Noname manuscript No.
(will be inserted by the editor)

Multi-GPU Load Balancing for In-Situ Simulation and Visualization

Yong Cao · Robert Hagan

Received: date / Accepted: date

Abstract Multiple-GPU systems have become ubiquitously available due to their support of massive parallel computing and more device memory for large scale problems. Such systems are ideal for *In-Situ* visualization applications, which require significant computational power for concurrent execution of simulation and visualization. While pipelining based parallel computing scheme overlaps the execution of simulation and rendering among multiple GPUs, workload imbalance can cause substantial performance loss in such parallel configuration. The aim of this paper is to research on the memory management and scheduling issues in the multi-GPU environment, in order to balance the workload between this two-stage pipeline execution. We first propose a data-driven load balancing scheme which takes into account of some important performance factors for scientific simulation and rendering, such as the number of iterations for the simulation and the rendering resolution. As an improvement to this scheduling method, we also introduce a dynamic load balancing approach that can automatically adjust the workload changes at runtime to achieve better load balancing results. This approach is based on an idea to analytically approximate the execution time difference between the simulation and the rendering by using fullness of the synchronization data buffer. We have evaluated our approaches on an eight-GPU system and showed significant performance improvement.

.

Keywords Load balancing · In-Situ visualization · Multi-GPU computing · Dynamic scheduling

Yong Cao
Virginia Tech, Computer Science Department
Blacksburg, VA, USA
E-mail: yongcao@vt.edu

1 Introduction

Graphics processing unit (GPU) offers massively parallel processing that can greatly accelerate a variety of data parallel applications. Use of multiple GPUs leads to even greater performance gains by executing multiple tasks on different GPUs simultaneously. This provides an opportunity for handling larger scale problems that require more memory and computational resources. In GPU computing, especially multi-GPU approaches, workload imbalance is one of the major issues for performance loss. This is because the data partition or task partition in the parallel configurations is not usually even. Effective load balancing can greatly increase utilization and performance in a multi-GPU environment by distributing workloads evenly.

There are two general approaches for multi-GPU computing: *data parallel* and *pipelining*. The data parallel approach distributes data among available processing units. The pipelining approach splits the processing of data into stages for different processors to handle. In this paper, we focuses on the load balancing problem of the pipelining approach since data parallel approaches that equally distribute data require little effort in load balancing. Specifically, we are interested in the *In-Situ Visualization* applications, which have become popular in the era of exa-scale (even peta-scale) computing because of its support of immediate visual feedback of the simulation for early error detection and computational steering. In-situ visualization includes two computational stages for pipelining: simulation and rendering. At each simulation time frame, the simulation is executed and followed with the rendering of simulation result of the current time frame, before moving to the next simulation-rendering cycle for the next time frame. A data buffer of multiple simulation time frames is used to communicate and synchronize the pipeline execution so that the simulation of the current time frame and the rendering of the previous time frame can be overlapped, as shown in Figure 1. However, if the workload

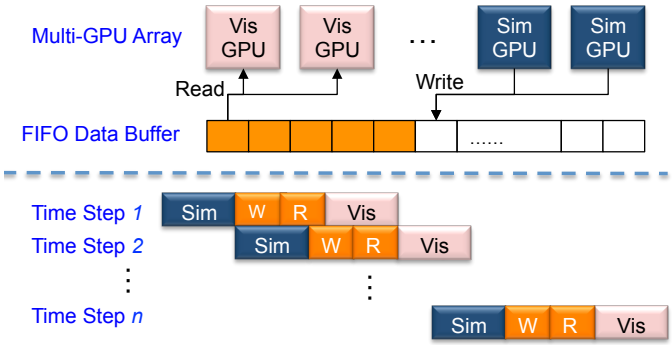


Fig. 1 Top figure shows the multi-GPU configuration used in an in-situ visualization application, where simulation GPUs and visualization GPUs synchronize the work in a pipeline fashion with the support of a FIFO data buffer. Bottom figure illustrates the pipelining stages in the application.

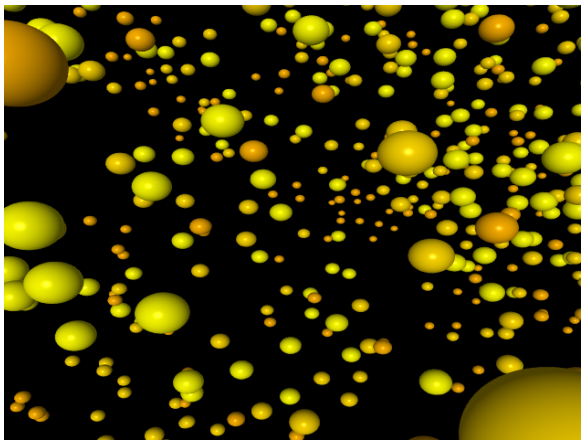


Fig. 2 A rendered image of the N-body simulation, which is the case-study application for this paper. The colors of the particles represent different masses that affect the simulation.

between the simulation and the rendering is not equal, a "bulb" will be created inside the pipeline causing performance loss.

Load balancing in the multi-GPU environments introduces special issues compared with multi-threaded CPU environments. The most significant one is the high communication cost of the central data buffer. GPUs have limited device memory. Therefore, the data buffer has to be resided at the large host memory. The data write (simulation stage) and data read (rendering stage) to the host buffer from the GPU can cause significant performance lost because the PCI-express bus which connects the host memory and the GPU is several magnitude slower than direct GPU memory access. Another challenge in load balancing for in-situ visualization applications is that the workload of simulation and rendering is constantly changing between time frames. A static load balancing scheme without the consideration of these performance factors, including simulation time step and rendering resolution, will fail to evenly distribute the tasks during the long period of scientific simulation.

To address these issues, we propose two methods of load balancing suitable for different applications, as the main contributions of this paper. First, based on our previous work [8], we introduce a *data-driven load balancing* approach, which balances the workload based on the past performance and improve utilization by determining the best configuration for a given application beforehand. However, this method requires the collection of various performance data in a pre-analysis stage to determine the best workload distribution for a given application. Furthermore, this method does not adapt to unpredicted changes at runtime for an application. Therefore, we propose the second approach, *dynamic load balancing*, which can adapt workload distribution to take into account changes at runtime without data preparation. Without losing generality, we evaluate our approaches on an in-situ visualization application with a gravitational N-body simulation and ray-tracing based

rendering algorithm. The N-body simulation is based on the algorithm of Nyland et al. [12]. The simulation result of a certain time frame is rendered as shown in Figure 2. The interactive application allows the user to navigate the scene by rotating and zooming using the mouse.

