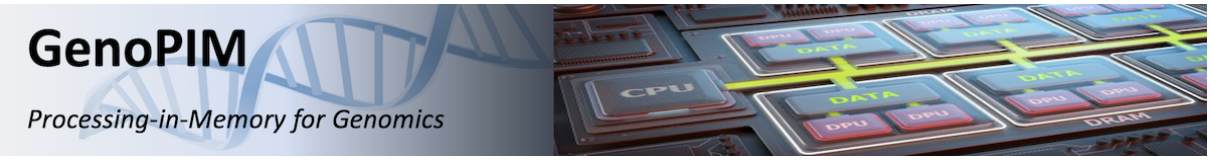


BWA-MEM on UPMEM Data Processing Units

Usecase

BWA is a read mapper. “Bwa mem” is a utility of this mapper that is particularly used for short read mappings (reads which are less than 300 nucleotides long). In the work we did, we only considered short read mapping onto a human genome (about 3 billion nucleotides).

High level algorithm overview

The bwa mem algorithm is divided into two main parts : seeding and chaining, which are done consecutively for each read.

First, a number of smems are found in a read. Smem are SuperMaximal Exact Matches. They are the longest exact match covering a position. An exact match is a section of a read which maps exactly on at least one part of the genome. The set of mappings in the genome is kept along with the part of the read that matches when looking for smems.

These are found for every position in a read. Although this does not mean that bwa iterates over the read positions to run an smem lookup. As the seeding algorithm can extend seeds both ways, it can return smems for multiple positions each time it runs. In order not to miss possible alignments, smems which are too long (above 28 base pairs) are cut-up so that more mappings can be considered (a smaller exact match will have at least as many mappings in the genome, often more).

Then once those seeds are found (smems and cut-up smems; all exact matches), they are chained. This means that bwa finds sets of non overlapping seeds which map to closeby regions of the genome. Chains are then filtered to reduce the amount of them to consider. After this step, starting with the most promising seeds of the most promising chains, seeds are “extended” using ksw2 to try and find a non exact mapping for the whole read.

