Multi-GPU Load Balancing for Simulation and Rendering

Robert D. Hagan

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Science

Yong Cao, Chair
Chris North
Eli Tilevich

June 24, 2011
Blacksburg, Virginia

Keywords: Multi-GPU Computing, Load Balancing, Simulation, Rendering

Multi-GPU Load Balancing for Simulation and Rendering

Robert D. Hagan

(ABSTRACT)

GPU computing can significantly improve performance by taking advantage of massive parallelism of GPUs for data parallel applications. Computation in visualization applications is suitable for parallelization on the GPU, which can improve performance and interactivity in these applications. If used effectively, multiple GPUs can lead to a significant speedup over a single GPU. However, the use of multiple GPUs requires memory management, scheduling, and load balancing to ensure that a program takes full advantage of available processors. This work presents methods for data-driven and dynamic multi-GPU load balancing using a pipelined approach and a framework for use with different applications. Data-driven load balancing can improve utilization for applications by taking into account past performance for different combinations of input parameters. The dynamic load balancing method based on buffer fullness can adjust to workload changes at runtime to gain an additional performance improvement. This work provides a framework for load balancing to account for differing characteristics of applications. Implementation of a multi-GPU data structure allows for use of these load balancing methods in the framework. The effectiveness of the framework is demonstrated with performance results from interactive visualization that shows a significant speedup due to load balancing.

Acknowledgments

I would like to thank my advisor Dr. Yong Cao for his guidance throughout the project. I would also like to thank Dr. Eli Tilevich for his help on software engineering aspects, and I would like to thank Dr. Chris North for his feedback on the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

GPU computing offers massively parallel processing that can greatly accelerate a variety of data parallel applications. Interactive visualization is one application that is often highly suitable for computation on the GPU. Use of multiple GPUs can lead to even greater performance gains in these applications by executing multiple tasks on different GPUs and overlapping computations. This provides an opportunity for handling larger scale problems that a single GPU cannot process in real-time. The resulting increase in runtime speeds can allow for real-time navigation and interaction in visualization, which can significantly improve the visualization experience. However, multi-GPU computing presents several challenges that must be addressed by developers. Scheduling of workloads and memory management create additional developer requirements. Workloads with multiple processors can become unevenly distributed and can lead to significant underutilization. This problem can be addressed by providing load balancing to obtain equal workloads for each processor. Effective load balancing can greatly increase utilization and performance in a multi-GPU environment by distributing workloads equally, which is the main focus of this work.

There are two methods for multi-GPU parallelization: a *data parallel* method and a *pipelined* method. The data parallel approach distributes data among available processing units. The pipelined approach splits processing of data into stages for different processors to handle. For

example, an in-situ visualization application involves concurrent simulation and rendering, where a simulation stage writes data and a rendering stage reads data. Multiple buffers of frames can be used to pipeline the data. Our work provides both data parallel and pipelined load balancing, where data parallel distribution can be used within a stage of a pipelined approach. This approach allows for improved flexibility that can help to improve utilization in a multi-GPU processing environment. However, our load balancing focuses on a pipelined approach since data parallel approaches that equally distribute data require little effort in load balancing. Applications that use multiple GPUs can improve performance by using this load balancing to equally distribute workloads and avoid idle time for individual GPUs. By using a pipelined approach, each GPU can be kept busy by ensuring that each GPU can process work and then pass the data to another GPU to handle the next stage of the computation.

Providing load balancing in applications introduces several issues. For example, memory management and synchronization in a multi-GPU environment must also be handled. Data for each frame in visualization must be transferred through host memory to another GPU in order to facilitate the pipelining process. Scheduling needs to account for varying performance and workloads of different applications. Tasks may also need to be partitioned in order to properly distribute workloads.

Due to the advantages of multi-GPU computing, we propose a framework for multi-GPU data-driven and dynamic load balancing. This framework handles load balancing of *data producer* and *data consumer* stages. A producer stage writes data to host memory, while a consumer stage reads data from host memory. In interactive visualization applications, the producer stage is simulation, while the consumer stage is rendering. Figure 1.1 shows a multi-GPU configuration that can be used for the proposed load balancing method. It identifies how the method allows for multiple GPUs to be used for both simulation and rendering tasks simultaneously, while host memory is used to transfer results among GPUs.

To test our framework, we have applied it to an in-situ visualization application. This

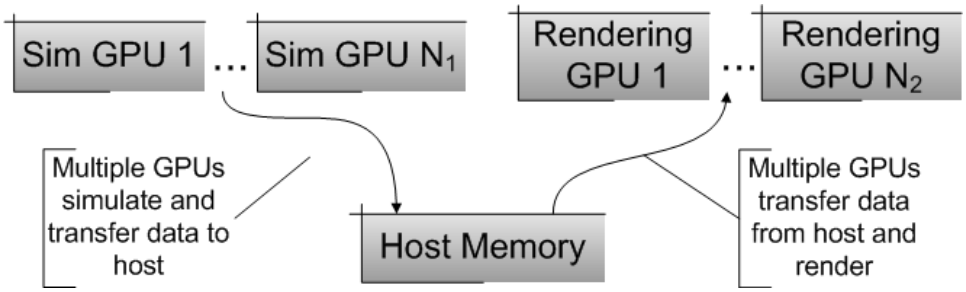

Figure 1.1: Multi-GPU configuration used in the framework. "Sim" refers to a "Simulation" task run on GPUs. Here N_1 is the number of simulation GPUs and N_2 is the number of rendering GPUs.

application involves continuous parallel simulation and rendering that allows for interactive visualizations. The N-body simulation involves computation of the interaction among a group of bodies, and it can be solved by computing the force of all bodies on each other. This problem is used in many domains, including biomolecular and physics applications. As in the work of [38], the gravitational N-body problem can be expressed as

$$F_i = Gm_i \sum_{1 \leq j \leq N, j \neq i} \frac{m_j r_{ij}}{||r_{ij}||^3} \tag{1.1}$$

where F_i is the computed force for body i, m_i is the mass of body i, r_{ij} is the vector from body i to j, and G is the gravitational constant.

In a molecular simulation, an N-body simulation algorithm can be used to compute the interaction of each atom in the molecule. This application can benefit from use of multiple GPUs to both compute new frames of simulation and render these new frames in parallel. Computation of simulation together with rendering in real-time allows for interactive update in visualization applications. This can result in a smooth interaction experience with use of increased processing power to accelerate computing.

While multiple GPUs can offer a large performance gain in visualization applications, many challenges exist in scheduling multiple tasks. Load imbalance can lead to underutilization of

available resources and reduced performance in the visualization. Multi-GPU computing can therefore benefit from a method for load balancing to improve workload distribution. Load balancing needs to account for performance in order to maximize use of available resources. Use of the load balancing method in a visualization accounts for workload differences between simulation and rendering to maintain more equal workload distribution. Visualization algorithms such as ray tracing can be computationally expensive, so specific techniques to distribute and load balance rendering workloads can offer considerable performance gains for visualization applications. Taking advantage of concurrent computations of visualizations with simulation can lead to a significant performance improvement to maintain interactivity in these applications.

Many different factors can determine the workloads in the visualization that affect the optimal load balance. The cost of simulation and rendering can vary depending on the chosen methods for each. The input data size may differ for applications, which can have a varying effect on simulation and rendering time. Accuracy of simulation can be improved by using more accurate techniques or by decreasing the timestep in simulation. In visualization, image quality is important to produce a better result for interactive viewing. Ray tracing is a rendering method that can produce realistic results on the GPU for visualization. Supersampling can further improve image quality by using multiple samples per pixel that can decrease aliasing. Use of ray tracing with supersampling can greatly improve the results in interactive visualization but significantly increases computations that can be accelerated through use of multiple GPUs. On the other hand, the timestep of simulation affects the accuracy and a large timestep can lead to an inaccurate simulation. A smaller timestep, however, requires more simulation iterations to achieve the same speed of simulation as before. Therefore, while a smaller timestep improves the accuracy of simulation, it increases the cost of the simulation workload. Supersampling and timestep can vary the cost of computations, which will require adjusting load balancing for multi-GPU processing.

To address these issues, we propose two methods of load balancing suitable for different applications. Load balancing based on past performance, or *data-driven load balancing*, can

improve utilization by determining the best configuration for a given application beforehand. This must take into account the workloads in an application as well as the available hardware. However, this method can require collecting various performance data beforehand that can be used to determine the best workload distribution for a given application. Furthermore, this method does not adapt to unpredicted changes at runtime for an application. *Dynamic load balancing*, on the other hand, can be used to adapt workload distribution to take into account changes at runtime. This can involve changing the stage of the pipeline assigned to a given GPU in order to have more effective load balancing.

Our work provides the following contributions:

- Data-driven and dynamic multi-GPU load balancing methods based on a pipelined approach to improve utilization and performance in visualization applications
- A multi-GPU framework that handles load balancing on multiple GPUs for different applications
- Demonstration of results with in-situ visualization

Chapter 2

Related Work

Several areas of past research have addressed issues related to this work, including multi-GPU load balancing, visualization, and frameworks.

2.1 Load Balancing

Past works have explored load balancing to improve utilization in GPU computing. Several of these works have addressed load balancing in multi-GPU computing, but most have addressed top-down partitioning for load balancing and have not addressed using a pipelined approach with multiple stages of computation. Fogal et al. implement GPU cluster volume rendering that uses load balancing for rendering massive datasets [22]. 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. They do not provide a general framework for load balancing for various types of applications. Monfort et al. use load balancing methods in a game engine for rendering that use split screen and alternate screen partitioning [43]. 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. As in our work, they deal with rendering in an application, but they do not account for other types of tasks on the GPU, such as simulation, that can introduce muliple stages in a pipelined approach. It is also limited to a specific rendering application, while our work provides a more general framework suitable for various applications. Binotto et al. present work in load balancing in a CFD simulation application [9]. They use both an initial static analysis followed by adaptive load balancing based on various factors including performance results. While this previous work presents scheduling techniques, they do not focus on load balancing for in-situ visualization using a pipelined approach based on rendering and simulation tasks. Hagan et al. present a data-driven method for load balancing based on input parameters for in-situ visualization applications [28]. 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 work and is extended with dynamic load balancing in our work, which also takes into account unpredicted, dynamic changes at runtime.

Other works in load balancing have included Becker et al., who describes a system that handles finite element simulations on clusters that effectively utilizes both CPU and GPU processing [8]. The system uses the ParFUM framework, a library in the parallel runtime framework Charm++, and it presents an API and software layer for the finite element method. By allowing similar code for GPU and CPU functions, it allows users to have programs that load balance computations between the CPU and GPU. They found that the optimal setup distributed two mesh partitions per CPU and seventeen partitions per GPU. This work applies to our project because it employs a heterogeneous environment with both CPUs and GPUs and adapts its algorithm for this case. In future work their method could be generalized to work with other applications that must effectively utilize multiple GPUs.

The authors also state that they hand-tuned the performance, so this hand-tuning could be adapted to automatic tuning of the application as in our work. Moreover, generalizing this application to other types of scientific computation would be helpful in the reuse of this work for applications that use GPU computations. While this work handles load balancing between CPU and GPU, our work focuses on load balancing only among GPUs. In addition, this work focuses on a top-down approach instead of a pipelining approach as in our work.

Joselli et al. present a novel physics engine that uses automatic distribution of processes between the CPU and GPU to improve performance [35]. It proposes a dynamic scheduling scheme that approximates the load on the CPU and GPU, and uses this to distribute physics computations. Their method uses a virtual base class called Distributor that allows for customized heuristics to be derived. Moreover, the framework is extensible in that it allows a C function or script to be called to determine scheduling adapted to a user's program. For future work, more advanced load balancing and transfers could improve the performance for multiple GPU applications. This work handles load balancing between GPU and CPU in contrast to our work that focuses on load balancing only among multiple GPUs. We also present a framework to handle scheduling, but our work can be extended to various applications rather than having a focus on physics computations.

Kim et al. presents a new collision detection method that uses both CPU and GPU processing [37]. It performs lazy reconstructions of a bounding volume hierarchy data structure on-the-fly in order to achieve better performance. The CPU handles traversal that requires more branching, while the GPU performs collision tests by solving cubic equations. Furthermore, they propose a task decomposition that allows the algorithm to avoid use of locks to greatly improve performance. It utilizes a scheduling queue for dynamic distribution of work at runtime for load-balancing. Its analysis of synchronization would be helpful in determining useful abstractions for computations on multiple GPUs. For example, they analyze use of locks, atomic operations, asynchronous transfer from CPU to GPU, a task queue, and dynamic task reassignment. The dynamic task reassignment works by keeping a queue of threads that have finished work, and they accept work from the threads that currently have

the greatest amount of work while taking into account data dependencies. They also use a dynamic load balancing scheme that adapts usage of CPU versus GPU utilization by checking work queue sizes and distributing work accordingly. In future work many of the methods they use for parallelization could be wrapped in abstractions to make them easier to use, including dynamic task reassignment, data dependencies, and communication. This would create a more reusable framework as presented in our work. Furthermore, they load balance by offloading work from the GPU to the CPU but do not offload from the CPU to the GPU. By exploring possibilities to offload work in a more automated fashion without having to rewrite code, existing algorithms could become more efficient by using load-balancing. While this work handles load balancing between the CPU and GPU, it does not offer a general framework for a variety of applications as our work does. It uses task queues that load balancing can be applied to, but it does not focus on a pipelined approach for load balancing as in our work.

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 [2]. 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 [3]. 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 [5]. 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.

Several works have explored additional techniques for load balancing. Cederman et al. show results comparing GPU load balancing methods applied to octree construction [11]. 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. Our work is similar in that it uses task switching to provide additional processing for another task, but it focuses more on a pipelined approach for multi-GPU computing. Heirich et al. compare load balancing techniques for ray tracing, and they find that static load balancing results in poor performance [30]. 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. However, they do not focus on a general load balancing framework for pipelined multi-GPU computing. Fang et al. propose a method for scheduling on multicore systems to account for data locality [20]. 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. Jimenez et al. present a scheduler for GPU-CPU systems that uses past performance in order to schedule based on past performance [34]. 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. However, they do not focus on pipelined method of scheduling that our work focuses on.

While these works provide methods for load balancing, they do not focus on multi-GPU load balancing using a pipelined approach as our method does. In addition, many focus on specific cases of use and do not provide a general framework for different applications.

2.2 GPU Visualization

Previous research in multi-GPU visualization has included several works that use multiple GPUs for rendering. Certain works have focused on streaming for out-of-core rendering that manages memory transfers to and from the GPU. Gobbetti et al. present Far Voxels, a visualization framework for out-of-core rendering of large datasets using level-of-detail and visibility culling [25]. Crassin et al. present GigaVoxels, an out-of-core rendering framework for volume rendering of massive datasets using a view-dependent data representation [14]. These two works provide rendering methods for out-of-core datasets that manage data transfers to and from host memory. While our load balancing method uses GPUs and similar streaming of data for visualization, we focus specifically on pipelined load balancing using a general framework.

