2015 IEEE International Parallel and Distributed Processing Symposium Workshops

Perfect Hashing Structures for Parallel Similarity Searches

Tuan Tu Tran
Johannes Gutenberg-Universität Mainz
Mainz, Germany
trant@uni-mainz.de

Mathieu Giraud
Jean-Stéphane Varré
CRIStAL (UMR CNRS 9189, Université de Lille) and Inria Lille
Lille, France
mathieu.giraud@univ-lille1.fr *jean-stephane.varre@univ-lille1.fr*

***Abstract*—Seed-based heuristics have proved to be efficient for studying similarity between genetic databases with billions of base pairs. This paper focuses on algorithms and data structures for the filtering phase in seed-based heuristics, with an emphasis on efficient parallel GPU/manycores implementation. We propose a 2-stage index structure which is based on neighborhood indexing and perfect hashing techniques. This structure performs a filtering phase over the neighborhood regions around the seeds in constant time and avoid as much as possible random memory accesses and branch divergences. Moreover, it fits particularly well on parallel SIMD processors, because it requires intensive but homogeneous computational operations. Using this data structure, we developed a fast and sensitive OpenCL prototype read mapper.**

***Keywords*-seed-based heuristics; perfect hash function; parallelism; GPU; OpenCL; read mapper**

I. Introduction

Finding similarities between sequences is a way to provide insight in biological functions and to understand the evolution [1], [2]. Similarity studies are also used for re-sequencing [3], [4] or metagenomics [5] where the short genetics fraction collected from the new breeds or from the environment are mapped to the known genomes to evaluate their relations or interactions.

Formally, the *edit distance* between two sequences is the minimal number of *edit operations* to tranform one sequence into the other. The common edit operations are the change of a character (subtitution), the addition of a character (insertion) or the removal of a character (deletion). An edit distance is the *Hamming distance* if only substitutions are allowed, and the *Levenshtein distance* if substitutions, insertions and deletions are allowed. More elaborated *alignment distances* are defined by assigning different *scores* for each type of point mutation.

From a computational point of view, computation of the similarity can be done with *alignment algorithms*. Given two genetic sequences and a score system, those algorithms find the most optimal *global* or *local alignment* (a description of the similarity) together with a *score* (an evalutation of the similarity).

The usual dynamic programming algorithms – Needleman-Wunsch [6] (global alignment) and Smith-Waterman [7] (local alignment) – are not suitable for long sequences due to their quadratic time complexity of $\mathcal{O}(mn)$ with two sequences of length m and n. Current bioinformatics projects need to compute alignments between sequences of hundred millions to billions of base pairs. For example, the re-sequencing process usually maps tens to hundreds millions short *reads* (sequences of tens to hundreds base pairs) to a complete *genome* (the human genome contains 25 chromosomes with a total of 2.7 billions base pairs) or even to databanks of all known sequences. To cope with these data, large computing facilities involving parallelism can be used, but also improved algorithms.

Seed-based heuristics: The seed-based heuristics algorithms, such as FASTA [8] and BLAST [9], were proposed in the late 1980s. Seed-based heuristics rely on the assumption that two similar sequences share some identical parts. They consider short words (4 to 20 characters) named *seeds* to "anchor" the alignments. Once a common seed has been identified in both sequences, further *extension* is realized to get the full local alignments. We call the pair of common seed occurrences in two sequences a *candidate* (Figure 1). In

```
<- extension --- seed --- extension --->
                 AACTagg-ccgatggaga...
                 |||| || | |||| |||
                 AACTcggacagatg-aga...
<- extension --- seed --- extension --->
```

Figure 1. Seed-based heuristics. The common 4-character seed AACT anchors the alignment between the two sequences.

this context, a *good candidate* means a candidate that leads to the local alignment with a score greater than or equal to a choosen threshold that may reflect an e-value [10].

The seed-based heuristics approaches significally reduce the *search space* of the dynamic programming. With full dynamic programming, two sequences of length m and n need $O(mn)$ time to be compared. After a seed-based heuristics, this complexity becomes $\mathcal{O}(r \cdot m'n')$, where r is the number of candidates, and m' and n' the extension lengths, that can be much less than m and n. The candidates contain both true positive (correctly extending into an alignment, *i.e.* final score greater than the threshold) and false positive that will

/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.105

be further discarded. The shorter the seed, the more false postive but also the more sensitivity. To improve selectivity a *filtering phase* is added to the whole seed-based heuristics aligment process. The filtering phase is usually applied to the flanking regions around the seeds: the *neighborhoods*. This phase is called: *neighborhood filtering* (Figure 2). The assumption is that sequences in the vicinity of the seeds in the two sequences compared should be very closed.

Read mappers: Read mapping consists in aligning a large number of *reads* produced by high-throughput sequencing technologies (relatively short sequences of length ranging from tens to hundreds base pairs) against a long reference sequence (often a genome). Many tools have been proposed since the last few years (e.g. BWA [11], Bowtie [12], the CUSHAW suite [13]–[15]). Although they implement different strategies, they have in common the use of indexation in order to speed up the search. Surveys and evaluations such as [16]–[19] use the type of indexing algorithm to classify read mappers. Following [16], one can distinguish *hashing-based* and *Burrow-Wheeler Transform* read mappers. Both strategies achieve good results.

Among the hashing-based read-mappers, some of them use full seed-based heuristics as presented in Section II-B, with an extension of the neighborhood regions to select the good candidate before doing the full extension. This type of read-mapper is also called "seed-and-extend" or "BLAST-like".

Our work: Seed-based heuristics alignment process is thus based on three phases: 1) seed indexing, 2) neighborhood filtering and 3) full extension. In this article, we focus on the second phase, using a *redundant neighborhood indexing* tailored to *parallel computing*. Given a candidate, that is a pair of seeds between the two sequences, the neighbors around the seeds are compared allowing errors. Those that do not meet a given criterion are rejected. For those that have been accepted, the full extension phase is realized. This means that neighborhood filtering is just a filter to select locations of interest. Full extension is made from both ends of the seeds. The length of the neighbors is a parameter of the method as the seed length was.

```
seed
AACTaggcc
|||| || |
AACTcggac
seed
    neighb.
```

Figure 2. Neighborhood filtering leading to the alignement of Figure 1. Firstly, after identifying a candidate with a seed of length 4, the neighborhoods of length 5 are compared. Here we use the Hamming distance with athreshold of 50%. Secondly, if the neighborhoods are sufficiently similar, the extension is done from the candidate (see Figure 1).