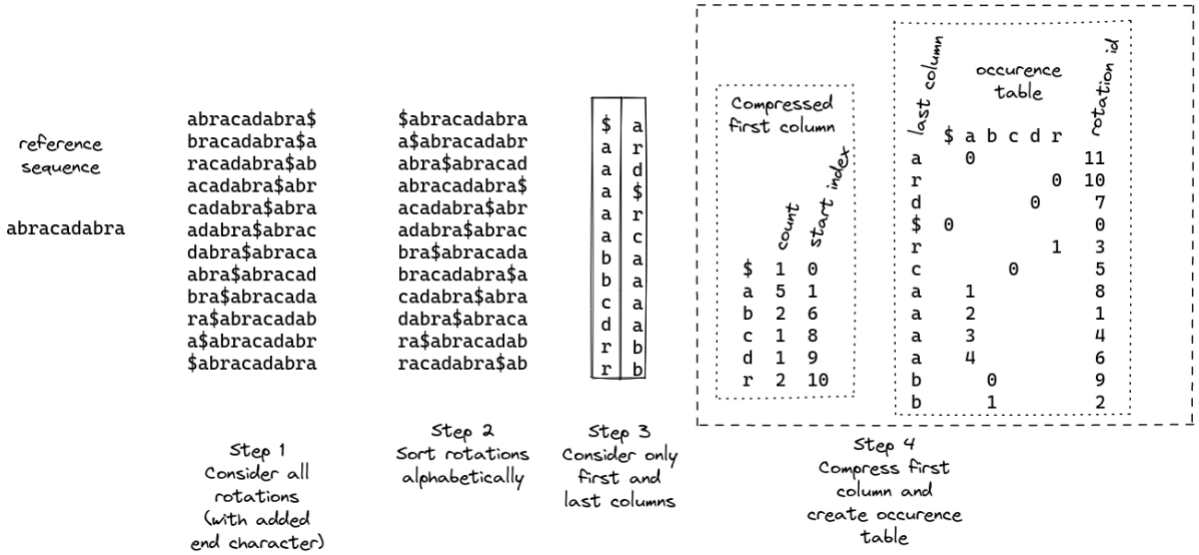
Seeding algorithm

FM-index and exact match search

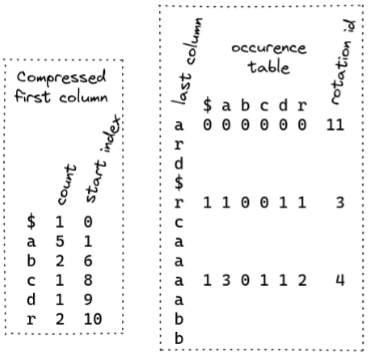
The FM-index

An FM-index is an index based on the bwt. It is in essence a lexicographically sorted list of all the suffixes of a string made to easily lookup the location(s) of any substring. It is made up of the bwt itself which is a list of the characters in the string sorted by the lexicographical index of the suffix of the string which starts right after the character. The index also contains a Suffix Array (SA). This is a list of the suffixes of the original string (again sorted lexicographically) represented by their starting position in the string. Note that the suffixes themselves can be found by cross-referencing the SA and the reference string. In practice, in BWA this SA is not stored in its entirety but is instead downsampled to save on RAM usage. This memory space

saving comes at the cost of more computations needed to find the missing values when they are needed. The FM-index also contains a list of occurence numbers which counts the number of times each character appears in the bwt between the start of the bwt and a given index. As with the SA, this information is fully redundant with the bwt and could in theory be computed directly from the bwt. Instead, in bwa, a downsampling factor is chosen to find a good balance between memory use and time taken to recompute any given value at runtime.



Creating a simple FM-index

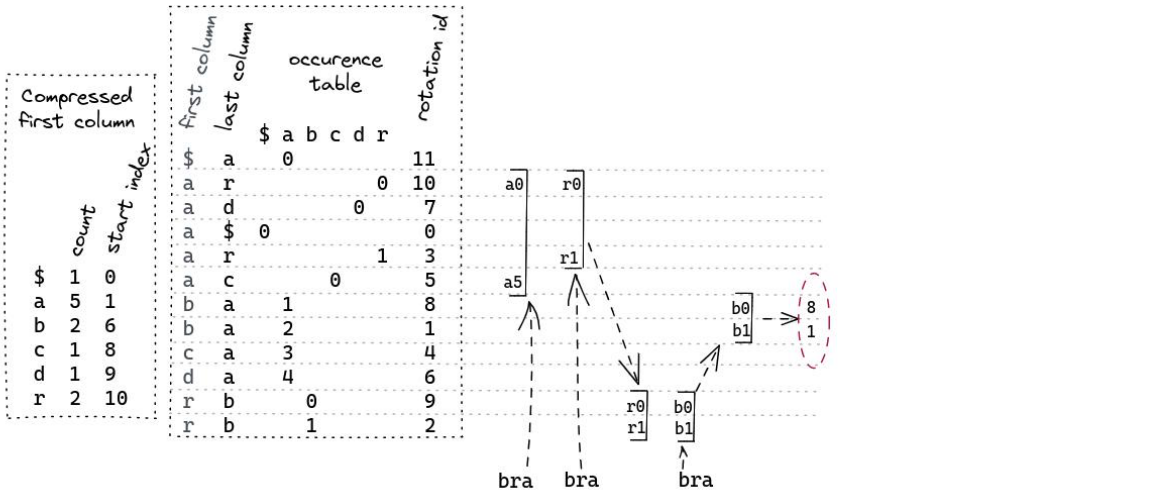


- To find occurrence count at a specific location, just count occurrences of considered letter between location and closest sampled line of occurrence table (and subtract/add result to number found in sampled line)

- To find rotation id at a specific location, continue jumping through the index, similarly to the way lookups are done, until you end-up on a sampled rotation id line.