The rest of the paper is organized as follows. We first describe the related research to this work in Section 2. Then, we provide a formal problem statement with some background information in Section 3. In Section 4, we introduce our data-driven load balancing approach and provide the performance evaluation of the approach. In Section 5, we propose the dynamic load balancing approach and show the performance results. We conclude our work and describe the future research directions in Section 6.

2 Related Work

Several areas of past research have recently addressed issues related to the load balancing. We will present some researches for GPU-based and multi-GPU based load balancing work.

2.1 GPU-based Load Balancing

Cederman et al. show results comparing GPU load balancing methods applied to octree construction [5]. Their methods tested include centralized blocking task queue, centralized non-blocking task queue, centralized static task list, and task stealing. They find that task stealing performs the best out of these. Heirich et al. compare load balancing techniques for ray tracing, and they find that static load balancing results in poor performance [9]. They apply a new method based on a diffusion model with static randomization. Similar to our findings with a visualization application, a dynamic method can adapt to additional applications that static load balancing cannot adjust to. Fang et al. propose a method for scheduling on multicore systems to account for data locality [6]. This can help to reduce communication in order to improve performance. They test a simulation on several benchmarks and show that their method can improve performance by taking into account scheduling on cores to decrease latency. Jimènez et al. present a scheduler for GPU-CPU systems that uses past performance in order to schedule based on past performance [10]. They achieve a significant speedup using adaptive runtime scheduling. Like our data-driven approach, they take into account past performance in order to predict configurations for use. Joselli et al. present a novel physics engine that uses automatic distribution of processes between the CPU and GPU to improve performance [11]. It proposes a dynamic scheduling scheme that approximates the load on the CPU and GPU, and uses this to distribute physics computations.

Other works have presented work stealing as a method for load balancing. Acar et al. present an analysis of work stealing methods and how their data

locality affects performance [1]. They provide a new work stealing method based on locality that improves over past methods. This method works by having certain processors take on work from processors that are overworked. This allows for a significant performance speedup. Like our work, they focus on load balancing, but do not focus on a pipelined approach with a general framework as ours does. Agrawal et al. propose A-steal, a load balancing algorithm based on work stealing for CPU systems [2]. The method uses past performance data as the algorithm runs in order to schedule tasks. Statistical techniques are used to determine the optimal work stealing configuration. Similar to our data-driven load balancing method, they present a technique that uses past performance to apply load balancing. However, their technique could be extended in future work to account for more dynamic changes in GPU applications at runtime as our work does. Arora1 et al. present work on load balancing on CPUs [3]. They provide a non-blocking work stealing algorithm with a two-level scheduler, where the two levels are used to partition the scheduling of the system. This new method introduces a way to load balance by reducing synchronization costs. Our work also deals with scheduling and synchronization for load balancing, but focuses on a pipelined approach for multi-GPU computing.

While these works provide methods for load balancing, they do not focus on multi-GPU load balancing using a pipelining approach as our method does.

2.2 Multi-GPU Load Balancing

Fogal et al. implement GPU cluster volume rendering that uses load balancing for rendering massive datasets [7]. They present a brick-based partitioning method that distributes data to each GPU in order to be processed, where the final image is composited from these separate rendered images. However, their load balancing does not use a pipelined approach and is limited to a specific volume rendering method. Monfort et al. use load balancing methods in a game engine for rendering that use split screen and alternate screen partitioning [13]. Split screen load balancing involves partitioning a single frame for multiple GPUs, while alternate frame rendering distributes single consecutive frames to GPUs to be rendered. They find that alternate frame rendering leads to better performance due to fewer data dependencies, and they present a hybrid method that improves performance by load balancing both within and across rendered frames. Binotto et al. present work in load balancing in a CFD simulation application [4]. They use both an initial static analysis followed by adaptive load balancing based on various factors including performance results. But these only focused on the top-down data partitioning for load balancing and have not researched on the pipelining approach with multiple stages of computation.

Hagan et al. present a data-driven method for load balancing based on input parameters for in-situ visualization applications [8]. By taking into account past performance for different combinations of supersampling, simulation, and

dataset size, the best configuration can be chosen. This work is presented in this paper and is extended with a new dynamic load balancing approach, which also takes into account unpredicted changes at runtime.

3 Problem Statement

As we discussed in the previous sections, our focus is on a two-stage pipeline execution of in-situ visualization applications. The computation stages are the *simulation* and followed by the *rendering*. The two-stage pipeline has to be executed on a sequence of simulation time frames. Therefore, if multiple computation units are available, the simulation and rendering can be executed concurrently where the simulation of the current frame can be overlapped with the rendering of the previous frame, as illustrated in the bottom picture of Figure 1. This is achieved by a data buffer into which is written by the simulation process and read by the rendering process. To avoid the read-write conflict by the two processes, double-buffer is introduced to enable simultaneous read and write. In double-buffer, two buffer frames are introduced, each of which can hold the simulation data of one time frame.

However, if the workload of the simulation is larger than the rendering, then the rendering process has to wait for the simulation to write a frame of data to the buffer. If the rendering workload is larger than the simulation, the double buffer frames will be filled soon, and the simulation process has to wait until the rendering process consumes a buffer frame. In theory, if infinite number of buffer frames are available, there will be no waiting time for the simulation process. In practice, we can only provide a fix number of buffer frames, therefore, the waiting time is unavoidable.

In the scope of this paper, we assume a system with a multi-GPU array, in which each GPU can be assigned to either simulation process or rendering process. As such, the load balancing can be achieved by scheduling the number of GPUs assigned to each process. For example, when workload of the simulation process is larger, the data buffer will be almost empty, and more GPUs will be assigned to the simulation process to avoid the idle time for the rendering GPUs. When the workload of the rendering process is larger, the data buffer will be filled quickly, and more GPUs will be assigned to the rendering process. And this assignment procedure should adapt the workload difference between the simulation and rendering time frames. The entire adaptive load balancing process is illustrated in Figure 3.

