Large-scale Virtual Texturing on a Distributed Rendering System

Revanth N R
Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India 500032
Email: revanth.nr@research.iiit.ac.in

P J Narayanan
Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India 500032
Email: pjn@iiit.ac.in

***Abstract*—There has been a profound interest of late in the digitization and reconstruction of historical monuments. Rendering massive monument models requires a cluster of workstations because of the computational infeasibility of rendering over a single machine. Moreover, interactive rendering of these massive models in an immersive environment is only possible over a cluster of machines. In this paper, we present a design of distributed rendering system to efficiently handle models with massive textures. A server holds the skeleton of the whole model and divides the screen space balancing the rendering load among multiple clients. Each client loads only the required geometry and textures to render its sub-scene. We present a virtual texturing method for handling massive textures over the distributed rendering system. These textures are combined into a texture atlas which is split into equally sized tiles. A virtual texture is built over this atlas with each pixel representing a tile in the atlas. An efficient caching module loads only the required tiles into the memory, that are identified using the virtual texture. A fragment shader uses the virtual texture as a mapping to the physical texture in memory to generate the fragments. We demonstrate the performance of our system over a 350M triangles and 500 gigapixel textured model of Vittala temple.**

I. Introduction

3D models are becoming massive with complex geometrical meshes, heavy textures etc. Conventional rendering architectures and techniques cannot render such massive environments even with the latest hardware. These massive models and their high detail textures may not fit into any single GPU memory or even CPU memory. Using the CPU memory as a secondary cache to GPUs allows us to render larger models to an extent. When this surpasses the CPU memory, the storage device on the system can provide buffering to the CPU memory at high latency. However, the ever growing model sizes require a scalable system. A distributed system can solve this problem.

The latest hardware provides distributed storage with hardly any higher latency compared to the local storage devices. This distributed storage along with multi-level caching allows the GPU to be able to render the massive models. However, a single GPU is always limited by the amount of data processed per second, thereby not allowing us to render a massive model at interactive frame rates. Distributed rendering provides a solution for this. This situation is already prevalent in the rendering of computer generated movies, where they break up the rendering process into a series of computation tasks performed on large render farms. But these techniques are not acceptable when interactive rendering is required to navigate through a massive model.

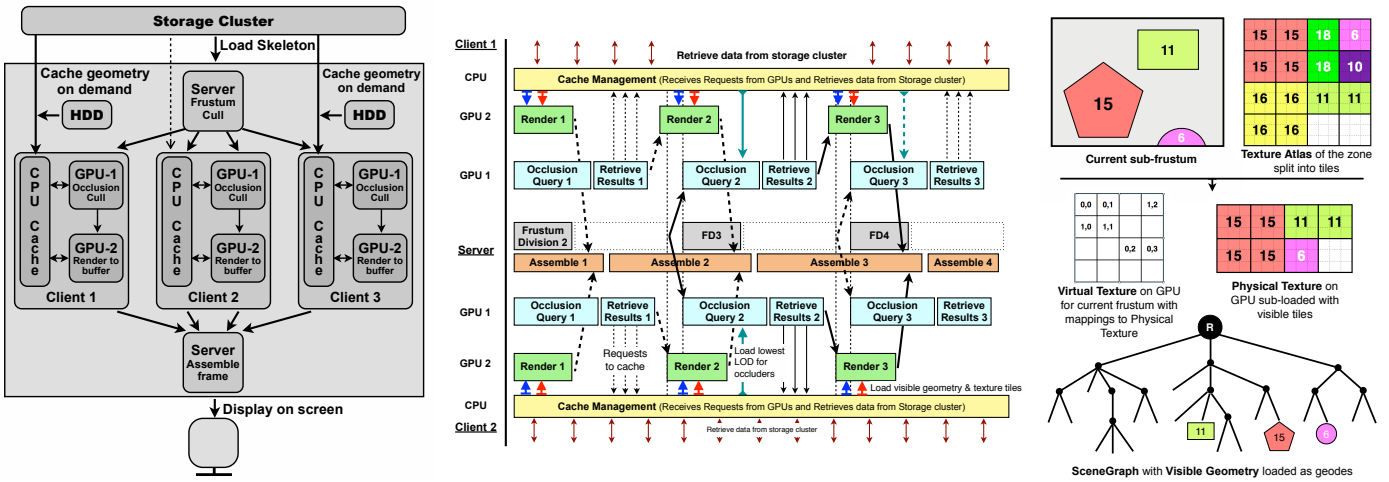
Distributed Massive Model Rendering (DMMR) system [1] provides a scalable method to efficiently handle and render massive geometric models on a cluster of multi-GPU systems. However, this system does not support textured geometries in the model. In this paper, we present an efficient method to handle massive textures with a massive model on the DMMR system. Our virtual texturing method operates on each client in the cluster to determine the visible parts of the massive textures using the first GPU. Appropriate mipmaps of these visible chunks are loaded and rendered on the second GPU. We demonstrate the design and implementation of virtual texturing and multi-level caching for the DMMR system over a 350M triangles and 500 gigapixel textured model of Vittala temple.

