Load Balanced Parallel GPU Out-of-Core for Continuous LOD Model Visualization

Chao Peng
Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24060
Email: chaopeng@vt.edu

Peng Mi
Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24060
Email: mipeng@vt.edu

Yong Cao
Department of Computer Science
Virginia Tech
Blacksburg, Virginia 24060
Email: yongcao@vt.edu

***Abstract*—Rendering massive 3D models has been recognized as a challenging task. Due to the limited size of GPU memory, a massive model containing hundreds of millions of primitives cannot fit into most of modern GPUs. By applying parallel level-of-detail (LOD), as proposed in [1], only a portion of primitives instead of the whole are necessary to be streamed to the GPU. However, the low bandwidth in CPU-GPU communication is still the major bottleneck that prevents users from achieving high-performance rendering of massive 3D models on a single-GPU system. This paper explores a device-level parallel design that distributes the workloads for both GPU out-of-core and LOD processing in a multi-GPU multi-display system. Our multi-GPU out-of-core takes advantages of a load-balancing method and seamlessly integrates with the parallel LOD algorithm. By using frame-to-frame coherence, the overhead of data transferring is significantly reduced on each GPU. Our experiments show a highly interactive visualization of the "Boeing 777" airplane model that consists of over 332 million triangles and over 223 million vertices.**

I. Introduction

Recently, GPU hardwares have been praised not only because of their dramatically increased computational power but also due to their programability for general-purpose computation. GPU architectures allow a large number of threads launched simultaneously to support high-performance applications. However, because of the enormous amount of renderable primitives in a massive model, the power and memory of a single GPU may not be sufficient to visualize them at a decent rendering rate. In the research domains, such as Computer-Aided Design (CAD) products, mechanical visualization and virtual reality applications, researchers develop very complex 3D models that may consist of millions, or even hundreds of millions of polygon primitives and consume several gigabytes on storage. Since these gigabyte-sized models cannot fit into most of commodity GPUs, the polygon primitives have to be transferred from a CPU host before rendering each frame. Although parallel LOD algorithms [1], [2], [3] have been proposed to reduce the amount of GPU resident primitives, the bandwidth in CPU-GPU communication is still too low to transfer the data efficiently, which usually is the major performance bottleneck in massive model rendering.

To address this issue, multi-GPU systems have caught attention of researchers since it can provide more computational power and memory. With the rapid hardware development, a motherboard is commonly configured with multiple PCIe slots that enable installations of two or more graphics cards. This configuration would have high potentials to increase the performance by distributing workloads across GPUs. Meanwhile, with additional GPU display ports, multiple display monitors can be connected, so that the rich information embedded in massive data can be visualized at much higher resolutions as what they should deserve. However, it is not trivial to transplant a parallel approach from a single-GPU to a multi-GPU system. One major reason is the lacks of both programming models and well-established inter-GPU communication for a multi-GPU system. Although major GPU suppliers, such as NVIDIA and AMD, support multi-GPUs by establishing Scalable Link Interface (SLI) and Crossfire, respectively, their technologies are primarily designed for gaming and short of the functionalities for both general programming and software implementations. Also, when enabling SLI or Crossfire, the GPUs behave as one hardware entity and per GPU execution is not allowed. Another reason is the workload-balancing issue between GPUs. Imbalanced workload distribution will hurt the performance.

Main contributions. In this paper, we present a device-level parallel approach for real-time massive model rendering in a multi-GPU multi-display system. Our contributions include the following two features:

1) **Parallel GPU out-of-core.** We employ a device-level parallelism for efficient data fetching from CPU main memory to multiple GPU devices. Our parallel out-of-core is seamlessly integrated with LOD processing and frame-to-frame coherence schemes.
2) **Balanced data distribution for parallel rendering.** We propose a load balancing algorithm to dynamically and evenly distribute workloads to each GPU, so that both performance and memory usage achieve an optimal standard.

System configuration. While the performance of multicore CPUs and GPUs are scaling along Moore's law, it has been becoming feasible to build clusters of heterogeneous multi-GPU architectures for high-performance graphics. However, before carrying on an efficient algorithm on a cluster system, it is essential to make a single node perform optimally. Upon this

demand, we use a single node machine, which is a standard PC platform configured with two GPU devices. Each GPU is connected with a dedicated PCI-Express and able to execute their local instructions and perform rendering tasks.

Input 3D model. CAD datasets are one of major digital representations for many of man-made designs. In our approach, we target on the visualization of "Boeing 777" CAD model, which organizes rich geometric information in a huge number of loosely connected objects. This airplane model contains over 332 million triangles and over 223 million vertices that consume more than 6 gigabyte memory.

Organization. Section II briefly reviews some related works. In Section III, we describe the current state-of-art technologies in massive model rendering. In Section IV, we state the problems in high-performance rendering and give an overview of our approach. In Section V, we discuss the methods of CUDA-OpenGL interoperability for a multi-GPU architecture. In Section VI, we present our load balancing algorithm for GPU data distribution. Section VII discusses the methods of GPU-GPU communication and synchronization. We evaluate our approach in Section VIII. Finally, we conclude our work in Section IX.

II. Related Work

In the past, researchers have dedicated their efforts in massive model rendering by using many different acceleration data structures. We review some of them in Section II-A. Since multi-GPU systems have become a new trend for High-Performance Computing (HPC). We provide some of the previous works that concentrated on device/cluster-level parallel designs in Section II-B.