Down sampling the occurence table

Exact match lookup in an FM-index



1. Starting range is first column range of last letter of query
2. Now consider previous letter in query
3. Reduce range to the smallest range containing all occurrences of considered letter within the previous range
4. Move start and end of range to <start index of considered letter>+<occurrence count at this index in table>
5. Repeat steps 2-4 until first letter of query or until range is empty
6. Look at all rotation numbers within range to get indices of query found in reference.

Querying a simple FM-index

Code analysis

worker1

mem_align1_core

mem_chain

mem_collect_intv

/ \

bwt_smem1
bwt_smem1a

bwt_seed_strategy1

bwt_extend

bwt_2occ4
bwt_occ4

bwt_extend

bwt_2occ4
bwt_occ4

In this work, we focus on the seeding algorithms. There are two main call stacks related to seeding. The call stacks share a common base: the functions which preprocess the read, which iterate over the read to call the seeding functions and which then call the

chaining functions over the resulting seeds. The call stacks also share a common ending as both seeding functions use the same functions to navigate in the fm-index.

Function roles

Starting from the leaf functions, the lower-level ones:

The bwt_occ4 and bwt_2occ4 functions are used to count the occurrence number of each nucleotide at a specific address in the index. This value is stored in the index but downsampled. Therefore, the functions need to lookup the last occurrence value before the query address and then count the remaining occurrences between there and the queried address. Note that the code could be slightly optimised by looking up the closest stored value and then counting up or down from there instead of always counting up from the prior value.

bwt_occ4 works on a single address of the index while bwt_2occ4 works on two addresses and has some slight optimizations by reusing some of the computations when the two addresses are sufficiently close to each other.

The bwt_extend function takes ranges in the index (represented by multiple bwt_intv_t struct as detailed below) which represent a current match. It then extends this match by taking a “step” in the index.

bwt_smem1 does pretty much nothing other than call bwt_smem1a.

bwt_smem1a and bwt_seed_strategy1 are the two seeding functions. The main difference is that bwt_seed_strategy1 takes in an address in the query and only tries extending the seed forward as much as possible. While bwt_smem1a first extends the seed forward as much as possible while storing intermediate results; it then extends the seeds backward as much as possible. It then returns all seeds (extended both forwards and backwards) which are not included in another.

mem_collect_intv takes in a read and it iterates over it to compute and then output all the found seeds in this read (all the smems which are not too long and multiple seeds where long smems were found). Note that seeds are returned as intervals in the fm-index which represent all the possible mappings for a given seed.

mem_chain first calls mem_collect_intv and then chains the found seed mappings.

mem_align1_core calls mem_chain and then does some post-processing and filtering on the chains which were found and then transforms them into alignments (with a ksw2 function being called further down the line)

worker1 is the function which is called on each read (or pair of reads) through a kthread loop. It then calls mem_align1_core on the read or on each read of a pair.

Data structures

Note here that bwtint_t is a uint64_t.

```C/C++
typedef struct {
	bwtint_t x[3]
	bwtint_t info;
} bwtintv_t;
```

bwtintv_t corresponds to the exact matches of a given substring. It represents two ranges. x[0] and x[1] are the starts of those two ranges and x[2] is the length of both ranges

which corresponds to the number of matches. The two ranges both represent the same matches but one is used for forward search (x[1]) and the other for backward search (x[0]). Note that both need to be updated when extending the match as they represent the same results. info is used as a miscellaneous storage. In the parts which are of interest to us, info is mostly used as two separate uint32_t. The most significant bits are used to store the index in the query sequence corresponding to the start of the seed while the least significant bits are used to store the index of the end of the seed in the query sequence.

```
C/C++
      typedef struct {
             size_t n, m;
             bwtintv_t *a;
      } bwtintv_v;
```

Bwtintv_v corresponds to a list of seed matches. The bwtintv_t structs pointed to by the “a” pointer can each correspond to different seeds in the same query sequence. The “info” field of bwtintv_t is used to store which seed each range refers to. This bwtintv_v struct is meant to be a resizeable list. The “m” field corresponds to the size of the list which has been allocated in “a” while “n” corresponds to the number of items already stored in “a”. To abstract away these size fields, macros such as kv_push can be used to manage bwtintv_v structs.