II. Prior Work

MegaTexture by John Carmack is a texture allocation method that uses a single large texture instead of multiple small textures, loading different sections of it into memory as they become visible. Virtual texturing [2] is an advanced approach to megatextures which adds a level of indirection to the textures, similar to pagetables. This virtual texturing allows each fragment to access a suitable tile based on the required resolution following the standard OpenGL mipmapping calculations. By selectively loading parts of the texture, virtual texturing allows for higher resolution textures than possible with traditional techniques. However, this technique also adds a significant overhead to the renderer by having to determine the working set and loading it to the GPU, thus making it expensive. A number of parallel rendering techniques [3], [4] have been introduced before, that are based on different rendering pipelines. Some of these systems and APIs like Garuda [5] and Equalizer [6] provide scalable rendering on shared memory systems. Works like Gigawalk [7], [8] address rendering a large model interactively on a single system while Garuda and Equalizer systems present methods to render models on multi-headed displays for high resolutions.

Our earlier work, the DMMR system [1] provides a new approach to efficiently handle and render massive geometric models on rendering clusters. This uses a client-server framework with the server distributing the render load among the clients. Each client performs occlusion culling on its sub-

978-1-4673-8564-0/15/$31.00 2015 IEEE



(a) Distributed Rendering Framework

(b) Distributed Rendering Pipeline

(c) Virtual Texture and Physical Texture states for current frustum

Fig. 1. Components of the Distributed Rendering System with Virtual Textures

frustum and renders the visible geometry. In this paper, we extend this system to handle massive textures using virtual texturing and a multi-level cache management system.

III. DMMR WITH TEXTURES

The DMMR system employs a client-server approach on a cluster of multi-GPU systems with the server and clients communicating using MPI (fig. 1a). Additionally, a storage cluster maintains communication with the clients of the DMMR cluster over high speed network. This entire system comprises of three primary components.

1) Load Division: The DMMR system provides load balanced frustum division which runs on a server equipped with a CUDA enabled GPU. It is responsible for allocation of sub-frustums to the clients. The rendered sub-frames are collected from each client and are assembled as a whole frame.

2) Distributed Rendering: This module operates on each client in the rendering cluster. Each client is equipped with two GPUs; one supporting hardware occlusion queries and the second GPU with more texture memory for rendering. DMMR supports visibility determination of geometries for the current sub-frustum on the first GPU and sub-frame rendering on the second GPU. We extend the DMMR system with texture visibility determination and sub-frame rendering with virtual textures to support massive textures. A multi-level cache is also maintained for each client. This module communicates with the remote storage to cache the geometries and textures.

3) Remote Storage: The huge dataset comprising of preprocessed data, geometric meshes and textures required to render any model, is stored on a storage cluster. This module operates on the storage cluster providing uninterrupted data transfer to the rendering cluster, sending the requested data to the clients.

A. Scene Representation and Pre-processing

A 3D model comprises of massive geometry and large number of massive textures. The DMMR system represents the model as a scenegraph which is split into geometry nodes (geodes) and a skeleton scenegraph containing the transformations which are stored separately. Additionally, we combine the textures into large texture atlases. The 3D model is preprocessed through a series of steps as shown in fig. 2. A spatial kd-tree is built and the entire model is categorised into multiple zones based on the spatial location of the geodes. A texture atlas is created for each zone with the textures of all the geodes in the corresponding zone. Each texture atlas is split into tiles of fixed size. The weight of each geode is computed based on its polygon count and number of tiles mapped onto it. All this pre-processed data is stored on the storage cluster.

