GPU-MEME: Using Graphics Hardware to Accelerate Motif Finding in DNA Sequences

Chen Chen, Bertil Schmidt, Liu Weiguo, and Wolfgang Müller-Wittig

School of Computer Engineering, Nanyang Technological University, Singapore
`{cchen,asbschmidt,liuweiguo,askwmwittig}@ntu.edu.sg`

Abstract. Discovery of motifs that are repeated in groups of biological sequences is a major task in bioinformatics. Iterative methods such as expectation maximization (EM) are used as a common approach to find such patterns. However, corresponding algorithms are highly compute-intensive due to the small size and degenerate nature of biological motifs. Runtime requirements are likely to become even more severe due to the rapid growth of available gene transcription data. In this paper we present a novel approach to accelerate motif discovery based on commodity graphics hardware (GPUs). To derive an efficient mapping onto this type of architecture, we have formulated the compute-intensive parts of the popular MEME tool as streaming algorithms. Our experimental results show that a single GPU allows speedups of one order of magnitude with respect to the sequential MEME implementation. Furthermore, parallelization on a GPU-cluster even improves the speedup to two orders of magnitude.

1 Introduction

A major challenge in computational genomics nowadays is to find patterns (or *motifs*) in a set of sequences. In particular, discovering motifs that are crucial for the regulation of gene transcription in DNA (such as Transcription Factor Binding Sites) are of growing importance to biological research. With the production of vast quantities of data, genomic researchers want to perform this analysis on a larger scale, which in turn leads to massive compute requirements. In this paper we show how modern streaming architectures can be used to accelerate this highly compute-intensive task by one to two orders of magnitude.

Algorithmic approaches to motif discovery can be classified into two main categories: *iterative* and *combinatorial*. Iterative methods are based on local stochastic search techniques such as expectation maximization (EM) [1, 2] or Gibbs sampling [5], while combinatorial algorithms use deterministic methods like dictionary building [8] or word enumeration [11]. Iterative methods are often preferred since they are using PSSMs (*Position Specific Scoring Matrices*) instead of a simple Hamming distance to describe the matching between a motif instance and a sequence. Among the iterative approaches, MEME (*Multiple EM for Motif Elicitation*) [2, 3] is a popular and well established method. However, its complexity is $O(N^2 \cdot L^2)$, where N is the number of input sequence and L is the length of each sequence. Therefore, this approach is time consuming for applications involving large data sets such as whole-genome motif discovery. Corresponding runtime requirements are likely to become even more severe due to the rapid growth in the size of available genomic sequence and transcription data. An approach to get results in a shorter time is to use high performance computing. Previous approaches to accelerate the motif finding process are based on expensive compute clusters [3] and specialized hardware [9].

This paper presents a proof-of-concept parallelization of motif discovery with MEME on commodity graphics hardware (GPUs) to achieve high performance at low cost. Our software currently supports the OOPS (one occurrence per sequence) and ZOOPS (zero or one occurrence per sequence) search models for DNA sequences. Our future work includes integrating the more complex TCM (two-component mixture) model and making the software available for public use. We are also planning to port the presented GLSL code to the newly released CUDA programming interface for GPU programming, which was not was not available at the time of writing the GPU-MEME code. Our achieved speedups on an NVIDIA GeForce 8800 GTX compared to the sequential MEME implementation are between 9 (for small data sets) and 12 (for large data sets). The runtime on a single GPU also compares favourably to the MPI-based ParaMEME running on a cluster with 12 CPUs. Furthermore, we have combined the fine-grained GPU parallelization with a coarse-grained parallel approach. This hybrid approach improves the speedup on a cluster of six GPUs to over 60.

The rest of this paper is organized as follows. In Section 2, we provide necessary background on motif discovery and general-purpose computing on GPUs. Section 3 presents our parallel streaming algorithm for motif finding. Performance is evaluated in Section 4. Finally, Section 5 concludes the paper.

2 Background

2.1 Motif Discovery

Iterative methods like EM search for motifs by building statistical motif models. A motif model is typically represented by a matrix (θ). For a motif of width W and an alphabet $\Sigma = \{x_0,\ldots,x_{A-1}\}$ of size A the matrix θ is of size $A \times (W+1)$. The value at position (i,j), for $0 \le i \le A-1$, $0 \le j \le W$, of the matrix is defined as follows:

$$\theta_{i,j} = \begin{cases} \text{Probability of } x_i \text{ appearing at position } j \text{ of the motif} & \text{if } 1 \le j \le W \\ \text{Probability of } x_i \text{ appearing at positions outside the motif} & \text{if } j = 0 \end{cases}$$

The overall goal of the EM approach is to find a matrix with maximal posterior probability given a set of input sequences.

