Parallel Position Weight Matrices Algorithms

Mathieu Giraud, Jean-Stéphane Varré

Parallel Position Weight Matrices Algorithms

Mathieu Giraud, Jean-Stéphane Varré

LIFL, UMR CNRS 8022, Université Lille 1
INRIA Lille-Nord Europe
Lille, France

Abstract

Position Weight Matrices (PWMs) are broadly used in computational biology. The basic problems, Scan and MultipleScan, aim to find all the occurrences of a given PWM or a set of PWMs in long sequences. Some other PWM tasks share a common NP-hard subproblem, ScoreDistribution. The existing algorithms rely on the enumeration on a large set of scores or words, and they are mostly not suitable for parallelization. We propose a new algorithm, BucketScoreDistribution, that is both very efficient and suitable for parallelization. We bound the error induced by this algorithm. We realized a GPU prototype for Scan, MultipleScan and BucketScoreDistribution with the CUDA libraries, and report for the different problems speedups larger than 10× on several Nvidia cards.[1]

Keywords: Bioinformatics, Position Weight Matrices, P-value estimation, pattern matching, score distribution, many-core architectures, GPU

1. Introduction

In molecular biology; the "central dogma" describes the transfer of the information from the DNA to the protein. *Genes* are portions of the DNA, they are first transcribed into *messenger RNAs* which are then translated into *proteins* [2]. Proteins interact between themselves and also with the

Email address: mathieu.giraud@lifl.fr, jean-stephane.varre@lifl.fr (Mathieu Giraud, Jean-Stéphane Varré)

[1] A preliminary version of this paper appeared in [1]. This extended article includes new benchmarks, new results on the Slices Strategy, and new results on MultipleScan (see Section 3.4 and 5.2).

DNA, enabling transcription of other genes. To process the transcription of a gene, proteins called *transcription factors* (TFs) bind to the DNA. In order to bind to the DNA, TFs have a spatial conformation enabling them to recognize a small stretch of the DNA, called the *transcription factor binding site* (TFBS). The TFBSs are located mostly in regions preceding the genes. Discovering such sites is rather difficult because of their very low information content: For a given TF, the DNA fragment on which it can bind may vary both in length and in the sequence of nucleic acids (see Figure 1 for example).

In order to locate putative TFBSs along the DNA, Position Weight Matrices (PWMs) are often used. Basically, such a matrix is a kind of pattern which associates, at each position, a score for each nucleic acid (see Figure 1, and Section 2 for more details). Locating TFBSs requires to use a pattern matching algorithm with a score threshold. The reference databases JASPAR [3] and TRANSFAC [4] respectively contain 123 and 856 matrices of TFBSs. Such matrices are broadly used in computation biology to model conserved sequence patterns. New sequencing technologies enable large-scale mapping of DNA-protein interactions: those “ChIP-Seq” methodologies produce new collections of sequences and matrices [5, 6].

1.1. Solving the computation bottleneck in PWM algorithms

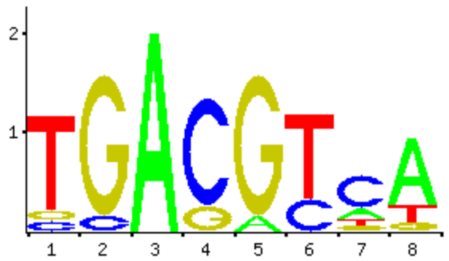
The exponential nature of some PWM problems is a limiting factor for using matrices of medium or large length. Score threshold computations for matrices whose length is greater than 15 usually require several seconds, and hours or days for matrices of length greater than 20. The JASPAR and TRANSFAC databases already contain almost 200 matrices of length 15 to 30, that is 20% of their total number of matrices. Moreover, the new techniques using data from next-generation sequencers should produce longer matrices.

In this paper, we give some parallel solutions to the common PWM problems. We propose a prototype implementation on graphic processing units (GPUs) to test the ability to compute with longer matrices.

1.2. Parallel computation on GPUs

Everyone can have some teraflops of cheap computing power with the recent Graphics Processing Units (GPUs). GPUs are a first step toward new massively many-core architectures. GPUs were used in bioinformatics since 2005 for phylogenetic studies [8], then for multiple sequence alignment based on an optimized Smith-Waterman implementation [9]. The CUDA

```
          aaTTGCGTCAtttc
gccgtcatactgTGACGTCTttcag
       actgaTGACGTCCatg
      gctcgtTGACGTCAccaaga
gagcggagcccgTGACGCGGccgagcggc
   tctctctttCCAGGTATctc
               ...
       ggcttTGACGTCAgcctggc
tggaatctctgcTGACGTCAcgacactccgca
  cggcgggcatTGACGTCAaacggcagc
acccctccccgcTGACCTCActcgagccgccg
```



$$M(i,x) = \log_2 \frac{\text{frequency of letter } x \text{ at position } i}{\text{background frequency of letter } x} \qquad (1)$$

$$\begin{array}{cc}
A & [\begin{matrix} -3.219 & -3.219 & 3.785 & -3.219 & 1.396 & -3.219 & 2.084 & 3.467 \end{matrix}] \\
C & [\begin{matrix} 1.396 & 1.396 & -3.219 & 3.585 & -3.219 & 2.488 & 3.334 & -3.219 \end{matrix}] \\
G & [\begin{matrix} 1.396 & 3.690 & -3.219 & 2.084 & 3.690 & -3.219 & 1.396 & 1.396 \end{matrix}] \\
T & [\begin{matrix} 3.585 & -3.219 & -3.219 & -3.219 & -3.219 & 3.467 & 1.396 & 2.084 \end{matrix}]
\end{array}$$

Figure 1: A Position Weight Matrix (PWM) modeling the CREB1 transcription factor binding site (from the JASPAR database) and the corresponding sequence logo [7] denoting columns with high or low information content. The coefficients are log-odds ratios of letter frequencies (computed thanks to equation 1): they indicate affinities between letters and positions at the binding site.

libraries, first released in 2007 [10], have deeply simplified the development on GPUs. Recent papers provide speedups on bioinformatics applications involving suffix trees [11, 12] or again Smith-Waterman comparisons [13] or motif discovery [14]. See [15] for a review on bioinformatics on GPUs.

The current Nvidia architectures [10] offer two levels of parallelism. For the coarse-grained level, several multiprocessors execute *blocks* of independent computations. Each multiprocessor is then a kind of large SIMD device, able to process several different fine-grained *threads* at a given time. All those threads are executing exactly the same instructions: if a *divergence* inside a conditional expression occurs, the two branches are serialized. A 16 KB *shared* memory is available for the threads in a same block. This local memory is very fast and should be used to maximize the efficiency.

1.3. Contents

The next section provides some bioinformatics background for this study, defining Position Weight Matrices (PWMs) and common PWM tasks. Section 3 addresses the Scan and MultipleScan problems, for which a simple parallelization is very efficient. Section 4 is related to the three problems ScoreToPvalue, PvalueToScore and Compare. Those three problems share a common NP-hard sub-problem, ScoreDistribution, whose current solutions are not suitable for parallelization. We propose a new algorithm, BucketScoreDistribution, that is both precise and suitable for parallelization. Section 5 reports our prototype implementation of the three first algorithms with the CUDA libraries [10], and discuss our solution compared to other softwares and other parallelizing techniques.

