Exploring the Problem of GPU Programming for Data-Intensive Applications: A Case Study of Multiple Expectation Maximization for Motif Elicitation

Yuki Kitsukawa
Nagoya University
yuki@ertl.jp

Manato Hirabayashi
Nagoya University
hirabayashi@ertl.jp

Shinpei Kato
Nagoya University
shinpei@is.nagoya-u.ac.jp

Masato Edahiro
Nagoya University
eda@ertl.jp

ABSTRACT

Recently General-Purpose Computing on Graphics Processing Units (GPGPU) has been used to reduce the processing time of various applications, but the degree of acceleration by the Graphical Processing Unit (GPU) depends on the application. This study focuses on data analysis as an application example of GPGPU, specifically, the design and implementation of GPGPU computation libraries for data- intensive workloads. The effects of efficient memory allocation and high-speed read-only memories on the execution time are evaluated. In addition to employing a single GPU, the scalability using multiple GPUs is also evaluated. Compared to a Central Processing Unit (CPU) alone, the memory allocation method reduces the execution time for memory copies by approximately 60% when a GPU is used, while utilizing read-only memories results in an approximately 20% reduction in the overall program execution time. Moreover, expanding the number of GPUs from one to four reduces the execution time by approximately 10%.

Keywords

GPU, GPGPU, many-core, parallelization

1. INTRODUCTION

Improving processor performance by increasing the clock frequency is approaching its limit due to various issues, such as power consumption and heat generation. Hence, multi-/many-core technologies have been investigated using multiple cores within a chip. Graphical Processing Units (GPUs) are widely available in commercial products using the many-core technology. Although initially developed for graphics processing, the large-scale parallel computation capabilities of GPUs are currently used for general-purpose computing. This concept, which is called General-Purpose Computing on GPUs (GPGPU), has been applied to many fields.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SoICT '14 December 04 - 05 2014, Hanoi, Viet Nam
Copyright 2014 ACM 978-1-4503-2930-9/14/12 ...$15.00.
http://dx.doi.org/10.1145/2676585.2676616

This study focuses on data analysis applications as examples of GPGPU. We discuss the design and implementation of GPGPU computation libraries for data-intensive workloads with an emphasis on the performance evaluation of two types of acceleration approaches that take advantage of GPUs: memory allocation and texture memory utilization. The former collectively allocates the memory region required for computation by GPUs, while the latter is specifically designed for reading. In addition to evaluating a single GPU, the scalability of employing multiple GPUs is examined.

Because applications implementing acceleration approaches consume a large portion of their overall execution time on referencing and rewriting massive multiple arrays, the results herein should be applicable to similar data-intensive data analysis programs.

2. BASIC PLATFORMS

In addition to describing GPUs, this chapter introduces concepts and related studies of GPGPU and MEME. MEME is the genetic analysis program considered in this study.

2.1 GPU(Graphics Processing Unit)

A GPU was originally designed for image processing, and has several hundred to several thousand cores. Currently, GPUs are commonplace in desktop computers, laptop computers, and workstations. The computational capability and the number of GPU cores have been steadily increasing. GPUs are attracting attention as a next-generation system performance enhancement technology due to their parallel computation processing capability, ease of implementation, etc.

Figure 1 shows the performance improvements of CPUs and GPUs with GFLOPS[1] over time. Although the CPU performance has not increased dramatically since 2003 due to clock frequency limitations caused by power consumption and heat generation issues, the GPU performance has steadily increased. In 2013, the peak performance of the

[1]Giga Floating-point Operations Per Second is a performance indicator in terms of the number of floating-point operations in one second.

