Inexact Sequence Mapping Study Cases: Hybrid GPU Computing and Memory Demanding Indexes

José Salavert[1]*, Andrés Tomás[1], Ignacio Medina[2], and Ignacio Blanquer[1]

[1] GRyCAP department of I3M, Universitat Politècnica de València,
Building 8B, Camino de vera s/n, 46022, Valencia, Spain
josator@i3m.upv.es,antodo@i3m.upv.es,iblanque@dsic.upv.es
[2] Bioinformatics department of Centro de Investigación Príncipe Felipe
Autopista del Saler 16, 46012, Valencia, Spain
imedina@cipf.es

Abstract. Due to the NGS data deluge, sequence mapping has become an intensive task that, depending on the experiment, may demand high amounts of computing power or memory capacity. On the one hand, GPGPU architectures have become a cost-effective solution that outperforms common processors in specific tasks. On the other hand, out-of-core implementations allow to directly access data from secondary memory, which may be useful when mapping against big indexes in systems with low memory configurations. In this paper we discuss the implementation of backward search methods for inexact mapping in these two study cases.

Keywords: Inexact Mapping, FM-Index, GPU, MMAP

1 Introduction

With the advent of next generation DNA sequencers [3][7] in the mid-2000s, the term *Next Generation Sequencing* (NGS) emerges. NGS sequencers have been constantly improved, generating a genomic data deluge [4] due to their increased performance and lowered operating costs. A topic actively revised to satisfy NGS needs is the alignment of DNA sequences [11]. In the field of bioinformatics, we refer with the term sequence mapping to the alignment of small reads against a DNA.

Sequence alignment may reveal functional or evolutionary relationships between genes or proteins. Furthermore, the existence of DNA similarities between a patient and an individual with a detected genetic disease may be used effectively in diagnostic medicine. In order to detect these similarities, a sequence

* The authors would like to thank the Universitat Politècnica de València in the frame of the grant "High-performance tools for the alignment of genetic sequences using graphic accelerators (GPGPUs)", research program PAID- 06-11, code 2025.

mapping algorithm must allow a certain number of errors (insertions, deletions and mismatches).

Nowadays, several mapping techniques are based in backward search methods over *Suffix arrays* [16] (SA). Some of these methods require the generation of an index with the *Burrows Wheeler Transform* [2][17] (BWT). One of these indexes is the *Ferragina and Manzini Index* [5] (FM-Index), with applications in sequence mapping [10].

Among the different available choices that provide faster computation models, *General Purpose Graphic Processing Units* (GPGPUs) based on CUDA [21] or OpenCL [20] are a very cost-effective option. Thanks to these frameworks the GPGPU architecture can be exploited efficiently in general purpose problems, taking into account its micro-grain parallelism and memory hierarchy. Among the mapping tools supporting GPGPU computing we can name SOAP3-dp [13], CUSHAW2 [14] and Barracuda [8].

The computational complexity of backward search methods grows exponentially with the number of errors allowed. For this reason they are commonly employed to find the pair-ends of a read or as a seeding step before a local alignment algorithm.

In the first study case, we overview a real hybrid CPU-GPU algorithm based on backward search. This algorithm separates the computation that will achieve a higher speed-up on the GPU from the computation that will run better on the CPU. The computation done in the GPU is used to obtain both the pair-ends of a read and, with very little CPU overhead, the mappings with one error. The overall implementation takes advantage of the GPU computing power, which reflects in an optimised speed-up. This is an extension of previous work [24], in which we developed an FM-Index implementation for GPUs.

In the second study case, we compare the performance of our preprocessing algorithm using *csalib* out-of-core index [1] against similar algorithms that work on main memory, like those found in Bowtie 1 [9] and SOAP2 [12]. This implementation is useful when working with big indexes in systems with low memory profiles. We describe the compatibility interfaces that allow to use our algorithm with different index implementations easily.