2. Background: Position Weight Matrices (PWMs)

2.1. Definitions

Given a finite alphabet Σ and a positive integer m, a PWM M is a matrix with $|\Sigma|$ rows and m columns (Figure 1). The coefficient[2] $M(p, x)$ gives the score at position p for the letter x in Σ. The PWM defines a function from Σ^m to $\mathbb{R}$, that associates a *score* to each word $u = u_1 u_2 \ldots u_m$ of Σ^m:

$$\texttt{Score}_M(u) = \sum_{p=1}^{m} M(p, u_p),$$

Let α be a score threshold. We say that M has an *occurrence* in a text T at position k if $\texttt{Score}_M(T_k \ldots T_{k+m-1}) \geq \alpha$. Biologically, a successful occurrence at a given position means that the TF corresponding to the TFBS has a chance to bind the DNA at this position.

2.2. PWM Tasks

The most recurrent task is to predict binding sites in a long DNA sequence, that is to look for occurrences of a PWM given a text and a score threshold. The two problems are:

- Scan. Given a matrix M with a score threshold α and a text T, find all the occurrences of M in T.

[2] Here the element M(i,j) refers to column i and row j.

- MultipleScan. Given a set of matrices $\mathcal{M} = \{M_1, \ldots, M_k\}$ with associated score thresholds $\alpha_1, \ldots, \alpha_k$ and a text T, find all the occurrences of each matrix of $\mathcal{M}$ in T.

These basic tasks are often involved into a more general analysis pipeline. After the occurrences have been found, a filter is applied in order to refine the results. For example, TFM-Explorer [16] requires to look for occurrences in all gene promoters regions (sequences of length 10000 bases that are upstream the genes, that is a total of almost 300 megabases for the human genome) for all matrices available in databases (about 800). Tools which compute occurrences often output the statistical significance, called the *P-value*, of each occurrence.

Genomes are modeled through a *background model.* The simpliest background model is when one considers identically and independently distributed character symbols. We call this the *iid background model* (which is a Markov Model of order 0). Extensions to Markov Models with higher orders are possible, but most of the time the iid background model is used.

The P-value $\texttt{Pv}_M(s)$ is defined as probability that the background model achieves a score at least equal to s: it is the proportion of words u (randomly chosen according to the observed letter frequencies) whose score is greater than s. In the iid background model, the P-value is simply

$$\texttt{Pv}_M(s) = \frac{|\{u \in \Sigma^m \,|\, \texttt{Score}_M(u) \geq s\}|}{|\Sigma|^m}$$

but other background models can be used by weighting the words u with their relative probability in the background model. Figure 2 illustrates the definition.

A P-value close to 0 means that the fact the matrix achieves such a score is an exceptional event. Biologically, this means that the probability that the corresponding TF binds the DNA at this position is very high. On the contrary, a P-value close to 1 means that the score is random.

To decide if a PWM occurs at a position in a text, the score threshold has to be chosen. Once again, the P-value is used to determine the score threshold. Generally, a P-value p is chosen independently from the matrix and the background model: this P-value reflects the expected number of occurrences that will be found in the text [17]. Common thresholds used in real applications range from 10^{-7} to 10^{-3}. Then the score threshold is computed for a given matrix M: the goal is to find the score α such that

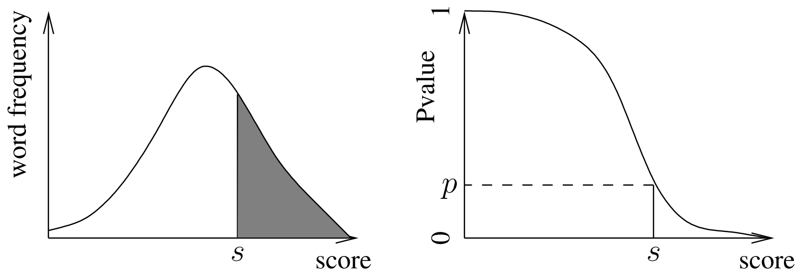


Figure 2: Illustration of the P-value. Left: The distribution of the scores among all possible words. Right: The corresponding P-value. Considering an iid background model, the P-value p for the score s is the sum of the frequencies of all the scores greater than s: That is the gray area on the left side.

$\mathtt{Pv}_M(\alpha) = p$. The two underlying problems, which are both NP-hard [18, 19], are:

- ScoreToPvalue. Given a matrix M and a score s, compute the P-value $\mathtt{Pv}_M(s)$.
- PvalueToScore. Given a matrix M and a P-value p, compute the score s such that $\mathtt{Pv}_M(s) = p$.

Another task that attracted interest in the past few years is the design of a method to compare matrices. Column-to-column comparisons with correlation coefficients have been proposed by several authors [20, 21, 22]. A better method is to consider that two matrices are similar if their occurrences are almost the same [23]. The formalized problem is:

- Compare. Given two matrices M and M' of same length with respective score thresholds α and α', compute the number of words with a score greater than α for M and than α' for M'.

Being able to compare matrices serves several goals. Firstly, it can be used to detect if two matrices are similar. This is useful to remove redundancy from databases or to test if a new matrix has a similar one in a database [21, 22, 23]. Secondly, it can be used to improve the Scan problem (see page 7).

3. Looking for occurrences of matrices

Finding all the occurrences of a matrix M of length m in a text T of length n may be done by a naive algorithm in $O(mn)$ time: for each position k of the text, $\texttt{Score}_M(T_k \dots T_{k+m-1})$ is computed.

3.1. Optimizations for Scan

In 2000, [24] proposed an improvement: for a given word $u \in \Sigma^m$, the computation of $\texttt{Score}_M(u)$ can be stopped as soon as

$$\texttt{Score}_{M[1..j]}(u_1 \dots u_j) < \alpha - \max M[j+1..m]$$

with $\max M[j+1..m] = \sum_{k=j+1}^{m} \max M[k]$, where $\max M[k]$ is the maximal score of the k-th column of the matrix and α the score threshold. Indeed, when the above condition is met, the score of u cannot be greater than α. This property does not change the $O(mn)$ worst-time complexity, but gives an average $O(m'n)$ time complexity, where m' is the average stop position. We reference this algorithm as the Lookahead Strategy Algorithm (LSA for short).

In 2006, [25] proposed to precompute an index for the Scan of one or several matrices. The main idea was to split the matrix into sub-matrices called *slices* of length ℓ (typically 7 or 8, depending on the available memory). For each slice, the $|\Sigma|^\ell$ scores for each word are computed and stored into a table. The time complexity remains $O(m'n)$, but with only $O(m'n/\ell)$ memory accesses. When several matrices are scanned (MultipleScan), the table is organized such that scores for the set of matrices are in the same memory location, thus avoiding memory latencies. On a MultipleScan involving a large set of matrices, [25] reported a practical 8× speedup compared to the LSA. For a single matrix, no significant speedup is obtained. We call this idea the Slices Strategy.

