Bit-Parallel Multiple Pattern Matching

Tuan Tu Tran, Mathieu Giraud, Jean-Stéphane Varré

Bit-Parallel Multiple Pattern Matching

Tuan Tu Tran, Mathieu Giraud, and Jean-Stéphane Varré

LIFL, UMR 8022 CNRS, Université Lille 1, France
INRIA Lille, Villeneuve d'Ascq, France

Abstract. Text matching with errors is a regular task in computational biology. We present an extension of the bit-parallel Wu-Manber algorithm [16] to combine several searches for a pattern into a collection of fixed-length words. We further present an OpenCL parallelization of a redundant index on massively parallel multicore processors, within a framework of searching for similarities with seed-based heuristics. We successfully implemented and ran our algorithms on GPU and multicore CPU. Some speedups obtained are more than 60×.[1]

Keywords: bit parallelism, pattern matching, sequence comparison, neighborhood indexing, GPU, OpenCL

1 Introduction

With the advances in "Next Generation Sequencing" technologies (NGS), the data to analyze grows even more rapidly than before and requires very efficient algorithms. In *read mapping*, the goal is to map short "reads" produced by NGS on a reference genome; another example is *metagenomics analyses*, where one of the goals can be to identify at which species belongs each read. Such applications require to search a short sequence against a set of sequences. All these problems are thus related to a generic basic problem where we query a *pattern* into one or several *texts*, allowing some *errors*:

Problem 1 (Approximate Pattern Matching in Large Sequences). Given a pattern p, a parameter e and a set of sequences $\mathcal{S}$ over an alphabet Σ, find all occurrences of p in $\mathcal{S}$ within a Levenshtein distance of e.

The Levenshtein distance between two words is the minimal number of insertions, deletions and substitutions needed to transform one word into the other one. In the case of NGS read mapping, the lengths of the patterns are typically from several dozens to several hundreds. The set of sequences can be either a large number of short sequences or a unique large sequence. There are multiple strategies to address this problem (see Section 2.2). The well-known *seed-based heuristics* assume that there is a conserved short word between the pattern and its occurrence in the text. There are numerous tools implementing this strategy,

[1] This research was partially supported by the French ANR project MAPPI. Cards for experiments were provided through a "Action Incitative LIFL" grant.

including the popular BLAST [1], but also tools using improved seeds such as PatternHunter [9] or YASS [12]. Such methods involve several stages, including a neighborhood extension that can be formalized as follows:

Problem 2 (Approximate Pattern Matching in Fixed-Length Word). Given a pattern p and a parameter e, find all words in a set of fixed-length words such that the Levenshtein distance with p is at most e.

The length of such fixed-length words is often small (4, 8, 16 in this paper), and corresponds only to the neighborhood of matched seeds. Positive results at this step lead to alignments on the full pattern length to solve Problem 1.

GPU/manycore computing. Graphics processing units (GPUs) were used in bioinformatics since 2005 [4]. Since then, lots of different studies were proposed on Smith-Waterman sequence comparisons or other bioinformatics applications (review in [15]). Computing with GPUs was firstly done by tweaking graphics primitives. The CUDA libraries, released by NVIDIA in 2007[2], and the OpenCL standard[3] now enable easy programming on GPU/manycore architectures. The same OpenCL code can be compiled and optimized for different platforms [5]. As of today, at least five different implementations of this standard are available: NVIDIA, AMD, Apple, IBM, and Intel. In the following years, the OpenCL standard could become a practical standard for parallel programming.

Contents. This paper brings two contributions. In Section 3, we propose a simple extension to the bit-parallel algorithms of [16] to a collection of fixed-length words to address Problem 2. In Section 4, extending ideas from [13], we show how, for solving Problem 1 with a seed-based heuristic, a redundant neighborhood indexing is more efficient than an offset indexing and scales well on GPU/manycore architectures. Performance tests on CPU and GPU are given in Section 5.