The outline of the MEME (Multiple EM for Motif Elicitation) [2] algorithm is shown in Figure 1. The search for a motif at each possible motif width W consists of two phases. Since EM is easily trapped in local minima, the first phase iterates over a large number of possible starting points to identify a good initial model $\theta^{(0)}$. In the second phase, the algorithm then performs the full EM algorithm until convergence using $\theta^{(0)}$. Profiling of the MEME algorithm (see Table 1) reveals that over 96% of the overall running time is usually spent on the first phase (called "*starting point search*"). We therefore describe the starting point search algorithm in more detail in the following.

```
procedure MEME(X:set of sequences)
  for pass = 1 to num_motifs do
    for W = W_min to W_max do
      for all starting points (i,j) in X do
        estimate score of the initial motif model which includes the
        W-length substring starting at position j in sequence i;
      end
      choose initial model $\theta^{(0)}$ from starting position with maximal
      estimated score;
      run EM to convergence starting with model $\theta^{(0)}$;
    end
    print converged model with highest likelihood;
    "erase" appearance of discovered shared motif in X;
  end
end
```

Fig. 1. Outline of the MEME algorithm

Given is the input dataset $X = \{S_1, S_2,\ldots, S_n\}$ consisting of n sequences over the alphabet $\Sigma = \{x_0,\ldots,x_{A-1}\}$ and the motif width W. Let sequence S_i be of length $L(i)$ for $1 \le i \le n$. Then, the total number of substrings of length W in X is $\left(\sum_{i=1}^{n} L(i)\right) - N \cdot (W-1) \cdot$ Let $S_{i,j}$ denote the substring of length W starting at position j in sequence S_i for all $1 \le i \le n$, $1 \le j \le L(i)$, The starting point search algorithm considers all these substrings as possible starting points using the three steps shown in Figure 2.

```
for each length-W substring $S_{i,j}$ in X do
  [Step 1] Compare $S_{i,j}$ to all other length-W substrings $S_{k,l}$ to
           calculate the score $P(S_{k,l}, S_{i,j})$;
  [Step 2] For each sequence k determine the substring $S_{k,\max k}$ where
           $\max k = \text{argmax}_{1<=l<=L(k)-W+1}\{P(S_{k,l}, S_{i,j})\}$;
  [Step 3] Sort and align the identified N substrings in Step 2 to
           determine the estimated score of starting point (i,j);
end
```

Fig. 2. Starting point search algorithm

Table 1. Percentage of MEME (version 3.5.4) execution time spent on starting point search for data sets of various sizes

Dataset	Number of sequences	Average sequence length	Runtime using default parameters	Percentage spent on "starting point search"
Mini-drosoph	4	124,824	15,642 sec	99.4%
Hs_100	100	5000	16,017 sec	96.3%
Hs_200	200	5000	60,142 sec	97.5%
Hs_400	400	5000	233,228 sec	98.7%

In practice, *W* can be considered to be much smaller than the sequence lengths. Therefore, we assume that *W* is a constant and determine the time complexities of the three steps in Figure 2 as shown in Table 2. The score between two substrings of length *W* in Step 1 is calculated using Equation (1). In Equation (1) $S_i[j]$ denotes the letter occurring at position *j* of sequence *i* and *map* is a letter frequency matrix of size $A \times A$.

$$P(S_{k,l}, S_{i,j}) = \sum_{r=0}^{W-1} map(S_k[l+r], S_i[j+r]) \qquad (1)$$

Table 2. Time complexities of the three steps in Figure 2

Step	Computational requirement	Time complexity
1	Requires an all-against-all comparison of length-*W* substrings in *X*	$O\left(\left(\sum_{i=1}^{n} L(i)\right)^2\right)$
2	Requires a linear search of all scores computed in Step 1	$O\left(\left(\sum_{i=1}^{n} L(i)\right)^2\right)$
3	Only deals with the *n* maximum scores identified in Step 2 and therefore has a lower overall complexity	$O\left(n \cdot \sum_{i=1}^{n} L(i)\right)$