The manual, source code and datasets are available at (`http://josator.github.io/gnu-bwt-aligner/`). The algorithms presented here are currently being included in the OpenCB pipeline (`https://github.com/opencb`). The source code is distributed under the LGPLv3 license terms.

2 Theoretical approximation

Let $A = \{A, C, G, T\}$ be an alphabet, and \$ a symbol not included in A with less lexicographic value than all the symbols in A. Let X be a reference string terminated by \$ with size n. Let $X[i] = a_i$ be the i-th symbol of string X.

We construct the suffix array of X in matrix M by rotating the values of the reference string. Then, we sort alphabetically matrix M to obtain the SA of the reference genome [23]. From the SA we obtain the BWT, which can be used to build an index of the reference genome. However, recent approaches compute the BWT directly [22], obtaining the compressed SA later [6].

For example, given the string "AGGAGC$":

$$M = \begin{pmatrix} A\,G\,G\,A\,G\,C\,\$ \\ G\,G\,A\,G\,C\,\$\,A \\ G\,A\,G\,C\,\$\,A\,G \\ A\,G\,C\,\$\,A\,G\,G \\ G\,C\,\$\,A\,G\,G\,A \\ C\,\$\,A\,G\,G\,A\,G \\ \$\,A\,G\,G\,A\,G\,C \end{pmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} \quad \rightarrow \quad SA = \begin{pmatrix} \$\,A\,G\,G\,A\,G\,C \\ A\,G\,C\,\$\,A\,G\,G \\ A\,G\,G\,A\,G\,C\,\$ \\ C\,\$\,A\,G\,G\,A\,G \\ G\,A\,G\,C\,\$\,A\,G \\ G\,C\,\$\,A\,G\,G\,A \\ G\,G\,A\,G\,C\,\$\,A \end{pmatrix} \begin{matrix} 6 \\ 3 \\ 0 \\ 5 \\ 2 \\ 4 \\ 1 \end{matrix}$$

Let S be an array composed by a permutation of the integers $0 \dots n-1$. After the ordination of the SA, S contains the original positions of each suffix in the reference. Vector B always corresponds with the elements of the last column of matrix M.

$$S = \begin{pmatrix} 6\;3\;0\;5\;2\;4\;1 \end{pmatrix} \quad B = \begin{pmatrix} C\;G\;\$\;G\;G\;A\;A \end{pmatrix}$$

Vectors S and B constitute the BWT, from which we obtain the FM-Index [5] data structures needed by the search algorithm. Let $C(a)$ be the number of symbols in X (excluding \$) lexicographically smaller than $a \in A$:

$$C = \begin{pmatrix} 0\;2\;3\;6 \end{pmatrix}$$

Let $O(a,i)$ be the number of occurrences of symbol $a \in A$ in $B[0:i-1]$ (first column denotes position -1, before the start of the search):

$$O = \begin{pmatrix} 0\,0\,0\,0\,0\,0\,1\,2 \\ 0\,1\,1\,1\,1\,1\,1\,1 \\ 0\,0\,1\,1\,2\,3\,3\,3 \\ 0\,0\,0\,0\,0\,0\,0\,0 \end{pmatrix} \begin{matrix} A \\ C \\ G \\ T \end{matrix}$$

Regarding matrix O this data structure must be compressed in order to fit the GPGPU memory. We use a compression scheme similar to the one described in [10], but using bitcount operations which are implemented at hardware level on the GPGPU. Vector S can be compressed following [6], but it remains in the CPU main memory.

We also define S_r, B_r, and O_r as the data structures of the BWT of the reverse reference. This reverse FM-index allows to analyse the read in forward direction.