A. Massive Model Visualization

Interactively rendering of massive 3D models has been an active research domain. To address performance issues, mesh simplification is commonly used to reduce the workload of rendering. The basic idea of mesh simplification is to simplify a complex model until it becomes manageable by renderers. Some previous approaches, such as Progressive Meshes ([4], [5]), Quadric Error Metrics ([6], [7]) and adaptive LOD [8], where meshes are simplified based on a sequence of edge-collapsing modifications. In order to handle large-scale models, out-of-core methods have been proposed. Cignoni et.al [9] presented a geometry-based multi-resolution structure, known as the tetrahedra hierarchy, to pre-compute static LODs. Yoon et.al [10] presented a clustered vertex hierarchy (CHPM) as their out-of-core structure for view-dependent simplification and rendering.

More recently, GPUs have been used to simplify complex models. Hu et.al [3] implemented a view-dependent LOD algorithm on GPU without fully considering the vertex dependencies. Derzapf et.al [11] used a compact data structure for Progressive Meshes that requires less GPU memory, so that the run-time parallel processing can be optimized, later, their work was extended for parallel out-of-core LOD in [12]. In order to render gigabyte-scale 3D models in parallel, Peng et.al [1] presented an parallel approach that successfully removed the data dependencies for efficient run-time processing. By utilizing temporal coherence between frames, the large amount of data can be streamed and defragmented efficiently.

B. Multi-GPU Approaches

As the rapid development of hardwares, many computing systems are built with GPU clusters for high-performance computations. Eilemann [13] summarized and analyzed existing approaches targeting on parallel rendering designs with multiple or clustered GPUs. One popular software package, *Equalizer*, has been commonly used in multi-GPU rendering society. As introduced in paper [14], Equalizer is a scalable parallel rendering framework suitable for large data visualization and OpenGL-based applications. It utilizes a flexible compound tree structure to support its rendering and image compositing strategy; however, there are some issues remaining unsolved and affecting the parallel performance, for example, the load balancing issue between GPUs or clusters for data distribution and replication.

Because of the importance of load-balancing issues, Fogal et.al [15] discussed it by considering data-transferring overhead and the balance among the rendering, the compositing and the viability of GPU cluster in a distributed memory environment. Erol et.al [16] concentrated on the cross-segment load balancing within Equalizer framework. Their work proposed a dynamic task partition strategy for the best usage of the available shared graphics resources. Another approach, presented in [17], was for dynamic load balancing. By utilizing frame-to-frame coherence, this approach redistributed data based on the historical frame rates; If one GPU has a higher frame rate in pervious frames, it would be allocated larger workloads; otherwise, its workloads would be reduced accordingly. To achieve higher resolutions in a visualization, the tiled display systems have been widely used in many visualization researches. As discussed in [18], the whole picture is projected onto multiple display nodes with proportional viewports. Besides the descriptions of system setting, the authors also provided their solutions for the multi-display synchronization.

III. Description of the Current State-of-Art

Among standard real-time visualizations, mesh simplification is one of acceleration techniques that reduces the complexity of 3D models for fast rendering. Traditional algorithms represent data in hierarchical structures, such as multi-resolution of the static LODs [9] and the clustered vertex hierarchy [10]. To build them, a bottom-up node-merging process is used, and inter-dependency is introduced between levels of the hierarchies.

Peng and Cao [1] proposed a dependency-free approach that makes simplification of massive models suitable on GPU. Our parallel LOD implementation is extended from Peng and Cao's work. We give more details in this section. In that work, edge-collapsing information is encoded in an array structure that is generated by collapsing edges iteratively. At each iteration,

two vertices of an edge are merged, and the corresponding triangles are eliminated. To assure a faithful look of low-poly object, the edge can be chosen based on the rule that, when collapsed, visual changes are minimal (e.g., the rule introduced in [19]). Each element in the array corresponds to a vertex, and its value is the index of the target vertex that it merges to.

According to the order of edge-collapsing operations, Storage of vertices and triangles are re-arranged. Basically, the first removed vertex during iterative edge-collapsing is re-stored to the last position in the set of vertex data; and the last removed vertex is re-stored to the first position. The same re-arranging strategy is applied to the triangle data as well. As a result, the order of storing re-arranged data reflects the levels of details of the model. Consequently, if needing a coarse version of the model, a small number of continuous vertices and triangles are sufficient and selected by starting from the first element in their sets.

At run-time, based on LOD selection criteria, such as those used in [20], [21], [22], [23], only a portion of data is active to generate the simplified version of original model as the alternative for rendering. By using GPU parallel architectures, each selected triangle is assigned to a GPU thread and is reformed to a new shape, where all three vertex indices of the triangle is replaced with an appropriate target vertex by walking backward through the array of edge-collapsing information.

The increase of GPU memory does not catch up the capability on CPU main memory. Most of todays GPUs cannot hold the data requiring several gigabytes on storage. Thus, at each rendering frame, the selected portion of data have to be streamed to GPU to perform parallel computation and rasterization.

IV. Problem Statements and Overview

In this section, we describe our research problems by identifying performance bottlenecks and load balancing issues. We also give the overview of our parallel design.

A. Performance Bottlenecks

CPU-GPU data streaming is unavoidable in large-scale data visualization. Although the size of renderable data can be reduced using simplification algorithms, to preserve a decent level of visual fidelity, the simplified data is usually still too large to be streamed efficiently. Thus, the size of to-be-streamed data becomes the major issue that prevents the achievement of high interactive rates.

