Many-core Architecture Oriented Parallel Algorithm Design for Computer Animation

Yong Cao

Department of Computer Science, Virginia Tech
yongcao@vt.edu

Abstract. Many-core architecture has become an emerging and widely adopted platform for parallel computing. Computer animation researches can harness this advance in high performance computing with better understanding of the architecture and careful consideration of several important parallel algorithm design issues, such as computation-to-core mapping, load balancing and algorithm design paradigms. In this paper, we use a set of algorithms in computer animation as the examples to illustrate these issues, and provide possible solutions for handling them. We have shown in our previous research projects that the proposed solutions can greatly enhance the performance of the parallel algorithms.

1 Introduction

The research of parallel computing has a long history and certainly is not new to the computer animation community. However, because of restricted access and lack of tools, computer animation researchers are reluctant to use supercomputers, on which most of parallel computing research focuses in the past few decades. In recent years, especially after the emergence of multi-core CPUs and many-core graphics processing units (GPUs), the situation has been changed. The computational power of a desktop machine is equivalent to the top supercomputers from ten years ago. Provided with these commoditized and easily accessible desktop supercomputers, all research fields in computer science are now embracing new opportunities for significant performance increase for their applications. At the same time, however, we also face a tremendous research challenge: *How to redesign all of our existing sequential algorithms towards a massive parallel architecture?*

Computer animation researchers, as part of the computer graphics community, may be ahead of some other research fields in terms of experiencing parallel algorithm design on GPUs, since GPUs were originally developed to enhance the performance of computer graphics applications. For example, the canonical smooth skinning algorithm for character animation has a standard GPU implementation using Vertex Shaders, and has been widely used in video games. However, after the release of the general proposal computing architecture for GPUs around 2006 (NVIDIA's GeForce 8 series), shader programming suddenly

lost advantages for parallel algorithm design for general purpose animation algorithms. People have also realized that the traditional parallel programming models for shared memory systems should not be directly applied towards this emerging many-core architecture. Instead, a set of parallel algorithm design issues, such as problem decomposition and load balancing, should be addressed during the algorithm design process for the computer animation algorithms.

In this paper, I first describe some key features of the current many-core parallel computing architecture, and elaborate on the trend of development for such architecture in the near future. Some important parallel algorithm design issues will then be discussed using some well-known animation algorithms as the examples. I, then, provide a set of solutions as the results of my recent work to address these design issues. At the end, the paper is concluded with some suggestions on the new research frontiers for parallel algorithm design in computer animation.

2 Parallel computing on many-core architecture

The idea of parallel computing and concurrent execution appeared during the 50's of the last century. The parallel computing research, especially on supercomputers, boomed in 70's and 80's. Many different parallel architectures were introduced, parallel algorithm design strategies were explored, and parallel algorithm analysis models were developed. However, many research areas, including computer animation, did not invest much into the parallel computing and algorithm design research, because the supercomputer resources were limited, and for their applications, a fast single processor are sufficient in most research cases. People were satisfied with the rate of the growth in computational power of CPUs stated in the Moores Law. A nearly doubled performance of the algorithm without any major revision of the code was a comfortable situation of most of applications.

During an interview in 2005, Gordon Moore stated that the law cannot last forever. Intel also gave a prediction that end will come soon due to quantum tunneling, which will flatten the increase rate of the density of transistors on an IC chip. Since then, there is a sudden change in the strategies for CPU development: no more clock speed increases, but more processing cores on a chip. The free lunch of automatic performance increase of an application is over. For any application, if the performance needs significant improvement, a major revision of the source code is necessary to transfer the algorithm from sequential execution to parallel execution.

The development of GPUs is ahead of such trend in CPUs, because, as a coprocessor for graphics processing, GPUs have already adopted a massive parallel architecture. However, before the appearance of general purpose GPUs, the application areas were very limited because the architecture can only be accessed by graphics programming libraries and, more importantly, GPUs were strictly designed for a small set of data-parallel algorithms.