Let us have a formal definition of the execution time associated with this process so that we can formulate the problem we address in this paper. T_i^{vis} and T_i^{sim} are the execution time of the rendering stage and simulation stage for the time frame i respectively. There are a set of parameters, $p_1, p_2, \ldots$, including hardware settings and algorithm settings, that affect the execution time, such that

$$T_i^{task} = \psi_i^{task}(n, p_1, p_2, \ldots), \tag{1}$$

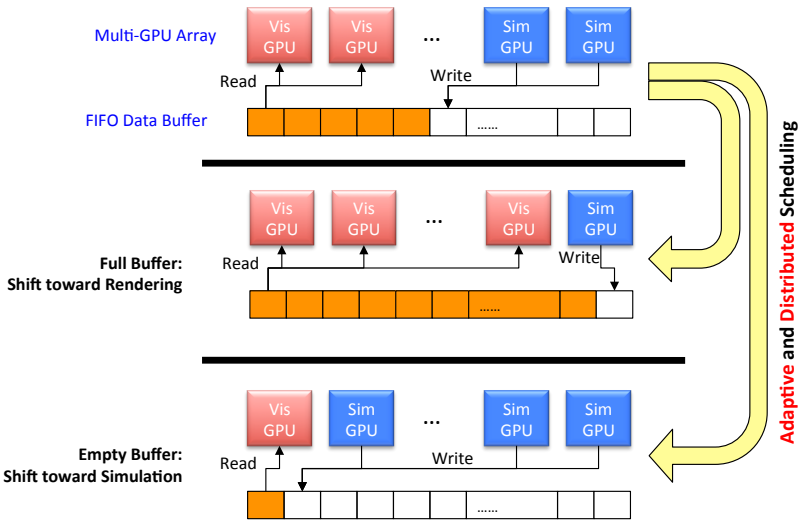


Fig. 3 The multi-GPU configuration and the problem definition of our load balancing approaches.

where n is the number of GPUs assigned to the *task* (vis or sim); the function ψ_i^{task} maps the performance parameters of the *task* to an execution time at frame i, which is monotonically decreasing for when the number of GPUs assigned to the task, n, increases.

The load balancing problem in a multi-GPU environment of N available GPUs can be defined as the following optimization process,

$$M_i = \underset{M_i \in \{1,\dots,N-1\}}{argmin} (\|\psi_i^{sim}(M_i, p_1, p_2, \dots) - \psi_i^{vis}(N - M_i, p_1', p_2', \dots)\|), \quad (2)$$

where M_i is the final number of GPUs assigned to the simulation process ($N - M_i$ GPUs are assigned to the rendering process), so that execution time difference between T_i^{vis} and T_i^{sim} is minimized. We need to note that M_i needs to be determined at every time frame i.

4 Data-driven Load Balancing

As we described in the previous section, our load balancing approaches are rooted on the solution of the minimization problem defined in Equation 2, which is to find the optimal number $M_i \in \{1, \dots, N-1\}$ for each frame i that balanced execution time for the simulation and rendering. In this section, we assume that the hardware settings and software settings do NOT change between time frames. Therefore, all the M_i values should be the same across all the frames.

The key to this minimization process is the understanding of the mapping function ψ, which determines the execution time of a task based on the

hardware and algorithmic parameters. Due to the complexity of the hardware architecture and algorithm configuration, it is difficult to provide an analytical description of ψ. The common practice is either rigorously analyze the performance through a multi-dimensional characterization approach [**?**], or model an approximated performance model by significantly simplifying the GPU architecture, e.g., only focusing on the memory transactions [**?**]. Therefore, we introduce a data-driven approach to estimate the function ψ by sampling the parameter space of the function during a performance pre-evaluation phase. In this section, we provide a brief illustration of this data-driven load balancing approach. For the detail about the approach, please refer to our previous work [8].

4.1 Task Partition and Scheduling Algorithm

Before we introduce our data-driven approach, we like to lay some ground work for the task scheduling algorithm.

Load balancing for our case-study application requires partitioning the work load of a task in order to distribute to different GPUs. Two types of work partitioning are possible for the application: *intra-frame* and *inter-frame*, as shown in Figure 4. Intra-frame partition splits the work load of the task in a time frame into n portions for n GPUs, and each GPU is assigned one portion to finish. Inter-frame partition collects n frames of work load of a certain task for n GPUs, and each GPU is assigned to finish one frame of work load.

The N-body simulation utilizes intra-frame partitioning by splitting up a simulated particles into several portions. Since each frame of simulation requires the data from previous frame, the simulation for consecutive frames cannot be computed concurrently. Thus, inter-frame partition is not possible for the simulation task. While this requires additional data communication between different simulation GPUs to combine the simulation results for each frame, the computation can be distributed among multiple GPUs.

Ray tracing (rendering) in this implementation uses inter-frame partitioning, so that each GPU will render an individual frame of data in order to avoid the communication for combining results. Since rendering each frame only depends on the required simulation data, each frame can be rendered independently using the inter-frame partition approach.

Note that the intra-frame partition needs to combine the data from different GPUs into a complete frame of simulation data. Therefore, all the simulation GPUs have to be synchronized before moved to the simulation of the next frame. For the rendering GPUs, since their tasks are independent with each other, no synchronization is necessary.

Based on this task partition strategy, the distributed scheduling algorithm for each GPU can be illustrated as Figure 5. As shown, all GPUs begin with initialization to allocate host and GPU memory. Afterwards, each GPU continually carries out the execution loop, which involves reading, execution, and writing of results. A barrier is used for simulation for every frame to ensure

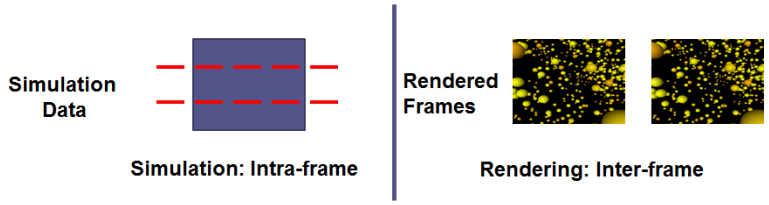


Fig. 4 Intra-frame and inter-frame task partitioning schemes for simulation and rendering, respectively.