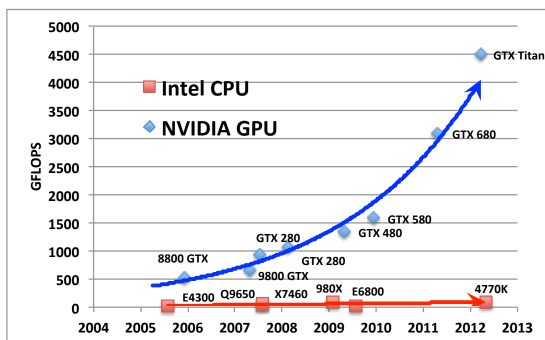

Figure 1: CPU and GPU computation performances over time

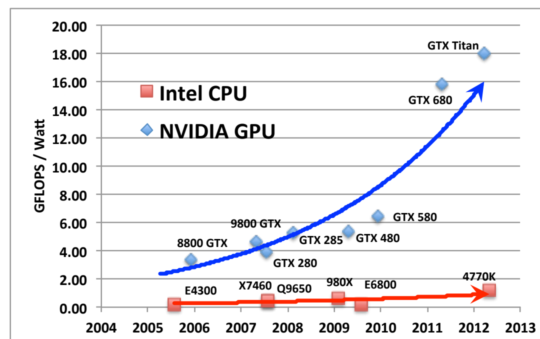

Figure 2: CPU and GPU computation performance per unit power

floating-point operations per second is 45 times that of a CPU.

Figure 2 shows changes in the computational performance per unit power for GPUs in recent years. The computational performance per unit power has been steadily increasing every year. It is foreseen that in addition to improving performance, minimizing power consumption will become an important challenge for future systems. Excessive power consumption generates a lot of heat, which requires increased cooling costs and performance restrictions. These are further reasons why GPUs are advantageous to improve system performance.

2.2 GPGPU(General-Purpose Computing on GPUs)

GPGPU is a technology that utilizes the high parallel computing capability of GPUs, which were originally designed for image processing, to applications in other areas. GPGPU is used in many fields, including linear algebra, biological information, databases, machine learning, and data mining. For example, a study by Pınar Muyan-Özçelik et al. demonstrated that employing GPUs to accelerate the deformable image registration algorithm in medical image processing resulted in a 55-fold increase in the program execution speed compared to the CPU-based implementation regardless of the dataset size[1]. Additionally, Mars, which is a MapReduce runtime system designed by Wenbin Fang that is accelerated by GPUs[2], generally achieves a higher speed than Phoenix, which is a conventional multi-core MapReduce runtime system; one experiment showed that Mars has as high as a 72-fold increase in speed compared to Phoenix.

Although the application development environment is gradually becoming diversified, a predominant one is Compute Unified Device Architecture (CUDA), which is a comprehensive development environment provided by NVIDIA. Because CUDA is based on C, its syntax is easy for C programmers to learn. The advent of CUDA has made GPU programming more accessible and accelerated the utilization of GPU for parallel numerical computations.

2.3 MEME

Multiple Expectation Maximization for Motif Elicitation (MEME) [3, 4] is a genetic analysis application developed by Timothy L. Bailey et al. MEME performs motif elicitation within DNA and protein sequences, where a motif refers to a small partial structure within an amino-acid sequence in DNA or a protein. There are two major purposes to elicitate motifs: to identify common motifs within a sequence and to infer the overall structure. In the former, functional similarity between different proteins is detected to infer a function in an unknown protein, while the latter helps determine the overall structure of a protein based on each motif. Motif elicitation is a data-intensive analysis with vast amounts of data, but a relatively small computation. MEME is becoming a de facto standard tool in genetic analysis. However, one disadvantage of MEME is that the analysis time increases exponentially with the input data size and analysis granularity.

3. RELATED WORK

3.1 mCUDA-MEME

mCUDA-MEME, which is developed by Yongchao Liu et al., is scalable algorithm for multiple GPUs using combination of CUDA, MPI and OpenMP.[5] Recently, genomic sequence data is growing rapidly in the field of biology. Accordingly, the importance of data analysis using computer is increasing. In this situation, to handle with big genomic data, mCUDA-MEME introduces two parallelization approaches, sequence-level and substring-level parallelization.