Let W be a substring of X. All suffixes that contain W as prefix are listed together between a unique interval $[k,l]$ in the SA. Algorithm 1 consists on a backward search that determines if W is a substring of X in $O(|W|)$ time. On each step, a symbol of W is analysed obtaining new values of $[k,l]$ for the current substring. At the beginning $k=0$, $l=|S|-1$ and $pos=|W|-1$. On each iteration this interval is approximated. At the end, if $k \leq l$ string W belongs to X. We return the result in variable r using a special notation ($[k,l]$ with $error$), this means that we return interval $[k, l]$ with a single possible *error* received as parameter. When the algorithm has finished we can use vector S to recover the original positions in X of the values between $[k, l]$.

Algorithm 1 Exact Backward Search

```
1: exact_backward(IN: $W, [k,l], pos, error, index$. OUT: $r_{out}$).
2:   for $i \leftarrow pos \ldots 0$
3:     $[k,l] \leftarrow$ search_iteration($[k,l], W[i], index$)
4:     if $k > l$ break
5:   end for
6:   $r_{out} \leftarrow [k,l]$ at -1 with $error$
7: end function
```

In the case of the FM-Index **search_iteration** $\leftarrow$ **fm_iteration** (algorithm 2). A more detailed explanation can be found in our previous publication [24].

Algorithm 2 FM-Index iteration

```
1: fm_iteration(IN: $[k,l], b, index$. OUT: $[k', l']$.)
2:   $k' \leftarrow index.C[b] + index.O[b][k] + 1$
3:   $l' \leftarrow index.C[b] + index.O[b][l+1]$
4: end function
```

3 Hybrid GPU algorithm

Some bioinformatics applications, like RNAseq analysis [19][18], take advantage of backward search methods to find the pair-ends of a sequence, whose contents are then analysed with a local alignment algorithm. The computation of the pair-ends can be effectively done in a GPGPU and then combined in an hybrid pipeline with a lightweight CPU function allowing one error sequence mapping. This special case is an improvement over non-hybrid exact mapping on GPU. Its main advantage is that it allows to find the pair-ends plus one error mappings with very little overhead. Moreover, it constitutes a real hybrid computation approach: some steps are executed on the GPU and the rest on the CPU (algorithm 3).

The hybrid inexact mapping method is based on the pseudo-code in algorithms 4 and 5, which describe the backward direction routines. This design separates the code that will execute better in the *Single Instruction Multiple Data* architecture of the GPU from the code that will be faster on the CPU.

The **vector_gpu** functions (algorithm 4) return the values of all the subsequent $[k, l]$ intervals calculated during an exact search of a read W. These values

Algorithm 3 Sequential execution

```
1: main()
2:    $vk, vl, pair \leftarrow$ backward_vector_gpu($W, index$)
3:    $vk_i, vl_i, pair_i \leftarrow$ forward_vector_gpu($W, index$)
4:    $results$ += backward_helper_cpu($W, vk, vl, pair_i, index$)
5:    $results$ += forward_helper_cpu($W, vk_i, vl_i, pair, index$)
6: end program
```

are stored in vectors *vk* and *vl*. Also, they return the last position with an interval satisfying $k \leq l$ in variable *pair*, this is used to obtain the pair-ends after the execution of the algorithm.

When an interval does not satisfy $k \leq l$, all the remaining elements of the vector are filled with the last non-satisfying values of $[k, l]$. The time consumed by the filling loop in the GPU is insignificant and it is needed to stop the exploration in the **helper_cpu** algorithm when reaching the position where a substring is not present.

Algorithm 4 Backward Vector GPU

```
1: backward_vector_gpu(IN: $W, index$. OUT: $vk, vl, pair$.)
2:    $[k, l] \leftarrow [0, \mathbf{size}(\text{index}) - 1]$
3:    $pair \leftarrow 0$
4:    for $i \leftarrow |W| - 1 \dots 0$
5:       $[k, l] \leftarrow$ search_iteration($[k, l], W[i], index$)
6:       if $k > l$ then
7:          $pair \leftarrow i + 1$
8:          break
9:       end if
10:      $[vk(i), vl(i)] \leftarrow [k, l]$
11:   end for
12:   for $i \leftarrow pair - 1 \dots 0$
13:      $[vk(i), vl(i)] \leftarrow [k, l]$
14:   end for
15: end function
```

The **helper_cpu** functions (algorithm 5) perform the inexact search, they receive as parameters the value *pair* and the interval vectors *vk* and *vl* calculated by the **vector_gpu** functions. Before starting the analysis with one error, we check the values of *vk* and *vl* at the starting position to include the exact matching case in the *results*. In each iteration the helper function reads the values of the vectors, instead of spending time calculating them.

