




A BLAS-Based Algorithm for Finding Position Weight Matrix Occurrences in DNA Sequences on CPUs and GPUs

Jan Fostier^(✉) 

IDLab, Department of Information Technology,
Ghent University - imec, Ghent, Belgium
jan.fostier@ugent.be
<http://idlab.ugent.be>

Abstract. Finding all matches of a set of position weight matrices (PWMs) in large DNA sequences is a compute-intensive task. We propose a light-weight algorithm inspired by high performance computing techniques in which the problem of finding PWM occurrences is expressed in terms of matrix-matrix products which can be performed efficiently by highly optimized BLAS library implementations. The algorithm is easy to parallelize and implement on CPUs and GPUs. It is competitive on CPUs with state-of-the-art software for matching PWMs in terms of runtime while requiring far less memory. For example, both strands of the entire human genome can be scanned for 1404 PWMs in the JASPAR database in 41 min with a p -value of 10^{-4} using a 24-core machine. On a dual GPU system, the same task can be performed in under 5 min.

Keywords: Position weight matrix (PWM)
High performance computing (HPC)
Basic linear algebra subprograms (BLAS)
Graphics processing units (GPUS)

1 Introduction

Short biologically relevant patterns such as transcription factor binding sites are often represented using a position weight matrix (PWM), also referred to as a position-specific scoring matrix (PSSM) [1]. In contrast to consensus patterns, a PWM can model variability at each position in the pattern. A PWM representing a pattern of length m is a $4 \times m$ matrix where each matrix element $\text{PWM}(i, j)$ represents the log-likelihood of observing character i ($0 = \text{'A'}$; $1 = \text{'C'}$; $2 = \text{'G'}$; $3 = \text{'T'}$) at position j , taking into account the nucleotide composition of the background sequences. Given a sequence of length m , the PWM score of that sequence can be computed by summing over the PWM values that correspond to each nucleotide at each position in the sequence. Higher scores indicate a better correspondence to the pattern represented by the PWM.

Given an input sequence of length n and a PWM of length m , the PWM matching problem involves the identification of all matches of the PWM, i.e., subsequences for which the PWM score exceeds a user-defined threshold. A brute-force approach simply involves the computation of the PWM score at all positions in the input sequence and hence has a time complexity of $O(nm)$. More complex algorithms for PWM matching build upon ideas that were initially developed for exact pattern matching and rely on the preprocessing of the input sequence and/or the preprocessing of the search matrix. In [2], a suffix tree is constructed from the input sequence and PWM matches are found using a depth-first traversal of the tree up to depth m . By using a lookahead scoring technique, subtrees that contain no PWM matches can be detected and discarded from the search procedure. A similar methodology has been implemented in PoSSuM [3], where an enhanced suffix array is used as a more memory-friendly alternative to suffix trees. Both methods however have the disadvantage of requiring $O(n)$ memory to build and store the index structure. In [4], the Morris-Pratt and Knuth-Morris-Pratt algorithms are extended to PWM matching. Similarly, in [5], the Aho-Corasick, filtration and super-alphabet techniques developed for exact string matching are generalized to PWM matching and further extended to the case where matches of multiple PWMs are searched for [6]. These algorithms are implemented in the MOODS software package [7]. Finally, in [8], some of these algorithms are implemented on graphics processing unit (GPU) architectures.

Compared with the naive brute-force algorithm, these more complex PWM matching algorithms reduce the runtime by eliminating parts of the search space that are guaranteed not to contain matches. As such, their runtime is dependent on the PWM threshold that is used. The higher this threshold is taken, the more the PWM matching problem approaches that of exact pattern matching with its $O(n + m)$ time complexity. Because for most practical problems, m takes values between 5 and 15 while n is very large, they yield a speedup of approximately one order of magnitude over the $O(nm)$ brute-force algorithm.