2.2 General Purpose Computations on GPUs

In the past few years, the fast increasing power of the GPU (Graphics Processing Unit) has made it a compelling platform for computationally demanding tasks in a wide variety of application domains. Currently, the peak performance of state-of-the-art consumer graphics cards is more than ten times faster than that of comparable CPUs. Furthermore, GPU performance has been increasing from two to two-and-a-half times a year. This growth rate is faster than Moore's law as it applies to CPUs, which corresponds to about one-and-half times a year. The high price/performance ratio, rapid increase in performance, and widespread availability of GPUs has propelled them to the forefront of high performance computing.

Recently, NVIDIA has released the multi-threaded CUDA programming interface for GPU programming. However, CUDA was not available at the time of writing our GPU-MEME code. Therefore, the presented GPU-MEME algorithm is implemented using the graphics-based GLSL language [4]. Computation using GLSL on a GPU follows a fixed order of processing stages, called the graphics pipeline (see Figure 3). The streaming pipeline consists of three stages: vertex processing, rasterization and fragment processing. The vertex processing stage transforms three-dimensional vertex world coordinates into two-dimensional vertex screen coordinates. The rasterizer then converts the geometric vertex representation into an image fragment representation. Finally, the fragment processor forms a color for each pixel by reading texels from the texture memory. In order to meet the ever-increasing performance requirements set by the gaming industry, modern GPUs support programmability of the vertex and fragment processors using two types of parallelism. Firstly, multiple processors work on the vertex and fragment processing stage, i.e. they operate on different vertices and fragments in parallel. Secondly, operations on 4-dimensional vectors (the four channels Red/Green/Blue/Alpha (RGBA)) are natively supported without performance loss.

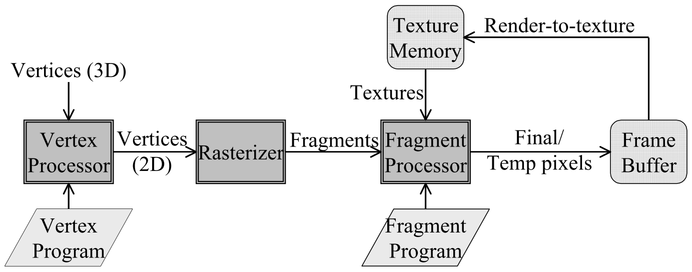


Fig. 3. Graphics pipeline

3 GPU-Accelerated Motif Discovery

3.1 Parallel Streaming Algorithm

The GPU analog of arrays on the CPU are textures. GPUs treat objects as polygon meshes, textures can then be attached to the polygon. Each vertex of the polygon contains texture location information in the form of (x,y) coordinates and the requested texture is interpolated across the polygon surface. This process is called *texture mapping*.

From Step 1 in Figure 2, we can see that for a given length-W substring $S_{i,j}$ the scores $P(S_{k,l},S_{i,j})$ need to be calculated independently from each other for all $1 \le k \le n$ and $1 \le l \le L(k)–W+1$. Our method takes advantage of the fact that all $n\cdot(L(k)–W+1)$ scores can be computed independent of each other. Therefore, we map the sequence dataset (X), the letter frequency matrix (*map*) and the score matrix (i.e. all scores for a fixed (i,j), denoted as: $[P(S_{k,l},S_{i,j})]_{1\le k\le n,1\le l\le L(k)-W+1}$) onto the following three textures:

