2011 IEEE International Parallel & Distributed Processing Symposium

An Ultrafast Scalable Many-core Motif Discovery Algorithm for Multiple GPUs

Yongchao Liu, Bertil Schmidt, Douglas L. Maskell
School of Computer Engineering
Nanyang Technological University
Singapore
e-mail: {liuy0039, asbschmidt, asdouglas}@ntu.edu.sg

***Abstract*—The identification of genome-wide transcription factor binding sites is a fundamental and crucial problem to fully understand the transcriptional regulatory processes. However, the high computational cost of many motif discovery algorithms heavily constraints their application for large-scale datasets. The rapid growth of genomic sequences and gene transcription data further deteriorates the situation and establishes a strong requirement for time-efficient scalable motif discovery algorithms. The emergence of many-core architectures, typically CUDA-enabled GPUs, provides an opportunity to reduce the execution time by an order of magnitude without the loss of accuracy. In this paper, we present mCUDA-MEME, an ultrafast scalable many-core motif discovery algorithm for multiple GPUs based on the MEME algorithm. Our algorithm is implemented using a hybrid combination of the CUDA, OpenMP and MPI parallel programming models in order to harness the powerful compute capability of modern GPU clusters. At present, our algorithm supports OOPS and ZOOPS models, which are sufficient for most motif discovery applications. mCUDA-MEME achieves significant speedups for the starting point search stage (and the overall execution) when benchmarked, using real datasets, against parallel MEME running on 32 CPU cores. Speedups of up to 1.4 (1.1) on a single GPU of a Fermi-based Tesla S2050 quad-GPU computing system and up to 10.8 (8.3) on the eight GPUs of a two Tesla S2050 system were observed. Furthermore, our algorithm shows good scalability with respect to dataset size and the number of GPUs (availability:https://sites.google.com/site/yongchaosoftware/mcuda-meme).**

Motif discovery; MEME; CUDA; MPI; OpenMP; GPU

I. INTRODUCTION

De novo motif discovery is crucial to the complete understanding of transcription regulatory processes by identifying transcription factor binding sites (TFBSs) on a genome-wide scale. Algorithmic approaches for motif discovery can be classified into two categories: iterative and combinatorial. Iterative approaches generally exploit probabilistic matching models, such as Expectation Maximization (EM) [1] and Gibbs sampling [2]. These approaches are often preferred because they use position-specific scoring matrices to describe the matching between a motif instance and a sequence, instead of a simple Hamming distance. Combinatorial approaches employ deterministic matching models, such as word enumeration [3] and dictionary methods [4].

MEME [5] is a popular and well established motif discovery algorithm, which is primarily comprised of the starting point search (SPS) stage and the EM stage. However, the high computational cost of MEME constrains its application for large-scale datasets, such as motif identification in whole peak regions from ChIP-Seq datasets for transcription factor binding experiments [6]. This encourages the use of high-performance computing solutions to meet the execution time requirement. Several attempts have been made to improve execution speed on conventional computing systems, including distributed-memory workstation clusters [7] and special-purpose hardware [8]. The emerging many-core architectures, typically Compute Unified Device Architecture (CUDA)-enabled GPUs [9], have demonstrated their power for accelerating bioinformatics algorithms [10] [11] [12] [13]. This convinces us to employ CUDA-enabled GPUs to accelerate motif discovery. Previously we have presented CUDA-MEME, based on MEME version 3.5.4, for a single GPU device which accelerates motif discovery using two parallelization approaches, namely: sequence-level parallelization and substring-level parallelization. The detailed implementations of the two approaches have been described in [14].

However, the increasing size and availability of ChIP-Seq datasets established the need for parallel motif discovery with even higher performance. Therefore in this paper, we present mCUDA-MEME, an ultrafast scalable many-core motif discovery algorithm for multiple GPUs. Our algorithm is designed based on MEME version 4.4.0 which incorporates position-specific priors (PSP) to improve accuracy [15]. In order to harness the powerful compute capability of GPU clusters, we employ a hybrid combination of CUDA, Open Multi-Processing (OpenMP) and Message Passing Interface (MPI) parallel programming models to implement this algorithm. Compared to CUDA-MEME, mCUDA-MEME introduces four new significant features:

- Supports multiple GPUs in a single host or a GPU cluster through a MPI-based design;
- Supports starting point search from the reverse complements of DNA sequences;
- Employs multi-threaded design using OpenMP to accelerate the EM stage;
- Incorporates PSP prior probabilities to improve accuracy.