B. Distributed Rendering Pipeline

The distributed rendering pipeline of DMMR system splits the regular graphics pipeline into three parallel modules. Each module executes on a different GPU. We extend these modules to process textures. The pipeline is extended to include a fourth cache management module running in parallel to the rest, efficiently utilising the idle CPU cycles (fig. 1b).

Load Balanced Frustum Division: The server performs the frustum culling and division on its GPU to generate sub-frustums for each client. The DMMR system performs an optimal load balanced frustum division based on the precomputed weights of the geodes in the current frustum.

Visibility Determination: DMMR performs visibility determination on each client on the geodes in its sub-frustum. Occlusion queries are issued on the bounding boxes of the geodes, to be tested against lowest LOD geometry of the visible geodes from previous frame. These queries are resolved by rendering the sub-frustum at half the respective viewport dimensions. We intercept the query results to collect the list of visible geodes. Their LODs are determined based on number of pixels they are visible in. The texture tiles associated with each of the visible geodes are also marked visible and their appropriate MipMap is chosen similar to the LOD. The list of visible geodes and tiles, is split into cached and uncached components. Requests are issued to the cache management system to cache the uncached components.

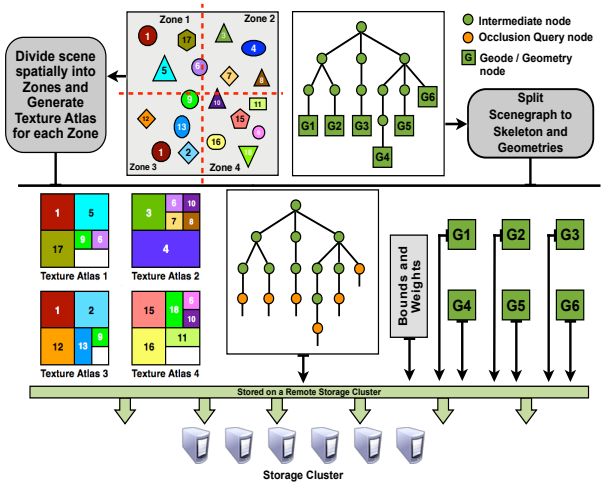


Fig. 2. Pre-processing and Storage of Scenegraph and Textures

Sub-frame Rasterization: Rasterization is performed by each client on its second GPU for the visible objects. The appropriate LOD geometries of visible geodes are attached to the skeleton scenegraph. The virtual texture on the GPU is updated for the visible tiles. The visible cached tiles are loaded to the physical texture on the GPU in their appropriate location, followed by loading the uncached list of tiles to the GPU. We wait till all of them are cached and loaded to GPU by the cache management module. A fragment shader renders the appropriate texel for each pixel using the virtual texturing method. Rendered sub-frames are then transferred to the server to be assembled into complete frame.

Cache Management: The cache management system runs on each client asynchronous with the rest of the modules. The requests to cache the needed data are queued and processed by retrieving it from the storage cluster. A multi-level cache is maintained and handled by this module. It utilises the CPU cycles and network bandwidth while the other modules perform GPU operations. This ensures that the data is cached when the rasterization module accesses it.

All the clients operate asynchronously and the four modules execute in parallel. However, for a single frame, the result of one module is required for the next module thereby inducing a delay of 3 frames for the user events to reflect. For example, when the 2^{nd} frame is on display, frustum culling of 5^{th} frame is run on the server, occlusion culling of 4^{th} frame is run on the first GPU of each client and the rasterization of 3^{rd} frame is done on the second GPU of each client, thereby executing the three modules in parallel. Hence the user events during 2^{nd} frame are reflected on the 5^{th} frame.

IV. Virtual Texturing

Virtual textures allow us to optimise the GPU memory usage by dynamically loading only the visible parts of large textures every frame without having to modify the texture mapping of the geometry. A virtual texture (VT) is a mapping between the texture atlas and the physical texture residing on the GPU. Each texel in VT represents a tile in the atlas. This texel consists of the location and mipmap level of the tile in physical texture. The dimensions of a VT is same as the number of tiles in the corresponding atlas. Every mipmap level of the texture atlas has a corresponding mipmap level of VT.

Physical texture (PT) is the largest possible texel array allocated in the GPU texture memory. This is represented as a large array of tiles into which the visible tiles are loaded. A texel in VT consists of the mipmap level of the tile and index of tile array where the actual tile resides. The corresponding memory location of the tile in PT is obtained from this index and mipmap level. Unlike VT, PT does not have any mipmap levels. This is just a collection of tiles from various locations and mipmap levels of the atlas.