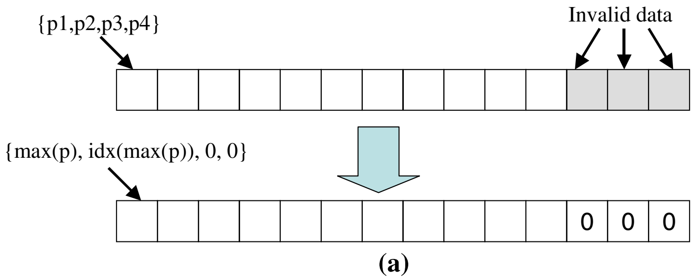
1) **Sequence dataset texture (Tex_{seq}).** We are using one row of the texture memory to store one sequence. If the sequence length is longer than the row width of the texture, several rows of texture memory will be used. Since the maximum texture size of modern GPUs is 4096×4096 and one texture element can store up to four values (RGBA), a sequence of length L requires $\lceil L/16384 \rceil$ rows of texture memory. In this section, we assume that one sequence fits into one row of texture memory. The partitioning of a sequence onto multiple rows is discussed in Section 3.2.
2) **Letter frequency matrix texture (Tex_{freq}).** This is a relatively small matrix of size $A\times A$. The utilized alphabet for DNA sequences in MEME is $\Sigma = \{$A, C, G, T, X$\}$, where X represents an unknown nucleotide. Hence, the letter frequency matrix for DNA can be stored in a 5×5 texture.
3) **Score texture (Tex_{score}).** The output of each rendering pass will be written to graphics memory directly, which can then be fed back in as a new stream of texture data for further processing. The dimension of the score matrix texture is equal to the dimension of Tex_{seq}. This allows reusing the coordinates of Tex_{seq} to do lookup operations for Tex_{score}, thus reducing extra coordinate computations. If multiple sequence dataset textures have to be used, the same number of score textures is required to store the rendering results.

Fragment programs are used to implement the arithmetic operations on the above textures specified by Equation (1). Equation (1) requires W table lookups and $W-1$ additions to calculate $P(S_{k,l},S_{i,j})$. The number of operations can be reduced to two lookups and two additions/subtractions by using $P(S_{k,l},S_{i,j})$ to calculate $P(S_{k,l-1},S_{i,j+1})$ as follows:

$$P(S_{k,l},S_{i,j}) = P(S_{k,l-1},S_{i,j}) + map(S_i[j+W],S_k[l+W-1]) - map(S_i[j-1],S_k[l-1]) \quad (2)$$

As shown in Equation (2), during each rendering pass the newly computed score matrix $[P(S_{k,l},S_{i,j})]_{1\le k\le n,1\le l\le L(k)-W+1}$ is stored in the texture memory as a texture. The subsequent rendering pass reads the previous score matrix from the texture memory. Since the calculation of the score matrix $[P(S_{k,l},S_{i,j+1})]_{1\le k\le n,1\le l\le L(k)-W+1}$ depends on the score matrix $[P(S_{k,l},S_{i,j})]_{1\le k\le n,1\le l\le L(k)-W+1,}$ two score matrices have to be stored as separate texture buffers. We are using a cyclic method to swap the buffer function as follows: First, the score matrix $[P(S_{k,l},S_{i,j})]_{1\le k\le n,1\le l\le L(k)-W+1}$ is in the form of a texture input, and $[P(S_{k,l},S_{i,j+1})]_{1\le k\le n,1\le l\le L(k)-W+1}$ is the render target. In the subsequent iteration, $[P(S_{k,l},S_{i,j+1})]_{1\le k\le n,1\le l\le L(k)-W+1}$ is treated as the input texture and $[P(S_{k,l},S_{i,j})]_{1\le k\le n,1\le l\le L(k)-W+1}$ is the render target.

Once $[P(S_{k,l},S_{i,j})]_{1\le k\le n,1\le l\le L(k)-W+1}$ is calculated, the maximum score for each sequence sample has to be found (Step 2 in Figure 2). In order to collect the maximum score for each sequence in texture memory, a preprocessing step eliminates invalid scores in Tex_{score}. These scores will be zeroed in the preprocess step. Thus, they will not influence the final maximum comparison results. This step requires a new texture called Tex_{length}, which stores information about the length of each sequence. After a preprocessing operation, a series of parallel reduction steps are performed on Tex_{score}. Each parallel reduction step consists of two operations. Firstly,