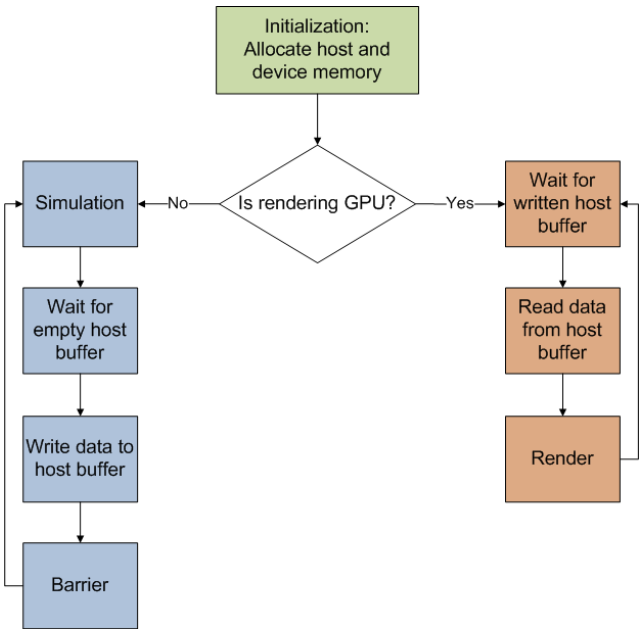


Fig. 5 The distributed scheduling algorithm for each GPU in our data-driven load balancing approach.

results are written together, while each rendering GPU can execute separately and does not require a barrier. The core of this algorithm is the determination of the task (simulation or rendering) for each GPU, which we will describe in the next section.

4.2 Execution Time Evaluation and Estimation

In order to solve the minimization problem defined in Equation 2, we reformulate the optimization as a general function,

$$M = f(i, s, p), \tag{3}$$

where i, s, p are the three most important performance factors that influence the optimal load balancing value M, which is the optimal number of GPUs

allocated for rendering task versus simulation task. i is the number of iterations for simulation for a single simulation time frame, s is the number of samples for supersampling used in the ray tracing algorithm, p is the number of particles simulated in the application.

We then build a data-driven model to evaluate and estimate the function in Equation 3. The model is built by evenly sampling the three-dimensional parameter space while experimentally determining the optimal value M for the best load balancing results for each sample. After the sampling stage, we have a 3-D rigid grid of data samples that covers the entire parameter space. The sampling resolution for each dimension is determined experimentally to avoid dramatical result changes for load balancing.

Based on the previous performance results for this data-driven model, the method computes the optimal workload balance for a new set of input parameters. Given a new set of input parameters, i', s', p', we estimate the new load balancing value M' by performing a trilinear interpolation in the 3D sampling space of previously evaluated results, as illustrated in Equation

$$\begin{aligned} M_{isp} = {} & M_{000}(1-i)(1-s)(1-p) && +M_{100}i(1-s)(1-p) \\ & + M_{010}(1-i)s(1-p) && +M_{001}(1-i)(1-s)p \\ & + M_{101}i(1-s)p && +M_{011}(1-i)sp \\ & + M_{110}is(1-p) && +M_{111}isp \end{aligned}$$

4.3 Data-driven Load Balancing Results

Our experimental results are obtained on a single workstation system with a quad-core Intel CPU with Hyper-Thread and four NVidia GTX 295 graphics card (two GPUs per card). In this section, we first characterize the three performance factors. We then present the load balancing performance of our data-driven approach.

4.3.1 Characteristics for Performance Factors

Supersampling is a commonly used techniques for improving the quality of ray-tracing based rendering. The differences in supersampling can be seen in Figure 6. Aliasing artifacts due to inadequate sampling can be seen in the image on the left with one sample per pixel. Using sixteen samples per pixel in a random fashion, on the other hand, significantly improves the results.

While supersampling improves the quality of the final image, it comes at a performance cost as shown in Table 1. The increase in execution time for a greater number of samples for supersampling is linear.

Table 2 shows the time for simulation when performing multiple iterations for each time frame. The performance of simulation shows a linear increase in time with an increase in number of iterations. While a smaller time-step provides more accurate simulations, it requires more iterations to advance the

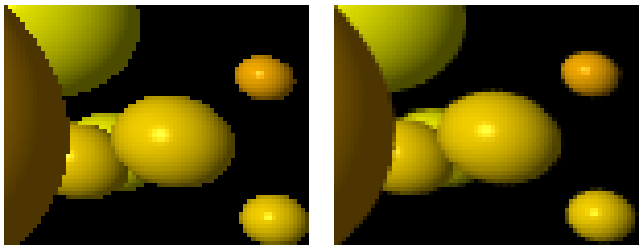


Fig. 6 Comparison of single sample (left) and 16 sample randomized supersampling (right).

1 sample	2 samples	3 samples	4 samples
45.9062 ms	85.5727 ms	124.571 ms	162.728 ms

Table 1 GPU execution time (ms) for ray tracing based on number of samples for supersampling.

20 iterations	40 iterations	60 iterations	80 iterations
31.87 ms	62.56 ms	92.70 ms	122.86 ms

Table 2 GPU execution time for simulation based on number of iterations.

Iterations	2000	4000	6000	8000	10000	12000
Percent	58.07	35.37	26.87	21.81	14.94	0.00

Table 3 Percent difference in positions of simulation from 12000 iteration simulation.

simulation for each frame. Thus, using a smaller time-step but increasing the number of iterations leads to an increase in execution cost.

Table 3 shows the percent difference in positions of simulation from 12000 iteration simulation, which uses the smallest time-step. Each simulation is carried out over the same total time, with a smaller time-step for simulations run for more iterations. With a smaller time-step, the accuracy of the simulation is improved due to the finer granularity used for integration in the N-body simulation.

Figure 7 shows the trends for performance times according to M (number of simulation GPU) with varying number of simulation particles using sixteen samples for ray tracing and eighty iterations for simulation. The minimum along each line segments represents the optimal load balance value M since it has the shortest execution time. The cost of simulation increases at a faster rate as particle number increases due to the nature of the N-body simulation, while ray tracing scales linearly with the particle number. This causes the overall performance to be increasingly limited by simulation time for larger problems.

Figure 8 shows performance for various number of particles with a varying workload distribution for four-sample ray tracing and 80 iterations for simulation. This graph shows that allocating more GPUs for simulation when the number of samples is low can result in performance gain. The difference in the

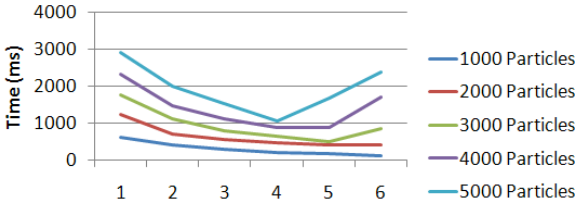


Fig. 7 The total execution time of the application when simulating various number of particles with 16 samples for ray-tracing supersampling and 80 iterations for the N-Body simulation. X axis is the number of rendering GPUs ($N - M$).