We combine here neighborhood indexing approaches [20] and perfect hashing [21]. The key idea of our parallelization is to use wisely these techniques to avoid as much as possible random accesses to the global memory and branch divergences, making the data structure particularly suited for SIMD parallelism such as GPUs. These ideas efficiently accelerate the *neighborhood filtering* phase, allowing time gains using a GPU, both on raw performance and potentially on applications such as read mapping.

The article is organized as follows. Section II presents some background on GPU parallel programming and on seed-based heuristics. Section III presents perfect hashing and Section IV shows how to efficiently parallelize perfect hashing on GPUs. Finally, Section V evaluates the performance of these methods. The neighborhood filtering is up to $10\times$ faster when using perfect hashing functions. It can also be used in a application such as a read mapper, as demonstrated by our prototype read mapper MAROSE.

II. Background

A. GPU processors

Since the early 2000s, integrating multiple cores within a single chip is the main trend to maintain the continuous improvement of computing power of the processors [22], [23]. Graphical Processing Units (GPUs) with hundreds to thousands cores are currently main representatives of many-core processors. Thanks to the popularity and the massively computing power of the GPUs, they have been used for many tasks other than graphics processing. The so-called General Purpose Computation on GPU (GPGPU) takes benefits from the improvements in both GPU architecture and the GPU programming languages such as Brook, C for CUDA, OpenCL, HMPP, OpenACC, etc. With the rapid development of GPGPU, numerous applications in all field of bioinformatics have been mapped onto GPU [24], [25].

Ideally, the most intensive computing tasks of an application should be mapped into a kernel. The same kernel is then executed many times by different *work-items*, working on different data sets. The *host program* calls the kernel and ensures that data is transferred to and back from the GPU.

To achieve good performance on the GPU, algorithms designers have to be aware of specific features of these platforms. Several problems could lead to a significant reduction in performance, including random memory accesses and branch divergence.

Memory accesses. GPUs have a large off-chip memory, on which the global memory is located, with high bandwidth but relatively high latency. This type of memory is advantageous when accessing large and contiguous regions. As long as the memory access pattern is optimized, it can effectively handle hundreds or thousands simultaneous data read or write transactions [26]. On the other hand, frequent lightweight memory accesses to random regions of the global memory lead to bottlenecks in data transfer due

to the high latency, resulting in a serious decrease of the performance of the parallelized application.

SIMD execution and branch divergence. On GPUs, an instruction is dispatched to run on a group of computational elements called lane. At a point of time, one lane execute only one instruction. The computational elements in a lane are thus handled in a SIMD (Single Instruction Multiple Data) way. In some cases, conditional branches are sequentially executed, causing a stall in parallel work-items and thus a waste of computational resources [24].

Depending on the application and the size of the data, it is not always possible to have a good parallelization of any algorithm. We will explain in Section IV how our data structure with perfect hashing successfully adresses these challenges, reducing as much as possible random global memory accesses and branch divergence.

B. Offset and Neighborhood Indexing

Seed-based heuristics builds an index for a reference sequence (or a database) that is then queried with sequences of interest. All seeds s of a query are used to anchor an alignment with the reference sequence and database (Figure 3, a). We now present the usual scheme for indexing, the *offset indexing*, where the index stores the offset of the reference sequences, and the redundant *neighborhood indexing*, where the index also stores a copy of the neighborhood of the reference sequence (Figure 3, b).

In both cases, given a seed s of length ℓ_s, the process ends with an extension phase on the neighborhoods of s (Figure 3, c) and possibly further extensions. We call ℓ_n the length of the neighborhood used in the first extension phase.

1) Offset indexing: In the common implementation of seed-based heuristics, the index stores, for a given seed s, all the positions of the reference sequence where s occurs. The index contains thus the positions of all possible seeds. This way of indexing is called *offset indexing* by [20]. From a query sequence and a seed it contains, it remains to query the index to retrieve all the positions where the seed occurs in the reference sequence. This provides a list of candidates. This is the *seed matching phase* (see Figure 3, a). With such an index, to go further in the *neighborhood filtering phase*, each candidate returned needs several memory accesses to the reference sequence to gather the neighborhoods of the seeds between the reference and the query (see Figure3, b, left). To minimize these access times, software like PLAST build once this list of neighborhood, at runtime [27]. These memory accesses are unfortunately random, unpredictable and non contiguous. By the way, they are not efficiently cached and require high latencies, for both CPUs and GPUs.

2) Neighborhood indexing: The *neighborhood indexing* consists in directly storing the neighborhoods (of length ℓ_n) of the seed in the reference sequence in the index [20].

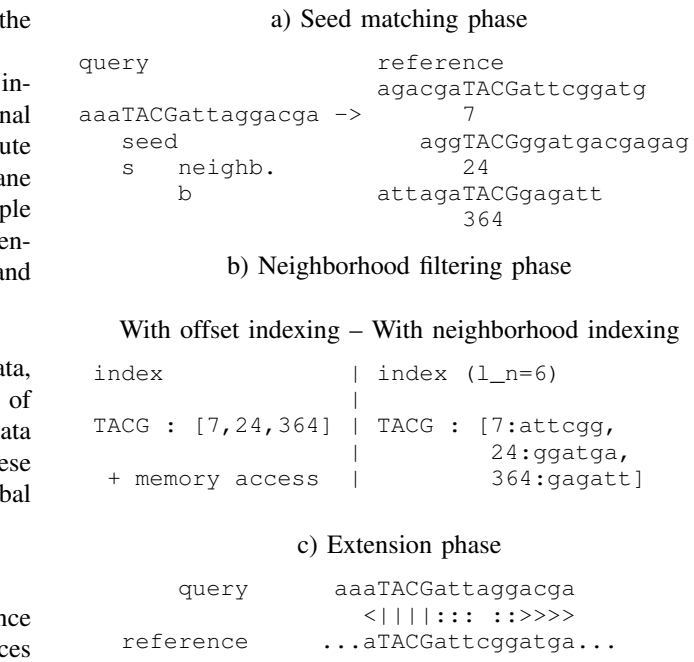

Figure 3. Seed-based heuristics with offset indexing or neighborhood indexing. a) From a seed taken from the query, one obtains b) the positions in the reference (offset indexing), possibly with the neighborhoods (neighborhood indexing). c) For neighborhoods that meets a criterion, the extension phase is done. Here, neighborhood indexing is done only on the right side of the seed. The left side can be handled in a similar way.