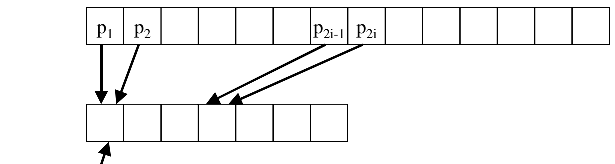


Fig. 4. (a) Preprocessing step; (b) Parallel reduction

all elements with odd indices in the score texture will be compared to their corresponding following elements with even indices. Secondly, an adjustment of texture coordinates is performed. These two operations iterate until the maximum score of each is calculated (see Figure 4(b)). Note that the operations in the first reduction step are slightly different from the following steps. In the first step, each fragment processor compares the four scores in the *R*, *G*, *B* and *A*-channels of a single texture pixel and then outputs the maximum score together with its index into the *R* and *G* channels respectively (see Figure 4(a)). Assuming a maximal sequence length of $L_{\max}$, the number of reduction passes for the maximum computation procedure is therefore $1+\log_2\lceil L_{max}/4\rceil$.

```
activate, enable and create texture $Tex_{seq}$ and load sequence data into it;
activate, enable and create texture $Tex_{freq}$ and load letter frequency
   matrix into it;
activate, enable and create texture $Tex_{length}$ and load sequence length
   information into it;
enable and create textures $Tex_{score\_j}$ and $Tex_{score\_j+1}$;
create and initialize a render buffer rBuffer;
for each sequence sample i do
   for each substring j in sequence i do
      set $Tex_{score\_j}$ as render buffer and $Tex_{score\_j+1}$ as read buffer;
      set texture coordinates $Tex_{seq}$[4], $Tex_{freq}$[4], $Tex_{length}$[4];
      set vertex coordinates vertex[4];
      DrawQuad($Tex_{seq}$, $Tex_{freq}$, $Tex_{length}$, vertex); /*call kernel program */
      do parallel reduction operation on the score matrix texture to
      get the maximum score for each sequence sample;
      change the functions of $Tex_{score\_j}$ and $Tex_{score\_j+1}$ in a cyclic way;
      Read back the maximum scores to CPU for further processing;
   end
end
```

Fig. 5. Pseudocode of our streaming algorithm for starting point search

As mentioned in Section 2.1, Step (3) in the starting point search algorithm has a lower time complexity than Steps (1) and (2). Therefore, the produced maximum scores are read back from texture memory to the CPU. The CPU then performs Step (3) sequentially. We will show in Section 4 that the runtime for Step (3) on the CPU is dominated by the runtime for Steps (1) and (2) on the GPU. The pseudocode of our streaming algorithm for starting point search is shown in Figure 5.

3.2 Partitioning and Implementation

So far, we have assumed that each sequence fits into one row of texture memory. In practice, the length of the sequences may be larger and the computation must be partitioned onto several rows. This is incorporated into our streaming algorithm as follows.

1) **Multi-row storage in the sequence dataset texture.** As mentioned in Section 3.1, we are using one row of texture memory to store one sequence. If the sequence length is longer than the row width of the texture, several rows of texture memory will be used. Since the maximum texture size of modern GPUs is

4096×4096 and one texture element can store up to four values (RGBA), a sequence of length L requires $\lceil L/16384 \rceil$ rows of texture memory. Assume L_{max} is the length of the longest sequence in the dataset, in practice we let all long sequences take the same $R_{max} = \lceil L_{max}/16384 \rceil$ rows in the texture memory for simplicity. In this case, a texture can contain $N_{max} = \lfloor 4096/R_{max} \rfloor$ long sequences. Overall, we need $\lceil n/N_{\max} \rceil$ textures to store the complete sequence dataset. Correspondingly, $\lceil n/N_{\max} \rceil$ score textures will be used to store the rendering results.

2) **Multi-row indexing for texture lookups.** If $L_{\max} > 16384$, there exist cases where the letters $S_i[j]$ and $S_i[j+1]$ are stored in different texture rows. In order to handle these cases correctly, we use $(i\%N_{max} + \lceil j/16384 \rceil, j\%16384)$ instead of (i, j) to do texture lookups for $(i, j+1)$.
3) **Multi-row parallel reduction.** According to Section 3.1, $1+\log_2\lceil 16384/4 \rceil$ parallel reduction steps are required to get the maximum in each texture row. Additional $\log_2 R_{max}$ passes are required to get the maximum scores for sequences occupying R_{max} texture rows.

In order to make full use of the computing power in a PC, we have designed and implemented a multi-threaded CPU-GPU collaborative architecture for our streaming algorithm. Figure 6 illustrates the structure of this architecture. It contains three kinds of threads:

1) **Daemon thread:** This thread runs in the background and takes care of the execution of the whole process. It will respond to the data readback operations between the CPU and GPU threads.