Lastly, similarities between matrices can lead to another improvement when searching for occurrences: one can avoid to look for occurrences of each matrix but only for occurrences of one representative matrix [21, 25]. Occurrences of the other matrices are then computed only for positions where the representative matrix occurred.

3.2. Preprocessing and indexation

Ideas can also be borrowed from the algorithms searching a pattern in a text, as with the classical Aho-Corasick [26], Knuth-Morris-Pratt (KMP)

[27], or Boyer-Moore [28] algorithms and their variants. This is more difficult than in the usual pattern matching case, as the combination of the scores does not always allow large shifts in the matrices. With KMP, [29] obtains a 2× – 3× improvement on the LSA method. With Aho-Corasick, [30, 31] obtains better speedups. For such techniques, the size of the automaton becomes problematic for large matrices.

Instead of indexing the matrices, another way to speedup the Scan problem is to preprocess the text: [32] uses suffix trees (speedup 2× – 5×), [33] uses suffix arrays and [34] compress the text (speedup 2× – 5×). Some parallel implementations of suffix trees have been reported [11], but those parallelizations are especially difficult due to the non-locality of memory accesses and the structure of the tree.

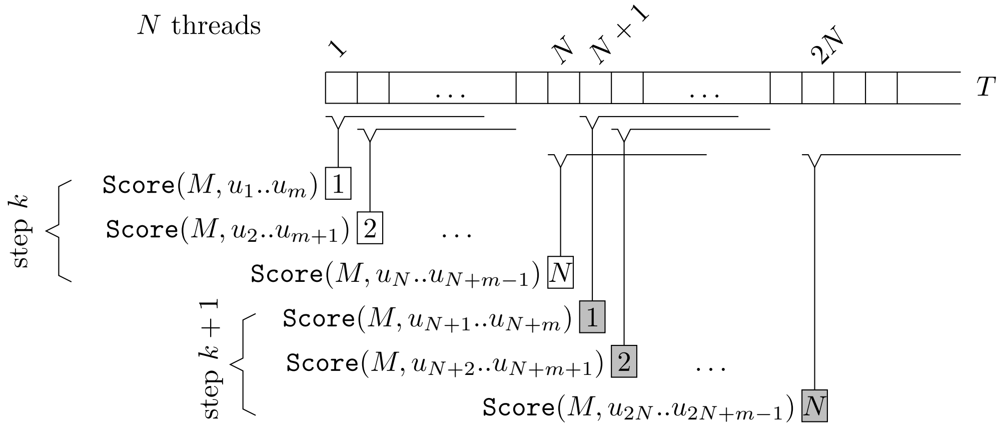
In fact, those evolved data structures do not fit well with *approximate* pattern matching. The PWM Scan problem is far more difficult, as it can be seen as a generalized pattern matching with a complex error function. Other solutions could use seed-based indexing, in particular spaced seeds that handle better errors [35].

3.3. A simple parallelization of Scan

The parallelization of the Scan problem can be done easily by splitting the different positions of the text across several threads (Figure 3). Each block of threads works on a segment of the text (of size 2048), that is copied into the local shared memory. At a given time, each thread computes the score of one word. If some occurrences are found, their positions and scores are stored into the local memory, and copied to the global memory at the end of the block.

As already mentioned, Liefooghe *et al.* [25] obtained a good speedup by preprocessing the matrices in an index. This Slices Strategy could here bring a small improvement. The limiting factor for the size of the slices is the amount of available memory. For a parallel architecture, the tables with the slices can be stored in a local, quickly accessible memory, or a more distant but large memory. As the speedup is barely $O(\ell)$ for a $O(|\Sigma|^\ell)$ memory size, on most architectures the solution with local memory will be faster. Section 5 details some results on the Nvidia GeForce 8 or GT200 architecture.

On the contrary, LSA and KMP-like improvements do not apply on SIMD-like architectures such as GPU. Their efficiency is indeed based on a variable number of iterations in the most inner loops. As even close words do not always lead to the same number of iterations, different threads with



N threads
1
N
N + 1
2N
. . .
. . .
T
step k
Score($M, u_1..u_m$) 1
Score($M, u_2..u_{m+1}$) 2
. . .
Score($M, u_N..u_{N+m-1}$) N
step k + 1
Score($M, u_{N+1}..u_{N+m}$) 1
Score($M, u_{N+2}..u_{N+m+1}$) 2
. . .
Score($M, u_{2N}..u_{2N+m-1}$) N

Figure 3: Parallel GPU Scan. The sequence is divided into small segments that are processed by several blocks. The scores at each position are distributed across several threads: N threads compute the scores of words from position 1 to N, then all threads are shifted by N characters onto the sequence.

different words will diverge most of the time, thus providing a very bad parallel performance.

3.4. Parallelization of MultipleScan

Real instances of the Scan problem require to compute the occurrences of a set of matrices, as example to look for all known matrices in a newly sequenced genome.

In a parallel implementation of the brute-force algorithm, the best way to deal with several matrices is to put the loop over the matrices as deep as possible to avoid memory latencies due to sequence copy. This solution is far better than repeating the Scan over the matrices. The straightforward strategy is thus to copy several matrices once in the shared memory before the loop over the matrices.

In the MultipleScan problem, the Slices Strategy is again better. We saw that the limiting factor of this strategy is the memory latencies. However, in the case of several matrices, the index is common: for each slice, only one memory access is sufficient to get the values for all matrices. It remains that the slices could not be very large and the number of matrices must be small if one wants to store the index in shared memory. In this case, the set of matrices is divided into smaller ones to fit the constraints imposed by the available memory. On the other side, it is also possible to store a larger index in global memory. We will discuss the two approaches in the Section 5.2.

4. Computing score threshold, P-value and comparing matrices

4.1. Methods for score threshold and P-value computation

The computation of the P-value (ScoreToPvalue) can be done using probability generating functions or dynamic programming [36, 17, 37, 18, 19]. In both cases, the time complexity is $O(S)$, where S is the number of possible different scores. Let $Q_M(s)$ be the probability to achieve exactly the score s. For a given score s', the P-value is obtained with the equation:

$$\texttt{Pv}_M(s') = \sum_{s \geq s'} Q_M(s)$$

Computing ScoreToPvalue (that is the score associated to a given P-value) can be done by adding the values of $Q_M(s')$ from the maximal score until the desired P-value is obtained.

Let $M[1..i]$, $0 \leq i \leq n$, denote the matrix consisting of the columns 1 to i of M. The matrix $M[1..0]$ is the empty matrix. In a iid background model, the $Q_{M[1..i]}$ score distribution can be expressed from the $Q_{M[1..i-1]}$ score distribution by the following dynamic programming algorithm, where $p(x)$ is the probability of the letter x in the background model [17]:

$$\begin{aligned} Q_{M[1..0]}(s) &= \begin{cases} 1 & \text{if } s = 0 \\ 0 & \text{otherwise} \end{cases} \\ Q_{M[1..i]}(s) &= \sum_{x \in \Sigma} Q_{M[1..i-1]}(s - M(i,x)) \times p(x) \end{aligned}$$

If the matrix has non-negative integer coefficient values, then S, the number of possible different scores, is bounded by $\sum_{i=1}^{m} \max M[i]$. Some polynomial algorithms use these conditions. However, PWMs are built from log-ratios and do not fulfill this constraint: S can be as large as $|\Sigma|^m$, and thus the worst-case time complexity is $O(|\Sigma|^m)$. Some methods [33, 19, 23] round the coefficients of the matrix to maintain S low. Even if those methods claim to compute exact P-values, this rounding induces an error on the P-value and on the score threshold [19]. As an example, the implementation of Mosta [23] rounds each column of the matrix to the nearest multiple of $\varepsilon = 0.05$, and thus the total score has an error of at most $m\varepsilon$.

4.2. A new algorithm to compute the score distribution

We propose to compute the score distribution by splitting the matrix M in N slices $M_1, M_2, \ldots M_N$, and by combining the score distributions of the slices (Figure 4):

$$Q_M(s) = \sum_{s_1+\ldots+s_N=s} (Q_{M_1}(s_1) \times \ldots \times Q_{M_N}(s_N))$$

In the following algorithm, we use tables with B elements called *buckets*. For any slice M_i, the scores are in the range $[\min M_i, \max M_i]$, and we store this distribution in a table with B buckets, thus rounding down the scores to the nearest multiple of Δ_{M_i}, where $\Delta_{M_i} = (\max M_i - \min M_i)/B$. We denote by $\underline{s}$ the discretized score of s, $\mathcal{S}_M$ the set of all possible scores the matrix M can achieve and $\underline{\mathcal{S}}_M$ the set of all possible discretized scores the matrix M can achieve.

Algorithm BucketScoreDistribution

- For each slice M_i, compute a score distribution Q_{M_i} by enumeration of $4^{m/N}$ words, rounding the scores of each word. The result is stored in a table Q_{M_i} with B entries. For a score s in $\underline{\mathcal{S}}_{M_i}$,

$$Q_{M_i}[s] = \sum_{\substack{s' \in \mathcal{S}_{M_i} \\ \underline{s'} = s}} Q_{M_i}(s') \tag{2}$$

  Note that we round the score of each word, and not of each column, thus keeping the error low.

- A score distribution for M is then computed by recursively merging the score distributions of the N slices (Figure 4). For a score s in $\underline{\mathcal{S}}_{M_{1\oplus 2}}$,

$$Q_{M_{1\oplus 2}}[s] = \sum_{\substack{s_1 \in \underline{\mathcal{S}}_{M_1} \\ s_2 \in \underline{\mathcal{S}}_{M_2} \\ \underline{s_1 + s_2} = s}} Q_{M_1}[s_1] \times Q_{M_2}[s_2]$$

  All the tables have B buckets. One merge can be done in $O(B^2)$ time, for a total time of at most $O(NB^2)$. The merge can also be computed in $O(NB \log B)$ time using rapid convolution algorithms [38], but here this step is not a limiting factor.

Assuming that m is large enough, the time complexity of BucketScoreDistribution is $O(N4^{m/N})$, enabling the study of matrices N times larger than the previous methods.

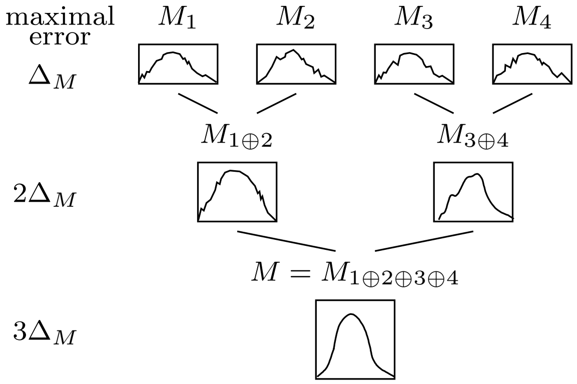


Figure 4: Algorithm BucketScoreDistribution. The matrix is split in several submatrices (4 here). The score distributions are computed for them with an error bounded by Δ_M. Then they are combined to obtain the overall score distribution.

Once $Q_{M_{1\oplus 2\oplus \cdots \oplus N}}$ is known, the final step of computation, in $O(B)$ time, depends on the problem. For ScoreToPvalue, we obtain an approximate P-value, $\texttt{Pv}_M[s]$, defined for a score s in $\underline{\mathcal{S}}_M$ by

$$\texttt{Pv}_M[s] = \sum_{\substack{s' \in \underline{\mathcal{S}}_M \\ s' \geq s}} Q_M[s']$$

For PvalueToScore, the score threshold s in $\underline{\mathcal{S}}_M$ is the biggest one such that

$$p \leq \sum_{\substack{s' \in \underline{\mathcal{S}}_M \\ s' \geq s}} Q_M[s']$$

4.3. Precision evaluation

We now bound the error induced by the score discretization and the error induced by the convolution to show that they are similar to the errors of other algorithms. At the first step, the maximum error when discretizing the scores for a slice M_i is Δ_{M_i}. When combining two slices M_i and M_j into $M_{i\oplus j}$, the equation (2) is not more valid: now the maximum total error is

$$\Delta_{M_i} + \Delta_{M_j} + \Delta_{M_{i\oplus j}}$$

where $\Delta_{M_{i \oplus j}}$ is the maximum error when discretizing the result. As the combined scores are in the range $[\min M_i + \min M_j, \max M_i + \max M_j]$, we have $\Delta_{M_{i \oplus j}} = \Delta_{M_i} + \Delta_{M_j}$. The maximum total error is thus $2\Delta_{M_{i \oplus j}}$. With N slices, the maximal total error on M is

$$\lceil 1 + \log N \rceil \sum \Delta_{M_i} \;=\; \lceil 1 + \log N \rceil \Delta_M$$

with $\Delta_M = (\max M - \min M)/B$. This $O((\log N)/B)$ maximal error is on the scores. The actual error on the number of words depends on $\underline{s}$ and on the score distribution of M: it is at most the number of words in the $\lceil 1 + \log N \rceil$ buckets (gray area on the Figure 5). For the PvalueToScore and the ScoreToPvalue problems, the error on the P-value is thus

$$\mathtt{Pv}_M[\,\underline{s}\,] \leq \mathtt{Pv}_M(s) \leq \mathtt{Pv}_M[\,\underline{s} + \lceil 1 + \log N \rceil \Delta_M]$$

.