The neighborhood filtering phase remains the same process with the advantage to access the neighborhoods more efficiently, from a single contiguous memory region. This approach can thus avoid some random memory accesses in the process of gathering the neighborhoods of each seed. Obviously, implementing such an index leads to a memory overhead compared to the offset indexing approach. Nevertheless, this overhead is not so large: For 32-bit offsets and neighborhood lengths $\ell_n = 4$, 8, 16, the total size of a complete neighborhood index is about $1.25\times$, $1.5\times$ or $2\times$ larger than the offset index, respectively.

Storage of the neighborhoods. Given a seed s, the neighborhood indexing thus requires to store the list of all positions and neighborhoods of the seed s. We call $\mathsf{nb\_block}(s)$ this list, and $\mathcal{N}$ the number of its elements. The most simple solution is to store $\mathsf{nb\_block}(s)$ as a plain list:

- the list $\mathsf{nb\_block}(s)$ can be *sorted according to the positions* [27], [28]. Querying the data structure with a pattern b is then done by an iteration in $\mathcal{O}(\mathcal{N})$ steps. A simple exact pattern matching can be used, or an approximate pattern matching algorithm, computing Hamming, Levenshtein, or any more elaborated distance.
- the list $\mathsf{nb\_block}(s)$ can be *sorted according to the neighborhoods*, enabling fast access for exact matching with binary search in $\mathcal{O}(log(\mathcal{N}))$ time. In this case,

approximate matching can also by done by iterating exact matching over a set of degenerated patterns $\Pi(b)$ (that is a set of patterns built from the orginal one by introducing some erros). The advantage is that each degenerated pattern can be processed independently from the others, leading to parallel processing. However, implementing binary search requires a lot of possible conditional branches, decreasing the parallel efficiency on SIMD architectures such as GPUs.

III. Perfect hashing, without collisions

To improve the neighborhood indexing, there is the need for a data structure able to query neighborhoods in parallel, in constant time (i.e. independently from the size of the nb_block), and in which all patterns queries uses the same instructions, without branch divergence. We now discuss various hashing techniques and present the *perfect hashing functions* [29], [30].

A. Hashing and parallel hashing

Hash functions maps a *key* (a neighborhood in our case) to an *address*. Several hash table designs allow queries in almost constant time. In our case, the goal is to represent nb_block(s) as a hash table, and to be able, in $\mathcal{O}(1)$ time, to access the list of positions of a given neighborhood. To achieve such a complexity, the *hash function* has to be designed to ensure that the addresses of the keys are uniformaly distributed among the set of possible addresses.

Parallel hashing. The parallelization of the hashing technique is usually based on the fact that the indices of the keys can be calculated independently. The study [31] proposes GPU parallelizations of several hashing algorithms, such as "Open adressing", "Chaining" or "Cuckoo hashing". However, these works mainly focus on the speed of the hash table building phase and on subsequent dynamic updates.

In our neighborhood filtering approach, the index is built only once without requiring dynamic updates: We focus here on the *key retrieval phase*, which should be efficiently parallelized. Moreover, the existing parallel hashing methods share a common problem: Although the queries can be executed in parallel, there is always the possibility of nondeterministic accesses to the hash table, which lead to both random memory accesses and branch divergences.

Collisions. A disadvantage of usual hash functions is the eventuality of *collisions* when multiple neighborhoods b are hashed to the same address. Usually, libraries using hashing techniques explicitely handle collisions. But the problem is that the access time to the value does not remain constant, and can be different between elements.

On parallel implementations, the need for collision handle in the key retrieval phase still remains. On SIMD architecture, such as GPUs, this non-determinism can lead to branch divergence and a decrease in performance.

We used another technique, the *perfect hashing functions* that have *no collision at all*. In this case, the test of an exact match of a neighborhood in nb_block(s) can be done exactly in constant $\mathcal{O}(1)$ time, with a fixed number of memory accesses. Moreover, it fits particularly well on parallel SIMD processors, because it requires *intensive but homogeneous* computational operations.

B. Graph-based perfect hash functions

Perfect hash functions, without collisions, can be implemented using the BDZ algorithm [29], which is largely based on [30]. The idea is to use in a graph $G = (\mathcal{V}, E)$, where

- The set of vertices $\mathcal{V}$ are the choosen interval of addresses.
- The set of edges E are randomly built from the set of keys (the neighborhoods).

Finding a perfect hash function exactly means finding an injective function mapping an edge $e = (v_0, v_1)$ to a vertex $phf(e)$. A practical way to build such a function that is to make an *assignment*. That is selecting for each edge e, a value $phf(e) = v_j$ with either $j = 0$ or $j = 1$.

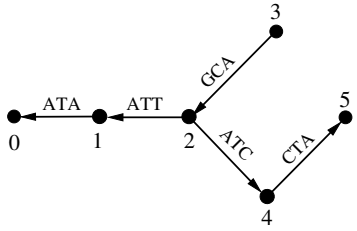

Figure 5. Perfect hashing with graph assignment. On each edge (corresponding to one key, here one neighborhood), the arrow shows the injective assignment to a vertex: each vertex is assigned to at most one edge.

If the graph is *acyclic*, this is fairly simple to find such assignment. One picks an edge e with a vertex v of degree 1 and assigns $phf(e) = v$, then one removes the edge e, and one iterates (Figure 5).

As soon as $|\mathcal{V}| > 2 \times |E|$, a random graph is acyclic with a probability of almost 1 [30]. To build a perfect hash function, it is thus sufficient to randomly draw a graph with $|\mathcal{V}| > 2 \times |E|$ and check its acyclicity. One would like also to remember the phf assignment, without actualling storing anything related to the edges, as there are 4^{ℓ_n} possible values for the edges. A solution is to store one bit of information $g[v_i]$ per vertex, and aggregate the values of the vertices of the same edge to compute $phf(e)$ (Figure 4, top).