When a tile of the texture atlas is identified as visible, the tile is copied into an unmapped location in the PT tile array. The mipmap level of this tile and the index of this tile in PT is updated in the corresponding texel of VT (fig. 1c). During the fragment shader, the VT is looked up using the texture coordinates and the retrieved texel provides the location of the tile in PT. Using these, the appropriate texel is retrieved from PT and the fragment is updated with this value.

Using a virtual texture for a texture atlas effectively loads a scaled down version of the atlas, only that each texel in this scaled down version points to a dynamic memory location of the actual tile in the physical texture. As the texture atlas becomes massive, the VT itself becomes a huge texture allowing lesser space for PT, which destroys its purpose. Hence we induce another level of indirection by dividing the scene into zones and having a texture atlas for each zone. This limits the size of a texture atlas and the size of a VT at an optimal value. When the view frustum crosses over into another zone, the VT is updated to represent the atlas corresponding to the new zone. The appropriate tiles in the PT are unmapped when the VT is updated every frame. This allows the newer tiles to replace the unmapped tiles when needed.

V. Cache Management

A multi-level cache management module (CMM) operates in parallel to other modules asynchronously on each client. The GPU memory acts as L1 cache, holding the geometries as VBOs and the texture tiles in the physical texture. L2 cache resides on the CPU memory providing the lowest latency transfers to the GPU. L3 cache resides on the local storage device of each client. L2 and L3 caches are managed by the CPU while GPU manages the L1 cache. An LRU algorithm is followed to unload data from each cache level when they run low on memory.

All requests to cache the required data are queued. The CMM processes each of these requests, checking for the availability of the data in L2. If the data is missing in L2, L3 is looked up for the data and loaded into L2 if available. If the lookup in L3 fails, a request to retrieve data from storage cluster is added which is processed by a different thread. This thread proceeds to process the next cache request.

Data retrieval thread processes the requests and retrieves the data from the storage cluster. The retrieved data is updated in L3 and L2 caches. While this data is being retrieved over network, the other thread utilises the CPU cycles by processing the requests and loading locally available data onto L2. The rasterization module manages the L1 cache which resides on

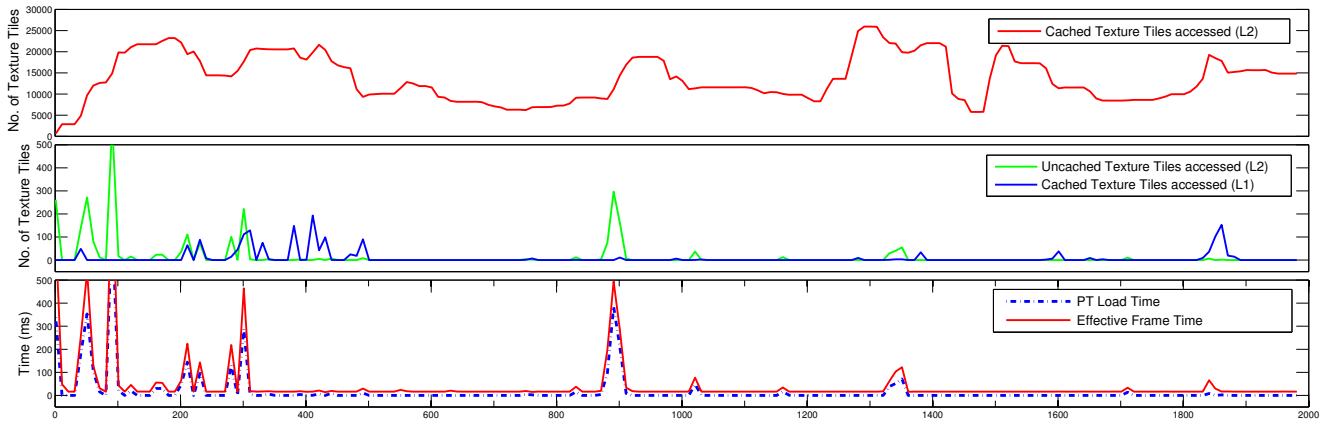


Fig. 3. Averaged over every 10 frames. (a) Cached tiles in L2. (b) Uncached tiles in L2, Cached tiles in L1. (c) Frame time, PT load time (in ms).

the GPU. It queries the CMM for data to be loaded onto the L1 cache. These queries for data are immediately returned with a pointer to data location in L2 cache if available. A miss is returned if the data is unavailable in L2 cache, thereby allowing the module to wait until L2 has been updated before repeating the query. All the threads are notified by a signal every time L2 cache has been updated so that any waiting threads can query for the cached data.