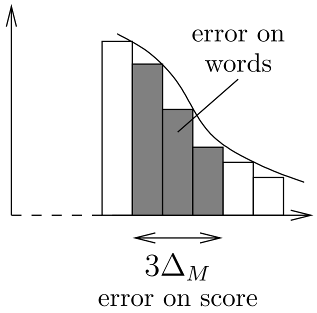
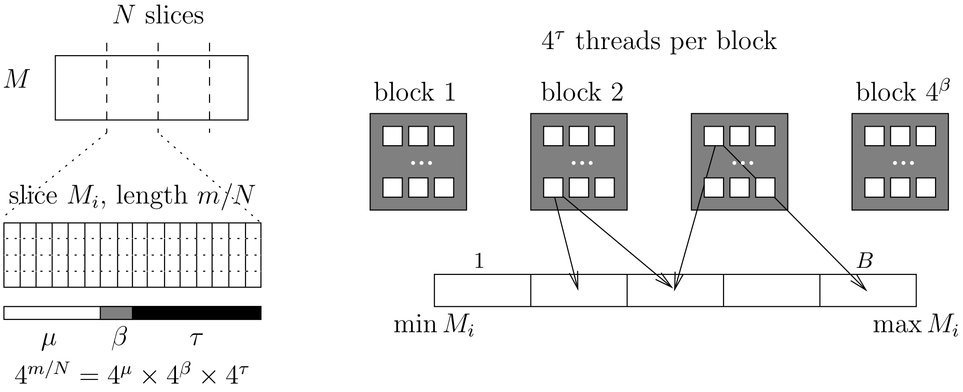


Figure 5: Error on P-value when computing the score distribution using BucketScoreDistribution. The curve shows the actual score distribution and the rectangles the one computed (scores on the x-axis, and the number of words achieving a score greater than a given one on the y-axis). The error when computing the score threshold is bounded by $3\Delta_M$ and the error on the number of words achieving a score greater than the threshold is bounded by the sum of heights of the grey rectangles.

4.4. Parallelization

This new algorithm is perfectly suited for parallelization, as the word enumerations can be split across different independent computations. In our GPU prototype implementation, the enumeration of $4^{m/N}$ words is split on 4^β blocks with 4^τ threads by block, leaving 4^μ words to enumerate within each thread, with $\mu = m/N - (\beta + \tau)$ (Figure 6, on the left). At a given time, all the threads of a same block are enumerating the same μ leftmost characters. From one word to another, the score is evaluated only on the modified positions, thus bringing no divergence between the threads of a same block. Then each thread increments one of the B buckets (Figure 6, on the right). As several threads can increment the same bucket at a given time, atomic instructions or similar mechanisms must be used there. The final merging operations, in $O(NB^2)$ time, can be performed on the host.



N slices
M
slice M_i, length m/N
μ
β
τ
$4^{m/N} = 4^\mu \times 4^\beta \times 4^\tau$
4^τ threads per block
block 1
block 2
block 4^β
1
B
$\min M_i$
$\max M_i$

Figure 6: Parallel GPU BucketScoreDistribution. Each thread enumerates 4^μ words and there are $4^\beta \times 4^\tau$ threads for computing the score distribution of a given slice of the matrix.

4.5. Similarity between two matrices

The computation of similarity between two matrices is very similar. Given two matrices[3] M and M' of length m with their respective score thresholds α and β, the goal is to measure $\mathtt{TP}_M^{M'}$, the number of true positive words $u \in \Sigma^m$ such that $\mathtt{Score}_M(u) \geq \alpha$ and $\mathtt{Score}_{M'}(u) \geq \beta$. This number can be computed with the following equation:

$$\mathtt{TP}_M^{M'}(\alpha, \beta) = \sum_{s \geq \alpha, s' \geq \beta} Q_M^{M'}(s, s')$$

[3] If the two matrices have different lengths m and m', one has to choose an alignment, *i.e.* a shift between the two matrices. Then it is sufficient to pad the two matrices with zeros to obtain two matrices of same length. When one wants to compute such a similarity score, one has to compute TP for all possible $m + m'$ shifts.

where $Q_M^{M'}(s, s')$ is now the probability to achieve exactly the score s with M and the score s' with M'. False positives, true negatives and false negatives are computed in a similar way. The score distribution is now expressed by the following recurrence [25]:

$$Q_{M[1..0]}^{M'[1..0]}(s, s') = \begin{cases} 1 & \text{if } s = 0 \text{ or } s' = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$Q_{M[1..i]}^{M'[1..i]}(s, s') = \sum_{x \in \Sigma} Q_{M[1..i-1]}^{M'[1..i-1]}(s - M(i, x), s' - M'(i, x)) \times p(x)$$

The same BucketScoreDistribution algorithm applies, but now the B buckets induce a maximal $O((\log N)/\sqrt{B})$ error on words.

5. Results and discussion

5.1. Testing environment

We benchmarked the four Scan, MultipleScan, ScoreToPvalue and PvalueToScore algorithms with the CUDA 2.3 libraries from Nvidia [10]. The sources or our implementation are available from our web site[4]. Two GPU generations were tested: the Nvidia GeForce 8800 (GeForce 8 architecture, 16 × 8 cores, 1.3 GHz, 768 MB RAM), and the Nvidia GTX 280/285 (GT200 architecture, 30 × 8 cores, 1.3/1.4 GHz, 1 GB RAM).

The host systems and the CPUs are the following: CPU 1: Intel Core 2 Duo 6600 (2.40 GHz) with 3 GB RAM and 4 MB cache, CPU 2: Xeon W3520 (2.66 GHz) with 1 GB RAM and 8 MB cache. These CPU have several cores, but only one core was used in the benchmarks. The compiler was Nvidia `nvcc` used with the `-O3` option.

Benchmarks were done on real data (Table 1 and Figure 12) and on random data (Figures 7 to 11). We found no significant difference in terms of speedups between those datasets.

[4] `http://bioinfo.lifl.fr/TFM/TFM-CUDA`

5.2. Scan *and* MultipleScan

For the parallel Scan, the target sequence was always in the shared memory, enabling different threads in the same block to work on the same data (see Figure 3). Table 1 details the results on a human chromosome with 225×10^6 nucleotides. Using the GPU implies an overhead of 0.27s. This overhead is mostly due to the transfer of the sequence data to the GPU (even large collections of matrices are small compared to megabytes of sequence data). The overhead is negligible starting with matrices of length 30, or as soon as a collection of matrices is scanned, as there is in this case only one transfer of the sequence.

matrix	length	init + I/O	kernel	total
JASPAR 0075	5	0.27	0.27	0.54
JASPAR 0023	10	0.27	0.34	0.61
JASPAR 0106	20	0.27	0.48	0.75
random	40	0.27	2.04	2.31
random	80	0.27	2.88	3.15

Table 1: Results for parallel Scan on GeForce 8800, on the chromosome 1 of the human genome ($225 \cdot 10^6$ bases, release hg18). Times are in seconds.

Figures 7, 8 and 9 detail large-scale benchmarks, comparing the times of the GPU implementations to a 1-thread CPU implementation. For small sequences (Figures 7), no significant speedup is achieved. The best speedups, between 15× and 20×, are obtained when the kernels perform a large number of computations, either on long sequences, from $30 \cdot 10^9$ positions computed (Figure 8), or on MultipleScan, when several matrices are simultaneously searched (Figure 9).