The renaissance of GPU computing started with the release of NVIDIA's GeForce 8 series GPUs and general programming framework, CUDA. The researchers outside of the computer graphics community soon realized that the commoditized GPUs can be an significant accelerator for certainly data-parallel algorithms, and the parallel implementation of these algorithms is trivial when using C-language based CUDA programming framework. Many data parallel applications, such as image processing and physics-based simulation, have reported a 50X or more than 100X of performance speedup on GPUs. On the other hand, researchers also found that some other non-data-parallel algorithms, such as quick-sort, cannot achieve a large performance gain with a direct algorithm mapping towards GPUs. Sometimes, such algorithms will result a slow-down in performance on GPUs.

In the rest of this section, I first present some key features of the current generation many-core architectures. Then, I point out several important parallel algorithm design issues for such architecture.

2.1 Many-core architecture

The concept of many-core is derived from multi-core architecture in CPU hardware design. There is no standard definition for many-core architecture. A commonly accepted description of many-core is its comparison against multi-core: *the number of processing cores on the same IC chip is too large that a multi-core technologies will fail to efficient deliver instructions and data to all the cores.* Therefore, the processing cores of a many-core system are normally designed as a much simplified version of the cores in multi-core systems. There is no support for advanced processing controls, such as branch prediction, instruction pre-fetching, and micro-instruction pipelining.

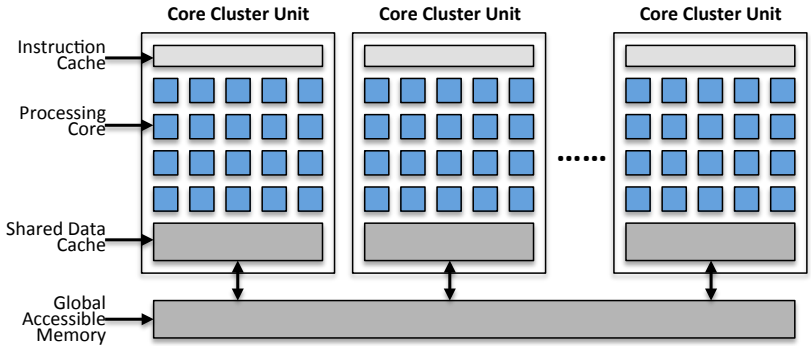

Fig. 1: A high-level overview of a many-core parallel architecture.

To manage a large number of cores in a many-core system, the cores are grouped together into higher level cluster units (called as stream multiprocessors in NVIDIA GPUs), as shown in Figure 1, so that the complexity for controlling these cores can be simplified. The control inside each cluster unit is as simple as the control of a vector processor: pure data parallel processing. There is normally only one instruction fetching unit for each cluster unit, and all the cores inside a cluster unit concurrently execute the same instruction on different data. For diverged branching instructions between the cores inside a cluster unit (e.g. core 1 takes one branch, core 2 takes another), the execution will simply be sequentialized.

The complex management of the parallel execution are not focused on the core level, but on the cluster unit level, where asynchronized communication and task-parallel execution are supported. Such high-level control is essential for developing efficient parallel algorithms, because overlapped computation between processing cores can be managed.

Many-core system uses a shared memory architecture for communication between processing cores. A shared data cache inside a cluster unit is used for the communications between the cores inside a cluster unit. A global accessible memory, normally a slower DDR memory, is used for communication between the cores on different cluster units. Since both shared data cache and global memory can be accessed concurrently, some parallel access designs are normally applied to the memory hardware. Such memory architecture design normally favors a large memory bandwidth, but not a short latency.

2.2 Algorithm Design Issues for Many-core Architecture

Due to the architecture design of the many-core architecture, especially the parallel memory access hardware, a certain type of algorithms, data parallel algorithms, execute much faster than the others. The common characteristics of a data-parallel algorithm include SIMD execution, little data dependency, and few branches. It is often reported that a large performance speedup can be achieved when porting a data-parallel algorithm from a single-core computing architecture to a many-core parallel architecture.