VI. Experimental Results

We setup the distributed rendering system with a server and 4 dual-GPU clients, all connected over a private gigabit ethernet. The server hosts an Nvidia K40 GPU. Each client hosts an Intel i7 processor, 32GB memory, an Nvidia C2070 GPU for occlusion culling and Geforce GTX Titan GPU for rendering. We performed our tests on a 3D model of Vittala temple which is made up of 350M triangles and 500 gigapixels of textures spread across 4000 geodes with 5 levels of texture mipmaps and 3 LODs for geometry. We used a virtual texture of size 4096x4096 pixels with each tile being 256x256 pixels and the physical texture limit being 16384 tiles (or 4GB GPU memory). We rendered a walkthrough of this model at 1080p resolution on the system and analysed the cache statistics as well as the rendering performances over 2000 frames. On an average we have been able to obtain interactive frame rates of approximately 25 FPS, with occasional delays (fig. 3) when a significantly large chunk of texture is needed to be cached into CPU memory and transferred to the GPU.

The plots show the cache statistics like the number of uncached texture tiles being loaded into CPU memory (L2) (fig. 3b), number of cached tiles being transferred to the GPU (L1) (fig. 3a) and the number of tiles residing in the GPU (L1) that are being accessed (fig. 3b) every frame on a single client. Also the time taken to render the frames (fig. 3c) and the time taken to load textures into PT (fig. 3c) are plotted. All these plots are averaged over every 10 consecutive frames.

It should be observed that in the initial frames where the caches are empty, uncached file accesses are very high. This leads to longer texture loading times and longer frame times. Frame time stabilised to an average of 25ms once the caches are stabilised. We can also notice occasional spikes in the number of uncached tile accesses and frame time, between 200^{th}-300^{th} frames and again at around the 900^{th} frame. This behaviour is due to a number of new objects with very high detail textures becoming visible in the frustum. Loading all these texture tiles into the memory delays the frame rendering.

It should be noted that having a large number of objects with high detail textures will distribute the caching load across multiple frames as and when each object becomes visible. Having a lesser number of large objects, where each object has a very huge texture, adds heavy load on cache management when that object becomes visible. It can be observed from these results that the system can scale well to provide interactive frame rates for massive models with larger number of high resolution textures and geodes, as opposed to smaller number of very high complexity geodes.

VII. Conclusion

In this paper, we presented a virtual texturing method to handle massive textures on a distributed rendering framework. We also presented a multi-level cache management system to efficiently utilise the idle CPU cycles and network bandwidth to cache the required geometries and textures while the GPUs are processing the model. We demonstrated these by rendering a walkthrough of a massive Vittala Temple model comprising of 500 gigapixel texture and 350M triangles on the DMMR system obtaining interactive frame rates.

References

[1] N. R. Revanth and P. J. Narayanan, "Distributed massive model rendering," ser. ICVGIP '12, 2012.

[2] J. Obert, J. M. P. van Waveren, and G. Sellers, "Virtual texturing in software and hardware," in *ACM SIGGRAPH 2012 Courses*.

[3] R. Samanta, T. Funkhouser, and K. Li, "Parallel rendering with k-way replication," ser. PVG, 2001.

[4] S. Eilemann, A. Bilgili, M. Abdellah, J. Hernando, M. Makhinya, R. Pajarola, and F. Schürmann, "Parallel rendering on hybrid multi-gpu clusters," in *EGPGV*, 2012.

[5] Nirnimesh, P. Harish, and P. J. Narayanan, "Garuda: A scalable, geometry managed display wall using commodity pc's," *TVCG*, 2007.

[6] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A scalable parallel rendering framework," *IEEE TVCG*, 2009.

[7] A. Sud, N. K. Govindaraju, and D. Manocha, "Gigawalk: Interactive walkthrough of complex environments," *EGWR*, 2002.

[8] C. Erikson, D. Manocha, and W. V. Baxter, III, "Hlods for faster display of large static and dynamic environments," ser. I3D, 2001.