Correa et al. proposes SPVN, a framework for interactive visualization of large datasets on clusters [13]. SPVN offers traditional pattern-oriented abstractions in their interface, including a proxy for reference counting, a strategy for renderers, a visitor pattern for scene graph traversal, a decorator for input and output streams, and a factor for readers and writers for data formats. Exposing these features to the user makes them much more intuitive to use, and they also improve the flexibility and extensibility of the system. Furthermore, they provide a data management layer that offers effective caching that keeps the data accessed most often in the cache. Future work will involve adding more spatial data structures such as a BSP, better LOD algorithms, support for lossy image compression to improve speed, and an optimized ray tracing back end. Furthermore, by optionally exposing additional caching and load balancing mechanisms, the framework may allow users to tune the framework for their own applications. This work, including the streaming methods and load balancing, could be generalized to other applications in heterogeneous clusters. Compared to our work, they provide abstractions for visualization, but they do not focus on load balancing using a pipelined approach as ours does.

Fan et al. discusses the implementation of the lattice Boltzmann method for fluid simulation

on a GPU cluster [18]. This CFD method is much more parallelizable than other methods since each cell only requires data from each immediate neighbor's cell. The method achieves efficient performance by overlapping network communication with computations and limiting interruptions that can be costly. They also note that they used a simplified communication pattern because they noted that this resulted in the lowest overhead. They adapt their algorithm to take best advantage of the CPU and GPU, which cannot handle complex data structures and control flow. Future work can involve incorporating lossless compression to increase network performance. The system could also be extended to handle other types of fluid simulation, such as Eulerian or Lagrangian methods, but these would require modifications to the communication scheme. This method could be adapted to more efficiently account for multi-GPU on one machine rather than a cluster since this requires accounting for different communication schemes. The work could also account for simultaneous simulation and rendering that could better overlap computation and communication. Unlike our work, they do not focus on providing a general framework using stages of computation.

Frank et al. presents a system for path tracing for rendering scenes using out-of-core and scheduling techniques in a GPU and CPU environment [23]. The system involves an application layer that handles rendering and a scheduling, and a resource management layer that handles out-of-core streaming. Furthermore, it takes advantage of data locality and coherence in rendering which is an important aspect in CPU-GPU computing environments. The scheduler maintains a queue of tasks and manages distribution of the work. Use of their scheduling and load balancing methods could be helpful for developing abstractions for any application using multiple GPUs. Also, the authors state that the system should in the future use out-of-core queues, which would allow for running more jobs in parallel since the queues are currently kept in-core. Currently, they also keep all data in the local disk of each node in the cluster, but sharing caches between nodes may be faster than reading from disk. Another large area for improvement is scheduling, including pre-fetching and a better cache eviction policy. Our work also focuses on visualization and scheduling, but we focus more on a pipelined approach using a general framework.

Other previous work has focused on GPU computing in molecular dynamics applications, relating to the N-body simulation and visualization used in our work. Past work has included Amber, a molecular dynamics software package that offers tools for molecular simulation [44]. This simulation can be used to compute the change in atoms over time due to an N-body simulation as used in our work for visualization. Other research in molecular dynamics has included work by Anandakrishnan et al. to use an N-body simulation to compute the interaction of atoms in a molecule [4]. This simulation also uses an N-body computation as in our work, but does not provide a visualization that we provide. Humphrey et al. present Visual Molecular Dynamics (VMD), a software package for visualization of molecular datasets [33]. However, VMD provides primarily off-line rendering that does not simulate and render each frame interactively to allow for real-time user interaction. Stone et al. present work that computes molecular dynamics simulations on multi-core CPUs and GPUs [48]. Furthermore, they visualize molecular orbitals in an interactive rendering in VMD. We also apply our work to an N-body simulation, but we focus on load balancing using multiple GPUs for both simulation and rendering while their work focuses on data parallel algorithms on single GPUs. Chen et al. present work in multi-GPU load balancing applied to molecular dynamics that uses dynamic load balancing with a task queue [12]. Their framework focuses on fine-grained load balancing usable within a single GPU, while our work focuses on coarse-grained task scheduling among GPUs for both simulation and visualization. While there has been considerable work in visualization and multi-GPU computing, we provide a load balancing framework for simulation and rendering to improve utilization in the pipelined approach.

2.3 Multi-GPU Frameworks

Several other frameworks have been proposed that use GPUs for general-purpose computations in order to provide a reusable system for different applications. We next present a summary of some of the major work in multi-GPU frameworks, including GPU cluster

scheduling frameworks, performance model frameworks, and higher level language frameworks.

2.3.1 GPU Scheduling Frameworks

The first area of GPU frameworks includes frameworks that facilitate scheduling of GPU workloads. Many of these frameworks handle data transfers and distribution of workloads on a GPU cluster. Two works have provided frameworks to facilitate GPU programming that use static communication. Zippy provides a framework for GPU clusters that accounts for both message passing and a shared memory model [19]. CUDASA presents a framework for GPU clusters and multi-GPU systems that easily works with existing single GPU programs [51]. These two frameworks presented later in more detail are limited to static communication and do not handle dynamic scheduling. By using arrays of data that are treated contiguously across devices, they ease developer cost, but the frameworks can also introduce additional overhead due to the automated GPU processing. Unlike these works, our framework provides a dynamic load balancing method that can modify the distributions of workloads at runtime for dynamic applications. DCGN is another framework that allows for dynamic communication with a message passing API in a data parallel architecture, but it does not provide tools for dynamic load balancing using a pipelined approach [52]. Our multi-GPU data structure in the framework offers more flexible scheduling and load balancing in a multi-GPU computing environment. Merge provides a framework for heterogeneous scheduling that exposes a map-reduce interface [41]. Their work exposes map and reduce constructs that allow the framework to easily incorporate parallelization in various applications for CPUs and GPUs. However, they do not provide a general framework for load balancing as our work does.

The two following works present the frameworks Zippy and CUDASA that handle static communication in a GPU cluster. Fan et al. present Zippy, a GPU cluster programming framework that uses a two-level parallelism hierarchy and a non-uniform access (NUMA)

model [19]. It tries to incorporate the advantages of both a shared-memory and message passing model. It uses global arrays that act as arrays of contiguous memory across nodes to simplify communication for the programmer, and it allows the user to incorporate data locality into the system to improve performance. By allowing the programmer to control what memory is transferred when, it allows programs to have higher scalability. Zippy uses a higher level API that allows programmers to develop a system more rapidly, but it is limited to static communication. Similar to our work, they provide a framework for multi-GPU computing. However, their work is limited to static communication, while ours handles dynamic load balancing for applications that have changes at runtime. Due to its limitation to static communication, it would not be able to handle more dynamic applications that are often used.

Strengert et al. introduce CUDASA, an extension to CUDA for multi-GPU systems and GPU clusters [51]. CUDASA offers abstractions that make programming with GPUs easier by hiding scheduling and communications. Moreover, it uses unmodified CUDA code, which means that it can require very little work on the part of the programmer from an original single GPU version. CUDASA includes a runtime library, a compiler, and languages extensions with syntax similar to that in CUDA. It has four abstraction layers, including the GPU, bus, network and application layer. CUDASA significantly simplifies creating programs for multi-GPU systems and GPU clusters, but still has areas for future work. CUDASA uses Global Arrays which are extensions of global memory across GPUs, so it is limited to static communication. Dynamic communication would allow for more complex algorithms to be used. Improving load balancing for performance would also allow for efficient scheduling. Automatic use of asynchronous transfers to GPUs would be another area for improvement. Like our work, they present a framework for multi-GPU systems, but they concentrate on clusters instead of a single machine.

Other frameworks, unlike CUDASA and Zippy, provide dynamic communication that can adapt to applications that change at runtime. Stuart et al. proposes the DCGN API, a message passing interface for multiple GPUs very similar in syntax to MPI [52]. They

propose the use of thread-groups to achieve MPI ranks, and they use a polling system for checking messages. An overhead between one and five percent is introduced by using their method. The authors argue that this work improves on previous work of Zippy and CUDASA which use Global Arrays and require a static communication pattern. DCGN improves over the GPU-as-slave model (gas) which limits the flexibility of the program by requiring the GPU to work for a single CPU. DCGN allows users to program for a dynamic communication scheme, and it also abstracts away the lower level communication handling and allows the programmer to deal with higher level constructs. Future work can include finding ways to reduce the CPU overhead of polling, transfers of GPU memory, and synchronization of kernels with messages. Furthermore, in a cluster, communications between nodes are globally synchronized to account for GPU-CPU transfers. Compared to our work, they also provide a framework that facilitates GPU programming. However, use of DCGN requires low-level programming as with MPI, so it requires considerable programmer work. In addition, our work focuses on a pipelined approach with stages through a simpler interface that allows for easier incorporation of our load balancing methods.

[16] present Harmony, a runtime framework for development in heterogeneous computing environments. It provides automatic dynamic scheduling of kernels, adaptive performance optimization, and portability and scalability across different computing environment hardware. The tool can be leveraged with existing programs with limited effort. The primary abstractions in Harmony are compute kernels that provide methods for execution, and control decisions, which allow users to specify conditions for running a kernel and provide scheduling logic. Compute kernels have several restrictions, including that the memory locations of input and output arguments must be known (no embedded pointers), multiple kernel implementation must be specified separately for each type of processor, and scratchpad memory is not persistent across kernel calls. Control decisions can use various input variables to optimize performance of the application by determining the best scheduling of execution kernels. Future work could involve optimization by distributing or grouping kernel execution to reduce communication. It could also be applied to hierarchical applications, where

Harmony could be applied first to the upper cluster level and then to the lower node level. While their work supports kernels that run on both CPU and accelerators such as GPUs, our work focuses on contributions in multi-GPU dynamic load balancing using a pipelined approach.

Other works have focused on abstracting away the memory hierarchy to provide easier programming across different types of processors, such as CPUs and GPUs. Fatahalian et al. present Sequoia, a programming language that provides abstractions that allow programmers to develop computer programs that take into account the entire memory hierarchy [21]. It provides ways to describe vertical data transfers across memory levels, and to specify in what memory computations take place. Their system includes a compiler and runtime system for the Cell and distributed memory clusters. Tasks, the operational unit, allow the programmer to specify explicit locality and communication, isolation between tasks, variations on tasks, and parameterization of tasks for each different levels of memory. A major area for future work is to include abstractions that account for dynamic communication. For applications that require dynamic computations and load balancing, Sequoia would not be suitable due to its current design. Adapting this framework to work with both out-of-core and for multiple GPUs would be a good area for future work. Abstracting this memory and the streaming of data across these multiple levels would be an interesting addition to this work. While their work focuses on providing a framework that can include both CPUs and GPUs, they do not focus on a pipelined approach to load balancing as ours does.

Yan et al. presents hierarchical place trees (HPT), which simplify parallel program development by abstracting out the memory hierarchies and parallel systems [54]. It adapts the previous models X10 and Sequoia and effectively unifies CPU and GPU code. A memory module, including DRAM, cache, or GPU memory, are abstracted as places, and the memory hierarchy is abstracted as a place tree. Processor cores are abstracted as worker threads. Each place contains information describing its memory properties and amount, and the place tree helps to manage the underlying transfers of data. Future work will involve using code profilers to analyze performance and locality and offer optimizations for HPT, incorporating

aspects similar to our data-driven load balancing approach. Offering features for accounting more for data locality could improve performance in these applications. Furthermore, HPT could be adapted for non-shared memory systems where greater communication could have a larger impact on performance. While this work presents a framework that helps to unify GPU and CPU code, it does not focus on a pipelining a approach that ours does.

2.3.2 Frameworks for Performance Models and Optimizations

Other works have proposed GPU frameworks for performance models and optimizations for use with different applications. Baghsorkhi et al. propose a GPU performance modeling tool, and through use of a work flow graph they analyze a GPU kernel to predict performance for an application [6]. However, this work focuses on single GPUs and could be extended to multiple GPUs. Ocelot provides a dynamic compiler that can optimize the scheduling and load balancing for multi-core CPUs and GPUs based on performance analysis [17]. Kerr et al. present a tool for analysis and modeling to help automation of program optimization for heterogeneous GPU-CPU systems [36]. They utilize Ocelot for their dynamic compiler in order to convert kernels to multi-core CPU code. They evaluate 25 CUDA applications on 4 GPUs and 3 CPUs, and they use 37 metrics in order to model and predict behavior of the program on different hardware setups. Their work demonstrates that it is possible to predict the performance of CUDA kernels. Their statistical model is based on PCA (principle component analysis) to find the most critical metrics. Their method uses various metrics, including branch divergence, instruction counts, memory intensity, registers per thread, and others. Their five principle components include MIMD parallelism, problem size, data dependencies, memory intensity, and control flow uniformity. Becchi et al. present another work for unifying CPU and GPU development which includes a runtime scheduler and memory manager [7]. However, unlike our work these works do not focus on a pipelined approach through a general framework.

Lefohn et al. propose Glift, which provides abstractions for allocating and accessing data

structures in a similar way to previous libraries such as STL and Boost [40]. This allows programmers to more easily manage and iterate over complex indexing structures such as octrees in a manner that efficiently uses the GPU hardware. Glift uses a virtual memory and address translator component to abstract out the indexing of GPU memory. Furthermore, it exposes iterators that add familiarity from CPU libraries and allow for various efficient patterns of access on the GPU. Future work could involve creating non-indexable structures such as graphs and linked lists. Moreover, they do not virtualize memory on multiple GPUs, account for limitations in the amount of GPU memory, or provide a file format for persistently saving data structures. They also do not address out-of-core streaming from hard disk memory to main memory and then to GPU memory, and they do not address usage of all types of CUDA memory, including global, texture, shared, and constant. The CPU features also do not take advantage of multiple CPU cores. While Glift provides a GPU data structure as ours does, it does not provide a framework for multi-GPU load balancing as ours does.

Other work in frameworks have included Brodman et al., which presents a survey and guidelines for using abstractions for parallel programming [10]. It outlines guidelines including the use of abstractions for encapsulation, allowing for either explicit or implicit synchronization, and portability across multiple types of processors. Furthermore, they advise that the programmer should be able to partition the memory for distribution in a heterogeneous environment. They also describe Hierarchically Tiled Arrays, which splits up memory into multiple sub-arrays for efficient parallel processing. The techniques and guidelines they present would be very useful in developing new abstractions and data structures for multiple GPU systems. Some related areas of current research include collections, dynamic tiling or partitioning, interactions in parallel operations, abstractions for portability across different processor types, and abstractions for optimization. While they provide abstractions for multi-GPU systems, they do not provide a pipelined approach for dynamic load balancing as ours does.

Quintana-Ortí et al. describes the application of linear algebra libraries FLAME and SuperMatrix to multicore and multi-GPU systems [45]. Their method uses the FLASH API that

uses hierarchical submatrices to decompose matrix computations. Few changes were needed due to the scheduling provided by SuperMatrix that utilizes data locality and coherence to improve performance. SuperMatrix handles scheduling, task decomposition, and assigning tasks to cores without the need of programmer intervention. Future work could involve making the algorithms support other types of data structures besides matrices. Furthermore, the system could be made more flexible by allowing for more programmer extensions to the functionality of the system. Similar to our work, they take advantage of multiple GPUs. However, they do not focus on a pipelined approach for load balancing as ours does.