In this contribution, we describe an orthogonal strategy to accelerate the brute-force algorithm. Our approach does not reduce the search space but rather improves the speed at which the brute-force algorithm can be evaluated. This is done by expressing the PWM matching problem in terms of matrix-matrix products. It is well known that matrix-matrix multiplications can be evaluated very efficiently on modern, cache-based CPUs using highly optimized Basic Linear Algebra Subroutines (BLAS) library implementations [9]. These BLAS implementations leverage SIMD (single instruction multiple data) operations and maximally exploit spatial and temporal locality of reference, thus ensuring that most data accesses are satisfied from cache memory. As such, matrix-matrix products are among a select class of algorithms that can be evaluated with a performance that approaches the theoretical peak performance of a CPU. Optimized BLAS library implementations are provided by all major CPU vendors. Alternatively, open-source implementations such as ATLAS [10] or Goto-BLAS [11] can be considered. We found that the BLAS-based approach yields a $5\times$ to $6.4\times$ speedup over a naive implementation of the brute-force algorithm.

Additionally, the proposed BLAS-based algorithm has minimal memory requirements whereas more complex algorithms may require tens of GBytes of memory for large problem sizes. Finally, we also present an implementation of the BLAS-based algorithm that leverages the cuBLAS library to perform the matrix-matrix multiplications on graphics processing units (GPUs). We demonstrate that on a dual-GPU system, this yields an additional $10\times$ speedup compared to using a 24-core CPU system. Using this GPU system, we report speedups of up to $43\times$ compared with the state-of-the-art MOODS software package.

An open-source implementation of the algorithm is available on <https://github.com/biointec/blstools>.

2 Algorithm Description

We consider the PWM matching problem in the general case where we have multiple PWMs over a DNA alphabet. The goal is to recast the naive algorithm into an algorithm that relies on matrix-matrix multiplications. In essence, this procedure involves three matrices:

- A pattern matrix P that contains all of the PWMs.
- A sequence matrix S that contains some sequence content.
- A result matrix R that is computed as $R = P * \text{sub}(S)$ and that contains the PWM scores of all PWMs at *some* positions in the sequence. The routine $\text{sub}(\cdot)$ denotes that a submatrix of S is used.

Below, we describe each matrix in detail. Figure 1 provides an overview of the algorithm.

2.1 Pattern Matrix P

The pattern matrix P is built once and remains fixed during the course of the algorithm. Matrix P has dimensions $c \times 4m$ where c denotes the total number of PWMs and $m = \max_i(m^i)$ refers to the maximum PWM length where m^i denotes the length of PWM ^{i} . Every row of P corresponds to a single PWM. The values in a row of P are obtained by unrolling the values of the corresponding PWM. For PWMs shorter than m characters, trailing zeros are appended to the corresponding row in P . Formally:

$$P(i, j) = \begin{cases} \text{PWM}^i(j \bmod 4, \lfloor j/4 \rfloor) & 0 \leq j < 4m^i \\ 0 & j \geq 4m^i \end{cases} \quad (1)$$

for all $0 \leq i < c$.

In case PWM occurrences on both strands of the input sequence(s) need to be identified, an additional c rows can be added to matrix P that represent the reverse-complement of each PWM.