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.183

IEEE computer society

Since the sequence-level parallelization is advantageous to the substring-level parallelization in execution speed [14], our following discussions only refer to the sequence-level parallelization. Real datasets are employed to evaluate the performance of mCUDA-MEME and parallel MEME (i.e. the MPI version of MEME distributed with the MEME software package). Compared to parallel MEME (version 4.4.0) running on 32 CPU cores, mCUDA-MEME achieves a speedup of the starting point search stage (and the overall execution) of up to 1.4 (1.1) times on a single GPU of a Fermi-based Tesla S2050 quad-GPU computing system and up to 10.8 (8.3) times on eight GPUs of a two Tesla S2050 system. Furthermore, our algorithm shows good scalability with respect to dataset size and the number of GPUs.

The rest of this paper is organized as follows. Section 2 briefly introduces the MEME algorithm and the CUDA, OpenMP and MPI parallel programming models. Section 3 details the new features of mCUDA-MEME. Section 4 evaluates the performance using real datasets, and Section 5 concludes this paper.

II. BACKGROUND

A. The MEME Algorithm

Given a set of protein or DNA sequences, MEME attempts to search for statistically significant (unknown) motif occurrences, which are believed to be shared in the sequences, by optimizing the parameters of statistical motif models using the EM approach. MEME provides support for three types of search modes: one occurrence per sequence (OOPS), zero or one occurrence per sequence (ZOOPS), and two component mixture (TCM). The OOPS model postulates that there is exactly one motif occurrence per sequence in the dataset, the ZOOPS model postulates zero or one motif occurrence per sequence, and the TCM model postulates that there can be any number of non-overlapping occurrences per sequence [5]. Since the OOPS and ZOOPS models are sufficient for most motif finding applications, our algorithm in this paper concentrates on these two models.

MEME begins a motif search with the creation of a set of motif models. Each motif model θ is a position specific probability matrix representing frequency estimates of letters occurring in different positions. Given a motif of width w defined over an alphabet $\Sigma = \{A_1, A_2, ..., A_{|\Sigma|}\}$, each value $\theta(i, j)$ ($1 \leq i \leq |\Sigma|$ and $0 \leq j \leq$ w) of the matrix is defined as:

$$\theta(i,j) = \begin{cases} \text{probability of } A_i \text{ at position } j \text{ of the motif}, & 1 \leq j \leq w \\ \text{probability of } A_i \text{ not in the motif}, & j = 0 \end{cases} \quad (1)$$

A starting point is an initial motif model $\theta^{(0)}$ from which the EM stage runs for a fixed number of iterations, or until convergence, to find the final motif model $\theta^{(q)}$ with maximal posterior probability. To find the set of starting points for a given motif width, MEME converts each substring in a sequence dataset into a potential motif model and then determines its statistical significance by calculating the weighted log likelihood ratio [15] on different numbers of predicted sites, subject to the model used. The potential motif models with the highest weighted log likelihood ratio are selected as starting points for the successive EM algorithm.

When performing the starting point search, independent computation from each substring of length w in the dataset $S=\{S_1, S_2, ..., S_n\}$ of n input sequences is conducted to determine a set of initial motif models. The following notations are used for the convenience of discussion: l_i denotes the length of S_i, $\overline{S_i}$ denotes the reverse complement of S_i, $S_{i,j}$ denotes the substring of length w starting from position j of S_i, $S_i(j)$ denotes the j-th letter of S_i, where $1 \leq i \leq n$ and $0 \leq j \leq l_i\text{-}w$. The starting point search process is primarily comprised of three steps for the OOPS and ZOOPS models:

- Calculate the probability score $P(S_{i,j}, S_{k,l})$ from the forward strand (or $P(S_{i,j}, \overline{S}_{k,l})$ from the reverse complement), which is the probability that a site starts at position l in S_k when a site starts at position j in S_i. The time complexity is $O(l_i \cdot l_k)$ for each sequence pair S_i and S_k.
- Identify the highest-scoring substring $S_{k,maxk}$ (as well as its strand orientation) for each S_k. The time complexity is $O(l_k)$ for each sequence S_k.
- Sort the n highest-scoring substrings $\{S_{k,maxk}\}$ in decreasing order of scores and determine the potential starting points. The time complexity is $O(nlogn)$ for OOPS and $O(n^2w)$ for ZOOPS.

The probability score $P(S_{i,j}, S_{k,l})$ is computed as:

$$P(S_{i,j}, S_{k,l}) = \sum_{p=0}^{w-1} mat[S_i(j+p)][S_k(l+p)] \quad (2)$$

where mat denotes the letter frequency matrix of size $|\Sigma|\times|\Sigma|$. To reduce computation redundancy, (2) can be further simplified to (3), where the computation of the probability scores $\{P(S_{i,j}, S_{k,l})\}$ in the j-th iteration depends on the resulting scores $\{P(S_{i,j\text{-}1}, S_{k,l\text{-}1})\}$ in the (j-1)-th iteration. However, $P(S_{i,j}, S_{k,0})$ needs to be computed individually using (2).

$$P(S_{i,j}, S_{k,l}) = P(S_{i,j-1}, S_{k,l-1}) + mat[S_i(j+w-1)][S_k(l+w-1)] - mat[S_i(j-1)][S_k(l-1)] \quad (3)$$

B. The CUDA Programming Model

More than a software and hardware co-processing architecture, CUDA is also a parallel programming language extending general programming languages, such as C, C++ and Fortran with a minimalist set of abstractions for expressing parallelism. CUDA enables users to write parallel scalable programs for CUDA-enabled processors with familiar languages [16]. A CUDA program is comprised of two parts: a host program running one or more sequential threads on a host CPU, and one or more parallel kernels able to execute on Tesla [17] and Fermi [18] unified graphics and computing architectures.

A kernel is a sequential program launched on a set of lightweight concurrent threads. The parallel threads are organized into a grid of thread blocks, where all threads in a thread block can synchronize through barriers and communicate via a high-speed, per block shared memory (PBSM). This hierarchical organization of threads enables thread blocks to implement coarse-grained task and data parallelism and lightweight threads, comprising a thread block, to provide fine-grained thread-level parallelism. Threads from different thread blocks in the same grid are able to cooperate through atomic operations on global memory shared by all threads.

The CUDA-enabled processors are built around a fully programmable scalable processor array, organized into a number of streaming multiprocessors (SMs). For the Tesla architecture, each SM contains 8 scalar processors (SPs) and shares a fixed 16 KB of PBSM. For the Tesla-based GPU series, the number of SMs per device varies from generation to generation. For the Fermi architecture, it contains 16 SMs with each SM having 32 SPs. Each SM in the Fermi architecture has a configurable PBSM size from the 64 KB on-chip memory. This on-chip memory can be configured as 48 KB of PBSM with 16 KB of L1 cache or as 16 KB of PBSM with 48 KB of L1 cache. When executing a thread block, both architectures split all the threads in the thread block into small groups of 32 parallel threads, called warps, which are scheduled in a single instruction, multiple thread fashion. Divergence of execution paths is allowed for threads in a warp, but SMs realize full efficiency and performance when all threads of a warp take the same execution path.

C. *The OpenMP Programming Model*

OpenMP is a compiler-directive-based application program interface for explicitly directing multi-threaded, shared-memory parallelism [19]. This explicit programming model enables programmers to have a full control over parallelism. OpenMP is primarily comprised of compiler directives, runtime library routines and environment variables that are specified for C/C++ and Fortran programming languages. As an open specification and standard for expressing explicit parallelism, OpenMP is portable across a variety of shared memory architectures and platforms.