A further improvement consists in using *hypergraphs* (generalization of graphs in higher dimensions): As soon as $|\mathcal{V}| > 1.23 \times |E|$, a random 3-hypergraph has an "acyclic" property with a probability of almost 1 [30], [32]. In practice, assignment and computation of phf on 3-hypergraphs are

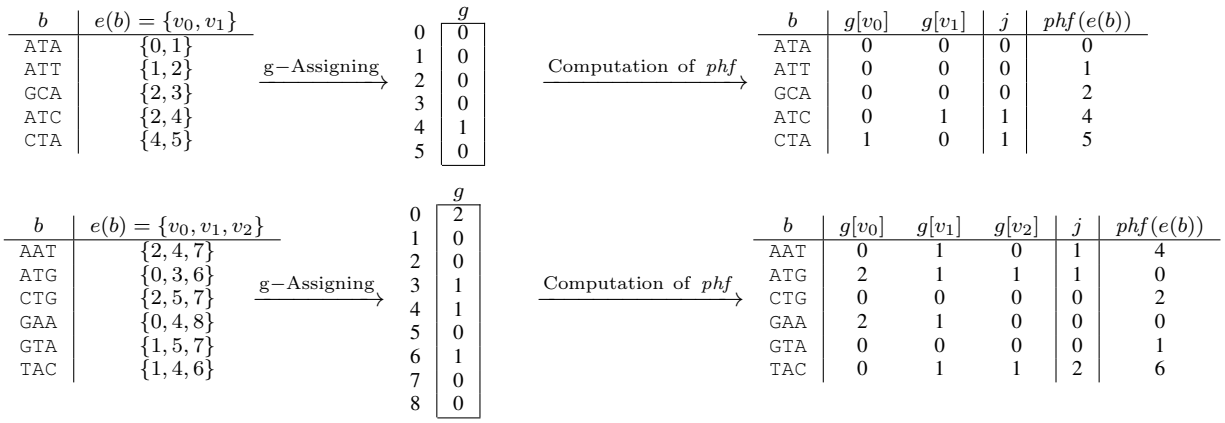

Figure 4. Perfect hash functions with graph assignment. Top: Assignment of the acyclic graph of Figure 5. Each edge $e = (v_0, v_1)$ is assigned to one of its 2 supporting vertices: $phf(e) = v_j$, where $j = (g[v_0] + g[v_1]) \mod 2$. Bottom: An "acyclic" 3-hypergraph with 9 vertices and 6 edges. Each edge $e = (v_0, v_1, v_2)$ connects 3 vertices and is assigned to one of this 3 vertices : $phf(e) = v_j$, where $j = (g[v_0] + g[v_1] + g[v_2]) \mod 3$.

very similar to the ones on regular graphs (Figure 4, bottom), requiring to store two bits of information $g[v_i]$ per vertex[1].

IV. Parallelizing perfect hash functions on GPU

Perfect hashing enables to retrieve, in constant time, the address associated to a given neighborhood, enabling further extension phases. We now describe how to build the data structure nb_block for indexing neighborhoods b with perfect hash functions, allowing to minimize branch divergences and random memory accesses. Our implementation needs only 3 random memory accesses per neighborhood and the program behaves deterministically, without branch divergence (except when there is a candidate).

A. Indexation with 3-hypergraphs generation

For each seed s, we want to represent nb_block(s) as a perfect hash function, that is generating (and storing) an acyclic 3-hypergraph.

- Generating random 3-hypergraphs can be done with a hashing with 3 different hash functions. Each neighborhood b will yield an edge $e(b) = (h_x(b), h_y(b), h_z(b))$. If the hash functions are universal [33], the graph will be random. Storing the graph is thus equivalent to be able to compute the hash functions h_x, h_y and h_z.
- Checking the acyclicity can be done with the algorithm of [32]. In the unlikely cases where the graph is not acyclic, another random graph can be generated by choosing three other hash functions.

As there are 4^{ℓ_s} different seeds, there are also 4^{ℓ_s} different acyclic graphs to generate and remember. The easiest way to do this is to use a *family of hash functions*, indexed by some integer. The BDZ algorithm proposes to use the hash functions family developed by B. Jenkins [34]. Although these hash functions have not been proved theoretically to be universal, they perform well in practice [21].

[1]Note that the actual BDZ algorithm has a further optimization to obtain a *minimal* perfect hashing function, but that was not useful here.

```
mix(x,y,z)
    x -= y; x -= z; x ^= (z >> 13);
    y -= z; y -= x; y ^= (x << 8);
    z -= x; z -= y; z ^= (y >> 13);
    x -= y; x -= z; x ^= (z >> 12);
    y -= z; y -= x; y ^= (x << 16);
    z -= x; z -= y; z ^= (y >> 5);
    x -= y; x -= z; x ^= (z >> 3);
    y -= z; y -= x; y ^= (x << 10);
    z -= x; z -= y; z ^= (y >> 15);
```

Figure 6. The `mix` procedure of the Jenkins hash functions mixes bits from three integers x, y and z.

The Jenkins functions uses a mixing procedure (Figure 6) that shuffles bits of three integers. After some initialization involving an integer J, the mix is done on successive 8-bit chunks of the neighborhood b. The mix function does not depend on J. For our parallelization, using the Jenkins hash functions has several advantages:

- The three integers x, y, z can be used as the result of three hash functions $h_x(b), h_y(b), h_z(b)$;
- These functions are represented only by the integer J, requiring no global memory access;
- These functions requires some arithmetic computations (54 `unsigned int` operations in our implementation), but these operations do not depend on J and have always the same number of instructions, with no branches, and are thus efficiently parallelized.

B. GPU parallelization

In our parallelization, each work-group finds all occurrences of a *pattern* sb, that is a combination of a seed s and a right neighbor b. In the work-group, each work-item computes the occurrence positions for a degenerated neighborhood that is computed by adding up to 2 mismatch errors to the initial neighborhood by Algorithm 1.

Algorithm 1 Seed and Filter (Kernel)

let $s_i b_i$ be the pattern corresponding to the current work-item number
let $\mathcal{L}$ be an empty list
computes $\Pi(b_i)$ the list of degenerated neighborhoods of b_i
for each neighborhood $b_i^k \in \Pi(b_i)$ **do**
  add to $\mathcal{L}$ the occurrence position of $s_i b_i^k$ computed by querying the index nb_block(s_i)
end for
from $\mathcal{L}$ computes a list of unique sorted occurrence positions L_{CPos} using parallel radix-sort

As the length of the neighborhood is fixed, the number of degenerated neighborhoods is always the same and all the work-items compute synchronously. Accesses to the global memory are limited to one query to the index. The only branch divergence occurs when a candidate is returned, triggering an evaluation of this candidate.

The index itself, which contains the neighborhoods and their positions, is made of the *assigning table* g and a two-level *position table* (see Figure 7). The assigning table is used to compute $phf(e)$, and this value is the offset in the first level of the position table. The second level of the position table then gives the list of positions for each neighborhood.