In computer animation, we have some algorithms express the features of data-parallel computation. For example, video-based tracking and image processing algorithms are widely used character animation. In one of our early projects [11], as shown in Figure 2, we analyzed a parallel video-based character tracking algorithm, called *Vector Coherence Mapping* (VCM), and implemented towards NVIDIA GPUs using CUDA programming framework. VCM includes three major processing steps: interest point extraction, normal correlation map (NCM) computation, and VCM computation. The operations used in these steps are mostly data parallel algorithms, such as image convolution, sub-image correlation, and image accumulation. By accelerating these operations on GPUs, we

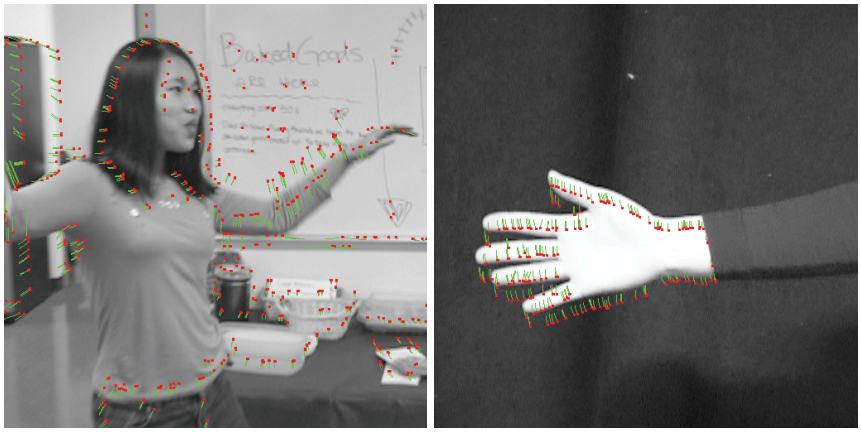

Fig. 2: The processing results of the GPU-accelerated motion tracking algorithm, VCM. Left: Girl dancing with camera zooming in. Right: Hand moving up.

had an over 40 times of performance speedup compared against a CPU implementation.

However, data-parallel algorithms only represent a small portion of the algorithms used in computer animation. It has been shown that, without careful algorithm design and optimization, the many-core implementation of task parallel algorithms, such as quick-sort, does not guaranty a large performance increase on GPUs [7, 15]. Intel also pointed out that, only applying standard optimization strategies on GPUs can only get an average of 2.5X speedup compared with the implementations on a Intel's CPU [13].

Designing an algorithm towards many-core architecture is not a simple Implementation and porting work. The process involves the consideration of many different parallel computing issues. Often, the newly developed parallel algorithm is a complete transformation of its sequential counterpart. For computer animation research, there are a few important parallel algorithm design issues, as listed below, need to be considered carefully when a sequential algorithm is optimized on a many-core architecture.

1. *Problem decomposition and resource utilization.*

2. *Load balancing.*

3. *Algorithm design paradigm.*

In the next three sections, I will discuss these issues in detailed by using some example algorithms in computer animation.

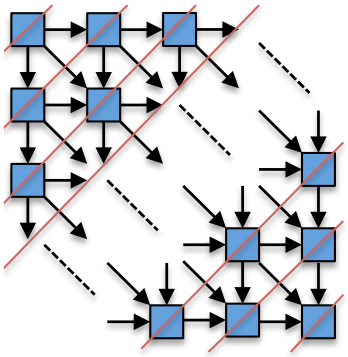

Fig. 3: The data dependency graph for an example dynamic programming algorithm. The diagonal lines in orange indicate the sweeping frontier for each computation step. All the nodes along these sweeping diagonal lines can be executed in parallel.