It should be noted that Hirashima *et al.* also studied bit-parallel algorithms on CUDA [8], but their algorithms were optimized for strings over a binary alphabet for stringology studies and do not apply here.

2 Background

We provide here some background in bit-parallelism (Section 2.1), as well as in seed-based heuristics and neighborhood indexing (Section 2.2).

2.1 Bit parallel matching

Text searching using bit-parallelism emerged in the early 90's. The approach consists in taking advantage of the parallelism of bit operations, encoding the states of a matching automaton into a machine word seen as a bit array. Ideally,

[2] `http://www.nvidia.com/cuda`
[3] `http://www.khronos.org/opencl`

these algorithms divide the complexity by w, where w is the length of a machine word. The book [11] is an excellent reference on the subject.

The Shift-or algorithm for exact pattern matching [2] is one of the first algorithms using this paradigm. In 1992, Wu and Manber [16] proposed an approximate matching algorithm. The Wu-Manber algorithm (called BPR, for Bit-Parallelism Row-wise, in [11]) allows substitution, insertion and deletion errors, and was implemented in the `agrep` software.

Exact matching. The pattern p of length m is encoded over a bit array R of length m. Characters of the text t are processed one by one, and we denote by $R^{[j]}$ the value of R once the first j letters of the text have been read. More precisely, the i^{th} bit of $R^{[j]}$ equals 1 if and only if the first i characters of the pattern $(p_1...p_i)$ match exactly the last i characters of the text $(t_{j-i-1}...t_j)$. The first bit of $R^{[j]}$ is thus just the result of the matching of $(p_1 = t_j)$, and, when $i \geq 2$, the i^{th} bit $R^{[j]}(i)$ of $R^{[j]}$ is obtained by:

$$R^{[j]}(i) = \begin{cases} 1 \text{ if } R^{[j-1]}(i-1) = 1 \text{ and } (p_i = t_j) \text{ (match)} \\ 0 \text{ otherwise} \end{cases}$$

With bitwise operators and (&) and shift ($\ll$), this results in Algorithm 1.1.

Algorithm 1.1. Exact Bit-Parallel Matching

$$\begin{cases} R^{[0]} \leftarrow 0^m \\ R^{[j]} \leftarrow \left((R^{[j-1]} \ll 1) \mid 0^{m-1}1 \right) \ \& \ B[t_j] \end{cases}$$

The *pattern bitmask* B is a table with $|\Sigma|$ bit arrays constructed from the pattern, such that $B[t_j](i) = 1$ if and only if $(p_i = t_j)$. This algorithm works as long as $m \leq w$, where w is the length of the machine word, and needs $O(z)$ operations to compute all $R^{[j]}$ values, where z is the length of the text.

Approximate matching. To generalize the matching up to e errors, we now consider $e+1$ different bit arrays $R_0, R_1, ...R_e$, each one of length m. The i^{th} bit of $R_k^{[j]}$ equals 1 if and only if the first i characters of the pattern match a subword of the text finishing at t_j with at most k errors, leading to Algorithm 1.2. Due to insertion and deletion errors, the length of a match in the text is now in the interval $[m-e, m+e]$.

This algorithm works as long as $m+e \leq w$, but now takes $O(ez)$ time. BPR has been reported as the best unfiltered algorithm in DNA sequences, for low error levels and short patterns (p. 182 of [11]). We thus focused on this algorithm instead of theoretically better ones such as BNDM, implemented in the `nrgrep` software [10]. Moreover, BPR is more regular than other solutions, enables a better performance on processors with large memory words such as those with SIMD instructions.