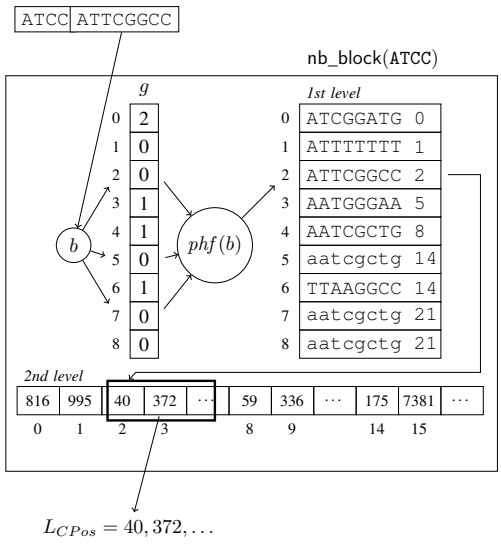

Figure 7. The neighborhood index structure for the seed ATCC. A neighborhood can be retrived in constant time thanks to the table g. From a neighbour b one computes $phf(b)$ with table g. As several positions can share the same neighborhood, a two-level table is then used. One accesses to the range of positions of b through the first level between values stored in $phf(b)$ and $phf(b)+1$ (2 and 5). Finally one obtains the list of positions of b stored in the second level. The list of candidates is sorted with a parallel radix sort algorithm before being returned. The implementation is detailed in [25].

V. Performance Evaluation

This section evaluates the performance of perfect hashing in the filtering phase in seed-based heuristics. both for the retrieval and approximate comparisons of the neighborhoods, and for a prototype read mapper application.

Two platforms were used for the tests. The first one consists of an NVIDIA GTX 480 (30×16 cores, 1.4 GHz) and an Intel Xeon E5520 (8 cores, 2.27 GHz). The memory size of the GPU and of the host are 1.5 GB and 8 GB, respectively. The OpenCL library is provided in the NVIDIA GPU Computing SDK 1.1 beta.

The second platform comprises an NVIDIA GeForce Titan GPU (192×14 CUDA cores, 837 MHz) and an Intel Core i7-2700K (8 cores, 3.5 GHz). The memory size of the GPU and of the host are 6 GB memory and 16 GB, respectively. The OpenCL library is provided in the NVIDIA CUDA Toolkit 6.5.

A. Comparing neighborhood matching techniques on GPU

Several techniques for neighborhood indexing were tested on the GTX 480 platform. Figure 8 illustrates the performance of perfect hashing (PH) and binary search (BS), compared to a simple list traversal (L). Seed length ℓ_s was set to 4 and the neighborhood length ℓ_n ranges from 4 to 16. The perfect hashing solution process the nb_blocks up to $10\times$ faster than the binary search. The length of the seed does not have any influence on the time complexity of any of these methods.

- *Exact matching.* As neighborhoods are internally stored as 32-bits integer, the performance of exact neighborhood matching is the same for all three neighborhood lengths. There are no significant differences between PH and BS solutions, both of them begin faster than plain lists.
- *Approximate pattern matching*. For the binary search and the perfect hashing methods, the length of neighborhood affects the performance, the number of degenerated patterns $|\Pi_e(b)| = \mathcal{O}(\ell_n^e)$ growing with ℓ_n.

The perfect hashing solution is thus adapted for SIMD architectures, and it is relatively simple to implement.

B. MAROSE, a prototype OpenCL read mapper using neighborhood indexing

MAROSE is a prototype OpenCL read-mapper which implements the filtering phase by using the neighborhood indexing approach. It ts designed to process a large number of reads and huge reference sequences. It takes as input a set of reads $\mathcal{R} = \{r_1, ..., r_n\}$, a set of sequences $\mathcal{T} = \{t_1, ..., t_m\}$ which will be indexed in an index I, and the number of errors authorized in the full alignment extension a_e.

MAROSE uses the seed-based technique, with (right) neighborhood indexing, presented in Section II-B2. The extension phase is done with a full semi-global alignment

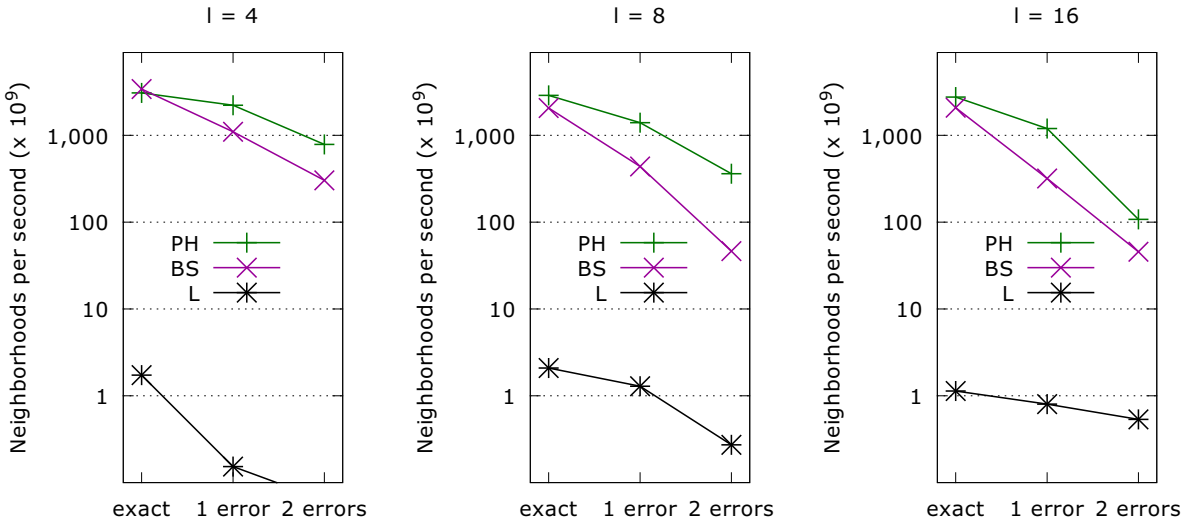

Figure 8. Performance of the binary search (BS) and the perfect hashing (PH) solutions, compared to a simple list traversal (L) for seed length $\ell_s = 4$ and neighborhood length $\ell_n = 4, 8$ and 16 (from left to right). Performance is measured in billions of *neighborhoods per seconds*, that is the total number of neighbors processed in the nb_block per second. The task here is to check whether the neighborhood is in the index, and does not include the actual retrieval and sorting of the position list. The algorithms are tested on an *exact* neighborhood match, as well as on an approximate match with 1 or 2 *substitution errors*.

to check the Levenshtein distance with at most a_e errors. The main program iterates over the genomes t_j (and their respective indexes), and launches two OpenCL kernels, detailed below. MAROSE outputs the list of matching positions for each read.