The basic idea behind OpenMP is the existence of multiple threads in the shared memory programming paradigm which enables data-shared parallel execution. In OpenMP, the unit of work is a thread and the fork-join model is used for parallel execution. An OpenMP program begins as the master thread, and executes sequentially until encountering a parallel region construct specified by the programmer. Successively, the master thread creates a team of parallel threads to execute in parallel the statements that are enclosed by the parallel region construct. Having completed the statements in the parallel region construct, the team threads synchronize and terminate, leaving only the master thread.

D. *The MPI Programming Model*

MPI is a de facto standard for developing portable parallel applications using the message passing mechanism [20]. MPI works on both shared and distributed memory architectures and platforms, offering a highly portable solution to parallel programming on a variety of hardware topologies.

In MPI, it defines each worker as a process and enables the processes to execute different programs. This multiple program, multiple data model offers more flexibility for data-shared or data-distributed parallel program design. Within a computation, processes communicate data by calling runtime library routines, specified for the C/C++ and Fortran programming languages, including point-to-point and collective communication routines. Point-to-point communication is used to send and receive messages between two named processes, suitable for local and unstructured communications. Collective (global) communication is used to perform commonly used global operations (e.g. reduction and broadcast operations).

III. METHODS

mCUDA-MEME employs a hybrid CPU+GPU computing framework for each of its MPI processes in order to maximize performance by overlapping the CPU and GPU computation. A MPI process therefore consists of two threads: a CPU thread and a GPU thread. mCUDA-MEME requires a non-overlapping one-to-one correspondence between a MPI process and a GPU device. Unfortunately, the MPI runtime environment does not provide a mechanism to perform the automatic mapping between MPI processes and GPU devices. In this case, mCUDA-MEME employs a registration-based management mechanism (as shown in Fig. 1) to dynamically map GPU devices to MPI processes.

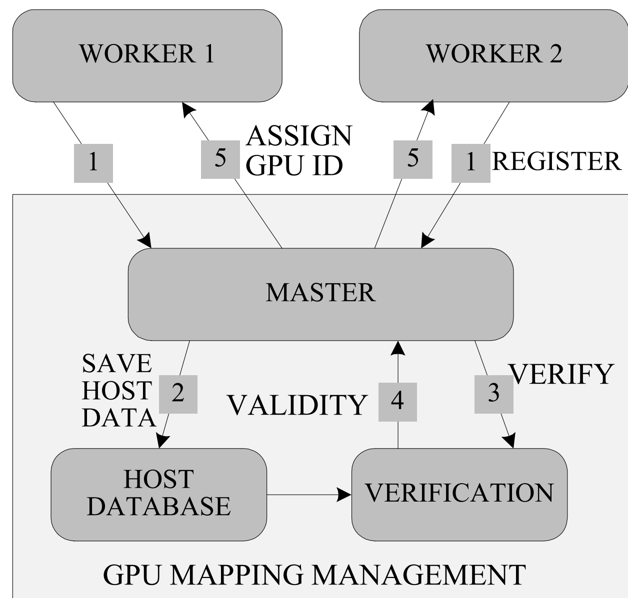

Figure 1. Registration-based GPU mapping management diagram.

Firstly, a master process is specified and each worker process (including the master process) registers the name of the host machine in which it resides, as well as the number of qualified GPUs in the host, to the master process. Secondly, the master process iterates each registered host and checks the number of processes distributed to it and the number of qualified GPUs it has. The registration is acceptable if each registered host does not have more processes than qualified GPUs, and is unacceptable, otherwise. If the mapping between processes and GPUs is valid, the master process enumerates each host and assigns a unique GPU identifier to each process running in the host.

