read length.

5 Discussion

The discrepancy between the theoretical and observed memory reduction can be explained by the fact that nucleotide sequence storage represents only part of the program's total memory footprint, Additional overhead arises from file parsing, data structures used during indexing, and auxiliary buffers needed during the mapping and alignment processes. These components dominate the overall usage, meaning that the optimization of base level sequence representation alone cannot account for a full fourfold reduction.

Furthermore, while the encoding itself was implemented correctly, there is potential for improvement in how it integrates with the rest of the pipeline. For example, reducing redundancy in how sequences are loaded, or streamlining intermediate data structures, may amplify the effect of the two-bit encoding. Similarly, improvements in cache locality and more efficient encoding/decoding routines could help reduce both memory consumption and runtime overhead.

An important observation is the unexpected spike in runtime during read alignment. This is most likely a side effect of additional encoding and decoding steps introduced by the current design. In this thesis, the hashing and alignment functions were deliberately left unchanged to avoid introducing new accuracy issues. As a result, if classes in `randstrobes.cpp` already applied partial encoding, the current implementation may now be unnecessarily decoding and re encoding data. This creates an inefficiency that explains why the runtime penalty is more visible during alignment than mapping. Addressing this redundancy is therefore an important step in further optimization.

5.1 Further research

Several directions remain open for further investigation

5.1.1 Resolving the Runtime Bug

As discussed above, the unusual runtime behavior in read alignment suggests redundant encoding and decoding. This likely stems from an interaction between the newly introduced two bit representation and pre existing bit level operations in `randstrobes.cpp`. Investigating and eliminating these redundancies could remove the alignment slowdown, while also further reducing memory usage.

There is also remaining future work to see if the bit wise operation could be improved upon, making the conversion faster, thus the program faster too.

5.1.2 Combining sequences

Another promising avenue is to explore storing multiple sequences in a compact, concatenated format, reducing per sequence overhead from identifiers, headers, and boundary markers. This approach could potentially unlock memory savings closer to the theoretical maximum, especially in datasets with millions of short reads. An example sketch of a possible data structure is shown in Listing 7, where metadata is stored separately from the bit encoded sequences:

Listing 7: Proposed changes to the reference class

```
Object References:
    vector names = <name1, name2, name3>
    bit sequences = 000111100000100110110001
    vector starting = <0,8,16>
```

6 Conclusion

This thesis set out to investigate whether a two-bit encoding scheme could reduce the memory footprint of `strobealign` while maintaining accuracy and runtime performance. The results demonstrate that the encoding does indeed save memory though no to the theoretical factor of four. As shown in Figure 5, the reduction amounts to approximately 14%, which, while smaller than initially hoped, is nonetheless a measurable improvement. Importantly, this gain does not come at cost of accuracy: the modified implementation produced nearly identical alignment and mapping correctness compared to the original version. Runtime was noticeably affected, as expected when introducing additional bit wise operations. However, excluding the runtime bug as mentioned above, both versions of `strobealign` remained competitive with `minimap2` and `BWA-MEM`. Overall, the two-bit encoding can be considered a successful optimization, albeit with more modest memory savings than anticipated.

7 References

[1] Illumina, Inc. *Advantages of paired-end and single-read sequencing.* URL: https://emea.illumina.com/science/technology/next-generation-sequencing/plan-experiments/paired-end-vs-single-read.html.

[2] Intel Corporation. *What Is Computer and Laptop RAM?* URL: https://www.intel.com/content/www/us/en/tech-tips-and-tricks/computer-ram.html. (accessed 25.06.2025).

[3] Shayna Joubert. *Earning a Bioinformatics Degree: Career Outlook and Job Prospects.* URL: https://graduate.northeastern.edu/knowledge-hub/what-can-you-do-with-a-bioinformatics-degree/. (accessed: 24.06.2025).

[4] Ben Langmead and Steven L Salzberg. "Fast gapped-read alignment with Bowtie 2". In: *Nature Methods volume 9* (2012).

[5] Lawrence Livermore National Laboratory. *El Capitan: NNSA's first exascale machine.* URL: https://asc.llnl.gov/exascale/el-capitan. (accessed 25.06.2025).

[6] Heng Li. *Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM.* 2013. arXiv: 1303.3997 [q-bio.GN]. URL: https://arxiv.org/abs/1303.3997.

[7] Heng Li. "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34 (2018).

[8] J.E. Luebering. *DNA.* URL: https://www.britannica.com/science/DNA.

[9] John C. McCallum. *Historical price of computer memory and storage.* URL: https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage. (accessed 24.06.2025).

[10] Microsoft. *Header files.* URL: https://learn.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=msvc-170. (accessed 02.07.2025).

[11] National Library of Medicine. *FASTA Format for Nucleotide Sequences.* URL: https://www.ncbi.nlm.nih.gov/genbank/fastaformat/. (accessed 07.07.2025).

[12] Pragmatic Genomics Limited. *Comparing FASTA and FASTQ Sequence Formats.* URL: https://sequenceserver.com/blog/fasta-vs-fastq-formats/. (accessed 10.08.2025).

[13] Kristoffer Sahlin. "trobealign: flexible seed size enables ultra-fast and accurate read alignment". In: *Genome Biol 23, 260* (2022).

[14] Sophie Schbath et al. "Mapping Reads on a Genomic Sequence: An Algorithmic Overview and a Practical Comparative Analysis". In: *Journal of Computational Biology* (2012).

[15] Standard C++ Foundation. *Placeholder type specifiers.* URL: https://cppreference.com/w/cpp/language/auto.html. (accessed 02.07.2025).

[16] Standard C++ Foundation. *std::vector.* URL: https://en.cppreference.com/w/cpp/container/vector.html. (accessed 27.06.2025).

8 Appendix A

Listing 8: The lookup table `seq_nt4_table` used for base encoding

```
unsigned char seq_nt4_table[256] = {
    0, 1, 2, 3,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 0, 4, 1,  4, 4, 4, 2,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  3, 3, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 0, 4, 1,  4, 4, 4, 2,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  3, 3, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,
    4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4,  4, 4, 4, 4
};
```

Datalogi
www.math.su.se

Beräkningsmatematik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm