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
mathieu.giraud@lifl.fr, jean-stephane.varre@lifl.fr

Abstract

Position Weight Matrices (PWMs) are broadly used in computational biology. The basic problem, SCAN*, aims to find the occurrences of a given PWM in large 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 *and* BUCKETSCOREDISTRIBUTION *with the CUDA libraries, and report for the different problems speedups of* $21\times$ *and* $77\times$ *on a Nvidia GTX 280.*

1. Introduction

Position Weight Matrices (PWMs). Position Weight Matrices (PWMs) are broadly used in computation biology to model conserved sequence patterns. The most common application of PWMs is about gene regulation: the transcription of a gene is controlled by regulatory proteins that bind to *transcription factor binding sites* (TFBSs) on DNA. The TFBSs are located mostly in non-coding regions preceding the genes. Discovering these motifs is rather difficult because of their very low information content. The reference databases JASPAR [1] and TRANSFAC [2] respectively contain 123 and 856 matrices of TFBSs. New sequencing technologies enable large-scale mapping of DNA-protein interactions : those "ChIP-Seq" methodologies produce new collections of sequences and matrices [3], [4].

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 $M(p, x)$ 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 $u = u_1 u_2 \dots u_p$ of Σ^m:

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

Let α be a score threshold. We say that M has an *occurrence* in a text T at position k if $\mathtt{Score}_M(T_k \dots T_{k+m-1}) \geq \alpha$.

The most recurrent task is to predict binding sites in a large DNA sequence, that is to look for occurrences of a PWM given a text and a score threshold. This basic task is often involved into a more general analysis pipeline. After the occurrences have been found, a filter is applied in order to refine the results. Tools which compute occurrences often output the statistical significance, called the *P-value*, of each occurrence. The P-value $\mathtt{Pv}_M(s)$ is the 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. If the background model is chosen with identically and independently distributed character symbols, the P-value is simply

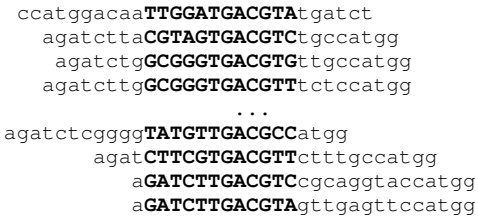
$$\mathtt{Pv}_M(s) = \frac{|\{u \in \Sigma^m \,|\, \mathtt{Score}_M(u) \geq s\}|}{|\Sigma|^m}$$

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

To decide if a PWM occurs at a position in a text, the score threshold has to be chosen. 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 [5]. Then the score threshold is computed for a given matrix M: the goal is to find the score α such that $\mathtt{Pv}_M(\alpha) = p$. The two problems of computing the P-value for a score and of computing the score from a given P-value are NP-hard [6], [7].

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 [8]–[10]. A better method is to consider that two matrices are similar if their occurrences are almost the same [11]. Being able to compare matrices serve 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 [9]–[11]. Secondly, it can be used to improve the SCAN problem (see page 3).

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



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

A	[-4.85826	-0.06247	-4.85826	-0.46381	···]
C	[0.37462	0.03957	-0.46793	-0.46793	···]
G	[0.03957	-0.18232	0.22107	0.82531	···]
T	[0.22314	0.22314	0.78009	-4.85826	···]

Figure 1. A Position Weight Matrix (PWM) modeling the CREB1 transcription factor binding site (from the JASPAR database). The coefficients are log-odds ratios of letter frequencies: they indicate affinities between letters and positions at the binding site.

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. We think that GPUs are a first step toward new massively many-cores architectures: we will discuss how our algorithms can be applied to other parallelization techniques.

General-purpose computation on GPU. Everyone can have some teraflops of cheap computing power with the recent Graphics Processing Units (GPUs). GPUs were used in bioinformatics since 2005 for phylogenetic studies [12], then for multiple sequence alignment based on an optimized Smith-Waterman implementation [13]. The CUDA libraries, first released in 2007 [14], have deeply simplified the development on GPUs. Recent papers provide speedups on bioinformatics applications involving suffix trees [15] or again Smith-Waterman comparisons [16].

The current Nvidia architectures [14] offer two level 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.

Contents. We investigate the four following problems in terms of parallel programming, especially on GPU architectures.

- SCAN. Given a matrix M with a score threshold α and a text T, find all the occurrences of M in T.
- SCORETOPVALUE. Given a matrix M and a score s, compute the P-value $\texttt{Pv}_M(s)$.
- PVALUETOSCORE. Given a matrix M and a P-value p, compute the score s such that $\texttt{Pv}_M(s) = p$.
- 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'.

Section 2 addresses the SCAN problem, for which a simple parallelization is very efficient. Section 3 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 more efficient than the existing algorithms (both in speed and in precision) and suitable for parallelization. Section 4 reports our prototype implementation of the three first algorithms with the CUDA libraries [14], and discuss our solution compared to other parallelizing techniques.

2. Looking for occurrences of a matrix

Finding 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.

Optimizations. In 2000, [17] 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, [18] 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, the table is organized such that scores for the set of matrices are in the same memory location, thus avoiding memory latencies. On a scan involving a large set of matrices, a practical $8\times$ speedup is obtained 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 occurences of one representative matrix [9], [18]. Occurrences of the other matrices are then computed only for positions where the representative matrix occurred.

Preprocessing and indexation. Ideas can also be borrowed from the algorithms searching a pattern in a text, as with the classical Aho-Corasick [19], Knuth-Morris-Pratt (KMP) [20], or Boyer-Moore [21] 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, [22] obtains a $2\times$ – $3\times$ improvement on the LSA method. With Aho-Corasick, [23] obtains a $2\times$ – $5\times$ speedup. Nevertheless, the size of the automaton is such that it does not fit in memory for matrices longer than 15.

Instead of indexing the matrices, another way to speedup the SCAN problem is to preprocess the text: [24] uses suffix trees (speedup $2\times$ – $5\times$), [25] uses suffix arrays and [26] compress the text (speedup $2\times$ – $5\times$). Some parallel implementations of suffix trees have been reported [15], 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 pattern matching with errors. 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 [27].

A simple parallel implementation. The parallelization of the SCAN problem can be done easily by splitting the different positions of the text across several threads (Figure 2). At a given time, each thread computes the score of one word.

The Slices Strategy could bring a very small improvement. On the contrary, the LSA as well as 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 different words will diverge most of the time, thus providing a very bad parallel performance.

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

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 [5]–[7], [28], [29]. 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:

$$\mathtt{PV}_M(s') = \sum_{s \geq s'} Q_M(s)$$

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. 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 [5]:

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

If the matrix has non-negative integer coefficient values, then S, the number of possible different scores, is bounded by $\sum_{i=1}^{m} \max M[i]$. It follows that known algorithms are pseudo-polynomial. 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 [7], [11], [25] 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 [7]. As an example, the implementation of MOSTA [11] 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 $m\varepsilon$.

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 3):

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

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

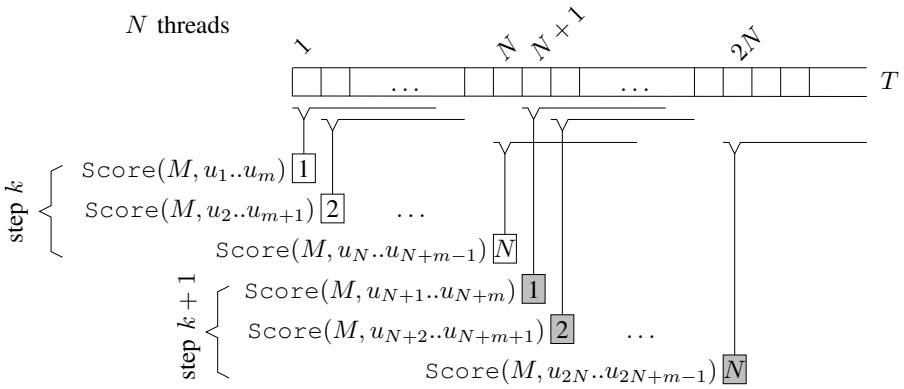


Figure 2. Parallel GPU SCAN. N threads compute the scores of words from position 1 to N, then all threads are shifted by N characters onto the sequence.

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 $\underline{\mathcal{S}}_{M_i}$,

$$Q_{M_i}[s] = \sum_{\substack{s' \in \mathcal{S}_{M_i} \\ \underline{s'} = s}} Q_{M_i}(s') \qquad (1)$$

  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 3). 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 [30], 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.