Brute-force streaming those selected data to GPU is very time-consuming. In many situations, a common effort to reduce the time spent on data streaming is utilizing frame-to-frame coherence, so that only frame-different data is identified and needs to be streamed. By combining this streamed data with the GPU existing data from the previous frame, we can assemble the new data coinciding with the currently rendering frame. As we know, standard mechanisms of graphics programming, such as OpenGL's Vertex Buffer Objects (VBO), take the graphics driver's hints about primitive usage patterns to increase the rasterization performance of graphics API. To enable VBO, the data have to be organized in the same order as they are originally stored. However, in general cases, the frame-different data will be stored in a memory block that is separate from those already existing data. For combining them with respect to the original data appearance in storage, the data in both blocks have to be shuffled around to the correct positions in the memory reserved by the current frame. This process is generally know as "*Defragmentation*". Unfortunately, *Defragmentation* on GPU is slow, because it involves many operations of non-coalesced global memory accesses. *Defragmentation* time scales with the size of the data residing on GPU, so that it will be a significant factor that influences the overall rendering performance, especially when the data size is massively large. Using multi-GPU systems, the data is able to be distributed. Each GPU will perform a less amount of data, which can reduce the *Defragmentation* overhead.

B. Load Balancing Issues between GPUs

Using multiple GPUs is a trend to increase computational power and memory capabilities. However, balancing workloads and resource utilizations between GPUs has not been satisfactorily addressed. Load balancing problems in massive model visualization are centered around how to distribute the renderable data to GPUs. Imbalanced distribution would cause underutilization of available GPU resources and waste memory spaces, so that both performance and visual quality may be decreased.

Commercial solutions, such as Nvidia SLI technology, have been introduced to balance geometry workloads between two GPUs. SLI is a bridge that spans two GPUs to send data directly within a master-slave configuration. For example, the master GPU send half of rendering work to the slave GPU. Then the slave GPU send its output image back to the master for compositing the images. However, SLI is not suitable for the problems that we want to solve in this paper. SLI does not incorporate with out-of-core and Nvidia CUDA development. It requires all data to fit in GPU memory. Also, its master-slave configuration is only for single-display applications, not suitable for multi-display applications, because the latter one connects each GPU to a display monitor, which breaks the master-slave concept. Therefore, it is essential to have a better load balancer for GPU out-of-core and CUDA-programmed parallel LOD.

C. Overview of Our Approach

In today's PC systems, multiple GPUs start to become the standard configuration for all levels of users. The goal of our work is to design a parallel system that provides the software supports for multi-GPU rendering. Each GPU will be driven by one CPU-core. In our implementation, each CPU-core executes an instance of the program and feeds the data from CPU main memory to the GPU that it associates with. The input 3D model and other necessary

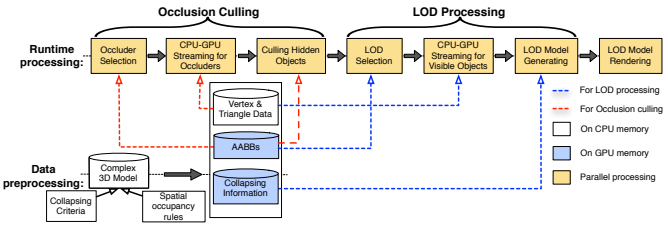

Fig. 1. The overview of our approach.

run-time control parameters are shared among processes by employing a method of Inter-Process Communication (IPC). We illustrate the overview of our approach in Figure 1. Each GPU is connected to a display monitor to show the rendered geometries. If the data is balanced between GPUs, our load-balancing algorithm will automatically calculate the optimal solutions. The direct inter-communication between GPUs is established in order to perform framebuffer exchanging as necessary for the final display.

V. Parallelization of CUDA-OpenGL Interoperability on Multiple GPUs

When asking a GPU to perform both general-purpose and graphics computations, the interoperation between Nvidia CUDA and OpenGL is desired. Like in our application, CUDA is suitable for running the calculations of data de-fragmentation and parallel triangle-level simplifications, while OpenGL is better to take the task of rendering the simplified models. Thanks to CUDA-OpenGL interoperability provided by CUDA SDK, but for multi-GPU implementations, inter-operation among multiple CUDA and OpenGL contexts will be an issue. In general, the possible solutions include: (1) A single CPU thread; (2) Multiple CPU threads (one GPU controlled by one CPU thread); (3) Multiple processes (one GPU controlled by one process). We will discuss about the details in the following paragraphs.

A. A Single CPU Thread

Since the release of CUDA v4.0, multi-GPU programming can be performed in a single CPU thread. By calling $cudaSetDevice()$, CUDA kernel executions and contexts are switched between GPUs. But switching OpenGL contexts between GPUs in a singe CPU thread is not allowed. For example, $cudaSetGLSetDevice()$ can be called only once at the start of the program. This problem will make the single CPU thread implementation not compatible.

B. Multiple CPU Threads

Creating multiple CPU threads allows one CPU thread to bind one GPU, so that each GPU has not only its own contexts but also its own host. A thread is able to maintain its local storage and control the device that is is assigned with. The OpenGL context associated to a CPU thread is able to interoperate with the CUDA context of the same thread. However, the problem is that OpenGL is not a thread-safe graphics API since it is actually asynchronous by nature. The GL calls within different threads cannot be executed in-order as they are issued. When the driver schedules these calls, OpenGL contexts would switch frequently, which is particularly time-consuming and would significantly decrease the overall performance.

C. Multiple Processes

To eliminate the overheads of switching contexts, a multi-process strategy will be a solution for interoperating CUDA and OpenGL on multiple GPUs. One process communicates to one GPU, and maintains its own memory spaces and its private run-time resources. The GL calls within a process are executed in-order without the affections of the other processes. Thus, in our implementation, we use this multi-process strategy plus the employments of Inter-Process Communication (IPC) for synchronization of shared properties (such as camera viewpoints).