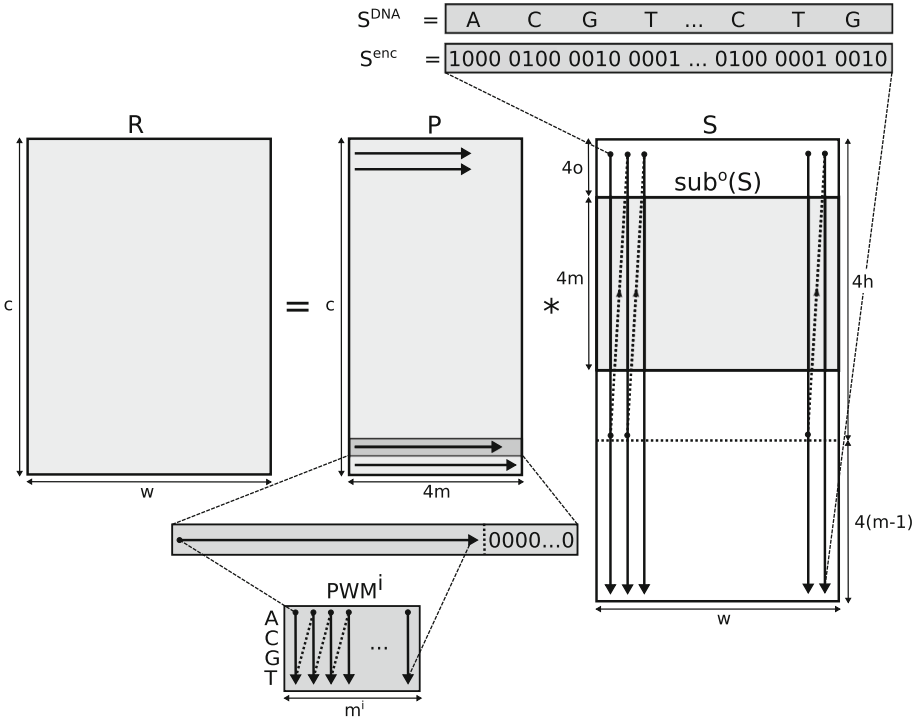


Fig. 1. The result matrix R is computed as the matrix-matrix product of pattern matrix P and a submatrix of sequence matrix S . Each row in P represents a single PWM. Matrix S represents (part of) the input sequence. Each element in R contains a PWM score at some position in the input sequence.

2.2 Sequence Matrix S

The sequence matrix S has dimensions $4(h + m - 1) \times w$ where h and w can be arbitrarily chosen ≥ 1 and where m again represents the maximum PWM length. The matrix S is used to encode (part of) the input sequence(s) S^{DNA} of exactly $hw + m - 1$ nucleotides. First, the string S^{DNA} is converted into an array S^{enc} of $4(hw + m - 1)$ zeros and ones by simply replacing character A by 1000; C by 0100; G by 0010; and T by 0001. Formally:

$$S^{\text{enc}}(i) = \begin{cases} 1 & S^{\text{DNA}}(\lfloor i/4 \rfloor) = A \wedge i \bmod 4 = 0 \\ 1 & S^{\text{DNA}}(\lfloor i/4 \rfloor) = C \wedge i \bmod 4 = 1 \\ 1 & S^{\text{DNA}}(\lfloor i/4 \rfloor) = G \wedge i \bmod 4 = 2 \\ 1 & S^{\text{DNA}}(\lfloor i/4 \rfloor) = T \wedge i \bmod 4 = 3 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

for all $0 \leq i < 4(hw + m - 1)$. The matrix S is constructed from this temporary array as follows:

$$S(i, j) = S^{\text{enc}}(4hj + i) \tag{3}$$

for all $0 \leq i < 4(h + m - 1)$ and $0 \leq j < w$.

Every column in S contains a contiguous subarray of S^{enc} and thus encodes a substring of S^{DNA} . The bottom $4(m - 1)$ elements of column j are identical to the top $4(m - 1)$ elements of column $j + 1$. In other words, subsequent columns of S encode overlapping substrings of S^{DNA} with an overlap of $m - 1$ characters.

2.3 Result Matrix R

The result matrix R had dimensions $c \times w$ and is computed as the matrix-matrix product of matrix P with a submatrix of S . Given an offset o with $0 \leq o < h$, R^o is computed as follows:

$$R^o = P * S([4o, 4(o + m)[, :]) \tag{4}$$

where the notation $S([4o, 4(o + m)[, :])$ refers to the $4m \times w$ submatrix of S where the first row in the submatrix corresponds to row with index $4o$ in S . Every element in R^o is thus computed as the dot product of a row in P and (part of) a column in S . The elements of S (zeros and ones) are multiplied with the elements of the PWM and thus generate the terms that, when added, correspond to the PWM score. As such, element $R^o(i, j)$ contains the score for PWM ^{i} at position $(hj + o)$ in S^{DNA} .

Algorithm 1 then provides a complete description of the workflow. In the outer for-loop, a portion of the input sequence(s) of length $hw + m - 1$ is read into S^{DNA} . In the inner for-loop, the PWM scores are exhaustively computed for all c PWMs at the hw first positions of S^{DNA} . Therefore, the S^{DNA} strings at consecutive outer for-loop iterations overlap by $m - 1$ nucleotides.