Algorithm 1.2. BPR Matching

$$\begin{cases} R_k^{[0]} \leftarrow 0^{m-k}1^k \\ R_0^{[j]} \leftarrow \left((R_0^{[j-1]} \ll 1) \mid 0^{m-1}1\right) \ \& \ B[t_j] \quad \text{(match)} \\ R_k^{[j]} \leftarrow \underbrace{\left((R_k^{[j-1]} \ll 1) \ \& \ B[t_j]\right)}_{\text{(match)}} \mid \underbrace{R_{k-1}^{[j-1]}}_{\text{(insertion)}} \mid \underbrace{(R_{k-1}^{[j-1]} \ll 1)}_{\text{(substitution)}} \mid \underbrace{(R_{k-1}^{[j]} \ll 1)}_{\text{(deletion)}} \mid \underbrace{0^{m-k}1^k}_{\text{(init)}} \end{cases}$$

2.2 Seed-based heuristics and redundant neighborhood index

Preprocessing and indexes. A common way to speed-up pattern matchings is to preprocess the text, for example by building an index in which performing a query of a pattern can be achieved in constant or linear time. A lot of work has been done to build non-redundant indexes in order to save memory, such as suffix trees or suffix arrays [6]. Obviously, there is always a trade-off between memory efficiency and time efficiency. Thus, simpler index techniques, even with some redundancy, could achieve better time performance.

Seed-based indexing. In seed-based indexing, the pattern p is divided in two parts: a *seed* p_s of size W, which has to occur exactly, and a *neighborhood* p_n of length ℓ. In the *filtering phase*, we search for occurrences of the seed in the index and retrieve the list of all its neighborhoods. In the *finishing phase*, the neighborhood of the pattern is compared to the neighborhoods of the occurrences of the seed.

Complete seed-based heuristics can include further finishing stages. Moreover, designing efficient seeds is a wide area of research, including spaced seeds and their extensions (see [3] for a review). We will not address these problems here, but focus on the *storage of neighborhoods*: how can we, for each seed, have access to the list of all its occurrences in the index ? We now follow the discussion of [14] to present different ways to store these neighborhoods.

Offset indexing. In the usual *offset indexing* approach, depicted on Figure 1, an offset is stored for each seed position. For each query position, each hit returned by the filtering phase leads to an iteration of the finishing phase. This iteration accesses some neighborhoods of the positions. These memory accesses are *random*, that are unpredictable and non contiguous. Such accesses are not efficiently cached and require high latencies [7]. This is still true for GPUs: despite high internal memory bandwidths, random access patterns decrease efficiency.

Neighborhood indexing. A way to reduce the computation time is thus to avoid as far as possible such random memory accesses. In [13], a *neighborhood indexing* approach has been proposed. The idea is to directly store in the index the neighborhood of size ℓ for every seed occurrence (Figure 2). Thus all neighborhoods corresponding to a seed are obtained through a single contiguous memory access.

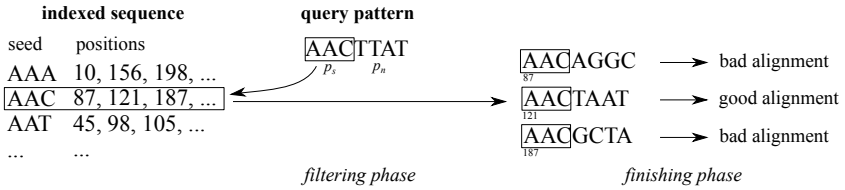

Fig. 1. Offset indexing [14]. During the filtering phase, the seed p_s (here `AAC`) is used to access to the index. Then the finishing phase get neighborhoods of this seed (one memory access per position), and compare them against p_n. If N is the total number of neighborhoods, each offset takes $\log N$ bits, thus index size is $N \times \log N$ bits.

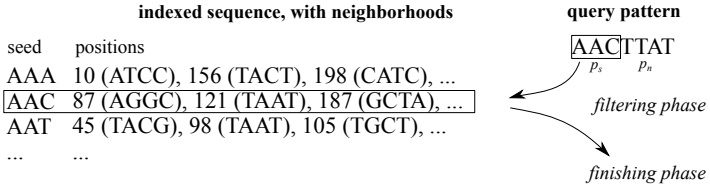

Fig. 2. Neighborhood indexing [14]. All neighborhoods are stored along the offset of each seed occurrence. One unique memory access gives all the data needed by the finishing phase. For nucleotide characters, stored in 2 bits, the overall index size is then equal to $N \times (\log N + \ell)$ bits, where ℓ is the length of the neighborhood.