Algorithm 2 MAROSE

Input: a set of reads $\mathcal{R} = \{r_1, ..., r_n\}$, a set of genomes $\mathcal{T} = \{t_1, ..., t_m\}$ and their index $I(\ell_s, \ell_n, e)$, a number of errors a_e

Output: for each read, matching positions in the genomes

for each t_j **do**
  launch kernel "**Seed and filter**" on all reads
  launch kernel "**Extend**" on all candidates
end for

If the sequences t_j are large, they are further divided into smaller subsequences in order that the corresponding indexes (with the subsequences) fit into the GPU global memory. The matching is then serialized with indexes and sequences loaded in turns, from the hard disk to the main memory of the host and then to the global memory of the GPU.

1) The "seed and filter" kernel: The sequence t is indexed with the seed length ℓ_s and the neighborhood length ℓ_n. The resulting index, containing all nb_block(s), is created once and stored on the hard disk.

Each input read is divided into a set of consecutive patterns $\{s_i b_i\}$, s_i being the seed part, of length ℓ_s and b_i being the (right) neighborhood part of length ℓ_n. For a read of length ℓ_r, there are thus $\ell_r - \ell_s - \ell_n + 1$ such patterns.

For each pattern $s_i b_i$, b_i is matched with at most e errors with the corresponding list of neighborhoods in nb_block(s_i). We build the set of degenerated patterns $\Pi(b_i) = \{b_i^k \in \Sigma^{\ell_n}, d_{\text{Hamming}}(b_i, b_i^k) \leq e\}$ and then we match exactly each b_i^k against nb_block(s_i). Here, one could use either indexing using binary search or perfect hasing.

Each pattern $s_i b_i^k$ can thus be matched exactly against nb_block(s) in parallel. Each pattern $s_i b_i$ is processed by one work-item independently and the whole set of patterns for one read is processed by one work-group (Figure 9).

Each matching of b_i^k leads to one putative alignment position on the genome, called the *absolute candidate position*. As the patterns overlap along the read, one position can be dectected multiple times, leading to duplicates. As in [35], we filter out these duplicates by sorting these positions. We used a parallel version of a radix sort in order to accelerate this process. The result of this "Seed and Filter" kernel is a list of unique positions.

2) The "extend" kernel: The "extend" kernel takes as input a read of length ℓ_r and a candidate at position p in the genome and do the full extention. This phase is implemented with a *banded semi-global alignment* algorithm between the read and the genome segment ranging from $p - a_e$ to $p + \ell_r + a_e$ [36]. This allows us to find the best alignment with a Levenshtein distance at most a_e.

The list of candidate positions from all reads returned by the work-groups are gathered to create the alignment candidate list. In this list, each element corresponds to an alignment between a read r and a subsequence of the genome. This phase is also parallelized as each alignment candidate can obviously be processed independently.

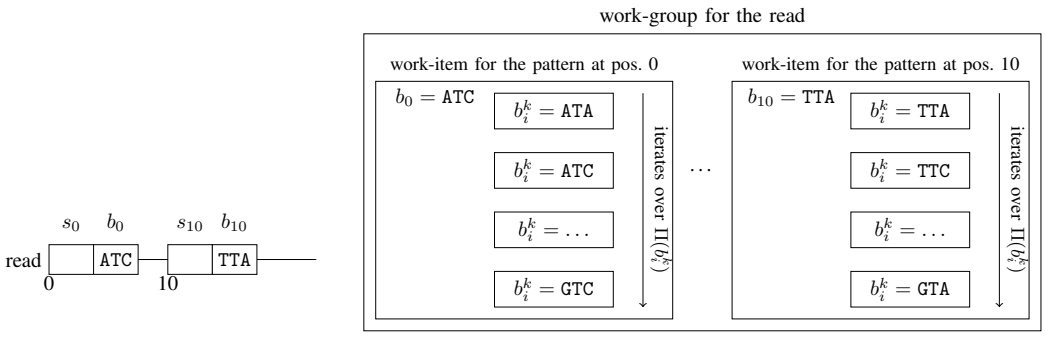

Figure 9. Parallel implementation used in MAROSE. Each read is processed in a work-group. Each work-group computes in parallel the matching over all possible patterns ($\ell_r - (\ell_s + \ell_n)$) in its work-items. Each work-item computes the matching for all possible degenerated neighborhoods. The pseudocode of the kernel is given Section V-B3.

3) Further optimizations: The design of MAROSE allowed to implement two other features to have a better control on sensitivity and specificity:

- *Non-consecutive patterns*. The successive patterns created from the input read do not need to be consecutive (that is at positions $p, p+1$,*etc.* They can be taken with a shift of δ positions to the right: For a seed of length ℓ_s and a neighborhood of length ℓ_n, the read r of length ℓ_r is processed into a set of $[(\ell_r - \ell_s - \ell_n)/\delta] + 1$ patterns. The matching phase can thus be faster, but with a trade-off in sensitivity. Moreover, using this feature, one can tune the number of created patterns to fit with the hardware capability of the device. On our platform, we processed reads of larger lengths, up to 400 bp without a dramatic decrease in sensibility.
- *Spaced seeds*. "Spaced seeds" improve the sensitivity of seed-based heuristics [37]–[39]: They allow to account for errors in seeds. A spaced seed is represented by a sequence of #'s and −'s, where the #'s indicate positions of exact matching. For example, the spaced seed `######-##` authorizes one mismatch at position 7. Indexing spaced seeds takes the same memory as the regular (contiguous) seeds, as long as the number of #'s is the same. Sensitivity can be improved with almost the same speed than regular indexes, using exactly the same OpenCL kernel code.

C. Performance evaluation of MAROSE

Sensitivity: We used the benchmark proposed in [16] to evaluate MAROSE. The reference genome is the human genome of length 2.7 Gbp, denoted $\mathcal{H}_{ref}$. The set of reads, denoted $\mathcal{H}_3$, is made of 10 millions reads of length 40, each read contains exactly three random substitutions.