Algorithm 1. BLAS-based PWM occurrence detection

Input: Sequence S^{input} ▷ DNA sequence
Input: PWMs = {PWM ^{i} } ▷ Set of PWMs
Input: thresholds = {threshold ^{i} } ▷ Set of thresholds

- 1: $P \leftarrow \text{createPatternMatrix}(\text{PWMs})$
- 2: **for** pos = 0 **to** length(S^{input}) - 1 **step** hw **do**
- 3: $S^{\text{DNA}} \leftarrow S^{\text{input}}[\text{pos}, \text{pos} + \text{hw} + \text{m} - 1[$
- 4: $S^{\text{enc}} \leftarrow \text{encodeString}(S^{\text{DNA}})$
- 5: $S \leftarrow \text{createSequenceMatrix}(S^{\text{enc}})$
- 6: **for** o = 0 **to** h - 1 **step** 1 **do**
- 7: $R^o \leftarrow P * S([4o, 4(o + m)[, :])$
- 8: reportOccurrences($R^o, o, \text{thresholds}$)
- 9: **end for**
- 10: **end for**

Note that in case the input data consists of multiple DNA sequences, these sequences can be concatenated when generating S^{DNA} . With minimal extra bookkeeping, one can prevent the reporting of occurrences that span adjacent DNA sequences.

2.4 Implementation Details

The algorithm is implemented in C++. Multithreading support is added through C++ 11 threads by parallelizing the outer for-loop in Algorithm 1. The BLAS `sgemm` routine [9] was used to perform the matrix-matrix multiplications using single-precision computations.

Recall that PWMs with a length shorter than the maximum PWM length m are represented in the pattern matrix P by adding trailing zeros to the corresponding row. In case many PWMs have a length that is substantially shorter than m , a large fraction of P consists of zero elements. In turn, this creates overhead during the matrix-matrix product when computing the result matrix R due to the including of many terms with value zero. This overhead can easily be reduced by representing the PWMs in P in a sorted manner (sorted according to length). The matrix-matrix product $R^o = P * \text{sub}^o(S)$ can then be computed as a number of smaller matrix-matrix products as follows:

$$R([c_i, c_{i+1}[, :]) = P([c_i, c_{i+1}[, [0, 4m_i]) * S([4o, 4(o + m_i)[, :]) \tag{5}$$

where the interval $[c_i, c_{i+1}[$ corresponds to a subset of the rows in R and P and where m_i denotes the maximum PWM length in that range. When $m_i < m$ overhead is reduced. For the JASPAR dataset (see description below) the idea is clarified in Fig. 2. The pattern matrix P represents 1404 PWMs with lengths

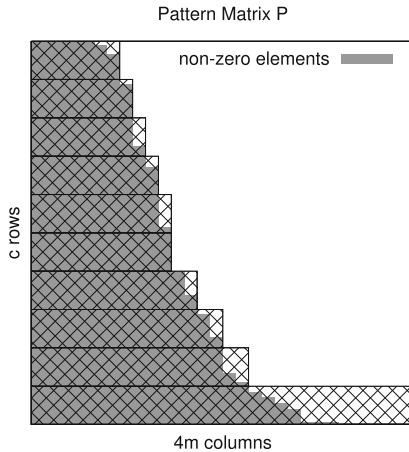


Fig. 2. Example of a pattern matrix P containing 1404 JASPAR PWMs where many rows contain trailing zeros because of differences in length of the corresponding PWMs. Matrix P can be subdivided in a number of smaller submatrices P^i (shaded areas) that each contain less zero fill.

between 5 and 30 and thus exhibits substantial zero fill. By subdividing P in 10 submatrices each representing 140 or 141 PWMs, the number of elements of P used in the matrix-matrix multiplication is more than halved. Note that the submatrices $P([c_i : c_{i+1}], [0 : 4m_i])$ should not become too thin such that the evaluation of (5) still corresponds to a meaningful matrix-matrix product. In other words, one could avoid zero fill altogether by subdividing P in c different vectors, however, this would result in loss of temporal locality of cache and hence, a considerable loss of speed.

For the same performance reasons, parameters h and w that govern the dimensions of matrix S should not be chosen too small. In our implementation, we set $h = 250$ and $w = 1000$ such that matrix S corresponds to a $1000 + 4(m - 1) \times 1000$ matrix.

Finally, note that BLAS routines have full support to specify submatrix ranges without any need to explicitly copy these submatrices onto separate data structures.