This index is redundant, as every character of the text will be stored in the neighborhoods of ℓ different seeds: the neighborhood indexing enlarges the size of the index. However, it can improve the computation time by reducing the random memory access. In [13], the authors claimed that the neighborhood indexing speeded up the execution time by a factor ranging between 1.5 and 2 over the offset indexing.

3 Multiple fixed-length bit-parallel matchings

In this section, we propose an extension of BPR (Algorithm 1.2, [16]) which solves Problem 2 and takes care of memory accesses. More formally, we compare a pattern p_n of length m against a collection of words t^1, t^2, ... t^n of length $\ell = m + e$, allowing at most e errors (substitutions, insertions, deletions).

The existing bit-parallel algorithms for multiple pattern matching (review in Section 6.6 of [11]) match a set of patterns within a large text. Our setup is different, as we want to match one pattern with several texts. Of course, one could reverse the multiple pattern matching algorithms and build an automaton on a set of all neighborhoods. This would result in a huge automaton, and the algorithm would not be easily parallelizable.

Algorithm. The idea of our algorithm is to store n fixed-length words into a machine word, so n matchings can be done simultaneously. As in BPR, to have

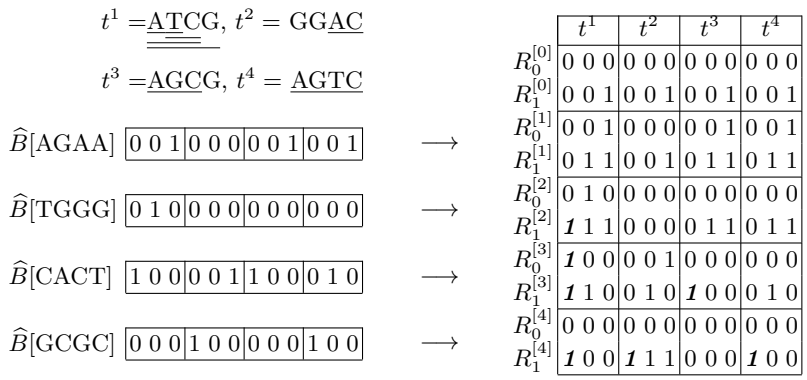

Fig. 3. Execution of the algorithm 1.3 on 12-bit machine words. The pattern $p =$ ATC, of length $m = 3$, is compared against $n = 4$ words with up to $e = 1$ error. The text data $t =$ {ATCG, GGAC, AGCG, AGTC} is stored in a stripped layout as AGAA TGGG CACT GCGC. After 2 iterations, there is one approximate match for t^1 (AT, one insertion). After 3 iterations, there is one exact match for t^1, and one approximate match for t^3 (AGC, one substitution). After 4 iterations, there are three approximate matches, for t^1 (ATCG, one deletion), t^2 (AC, one insertion) and t^4 (AGTC, one deletion).

a matching up to e errors, we consider $e + 1$ different bit arrays $R_0, R_1, ...R_e$, but each one is now of size mn, that is n slices of m bits. If $1 \leq r \leq n$ and $1 \leq i \leq m$, the i^{th} bit of the r^{th} slice of $R_p^{[j]}$ equals 1 if and only if the first i characters of the pattern match the last i characters of the r^{th} text $(t^r_{j-i-1}...t^r_j)$ with at most k errors. We thus obtain Algorithm 1.3.