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, $\mathtt{Pv}_M[s]$,

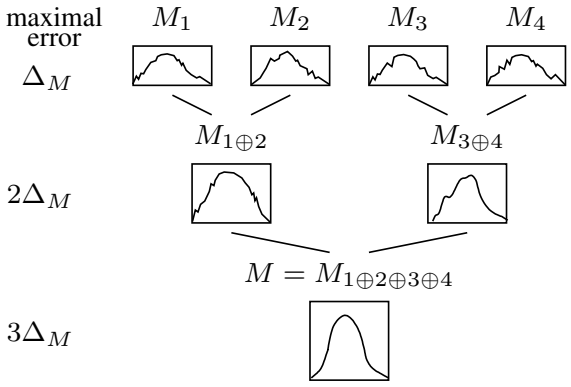


Figure 3. Algorithm BUCKETSCOREDISTRIBUTION

defined for a score s in $\underline{\mathcal{S}}_M$ by

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

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

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

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 (1) 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 $\Delta_{M_{i\oplus j}} = \Delta_{M_i} +$

Δ_{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 4). For the PVALUETOSCORE and the SCORETOPVALUE problems, the error on the P-value is thus

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

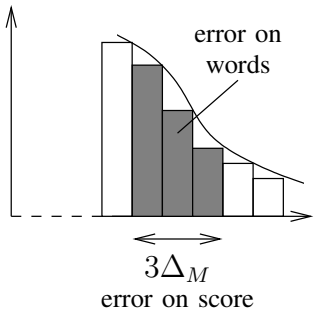


Figure 4. Error on P-value when computing BUCKETSCOREDISTRIBUTION

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 5, 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 5, 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.

Similarity between two matrices. The computation of similarity between two matrices is very similar. Given two matrices[1] 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:

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

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 [18]:

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

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

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

4. Results and discussion

Testing environment. We benchmarked the three SCAN, SCORETOPVALUE and PVALUETOSCORE algorithms with the CUDA 2.0 libraries from Nvidia [14]. The sources or our implementation are available at `http://bioinfo.lifl.fr/cudapwm`. Two GPU were tested: the Nvidia GeForce 8800 (16×8 cores, 1.3 GHz, 768 MB RAM), and the Nvidia GTX 280 (30×8 cores, 1.3 GHz, 1 GB RAM).

The host system and the CPU benchmarks were produced on an Intel Core 2 Duo 6600 (2.40 GHz) with 3 GB RAM and 4 MB cache (the CPU has two 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 8) and on random data (Figures 6 and 7). We found no significant difference in terms of speedups between those datasets.

4.1. Algorithm SCAN

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 2). Figure 6 and Table 1 detail the results. Using the GPU implies an overhead of 0.27s on a 225 Mbp chromosome (Table 1). 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

1. 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.

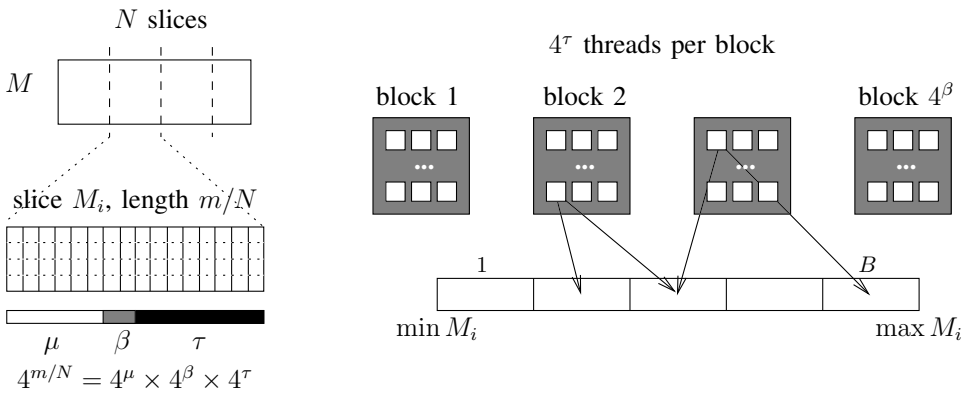


Figure 5. Parallel GPU BUCKETSCOREDISTRIBUTION. Each thread enumerates 4^μ words.

a collection of matrices is scanned, as there is in this case only one transfer of the sequence.