2.5 GPU Version

Through the use of the cuBLAS library [12], it is possible to execute the matrix-matrix multiplication on a graphics processing unit (GPU). The pattern matrix P is copied to the GPU memory only once, while a new sequence matrix S is copied during each outer for-loop iteration in Algorithm 1. To avoid copying the entire result matrix R^o from GPU memory to system RAM after each matrix-matrix product during each inner for-loop iteration, a kernel was developed in the CUDA language to report only the matrix indices (i, j) for which $R^o(i, j)$ exceeds the threshold score for PWM^i (a task known as stream compaction). Only those indices are copied from GPU to system RAM, thus minimizing data movements between GPU and host memory. Occurrences are written to disk by the CPU. Note that the programming effort to port the BLAS-based algorithm from CPU to GPU is minimal as most tasks are handled by CUDA library calls (e.g. copying data between CPU and GPU, calling `cublasSgemm, ...`). The only exception is the stream compaction kernel itself that consists of 7 lines of CUDA code.

3 Benchmark Results and Discussion

The performance of the BLAS-based algorithm was benchmarked against (i) a naive, scalar algorithm and (ii) the MOODS software package [7]. To ensure a fair comparison, the naive algorithm has the same code quality standards as the BLAS-based algorithm, the only difference being that three nested for-loops are used to scan for the occurrences: one for-loop over the input sequence, a second one over the different PWMs and a third for-loop to compute the PWM score.

The C++ source code was compiled against the Intel Math Kernel Library (MKL) version 2017.1.132 which implements optimized BLAS routines for Intel CPUs. In all cases, multi-threading *within* the MKL was disabled. In other words,

individual calls to `sgemm` were always executed in a single-threaded manner but multiple calls to `sgemm` are issued by different threads concurrently. The CUDA code was compiled with the `nvcc` compiler and linked against `cuBLAS` from CUDA SDK version 8.0.

From the JASPAR database [13], 1404 position frequency matrices were downloaded. As a sequence dataset, the human genome reference sequence (HG38) was used from the GATK Resource Bundle. Part of the tests were run only on chromosome 1 (230 Mbp). We scanned for PWM occurrences on both strands of the DNA sequences by also including the reverse complements of the PWM matrices. Thus effectively, 2808 PWM matrices were used in total.

The benchmarks were run on a node containing two 12-core Intel E5-2680v3 CPUs (24 CPU cores in total) running at 2.5 GHz with 64 GByte of RAM. The CPU is of the Haswell-EP architecture and disposes of AVX-256 instructions that can deal with 8 single precision floating point numbers in a single instruction. For configurations with p -value = 10^{-4} , MOODS required >64 GByte of RAM. Those runs were performed on a single core of a system containing two 10-core Intel E5-2660v3 CPUs running at 2.6 GHz with 128 GByte of RAM. When performing the benchmarks with fewer threads than CPU cores, the remaining CPU cores were idle. The GPU runs were performed on a system with a dual nVidia 1080 Ti GPU configuration. Runtime (wall clock time) and peak resident memory use were measured using the Linux `/usr/bin/time -v` tool.

Table 1 shows the runtime, memory use and parallel efficiency for the different approaches when considering chromosome 1 of the human genome as input

Table 1. Benchmark results of the naive algorithm, the MOODS algorithm and the proposed BLAS-based algorithm (on CPU and GPU). In all cases, the occurrences of 1404 JASPAR PWMs were searched on both strands of human chromosome 1 for two different PWM thresholds (p -value = 10^{-5} and 10^{-4}).

No. cores	p -value 10^{-5}				p -value 10^{-4}			
	Wall clock time	Parallel speedup	Parallel efficiency	Memory (GByte)	Wall clock time	Parallel speedup	Parallel efficiency	Memory (GByte)
<i>Naive algorithm (24-core CPU system)</i>								
1	21 999 s	-	-	0.01	22 024 s	-	-	0.01
4	5 495 s	4.0	100%	0.01	5 506 s	4.00	100%	0.01
8	2 752 s	7.99	100%	0.01	2 755 s	7.99	100%	0.01
24	926 s	23.76	99%	0.01	921 s	23.91	100%	0.01
<i>MOODS (CPU system)</i>								
1	402 s	-	-	19.02	1 028 s	-	-	64.89
<i>BLAS-based algorithm (24-core CPU system)</i>								
1	3 441 s	-	-	0.04	3 582 s	-	-	0.04
4	871 s	3.95	99%	0.11	889 s	4.03	101%	0.12
8	479 s	7.18	90%	0.20	473 s	7.57	95%	0.22
24	179 s	19.22	80%	0.59	183 s	19.57	82%	0.66
<i>BLAS-based algorithm (dual GPU system)</i>								
-	24 s	-	-	0.49	25 s	-	-	0.58