For this dataset, the best sensitivity (with acceptable runtime) was achieved with the following parameters: seed length $\ell_s = 8$, neighbor length $\ell_n = 7$, consecutive patterns ($\delta = 1$), space seed `######-##`. The runtime of the experiment of mapping $\mathcal{H}_3$ against $\mathcal{H}_{ref}$ is 2h41m. The sensitivity of MAROSE against other read-mappers is presented on Table I.

Software	Non-map. reads	Mapped reads	Uniquely hit	Multiple hits	
				Number	Mean
Novoalign [40]	47	9999953	8699117	1300836	15.12
Bowtie [12]	49	9999951	8496649	1503302	1161.98
BWA [11]	49	9999951	8496649	1503302	1161.98
SSAHA2 [41]	213	9999787	8286416	1713371	6.81
MAROSE	5715	9994285	8427129	1567156	517.927
PerM [42]	186752	9813248	8496655	1316593	147.25
BFAST [43]	199451	9800549	8476476	1324073	6.17
GASSST [35]	326598	9673402	8193650	1479752	1139.25

Table I
SENSITIVITY COMPARISION BETWEEN MAROSE AND OTHER READ MAPPERS ON READS OF SIZE 40. FOR THE OTHER READ MAPPERS, THE NUMBERS ARE TAKEN FROM [16]. THE MEAN NUMBER OF MULTIPLE HITS IS REPORTED TO EVALUATE THE CAPACITY TO FIND ALL THE OCCURRENCES OF EACH READ IN THE TARGET SEQUENCES. NOTE THAT MAROSE HAS EXACTLY THE SAME SENSITIVITY WHEN USED WITH BINARY SEARCH OR PERFECT HASHING.

While MAROSE is not as sensitive as some read mappers which do not make use of seed-based heuristics, it is more sensitive than other seed-based heuristics read mappers.

Runtime: According to [16], BWA is one of the most sensitive and fast short read mappers. We compared the speed of MAROSE and BWA (version 0.7.12). BWA ran with 8 threads. We performed approximate mapping with 3 substitutions of 460,544 reads of size 40 onto the human chromosome 10. To be indexed, the chromosome 10 is divided into 3 segments of size 50 MB. Results are reported in Table II for the Titan platform. For both read mappers, the time for indexing is not included. Since MAROSE does not output the SAM format, we only compared the runtime of MAROSE with the coordinate calculating phase of BWA (the time for the SAM creating phase of BWA is not included).

For BWA, the runtime is significantly different between ungapped alignment and gapped alignment. The number of allowed gaps (among the number of errors) is configured by using the `-o` parameter. When the gap is not allowed (`-o 0`), the runtime of MAROSE is nearly equivalent to

	MAROSE	BWA		
		-o 0	-o 1	-o 2
Runtime (s)	21.313	24.682	74.288	97.210

Table II
RUNTIME OF MAROSE AND BWA ON THE TITAN PLATFORM.

that of BWA. When the gaps are allowed, the runtime of BWA increases substantially while for MAROSE it does not change thanks to the full extend phase. Further research should be conducted to explore the trade-off between speed and sensibility.

VI. CONCLUSION

Neighborhood indexing is a seed-based heuristic techniques where one stores, for each seed, a list of the associated neighborhoods. With a little memory overhead, this technique is very efficient because it allows to minimize the number of random memory accesses.

We presented a perfect hashing solution to the neighborhood indexing, allowing to further reduce random memory accesses as well as branch divergence, making this technique fully adapted to massively parallel processors such as GPUs. This technique brings a up to $10\times$ improvement compared to standard neighborhood indexing. We also implemented a prototype readmapper with seed-based heuristics using perfect hashing.

The parallel perfect hasing fits particularly well on parallel SIMD processors, because it requires intensive but homogeneous computational operations. It could be also suitable for any high performance computing architectures which support vector processing, such as recent multi-core CPUs and Intel's Xeon Phi coprocessors [44] which includes 512-bit wide SIMD instrinsic functions. We thus believe that *parallel perfect hashing is an efficient solution to store a list of genomic sequences*, and could be used as well in other applications.

Acknowledgements

This work was supported by the ANR MAPPI (ANR-10-COSI-0004) grant. We thank Bertil Schmidt for allowing access to the computing platform of his group.

REFERENCES

[1] A. Varki and A. K. Tasha, "Comparing the human and chimpanzee genomes: Searching for needle in a haystack," *Genome Research*, vol. 15, pp. 1746–1758, 2005.

[2] H. Quach, D. Wilson, G. Laval, E. Patin, J. Manry, J. Guibert, L. B. Barreiro, E. Nerrienet, E. Verschoor, A. Gessain, M. Przeworski, and L. Quintana-Murci, "Different selective pressures shape the evolution of Toll-like receptors in human and African great ape populations," *Human Molecular Genetics*, vol. 22, no. 23, pp. 4829–4840, 2013.

[3] R. Kawahara-Miki, K. Tsuda, Y. Shiwa, Y. Arai-Kichise, T. Matsumoto, Y. Kanesaki, S.-i. Oda, S. Ebihara, S. Yajima, H. Yoshikawa, and T. Kono, "Whole-genome resequencing shows numerous genes with nonsynonymous SNPs in the Japanese native cattle Kuchinoshima-Ushi," *BMC Genomics*, vol. 12, no. 1, pp. 1–8, 2011.

[4] K.-T. Lee, W.-H. Chung, S.-Y. Lee, J.-W. Choi, J. Kim, D. Lim, S. Lee, G.-W. Jang, B. Kim, Y. Choy, X. Liao, P. Stothard, S. Moore, S.-H. Lee, S. Ahn, N. Kim, and T.-H. Kim, "Whole-genome resequencing of Hanwoo (Korean cattle) and insight into regions of homozygosity," *BMC Genomics*, vol. 14, no. 1, pp. 1–14, 2013.

[5] T. Thomas, J. Gilbert, and F. Meyer, "Metagenomics – a guide from sampling to data analysis," *Microbial Informatics and Experimentation*, vol. 2, no. 1, pp. 1–12, 2012.

[6] S. B. Needleman and C. D. Wunsch, "A general method applicable to the seach for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 433–453, 3 1970.

[7] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 3 1981.

[8] D. Lipman and W. Pearson, "Improved tools for biological sequence comparison," *Proc. Natl Acad. Sci.*, vol. 85, no. 8, pp. 2444–2448, 1988.

[9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–413, 1990.