Danalis et al. provides a set of benchmarks, called Scalable Heterogeneous Computing (SHOC) benchmarking suite [15]. They test SHOC with GPU and CPU applications to determine performance and communication costs. The benchmarks have different versions with various levels of parallelism and communication to characterize performance. Similar to our data-driven load balancing method, they use past performance to improve performance. However, their method requires significant performance tests to be useful for an application. Gelado et al. propose a new programming model, Asymmetric Distributed Shared Memory (ASDM), that manages shared memory that allows CPUs to access accelerator memory [24]. They propose an implementation called GMAC that eases programming and improves portability. It facilitates load balancing by transparently scheduling a function to run on an accelerator. It attempts to increase throughput and minimize latency in scheduling in order to optimize the performance of applications. While they provide a way to make multi-GPU programming easier, their work does not involve dynamic load balancing as ours does. Guim et al. present a method for scheduling and resource management in systems containing GPUs and CPUs [26]. The scheduling method takes into account various factors, including performance, communication, and energy usage. They offer different resource selection schemes in order to account for these factors. Like our data-driven load balancing method, they can take past performance into account. However, they do not focus on dynamic load balancing using a pipelined approach.

Gummaraju et al. propose Twin Peaks as a computing platform for use with GPU and CPU

systems [27]. It schedules tasks to run on CPUs and GPUs based on various factors, including cache coherence. It maps tasks as thousands of threads that can either be scheduled on GPU multi-processors or CPU cores. Their method takes into account factors like performance as our data-driven method does. In future work they could offer automatic dynamic load balancing similar to our method. Hong et al. present MapCG as a MapReduce-based programming model to allow software developers to run programs for GPU and CPU systems [31]. While their work presents a method for distributing work using map-reduce to multiple GPUs, they focus on a top-down approach to data distribution, unlike our pipelined approach. Ping et al. present a method for load balanced computation of similarities using Hidden Markov Models (HMMs) for protein sequences [56]. By taking advantage of both CPU and GPU, they gain a speedup over a pure GPU program. This use of concurrent computation demonstrates how host and device memory can be managed to effectively improve an application, but they do not focus on the same multi-GPU pipelined approach as we do. Lee et al. propose a compiler framework that converts OpenMP into CUDA programs to utilize the massive parallelism of GPUs [39]. Their work offers a method to greatly ease GPU programming, but they do not focus on dynamic load balancing as in our work.

Ryoo et al. propose methods for GPU optimization of matrix multliplication, including subexpression elimination, loop unrolling, occupancy, and register and shared memory usage [46]. They apply the tests to various applications, but in future work they could use these results to provide more automated optimizations. Ryoo et al. extended the previous work by addressing GPU programming as an optimization problem with multiple variables [47]. They incorporate a number of metrics, including performance, memory bandwidth, and occupancy. They select the parameters through optimization carving that determines the most critical components to improve performance. This method is more similar to our data-driven method that takes into account past performance. In future work their method could take into account more dynamic factors such as application parameters that can change at runtime as in our work.

Yixun et al. present G-ADAPT, a GPU adaptive optimization framework [42]. It takes into

account various program parameters, and then uses statistical learning for optimization. This method presents a similar approach to our data-driven load balancing. Other possibilities could include taking into account additional dynamic factors for workload changes at runtime. Becchi et al. present methods for data-aware scheduling of kernels using heterogeneous computing that uses virtual memory across GPUs and CPUs to ease programmer effort [7]. Similar to our work, they provide a framework to simplify programming for the developer. However, they do not focus on a pipelined approach for multi-GPU computing as our work does.

2.3.3 Higher Level Programming Frameworks

Several other works provide frameworks that attempt to provide higher level programming to make GPU development easier. Ueng et al. present CUDA-lite to assist programmers dealing with the complex memory hierarchy in CUDA programming environments. The user specifies the desired properties of data structures and control flow of the execution with annotations, then CUDA-lite generates the optimized kernel code for memory management with the information [53]. However, this work does not address a pipelined approach for multiple GPUs as our framework does. Kloeckner et al. propose PyCUDA, a system for programming for the GPU using Python [1]. In this system Python can be used to combine GPU code, and they offer abstractions such as GPUArray to make GPU programming easier. While this work presents a framework and memory management to ease GPU programming, they do not provide a general method for load balancing. Similarly, Han et al. present hiCUDA, which allows for high-level directives to be used in place of CUDA code [29]. It allows programmers to produce more readable CUDA code, without having to include explicit code for memory transfers and other aspects. This helps to make the GPU development process easier through abstractions, but they do not focus on load balancing methods as in our work.

Hou et al. propose BSGP, a C-like language that compiles to CUDA code [32]. It is based

on the bulk synchronous programming (BSP) model which allows for more intuitive programming than CUDA without needing to explicitly handle memory transfers. While this work helps to unify CPU and GPU code for more rapid development, it does not apply to multi-GPU systems. Stratton et al. provide a method for translating CUDA code into multicore CPU code [49]. This provides a method for integrating CPU and GPU code. Stratton et al. extended the previous work to analyze how SPMD programs such as CUDA can be mapped to a multi-core CPU system [50]. This work presents a framework that can help to unify CPU and GPU programming. However, they do not focus on dynamic load balancing for multiple GPUs as in our work. Yan et al. present JCUDA, a compiler that allows for Java programming for use with NVIDIA's CUDA [55]. This allows for a higher level, object-oriented interface in place of the original C language. This can make development for the GPU simpler, and in future work they could extend this work to multiple GPUs. These frameworks present methods that make it easier to develop CUDA programs. Our general framework also provides a method to easily incorporate use of multiple GPUs and accelerate existing applications. However, these previous works do not focus on providing a framework that incorporates multi-GPU dynamic load balancing using a pipelined approach as ours does.

As demonstrated, many previous works have proposed multi-GPU frameworks to ease developer effort. However, many have not focused on visualization or on a pipelined approach with multiple stages. Thus, we propose a framework that provides load balancing based on a pipelined approach for easy incorporation into existing programs.

Chapter 3

Problem Statement

Use of multiple GPUs can greatly improve performance but requires load balancing to ensure that all processors are utilized. Use of a pipelined approach can be useful for multi-GPU processing when using multiple producer and consumer stages. These stages can write and read data in multiple buffers on the host to allow for concurrent computations. Each stage can then be distributed to the optimal number of processors in order to keep workload balance and achieve better performance. We present the problem of load imbalance introduced by using multiple GPUs that motivates the use of our multi-GPU load balancing methods.

3.1 Multi-GPU Configuration

In comparison with use of multiple GPUs, a single GPU implementation has significant limitations. Use of a single GPU eliminates the need to transfer data that can remain in GPU memory. However, a single GPU implementation requires sequential execution of programs since only one GPU can run tasks. Use of multiple GPUs allow for a significant performance gain by allowing for concurrent computation on many processors at once, but it requires data management of transfers of data to and from host memory.

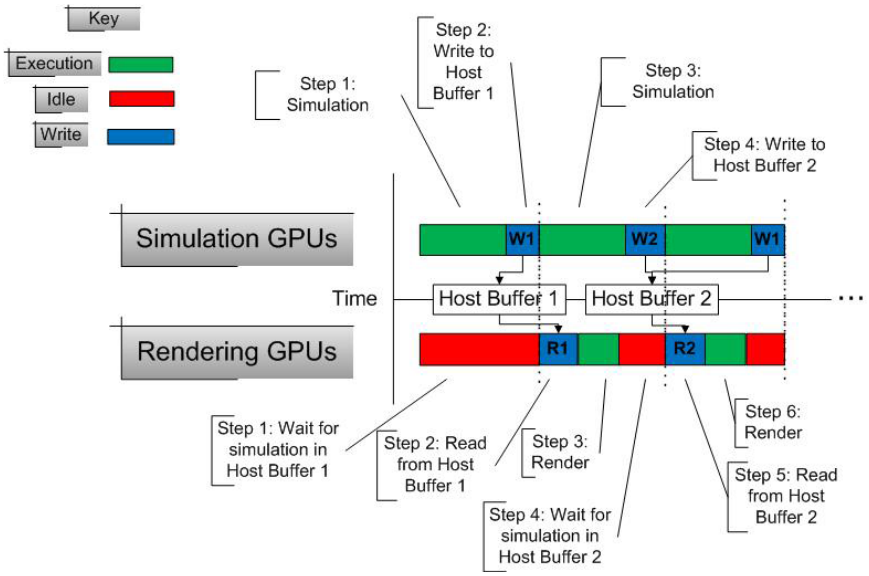

Figure 3.1: Use of double buffer where the rendering task waits.

Use of a double buffer is a standard data management scheme in visualization that allows for concurrent computations with the use of two host buffers that can be read and written by GPUs. The double buffer allows for a pipelined approach where each GPU can process a single stage of the computation and pass the data onto another GPU. By swapping the buffer currently being written or read, one task can read while another writes. This allows for the ability to overlap computations and transfers to different buffers. However, one task may still have to wait for data to be read or written by another task, as shown in Figure 3.1. This figure demonstrates that if the consumer stage takes less time than the producer, the consumer task will have to wait for the producer to write to memory to the host. The double buffer allows for concurrent computation for simulation and rendering that allows for in-situ visualization. However, the idle time introduced by load imbalance can reduce performance in the entire system.

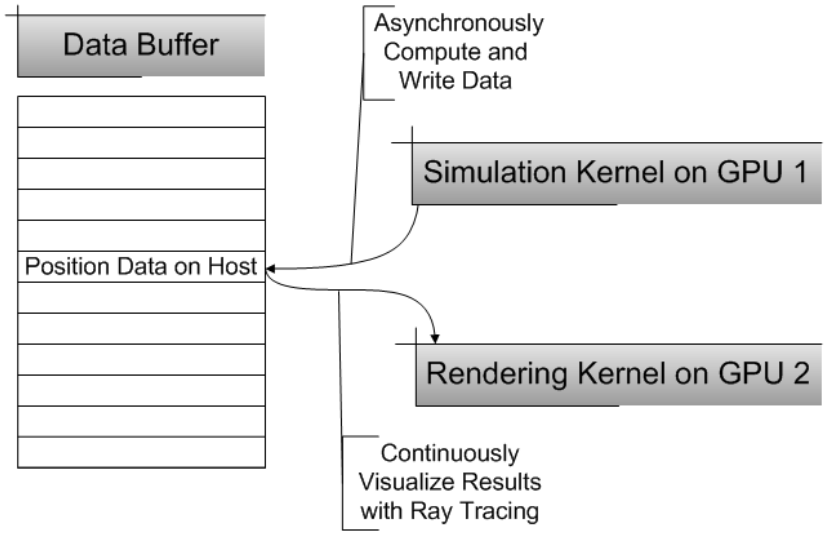

Figure 3.2: Memory transfers with Data Buffer Queue.

3.2 Multi-GPU Streaming

An alternative method that can address some of the issues of the double buffer is the Data Buffer Queue, a multi-GPU data structure implemented in our method that can store multiple frames of data in host memory as shown in Figure 3.2. The use of multiple storage buffers requires more memory, but it allows for more frames for concurrent computation. More than one task can carry out computations on separate frames of data in parallel due to a greater number of data slots. The Data Buffer Queue therefore allows for more flexible load balancing by exposing several frames of data for writing and reading. Without dynamic load balancing, however, the Data Buffer Queue can eventually become full and can suffer from the same underutilization as a double buffer with unequal workloads as shown in Figure 3.3. This shows how simulation may eventually need to wait if rendering takes longer and the buffers eventually become full. While multiple frames of simulation can be written before introducing idle time, the load imbalance eventually hurts performance as much as idle time with the double buffer.

Idle time can also occur if simulation takes longer than rendering, as shown in Figure 3.4.

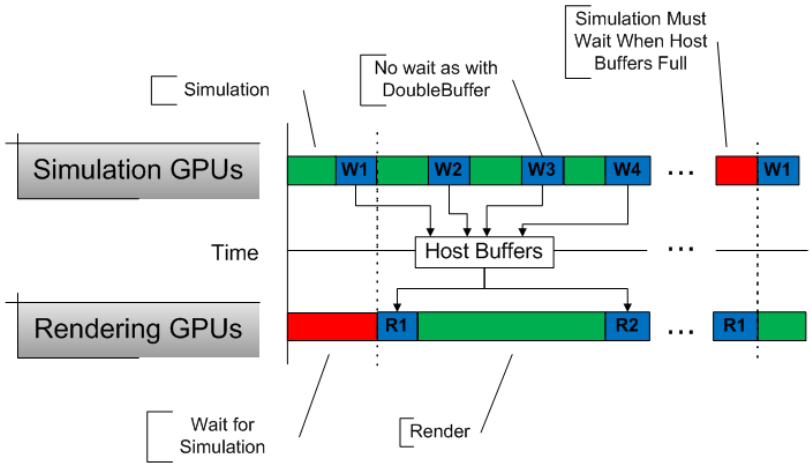

Figure 3.3: Use of Data Buffer Queue where simulation GPUs wait.

However, this case causes immediate waiting since the host buffer starts out empty and rendering tasks need to immediately wait until simulation has written data. In this case the Data Buffer Queue has no advantage over the double buffer due to the immediate waiting.

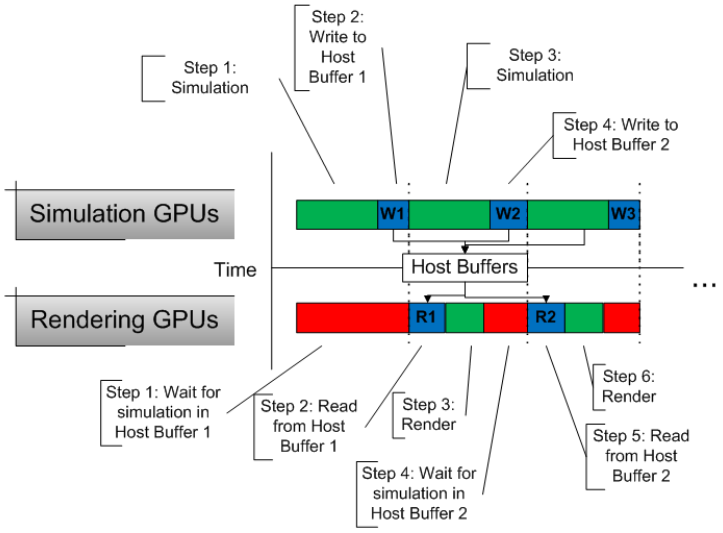

Figure 3.4: Use of Data Buffer Queue where rendering GPUs wait.

In order to solve this issue of workload distribution and idle time, load balancing methods

must be applied which make the workload between simulation and rendering more equal. As shown in Figure 3.5, the goal is to obtain an equal rate in and out to avoid idle time due to an empty or full buffer. By applying load balancing, the utilization of all processors can be improved to increase the performance of the entire system.