The analysis starts in the position of the last valid interval of the opposite direction ($pair_i$), but only if it is smaller than the middle position of the read. The $pair_i$ value indicates the longest valid substring of the read starting from the beginning. As we are searching allowing just one error and we know that at

$pair_i$ position we must allow a dissimilarity, we will only find mappings if the substring between $pair_i$ and the end of the read is present in the reference. This is similar to the strategy presented in [10], but in this case we do not need to implicitly calculate a bounding vector.

Algorithm 5 Backward Helper CPU

```
1: **backward_helper_cpu**(IN: $W, vk, vl, pair_i, index$. OUT: $results$
2:   if $vk(0) \le vl(0)$ then
3:     $results$ += $[vk(0), vl(0)]$ with [ ]
4:   end if
5:   $pos \leftarrow min(|W|/2, pair_i) + 1$
6:   $range \leftarrow vl(pos+1) - vk(pos+1)$
7:   for $i \leftarrow pos \ldots 0$
8:     $range_p \leftarrow range$
9:     $range \leftarrow vl(pos) - vk(pos)$
10:    if $range_p = range$ continue
11:    $results$ += exact_backward($W, [vk(i+1), vl(i+1)], i-1, D(i), index$)
12:    for $b \in \{A, C, G, T\}$
13:      $[k,l] \leftarrow$ search_iteration($[vk(i), vl(i)], b, index$)
14:      if $k \le l$ then
15:        if $b \ne W[pos]$ then
16:          $results$ += exact_backward($W, [k,l], i, I(i,b), index$)
17:          $results$ += exact_backward($W, [k,l], i-1, M(i,b), index$)
18:        end if
19:      end if
20:    end for
21:  end for
22: end function
```

During the mapping with one error, the algorithm only explores the possible deletions, insertions and mismatches (D,I,M) if the number of suffixes in the current $[vl(pos), vk(pos)]$ interval is different to the values of the last interval ($range_p = range$). When the $range$ value becomes smaller after analysing a symbol, it indicates that we have lost some strings that could be mapped allowing errors in the that position. Notice that after a position with an invalid $[k,l]$ interval $range_p = range$ will always be true, as we filled the rest of the vector with the same value in the **vector_gpu** function.

4 Out-of-core execution

The second study case is based on our inexact mapping algorithm implementation on CPU. As it is based on replaceable components, we recently added a new backward search runtime using *csalib* interfaces. The *csalib* library provides several backward search implementations which are either based on the FM-Index or SA. The main difference of *csalib* with our current implementation is that the data structures are not loaded into main memory [15], but accessed from disk by demand using *mmap*. Such properties may be useful in memory demanding tasks, like mapping against big genomes. We benchmark our inexact mapping algorithm, comparing our in memory implementation of the FM-Index with the *csalib* out-of-core implementation for DNA.

Our CPU algorithm is compatible with any backward search implementation providing the following interfaces:

$$[k', l'] \leftarrow \textbf{search\_iteration}([k, l], \mathit{symbol}, \mathit{index}) \tag{1}$$

$$\mathit{position} \leftarrow \textbf{get\_sa}(\mathit{suffix}, \mathit{index}) \tag{2}$$

$$\mathit{suffix} \leftarrow \textbf{get\_isa}(\mathit{position}, \mathit{index}) \tag{3}$$

$$\mathit{size} \leftarrow \textbf{size\_sa}(\mathit{index}) \tag{4}$$

This simplicity eases portability. Function 1 is a single backward search iteration. In a single iteration we have an initial $[k, l]$ interval in the suffix array and after analysing a *symbol* we end with an equal or narrower $[k', l']$ interval. This function must also work in forward direction by only changing the *index*.