In the evaluation using NVIDIA GeForce GTX 280 GPU, it results in runtime speedups of 20 approximately against execution using CPU alone. In addition, compared with Para-MEME[6] which runs on 16 CPU cores of a high-performance workstation cluster or GPU-MEME[7] which is implemented based on OpenGL, mCUDA-MEME is more effective algorithm and it can process genomic sequences faster.

4. ACCELERATION APPROACHES FOR GPU PROGRAMS

Here we evaluate the performance of two high-speed approaches that take advantage of GPUs: memory allocation and texture memories. Memory allocation is used to collectively allocate the memory region required for GPU computations, while utilization of texture memories of GPUs is specifically for reading.

4.1 Collective Memory Allocation Method

There are cases where memory regions are temporarily allocated to store computational data and results. If regions

```
1  int **array;
2  array = (int **)malloc(height*sizeof(int *));
3  for(i = 0; i < height; i++){
4    array[i] = (int *)malloc(width*sizeof(int));
5  }
6  for(i = 0; i < width; i++){
7    free(array[i]);
8  }
9  free(array);
```

Figure 3: Example of discrete memory region allocation

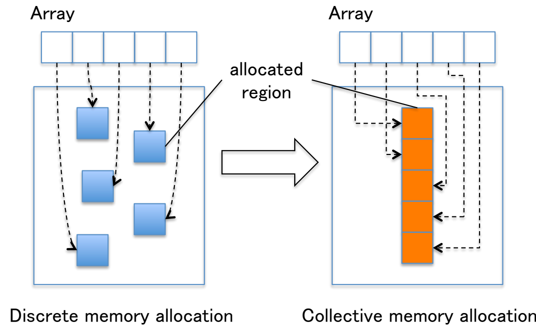

Figure 4: Collective memory allocation concept

are allocated using loops, as shown for the array in Figure 3, the regions exist in the address space discretely rather than continuously. Accelerating such a portion of an array using GPUs requires that the data in each discrete region be transferred, which results in numerous data transfers between the CPU and GPUs. Consequently, the communication cost is high and inefficient.

In contrast, collective memory allocation is an approach to continuously allocate all required memory regions (Figure 4). First, the required memory region is collectively allocated. Next, the allocated region is sectioned for each purpose and distributed from the top. Figure 5 shows an example of memory allocation using this concept. The referenced memory region addresses are sequential from the top of such an array. The data transfer to the GPUs is efficient because the entire memory from the top address of tmpArray (height * width * sizeof(int) byte in Figure 5), which contains necessary and sufficient data, is transferred all at once.

```
1  int **tmpArray = (int **)malloc(height*sizeof(int*));
2  /* consolidated alloation */
3  int *dst_tmpArray = (int *)malloc(height*width*sizeof(int));
4  /* distribute memory region */
5  unsigned long long int pointer = (unsigned long long int)
       dst_tmpArray;
6  for(int i = 0; i < height; i++){
7    tmpArray[i] = (int *)pointer;
8    pointer += (unsigned long long int)width*sizeof(int);
9  }
10 /* free memory region */
11 free(tmpArray[0]);
12 free(tmpArray);
```

Figure 5: Example of collective memory region allocation

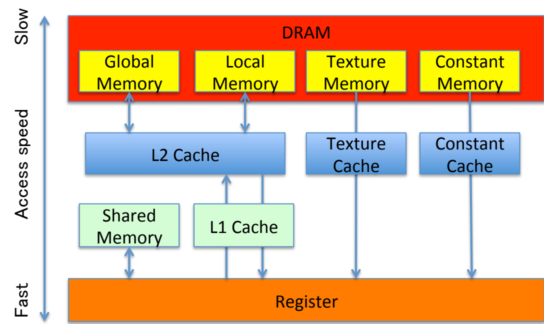

Figure 6: GPU memory model

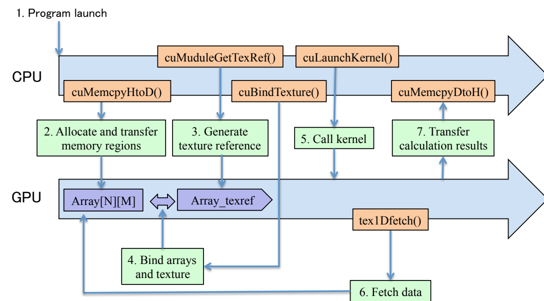

Figure 7: Program flow using texture memory

Collective memory allocation has other advantages. For example, if multiple GPU threads are referencing different memory addresses, the GPU accesses neighboring addresses to improve the efficiency (memory coalescing). Consequently, the processing efficiency is improved because the sequential memory region addresses used for computation can more easily utilize memory coalescing.