DPU acceleration

Problems

Lookups in the FM-index consist of small operations separated by chaotic jumps through the index. Those chaotic jumps lead to a lot of cache misses in BWA as it is working at the moment. But it also means that a naive offloading of the index and the work to the dpus would require sequentially sending the same request to multiple different dpus. Even batching requests would still lead to a lot of memory transfers. So much so that memory transfers alone would make this code slower than its CPU counterpart.

Algorithm alternatives

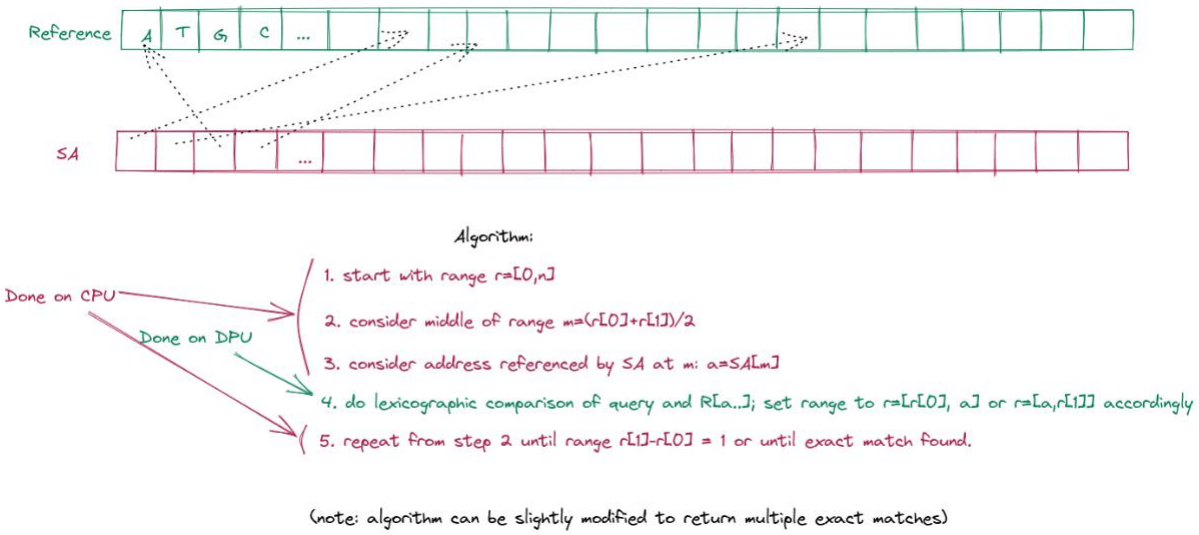
The goal of this work is to produce an acceleration of BWA while not modifying its outputs. This means the keypoint here is to find an algorithm with fewer chaotic jumps through the memory while providing the exact same results.

Binary search in SA

One alternative algorithm found is to use the suffix array directly without using the bwt itself. The idea is that as the suffix array is in essence a lexicographically sorted list of suffixes, an exact match can be looked-up by doing a binary search.

This algorithm has a slightly worse complexity in theory, although in practice, with the size of the datasets we use the amount of operations is somewhat similar. And the amount of chaotic jumps is greatly reduced (about 30 jumps needed for a binary search in human genome).