dataset. Even though it has perfect scaling behavior with respect to the number of CPU cores used and negligible memory use, the naive algorithm is also the slowest. MOODS has very good performance in terms of runtime, especially when taking into account that the software is single-threaded. However, it has much higher memory requirements, over 64 GByte of RAM. Additionally, both the runtime and memory use depend on the PWM thresholds that are used: more relaxed thresholds (i.e., higher p -values) result in additional resource requirements. The BLAS-based algorithm also shows very good multi-threading scaling behavior and outperforms the naive algorithm by a factor between $5\times$ and $6.4\times$ while still maintaining very low memory requirements. Compared with MOODS, the BLAS-based algorithm is slower when using only a single thread but outperforms the latter when using multiple cores. Additionally, like the naive algorithm, its resource requirements do not depend on the p -value that is used. Finally, the BLAS-based algorithm attains maximal performance when executed on the GPU system.

Table 2 shows runtime and memory use when considering the entire human genome as input dataset. Due to its dependence on the p -value, MOODS has runtimes ranging from 43 min to 3.5 h and memory requirement ranging from 20 GByte to 103 GByte. In contrast, the BLAS-based algorithm has a runtime that is nearly constant and requires very little memory. On the GPU system, the BLAS-based algorithm shows speedups of $9.5\times$ and $43\times$ over MOODS.

Finding the occurrences of a PWM in a sequence can be seen as an imprecise string matching problem. When only the very best PWM matches are needed (by using a low p -value and hence, a high PWM score threshold), the problem eventually approaches that of exact string matching for which very efficient algorithms have been designed by either indexing the sequence or preprocessing the patterns. These algorithms yield $O(n + m)$ time complexity instead of the brute-force $O(nm)$. Nevertheless, for less strict p -values, these algorithms

Table 2. Benchmark results of the MOODS algorithm and the proposed BLAS-based algorithm (on CPU and GPU). In all cases, the occurrences of 1404 JASPAR PWMs were searched on both strands of the entire human genome for three different PWM thresholds (p -value = 10^{-6} , 10^{-5} and 10^{-4}).

No. cores	p -value 10^{-6}		p -value 10^{-5}		p -value 10^{-4}	
	Wall clock time	Memory use (GB)	Wall clock time	Memory use (GB)	Wall clock time	Memory use (GB)
<i>MOODS (CPU system)</i>						
1	43 min 8 s	20.71	71 min 42 s	30.25	3 h 35 min 26 s	103.20
<i>BLAS-based algorithm (24-core CPU system)</i>						
24	36 min 39 s	0.61	37 min 29 s	0.78	40 min 50 s	3.49
<i>BLAS-based algorithm (dual GPU system)</i>						
-	4 min 29 s	0.51	4 min 33 s	0.73	4 min 57 s	3.86

perform considerably worse because they cannot a-priori eliminate large parts of the search space. Even though the proposed BLAS-based algorithm does not reduce the search space it has several advantages:

- The runtime is independent of the chosen p -value and hence of the number of occurrences that are found, at least for as long as writing the occurrences to disk does not become a bottleneck of the system.
- The memory use of the proposed algorithm is negligible and again independent of the chosen p -value. In our configuration, we effectively use only a few MBytes of RAM per thread. All matrices involved are thread-local and hence, the multi-threaded algorithm scales very well to a high number of CPU cores, even on non-uniform memory architectures (NUMA).
- As the vast majority of the compute time is spent inside the BLAS library the performance of the code is fully dependent on the quality of the BLAS implementation. As CPU vendors provide optimized BLAS libraries for their hardware optimal performance is guaranteed on all systems, including future ones. For example, AVX-512 instructions will be available on next generations of CPUs and will thus offer doubled performance compared to the AVX-256 system used in the benchmarks. Additionally, support for half-precision floating point computations is increasingly adopted might as also double throughput.
- Arguably, the implementation of the algorithm is very simple.
- The algorithm is easily portable to GPUs, through the use of the cuBLAS Libra that enables very high-performance matrix-matrix multiplications on GPUs. As the peak performance of modern GPUs exceeds that of CPUs one can observe very high performance on GPUs. The same argument holds for other co-processors/hardware accelerators.