4.2 Texture Memory

In some GPU computation cases, the array itself does not change, but it is repeatedly referenced. Because neighboring threads in this type of processing are often referencing neighboring addresses, the performance can be improved by storing referenced arrays in a texture memory.

Figure 6 shows a GPU memory model. A texture memory is a type of read-only memory with its own cache on a chip, which reduces the number of references to an external DRAM and thereby, improves performance. Texture references are created on the texture memory to bind the references with the memory region in the GPU memory. Then by fetching the texture references using a GPU function, the memory contents bound to the references can be read. Figure 7 shows a program flow using a texture memory.

5. ACCELERATION USING MULTIPLE GPUS

This section discusses an implementation method for multiple GPU programs and a data synchronization method among multiple GPUs.

5.1 Multi-GPU Implementation Method

To implement multiple GPU programs, we employed pthread, which is a POSIX standard to perform parallel processing.

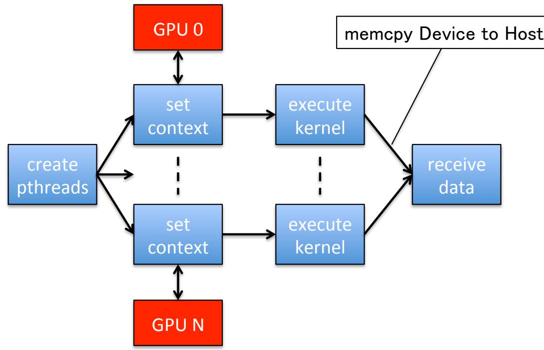

Figure 8: Schematic of a multi-GPU program using pthreads

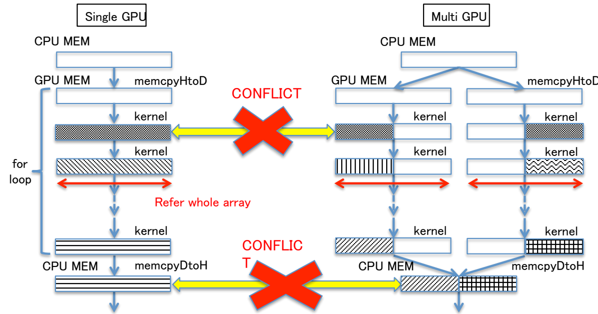

Figure 9: Example where data is not shared and computational integrity is not maintained

One GPU is assigned for each pthread for parallel processing. Figure 8 schematically diagrams a CPU generating pthreads, the connecting threads with the respective GPU, and the starting process.

After the GPUs are initialized, the CPU generates a pthread for each GPU. When an API is called to generate pthreads, pthreads are launched and each initiates a process in parallel. A processing series where each pthread performs is called a pthread function. The pthread function first calls a function to connect the pthreads and a processing series at each GPU (context) to provide a one-to-one correspondence between each pthread and context. After a thread and context are connected, the processing at each pthread is the same for all GPUs. First the region necessary for data storage is allocated in the GPU memory, and the data required to compute the region is copied from the CPU. Next the arguments for functions to be performed on the GPUs and the number of GPU threads are determined. Then the GPU functions are called to perform each computation within its tasked area using the data and other resources that were transferred to the GPU memory in the previous step. Upon completion of the GPU computation, the results within each tasked area are copied from the GPUs to the CPU. In this manner, the pthread function consolidates the computation results from each GPU.