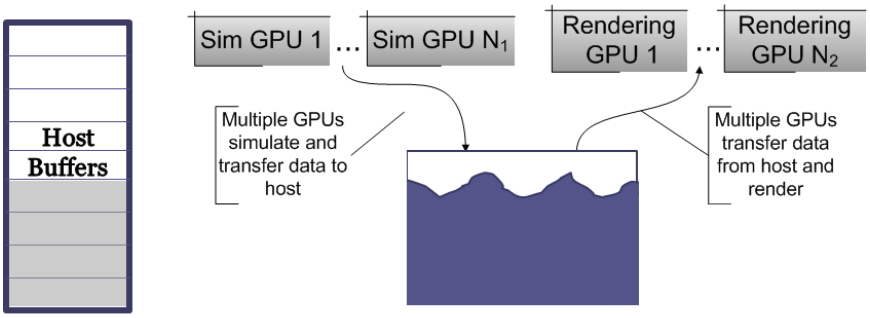

Figure 3.5: Metaphor of a water basin to host memory, where the goal of load balancing is to achieve an equal fill and empty rate to avoid a full or empty buffer.

Chapter 4

Data-driven Load Balancing

In order to address the issues of load imbalance for applications with predefined configurations, we present the first load balancing approach, data-driven load balancing. This method samples the input parameter space for performance data and uses this data to predict the optimal load balance for a new set of input parameters.

4.1 Data-driven Load Balancing Approach

Our data-driven load balancing approach determines the optimal static workoad distribution based on past performance data. Our test application for the load balancing methods uses a gravitational N-body simulation to compute interactions of particles, while ray tracing renders the results. The N-body simulation is based on the algorithm of Nyland et al. [38]. Particle position data is transferred through the host using multiple buffers in order to pipeline position data among simulation and rendering tasks. Since simulation and rendering may have different amounts of workload, it is important to address the possibility of workload imbalance between the tasks. The use of data-driven load balancing can achieve the optimal predefined workload distribution as shown in Figure 4.1. This can greatly reduce idle time

by making the rate of read and write for the buffer approximately equal.

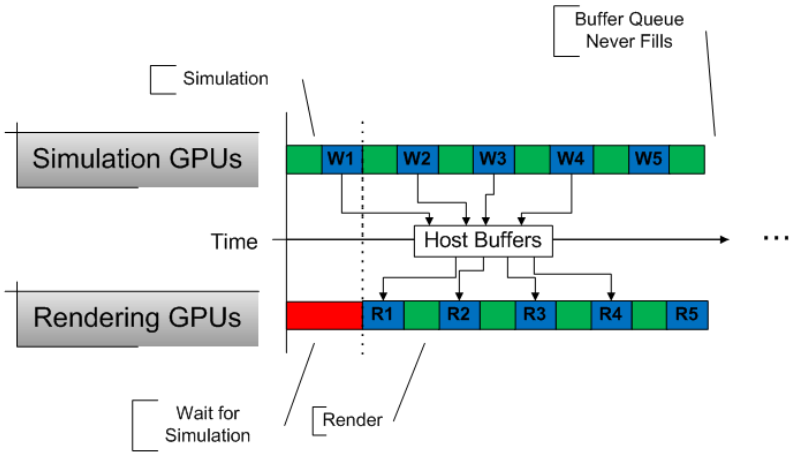

Figure 4.1: Data-driven load balancing determines optimal static workload distribution to avoid idle time.

The goal of the data-driven load balancing method is to find the optimal load balancing configuration M for a set of input parameters. The optimal load balance in this system is defined as M, the total number of rendering GPUs allocated for rendering that leads to optimal performance. M can range from 1 to N-1, where N is the total number of GPUs for load balancing. In our system N is seven out of eight total GPUs, since the eighth GPU is used for displaying the final image. The data-driven load balancing approach works by predicting the optimal load balancing for which the execution time t is minimized for a given set of input parameters:

$$M = \underset{M_S=1,\ldots,N-1}{argmin}(t)$$

where M is the number of rendering GPUs for the optimal load balance, M_S are tested configurations, t is execution time, and N is the number of GPUs available for load balancing.

The execution of the data-driven load balancing method is shown in Figure 4.2. As shown, all tasks begin with initialization to allocate host and GPU memory. Afterwards, GPUs continually carry out the execution loop, which involves reading, execution, and writing of

results. A barrier is used for simulation for every frame to ensure results are written together, while each rendering GPU can execute separately and does not require a barrier.

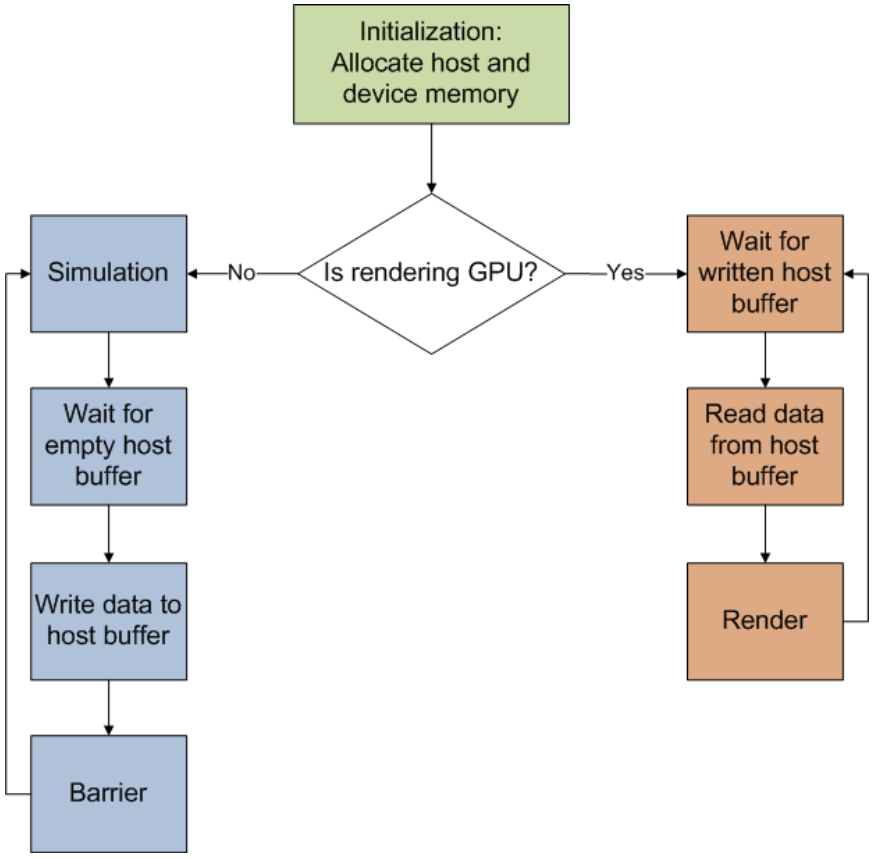

Figure 4.2: Flowchart showing the execution for data-driven load balancing

4.1.1 Task Partitioning

Load balancing for visualization requires partitioning the dataset in order to distribute work to tasks. Two types of work partitioning are possible with the application: inter-frame and intra-frame, as shown in Figure 4.3. Inter-frame partitioning involves distributing complete frames of data in order to achieve load balancing. Ray tracing in this implementation uses inter-frame partitioning to render individual frames in order to avoid communication in combining results. Since rendering each frame only depends on the required simulation

data, each frame can be rendered independently. The N-body simulation utilizes intra-frame partitioning by splitting up a single frame of data. Since each frame of simulation requires previous data, simulation for consecutive frames cannot be computed concurrently. Thus, multiple GPUs can only accelerate simulation as a group by having each GPU update a subset of the dataset. Thus, intra-frame partitioning is necessary for simulation to improve performance through load balancing. While this requires communication to combine the results for each frame, the computation can be distributed among multiple GPUs.

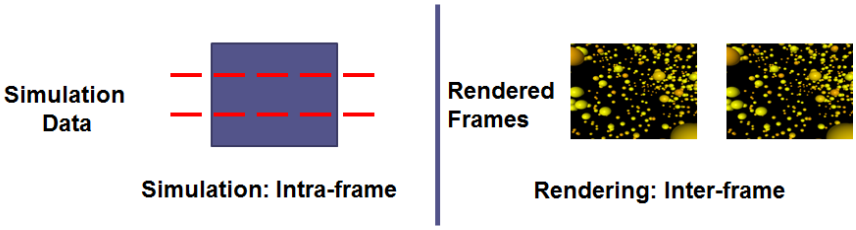

Figure 4.3: Intra-frame and inter-frame task partitioning schemes for simulation and rendering, respectively

The partitioning of both rendering and simulation tasks allow for load balancing in the visualization application. As more processors are dedicated to simulation, fewer are dedicated to rendering consecutive frames. The total computation time for a frame is used to determine the optimal load balance between rendering and simulation among the available processors. Since an arbitrary number of simulation tasks can process a single frame of data, this allows the simulation group to grow and shrink as necessary. Similarly, the number of rendering tasks can also increase or decrease since there is no dependency between any two rendering tasks, but the rendering execution time for a single frame does not change with different workload distributions since each frame is rendered separately. Synchronization for simulation uses the barrier implementation provided by our framework to ensure that results are written to host memory. Rendering tasks in the visualization, on the other hand, can process consecutive frames independently and do not need direct communication with each other. Therefore, each rendering GPU carries out ray tracing on a single frame only and transfers the frame to host memory without direct synchronization with other rendering GPUs. A

separate display GPU then reads this frame and displays it for viewing.

4.1.2 Preprocessing and Interpolation

To address the issue of workload imbalance, we present a load balancing method for predefined configurations to achieve the optimal distribution of work to improve performance. We present a three-dimensional parameter matrix P that can be used to determine the appropriate balance for our visualization application, as shown in Figure 4.4. The dimensions of P include the number of samples for supersampling, the number of iterations for simulation, and the input size, while the associated values are the optimal load balance for a set of input parameters. The data-driven method requires a preprocess step where the optimal load balance M of each point on the matrix P is first determined for the entire desired input parameter space.

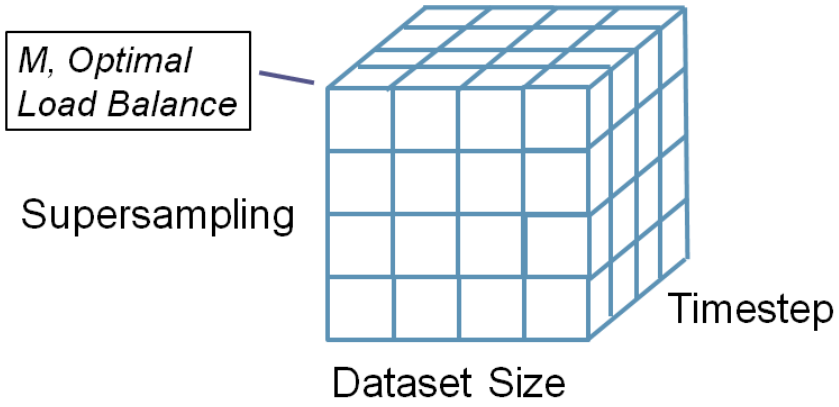

Figure 4.4: Input parameter matrix P that is sampled as a preprocess for data-driven load balancing

For each point on the grid, each combination of rendering and simulation GPUs is tested for execution time, and the number of rendering GPUs with the shortest execution time is used as the optimal load balance for that point. Based on the previous performance results for this matrix, the method computes the optimal workload balance for a new set of input

parameters. For a new set of input parameters, we thus solve the function:

$$f(i, s, p) = M$$

where f is the function to compute optimal workload distribution, i is the number of iterations for simulation, s is the number of samples for supersampling, p is the number of particles in the simulation, and M is the optimal number of GPUs allocated for rendering versus simulation. The resulting load balance is computed by interpolating between previously tested results. Thus, given known optimal workload distributions for given input parameters, our model deduces the load balancing result M for a new set of input parameters through trilinear interpolation:

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

where M_{isp} is the optimal load balance, i is the number of simulation iterations, s is the number of samples for ray tracing, and p is the number of particles. This result gives a prediction for the optimal load balance for a given set of input parameter values.

4.1.3 Test Application

The load balancing methods were tested with an N-body simulation that uses ray tracing to visualize thousands of spheres. A final result of the simulation and rendering is shown in Figure 4.5. The colors of the particles represent different masses that affect the simulation. The interactive application allows the user to navigate the dataset by rotation and zooming with the mouse.

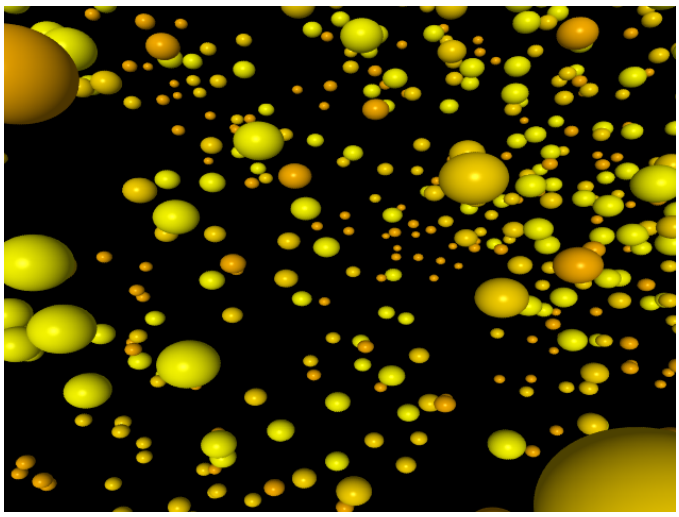

Figure 4.5: Example image of the N-body Rendered Simulation

4.2 Data-driven Load Balancing Results

Supersampling was used to improve the quality of rendering for the in-situ visualization application. The differences in supersampling can be seen in Figure 4.6. 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, however, significantly improves the results.

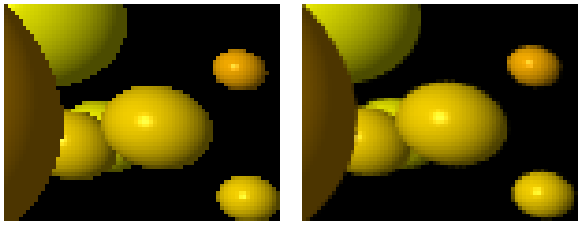

Figure 4.6: Comparison of single sample (left) and 16 sample randomized supersampling (right)

While supersampling improves the quality of the final image, it comes at a performance cost as shown in Table 4.1. The increase in execution time for a greater number of samples for supersampling is linear. Thus, the tradeoff between performance and image quality must be

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

Table 4.1: 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 4.2: GPU execution time for simulation based on number of iterations

considered when choosing an appropriate number of samples for supersampling.

Table 4.2 shows the time for simulation when performing multiple iterations. The performance of simulation shows a linear increase in time with an increase in number of iterations. While a smaller timestep provides more accurate simulations, it requires more iterations to advance the simulation for each frame. Thus, using a smaller timestep but increasing the number of iterations leads to an increase in execution cost.

Table 4.3 shows the percent difference in positions of simulation from 12000 iteration simulation, which uses the smallest timestep. Each simulation is carried out over the same total time, with a smaller timestep for simulations run for more iterations. With a smaller timestep, the accuracy of the simulation is improved due to the finer granularity used for integration in the N-body simulation.

Simulation is split up among a number of GPUs for workload distribution. Figure 4.7 shows the performance speedup for execution time of simulation on multiple GPUs. This shows that there is an approximately equal speedup in execution time when using multiple GPUs since each GPU can compute simulation on an increasingly smaller subset of the entire dataset.

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

Table 4.3: Percent difference in positions of simulation from 12000 iteration simulation

This contrasts with the task partitioning for rendering which still has the same execution time for each GPU but has additional GPUs to process separate frames concurrently.