Algorithm 1.3. Multiple Fixed-Length BPR (mflBPR) Matching

$$\begin{cases} R_k^{[0]} \leftarrow (0^{m-k}1^k)^n \\ R_0^{[j]} \leftarrow \left((R_0^{[j-1]} \ll 1) \mid (0^{m-1}1)^n\right) \ \& \ \widehat{B}[\widehat{t_j}] \quad \text{(match)} \\ R_k^{[j]} \leftarrow \left((R_k^{[j-1]} \ll 1) \ \& \ \widehat{B}[\widehat{t_j}]\right) \mid \underset{\text{(insertion)}}{R_{k-1}^{[j-1]}} \mid \underset{\text{(substitution)}}{(R_{k-1}^{[j-1]} \ll 1)} \mid \underset{\text{(deletion)}}{(R_{k-1}^{[\mathbf{j}]} \ll 1)} \mid \underset{\text{(init)}}{(0^{m-k}1^k)^n} \end{cases}$$

Figure 3 shows a run of this algorithm. Compared to BPR, the initialization is $(0^{m-k}1^k)^n$ instead of $0^{m-k}1^k$. This initialization puts 1's at the k first bits of each slice, thus overriding any data shifted from another slice. Moreover, to allow better memory efficiency:

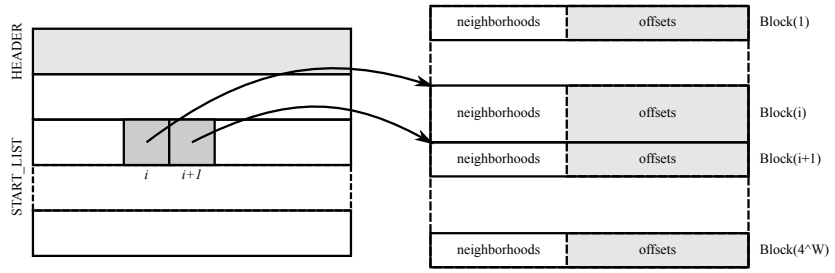

Fig. 4. Structure of the index file. The **START_LIST** contains the positions of 4^W Blocks. The main part is the list of Blocks, each one containing, for a given seed, its neighborhoods as well as its offsets.

- The set of n fixed-length words $t = \{t^1, t^2, ..., t^n\}$ is stored and accessed through a stripped layout, as each access j returns the j^{th} characters of every word: $\widehat{t_j} = t_j^1 t_j^2 ... t_j^n$
- The block mask $\widehat{B}[t_j^1 t_j^2 ... t_j^n] = B[t_j^1]B[t_j^2]...B[t_j^n]$ is thus now larger, having $|\Sigma|^n$ bit arrays (instead of $|\Sigma|$). As in BPR, the computation of this table still depends only of the pattern. This table is somehow redundant, but now allows the match of n characters with one unique memory access.

This algorithm works as long as $mn \leq w$, where w is the length of the machine word, and needs $O(ez/n)$ operations to compute all $R_k^{[j]}$ values, where z is the total length of all texts. Comparing to the BPR algorithm, there are n times less operations. Of course, the limiting factor is again the size of the machine word.

4 Redundant Parallel Neighborhood Indexing

We now describe our redundant neighborhood index for DNA ($|\Sigma| = 4$), and show how it may be used on regular processors as well as on GPUs. The structure of the neighborhood index file is depicted on Figure 4. Searching for a pattern $p = p_s p_n$ is done through the following natural steps:

Pattern pre-processing.
Compute the pattern bitmask $B(p_n)$ (for BPR) or $\widehat{B}(p_n)$ (for mflBPR)
Filtering phase.
Retrieve the position of Block(p_s) in the index
Finishing phase.
Compare p_n against all the neighborhoods of Block(p_s)

This index works either with BPR or mflBPR, or with any other neighbor comparing method. In all cases, there are very few random memory accesses since the Block(p_s) is stored contiguously in the memory.

Usage with OpenCL devices. All the index data are precomputed and transferred only once to the device. Then the application runs looping on each query. The pattern pre-processing as well as the $\mathsf{Block}(p_s)$ position retrieving are done on the host. Then the block bitmask $B(p_n)$ or $\widehat{B}(p_n)$ and the positions of $\mathsf{Block}(p_s)$ are sent to the global memory of the device. The device is devoted to the finishing phase. Depending on the size of $\mathsf{Block}(p_s)$, several comparing cycles may be run. In each comparing cycle, neighborhoods are distributed into different work groups and loaded in the local memory of each work group, and processed by several work items. The positions of the matching neighborhoods are then written back to a result array in the global memory, then transferred back to the host. This index is intrinsically parallel, as the neighborhoods are processed independently.