Those speedups should be compared to a maximum 8× speedup using 128-bit SIMD instructions using a 16-bit precision, and to the 2× to 8× speedups of the methods cited on page 7. We also provide comparisons with Moods [30, 31], with default settings. As expected, Moods is a very good solution for small matrices, but does not scale to larger matrices.

The Slices Strategy was tested with several slices length, and, as a consequence, with different limits in the number of matrices per slices. To fit in the shared memory, one can use all 123 matrices of the JASPAR collection with no slices ($\ell = 1$), or sets of 4 matrices sliced with $\ell = 2$. In the global memory, sets of 50 matrices can be stored with slices of length $\ell = 7$. In all cases, the search is done sequentially over different sets. The results are

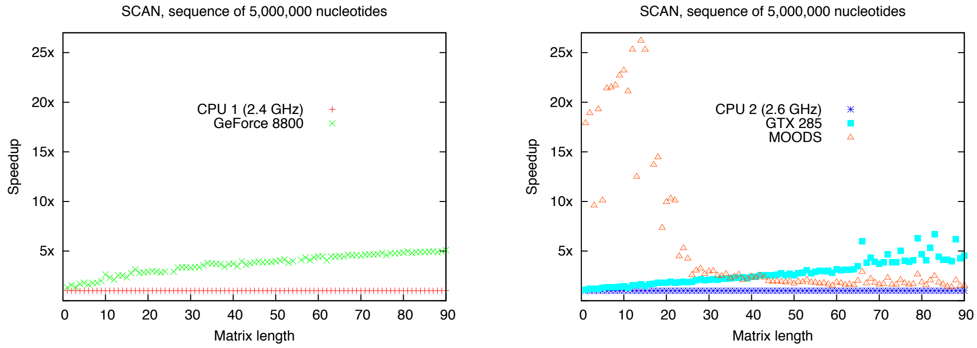


Figure 7: Results for SCAN (random matrices) on a small random sequence (length $5 \cdot 10^6$). No significant speedup is achieved.

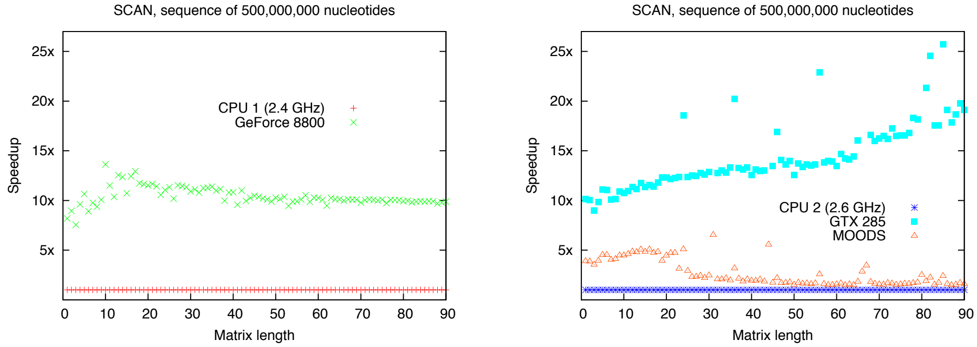


Figure 8: Results for SCAN (random matrices) on a long random sequence (length $500 \cdot 10^6$).

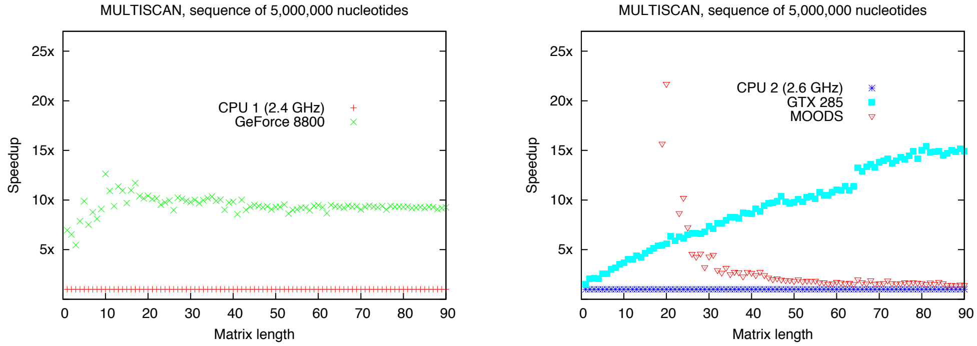


Figure 9: Results for MULTIPLESCAN (50 random matrices) on a small random sequence (length $5 \cdot 10^6$).

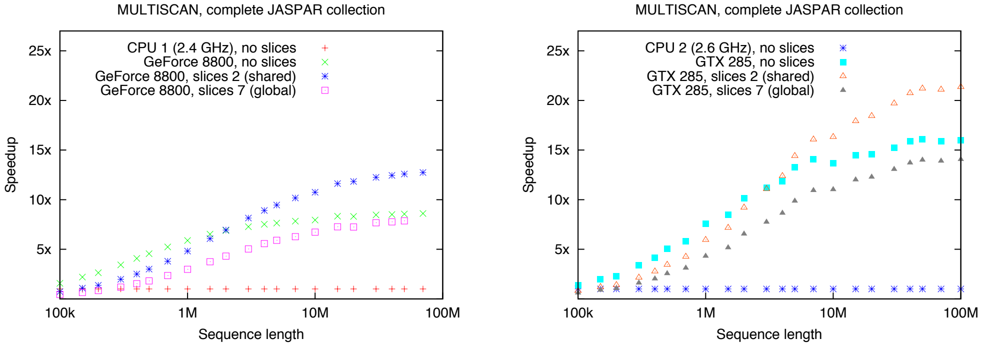


Figure 10: Results for MultipleScan on 123 JASPAR matrices (random sequences). Different strategies are used to store the index: no slices, slices of length $\ell = 2$ in shared memory, and slices of length $\ell = 7$ in global memory.

presented Figure 10. The algorithmic gain of using the slices is visible while one works into the shared memory.

As expected, the strategies with some divergence in the threads (LSA and KMP) do not provide any speedup (results not shown): the naive algorithm is here the more efficient algorithm to parallelize.

5.3. Algorithms based on BucketScoreDistribution

Implementation and speedups. We required that the bucket table fit in the shared memory, leading to $B = 3600$ 32-bit buckets for one matrix. Figure 11 details the results for PvalueToScore with $N = 4$. The computation times are the same for the ScoreToPvalue problem. Due to some overheads in initializations and in memory transfers between the host and the GPU (results not shown, similar to Table 1), a significant speedup starts only from a slice size of $\ell = 48$, that is from the enumeration of 4^{12} words. Because of its increased number of cores, the GTX 285 gives a 80% increased speedup on GeForce 8800. For $\ell = 65$, the CPU takes 1426 seconds: the speedup is 3.6× for the GeForce 8800 and 9.8× for the GTX 285.