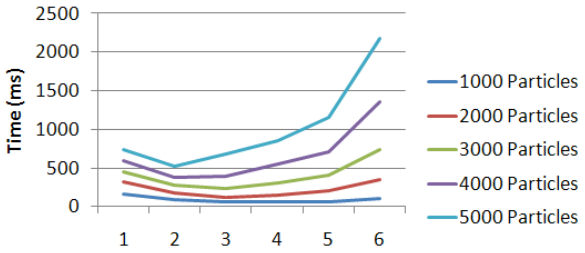


Fig. 8 The total execution time of the application for 4 samples for ray-tracing supersampling and 80 iterations for the N-Body simulation.

trend from Figure 7 also demonstrates that different algorithmic parameters can lead to significantly different optimal workload distributions.

Figure 9 shows the trend for varying simulation iterations with a constant number of supersampling samples (4) and particle number(1,000). This diagram shows that for a small particle number, the execution time is mostly limited by rendering. However, with the largest number of rendering GPUs we can see that introducing more iterations increases the workload for simulation tasks and shows that execution time is then limited by simulation.

In comparison to the previous graph, Figure 10 shows the trend for varying the number of samples for supersampling with a constant number of simulation iterations and input size. Due to the low number of simulation iterations, it is always necessary to allocate many rendering GPUs for the optimal workload balance. However, as the number of samples increases, it becomes increasingly more costly to have fewer GPUs allocated for rendering.

4.3.2 Load Balancing

The optimal load balancing demonstrates some trends in the data that must be accounted for by our data-driven load balancing. Figure 11 (a) shows a trend of optimal load balance based on the number of iterations for simulation. As shown, increasing the number of iterations requires a greater number of GPUs dedicated to simulation to achieve optimal load balance. With the fewest iterations for simulation, the majority of GPUs should be allocated for ray tracing due to the greater cost of ray tracing.

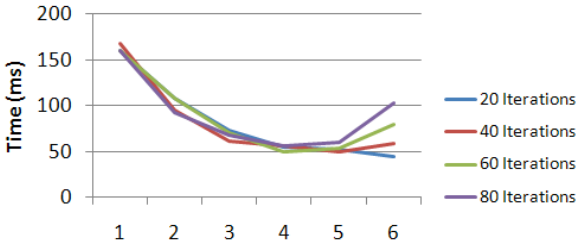


Fig. 9 The total execution time of the application for 4 samples for ray-tracing supersampling and 1000 simulated particles.

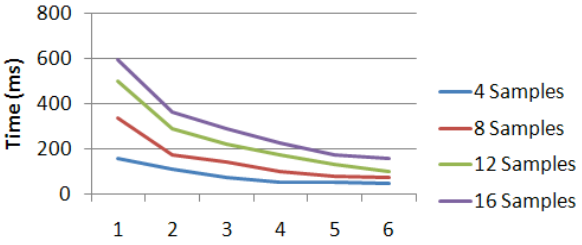
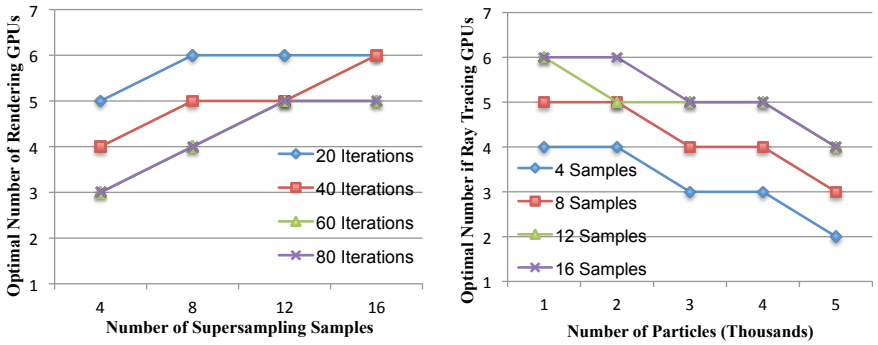


Fig. 10 The total execution time of the application for 20 simulation iterations and 1000 simulated particles.



(a) (b)

Fig. 11 The optimal load balance, number of rendering GPUs, affected by (a) number of supersampling samples and number of simulation iterations, with 3,000 simulated particles; (b) number of simulated particles and number of supersampling samples, with 60 simulation iterations.

Increasing the number of samples for supersampling increases the cost of ray tracing and also impacts the load balancing scheme. Figure 11 (b) shows that increasing the number of samples for supersampling results in need of additional ray tracing GPUs to improve workload balance. A larger number of particles requires fewer GPUs for ray tracing due to the smaller increase in cost of ray tracing with larger datasets.

Figure 12 shows the average performance speedup of our data-driven load balancing compared with non-balanced execution. In the Figure, we show the speedup percentage when we perform the interpolation in the data-driven model using one, two, and there parameters. These results show that a significant speedup can consistently be achieved with different parameter configurations.

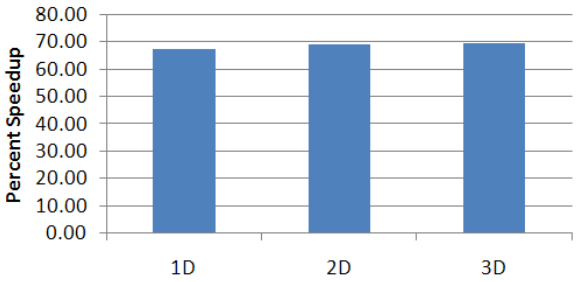


Fig. 12 Average data-driven load balancing speedup when predicting the optimal configuration using one, two, and three parameters.

5 Dynamic Load Balancing

Although data-driven load balancing is suitable for the applications with predefined hardware and software settings, it has several limitations. If the hardware or the algorithm changes, the performance data has to be re-collected and the data-driven model has to be re-built. In addition, if the algorithm allows some performance factors, such as the number of simulation iterations and supersampling samples, to change at run-time, the data-driven approach will become useless if the pre-collected performance data does not cover the changes.