For the starting point search, most execution time is spent on the computation of probability scores to select the highest-scoring substring $S_{k,maxk}$ for each S_k. mCUDA-MEME selects the highest-scoring substrings while computing the scores from the forward or reverse strands on GPUs. When using the reverse complements, mCUDA-MEME calculates the probability scores $\{P(S_{i,j}, S_{k,l})\}$ and $\{P(S_{i,j}, \overline{S}_{k,l})\}$ simultaneously and selects the highest-scoring substring in a single CUDA kernel launch. This is different to MEME which performs two iterations of probability score calculation separately from the two strands of S_k. Fig. 2 shows the CUDA kernel pseudocode. Hence, when searching from two strands, MEME will (nearly) double the execution time compared with that from a single strand, whereas mCUDA-MEME is able to save some execution time by reusing common resources (e.g., the reference substring $S_{i,j}$ is fetched only once from texture memory instead of twice). Our benchmarking shows that when searching from two strands, the execution time of mCUDA-MEME only increases by about 60% compared to MEME.

The sequence-level parallelization uses multiple iterations to complete the starting point search [14]. Generally, the execution time of all iterations heavily depends on the grid size (*dimGrid*) and the thread block size (*dimBlock*) used in each iteration. When determining the grid size, we consider the maximum number of resident threads per SM. As in [14], we assume the maximum number of resident threads per SM is 1024, even though Fermi-based GPUs support a maximum number of 1536 resident threads per multiprocessor. Thus, mCUDA-MEME calculates the grid size per iteration as

$$dimGrid = 2 \times \#SM \times \frac{1024}{dimBlock} \quad (4)$$

where $\#SM$ is the number of SMs in a single GPU device and *dimBlock* is set to 64. For the sequence-level parallelization, the GPU device memory consumption only depends on the grid size and the maximum sequence length. The consumed device memory size can be estimated as

$$memSize = 32 \times dimGrid \times \max\{l_i\}, \quad 1 \le i \le n \quad (5)$$

From (5), we can see that the device memory consumption is directly proportional to the maximum sequence length for a specific GPU device. Thus, mCUDA-MEME sets a maximum sequence length threshold (by default 64K) to determine the use of the sequence-level parallelization or the substring-level parallelization that is slower but much more memory efficient.

When employing multiple GPUs, an appropriate load balancing policy is critical for high overall performance. In this paper, we assume that mCUDA-MEME is deployed in a homogenous GPU cluster, which is often the case. Since the workload is aware of S, our assumption simplifies the workload distribution between GPUs. In mCUDA-MEME, both the SPS stage and the EM stages are parallelized using MPI. For the SPS stage, to achieved good workload balance, we calculate the total number of bases in the input sequences, and (nearly) equally distribute these bases to all processes using a sequence as a unit, because a thread block in the CUDA kernel processes a sequence pair every time [14]. This makes each process hold (nearly) equal number of bases in (likely) different number of sequences distributed to it, thus having (nearly) equal number of substrings for a specific motif width. In this case, each MPI process holds a non-overlapping sequence subset of S, where the process converts each substring of this subset into a potential motif model and calculates the highest-scoring substrings of all sequences in S for this substring by invoking the CUDA kernels. After each process obtains the best starting points in its subset, a reduction operation is conducted across all processes to compute the new best starting points. For the EM stage, as we are aware of the number of initial models, we simply divide the set of initial models equally among all processes as in [7]. Unfortunately, because the execution time of the EM algorithm from each initial model is not equal and is dependent on the real data, we might not be able to balance the workloads perfectly. However, considering that the EM stage takes only a very small portion of the total execution time, the imperfect workload balancing does not impact significantly. After obtaining the best motif model for each MPI process from its subset of initial models, all

```
******************************************************************
Define S̄_i to denote the reverse complement of S_i.
******************************************************************
Step 1: Load letter frequency matrix to shared memory from constant memory;
Step 2: Get the sequences information;
Step 3: All threads calculate the probability scores {P(S_i,0, S_k,l)} and {P(S_i,0, S̄_k,l)}
        in parallel, and select and save the highest-scoring substring S_k,maxk for S_i,0,
        as well as its strand orientation.
Step 4:
   for j from 1 to l_i − w
      All threads calculate the probability scores {P(S_i,j, S_k,l)} and {P(S_i,j, S̄_k,l)}
      in parallel using {P(S_i,j-1, S_k,l-1)} and {P(S_i,j-1, S̄_k,l-1)}, and select and save
      the highest-scoring substring S_k,maxk for S_i,j, as well as its strand orientation.
   end
```

Figure 2. CUDA kernel pseudocode for probability score calculation from two strands.

processes conduct a reduction operation to get the final motif model. Since all processes only need to communicate to collect the final results in the SPS and EM stages, the communication overhead can be neglected.