3 Problem decomposition and resource utilization

The concurrent execution of different tasks among all available computational resources is the key issue in parallel computing. If the resources, especially the processing cores, are under-utilized, the performance loss will occur. In parallel algorithm design, the issue is normally addressed by problem decomposition. In general, if a processing architecture has N cores, the problem should be decomposed into more than N (often multiple of N) sub-tasks. Each processing core is assigned with at least one sub-task, and no core is left idle.

In some scenarios, the size of problem is dynamically changing and unpredictable. Therefore, a static problem decomposition scheme can not optimize the final number of sub-tasks for better resource utilization.

For example, as being used in a motion graph algorithm [2, 3], dynamic programming usually features a "slow-start" initial processing phase. In Figure 3, we illustrate the data dependencies for all the computation of an example dynamic programming algorithm. Because the dependency, the concurrent computation can only occur among the sub-tasks among the diagonal lines, starting at the top-left corner. We can sweep the computation diagonally from top-left to bottom right, each time executing the sub-tasks along a sweeping line in parallel. It is obvious that, the concurrently executed sub-tasks will increase until reaching the half-way point of the whole computation, and will decrease after. This is a typical slow-start and slow-end problem. The question is how to increase the resource utilization rate at the beginning and the end of the computation, where only very few sub-tasks are available for N cores to finish.

The solution to the resource under-utilization problem is to apply a dynamic problem decomposition scheme based on the problem size. It is also call adaptive computation-to-core mapping. The main idea is to adjust the granularity of the problem decomposition in an adaptive fashion: when more processing cores are available (resource under-utilization), the sub-tasks can be decomposed into even smaller tasks so that the total number of sub-tasks for parallel execution is increased and all the cores will be utilized. However, we also do not want to create too many sub-tasks because of the high management cost.

Some other widely used algorithms in computer animation also expressed a slow-start or slow-end feature, such as breadth-first graph search and multi-resolution analysis. Adaptive computation-to-core mapping can be directly used in these algorithms. In one of our previous work for parallel temporal data mining [6], we have developed a hybrid solution to incorporate two different computation-to-core mapping schemes, and got a result of 263% performance increase when compared with a standard many-core implementation.

4 Load balancing

In some data-parallel algorithms and most task-parallel algorithms, such as animation or simulation Level Of Detail (LOD), the concurrently executed sub-tasks have different workload. Some sub-tasks can finish much earlier than the others. It can cause a significant performance loss because a processing core for executing a early completed sub-task has to wait until all other cores finish their tasks before continuing with the next task.

In many-core architecture, load-balancing problem can result in significant performance loss. The main cause of workload imbalance is due to the branching statements in the algorithm, which many-core architecture can not handle efficiently. If a branching statement causes a diverged execution between two cores in a core cluster unit, as shown in Figure 1, the execution of all diverged instructions has to be sequentialized, because a core cluster unit only has one instruction dispatch unit. In addition, if one branch takes much long to finish than another, non-balanced workload will cause further performance loss. For example, in agent-based crowd simulation algorithms, all agents are simulated concurrently. But there are some types of agents, such as leader agents, can take much heavier workload than the others.

Load balancing problem has been well studied in parallel computing research before many-core parallel architecture. The common strategies to resolve the problem, as listed below, can also be applied to many-core architecture.

1. Divide the subtasks into smaller ones to eliminate significant workload difference.
2. Group subtasks according to the workload, and assign the subtasks with similar amount of workload to the same core cluster unit.

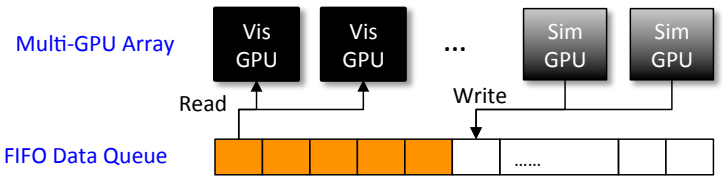

Fig. 4: A distributed task scheduling framework for balancing the workload between visualization and simulation on a multi-GPU architecture.