Index size. An obvious drawback of the neighborhood indexing is the additional memory requiring to store neighborhoods. The ratio between the overall index sizes of the neighborhood indexing and the offset indexing is $r_\ell = 1 + 2\ell/\log N$. For alignment purposes, considering offsets of $\log N = 32$ bits gives ratios that are acceptable, $r_8 = 1.5$ and $r_{16} = 2$. For example, a 100 Mbp sequence with a neighborhood of size $L = 8$ with small seeds ($W \leq 6$) gives an index of size 630 MB, fitting in the main host memory as well as in GPU cards.

5 Performance Results and Perspectives

Testing environment. We benchmarked the algorithms BPR and mpfBPR on GPU and on multicore CPU. The same OpenCL code was used, but with different OpenCL libraries. We thus target these two platforms:

- **GPU:** NVIDIA 480 (30×16 cores, 1.4 GHz, 1.5 GB RAM), with the OpenCL library was NVIDIA GPU Computing SDK 1.1 beta.
- **CPU:** Intel Xeon E5520 (8 cores, 2.27 GHz, 8 MB cache), with the OpenCL library was AMD APP SDK 2.4.

We also tested a C++ "CPU serial" version, which ran on only one core of the CPU. Programs were compiled by `GNU g++` with the `-O3` option. The host computer had 8 GB RAM.

Methodology. Tests were run on the first 100 Mbp of the human chromosome 1. We measured the performance of the algorithms in *millions of words matched per second* (Mw/s). This normalization allowed to benchmark the problem 2 independently. For the problem 1, the number of words was the total number of neighborhoods, removing the bias due to seed selection. We ran searches on 10 successive patterns, but we saw no significant difference between 1, 10, or 100 patterns, as soon as enough computations hided the transfer times.

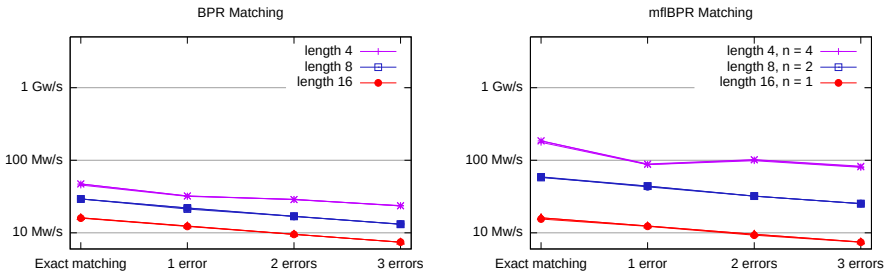

Fig. 5. Performance of BPR (left) or mflBPR (right), both on CPU serial version.

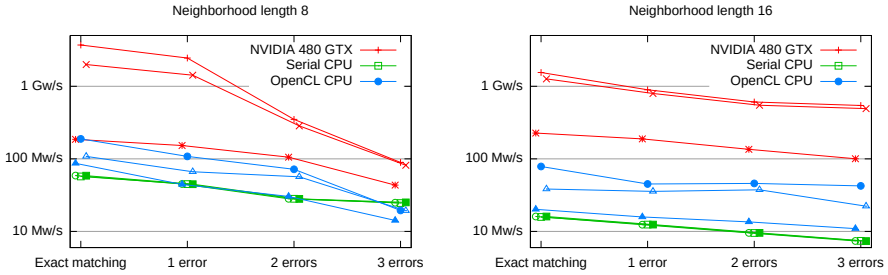

Fig. 6. Performance of the neighborhood indexing with a neighborhood length of 8 (left) and 16 (right). For each platform, there are three different curves, corresponding to different seed lengths (3, 4 and 6), hence to different total numbers of neighborhoods. Times used in the OpenCL versions include transfer times between host and device for the queries and the result, but not for the index.