5.2 Data Synchronization Method for Each Thread

Not sharing a change in one set of data in a thread with the other threads may result in a data mismatch. This may

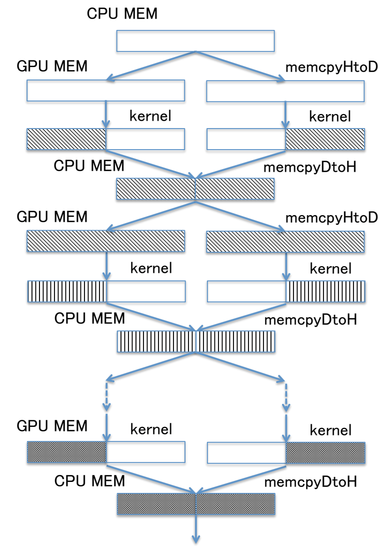

Figure 10: Method to copy the latest data to the CPU each time the data is updated

occur when the data being processed in one thread is referenced from another thread or when a set of data is processed simultaneously in multiple threads.

Figure 9 shows an example of a mechanism for such a mismatch. Herein a specific example is considered where a GPU function that includes overwriting an array is repeatedly called within a loop and multiple threads perform parallel processing of this GPU function. A mismatch can occur when the overwriting of the array is not shared with the other threads, and the latest values are not referenced in processing the next loop.

Our implementation includes a copy of the memory data to the CPU memory in order to reference the latest data and to maintain the computation integrity. When a single thread is involved, the results of the entire loop process are copied upon completion. However for multiple threads, all the threads copy the memory every time a GPU function within the loop is executed, ensuring that the latest data is referenced during processing within a loop (Figure 10).

One issue with this synchronization method is that overhead increases when there are multiple threads. Instead of copying the memory once from the GPU to the CPU at the end of the loop processing, the number of memory copies depends on the number of loops and executed functions.

6. EVALUATION

The above approach is implemented and evaluated for MEME, which is a genetic analysis application, as a data analysis example.

6.1 Evaluation Environment

Two machines were used in the evaluation: PC A and PC B. Table 1 summarizes their configurations and performances. Although both PC A and B run CentOS 64 bit, they have different versions, 6.3 and 6.5, respectively. The CUDA versions for PC A and B are 5.0 and 5.5, respectively.

PC A features two NVIDIA Tesla K20Xm cards. These cards are designed for scientific and engineering computa-

Table 1: Machine configurations and performances

PC	PC A	PC B
OS	CentOS 6.3	CentOS 6.5
CPU	Intel Xeon E5-2643	Intel Xeon E5-2687
Clock frequency [GHz]	3.30	3.40
CPU cores [cores]	4	8
CPU memory [GB]	64	64
GPU	NVIDIA Tesla K20Xm	NVIDIA GeForce GTX TITAN
GPU cards [cards]	2	4
GPU cores [cores]	2688	2688
Clock frequency [GHz]	0.73	0.88
GPU memory [GB]	6	6

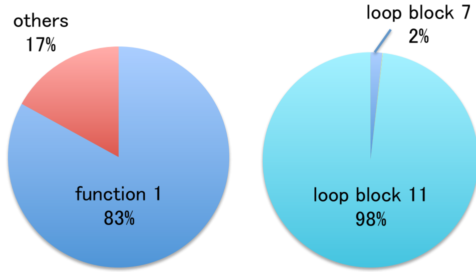

Figure 11: Left: Breakdown of the execution time for the entire MEME
Right: Breakdown of the execution time for function 1