The Slices Strategy does not bring here a significant improvement. 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.

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
JASPAR collection	1304	0.27	36.60	36.87

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.

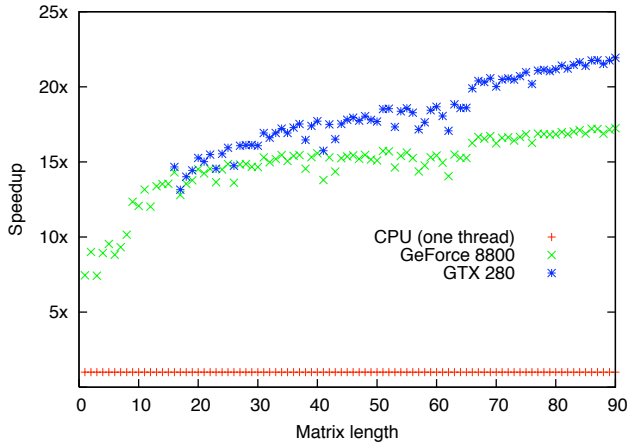


Figure 6. Results for SCAN (random matrices, random sequence of length $500 \cdot 10^6$).

Figures 6 details the speedup of the GPU implementations compared to a 1-thread CPU implementation. The best speedups, $16\times$ on GeForce 8800 and $21\times$ on GTX 280, are obtained starting from $30 \cdot 10^9$ positions computed. Those speedups should be compared to the $2\times$ to $8\times$ speedups of the methods cited on page 3, and to a maximum $8\times$ speedup using 128-bit SIMD instructions using a 16-bit precision.

4.2. 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 7 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 280 gives a 80% increased speedup on GeForce 8800. For $\ell = 65$, the CPU takes 1426 seconds: the speedup is $3.6\times$ for the GeForce 8800 and $9.8\times$ for the GTX 280.

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, the atomic instructions in shared memory give an additional $8\times$ speedup compared to the serialized incrementation. For $\ell = 65$, the total speedup is thus $77\times$.

Precision. Figure 8 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

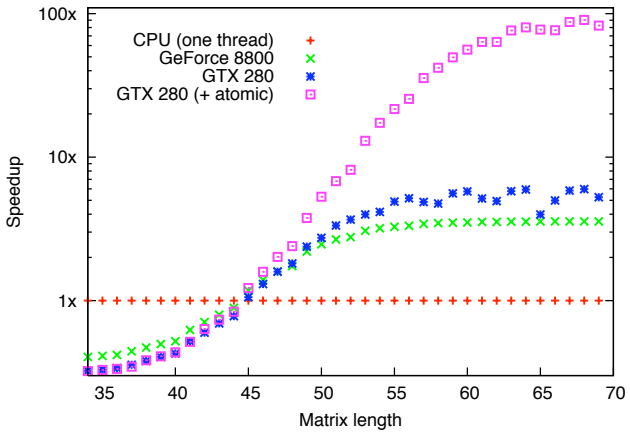


Figure 7. Results for PVALUETOSCORE, with $N = 4$ slices on random matrices (other N give similar speedups).

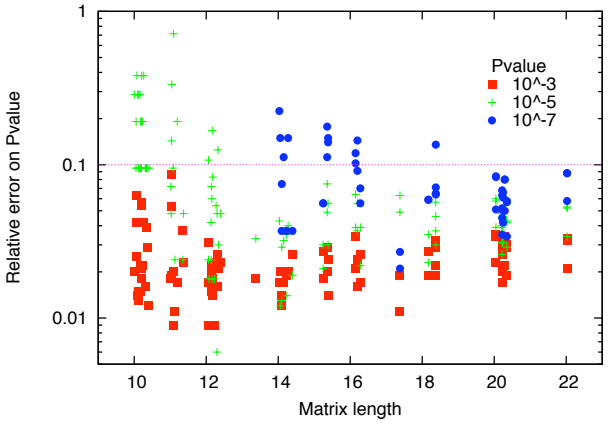


Figure 8. Relative error on P-value when computing PVALUETOSCORE with $N \in [2, 4]$ for all 101 matrices of JASPAR of length ≥ 10.

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.

Other parallelization techniques. On any multi-cores 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 5). Nevertheless, at 16-bit precision, the maximum theoretical speedup using SSE 128-bit SIMD extensions is $8\times$, far beyond the $77\times$ speedup achieved on GTX 280.