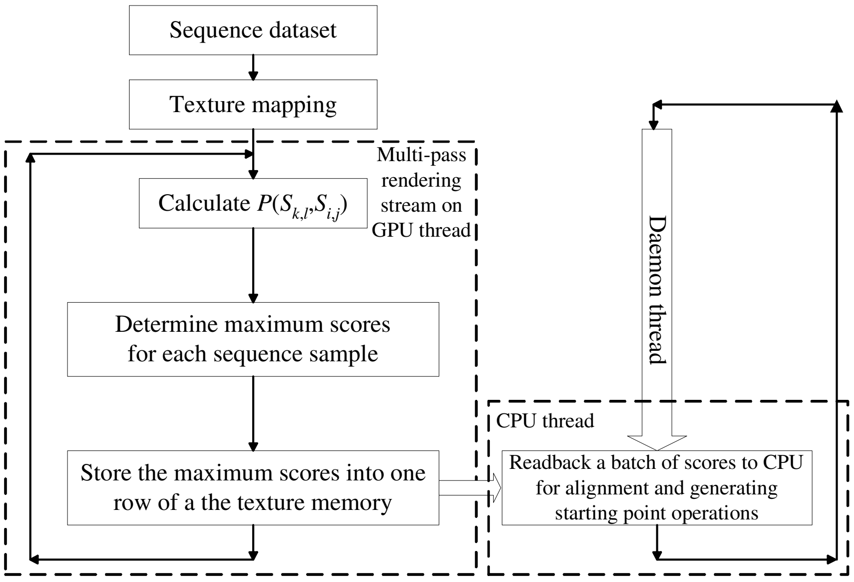

Sequence dataset
Texture mapping
Multi-pass rendering stream on GPU thread
Calculate $P(S_{k,l}, S_{i,j})$
Determine maximum scores for each sequence sample
Store the maximum scores into one row of a the texture memory
Daemon thread
CPU thread
Readback a batch of scores to CPU for alignment and generating starting point operations

Fig. 6. The structure of our multi-threaded collaborative CPU-GPU architecture

2) **GPU thread:** Because of the implicit data-parallelism processing power of the GPU, it is used to process the compute-intensive calculations. Tasks such as the calculation of $[P(S_{k,l},S_{i,j})]_{1\leq k\leq n,1\leq l\leq L(k)-W+1}$ and parallel reduction operations on $[P(S_{k,l},S_{i,j})]_{1\leq k\leq n,1\leq l\leq L(k)-W+1}$ are all done by the GPU thread. In order to increase the readback efficiency, the parallel reduction scores during each rendering pass will be first stored in one row of a texture Tex_{max}. After a constant number of rendering passes, a batch of data in Tex_{max} are read back to the CPU for further processing.
3) **CPU thread:** Because of the sequential computing characteristics and the lower time complexity of Step 3 in Figure 2, we let the CPU process this step. When the CPU gets a batch of rendering data from the GPU, it will do the global maximum alignment and starting point generation operations on the data sequentially.

According to our experiments (see Section 4), the GPU thread dominates the runtime. Thus, the runtime of the CPU thread does not influence the overall runtime, since it runs concurrently to the GPU thread.

4 Performance Evaluation

We have implemented the proposed algorithm using C and the GPU programming language *GLSL* (OpenGL Shading Language) [4] and evaluated it on the following graphics card:

– *Nvidia GeForce 8800 GTX*: 1.35 GHz engine clock speed, 900 MHz memory clock speed, 128 stream processors, 768 MB device memory. Tests have been conducted with this card installed in a PC with an Intel Petium4 3.0GHz, 1 GByte RAM running Fedora Core 6 Linux.

A set of performance evaluation tests have been conducted using different numbers of DNA sequences to evaluate the processing time of the GPU implementation versus that of the original MEME implementation. The sequential MEME application is benchmarked on an Intel Pentium4 3GHz processor with 1 Gbyte RAM running Fedora Core 6 Linux. We have used MEME Version 3.5.4, which is available online at http://meme.nbcr.net/meme/intro.html for our evaluation.

The evaluated datasets are the largest dataset supplied by MEME (called mini-drosoph) and three datasets of human promoter regions consisting of 100, 200, and 400 sequences of lengths 5,000 base-pairs each (called HS_5000_100, HS_5000_200, HS_5000_400). We have used MEME's default parameters for evaluation. The results for our experiments are shown in Table 3. The CPU alignment part (rows shaded in gray) and the computations on the GPU run concurrently. Since the CPU alignment requires less time, its runtime does not influence the overall runtime. From Table 3 we can see that our GPU implementation achieves speedups of almost fourteen compared to the starting point search stage in MEME and twelve compared to the overall runtime.