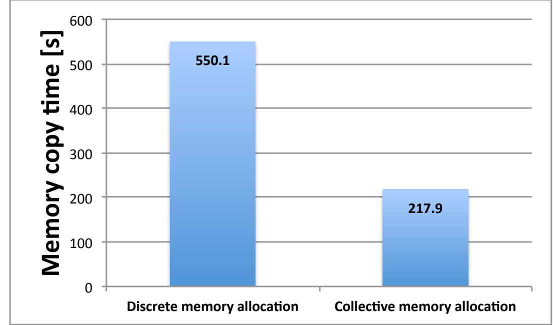

Figure 12: Execution time with and without collective memory allocation

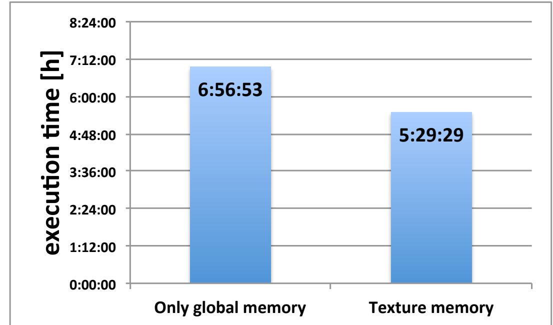

Figure 13: Execution time when only a global memory or a texture memory is used

tions, and have superior performance in double-precision floating-point computations and data reliability. PC B features four NVIDIA GeForce GTX Titan cards. These cards are designed to improve the graphics capability primarily for computer games due to their superior image processing performance.

6.2 High-Speed MEME Design

One drawback of MEME is that the analysis time increases exponentially with the input data size and analysis granularity. However, MEME allows users to set the analysis granularity for each partial structure in the genetic sequence through a parameter called maxsites.

Figure 11 (left) shows the breakdown of the overall program execution time. It should be noted that the multiple-double type arrays used for the array indexes and functions arguments are only referenced and not overwritten. Hence, these arrays are stored as texture memories. Figure 11 (right) shows the execution time for function 1, which contains 12 loops. Because locations with a large processing load (loop block 11) used discretely allocated memory regions, the process allocates memories collectively so that the CPU-GPU transfer can be performed simultaneously. The array values in function 1 are repeatedly overwritten in the GPU functions as part of parallel processing using multiple GPUs. Specifically, when N cards of GPUs are used for parallel processing, each GPU card overwrites the array values according to the number of array elements divided by N.

6.3 Experimental Results

6.3.1 Collective Memory Allocation

Figure 12 compares the execution time required for CPU-GPU memory copy with PC A when the memory region is allocated using discrete memory regions to that using a collective memory allocation with maxsites = 10000. Approximately 550 seconds are necessary to copy the entire memory using the discrete memory regions, whereas the same processing using collective memory allocation requires only approximately 220 seconds. This represents an approximately 2.5-fold increase in the memory copy speed.

6.3.2 Texture Memory

Figure 13 compares the execution time for the entire program with PC A using a global memory and to that using a texture memory with one GPU and maxsites = 10000. The

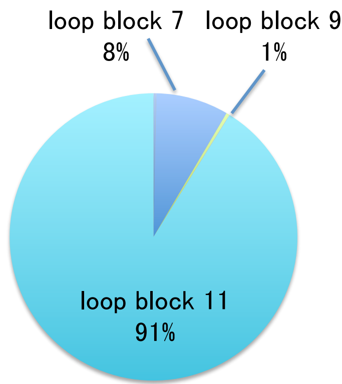

Figure 20: Breakdown of function 1 with four GPUs

global memory requires approximately 7 hours, but the execution time is reduced to approximately 5.5 hours using the texture memory. This is an approximately 20% increase in speed.

6.3.3 Multi-GPU Implementation

Figures 14 and Figure 15 plot the execution time for PC A and PC B with different numbers of GPUs [0 (CPU only) - 4 GPUs] versus of the maxsites values, respectively. Figures 16 - 19 plot the execution times of PC A and PC B versus the number of GPUs. Finally, Figure 20 shows the breakdown of execution time in each loop for function 1 when four GPUs are used.

Although an increase in the maxsites value significantly increases the execution time, the degree of acceleration using GPUs is greater for larger values of maxsites. Additionally, the execution time becomes shorter as the number of GPUs increases. Moreover, Figure 11 (right) and Figure 20 show that GPU-assisted processing of a process occupying a large portion of the entire program reduces the ratio of that process.