5. Conclusion and perspectives

We proposed a parallel SCAN of a Position Weight Matrix (PWM) 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 of the GTX 280.

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-cores architectures, in particular through the new OpenCL standard [31].

Acknowledgements

This research was carried out with a grant "Action Incitative LIFL" and through the "NVIDIA Professor Parternship" program. We thank Jérémie Allard (INRIA Lille) for giving us access to his GTX 280 card, and anonymous reviewers for comments on a previous version on the manuscript.

References

[1] A. Sandelin and al., "JASPAR: an open-access database for eukaryotic transcription factor binding profiles," *Nucleic Acids Research*, vol. 32, pp. D91–D94, 2004.

[2] E. Wingender and al., "TRANSFAC: an integrated system for gene expression regulation," *Nucleic Acids Research*, vol. 28, no. 1, pp. 316–319, 2000.

[3] J. Shendure and H. Ji, "Next-generation DNA sequencing," *Nat. Biotech*, vol. 26, no. 10, pp. 1135–1145, 2008.

[4] G. Robertson and al., "Genome-wide profiles of STAT1 DNA association using chromatin immunoprecipitation and massively parallel sequencing," *Nat. Meth.*, vol. 4, no. 8, pp. 651–657, 2007.

[5] J. M. Claverie and S. Audic, "The statistical significance of nucleotide position-weight matrix matches," *CABIOS*, vol. 12, no. 5, pp. 431–9, 1996.

[6] J. Zhang, B. Jiang, M. Li, J. Tromp, X. Zhang, and M. Zhang, "Computing exact p-values for DNA motifs," *Bioinformatics*, vol. 23, no. 5, pp. 531–537, 2007.

[7] H. Touzet and J.-S. Varré, "Efficient and accurate p-value computation for position weight matrices," *Algorithms for Molecular Biology*, vol. 2, no. 1, 2007.

[8] D. E. Schones, P. Sumazin, and M. Q. Zhang, "Similarity of position frequency matrices for transcription factor binding sites," *Bioinformatics*, vol. 21, no. 3, pp. 307–313, 2005.

[9] S. M. Kielbasa, D. Gonze, and H. Herzel, "Measuring similarities between transcription factor binding sites," *BMC Bioinformatics*, vol. 6, no. 237, pp. 1–11, 2005.

[10] S. Gupta, J. A. Stamatoyannopoulos, T. L. Bailey, and W. S. Noble, "Quantifying similarity between motifs," *Genome Biology*, vol. 8, no. 2, 2007.

[11] U. J. Pape, S. Rahmann, and M. Vingron, "Natural similarity measures between position frequency matrices with an application to clustering." *Bioinformatics*, vol. 24, no. 3, 2008.

[12] M. Charalambous, P. Trancoso, and A. Stamatakis, "Initial experiences porting a bioinformatics application to a graphics processor," *Adv. in Informatics*, pp. 415–425, 2005.

[13] W. Liu, B. Schmidt, G. Voss, and 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.

[14] "Nvidia CUDA programming guide 2.0," 2008.

[15] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units." *BMC Bioinformatics*, vol. 8, p. 474, 2007.

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

[17] T. D. Wu, C. G. Nevill-Manning, and D. L. Brutlag, "Fast probabilistic analysis of sequence function using scoring matrices." *Bioinformatics*, vol. 16, no. 3, pp. 233–244, 2000.

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

[19] A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," *Comm. ACM*, vol. 18, pp. 333–340, 1975.

[20] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comp.*, vol. 6, no. 1, pp. 323–360, 1977.

[21] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Comm. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

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

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

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

[25] M. Beckstette, R. Homann, R. Giegerich, and S. Kurtz, "Fast index based algorithms and software for matching position specific scoring matrices," *BMC Bioinformatics*, vol. 7, 2006.

[26] V. Freschi and A. Bogliolo, "Using sequence compression to speedup probabilistic profile matching," *Bioinformatics*, vol. 21, no. 10, pp. 2225–9, 2005.

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

[28] R. Staden, "Methods for calculating the probabilities of finding patterns in sequences," *CABIOS*, vol. 5, no. 2, pp. 89–96, 1989.

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

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

[31] "The Khronos Group, OpenCL 1.0 specification," 2008.