When executing mCUDA-MEME with only one MPI process, the sequential EM stage limits the overall speedup as per Amdahl's law, after accelerating the SPS stage. As multi-core CPUs have become commonplace, we employ a multi-threaded design using OpenMP to parallelize the EM stage to exploit the compute power of multi-core CPUs. Since the number of threads used can be specified using OpenMP runtime routines, the multi-threaded design is well exploited when running multiple MPI processes. As mentioned above, each MPI process consists of a CPU thread and a GPU thread. We generally recommend that each MPI process runs the two threads on two CPU cores to maximize the performance. In this case, after completing the SPS stage, the one CPU core for the GPU thread of each MPI process will be wasted if the successive EM stage does not use it. Hence, it is viable to employ two threads to parallelize the EM algorithm within each MPI process. By default, mCUDA-MEME uses as many threads as the available CPU cores in a single machine when running our algorithm with only one MPI process, and two threads when using multiple MPI processes. Note that the number of threads used can also be specified by parameters.

As mentioned above, the accuracy of MEME 4.4.0 has been improved by incorporating PSP, as described in [15], which defines prior probabilities that a site starts at each position of each input sequence. The PSP approach enables the incorporation of multiple types of additional information into motif discovery and converts the additional information into the measure of the likelihood that a motif starts at each position of each input sequence. When incorporating PSP, for the SPS stage, the highest-scoring substring of each sequence S_k is determined using scores calculated from $P(S_{i,j}, S_{k,l})$ (or $P(S_{i,j}, \overline{S_{k,l}})$) and PSP prior probabilities, and for the EM algorithm, it substitutes the uniform assumption with the PSP prior probabilities. Details about the PSP approach in MEME can be obtained from [15]. Since mCUDA-MEME is implemented based on MEME 4.4.0, the PSP approach is also incorporated in our algorithm.

IV. PERFORMANCE EVALUATION

We use the peaks identified by MICSA [6] in the ChIP-Seq data for neuron-restrictive silencer factor (the peaks are available at http://bioinfo-out.curie.fr/projects/micsa) to evaluate the performance of mCUDA-MEME. To evaluate the scalability of our algorithm with respect to dataset size, three subsets of peaks with different numbers of sequences and base pairs (bps) are selected from the whole peaks (as shown in Table 1).

TABLE I. PEAK DATASETS FOR PERFORMANCE EVALUATION

Datasets	No. of Sequences	Min (bps)	Max (bps)	Total (bps)
NRSF500	500	307	529	226070
NRSF1000	1000	307	589	506378
NRSF2000	2000	307	747	1169957

All the following tests are conducted on a computing cluster with eight compute nodes that are connected by a high-speed Infiniband switch. Each compute node consists of an AMD Opteron 2378 quad-core 2.4 GHz processor and 8 GB RAM, running the Linux OS with the MPICH2 library. Furthermore, two Fermi-based Tesla S2050 quad-GPU computing systems are installed and connected to four nodes of the cluster, with each node having access to two GPUs. A single GPU of a Tesla S2050 consists of 14 SMs (a total of 448 SPs) with a core frequency of 1.15GHz and has 2.6 GB user available device memory. The common parameters used for all tests are "-mod zoops -revcomp", and the other parameters use the default values.

We have evaluated the performance of mCUDA-MEME running on a single GPU and on eight GPUs respectively, compared to parallel MEME (version 4.4.0) running on the eight compute nodes with a total of 32 CPU cores (as shown in Table 2). From Table 2, the mCUDA-MEME speedups, either on a single GPU or eight GPUs, increase as the dataset size grows, suggesting a good scalability for dealing with large-scale datasets. On eight GPUs, mCUDA-MEME significantly outperforms parallel MEME on 32 CPU cores for all datasets, where it achieves an average speedup of 8.8 (6.5) with a highest of 10.8 (8.3) for the SPS stage alone (and for the overall execution). Even on a single GPU, our algorithm still holds its own compared to parallel MEME on 32 CPU cores, with a highest speedup of 1.4 (1.1) for the SPS stage (and for the overall execution) for the large dataset, even though it only gives a speedup of 0.9 (0.7) for NRSF500 dataset.