Function 2 returns the original position in the reference of a given suffix array position, while function 3 is its inverse. Finally, function 4 returns the size of the suffix array.

5 Results and discussion

All the executions have been performed in a PC with an Intel(R) Core(TM) i7-3930K CPU running at 3.20GHz speed, 64GB of DDR3 1066 MHz RAM and a Raid 0 of two OCZ-VERTEX4 SSD drives. The machine has two Nvidia GeForce GTX 680 GPGPUs with 4GB of RAM.

5.1 GPU results

In the hybrid CPU-GPU tests the reads are mapped against the Drosophila Melanogaster genome. All the reads are extracted from the genome, being exact matches of lengths 50-200 bps.

The test in table 1 demonstrates the effectiveness of the hybrid parallelisation model. We employed a small set of 4000 reads. First, we measure the execution time of the original algorithm on CPU, this algorithm only allows one error. Second, we divide the logic of the original algorithm in the two subroutines described (**vector** and **helper**). We observe only a 7% overhead when separating the logic. Also, we see that the **vector** function performs the 97% of the computation. Finally we execute the **vector** subroutine on the GPU, obtaining a 10.5 speed-up (including memory transfers). As we did not introduced the code of the **helper** function on the GPU, we can still parallelise it on the CPU. This parallelisation will be more effective, since the helper function contains all the conditional execution code which is not suited for the GPU SIMD model.

Table 1. Effectiveness of the hybrid model

Function call	Microseconds
CPU original	61868
CPU Vector	60450
CPU Helper	5289
GPU Vector	5786

The test in table 2 consisted in executing all the function calls of the hybrid algorithm sequentially. We also employed a small set of 4000 reads, in this case to measure the impact in the total execution time of each step. Notice that writing the results to disk takes almost half of the time. For this reason, while the GPU is working we concurrently write results to disk.

Table 2. Sequential execution of the hybrid CPU-GPU algorithm

Function call	Microseconds	Percentage
disk_read	1045	4.5%
cpu_to_gpu	417	1.8%
vector_gpu	3392	14.76%
gpu_to_cpu	1977	8.6%
helper_cpu	5289	23%
disk_write	10849	47%
TOTAL	22969	100%

In figure 1, we compare the execution times of our GPU implementations for exact and one error mapping against our CPU implementation and SOAP3-dp. We employed a dataset of 2 million reads and measured the tools under the most similar conditions possible. As we already demonstrated in [24], we outperform SOAP3-dp when performing exact mapping on GPU. The hybrid CPU-GPU approach is slightly slower than SOAP3-dp when allowing 1 error, but as can be seen in figure 2 we are finding many more mapping locations due to the support for insertions and deletions with one error. Notice that in these tests most of the time is spent in disk writes.

We conclude that the hybrid model presented in this paper is a valid and different approach for inexact mapping on GPU. The main advantage of this model is that it allows to increase the sensitivity of sequence mapping with one error on GPU without decreasing the speed-up provided by the architecture. In addition, when a read is not found the algorithm returns its pair-ends, which is another advantage of this approach. The pair-ends can be used as seeds for a secondary local alignment algorithm (like Smith-Waterman).

5.2 Out-of-core results

In the out-of core tests the reads are mapped against the Ensembl 68 human genome built upon GRCh37. The program dwgsim 0.1.8 from SAMtools was used to simulate 2 million Illumina reads of 250 nucleotides length. A high quality dataset containing a maximum of 2 N's per read and 0.1% of mutations with 10% indels was generated.