Performance of mflBPR (CPU). Figure 5 shows performance of mflBPR on a CPU. With 32-bit integers, performance gain compared to BPR ranges from 2.73× to 3.92× for words of length 4 and from 1.89× to 2.06× for words of length 8, close to the 4× and 2× theoretical gains.

Performance of redundant neighborhood index (CPU and GPU). Figure 6 shows performance of the whole index. In the most simple instance (neighborhoods of size 8, no error), the serial CPU implementation peaks at 59 Mw/s, the OpenCL CPU at 189 Mw/s, and the OpenCL GPU at 3693 Mw/s. In this case, using OpenCL brings speed-ups of about 3.2× on CPU and about 62× on GPU. In the same setup, the offset indexing peaks at 4.0 Mw/s on serial CPU, 108 Mw/s on OpenCL CPU and 1706 Mw/s on OpenCL GPU (data not shown).

When the error rises, performance degrades in both implementations. On small neighborhoods, starting from $e = 2$, the performance of both GPU and CPU versions are limited by the number of matches in the output. However, even in the worst case (7.5 Mw/s, CPU serial implementation, 3 errors, seed size 4 and neighborhood length of size 16), using the neighborhood indexing takes less than 0.06 s for parsing a chromosome with 100 Mbp, while non-indexed approaches using bit parallelism takes 0.9 s with `agrep` and 0.7 s with `nrgrep`.

We thus demonstrated the efficiency of using OpenCL and GPUs to speed-up the neighborhood phase extension in seed-based heuristics.

Perspectives. Further work could include a complete evaluation of the redundant index, including a study on the influence of the seed design. It should be noted that our OpenCL code already works both on NVIDIA and AMD SDKs. However, tests on ATI GPU cards (Radeon 5870) give now poor performance (best result peaking at 39.6 Mw/s on a smaller index, not shown). We would thus like to benchmark and optimize our code on other OpenCL platforms.

References

1. Stefan F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *J Mol Biol.*, 215(3):403–413, 1990.
2. Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *SIGIR Forum*, 23(3-4):7, 1989.
3. Daniel G. Brown. *Bioinformatics Algorithms: Techniques and Applications*, chapter A survey of seeding for sequence alignment, pages 126–152. 2008.
4. Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. *Advances in Informatics*, pages 415–425, 2005.
5. Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Parallel architectures and compilation techniques (PACT 10)*, PACT '10, pages 205–216, 2010.
6. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* 1997.
7. John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, 2006.
8. Kazunori Hirashima, Hideo Bannai, Wataru Matsubara, Akira Ishino, and Ayumi Shinohara. Bit-parallel algorithms for computing all the runs in a string. In *Prague Stringology Conference 2009 (PSC 09)*, 2009.
9. Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *BIOINFORMATICS*, 18(3):440–445, 2002.
10. Gonzalo Navarro. NR-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, 31(13):1265–1312, 2001.
11. Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* 2002.
12. Laurent Noé and Gregory Kucherov. YASS: enhancing the sensitivity of DNA similarity search. *Nucleic Acids Research*, 33(S2):W540–W543, 2005.
13. Pierre Peterlongo, Laurent Noé, Dominique Lavenier, Van Hoa Nguyen, Gregory Kucherov, and Mathieu Giraud. Optimal neighborhood indexing for protein similarity search. *BMC Bioinformatics*, 9(534), 2008.
14. Nadia Pisanti, Mathieu Giraud, and Pierre Peterlongo. *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*, chapter Filters and Seeds Approaches for Fast Homology Searches in Large Datasets. 2011.
15. Jean-Stéphane Varré, Bertil Schmidt, Stéphane Janot, and Mathieu Giraud. *Advances in Genomic Sequence Analysis and Pattern Discovery*, chapter Manycore High-Performance Computing in Bioinformatics. World Scientific, 2011.
16. Sun Wu and Udi Manber. Fast Text Searching Allowing Errors. *Communications of the ACM*, 35(10):83–91, 1992.