3. Apply a distributed task scheduler for each processing core, so that after the completion of one subtask, it can immediately fetch the next subtask from a task pool.

The first two solutions are based on the static analysis of the algorithm. The last one, which is a run-time solution and more general than first two, is a very challenging solution one for many-core architectures. Distributed task control is commonly used strategy in supercomputers for task scheduling, where each processing unit coordinates with a global task pool to schedule its own execution of tasks. The strategy is an efficient solution for balancing the workload between core cluster units, and multiple many-core devices. For example, in one of our previous work [10], as shown in Figure 4, we used a data queue as the central task pool to schedule and balance the computation on multiple GPUs. A simulation task is assigned to a GPU when there is an empty slot in the queue. A visualization task reads the data queue and is assigned to a GPU to visualize the simulation result. The task scheduling criteria is based on the status of the data queue and performance history.

However, within a core cluster unit, such strategy will results in more divergent branching instructions, causing the sequentialized execution in the unit. To circumvent the problem in many-core architecture, the distributed task scheduler should be combined with subtask grouping methods to reduce divergent branching and load imbalance inside a core cluster unit. Since task scheduling inside a core cluster unit does not provide any meaningful performance gain, the subtasks with similar workload and the same instructions are sorted and grouped together before submitted to the same core cluster unit, as shown in Figure 5.

5 Algorithm design paradigm

It is a creative process to brainstorm new ideas for parallelize existing sequential algorithms. It is even more challenging to design entirely new parallel algorithms for certain problems. Such process should be guided by a set of design paradigms which are used to shape the mind of the design. For example, in traditional sequential algorithm design, when a search-based problem is presented, we instantly refer to some solution templates, such as divide-and-conquer, depth first

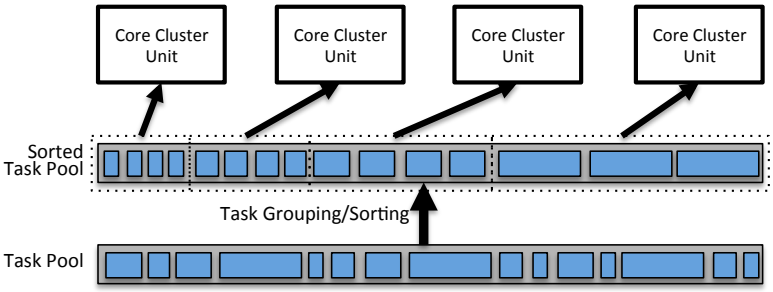

Fig. 5: A revised distributed task scheduling framework for many-core architectures, where the tasks in the task-pool are sorted first. The tasks with similar workload are submitted to core cluster unit for parallel execution.

search, and generic algorithms. Such design paradigms greatly simplify the process of algorithm design, and also enables the parallel implementation of some algorithm libraries, including Standard Template Library (STL).

In parallel algorithm design, there are some well-known design paradigms, such as Map-Reduce [8], which has been widely adopted in many-core algorithm design [16, 4, 9]. Since many-core system is an emerging architecture, we are expecting more parallel design paradigms being developed for this architecture.

In one of our previous work [14, 5], we developed a very efficient parallel processing paradigm, which can be used in many computer animation algorithms. The paradigm, called *deferred computing*, divides the overall copmutation into several stages. The first several stages are for data preparation or analysis, and can be significantly accelerated in many-core architecture. The major computation of the problem is deferred to the last stage. By using the results from the preparation and analysis stages, the computation of the last stage can be far more efficient than before. In such deferred computing scheme, the overall computation for the problem is often larger than the original algorithm. However, due to the execution time saved at the last stage, the overall performance is actually increased.