Figure 3 compares the execution time of our reprocessing algorithm using both our implementation of the FM-Index and the *csalib* out-of-core runtime. Also, Bowtie 1 and SOAP2 are included in the benchmarks, in order to compare our approach with existing algorithms implementing the same functionality. Figure 4 shows the total mapping locations found by each algorithm. It demonstrates that we are performing a similar computation, with a slightly better sensitivity.

The results of this test reveal that the out-of-core execution reduces memory consumption to 200MB, with a reasonable performance hit (60%). Modern aligners, which perform seeding alignment before local alignment, can take advantage of this new approach in order to reduce its memory requirements in early stages. Finally, this increases the effectiveness of overall backward search methods, being capable of dealing with bigger indexes in machines with cost-effective SSD disk configurations.

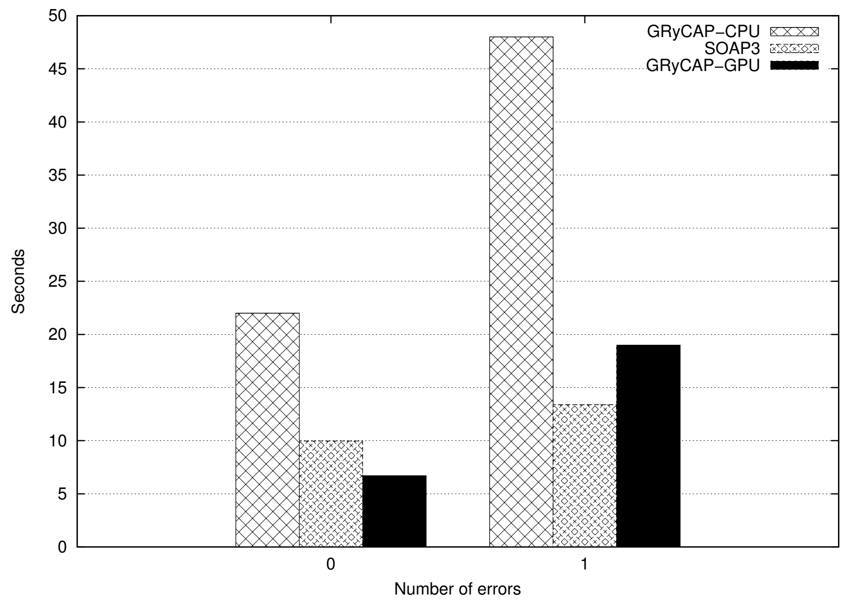

Fig. 1. Hybrid CPU-GPU, execution times

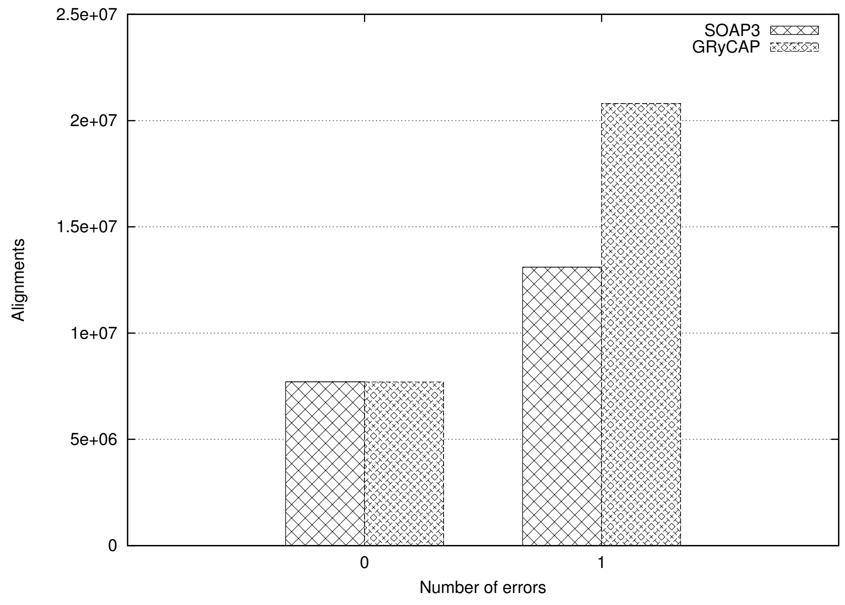

Fig. 2. Hybrid CPU-GPU, mapping locations found

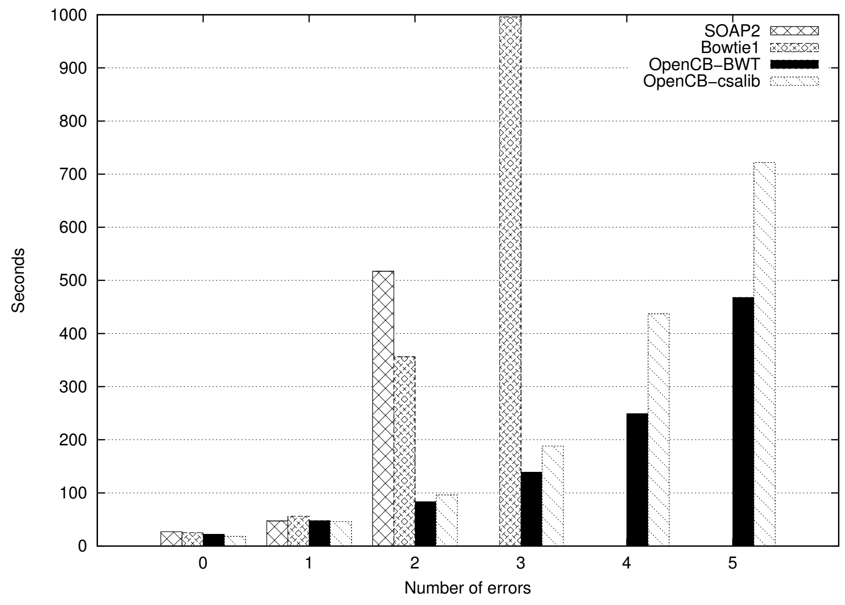

Fig. 3. Out-of-core, execution times

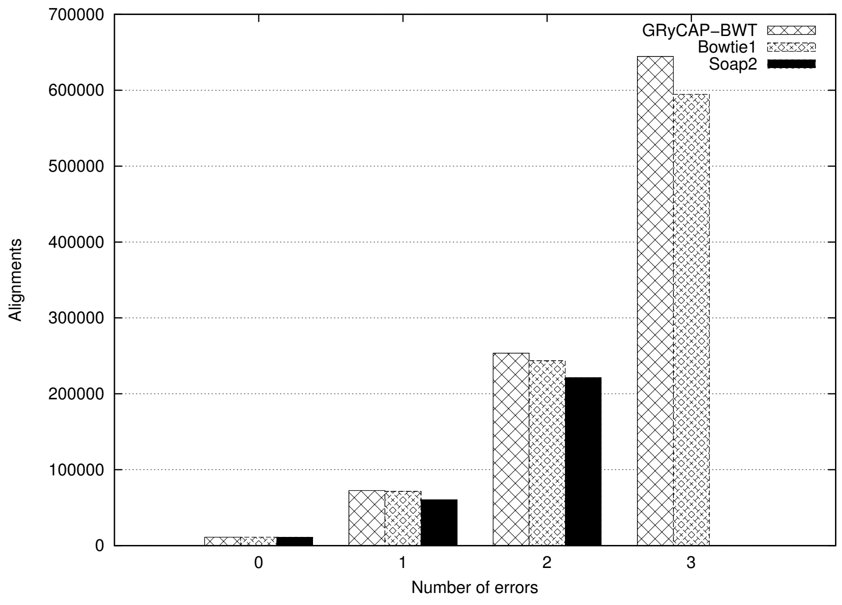

Fig. 4. Out-of-core, mapping locations found

References

1. A library for compressed full-text indexes, `https://code.google.com/p/csalib/`
2. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. Rep. 124, SRC (Digital, DEC, Palo Alto) (May 1994)
3. Church, G.: Genomes for all. Scientific American 294, 47–54 (2006)
4. Editorial: Metagenomics versus moore's law. Nature Methods 6(9), 623 (Sep 2009), `http://dx.doi.org/10.1038/nmeth0909-623`
5. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: FOCS. pp. 390–398 (2000)