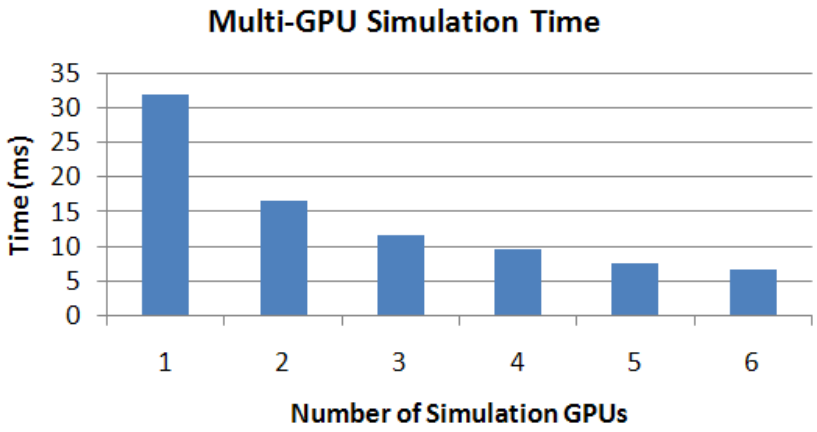

Figure 4.7: Performance of simulation on a different number of GPUs

Thus, the number of samples for ray tracing and the number of simulation iterations each affect the final workload costs. Differences such as these in workload affect the final optimal load balance.

4.2.1 Workload Characteristics

The multiple input parameters for this application result in many possibilities for varying workloads. These varying workloads can introduce a performance penalty if not accounted for in distribution of workloads. Figure 4.8 shows the trends for performance times for different workloads (number of ray tracing tasks) with varying input sizes using sixteen samples for ray tracing and eighty iterations of simulation. The minimum along each line represents the optimal load balance since it has the shortest execution time. The cost of simulation increases at a faster rate as dataset size increases due to the nature of the N-body simulation, while ray tracing scales linearly with dataset size. This causes the overall performance to be increasingly limited by simulation time for larger datasets.

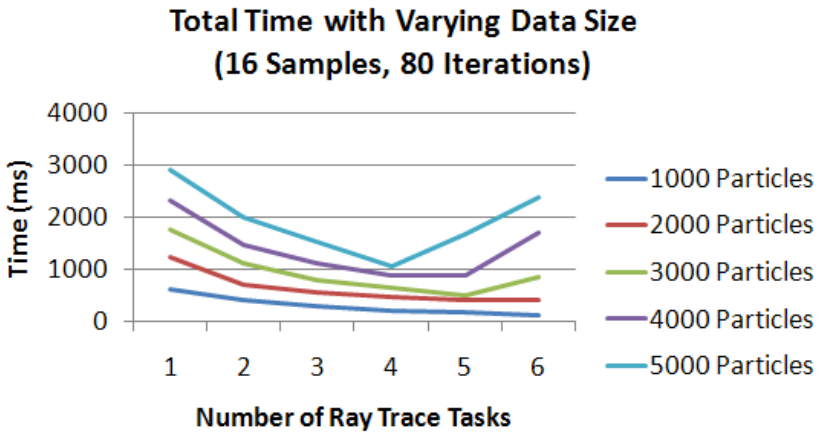

Figure 4.8: Performance time for various input sizes with 16 samples, 80 iterations

Figure 4.9 shows performance for different input sizes 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 trend from Figure 4.8 also demonstrates that different input parameters can lead to significantly different optimal workload distributions that requires load balancing.

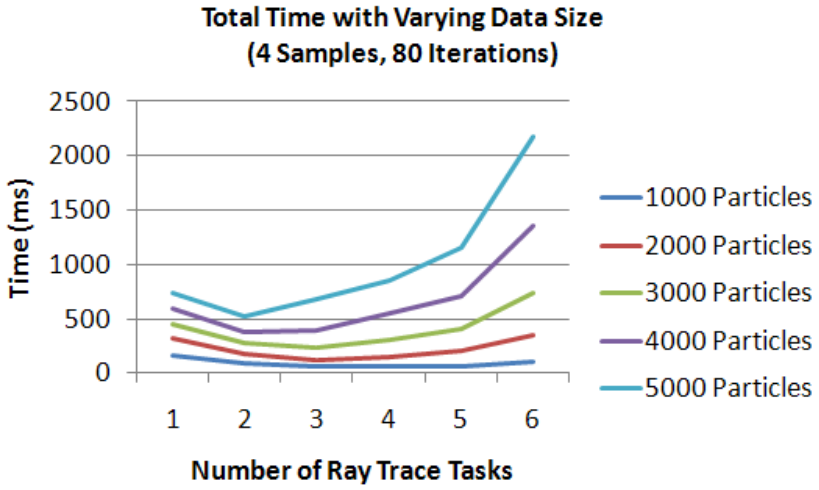

Figure 4.9: Performance time for various input sizes with four samples, 80 iterations

Figure 4.10 shows the trend for varying simulation with a constant number of samples and

input size. This diagram shows that for a small dataset size, 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.

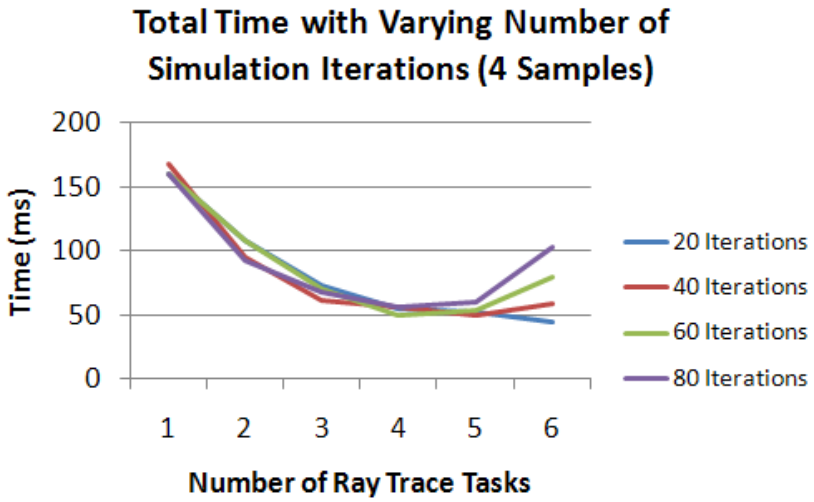

Figure 4.10: Load balancing for various simulation iterations with four samples, 1000 particles

In comparison to the previous graph, Figure 4.11 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.

4.2.2 Load Balancing

The optimal load balancing data demonstrate trends in the data that must be accounted for by data-driven load balancing. Figure 4.12 shows a trend of optimal load balance based on the number of iterations for simulation. As shown, increasing the number of iterations

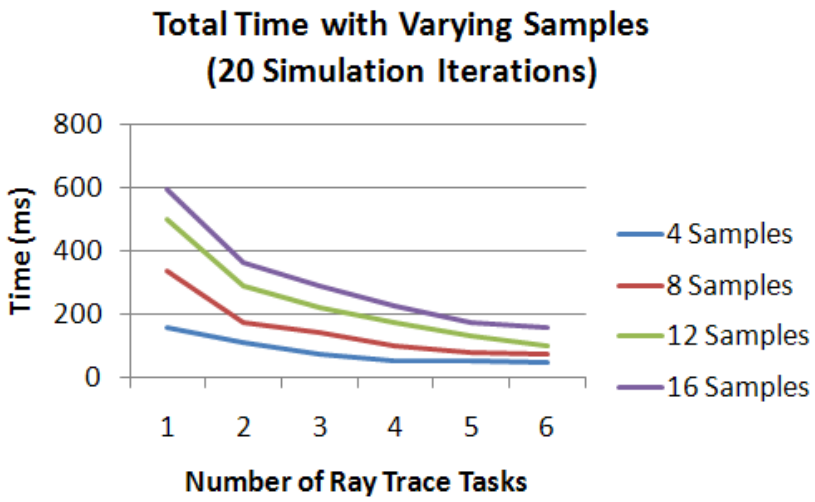

Figure 4.11: Load balancing for various simulation iterations with twenty simulation iterations, 1000 particles

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. We also see that since the load balance is limited to a discrete number of GPUs, many of the series have horizontal lines connecting two data points. This is a limitation of data-driven load balancing that restricts the granularity of workload distribution.

Increasing the number of samples for supersampling increases the cost of ray tracing and also impacts the load balancing scheme. Figure 4.13 shows that increasing the number of samples for supersampling results in need of additional ray tracing GPUs to improve workload balance. A larger dataset size requires fewer GPUs for ray tracing due to the smaller increase in cost of ray tracing with larger datasets.

These results demonstrate that significant workload imbalance can be introduced based on differing workloads of rendering and simulation. Additional samples for supersampling improves the quality of the final images, but increased workload can require additional GPUs to be allocated for ray tracing to maintain performance. Similarly, increasing the

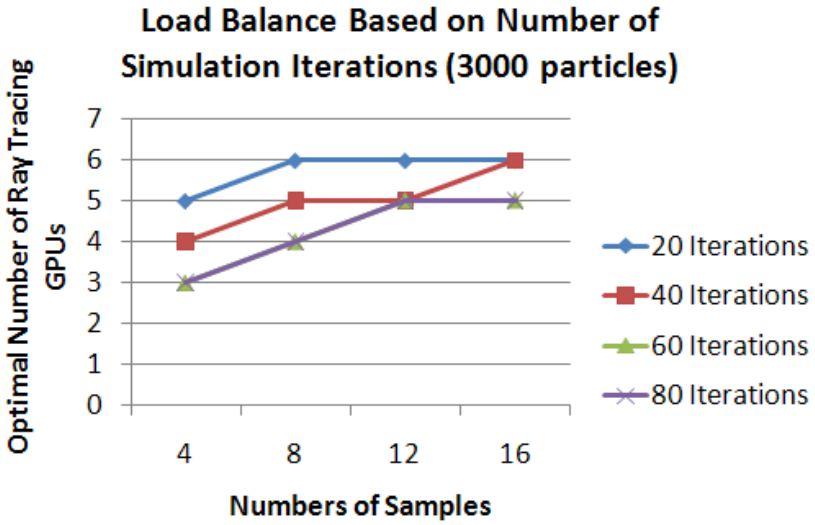

Figure 4.12: Load balancing for various simulation iterations with 3000 particles

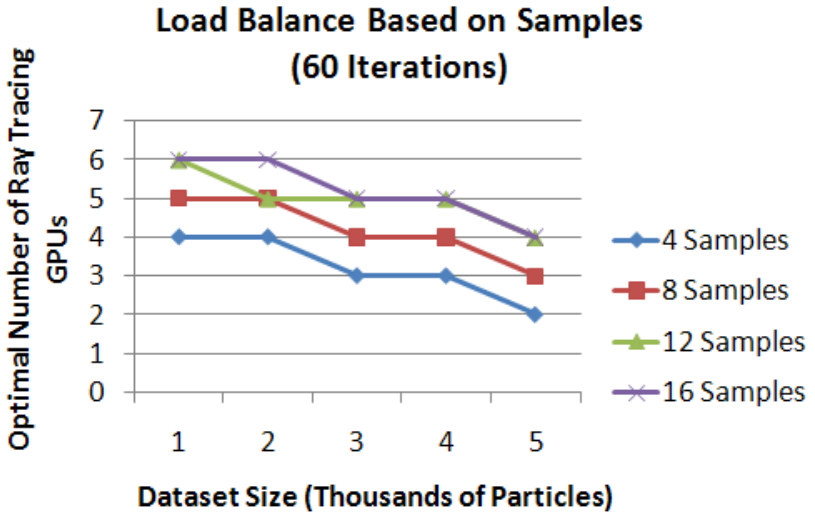

Figure 4.13: Load balancing for various numbers of samples for ray tracing with changing dataset size

number of simulation iterations increases the workload of simulation that requires increased computational power to achieve load balance. Each configuration leads to a different optimal load balancing configuration. These results show the significant speedup between the worst

and best cases of load balancing.

4.2.3 Performance Model

We present a summary of the results of applying our load balancing performance model. We tested the performance model by computing the average percent error for 12 samples using three dimensions for interpolation (iterations, samples, and data size). These results show that the average percent difference is 1.67 percent for interpolation using three dimensions, which demonstrates that the method is accurate at predicting the optimal static configuration. These results show that using the model, the optimal static configuration can be reliably found in order to improve utilization in the system.

Chapter 5

Dynamic Load Balancing

Although data-driven load balancing is suitable for applications with predefined configurations, it has several limitations. It requires past performance data, does not adapt to changes in workload at runtime, and only handles a discrete load balance of GPUs. To address these issues, we present the dynamic load balancing method.

5.1 Dynamic Load Balancing Approach

We present the dynamic load balancing method that addresses the limitations of data-driven load balancing. The use of the provided multi-GPU data structure allows for dynamic load balancing using a pipelined approach by providing multiple frames for concurrent reading and writing. To determine when to apply load balancing, a metric must be used. Since a producer task must wait if the buffer is full or a consumer task must wait if a buffer is empty, the processor can instead switch tasks to avoid idle time. Our implementation allows dynamic load balancing to take into account fullness of the host buffer at runtime to avoid this idle time. For example, if the frames of data become full in the Data Buffer Queue, then dynamic load balancing can redistribute more work through a task switch to simulation (see

Figure 5.1). This task switch works by switching a single GPU from one task to another, one at a time. When the buffer is full, a GPU switches from simulation to rendering to read more frames. When the buffer is empty, a GPU switches from rendering to simulation to write more frames.

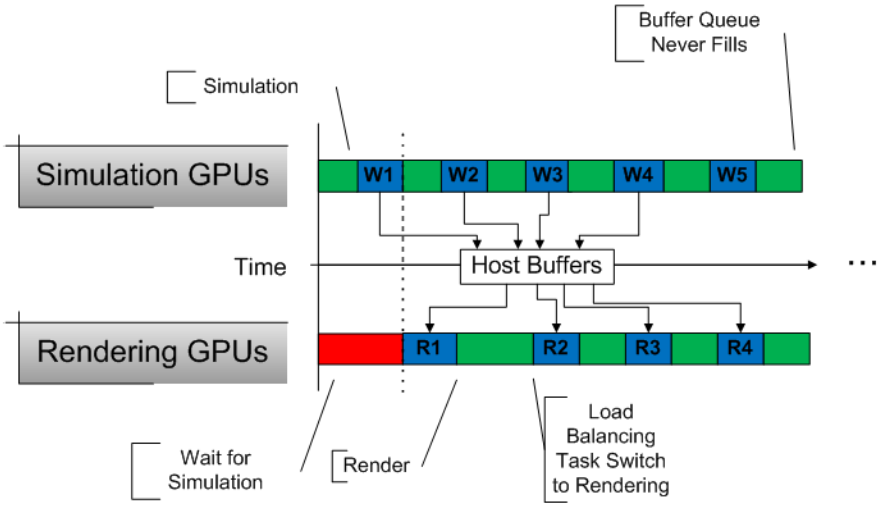

Figure 5.1: Use of Data Buffer Queue and dynamic load balancing reduces idle time

In order to provide dynamic load balancing for various applications, we implement scheduling and synchronization necessary to distribute workload at runtime to GPUs. Figure 5.2 shows the scheduling steps for execution of an application.