In addition to the scalability with respect to dataset scale, we further evaluated the scalability of our algorithm with respect to the number of GPUs. Figures 3 and 4 show the speedups of mCUDA-MEME on different numbers of GPUs compared to parallel MEME on 32 CPU cores for all datasets. From the figures, we can see that for each dataset, the speedup increases (nearly) linearly as the number of GPUs increases both for the SPS stage and for the overall execution.

The above evaluation and discussions demonstrate the power of mCUDA-MEME to accelerate motif discovery for large-scale datasets with relatively inexpensive GPU

TABLE II. EXECUTION TIME (IN SECONDS) AND SPEEDUP COMPARISON BETWEEN MCUDA-MEME AND PARALLEL MEME

Datasets	mCUDA-MEME				parallel MEME		Speedups			
	8 GPUs		*1 GPU*		*32 CPU cores*		*8 GPUs*		*1 GPU*	
	SPS	*All*	*SPS*	*All*	*SPS*	*All*	*SPS*	*All*	*SPS*	*All*
NRSF500	72	115	532	698	486	523	6.8	4.5	0.9	0.7
NRSF1000	279	392	2129	2699	2489	2615	8.9	6.7	1.2	1.0
NRSF2000	1244	1663	9554	12150	13428	13830	10.8	8.3	1.4	1.1

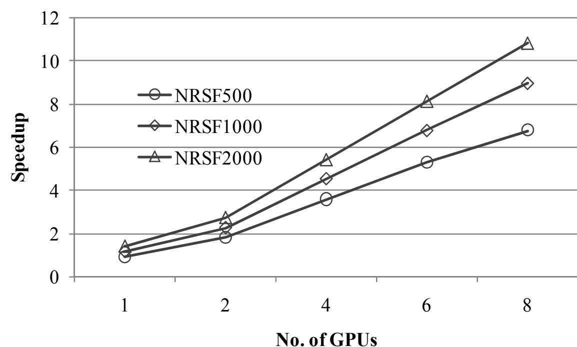

Figure 3. The speedups for the SPS stage of mCUDA-MEME on different numbers of GPUs.

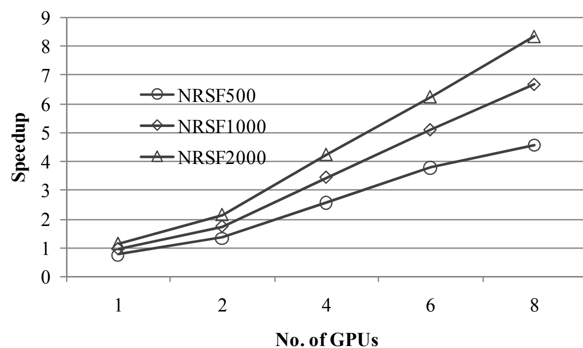

Figure 4. The speedups for the overall execution of mCUDA-MEME on different numbers of GPUs.

clusters. The genome-wide identification of TFBSs has been heavily constrained by the long execution time of motif discovery algorithms. Thus, we believe biologists will greatly benefit from the speedup of our algorithm.