6. Grossi, Vitter: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SICOMP: SIAM Journal on Computing 35 (2005)
7. Hall, N.: Advanced sequencing technologies and their wider impact in microbiology. J Exp Biol 210(9), 1518–1525 (2007), `http://jeb.biologists.org/cgi/content/abstract/210/9/1518`
8. Klus, P., Lam, S., Lyberg, D., Cheung, M., Pullan, G., McFarlane, I., Yeo, G., Lam, B.: Barracuda - a fast short read sequence aligner using graphics processing units. BMC Research Notes 5(1), 27 (2012), `http://www.biomedcentral.com/1756-0500/5/27`
9. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology 10(R25) (Mar 2009)
10. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. Bioinformatics 25(14), 1754–1760 (2009), `http://dx.doi.org/10.1093/bioinformatics/btp324`

11. Li, H., Homer, N.: A survey of sequence alignment algorithms for next-generation sequencing. Briefings in Bioinformatics 11(5), 473–483 (2010), http://dx.doi.org/10.1093/bib/bbq015
12. Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K., Wang, J.: Soap2: an improved ultrafast tool for short read alignment. Bioinformatics 25(15), 1966–1967 (2009), http://bioinformatics.oxfordjournals.org/content/25/15/1966.abstract
13. Liu, C., et al.: Soap3: Gpu-based compressed indexing and ultra-fast parallel alignment of short reads. In: Third Workshop on Massive Data Algorithmics, Paris, France (June 2011)
14. Liu, Y., Schmidt, B.: Long read alignment based on maximal exact match seeds. Bioinformatics 28(18), i318–i324 (2012), http://bioinformatics.oxfordjournals.org/content/28/18/i318.abstract
15. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of backward searching &#8212; efficient secondary memory and distributed implementation of compressed suffix arrays. In: Proceedings of the 15th International Conference on Algorithms and Computation. pp. 681–692. ISAAC'04, Springer-Verlag, Berlin, Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-30551-4_59
16. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327. SODA '90, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1990), http://dl.acm.org/citation.cfm?id=320176.320218
17. Manzini, G.: An analysis of the burrows-wheeler transform. In: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 669–677. ACM-SIAM, N.Y. (Jan 17–19 1999)
18. Martínez, H., Tárraga, J., Medina, I., Barrachina, S., Castillo, M., Dopazo, J., Quintana-Ortí, E.S.: Concurrent and accurate rna sequencing on multicore platforms. CoRR abs/1304.0681 (2013)
19. Martínez, H., Tárraga, J., Medina, I., Barrachina, S., Castillo, M., Dopazo, J., Quintana-Ortí, E.S.: A dynamic pipeline for rna sequencing on multicore processors. In: EuroMPI. pp. 235–240 (2013)
20. Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide. Addison Wesley, Upper Saddle River, NJ (2011)
21. NVIDIA: Nvidia cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (Jul 2013), version 5.0
22. Okanohara, D., Sadakane, K.: A linear-time burrows-wheeler transform using induced sorting. In: Karlgren, J., Tarhio, J., Hyyr, H. (eds.) String Processing and Information Retrieval, Lecture Notes in Computer Science, vol. 5721, pp. 90–101. Springer Berlin Heidelberg (2009)
23. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. 39(2) (Jul 2007), http://doi.acm.org/10.1145/1242471.1242472
24. Salavert Torres, J., Espert, B., Others: Using gpus for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform. IEEE/ACM Transactions on Computational Biology and Bioinformatics 9(4), 1245–1256 (2012)