6.4 Discussion

Allocation of a collective memory reduces the amount of overhead required for a memory copy by approximately 60%, demonstrating that collective copying of memory significantly improves the performance. Because this method requires only minor modifications to the program code, the burden on programmers is small for a large gain in performance.

The use of texture memory improves the overall program execution speed by approximately 20%. Thus, texture memories can effectively optimize a program from a memory viewpoint. Because MEME has numerous elements that can bind to the texture memory and repeated referencing to a memory is performed by loops, texture memory can have a significant impact.

Compared to using one GPU, computations using multiple GPUs reduce the execution time, except when maxsites = 1000, showing that a multi-GPU is an effective means for acceleration. However, increasing the number of GPU cards by one results in a negligible improvement in speed. Employing four GPUs results in an approximately 10% speed improvement compared to one because the MEME program requires a large number of synchronizations among the threads. MEME performs a number of referencing and overwriting arrays in the loops, and the GPUs must be synchronized every time. This structure appears to have a limited benefit on the speed as the number of GPUs increases because the reduction in execution time due to scaling up the GPUs is canceled by the overhead required to synchronize the GPUs. This execution results show that with maxsites = 1000, the synchronization overhead at each thread is greater than the reduction in the execution time due to the parallel processing owing to the small array size to be processed, whereas with maxsites $\geq$ 3000, the reduction in the execution time due to the parallel processing is greater than the overhead for synchronization at each thread.

7. CONCLUSIONS

As a GPGPU application, implementation and evaluation were performed on two types of acceleration methods for data analysis applications. First, the collective memory allocation method achieved an approximately 60% reduction in the memory copy overhead between a CPU and GPUs. Second, the use of the texture memory, which is a read-only memory for GPUs, improved the overall program execution speed by approximately 20%. Furthermore, the execution time was reduced by approximately 10% when a single-GPU system was expanded to a multi-GPU system.

Although this study demonstrates that each method is an effective means to improve the computational performance in GPU programming, the effectiveness of each method differs. Future challenges include investigating data synchronization methods among multiple GPUs in order to realize a scale-up in computational performance in accordance with the number of GPU cards.

8. REFERENCES

[1] Pı nar Muyan-Özçelik, John D. Owens, Junyi Xia, and Sanjiv S. Samant. Fast deformable registration on the gpu: A cuda implementation of demons. In *ICCSA Workshops*, pages 223–233. IEEE Computer Society, 2008.

[2] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K. Govindaraju. Mars: Accelerating mapreduce with graphics processors. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):608–620, 2011.

[3] Timothy L. Bailey, Nadya Williams, Chris Misleh, and Wilfred W. Li. Meme: discovering and analyzing dna and protein sequence motifs. *Nucleic Acids Research*, 34(Web-Server-Issue):369–373, 2006.

[4] Timothy L. Bailey, Mikael Bodén, Fabian A. Buske, Martin C. Frith, Charles E. Grant, Luca Clementi, Jingyuan Ren, Wilfred W. Li, and William Stafford Noble. Meme suite: tools for motif discovery and searching. *Nucleic Acids Research*, 37(Web-Server-Issue):202–208, 2009.

[5] Yongchao Liu, Bertil Schmidt, Weiguo Liu, and Douglas L Maskell. Cuda–meme: Accelerating motif discovery in biological sequences using cuda-enabled graphics processing units. *Pattern Recognition Letters*, 31(14):2170–2177, 2010.

[6] William N. Grundy, Timothy L. Bailey, and Charles P. Elkan. Parameme: A parallel implementation and a web interface for a dna and protein motif discovery