[10] S. Karlin and S. Altschul, "Methods for assessing the statistical significance of molecular sequence feature by using general scoring schemes," in *Proc. Natl. Acad. Sci.*, vol. 87, 1990, pp. 2264–2268.

[11] H. Li and R. Durbin, "Fast and accurate short read alignment with BurrowsWheeler transform," *Bioinformatics*, vol. 25 (14), pp. 1954–1960, 7 2009.

[12] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10 (3), 2009.

[13] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform," *Bioinformatics*, 2012.

[14] Y. Liu and B. Schmidt, "Long read alignment based on maximal exact match seeds," *Bioinformatics*, vol. 28, no. 18, pp. i318–i324, 2012.

[15] Y. Liu, B. Popp, and B. Schmidt, "CUSHAW3: Sensitive and Accurate Base-Space and Color-Space Short-Read Alignment with Hybrid Seeding," *PLos ONE*, vol. 9, no. 1, p. e86869, 2014.

[16] S. Schbath, V. Martin, M. Zytnicki, J. Fayolle, V. Loux, and J.-F. Gibrat, "Mapping reads on a genomic sequence: an algorithmic overview and a practical comparative analysis," *Journal of Computational Biology*, vol. 19, no. 6, pp. 796–813, June 2012.

[17] H. Li and N. Hormer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in Bioinformatics*, vol. 11 (5), pp. 473–483, 9 2010.

[18] D. S. Horner, G. Pavesi, T. Castrignanò, P. D'Onorio De Meo, S. Liuni, M. Sammeth, E. Picardi, and G. Pesole, "Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing," *Briefings in Bioinformatics*, vol. 11, no. 2, pp. 181–197, 2010.

[19] S. Bao, R. Jiang, W. Kwan, B. Wang, X. Ma, and Y.-Q. Song, "Evaluation of next-generation sequencing software in mapping and assembly," *Journal of Human Genetics*, vol. 56, pp. 406–414, 6 2011.

[20] P. Peterlongo, L. Noé, D. Lavenier, V. H. Nguyen, G. Kucherov, and M. Giraud, "Optimal neighborhood indexing for protein similarity search," *BMC Bioinformatics*, vol. 9, no. 534, 2008.

[21] F. C. Botelho, "Near-Optimal Space Perfect Hashing Algorithms," Ph.D. dissertation, Department of Computer Science, Federal University of Minas Gerais, September 2008.

[22] P. P. Gelsinger, "Microprocessors for the new millennium: Challenges, opportunities, and new frontiers," in *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*. IEEE, 2001, pp. 22–25.

[23] J. Shalf, K. Asanovic, D. Patterson, K. Keutzer, T. Mattson, and K. Yelick, "The Manycore Revolution: Will HPC Lead or Follow?" *SciDAC Review*, vol. 14, 2009.

[24] J.-S. Varré, B. Schmidt, S. Janot, and M. Giraud, "Manycore high-performance computing in bioinformatics," in *Advances in Genomic Sequence Analysis and Pattern Discovery*, L. Elnitski, H. Piontkivska, and L. R. Welch, Eds. World Scientific, 2011, p. chapter 8.

[25] T. T. Tran, "Bioinformatics Sequence Comparisons on Manycore Processors," Ph.D. dissertation, University of Lille 1, December 2012.

[26] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 105–118, Jan. 2011.

[27] V.-H. Nguyen and D. Lavenier, "PLAST: parallel local alignment search tool for database comparison," *BMC Bioinformatics*, vol. 10, p. 329, 2009.

[28] T. T. Tran, M. Giraud, and J.-S. Varré, "Bit-Parallel Multiple Pattern Matching," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7204, pp. 292–301.

[29] F. C. Botelho, D. d. C. Reis, D. Belazzougui, and N. Ziviani, "CMPH – C Minimal Perfect Hashing Library," online website, last updated: 09 June 2012, http://cmph.sourceforge.net/.

[30] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech, "A family of perfect hashing methods," *The Computer Journal*, vol. 39, no. 6, pp. 547–554, 1996.

[31] D. A. F. Alcantara, "Efficient Hash Tables on the GPU," Ph.D. dissertation, University of California Davis, 2011, supervised by N. Amenta.

[32] G. Havas, B. S. Majewski, N. C. Wormald, and Z. J. Czech, "Graphs, Hypergraphs and Hashing," in *19th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG 1993)*, 1994, pp. 153–165.

[33] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (Extended Abstract)," in *Proceedings of the ninth annual ACM symposium on Theory of computing*, ser. STOC '77. New York, NY, USA: ACM, 1977, pp. 106–112.

[34] B. Jenkins, "Algorithm alley: Hash functions," *Dr. Dobb's Journal of Software Tools*, vol. 22, no. 9, 1997, online version, http://www.drdobbs.com/database/algorithm-alley/184410284.

[35] G. Rizk and D. Lavenier, "GASSST: Global Alignement Short Sequence Search Tool," *Bioinformatics*, vol. 26, no. 20 2010, pp. 2534–2540, 2010.

[36] E. Ukkonen, "Algorithms for appriximate string matching," *Information and Control*, vol. 64, 1985.

[37] D. G. Brown, *Bioinformatics Algorithms: Techniques and Applications*, 2008, ch. A survey of seeding for sequence alignment, pp. 126–152.

[38] L. Noé and G. Kucherov, "YASS: enhancing the sensitivity of DNA similarity search," *Nucleic Acids Research*, vol. 33, no. S2, pp. W540–W543, 2005.

[39] B. Ma, J. Tromp, and M. Li, "PatternHunter: faster and more sensitive homology search," *Bioinformatics*, vol. 18, no. 3, pp. 440–445, 2002.

[40] N. Technologies. Novoalign. [Online]. Available: http://novocraft.com

[41] Z. Ning, A. J. Cox, and J. C. Mullikin, "SSAHA: a fast search method for large DNA databases," *Genome Research*, vol. 11 (10), pp. 1725–1729, 2001.

[42] Y. Chen, T. Souaiaia, and T. Chen, "PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds," *Bioinformatics*, vol. 25, no. 19, pp. 2514–2521, 2009.

[43] N. Homer, B. Merriman, and S. F. Nelson, "BFAST: An Alignment Tool for Large Scale Genome Resequencing," *PLoS ONE*, vol. 4, no. 11, p. e7767, 11 2009.

[44] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, 1st ed. MA 02451, USA: Elsevier, 2013.