Table 3. Comparison of runtimes (in seconds) and speedups of MEME running on a single Pentium4 3GHz to our GPU-accelerated version running on a Pentium4 3GHz with an Nvidia GeForce 8800 GTX for different datasets. The time and percentage spend on different parts of the algorithm are also reported.

Dataset Name, Number of sequences (average length)			HS_5000_100, 100 (5,000)		HS_5000_200, 200 (5,000)	
MEME (P4, 3GHz)	Overall		16017	[100.0%]	60142	[100.0%]
	Starting Point Search		15428	[96.3%]	58656	[97.5%]
	EM		589	[3.7%]	1486	[2.5%]
GPU-MEME (GeForce 8800 GTX)	Overall		1755	[100.0%]	5894	[100.0%]
	Starting Point Search	Score Comp. (GPU)	923	[52.6%]	3565	[60.5%]
		Parallel Red. (GPU)	182	[10.4%]	707	[12.0%]
		Result Readb. (GPU)	61	[3.5%]	136	[2.3%]
		Alignment (CPU)	*1042*	*[59.4%]*	*2045*	*[34.7%]*
	EM (CPU)		589	[33.6%]	1486	[25.2%]
Speedup	Overall		9.1		10.2	
	Starting Point Search		13.2		13.3	

Dataset Name, Number of sequences (average length)			HS_5000_400, 400 (5,000)		Mini-drosoph, 4 (124,824)	
MEME (P4, 3GHz)	Overall		233228	[100.0%]	15642	[100.0%]
	Starting Point Search		230283	[98.7%]	15545	[99.4%]
	EM		2945	[1.3%]	97	[0.6%]
GPU-MEME (GeForce 8800 GTX)	Overall		19895	[100.0%]	1375	[100.0%]
	Starting Point Search	Score Comp. (GPU)	13818	[69.5%]	1061	[77.2%]
		Parallel Red. (GPU)	2764	[13.9%]	209	[15.2%]
		Result Readb. (GPU)	368	[1.8%]	8	[0.6%]
		Alignment (CPU)	*4067*	*[20.4%]*	*244*	*[17.7%]*
	EM		2945	[14.8%]	97	[7.1%]
Speedup	Overall		11.7		11.4	
	Starting Point Search		13.6		12.6	

We have also compared our speedups to the MPI-based ParaMEME implementation ([3], available online at http://meme.nbcr.net/meme/intro.html) on a CPU cluster. The utilized cluster is a 6-node Intel Xeon Dual-Processor cluster with a 1GBit/sec Myrinet switch running Red Hat Linux 3.2.3-24. Table 4 shows a comparison of speedups achieved with ParaMEME compared to our GPU-MEME implementation. As can be seen, our implementation on a single GPU is comparable to the MPI approach on a cluster with 12 CPUs.

Table 4. Speedups of GPU-MEME on a single GPU and ParaMEME on a 12-CPU cluster

Dataset Name	Speedup GPU-MEME	Speedup ParaMEME
Mini-drosoph	11.4	12.6
HS_5000_100	9.1	11.4
HS_5000_200	10.2	11.2
HS_5000_400	11.7	11.1

Table 5. Comparison of the runtime and speedup of MEME running on a P4 3GHz to GPU-MEME running on a cluster with GeForce 8800 GTX cards. Speedup is compared to the sequential MEME code and denoted as "*speedup CPU*". Efficiency with respect to the number of utilized GPUs is denoted as "*efficiency GPU*".