To some extent, the data-driven model is not a general solution, since it tries to probe the optimization process, defined in Equation 2, as a black-box by tediously sampling the input-output relation space of the performance function of Equation 1. Instead, the best approach is to analytically solve the optimization problem at run-time.

We introduce a dynamic load balancing approach with a method to *approximate* the function $\|\psi_i^{sim}(M_i, p_1, p_2, \dots) - \psi_i^{vis}(N - M_i, p'_1, p'_2, \dots)\|$ used in Equation 2, which tells the difference in workload between the simulation and the rendering. The main idea behind it is to estimate the workload difference, which is influenced by a complicated set of hardware and algorithmic parameters $p_1, p_2, \dots$, with a simplified metric, as follows,

$$\psi_i^{sim}(M_i, p_1, p_2, \dots) - \psi_i^{vis}(N - M_i, p'_1, p'_2, \dots) \approx \phi(p), \tag{4}$$

where p is a performance indicator that is not directly related to hardware and algorithmic parameters $p_1, p_2, \ldots, p'_1, p'_2, \ldots$; and function ϕ should be computationally cheap to be evaluated. Such approximation will make the optimization process much simpler since it only depends on one performance factor instead of many algorithmic parameters.

The key to the success of the approximation defined in Equation 4 is how to choose the performance factor p and the mapping function ϕ. An obvious choice of p is the actual recorded execution time difference from the previous time frame, therefore,

$$\phi(p) = T_{i-1}^{sim} - T_{i-1}^{vis}.$$

However, only use the performance data from the previous time frame can not accurately approximate the execution time of the current time frame, since the algorithmic setting between these two frames can be dramatically different.

In this work, we introduce a new performance indicator, the number of the filled buffer frames F, for the approximation. Since the host buffer is used to synchronize between the simulation and the rendering tasks, its fullness can be used to indicate which task has more workload. For example, when the buffer is close to full, it means the simulation is executed faster than the rendering, and more GPUs should be assigned to the rendering task. When the buffer is close to empty, it means the simulation takes more time to finish, and more GPUs should be assigned to the simulation.

The final approximation function is then defined as follows.

$$\psi_i^{sim} - \psi_i^{vis} \approx \phi(p) = \phi(F) = \begin{cases} -1.0 & \text{if } F = 0, \\ 0.0 & \text{if } F \in \{1, \ldots, B-1\}, \\ 1.0 & \text{if } F = B, \end{cases} \tag{5}$$

where B is the maximal number of the buffer frames.

In Equation 5, we define the empty buffer as the simulation workload smaller than the rendering; the full buffer as the simulation workload larger than the rendering; and the other cases as the balanced workload.

5.1 Dynamic Load Balancing Algorithm

Based on this load balancing approximation and strategy, the workload scheduling algorithm will switch a GPU from one task to another when the buffer is full or empty; otherwise, the task assignment for each GPU keeps the same. When the buffer is full, all the simulation GPUs must wait and one simulation GPU will switch its own task to the rendering to avoid idle time. When the buffer is empty, all the rendering GPUs must wait and one rendering GPU will switch its own task to the simulation to avoid idle time. This task switch only allows one GPU switch at a time so that no dramatic workload change can happen and no scheduling/switch oscillation can occur.

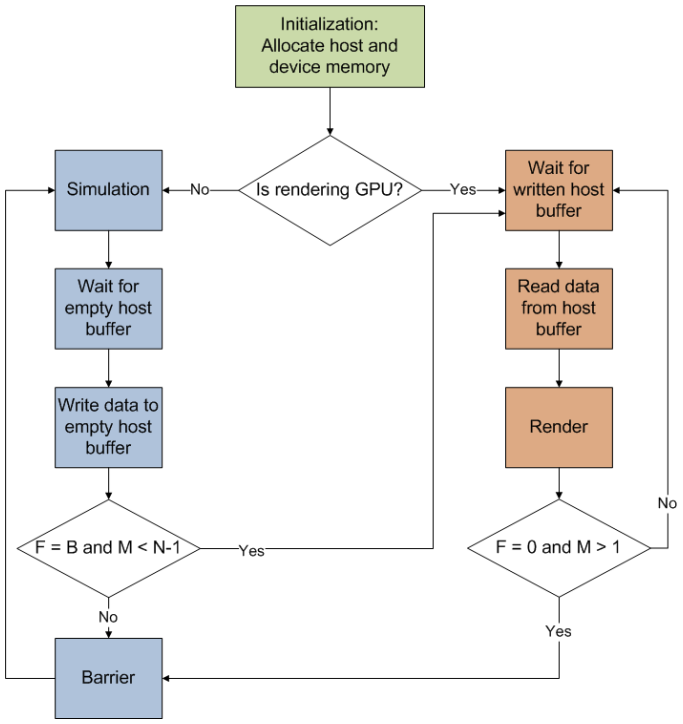


Fig. 13 This flowchart shows the execution and task switching for dynamic load balancing. F is the number of host buffers, B is the maximum number of host buffers, M is the current number of rendering GPUs, and N is the total number of GPUs available for load balancing.

The distributed scheduling algorithm that runs on each GPU can be illustrated as Figure 13. It is similar to the scheduling algorithm for data-driven load balancing, with additional steps for checking to switch tasks.

Before the task execution begins, initialization is necessary. Initialization involves allocating both host and GPU memory required for data. A barrier is used after initialization to ensure initialization has completed for each task before starting. After initialization, the task execution loop begins where each task iteratively executes each stage for all time frames. A task may either be executed by a group of GPUs or a single GPU. A barrier is used to implement synchronization for intra-frame task partition of the group of GPUs for the simulation, This is to ensure the completion of writing a frame of data to the host buffer. This barrier is also used for another rendering GPU that need to join this task group for simulation. However, this group barrier is unnecessary for the rendering GPUs since they process the data individually and do not require data partitioning. For example, if the host buffer is full and a simulation GPU needs to switch to a group of rendering GPUs , then the GPU must wait until the group of simulation GPUs have completed writing the current frame data. If a GPU needs to switch from the simulation to the rendering, then the GPU can switch without waiting for other rendering GPU.

5.2 Dynamic Load Balancing Results

We now present results that show the advantage of our dynamic load balancing method over data-driven (static) load balancing. The experiments were executed on a single workstation with a 2.67 GHz Intel Core i7 processor and four GTX 295 graphics cards (two GPUs per card).