To give an example, let us consider a problem to eliminate none-qualified patterns from a large array of candidate patterns [5]. Since the qualifying process for each pattern is the same for all candidate patterns, we can simply parallelize the processing of each pattern on each core. However, the qualifying process is very complex. The execution of the process on each core is not very efficient, due to some memory accessing constraints and the branching instructions in the process. We divide the qualifying processing into two passes, as shown in Figure 6. The first pass uses simplified constrains to process the patterns in parallel, where most of the memory constraints and branching instructions are removed in this pass. We found that the first pass can eliminate more than 90% of none-qualified patterns with its less constrained process, and can be executed much efficiently

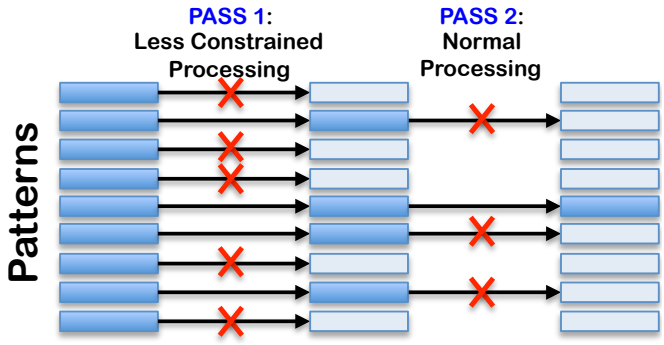

Fig. 6: An example of deferred computing: Pattern Elimination. In the first pass, most of the none-qualified patterns are eliminated by a less constrained and simple process. In the second pass, much less patterns needs to be processed by a complex elimination step.

on many-core architectures. The second pass operates on the undetermined patterns from the first pass using a normal qualifying process. Since the left-over patterns for the second pass is very few and the first pass is very efficient, the overall performance of the two-pass elimination approach can be more than two times faster than the original one-pass algorithm.

6 Future Development Directions

Many-core architecture is currently in a rapid development era. Many vendors are proposing and releasing new many-core based products. GPU venders, including NVIDIA and ATI, are aiming at more processing cores and complex control logic to enable task-parallel processing. For example, NVIDIA's recent GPU architecture, code name "Fermi", starts to support up to 16 different kernel functions in a GPU simultaneously. Such feature allows the programmers to parallelize different instructions among the core cluster units.

Another important advance in many-core architecture has been proposed by CPU venders, including both Intel and AMD. In their recent release, Intel's Sandy Bridge CPUs [12] and AMD's Fusion APUs [1] both focus on a tightly integrated heterogeneous system, where a multi-core CPU, a many-core GPU and the memory controls (or L1 cache) are put on the same IC chip. Such design significantly reduces the communication overhead between the CPU and GPU, which was a large performance bottleneck for the most GPU-based parallel computing applications.

The advance also casts a spotlight on a already popular research direction for high performance computing, hybrid computing, where computational tasks are

co-scheduled among a heterogeneous computing architecture. In hybrid computing, problems are analyzed and decomposed into sub-tasks based on their computational profiles. The sub-tasks suitable for data parallel processing are assigned to the GPU, and the sub-tasks suitable for task parallel processing are assigned to the CPU. Some central and distributed control is applied to synchronize the processing among these sub-tasks and computational resources. With a much improved architecture for inter-communication between CPUs and GPUs, hybrid computing research is embracing a booming period.

In computer animation researches, it is often that we have a very complex system including a variety of algorithms, which express totally different computational profiles. Therefore, the ability of concurrently executing these algorithms on their desired devices in a heterogeneous architecture will bring a significant performance gain. However, in my opinion, we are still in the stone-age for hybrid computing for computer animation applications. We need to focus on the algorithm design issues towards many-core architecture and hybrid computing.