The main limitation in those speedups is the bucket incrementation: most of the time, several threads are incrementing the same bucket. On the GeForce 8800, this incrementation is serialized between the threads of a same block at the end of each score computation. An improved algorithm could have here better results. On the GTX 280 and 285, we use atomic instructions to increment the buckets in shared memory. This gives an additional 8× speedup compared to the serialized incrementation. For $\ell = 65$, the total speedup is thus 77× on the GTX 280 against the CPU 1.

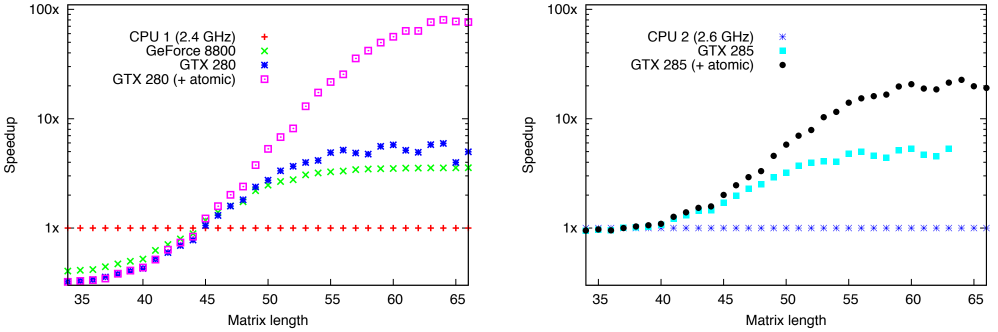


Figure 11: Results for PvalueToScore, with $N = 4$ slices on random matrices (other N give similar speedups).

Precision. For algorithms based on BucketScoreDistribution, we do not report times with other methods. Indeed, beside the parallelization, the times mainly depends on N. However, we must state what precision is achieved.

Figure 12 details the errors on P-value when computing PvalueToScore for all matrices of JASPAR with length ≥ 10. We accept a relative error of 0.1, suitable for the usage of PWMs in computational biology (for a P-value of 10^{-5}, the error will be at most 10^{-6}). Here 83 out of 101 matrices (of which all matrices of length ≥ 13) fulfill this precision for a P-value of 10^{-5}. The relative error decreases when the P-value is higher and when the matrix is longer. In both cases, this is because the position of the score $\underline{s}$ is more "on the left" relative to the shape of the score distribution.

As an example, let us take the JASPAR matrix MA0066 modeling an hormone nuclear receptor. This matrix has a length 20, and its total raw log score ranges from -52.052 to 22.385. With $N = 2$ and $B = 3600$, the maximal error made by BucketScoreDistribution is 0.042. As a comparison, the maximal error done by Mosta on the same matrix is $20 \times \varepsilon = 20 \times 0.05 = 1$.

Our algorithm runs in less than 2 seconds, whereas Mosta is almost immediate. However, Mosta does not scale to better precisions: setting ε to lower values leads to longer running time, and Mosta does not end when setting ε to 0.042/20.

Other parallelization techniques. On any multi-core architecture, the BucketScoreDistribution $4^{m/N}$ enumerations can be split among several cores. Even on general purpose CPUs, SIMD techniques can benefit from BucketScoreDistribution, as only one common memory access is needed between all the 4^{τ} "threads" that process the same leftmost characters at a given iteration (see page 13). Nevertheless, at 16-bit precision, the maximum theoretical speedup using SSE 128-bit SIMD extensions is 8×, giving a maximum 32× speedup on a quad-core CPU. These speedups remain beyond the 77× speedup achieved on GTX 285.

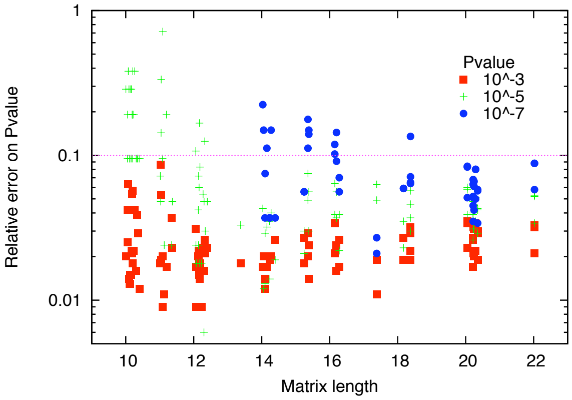


Figure 12: Relative error on P-value when computing PvalueToScore with $N \in [2, 4]$ for all 101 matrices of JASPAR of length ≥ 10.

5.4. Pipeline integration

Pipelines are common in bioinformatics. More specifically, projects such as BioPerl [39], Biopython [40] or BioJava [41] help the linking of several tools, handling different input and output formats through methods in a unified framework.

Through the `biomanycores.org` project [42], we developed interfaces for TFM-CUDA to the three frameworks cited above. With such an integration, performances are decreasing. We still observe a speedup of around 6× on large Scan queries (a few seconds) through a Biopython interface, demonstrating that such developments may offer good performance in real cases.

6. Conclusion and perspectives

We proposed parallel Scan and MultipleScan of one or several Position Weight Matrices (PWMs) in a text, and a new algorithm, BucketScoreDistribution, that computes the score distribution of a PWM and that resolves the ScoreToPvalue, PvalueToScore and Compare problems. The error on P-value induced by the BucketScoreDistribution algorithm for the ScoreToPvalue and PvalueToScore problems is not more than $O((\log N)/B)$, where B is the number of buckets used to store the distributions. The BucketScoreDistribution can be adapted to any parallel architecture, as it involves large blocks of independent computations. On a GPU, threads simultaneously enumerate neighbor words without divergence. The only bottleneck was in the buckets incrementation. The best speedup is here obtained through the use of atomic instructions on the GT200 architecture.

Further work could be done on benchmarking the Compare problem on real data to see if the $O((\log N)/\sqrt{B})$ error is suitable for clustering applications. Other perspectives include testing and improving the BucketScoreDistribution algorithm on other many-core architectures, in particular through the new OpenCL standard [43].

Acknowledgments

This research was carried out with a grant "Action Transversale Calcul Haute-Performance LIFL" and through the "NVIDIA Professor Partnership" program. We thank anonymous reviewers for comments on a previous version on the manuscript.

[1] M. Giraud, J.-S. Varré, Parallel position weight matrices algorithms, in: International Symposium on Parallel and Distributed Computing (ISPDC 2009), 2009, pp. 65–69.

[2] Central dogma of molecular biology, Nature 227 (5258) (1970) 561–563.

[3] A. Sandelin, W. Alkema, P. Engström, W. Wasserman, JASPAR: an open-access database for eukaryotic transcription factor binding profiles, Nucleic Acids Research 32 (2004) D91–D94.

[4] E. Wingender, , X. Chen, R. Hehl, H. Karas, I. Liebich, V. Matys, T. Meinhardt, M. Pruss, I. Reuter, F. Schacherer, TRANSFAC: an integrated system for gene expression regulation, Nucleic Acids Research 28 (1) (2000) 316–319.