VI. Data Distribution at Run-Time

In our system, two GPUs are installed in a single computer node. Each GPU drives a display monitor and visualizes half size of the frame, which will contain the data appearing its window. Unfortunately, this simple strategy usually results in poor performance if graphical primitives are not uniformly distributed over all GPUs.

A worse issue caused by imbalanced distribution is in memory usage. The “Boeing 777 airplane” used in our system has over 700 thousands of individual objects and contains more than 6 gigabytes of vertices and triangles. Since most of GPUs have much less memory than that size, the primitive count allocated for each object is well budgeted by ensuring the sum of all primitive counts is constrained within the given maximal amount. Of course, more GPUs mean more available memory, and consequently indicate a higher potential to increase the levels of details by adding more primitives to objects. However, imbalanced distribution may overburden a GPU’s memory capability. For example, in an extreme case, one GPU may obtain the whole of selected data that exceeds its maximal memory size, while the other one is idling without any.

Our load balancer uses a dynamic partitioning procedure that recursively splits space of the view frustum. The balancer will harmonize its execution time with the partition quality within an efficient parallel implementation on GPU. In the following paragraphs, we first introduce the fundamental method of view frustum partitioning, then we propose our dynamic load balancing algorithm.

A. The fundamental of View Frustum Partitioning for Data Distribution

From a viewpoint, only the objects inside the view frustum (represented with six planes defined by the camera) are visible to the renderer. We pre-calculate a tight Axis-Aligned Bounding Box(AABB) for each object. AABBs are used to determine the visibilities of objects by testing them against the view frustum. If an object is outside, it will be assigned

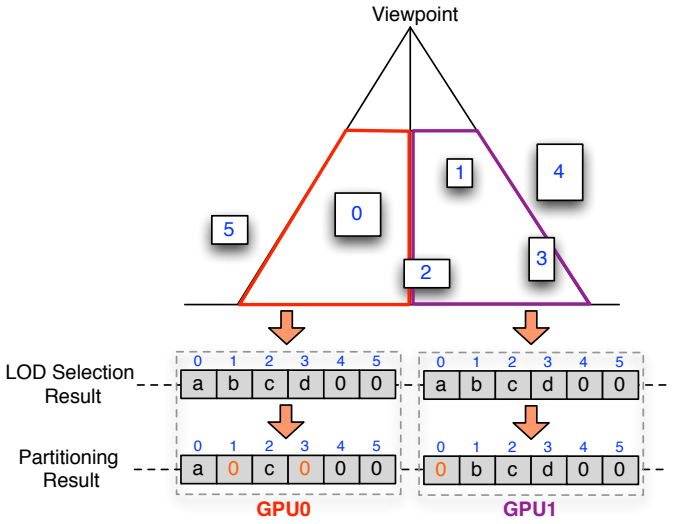

Fig. 2. **The view frustum partitioning.** The indices 0-5 represent the object's bounding boxes. a-d stand for the desired count resulted in LOD selection.

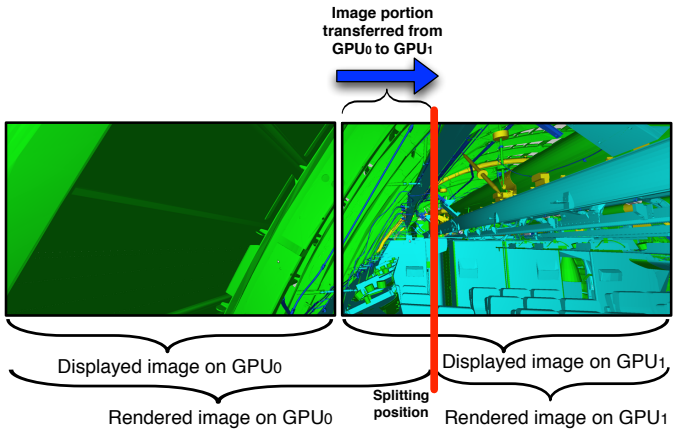

Fig. 3. **The dynamic load-balancing algorithm.** The whole screen is split by balancing the number of primitives distributed between GPUs. In this example, GPU_0 transfers a potion of the image to GPU_1.

with the lowest level of details (e.g., zero vertex and triangle), otherwise, it will be allocated a cut from the budget of the overall primitive count through the process of LOD selection.

After that, we distribute the primitive data to the GPUs. As shown in Figure 2, the view-frustum is divided into sub frustums, each of which is associated to a GPU. For each visible object, we identify the GPU it belong to by testing its AABB against the sub frustums. If an object is not in the sub frustum of a GPU, the detail level of the object will be set to zero for this GPU. At the stage of rasterization, the calls of perspective and projection transformations take the full size of the view frustum, but for displaying the contents on the screen, the viewport only needs to be set with the half of the framebuffer that contains rendered objects.

B. Parallel Dynamic Load-Balancing Algorithm

Obviously, the fundamental approach described in Section VI-A will lead to load-balancing problems since it always distributes the data by partitioning the frustum statically and evenly. The static partitioning method distributes the workloads usually according to the dimension proportions of the windows that are launched by GPUs, rather than by balancing the computational cost of data processing. To achieve the optimal rendering performance, we present a parallel dynamic load-balancing algorithm. Given a specific viewpoint, the screen is dynamically split by balancing the number of polygon primitives that are operated in each GPU. The rendered images will be exchanged between GPUs to adjust the image projection within inter-process communications. For example, as illustrated in Figure 3, the number of triangles are balanced between GPU_0 and GPU_1. GPU_0 renders a larger part of the screen, so that it transfers an image portion to GPU_1 to ensure the viewport correctness.