7 Conclusion

Computer animation, like the other application areas in computer science, is facing the new era of parallel computing. With the rapid development of many-core architectures, such as GPUs, the research in parallel algorithm design for computer animation has already fallen behind. Given that almost every computer has adopted a parallel processing architecture, there is no coming back to the world of sequential algorithm design. In this paper, we have discussed several important parallel computing design issues for many-core architectures, including resource utilization, load balancing and algorithm design paradigms. In my previous research, I have shown that careful consideration of these issues can greatly enhance the performance of the parallel algorithms.

Parallel algorithm design is not only for the scholars in the area of computing theory or high performance computing. It is also important for computer animation community to evaluate the algorithms in our applications, to analyze the time complexity of a proposed algorithm, and to discuss the scalability issues for a parallel implementation. We need to develop a set of software frameworks to facilitate the parallel implementation of computer animation applications based on such effort. We will be able to handle much larger scale problems and significantly increase the performance of the computer animation applications.

References

1. AMD: Fusion family of apus, `http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx`
2. Arikan, O., Forsyth, D.A.: Interactive motion generation from examples. ACM Trans. Graph. 21(3), 483–490 (2002)

3. Arikan, O., Forsyth, D.A., O'Brien, J.F.: Motion synthesis from annotations. In: ACM SIGGRAPH 2003 Papers. pp. 402–408. SIGGRAPH '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/1201775.882284
4. Bakkum, P., Skadron, K.: Accelerating sql database operations on a gpu with cuda. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 94–103. GPGPU '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1735688.1735706
5. Cao, Y., Patnaik, D., Ponce, S., Archuleta, J., Butler, P., chun Feng, W., Ramakrishnan, N.: Towards chip-on-chip neuroscience: Fast mining of frequent episodes using graphics processors. Tech. rep., arXiv.org (2009)
6. Cao, Y., Patnaik, D., Ponce, S., Archuleta, J., Butler, P., chun Feng, W., Ramakrishnan, N.: Towards chip-on-chip neuroscience: Fast mining of neuronal spike streams using graphics hardware. In: CF '10: Proceedings of the 7th ACM international conference on Computing frontiers. pp. 1–10. No. 978-1-4503-0044-5, ACM, Bertinoro, Italy (May 17 - 19 2010)
7. Cederman, D., Tsigas, P.: Gpu-quicksort: A practical quicksort algorithm for graphics processors. J. Exp. Algorithmics 14, 4:1.4–4:1.24 (January 2010), http://doi.acm.org/10.1145/1498698.1564500
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51, 107–113 (January 2008), http://doi.acm.org/10.1145/1327452.1327492
9. Fang, W., He, B., Luo, Q., Govindaraju, N.K.: Mars: Accelerating mapreduce with graphics processors. IEEE Trans. Parallel Distrib. Syst. 22, 608–620 (April 2011), http://dx.doi.org/10.1109/TPDS.2010.158
10. Hagan, R., Cao, Y.: Multi-gpu load balancing for in-situ visualization. In: To Appear in the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (2011)
11. Huang, J., Ponce, S., Park, S.I., Cao, Y., Quek, F.: Gpu-accelerated computation for robust motion tracking using the cuda framework. In: VIE 2008 - The 5th IET Visual Information Engineering 2008 Conference. pp. 437–442 (July 29 - August 1 2008)
12. Intel: Sandy bridge architecture, http://www.intel.com/content/www/us/en/processors/core/core-i5-processor.html
13. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In: Proceedings of the 37th annual international symposium on Computer architecture. pp. 451–460. ISCA '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1815961.1816021
14. Patnaik, D., Ponce, S.P., Cao, Y., Ramakrishnan, N.: Accelerator-oriented algorithm transformation for temporal data mining. vol. 0, pp. 93–100. IEEE Computer Society, Los Alamitos, CA, USA (2009)
15. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. pp. 97–106. GH '07, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2007), http://dl.acm.org/citation.cfm?id=1280094.1280110
16. Stuart, J.A., Chen, C.K., Ma, K.L., Owens, J.D.: Multi-gpu volume rendering using mapreduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. pp. 841–848. HPDC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1851476.1851597