	Mini-drosoph			*HS_5000_100*		
	runtime (sec.)	speedup CPU	efficiency GPU	runtime (sec.)	speedup CPU	efficiency GPU
Seq. MEME	15,642	1.0	N.A.	16,017	1	N.A.
GPU-MEME (**1**×8800GTX)	1,375	11.4	100.0%	1,755	9.1	100.0%
GPU-MEME-MPI (**2**×8800GTX)	760	20.6	90.5%	1,065	15.0	82.5%
GPU-MEME-MPI (**4**×8800GTX)	383	40.8	89.8%	538	29.8	81.5%
GPU-MEME-MPI (**6**×8800GTX)	260	60.2	88.2%	368	43.5	79.5%

	HS_5000_200			*HS_5000_400*		
	runtime (sec.)	speedup CPU	efficiency GPU	runtime (sec.)	speedup CPU	efficiency GPU
Seq. MEME	60,142	1.0	N.A.	233,228	1	N.A.
GPU-MEME (**1**×8800GTX)	5,894	10.2	100.0%	19,895	11.7	100.0%
GPU-MEME-MPI (**2**×8800GTX)	3,394	17.7	87.0%	11,107	20.1	89.5%
GPU-MEME-MPI (**4**×8800GTX)	1,699	35.4	86.8%	5,521	42.2	90.0%
GPU-MEME-MPI (**6**×8800GTX)	1,168	51.5	84.0%	3,817	61.1	86.8%

In order to achieve an even higher speedup, we have extended our GPU-MEME approach to a GPU cluster using MPI. The coarse-grained MPI parallelization assigns to each processor an approximately equal number of starting points to be compared to the input sequence dataset. Table 5 shows a comparison of runtime and speedups of the GPU-MEME cluster version for up to six GPUs compared to the sequential MEME implementation and to our GPU-MEME implementation on a single GPU.

5 Conclusion

In this paper, we have introduced a streaming algorithm for motif finding in biological sequences that can be efficiently implemented on modern graphics hardware. The design is based on data-parallel computing characteristics in the motif finding process and makes full use of the available computing power in a PC. Our implementation achieves speedups of over an order of magnitude compared to the widely used MEME tool. At least the same number of CPUs connected by a fast switch is required to achieve a similar speedup using the MPI-based ParaMEME code. A comparison of these two parallelization approaches shows that graphics hardware acceleration is superior in terms of price/performance. The presented GPU software is a proof-of-concept parallelization and can be used for the OOPS and ZOOPS search models. Our future work will include integrating the TCM model into our GPU framework and making the software available for public use. We are also planning to port the presented GLSL code to the newly released CUDA programming interface for GPU programming.

Acknowledgement

We thank Geir Kjetil Sandve for the sequence datasets. The work was supported by the A*Star BMRC research grant No. 04/1/22/19/375.

References

1. Bailey, T.L., Elkan, C.: Unsupervised learning of multiple motifs in biopolymers using expectation maximization. Machine Learning 21, 51–80 (1995)
2. Bailey, T.L., Williams, N., Misleh, C., Li, W.W.: MEME: discovering and analyzing DNA and protein motifs. Nucleic Acid Research 34, W369–W373 (2006)
3. Grundy, W.N., Bailey, T.L., Elkan, C.P.: ParaMEME: A parallel implementation and a web interface for a DNA and protein motif discovery tool. Computer Applications in the Biological Sciences (CABIOS) 12, 303–310 (1996)
4. Kessenich, J., Baldwin, D., Rost, R.: The OpenGL Shading Language, Document Revision 8 (2006), `http://www.opengl.org/documentation/glsl/`
5. Lawrence, C., Altschul, S., Boguski, M., Liu, J., Neuwald, A., Wootton, J.: Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. Science 262, 208–214 (1993)
6. Liu, W., Schmidt, B., Voss, G., Muller-Wittig, W.: Streaming Algorithms for Biological Sequence Alignment on GPUs. IEEE Transactions on Parallel and Distributed Systems 18(10), 1270–1281 (2007)
7. Manavski, S.A., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9(Suppl. 2), S10 (2008)
8. Sabatti, C., Rohlin, L., Lange, K., Liao, J.C.: Vocabulon: a dictionary model approach for reconstruction and localization of transcription factor binding sites. Bioinformatics 21(7), 922–931 (2005)
9. Sandve, G.K., Nedland, M., Syrstad, B., Eidsheim, L.A., Abul, O., Drablas, F.: Accelerating motif discovery: Motif matching on parallel hardware. In: Bücher, P., Moret, B.M.E. (eds.) WABI 2006. LNCS (LNBI), vol. 4175, pp. 197–206. Springer, Heidelberg (2006)
10. Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney, A.: High-throughput sequence alignment using Graphics Processing Units. BMC Bioinformatics 8(474) (2007)
11. Sumazin, P., et al.: DWE: Discriminating Word Enumerator. Bioinformatics 21(1), 31038 (2005)