Our method is illustrated in Algorithm 1. The goal of the algorithm is to calculate where the view frustum should be split, so that the amount of data can be balanced. In the algorithm, the returned value, $split$, ranges between (0,1). It tells the position of splitting the view frustum on the near plane. The algorithm is executed in a per-process manner. The vf represents the view frustum associated to the camera, and the id is the process index. Here, we use either the vertex count or the triangle count to represent the complexity (the level of details) of each object. In the algorithm, the list of object complexities is represented as $compLevel$, where the ith object's complexity is denoted as $compLevel[i]$.

In the initialization, we set the $split$ to be 0.5 indicating that the view frustum is divided evenly. Then, we iteratively find the optimal $split$ value. At each iteration, the sub frustum of each process is updated. We denoted the left sub frustum as $subF_l$. The objects' AABBs are tested against $subF_l$ to re-generate the $compLevel$ for the GPU. In the Algorithm 1, $compLevel_l$ represents the list of objects' complexities for the GPU rendering the left part of the screen; and $compLevel_r$ is for the GPU rendering the right part (refer to Line 8-16 in Algorithm 1). For efficiency purposes, we can employ an implementation using CUDA to compute $compLevel_l$ and $compLevel_r$ in parallel. In Line 19, the $ratio$ is defined to find out if the value of the $split$ has reached the satisfaction by comparing to the $threshold$ value. Ideally, setting $threshold$ to 0.5 will cut the data evenly for distribution. However, in most of cases, this may need too many iterations, which would potentially slow down the performance. an appropriate $threshold$ that weights between the execution time of the load balancer and the distribution proportion will be better learned in practice. After that, each GPU renders its own objects classified by the $split$ value. As shown in Figure 3, the GPU receiving a larger portion of the view frustum will send the "unwanted" portion of output frame to the other GPU.

VII. Synchronization and Inter-Process Communication (IPC)

In CUDA programming environments, GPUs cannot interact with each other directly. The only way of inter-GPU communication is going through the controls of inter-CPU

Algorithm 1 Computing Screen Splitting Value

LoadBalancing(
in id, vf, threshold, compLevel;
out split, compLevel)

1: $split \leftarrow 0.5$;
2: $increment \leftarrow 0.5$;
3: $subF_l \leftarrow vf$;
4: **while** 1 **do**
5: // updating the left sub frustum
6: **UpdateSubFrustum**$(subF_l, split, vf)$;
7: // balancing the amount of data
8: **for** ith object's AABB **in parallel do**
9: **if** $AABB_i$ inside $subF_l$ **then**
10: $compLevel_l[i] \leftarrow compLevel[i]$;
11: $compLevel_r[i] \leftarrow 0$;
12: **else**
13: $compLevel_r[i] \leftarrow compLevel[i]$;
14: $compLevel_l[i] \leftarrow 0$;
15: **end if**
16: **end for**
17: $sum_l \leftarrow$ the sum of the elements in $compLevel_l$ **in parallel**;
18: $sum_r \leftarrow$ the sum of the elements in $compLevel_r$ **in parallel**;
19: $ratio \leftarrow sum_l / sum_r$;
20: $increment \leftarrow increment \times 0.5$;
21: **if** $ratio < 1/threshold$ **then**
22: $split \leftarrow split + increment$;
23: **else if** $ratio > threshold$ **then**
24: $split \leftarrow split - increment$;
25: **else**
26: $compLevel \leftarrow compLevel_{id}$;
27: **return**;
28: **end if**
29: **end while**

communication, where each GPU is controlled by the process of a CPU core. We use Massage-Passing Interface (MPI) in our system. MPI is a specification that moves the data among processes through cooperative operations on each. MPI are portable, hardware optimized and widely used in High Performance Computing (HPC). Recently, researchers and developers have demonstrate the efficiency of multi-GPU applications using MPI communications to facilitate GPU data movements, such as [24], [25], [26]. In this section, we discuss the MPI-based methods of synchronization and IPC used in our multi-GPU multi-display system.

Synchronization and communication are necessary to unite GPUs, so that they act as co-processors to coordinate the rendering tasks. In Figure 4, we show the requirements from the pipeline of our system, including: (1) only one process is allowed to control the camera at a time. Values of the camera need to be shared between processes during run-time; (2) an inter-GPU communication scheme is needed to exchange

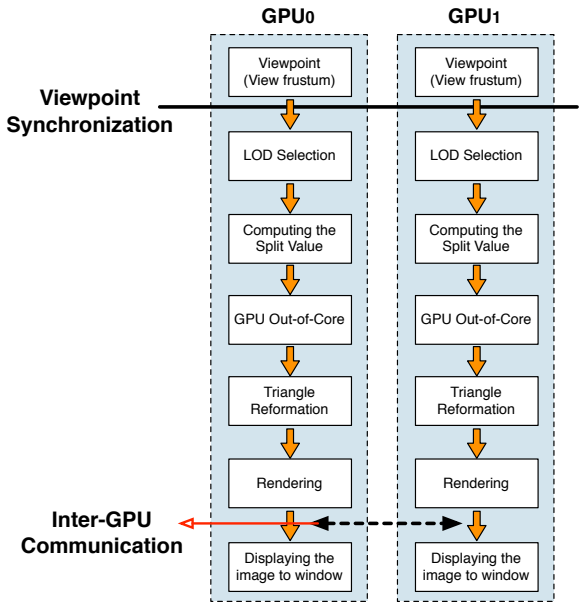

Fig. 4. **The principle of the synchronizations between GPUs.**

framebuffers between GPUs.