5.2.1 Dynamic Load Balancing Without Runtime Changes

Figure 14 shows the changes in buffer fullness F and number of rendering GPUs over time for the N-body simulation without algorithmic changes at runtime. Here the y-axis represents the number of filled buffers for buffer fullness, and it represents the number of rendering GPUs for the other series. The number of host buffers B is ten, where all buffers begin empty. This graph shows how the buffer begins empty, then fills, and then load balancing adjusts the workload until an equilibrium is reached. The execution begins with the maximum number of allocated simulation tasks, which causes the host memory to immediately fill as buffer fullness F approaches the number of host buffers B. When the buffer becomes full, dynamic load balancing switches simulation GPUs to rendering one at a time. This causes the buffer to eventually become empty again, causing a task switch back to simulation. The buffer then fills again, and after a final task switch the system reaches an equilibrium in fill and empty rate. At the final moment, the number of rendering GPUs M for the optimal load balance is reached. Dynamic load balancing thus allows for changing tasks at runtime to avoid waiting when the buffer becomes full or empty. Many cases of execution oscillate between two configurations due to the limitation of a discrete number of GPUs, but on average over time this results in the optimal load balance.

This advantage of dynamic load balancing of being able to switch tasks at runtime allows for a speedup over static load balancing. Figure 15 shows

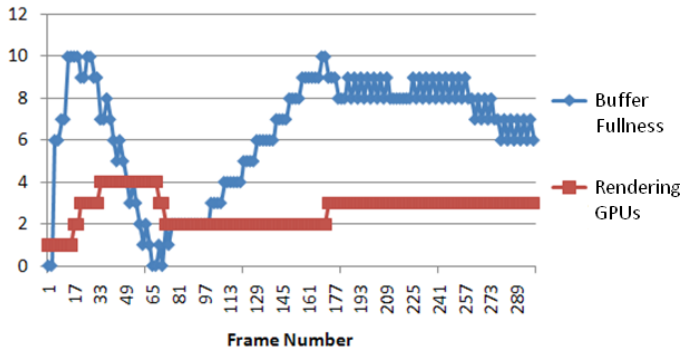


Fig. 14 Change in buffer fullness and the number of rendering GPUs over time for 3000 particles using dynamic load balancing without runtime changes. The maximum frames of host buffer is 10, while the maximum number of rendering GPUs is 6.

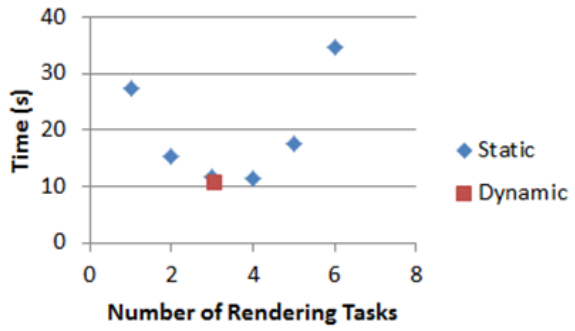


Fig. 15 Comparison of the execution time of the static configurations and dynamic load balancing with 2000 particles without runtime algorithmic changes.

the performance of six static configurations of GPUs compared to dynamic load balancing for 2000 particles. The figure demonstrates how different static configurations have varying performance due to the number of rendering GPUs and the varying amount of processing required for rendering and simulation. It also shows how dynamic load balancing can adjust workloads to the optimal static configuration without needing a performance model or performance data beforehand.

Figure 16 exhibits the ability of dynamic load balancing to find the optimal configuration for 4000 particles as well. More simulated particles require a relatively greater workload since the cost of the N-body simulation increases at a higher rate than ray tracing, which scales linearly with the number of particles. In both cases the dynamic load balancing method can adjust to the optimal workload distribution.

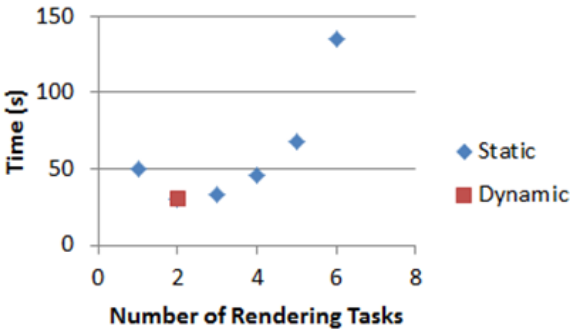


Fig. 16 Comparison of the execution time of the static configurations and dynamic load balancing with 4000 particles without runtime algorithmic changes.

5.3 Workload Changes at Runtime

Dynamic load balancing can adjust to dynamic algorithmic changes in workloads at runtime. For example, a user may want to change the quality of an image by increasing the number of samples for supersampling. Supersampling improves the quality of the rendering, but it will increase the cost of ray tracing as well. Many other dynamic changes such as zooming and evolving simulations can introduce similar changes in workload. These changes will lead to a change in optimal load balance in workload distribution. Static load balancing is not able to optimally handle such unpredicted change.

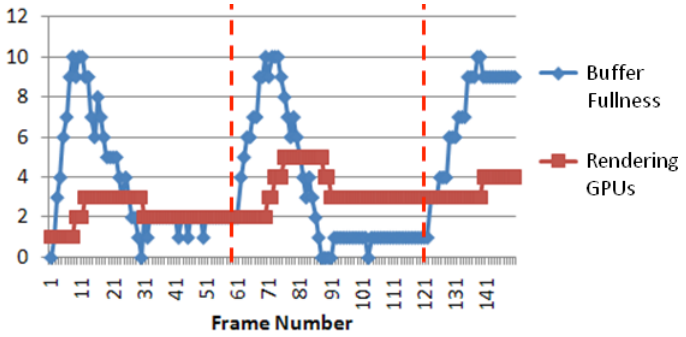


Fig. 17 Change in buffer fullness over time for 3000 particles with increases in supersampling every sixty frames as shown by red dotted lines. Here the maximum number of full host buffers is ten, while the maximum number of rendering GPUs is six.