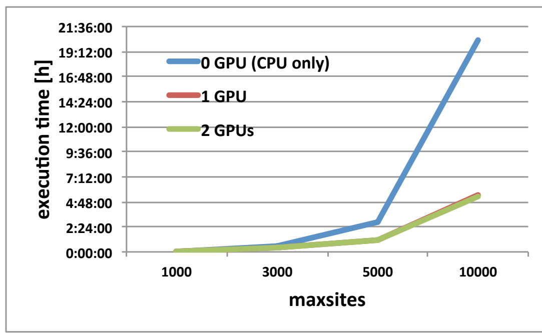

Figure 14: Execution time for PC A as a function of maxsites

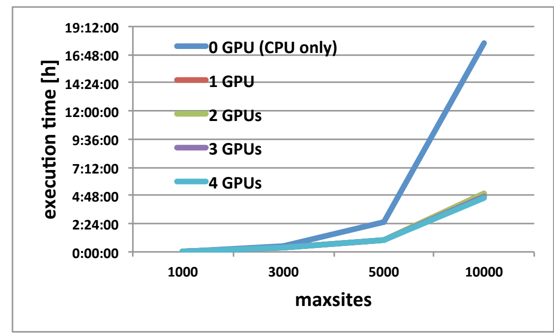

Figure 15: Execution time for PC B as a function of maxsites

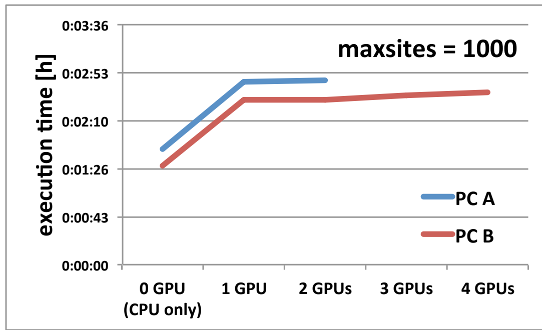

Figure 16: Execution time with PC A and B for maxsites = 1000 as a function of the number of GPUs

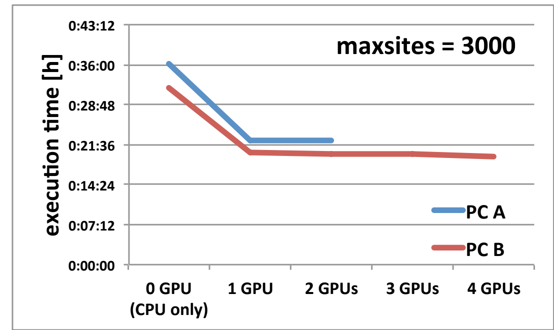

Figure 17: Execution time with PC A and B for maxsites = 3000 as a function of the number of GPUs

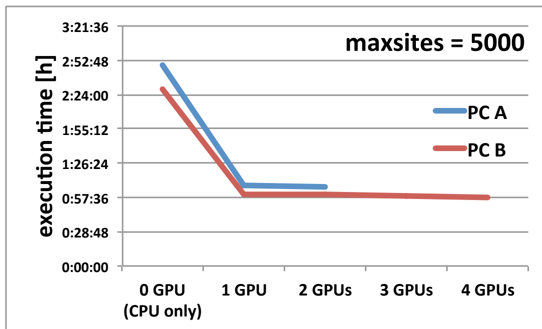

Figure 18: Execution time with PC A and B for maxsites = 5000 as a function of the number of GPUs

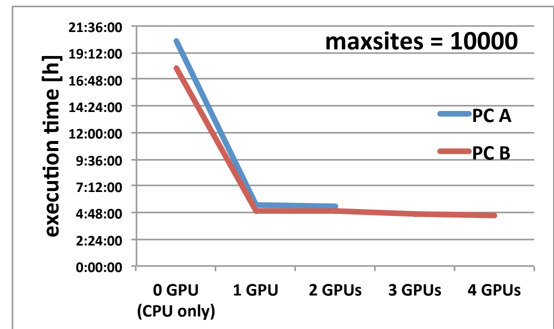

Figure 19: Execution time with PC A and B for maxsites = 10000 as a function of the number of GPUs

tool. *Computer Applications in the Biosciences*, 12:303–310, 1996.

[7] Chen Chen, Bertil Schmidt, Weiguo Liu, and Wolfgang MÃijller-Wittig. Gpu-meme: Using graphics hardware to accelerate motif finding in dna sequences. In *PRIB'08*, pages 448–459, 2008.