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 continually executes each stage.

After the execution loop begins, a stage may either execute with a group of processors or individually. A barrier is used to implement synchronization for per-frame or data parallel partitioning within a group. This ensures that all of the data for a frame is written for a group that is producing a buffer of data. This also requires a task that switches and joins this group to wait until the group reaches the barrier, as shown in Figure 5.3. However, this

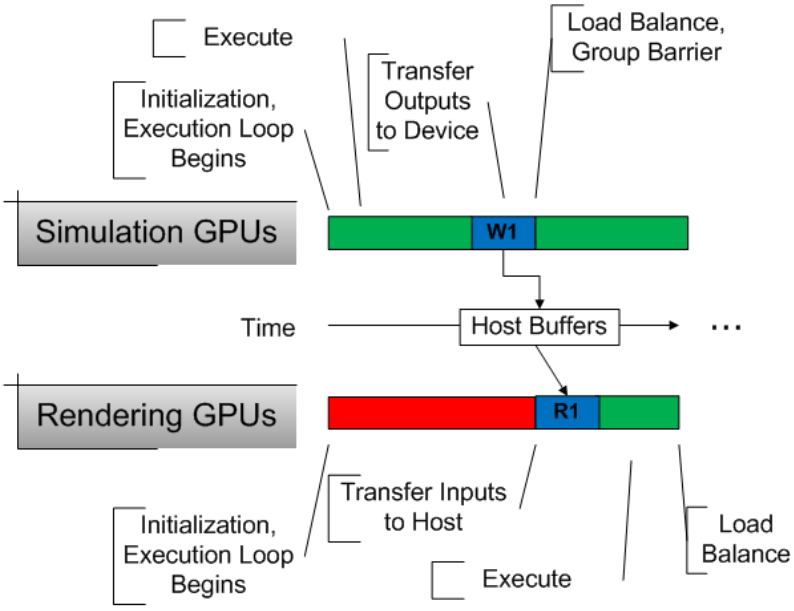

Figure 5.2: Scheduling steps for load balancing

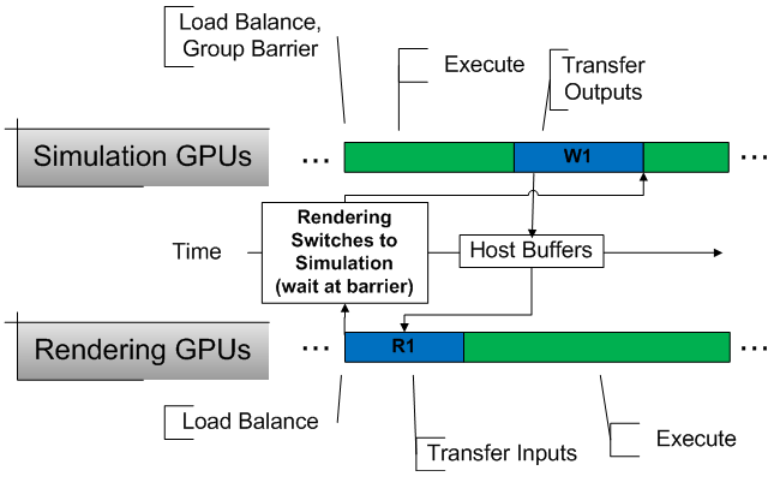

Figure 5.3: A GPU switching from rendering to simulation must wait at a barrier

group barrier is unnecessary for tasks that process data individually and do not require data parallel partitioning, as shown in Figure 5.4. For example, if a data buffer is full and a task should switch to a group of consumers to read data at a higher rate, then the task must

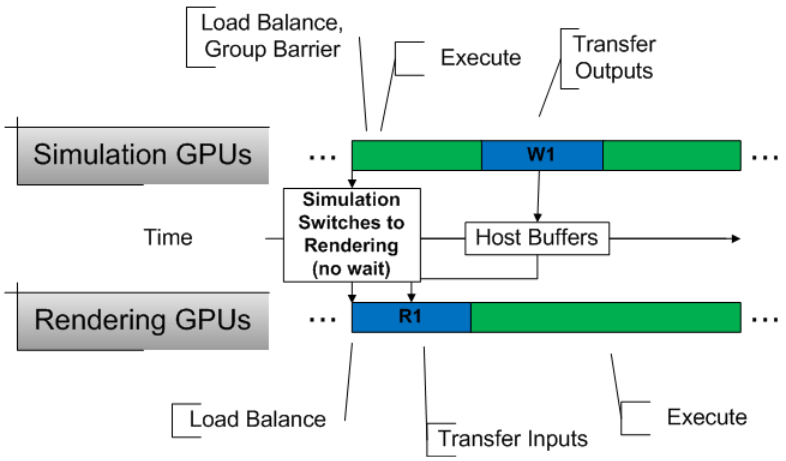

Figure 5.4: A GPU switching from simulation to rendering does not introduce additional synchronization

wait until the group of producer tasks have completed writing the current frame to join. If a GPU should switch from producer to consumer, then the GPU can switch without waiting for other consumer tasks if each consumer task executes individually.

The detailed algorithms for simulation and rendering are shown in Figure 5.5. It is similar to the process for data-driven load balancing, with additional steps for checking to switch tasks. This algorithm works by switching a GPU to rendering when the host buffers are full and there are available simulation GPUs to switch, or by switching a GPU to simulation when the host buffers are empty and there are available rendering GPUs to switch. At least one simulation and rendering GPU are required to execute at a given time. Only one GPU switches at a time to avoid idle time for full or empty host memory. While this can introduce oscillation between two configurations, this method ends up producing the optimal configuration on average.

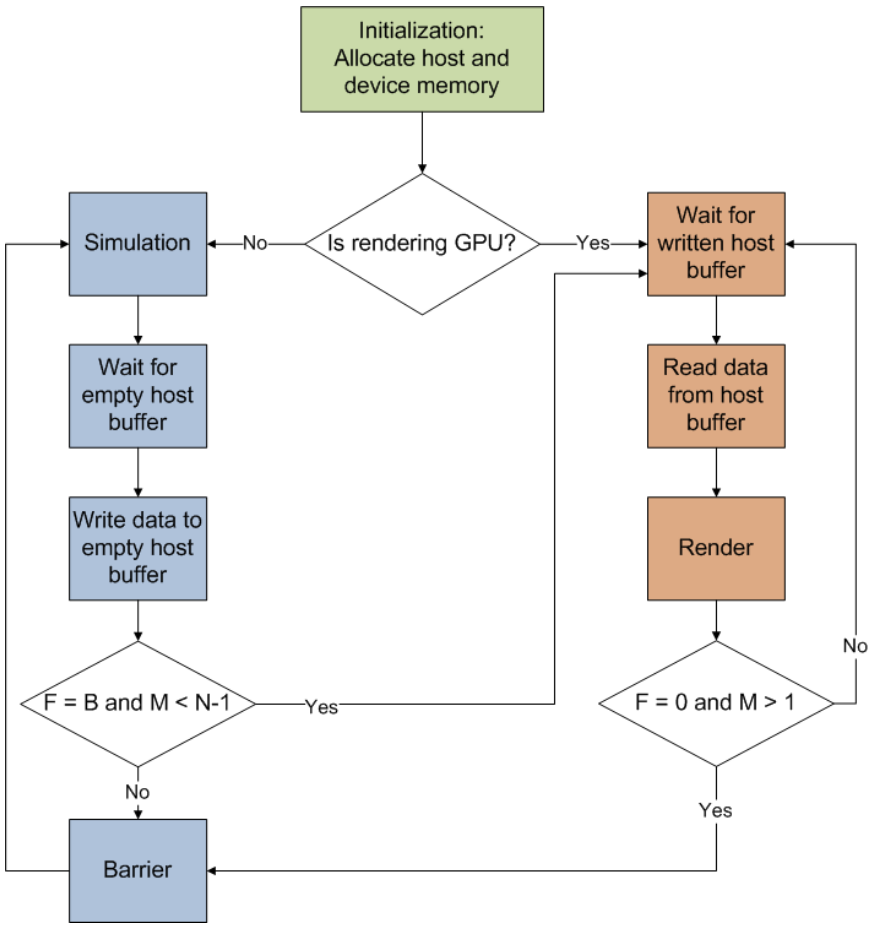

Figure 5.5: Flowchart showing the execution and task switching for dynamic load balancing. Here F is the number of full host buffers, B is the maximum number of full host buffers, M is the current number of rendering GPUs, and N is the total number of GPUs available for load balancing.

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 tests were conducted on a single computer with a 2.67 GHz Intel Core i7 processor and eight GTX 295 graphics cards.

5.2.1 Dynamic Load Balancing Without Runtime Changes

Figure 5.6 shows the changes in buffer fullness F and number of rendering GPUs over time for the N-body simulation without 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. Here, 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 5.7 shows the performance of six static configurations of GPUs compared to dynamic load balancing for 2000 particles. The x-axis represents the number of GPUs allocated for a static configuration, and it represents the average number of rendering GPUs allocated over time for rendering for the dynamic load balancing data point. 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 5.8 exhibits the ability of dynamic load

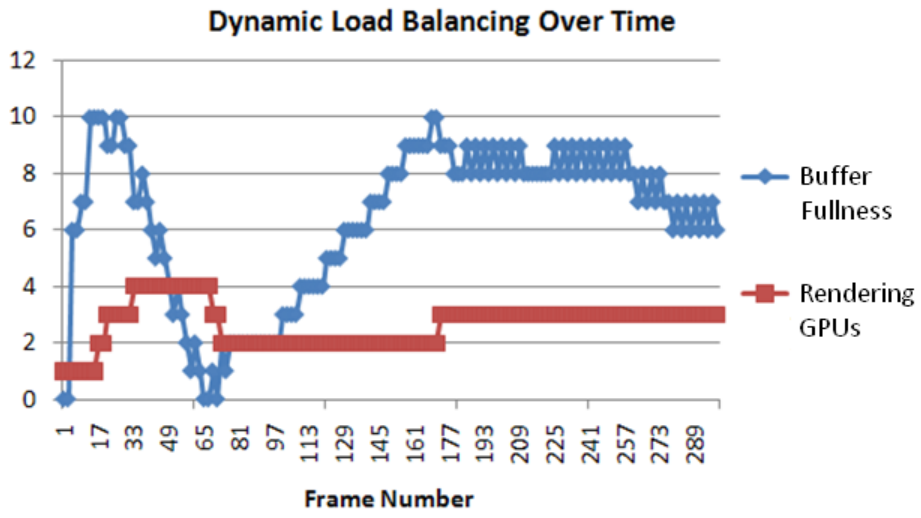

Figure 5.6: Change in buffer fullness over time for 3000 particles using dynamic load balancing without runtime changes. Here the maximum number of full host buffers is ten, while the maximum number of rendering GPUs is six.

balancing to find the optimal configuration for 4000 particles as well. A larger dataset creates 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.

Despite the ability of dynamic load balancing to achieve approximately equivalent speedup as optimal static load balancing, task switching introduces some overhead in synchronization, as shown in Figure 5.8. Switching a GPU from rendering to simulation requires some overhead since the task must wait at the simulation barrier until the next frame can be processed. However, switching from simulation to rendering does not introduce any significant overhead, since each rendering task processes a frame separately. As expected, the time to switch from rendering to simulation on average is about half the time of simulation since the GPU must wait for the current iteration of simulation to finish.

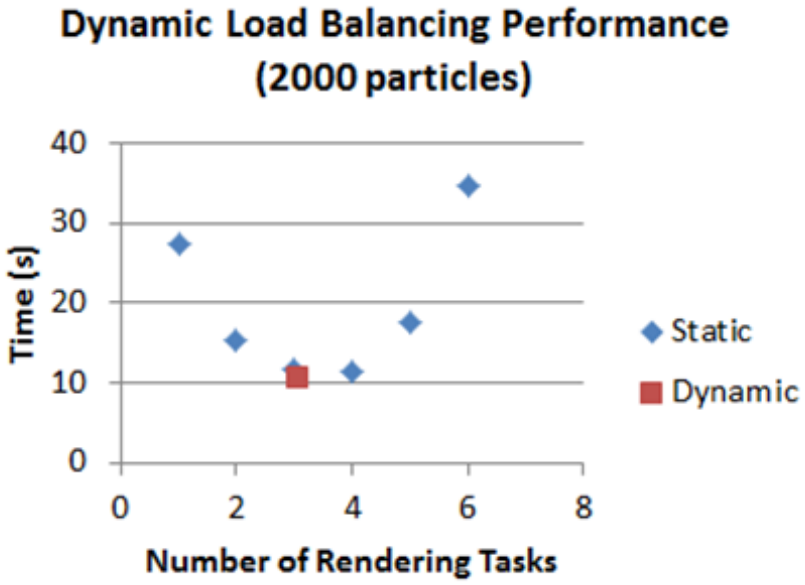

Figure 5.7: Execution time for static configurations and dynamic load balancing with 2000 particles without runtime changes

5.2.2 Workload Changes at Runtime

Not only can dynamic load balancing provide a speedup for a given set of input parameters, but it also can adjust to dynamic 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 this kind of unpredicted change.

In a test involving runtime changes, the same N-body simulation and visualization is used, but the level of supersampling changes at fixed intervals of 60 frames, with four, eight, and twelve samples. Figure 5.10 shows the changing buffer fullness and task switches for a 3000

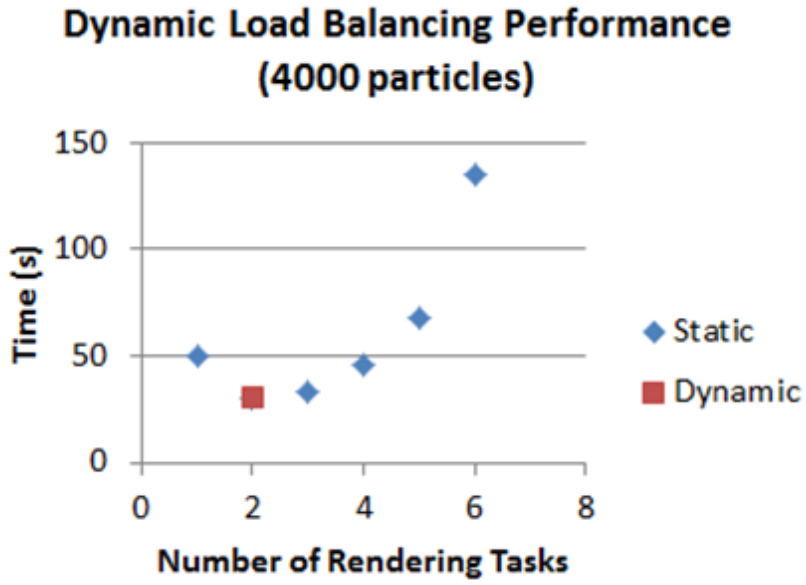

Figure 5.8: Execution time for static configurations and dynamic load balancing with 4000 particles without runtime changes

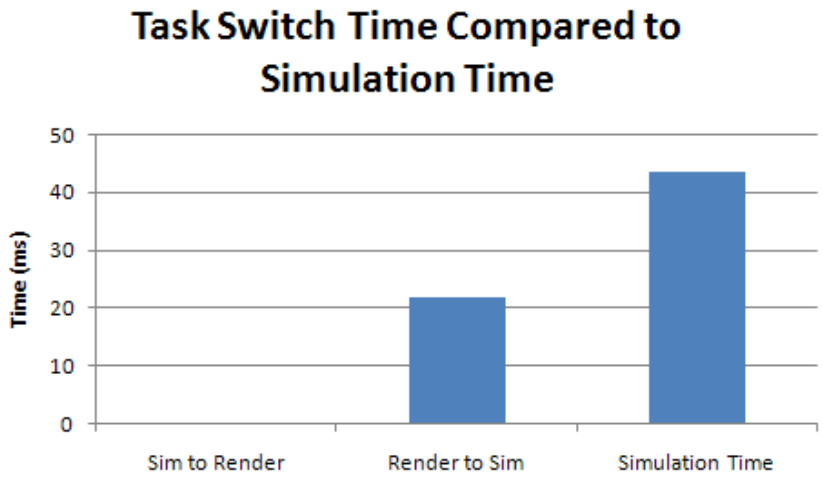

Figure 5.9: Time required for a GPU to switch tasks compared to total time to simulate a single frame for 3000 particles

particle dataset. This shows 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. This result shows how dynamic load balancing can adapt to changing workloads at runtime to reach the number of rendering GPUs M for an optimal load balance.