Figure 17 shows an experiment, in which the supersampling level changes at fixed intervals of 60 frames, with four, eight, and twelve samples. It shows the changing buffer fullness and the rendering GPU number switches for a 3000 particle simulation. It illustrates how dynamic load balancing continually increases the number of rendering GPUs as the sample count increases for supersampling. The maximum number of simulation GPUs is first allocated, causing the buffer to quickly fill as the fullness F reaches the number of host buffers B, and dynamic load balancing increases the number of rendering GPUs. After soon reaching an equilibrium for four samples for supersampling, an increase in supersampling is applied as shown by the dotted line. This causes the buffer fullness to quickly increase. Dynamic load balancing then switches more tasks to rendering, and an equilibrium is again reached. A second increase in supersampling then causes the buffer fullness to sharply increase again, followed by another task switch from dynamic load balancing.

Figure 18 (a) shows the performance of static configurations versus dynamic load balancing for trials that involves a change in the number of samples and simulation iterations from 1 to 16 and 320 to 20, respectively, for 2000 particles. This chart shows a performance improvement when using dynamic load balancing over all static configurations. The average number of rendering

GPUs is about three since the single change in supersampling and simulation iterations causes a switch from the least to the greatest number of rendering GPUs allocated. Figure 18 (b) shows the comparison with 4000 particles and also shows a significant performance improvement of dynamic load balancing over static configurations. Thus, these results show that dynamic load balancing can consistently reach the optimal load balance for different dataset sizes and can offer a performance improvement over static configurations.

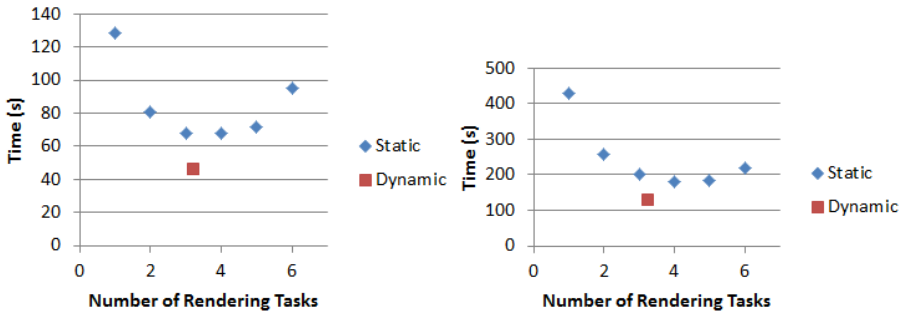


Fig. 18 Comparison of the execution time with changes in supersampling and simulation iterations for static configurations and dynamic load balancing. (a) The simulation with 2000 particles and (b) 4000 particles.

Dynamic load balancing leads to consistent speedup over both average and optimal static workloads balancing as shown in Figure 19. It shows a constant speedup of dynamic load balancing over the optimal static load balancing configuration. An much greater speedup can be seen for dynamic load balancing over worst case static configurations. This worst case speedup increases due to the greater difference in workloads as the dataset size increases due to the faster increase in cost of simulation with dataset size.

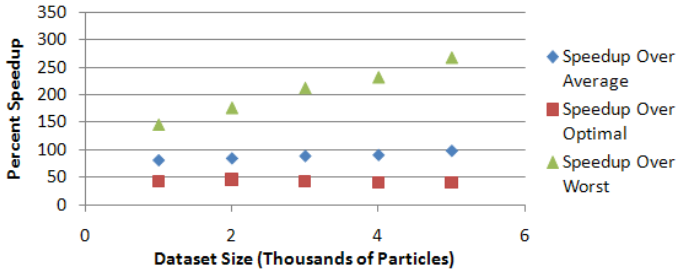


Fig. 19 Speedup of dynamic load balancing over optimal, average, and worst case for different dataset sizes (the number of simulated particles) with changes at runtime.

6 Conclusions and Future Work

We propose a data-driven and a dynamic multi-GPU load balancing approaches for the pipeline execution, and have shown the viability of the load balancing methods by applying them to an in-situ visualization application. The data-driven load balancing predicts the optimal scheduling configurations through an interpolation model built from the previously recorded performance data of an application. As an extension of the data driven model, in order to adapt the run-time changes of the algorithmic parameters, we introduce the dynamic load balancing approach. The approach dynamically schedules the GPUs between simulation and rendering tasks by approximating the workload difference with a performance metric, fullness of the data buffer. Dynamic load balancing can achieve equivalent performance to data-driven load balancing, and it can also gain additional performance improvement when an application has runtime changes in workloads.

Several areas of future work could be explored for this project. A possible additional criteria for load balancing is a performance model, which may more quickly find an optimal load balance and reduce oscillations in load balancing. A performance model could be used to calculate an approximation to the optimal load balancing scheme based on a small sample of data.

Another alternative for criteria is rate of change in buffer fullness. This could also be used in combination with buffer fullness as a criteria for switching, where a high enough fill or empty rate would trigger a switch in addition to an empty or full buffer. This would allow a switch to be triggered more quickly than buffer fullness alone after an immediate runtime change in workloads, since it could avoid waiting until the buffer is completely empty or full.

References

1. Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM.
2. Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive work stealing with parallelism feedback. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 112–120, New York, NY, USA, 2007. ACM.
3. Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, New York, NY, USA, 1998. ACM.
4. A.P.D. Binotto, C.E. Pereira, and D.W. Fellner. Towards dynamic reconfigurable load-balancing for hybrid desktop platforms. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –4, 04 2010.
5. Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

6. Zhibin Fang, Xian-He Sun, Yong Chen, and Surendra Byna. Core-aware memory access scheduling schemes. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.
7. Thomas Fogal, Hank Childs, Siddharth Shankar, Jens Krüger, R. Daniel Bergeron, and Philip Hatcher. Large data visualization on distributed memory multi-gpu clusters. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 57–66, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.
8. Robert Hagan and Yong Cao. Multi-gpu load balancing for in-situ visualization. In *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011.
9. Alan Heirich and James Arvo. A competitive analysis of load balancing strategies for parallel ray tracing. *J. Supercomput.*, 12(1-2):57–68, 1998.
10. Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.
11. Mark Joselli, Esteban Clua, Anselmo Montenegro, Aura Conci, and Paulo Pagliosa. A new physics engine with automatic process distribution between cpu-gpu. In *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 149–156, New York, NY, USA, 2008. ACM.
12. J. Prin L. Nyland, M. Harris. Fast n-body simulation with cuda. *GPU Gems 3*, pages 677–695, 2007.
13. Jordi Roca Monfort and Mark Grossman. Scaling of 3d game engine workloads on modern multi-gpu systems. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 37–46, New York, NY, USA, 2009. ACM.