V. CONCLUSION

In this paper, we have presented mCUDA-MEME, an ultrafast scalable many-core motif discovery algorithm for multiple GPUs. The use of hybrid CUDA, OpenMP and MPI programming models enables our algorithm to harness the powerful compute capability of GPU clusters. mCUDA-MEME achieves significant speedups for both the starting point search stage (and the overall execution) when benchmarked, using real datasets, against parallel MEME running on 32 CPU cores. Speedups of up to 1.4 (1.1) on a single GPU of a Fermi-based Tesla S2050 quad-GPU computing system and up to 10.8 (8.3) on the eight GPUs of a two Tesla S2050 system were observed. Furthermore, our algorithm shows good scalability with respect to dataset size and the number of GPUs. Since the long execution time of motif discovery algorithms has heavily constrained the genome-wide TFBS identification, we believe that biologists will benefit from the high-speed motif discovery of our algorithm on relatively inexpensive GPU clusters. As mentioned above, the load balancing policy of mCUDA-MEME is targeted towards homogenous GPU clusters, distributing workload symmetrically between GPUs. However, our current load balancing policy might not be able to work well for heterogeneous GPU clusters. Hence, designing a robust and appropriate load balancing policies for heterogeneous GPUs clusters is part of our future work. The source code of mCUDA-MEME and the benchmark datasets are available for download at https://sites.google.com/site/yongchaosoftware/mcuda-meme.

REFERENCES

[1] C. E. Lawrence, and A. A. Reilly, "An expectation maximization (EM) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences," Proteins, vol. 7, no. 1, 1990, pp. 41-51

[2] C. E. Lawrence, S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton, "Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment," Science, vol. 262, 1993, pp. 208–214

[3] P. Sumazin, G. Chen, N. Hata, A. D. Smith, T. Zhang, and M. Q. Zhang, "DWE: discriminating word enumerator," Bioinformatics, vol. 21, no. 1, 2005, pp. 31–38

[4] C. Sabatti, L. Rohlin, K. Lange, and J. C. Liao, "Vocabulon: a dictionary model approach for reconstruction and localization of transcription factor binding sites," Bioinformatics, vol. 21, no. 7, 2005, pp. 922–931

[5] T. L. Bailey, and C. Elkan, "Fitting a mixture model by expectation maximization to discover motifs in biopolymers," Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology, 1994, pp. 28-36

[6] V. Boeva, D. Surdez, N. Guillon, F. Tirode, A. P. Fejes, O. Delattre, and E. Barillot, "De novo motif identification improves the accuracy of predicting transcription factor binding sites in ChIP-Seq data analysis," Nucleic Acids Research, vol. 38, no.11, 2010, pp. e126

[7] W. N. Grundy., T. L. Bailey, and C. P. Elkan, "ParaMEME: a parallel implementation and a web interface for a DNA and protein motif discovery tool," Comput. Appl. Biosci., vol. 12, no. 4, 1996, pp. 303-310

[8] G. K. Sandve, M. Nedland, Ø. B. Syrstad, L. A. Eidsheim, O. Abul, and F. Drabløs, "Accelerating motif discovery: Motif matching on parallel hardware," LNCS, vol. 4175, 2006, pp. 197–206

[9] NVIDIA CUDA C programming guide version 3.2, http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[10] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," BMC Research Notes, vol. 2, no. 73, 2009

[11] Y. Liu, B. Schmidt, and D. L. Maskell, "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," BMC Research Notes, vol. 3, no. 93, 2010

[12] Y. Liu, B. Schmidt, and D. L. Maskell, "MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA," 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, 2009, pp. 121-128

[13] H. Shi, B. Schmidt, W. Liu, and W. Müller-Wittig, "A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware," J Comput Biol., vol. 17, no. 4, 2010, pp. 603-615

[14] Y. Liu, B. Schmidt, W. Liu, and D. L. Maskell, "CUDA-MEME: accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units," Pattern Recognition Letters, vol. 31, no. 14, 2010, pp. 2170 - 2177

[15] T. L. Bailey, M. Bodén, T. Whitington, and P. Machanick, "The value of position-specific priors in motif discovery using MEME, " BMC Bioinformatics, vol. 11, no. 179, 2010

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” ACM Queue, vol. 6, no. 2, 2008, pp. 40-53

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: a unified graphics and computing architecture,” IEEE Micro., vol. 28, no. 2, 2008, pp. 39-55

[18] Fermi: NVIDIA’s next generation cUDA compute architecture, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[19] OpenMP tutorial, https://computing.llnl.gov/tutorials/openMP

[20] MPI tutorial, https://computing.llnl.gov/tutorials/mpi