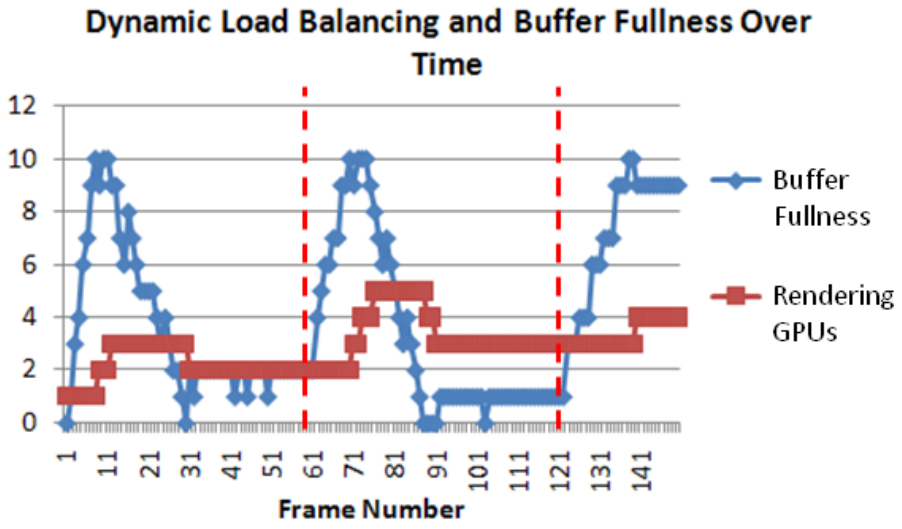

Figure 5.10: Change in buffer fullness over time for 3000 particles with increases in supersampling every sixty frames as shown by red dotted lines. Here the maximum number of full host buffers is ten, while the maximum number of rendering GPUs is six.

Figure 5.11 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 x-axis here represents the number of rendering GPUs for static configurations, and for the dynamic load balancing data point it represents average number of rendering GPUs. 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 5.12 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.

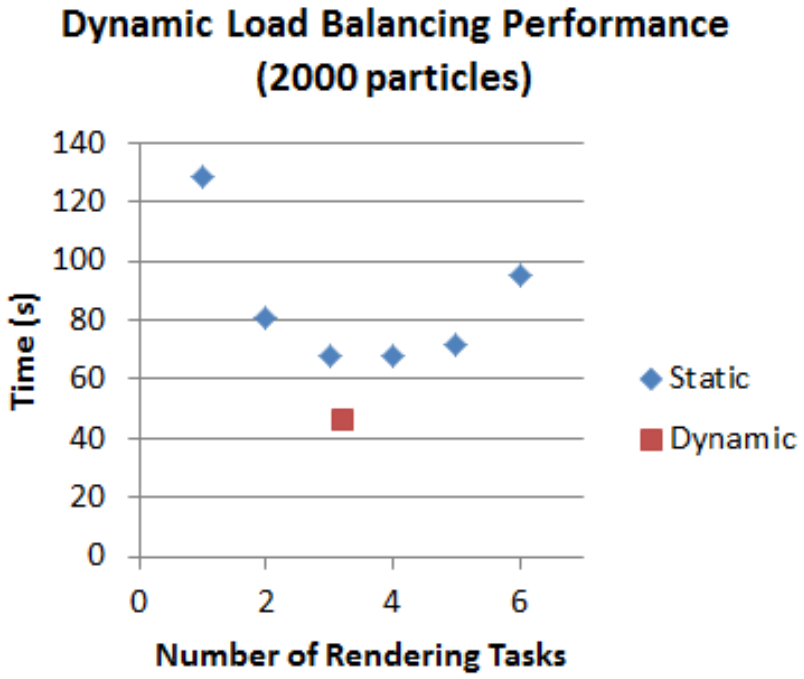

Figure 5.11: Execution time with change in supersampling and simulation iterations for static configurations and dynamic load balancing with 2000 particles and runtime changes.

Dynamic load balancing leads to consistent speedup over both average and optimal static

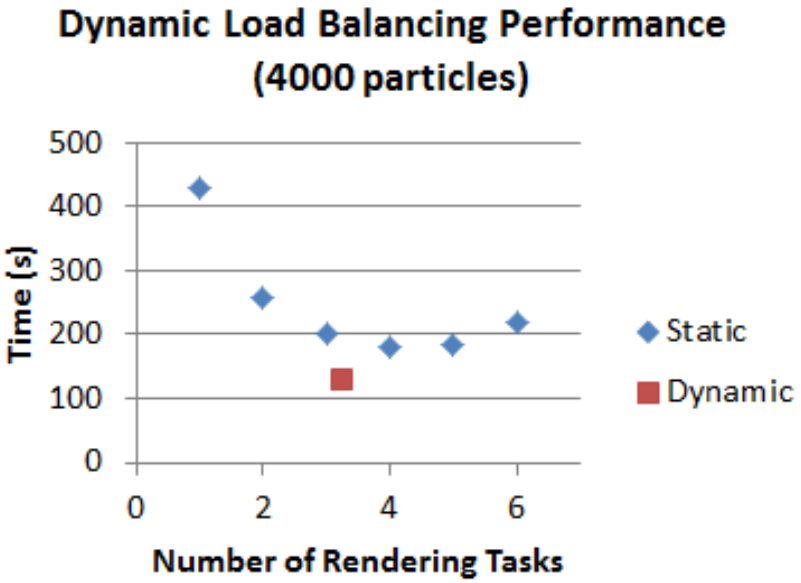

Figure 5.12: Execution time with change in supersampling and simulation iterations for static configurations and dynamic load balancing with 4000 particles and runtime changes.

workloads as shown in Figure 5.13. Static load balancing does not allow for adjusting to the changing workload as does dynamic load balancing. This results in a speedup of dynamic load balancing over the optimal static load balancing configuration. An even 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.

Our approach leads to a significant speedup for an application with changing input parameters at runtime, while for an unchanging application it still achieves equivalent performance to optimal static load balancing. Figure 5.14 shows the performance of dynamic versus optimal static load balancing for applications with and without runtime changes. It also demonstrates a significant speedup of dynamic load balancing over optimal static configurations for changes at runtime. As shown, there is no significant speedup for dynamic load balancing over the optimal static configuration without runtime changes, but there is

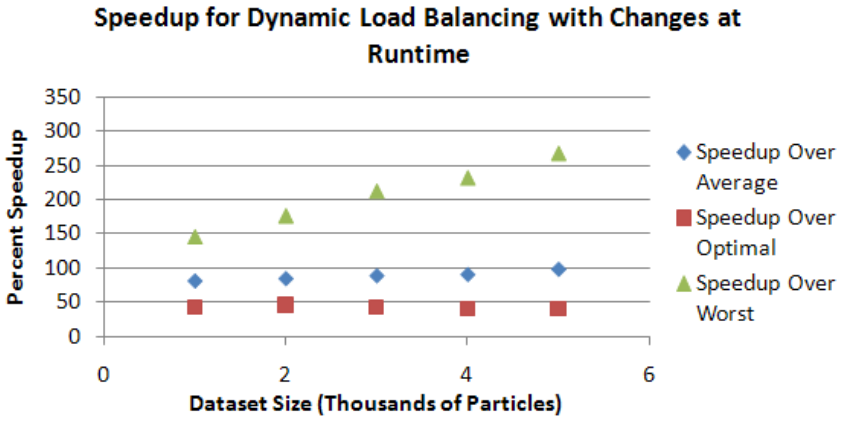

Figure 5.13: Speedup of dynamic load balancing over optimal, average, and worst case for different dataset sizes with changes at runtime

a speedup over the optimal static configuration with runtime changes. Both cases with and without runtime changes have a significant speedup for dynamic load balancing over the average static configuration.

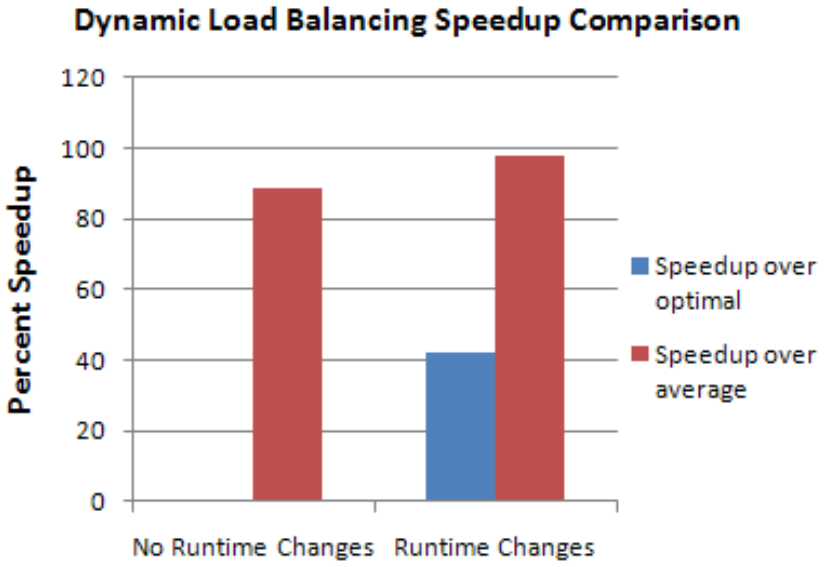

Figure 5.14: Average speedup of dynamic load balancing over optimal and average static configurations with and without runtime changes

Chapter 6

Framework

We present a framework that allows the load balancing methods to be incorporated with different applications.

6.1 Framework Components

Use of multiple GPUs requires significant programmer effort in memory management, synchronization, and load balancing. Software development for programs using multiple GPUs can therefore benefit from a framework to reduce the complexity of dynamic load balancing. This framework should provide techniques for load balancing for applications in order to address issues of workload distribution. Creation of a multi-GPU framework also presents challenges in accommodating varying needs of programs. Users should have the ability to easily integrate use of the framework with existing applications in order to best utilize load balancing.

To address these issues, we provide a framework for multi-GPU load balancing. The components of this framework allow for conversion of existing single GPU programs to multi-GPU ones. To take advantage of the framework, a user can specify tasks and input and out-

put data to incorporate use of multiple GPUs. Figure 6.1 shows the relationships of the framework components, and below is a description of the components and their roles.

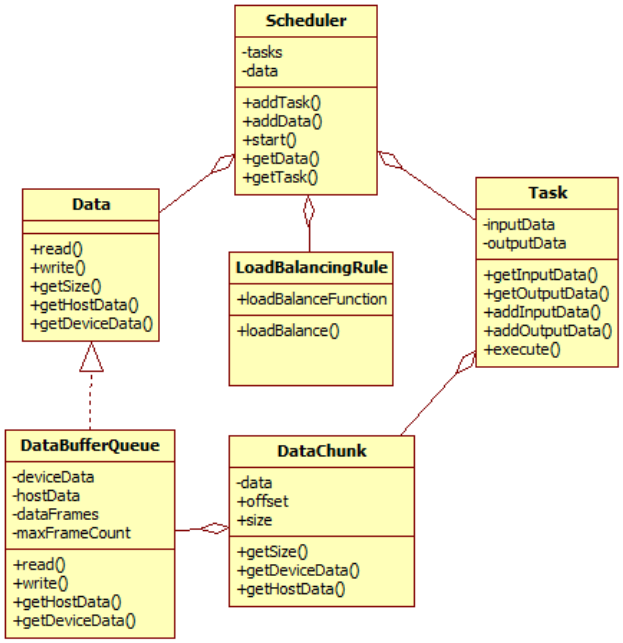

Figure 6.1: Framework Components

- Task: A Task represents a unit of work to be run on the GPU. It has input and output data dependencies that must be transferred to or from the GPU. The Task also provides scheduling properties, including load balancing and how data is partitioned for processing.

- Data: Data that must be processed by a Task. Input data must be transferred to a GPU from host memory, while output data must be transferred from the GPU. This allows the framework to transparently handle data transfers to reduce the effort of the software developer. Data also can have other properties, including data size and type.

- Scheduler: The Scheduler is responsible for managing tasks, data, and load balancing. Tasks and Data can be added through the scheduler, and it also manages execution of the program.

- LoadBalancingRule: A LoadBalancingRule determines how load balancing is carried out based on the current state of the application.
- DataBufferQueue: Implementation of the Data interface that allows for multiple frames of data to be streamed to and from the GPU. The DataFrames that are contained within a DataBufferQueue allow for multiple stored data that can be written by several tasks on different processors. The DataBufferQueue allows for load balancing between various types of tasks by allowing multiple frames of data to be read and written at once.
- DataChunk: Subset of a Data used by a Task. This allows for a Task to process a piece of an input or output data in order to partition a dataset for processing.

6.2 Load Balancing in the Framework

Load balancing is provided in the framework through the LoadBalancingRule that specifies scheduling properties for tasks. Users can specify load balancing by either creating an instance of a LoadBalancingRule or deriving a new rule that inherits from LoadBalancingRule. This flexibility allows the user to also implement further functionality if necessary. We provide an existing LoadBalancingRule that distributes workload based on buffer fullness, where a full buffer results in a task switch to avoid idle time.

6.3 Usage

As shown in Listing 6.1, the runtime API provided by the framework allows the user to easily add tasks, data, and scheduling to be managed. Tasks and Data can be created with desired scheduling properties for the program, and the Scheduler provides methods to add them. Then the program execution can be initiated through the framework. Host and

device data can then be accessed through the Data component to pass device parameters into kernel functions to run on the GPU. This significantly reduces the required work in creating a program that utilizes multiple GPUs in a multi-GPU computing environment. The components of the framework can also be extended for finer control over scheduling and load balancing in the framework.