Camera movement Sync. When updating the camera viewpoint, only one process is active to response the mouse/keyboard callback events. To have an efficient way of passing the camera values, we use the shared memory that may be accessed by all processes simultaneously without any redundant copy. We select one process as the root, and others send their activation status (e.g., if mouse-overing the window) to the root. The root identifies which process is activated and broadcasts the index of this process to all others. Then, after the active process finishes the camera updates in the shared memory, it will broadcast a finish flag. When other processes received the flag, they will read the updated values for their local computations. We use MPI blocking calls to suspend process executions until the values in the shared memory are safe to use. This is because non-blocking calls return immediately after the calls are initiated without waiting for their finish. In that situation, non-active processes may execute faster than the active one and read the values that have not been updated yet. As a disaster, LOD selection and GPU out-of-core would produce wrong results. Therefore, using the MPI blocking calls is a safe way to guarantee the camera synchronized for all processes.

CUDA Inter-Process Communication for Exchanging Framebuffers. When all processes finish rendering tasks, the output images are maintained in framebuffer on GPUs. The GPU assigned with a larger part of the screen gives its "unwanted" portion of the output to the other GPU. Traditionally, if a GPU wants to send data to another GPU, it has to first send the data to the shared memory on CPU host; then the processes associated to GPUs are synchronized to make the data safe to use on the host; and then this host data is sent to the target GPU. Although CUDA driver supports *pinned* memory

allocation to main memory (e.g., page-locked buffer on RAM) to help reduce the CPU-GPU data transfer overhead, the meander progress of data movements through PCIe buses still reduces performance and increases the needs of host resources. Recently, since the release of CUDA 4.1, Nvidia has removed the limitations of the traditional approach by introducing an efficient method of inter-process GPU-GPU communication, so-called as *CUDA-IPC*. It significantly reduce the overhead of GPU-GPU communication by transferring data through PCIe Express directly in a computer node, rather than going through the host. A process creates the handle that maps its GPU's data buffer. Then, the handle is sent to the target process communication with the help of MPI host-based communication. The target process will open this handle and tell its GPU to require a direct transfer of the associated data from the GPU of the source process. This new CUDA 4.1 feature offers us an optimal performance of exchanging GPU framebuffers. Based on the $split$ value defined in Algorithm 1, the roles of source and target are switched dynamically between GPUs. If $split = 0.5$, it indicates that the screen is partitioned evenly and no need to exchange; if $split < 0.5$, the GPU associated with the left display monitor will be the target that receive the data from the source associated to the right monitor; If $split > 0.5$, the roles between GPUs will be switched the other way round.

VIII. Evaluation and Experimental Results

We evaluate our approach by visualizing the massive 3D model composed of hundreds of millions of triangles. To have an in-depth understanding of its performance, we analyze each computational component by comparing our system with other implementations.

A. Implementation

We have implemented our dual-GPU approach on a workstation equipped with an AMD Phenom X6 1100T 3.30GHz CPU (6 cores), 16 GBytes of RAM, and two Nvidia GTX 580 graphics cards, each of which has 3 GBytes of GDDR5 device memory. Our system is developed using C++, Nvidia CUDA Toolkit v4.2 and OpenGL on a 64-bit Linux system. In our benchmarks, the resolution of OpenGL framebuffer is set to 1024 $\times$ 1024. We generated camera pathsto test the performance of our system. These paths are demonstrated in our complementary video.

B. Performance Evaluation

Our experimental results show that we can interactively visualize Boeing model at the rates of 9-35 fps. We compare our implementation, *Dual-GPU with load balancing (Dual-GPU(B))*, to the other two implementations: *Single-GPU* and *Dual-GPU without load balancing(Dual-GPU(NB)).* All these three implementations use the same camera paths and are constrained with the same amount of vertices and triangles. *Single-GPU* assigns all the workloads for one GPU; *Dual-GPU(NB)* always split the screen in the middle, so that they do not have costs on the computation of load balancing. In the tests, the system performance is bounded by the performance of the slower GPU. Our *Dual-GPU(B)* achieves 1.14 and 1.37 times of speedup comparing to *Dual-GPU(NB)* and *Single-GPU*, respectively.