4 Conclusion

We proposed a conceptually simple and easy to implement algorithm to identify position weight matrix matches in DNA sequences. The algorithm performs a brute-force evaluation of all PWM matrices at all possible starting positions in the DNA sequences, however, these evaluations are expressed entirely through matrix-matrix multiplications. On modern, cache-based CPUs that dispose of SIMD instructions, matrix-matrix products can be evaluated very efficiently through the use of highly optimized BLAS libraries. As a consequence, the BLAS-based algorithm outperforms the naive algorithm by a factor of 5 to 6.4. The runtime of the proposed algorithm is independent of the p -value and hence the PWM score threshold that is used and requires only very low amounts of memory. Additionally, the algorithm is trivial to parallelize and exhibits good scaling behavior. Compared with the state-of-the-art MOODS software package which implements more sophisticated online search algorithms that reduce the search space, the proposed BLAS-based algorithm is competitive in terms of runtime while requiring less memory. On GPU systems, the BLAS-based algorithm attains maximal performance and outperforms CPU-based algorithms by a large factor.

Acknowledgments. The computational resources (Stevin Supercomputer Infrastructure) and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by Ghent University, FWO and the Flemish Government – department EWI.

References

1. Stormo, G.D.: DNA binding sites: representation and discovery. *Bioinformatics* **16**(1), 16–23 (2000)
2. Dorohonceanu, B., Nevill-Manning, C.G.: Accelerating protein classification using suffix trees. In: *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*, 19–23 August 2000, La Jolla/San Diego, CA, USA, pp. 128–133 (2000)
3. Beckstette, M., Homann, R., Giegerich, R., Kurtz, S.: Fast index based algorithms and software for matching position specific scoring matrices. *BMC Bioinf.* **7**(1), 389+ (2006)
4. Liefoghe, A., Touzet, H., Varré, J.-S.: Self-overlapping occurrences and Knuth-Morris-Pratt algorithm for weighted matching. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) *LATA 2009*. LNCS, vol. 5457, pp. 481–492. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00982-2_41
5. Pizzi, C., Rastas, P., Ukkonen, E.: Fast search algorithms for position specific scoring matrices. In: Hochreiter, S., Wagner, R. (eds.) *BIRD 2007*. LNCS, vol. 4414, pp. 239–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71233-6_19
6. Pizzi, C., Rastas, P., Ukkonen, E.: Finding significant matches of position weight matrices in linear time. *IEEE/ACM Trans. Comput. Biol. Bioinf.* **8**(1), 69–79 (2011)
7. Korhonen, J., Martinmäki, P., Pizzi, C., Rastas, P., Ukkonen, E.: MOODS: fast search for position weight matrix matches in DNA sequences. *Bioinformatics* **25**(23), 3181–3182 (2009)
8. Giraud, M., Varré, J.S.: Parallel position weight matrices algorithms. *Parallel Comput.* **37**(8), 466–478 (2011)
9. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* **16**(1), 1–17 (1990)
10. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC 1998. IEEE Computer Society, Washington, DC, USA, pp. 1–27 (1998)
11. Goto, K., van de Geijn, R.A.: Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* **34**(3), 12:1–12:25 (2008)
12. Cook, S.: *CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs*, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2013)
13. Mathelier, A., Fornes, O., Arenillas, D.J., Chen, C.Y.Y., Denay, G., Lee, J., Shi, W., Shyr, C., Tan, G., Worsley-Hunt, R., Zhang, A.W., Parcy, F., Lenhard, B., Sandelin, A., Wasserman, W.W.: JASPAR 2016: a major expansion and update of the open-access database of transcription factor binding profiles. *Nucleic Acids Res.* **44**(D1), D110–D115 (2016)