Listing 6.1: Runtime API example

```
Scheduler scheduler;
scheduler.addData(new DataBufferQueue<float>(
    "myData", 1000));
scheduler.addTask(new Task(updateSim));
scheduler.addTask(new Task(updateRender));
...
scheduler.start();
```

To demonstrate the use of the framework, we have used the framework with an interactive ray traced physics application, as shown in Figure 6.2. In this application different spheres have physics-based motion and collisions with each other and the ground. While this computation is similar to the N-body simulation in that it involves interactions among all the objects, it shows how the framework can be used with different simulation and rendering in an application.

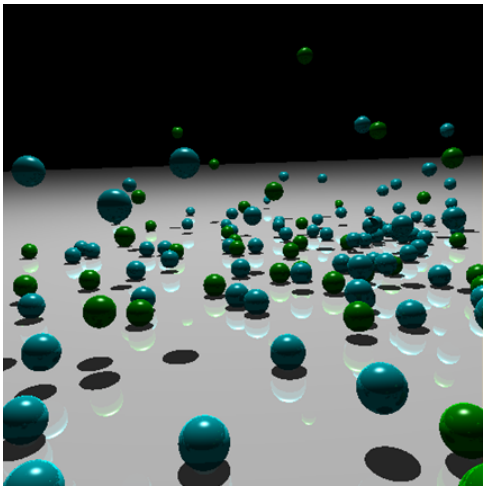

Figure 6.2: Interactive ray traced physics application with reflections and shadows

Chapter 7

Conclusions and Future Work

7.1 Conclusion

We propose data-driven and dynamic multi-GPU load balancing methods as well as a framework for incorporation with various applications. Multi-GPU processing offers a large performance gain over use of a single GPU, but the management of multiple GPUs introduces several challenges that we address. Scheduling multiple tasks can create load imbalance and resource underutilization. Managing memory transfers and synchronization among multiple GPUs adds complexity in programming. We have shown the viability of the load balancing methods by applying it to in-situ visualization for both data-driven and dynamic load balancing. Data-driven load balancing can provide a significant speedup by predicting optimal configurations when using previous performance data for an application. Dynamic load balancing can provide a significant speedup without previous performance data and can account for the limitation of a discrete number of GPUs for load balancing. Dynamic load balancing can achieve equivalent performance to data-driven load balancing, and it can also gain an additional performance improvement when an application has runtime changes in workloads. Our framework reduces the complexity in multi-GPU programming through scheduling, load balancing, and memory management, and it allows developers to use multiple GPUs with a

variety of applications.

7.2 Future Work

Several areas of future work could be explored, such as the inclusion of additional dynamic load balancing methods. Other possibilities of future work include improving framework performance and granularity of load balancing.

7.2.1 Performance Model

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. To use a performance model, samples of the data would first need to be taken to determine the scaling of the tasks among different numbers of GPUs. Use of the performance model would begin by using the simulation and rendering time to predict the optimal configuration. If simulation and rendering time of a single frame is not known, this can be sampled by first executing a single frame of simulation and rendering. In the current applications, the scaling of rendering and simulation is approximately linear when divided among multiple GPUs, which can be used as a performance model. The predicted optimal configuration based on this data is then computed by calculating:

$$M_1 = \underset{M=1,\ldots,N-1}{argmin}(abs(S/(N-M) - R/M)))$$

where M_1 is the predicted optimal number of rendering GPUs based on performance model, M is a tested configuration of GPUs, N is the total number of GPUs available for load balancing, S is the simulation time of a single frame for one GPU, and R is the rendering time of a single frame for one GPU. Thus, the goal is to minimize the difference in simulation and rendering to provide an optimal load balance. This is used for a linear model in these

applications, but this equation would need to be adjusted if a different scaling of simulation and rendering on multiple GPUs is needed.

However, load balancing based on a performance model does not take into account fullness of the buffer on host memory, which would prevent this method from continually switching to avoid idle time associated with a full or empty buffer. Thus, without modification, use of a performance model for task switching would be limited to the discrete optimal static configuration. In order to account for this issue, a combination of performance and buffer fullness could be used to continually switch to avoid idle time. Using a combination of the criteria would involve an initial optimal configuration from the performance model with an additional single switch L possible based on buffer fullness. The switch value L begins as zero, where the performance model is initially used to determine the number of rendering GPUs. If the buffer becomes full, then the switch L due to buffer fullness would be 1 to add an additional rendering GPU. If the buffer becomes empty, then the switch L due to buffer fullness would be -1 to subtract a rendering GPU. The goal of the combined metrics is thus to find the optimal configuration:

$$M_2 = M_1 + L$$

where M_2 is the optimal load balance for the combined criteria, M_1 is the predicted optimal configuration from the performance model, and L is -1, 0, or 1 based on switching due to buffer fullness. Thus, the minimum execution time is found through the performance criteria, while switching continues from this point due to buffer fullness.

A *buffer threshold* could also be added which would cause a switch before the buffer becomes full or empty. This could cause a switch a number of frames before the buffer becomes full or empty, and this number of frames could be determined by approximating the overhead of waiting for a task switch based on simulation cost of a single frame. Thus, this buffer threshold would trigger a task switch when the buffer is within the calculated threshold from empty or full. The criteria for switching from simulation and from rendering when using both a performance model and buffer fullness is shown in Algorithm 1 and Algorithm 2,

respectively. These criteria can be substituted as the criteria into Figure 5.5. The initial configuration is first computed by the performance model, while use of the buffer fullness and threshold adds continuous switching after this point.

Algorithm 1 Criteria for task switching from simulation to rendering based on a performance model and buffer fullness. Here M is the current number of rendering GPUs, M_1 is the optimal configuration predicted by the performance model, F is the fullness of the buffer, H is the buffer threshold for switching, B is the maximum number of buffers in host memory, and N is the total number of GPUs for load balancing.

if ($F > B$ - H and $M < N$-1 and $M \leq M_1$) $M \leftarrow M$+1

Algorithm 2 Criteria for task switching from rendering to simulation based on a performance model and buffer fullness. Here M is the current number of rendering GPUs, M_1 is the optimal configuration predicted by the performance model, F is the fullness of the buffer, and H is the buffer threshold for switching.

if ($F < H$ and $M > 1$ and $M \geq M_1$) $M \leftarrow M$-1

7.2.2 Performance Data Criteria

If a performance model is not known beforehand or would present challenges in calculating at runtime, then past performance data could be used as an alternative to a performance model. Incorporation of this metric could involve sampling new configurations one by one, and using performance of previously sampled configurations to choose new configurations to sample until the optimal configuration is found. The slope of the difference in performance between samples could be used to choose the next sample by choosing the next sample in order to continually come closer to the optimal configuration. The optimal configuration is determined when a local minimum in execution time is found, which occurs when performance begins decreasing in the performance samples. However, this method assumes that there is one local minimum in performance for the different workload configurations. As an alternative, sampling of performance data could begin with a sparser sampling of the dataset

in a binary search as a first phase, followed by a linear search as a second phase to find the exact optimal configuration. Using these two phases may also be able to more effectively find a global minimum when multiple local minima exist. Compared with switching based on buffer fullness, this method has certain tradeoffs. It allows for switching tasks based on performance, which could allow for more flexibility and more elegant handling of switching with multiple tasks. When using multiple tasks, different configurations would simply be sampled for performance data, and the minimum execution time would simply be chosen as the optimal. Similar to using a performance model, dynamic load balancing based on performance data may also more quickly find the optimal configuration since the method would not have to wait for the buffer to become full or empty to begin switching, and initial oscillations in load balancing could be avoided.

Similar to use of the performance model, performance data is limited by a discrete number of GPUs and could be combined with buffer fullness to address this issue. This could involve first sampling different configurations for performance data periodically until a minimum is found, and then switching based on buffer fullness after this optimal configuration has been found. This would achieve the advantages of both methods in that switching based on performance could be used for more flexibility and potentially faster determination of the optimal configuration, while switching based on buffer fullness could be used to address the limitation of a static configuration and avoid idle time due to a full or empty buffer.

7.2.3 Rate of Change in Buffer Fullness Criteria

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. The determination of the threshold for switching based on rate of change in buffer fullness is left for future work. 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. The method for switching based on buffer fullness could be the same as demonstrated for the combined performance model and buffer fullness method. The approach combining buffer fullness and rate of change in buffer fullness would seek to find the optimal load balance through the following method:

$$M_4 = \underset{M_3=1,...,N-1}{argmin}(abs(b)) + L$$

where M_3 is the load balance determined by rate of change in buffer fullness, M_4 is the optimal load balance for the combined criteria, b is the rate of change in buffer fullness, N is the total number of GPUs available for load balancing, and L is -1, 0, or 1 based on switching due to buffer fullness. Thus, the minimum absolute rate of change in buffer fullness is first found through the first criteria, and then switching can continue due to buffer fullness to avoid idle time due to an empty or full buffer.

7.2.4 Improving Granularity and Performance of Load Balancing

The performance of the load balancing framework and granularity of load balancing could also be improved. Use of Fermi by NVIDIA would allow for finer granularity in load balancing by scheduling on the multi-processor level. Each Fermi graphics card can have many concurrent kernels running at once, which would allow for both simulation and rendering tasks to be partitioned on a single card to take into account differing workloads. This could help address the limitation of having a discrete number of GPUs for the optimal static configuration using multi-GPU load balancing.

Other areas of future work could further improve the framework and performance. Taking advantage of the GPU Direct in CUDA would allow for faster memory transfers to improve performance by transferring memory directly between GPUs. In addition, asynchronous memory transfers would allow for overlap of transfers with computation. These various changes would help to improve the performance of the framework without significantly changing the API of the framework.

Bibliography

[1] Y. Lee B. Catanzaro P. Ivanov A. Kloeckner, N. Pinto and A. Fasih. PyCUDA: GPU Run-Time Code Generation for High-Performance Computing. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

[2] 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.

[3] 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.

[4] Ramu Anandakrishnan and Alexey V. Onufriev. An n log n approximation based on the natural organization of biomolecules for speeding up the computation of long range interactions. *Journal of Computational Chemistry*, 31(4):691–706, 2010.

[5] 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.

[6] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 105–114, New York, NY, USA, 2010. ACM.

[7] Michela Becchi, Surendra Byna, Srihari Cadambi, and Srimat Chakradhar. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 82–91, New York, NY, USA, 2010. ACM.

[8] Aaron Becker, Isaac Dooley, and Laxmikant Kale. Flexible hardware mapping for finite element simulations on hybrid cpu / gpu clusters. In *SAAHPC : Symposium on Application Accelerators in HPC*, July 2009.

[9] 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.

[10] James C. Brodman, Basilio B. Fraguela, María J. Garzarán, and David Padua. New abstractions for data parallel programming. In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[11] 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.

[12] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-gpu systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, 04 2010.

[13] Wagner T. Correa, James T. Klosowski, Chrisopher J. Morris, and Thomas M. Jackmann. Spvn: a new application framework for interactive visualization of large datasets. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 6, New York, NY, USA, 2007. ACM.

[14] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM.

[15] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, New York, NY, USA, 2010. ACM.

[16] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[17] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364, New York, NY, USA, 2010. ACM.

[18] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Zhe Fan, Feng Qiu, and Arie E. Kaufman. Zippy: A framework for computation and visualization on a gpu cluster. *Comput. Graph. Forum*, 27(2):341–350, 2008.

[20] 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.

[21] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[22] 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.

[23] S. Frank and A. Kaufman. Out-of-core and dynamic programming for data distribution on a volume visualization cluster. *Comput. Graph. Forum*, 28(1):141–153, 2009.

[24] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wenmei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 347–358, New York, NY, USA, 2010. ACM.

[25] Enrico Gobbetti and Fabio Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 878–885, New York, NY, USA, 2005. ACM.

[26] Francesc Guim, Ivan Rodero, Julita Corbalan, and Manish Parashar. Enabling gpu and many-core systems in heterogeneous hpc environments using memory considerations. pages 146 –155, sep. 2010.

[27] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 205–216, New York, NY, USA, 2010. ACM.

[28] R. Hagan and Y. Cao. Multi-gpu load balancing for in-situ visualization. In *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011.

[29] Tianyi David Han and Tarek S. Abdelrahman. hicuda: a high-level directive-based language for gpu programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, New York, NY, USA, 2009. ACM.

[30] Alan Heirich and James Arvo. A competitive analysis of load balancing strategies for parallel ray tracing. *J. Supercomput.*, 12(1-2):57–68, 1998.

[31] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: writing parallel program portable between cpu and gpu. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 217–226, New York, NY, USA, 2010. ACM.

[32] Qiming Hou, Kun Zhou, and Baining Guo. Bsgp: bulk-synchronous gpu programming. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–12, New York, NY, USA, 2008. ACM.

[33] William Humphrey, Andrew Dalke, and Klaus Schulten. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14:33–38, 1996.

[34] 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.

[35] 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.

[36] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling gpu-cpu workloads and systems. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42, New York, NY, USA, 2010. ACM.

[37] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-Eui Yoon. Hpccd: Hybrid parallel continuous collision detection using cpus and gpus. *Comput. Graph. Forum*, 28(7):1791–1800, 2009.

[38] J. Prin L. Nyland, M. Harris. Fast n-body simulation with cuda. *GPU Gems 3*, pages 677–695, 2007.

[39] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, 2009.

[40] Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.

[41] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43:287–296, March 2008.

[42] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for gpu program optimizations. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[43] 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.

[44] David A. Pearlman, David A. Case, James W. Caldwell, Wilson S. Ross, Thomas E. Cheatham, Steve DeBolt, David Ferguson, George Seibel, and Peter Kollman. Amber, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. *Computer Physics Communications*, 91(1-3):1 – 41, 1995.

[45] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 121–130, New York, NY, USA, 2009. ACM.

[46] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[47] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, and Wen-mei W. Hwu. Program optimization carving for gpu computing. *J. Parallel Distrib. Comput.*, 68(10):1389–1401, 2008.

[48] John E. Stone, Jan Saam, David J. Hardy, Kirby L. Vandivort, Wen-mei W. Hwu, and Klaus Schulten. High performance computation and interactive display of molecular orbitals on gpus and multi-core cpus. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 9–18, New York, NY, USA, 2009. ACM.

[49] John Stratton, Sam Stone, and Wen-Mei Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. pages 16–30. 2008.

[50] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 111–119, New York, NY, USA, 2010. ACM.

[51] Magnus Strengert, Christoph Muller, Carsten Dachsbacher, and Thomas Ertl. Cudasa: Compute unified device architecture and systems. In Kwan-Liu Ma Daniel Weiskopf, Jean M. Favre, editor, *Eurographics 2008 Symposium on Parallel Graphics and Visualization (EGPGV'08)*, pages 49–56. Eurographics Association, 2008.

[52] Jeff A. Stuart and John D. Owens. Message passing on data-parallel architectures. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[53] Sain-Zee Ueng, Melvin Lathara, Sara Baghsorkhi, and Wen-mei Hwu. Cuda-lite: Reducing gpu programming complexity. In Jose Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 2008.

[54] Y. Guo Y. Yan, J. Zhao and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Yonghong Yan, Jisheng Zhao, Yi Guo, Vivek Sarkar. Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2009.

[55] Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag.

[56] Ping Yao, Hong An, Mu Xu, Gu Liu, Xiaoqiang Li, Yaobin Wang, and Wenting Han. Cuhmmer: A load-balanced cpu-gpu cooperative bioinformatics application. pages 24 –30, jun. 2010.