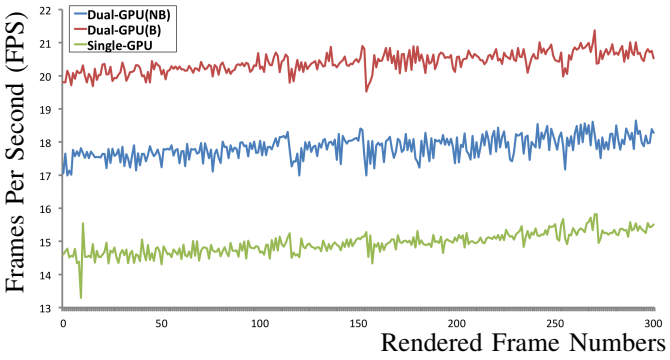

Fig. 6. **The frame rates comparison among three different implementations.**

In Table I, we provide the breakdown of the averaged experimental results based on a total of 300 rendered frames. The column of "*Diff. Triangle Num.*" means the difference of the number of triangles between two GPUs, which is not applicable in *Single-GPU*. The column of "*Visible Triangle Num.*" means the number of triangles visible to the camera (those inside the view frustum). The numbers of triangles are the same, which ensures the same rendering quality in all three implementations. The last five columns show the timing results of all computational components in our system. By using multiple GPUs, since the selected vertices and triangles are distributed to perform fast computation, the FPSs (frame per second) from multi-GPU systems are higher than the FPS of the single GPU system. The *Single-GPU* and *Dual-GPU(NB)* do not include any load balancing-related computation, so that there is no computing time spent on "*Splitting*". But, the computation time of "*GPU Defragmentation*" becomes the significant cost and forces the *Single-GPU* and the *Dual-GPU(NB)* to perform at the lower frame rates than *Dual-GPU(B)*. Figure 6 plots the FPS comparison of the three different implementations in the sequence of 300 frames. Our *Dual-GPU(B)* achieves the best FPS because of the advantages of the load balancing algorithm.

C. Analysis

We give more details of the performance analysis between two dual-GPU implementations. The performance of a GPU scales with the amount of data that it operates on. The larger value of "*Diff. Triangle Num.*" we get in Table I, the bigger performance difference between two GPUs will be in an implementation. Although *Dual-GPU(B)* spends the additional cost to balance distribution of the workloads, the computational times spent on GPU defragmentation, triangle reformation and GL rendering in the lower-performing GPU are smaller than those in the imbalanced implementation of *Dual-GPU(NB)*. Figure 7 shows the difference of the number of triangles between GPUs in the *Dual-GPU(B)* is much smaller than the

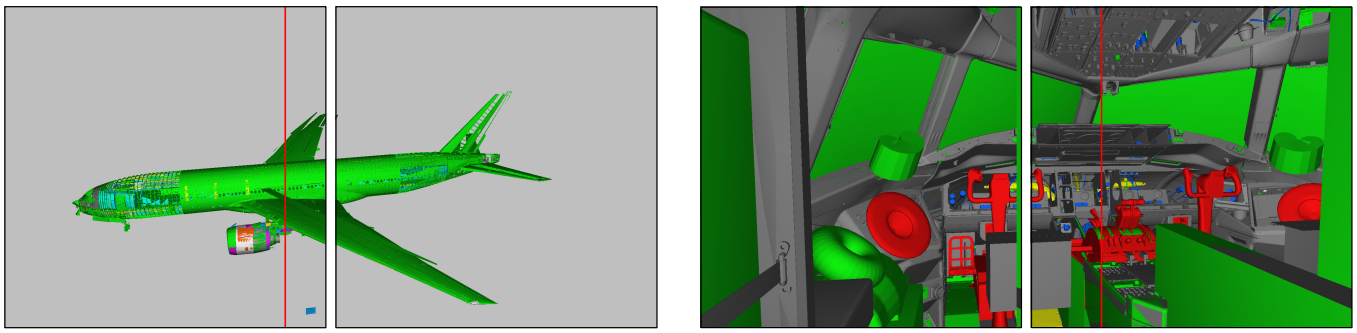

Fig. 5. **The rendered Boeing models in our experiments.** The red vertical lines split the screen space to balance the GPU workloads.

TABLE I
THE BREAKDOWN OF THE RUN-TIME PERFORMANCE.

Approach	FPS	Diff. Triangle Num.	Visible Triangle Num.	LOD Selection	Splitting	GPU Out-Of-Core	Triangle Reformation	GL Rendering
Single-GPU	14.94	—	12.29M	3.44ms	—	29.62ms	3.62ms	30.24ms
Dual-GPU(NB)	17.84	7.94M	12.29M	3.35ms	—	24.54ms	2.85ms	25.31ms
Dual-GPU(B)	20.40	0.37M	12.29M	3.98ms	5.38ms	18.56ms	1.97ms	19.13ms

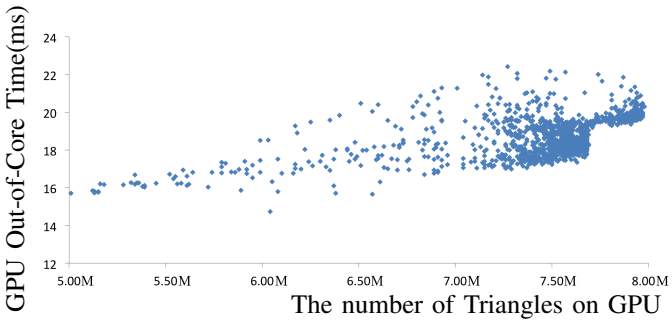

Fig. 7. **The scattered value pairs of GPU Out-of-Core time and the number of triangles on GPU from all rendered frames.**

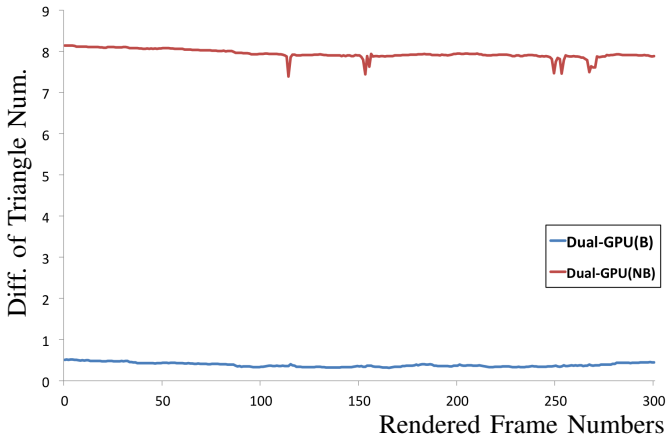

Fig. 8. **The difference of the number of triangles between GPUs with/without the load balancing algorithm.**

Dual-GPU(NB). . *Dual-GPU(B)* modifies the sub frustums at each frame, the frame-different data that needs to be streamed to a GPU may be more than the *Dual-GPU(NB)*. In Figure 8, we plot the relationship between GPU out-of-core time and the number of renderable data on GPU. Each dot in the graph represents a frame. It shows that *Dual-GPU(B)*'s GPU out-of-core time increases linearly towards the increase of the triangle and vertex counts.