In practice, the suffix array doesn't store the suffix themselves but stores pointers to where they start in the reference. This means that when implementing the binary search, for each comparison, the reference needs to be loaded at the address of the start of the considered suffix, before that suffix can be compared to the query sequence. The idea here would be to accelerate this comparison step by offloading it to DPUs. the reference sequence would be cut up and stored in multiple DPUs. And instead of loading a specific part of the reference to do the comparison on the CPU, the query sequence would be sent to the DPU which stores the required part of the reference.



Binary search in SA design for DPUs

Implementation

The binary search algorithm has been implemented on CPU. the code can be found here : https://github.com/upmem/bwa

As the algorithm uses a subset of the same index as the rest of bwa, the index creation part has not been modified. Most of the modifications were done in the file bwt.c which is the file where smem lookups were done. There exists multiple seed lookup functions but as the binary search in SA uses a subset of the same index and as it is supposed to give the exact same results, the two algorithms could coexist. In order to get to a first result quicker, only the first part of the bwt_smem1a function was adapted to the new algorithm.

There were two goals with this implementation: validate that the output can stay the same with the new algorithm. And estimate the performance improvement that could be achieved if the algorithm were to be adapted to DPUs.

To implement the binary search in the SA, the SA lookup function already exists (bwt_sa) and did not need to be reimplemented.

A sa_binary_step function was created to get an approximate midpoint between two indexes while somewhat privileging indexes which are sampled in the SA to limit the amount of times the SA value has to be recomputed.

A compare_suffix function was created. Given an index in the SA, it loads the appropriate section of the reference sequence. It then compares the query sequence with the fetched section of the reference and outputs the comparison. The comparison result's sign corresponds to which sequence is greater lexicographically. The result's absolute value corresponds to the amount of base pairs matched.

The sa_binary_search is the function which does the forward extension of a seed. It does a binary search to find the best matching sequence (where it has the longest common prefix). However, there might be multiple matches of the same length. To ensure all of them are returned, two more binary searches are done to find the start and end of the range in the SA with the best match length.

The sa_smem function is the one which replaces the bwt_smem1a function. The bwt_smem1a function was divided into, first a forward search section followed by a backward extension. The forward search was replaced with a call to sa_binary_search. Most of the rest of the function was copied from bwt_smem1a.

In the branch "sa_match_extension", a function sa_extend_match was created (and not tested) which could be used to adapt the backward extension. This work has been left uncompleted.

The rest of the functions in this callstack were mostly left untouched except to ensure the sa_smem function gets called with the proper arguments.

Results

As for the validation of the outputs, they do differ slightly. In the dataset used for testing, about 95% of reads were mapped in the same way as the original bwa. However, for about 5% of reads, the mappings differ. After some consideration, the origin of this reference was attributed to an oversight in the adaptation of the first part of the bwt_smem1a function. As a result, not all the smems are found. Which would lead to different mappings for a small portion of reads. And this is likely the main culprit in the different results.

This oversight could be solved somewhat easily. This work was started in the branch https://github.com/upmem/bwa/tree/sa_match_extension.

Although before finishing this fix, it was decided to first do a benchmark of the solution as it currently is. Indeed, while a promising result would not be final, we know that the fix will only make the algorithm slower. Which means we can already get to a first decision of whether we want to go forward with this solution or not.

As for the benchmark, while it should be noted that we expect a fully functional adaptation to be slower, a first interesting result was that the time taken for mapping the same dataset was roughly the same (within 10% difference).

But what is more of interest to us is the share of that time which is taken by function which can be offloaded to DPUs

The code implemented on CPU has been benchmarked through vtune.

Unfortunately, it was found that of the time taken by the sa_smem, about 75% is taken by bwt_sa. Despite the efforts made to reduce the need for recomputation of SA values. And the bwt_sa function was not planned to be offloaded to DPUs. This means that DPU acceleration could only hope to accelerate about 25% of this section of code. It is not

expected that other sections of code would get a much bigger share of work which could be offloaded to DPUs.

Sources

https://arxiv.org/abs/1303.3997
https://github.com/lht3/bwa