[5] J. Shendure, H. Ji, Next-generation DNA sequencing, Nat. Biotech 26 (10) (2008) 1135–1145.

[6] G. Robertson, M. Hirst, M. Bainbridge, M. Bilenky, Y. Zhao, T. Zeng, G. Euskirchen, B. Bernier, R. Varhol, A. Delaney, N. Thiessen, O. L. Griffith, A. He, M. Marra, M. Snyder, S. Jones, Genome-wide profiles of STAT1 DNA association using chromatin immunoprecipitation and massively parallel sequencing, Nat. Meth. 4 (8) (2007) 651–657.

[7] G. E. Crooks, G. Hon, J. M. Chandonia, B. S. E., Weblogo: A sequence logo generator, Genome Research.

[8] M. Charalambous, P. Trancoso, A. Stamatakis, Initial experiences porting a bioinformatics application to a graphics processor, Adv. in Informatics (2005) 415–425.

[9] W. Liu, B. Schmidt, G. Voss, W. Müller-Wittig, GPU-ClustalW: using graphics hardware to accelerate multiple sequence alignment, in: High Performance Computing (HiPC 2006), LNCS 4297, 2006, pp. 363–374.

[10] Nvidia CUDA programming guide 2.0 (2008).

[11] M. C. Schatz, C. Trapnell, A. L. Delcher, A. Varshney, High-throughput sequence alignment using graphics processing units., BMC Bioinformatics 8 (2007) 474.

[12] C. Trapnell, M. C. Schatz, Optimizing data intensive gpgpu computations for dna sequence alignment, Parallel Computing 35 (8-9) (2009) 429–440.

[13] S. A. Manavski, G. Valle, CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment, BMC Bioinformatics 9 (S2) (2008) S10.

[14] Y. Liu, B. Schmidt, W. Liu, D. L. Maskell, Cuda-meme: Accelerating motif discovery in biological sequences using cuda-enabled graphics

processing units, Pattern Recognition Letters In Press, Corrected Proof (2009) –.

[15] J.-S. Varré, B. Schmidt, S. Janot, M. Giraud, Genome-scale Pattern Analysis in the Post-ENCODE Era, (to appear), 2010, Ch. Manycore high-performance computing in bioinformatics.

[16] M. Defrance, H. Touzet, Predicting transcription factor binding sites using local over-representation and comparative genomics, BMC Bioinformaticsdoi:doi:10.1186/1471-2105-7-396.
URL `http://www.biomedcentral.com/1471-2105/7/396/abstract`

[17] J. M. Claverie, S. Audic, The statistical significance of nucleotide position-weight matrix matches, CABIOS 12 (5) (1996) 431–9.

[18] J. Zhang, B. Jiang, M. Li, J. Tromp, X. Zhang, M. Zhang, Computing exact p-values for DNA motifs, Bioinformatics 23 (5) (2007) 531–537. doi:10.1093/bioinformatics/btl662.

[19] H. Touzet, J.-S. Varré, Efficient and accurate p-value computation for position weight matrices, Algorithms for Molecular Biology 2 (1).

[20] D. E. Schones, P. Sumazin, M. Q. Zhang, Similarity of position frequency matrices for transcription factor binding sites, Bioinformatics 21 (3) (2005) 307–313.

[21] S. M. Kielbasa, D. Gonze, H. Herzel, Measuring similarities between transcription factor binding sites, BMC Bioinformatics 6 (237) (2005) 1–11.

[22] S. Gupta, J. A. Stamatoyannopoulos, T. L. Bailey, W. S. Noble, Quantifying similarity between motifs, Genome Biology 8 (2).

[23] U. J. Pape, S. Rahmann, M. Vingron, Natural similarity measures between position frequency matrices with an application to clustering., Bioinformatics 24 (3).

[24] T. D. Wu, C. G. Nevill-Manning, D. L. Brutlag, Fast probabilistic analysis of sequence function using scoring matrices., Bioinformatics 16 (3) (2000) 233–244.

[25] A. Liefooghe, H. Touzet, J.-S. Varré, Large scale matching for position weight matrices, in: Combinatorial Pattern Matching (CPM 2006), LNCS 4009, 2006, pp. 401–412.

[26] A. Aho, M. Corasick, Efficient string matching: An aid to bibliographic search, Comm. ACM 18 (1975) 333–340.

[27] D. E. Knuth, J. H. Morris, V. R. Pratt, Fast pattern matching in strings, SIAM J. Comp. 6 (1) (1977) 323–360.

[28] R. S. Boyer, J. S. Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 762–772.

[29] A. Liefooghe, H. Touzet, J.-S. Varré, Self-overlapping occurrences and Knuth-Morris-Pratt algorithm for weighted matching, in: LATA 2009, to appear.

[30] C. Pizzi, P. Rastas, E. Ukkonen, Fast search algorithms for position specific scoring matrices, in: BIRD 2007, LNCS 4414, 2007, pp. 239–250.

[31] J. Korhonen, P. Martinmäki, C. Pizzi, P. Rastas, E. Ukkonen, Moods: fast search for position weight matrix matches in dna sequences, Bioinformatics 25 (3) (2009) 3181–3182.

[32] B. Dorohonceanu, C. G. Nevill-Manning, Accelerating Protein Classification Using Suffix Trees, in: ISMB 2000, 2000, pp. 128–133.

[33] M. Beckstette, R. Homann, R. Giegerich, S. Kurtz, Fast index based algorithms and software for matching position specific scoring matrices, BMC Bioinformatics 7.

[34] V. Freschi, A. Bogliolo, Using sequence compression to speedup probabilistic profile matching, Bioinformatics 21 (10) (2005) 2225–9.

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

[36] R. Staden, Methods for calculating the probabilities of finding patterns in sequences, CABIOS 5 (2) (1989) 89–96.

[37] S. Rahmann, Dynamic programming algorithms for two statistical problems in computational biology, in: WABI 2003, LNCS 2812, 2003, pp. 151–164.

[38] D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addison-Wesley, 1997.

[39] J. E. Stajich, D. Block, K. Boulez, al., The Bioperl toolkit: Perl modules for the life sciences, Genome Research 12 (10) (2002) 1611–1618.

[40] P. J. A. Cock, T. Antao, J. T. Chang, B. A. Chapman, C. J. Cox, A. Dalke, I. Friedberg, T. Hamelryck, F. Kauff, B. Wilczynski, M. J. de Hoon, Biopython: freely available Python tools for computational molecular biology and bioinformatics, Bioinformatics (2009) btp163.

[41] R. C. G. Holland, T. A. Down, M. Pocock, A. Prlic, D. Huen, K. James, S. Foisy, A. Drager, A. Yates, M. Heuer, M. J. Schreiber, BioJava: an open-source framework for bioinformatics, Bioinformatics 24 (18) (2008) 2096–2097.

[42] J.-S. Varré, S. Janot, M. Giraud, Biomanycores, a repository of interoperable open-source code for many-cores bioinformatics, in: Bioinformatics Open Source Conference, 2009.

[43] The Khronos Group, OpenCL 1.0 specification (2008).