IX. Conclusion and Future Works

In this work, we present a multi-GPU multi-display system for high-performance high-resolution rendering of massive-scale 3D models. By using a load balancing strategy, our parallel GPU out-of-core evenly distributes the workload for the GPUs, and each GPU is able to perform its local LOD computations. our system also demonstrates the capability of fully utilizing all available GPU memories, which is important for the improvement of rendering quality. In the future, we would like to extend our multi-GPU approach to GPU clusters. Also, given an extreme complex model, only applying LOD algorithms may not be sufficient. We would like to research on visibility culling algorithms on multi-GPU parallel frameworks.

References

[1] C. Peng and Y. Cao, "A gpu-based approach for massive model rendering with frame-to-frame coherence," *Computer Graphics Forum*, vol. 31, no. 2, 2012.

[2] C. DeCoro and N. Tatarchuk, "Real-time mesh simplification using the gpu," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ser. I3D '07. New York, NY, USA: ACM, 2007, pp. 161–166.

[3] L. Hu, P. V. Sander, and H. Hoppe, "Parallel view-dependent refinement of progressive meshes," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ser. I3D '09. New York, NY, USA: ACM, 2009, pp. 169–176.

[4] H. Hoppe, "Progressive meshes," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 99–108.

[5] H. Hoppe., "View-dependent refinement of progressive meshes," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 189–198.

[6] M. Garland and P. S. Heckbert, "Surface simplification using quadric error metrics," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 209–216. [Online]. Available: http://dx.doi.org/10.1145/258734.258849

[7] M. Garland and P. Heckbert, "Simplifying surfaces with color and texture using quadric error metrics," in *Ninth IEEE Visualization(VIS '98)*, 1998, p. pp.264.

[8] J. C. Xia, J. El-Sana, and A. Varshney, "Adaptive real-time level-of-detail-based rendering for polygonal models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, pp. 171–183, April 1997.

[9] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models," in *ACM SIGGRAPH 2004 Papers*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 796–803.

[10] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha, "Quick-vdr: Interactive view-dependent rendering of massive models," in *Proceedings of the conference on Visualization '04*, ser. VIS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 131–138.

[11] E. Derzapf, N. Menzel, and M. Guthe, "Parallel view-dependent refinement of compact progressive meshes," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2010, pp. 53–62.

[12] E. Derzapf, N. Menzel, and M. Guthe., "Parallel view-dependent out-of-core progressive meshes." in *Vision, Modeling, and Visualization*, 2010, pp. 25–32.

[13] S. Eilemann, *An Analysis of Parallel Rendering Systems*, www.equalizergraphics.com/documents/ParallelRenderingSystems.pdf, 2007.

[14] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A scalable parallel rendering framework," *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 3, pp. 436–452, 2009.

[15] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher, "Large data visualization on distributed memory multi-gpu clusters," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 57–66.

[16] F. Erol, S. Eilemann, and R. Pajarola, "Cross-segment load balancing in parallel rendering," in *EGPGV*, 2011, pp. 41–50.

[17] S. Marchesin, C. Mongenet, and J.-M. Dischler, "Dynamic load balancing for parallel volume rendering," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2006.

[18] S. Deshpande, C. Yuan, and I. Daly, Scott asnd Sezan, "A large ultra high resolution tiled display system: Architecture, technologies, applications, and tools," in *16th International Display Workshops*, 2009.

[19] S. Melax, "A simple, fast, and effective polygon reduction algorithm." in *Game Developer*, 1998, pp. 44–49.

[20] T. A. Funkhouser and C. H. Séquin, "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '93. New York, NY, USA: ACM, 1993, pp. 247–254.

[21] C. Peng, S. I. Park, Y. Cao, and J. Tian, "A real-time system for crowd rendering: parallel lod and texture-preserving approach on gpu," in *Proceedings of the 4th international conference on Motion in Games*, ser. MIG'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 27–38.

[22] D. Schiffner and D. Krömker, "Parallel treecut-manipulation for interactive level of detail selection," in *20th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, vol. 20, no. 1-3, June 2012.

[23] M. Wimmer and D. Schmalstieg, "Load balancing for smooth levels of detail," Vienna University of Technology, Tech. Rep. TR-186-2-98-31, 1998.

[24] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. Panda, "Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Träff, S. Benkner, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2012, vol. 7490, pp. 110–120.

[25] S. Potluri, H. Wang, D. Bureddy, A. Singh, C. Rosales, and D. Panda, "Optimizing mpi communication on multi-gpu systems using cuda inter-process communication," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, may 2012, pp. 1848 –1857.

[26] H. Wang, S. Potluri, M. Luo, A. Singh, X. Ouyang, S. Sur, and D. Panda, "Optimized non-contiguous mpi datatype communication for gpu clusters: Design, implementation and evaluation with mvapich2," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, sept. 2011, pp. 308 –316.