GPU-based Image Compression for Efficient Compositing in Distributed Rendering Applications

Riley Lipinksi[*]
University of St. Thomas

Kenneth Moreland[†]
Oak Ridge National Laboratory

Michael E. Papka[‡]
Argonne National Laboratory
Northern Illinois University

Thomas Marrinan[§]
University of St. Thomas
Argonne National Laboratory

Abstract

Visualizations of large-scale data sets are often created on graphics clusters that distribute the rendering task amongst many processes. When using real-time GPU-based graphics algorithms, the most time-consuming aspect of distributed rendering is typically the compositing phase – combining all partial images from each rendering process into the final visualization. Compositing requires image data to be copied off the GPU and sent over a network to other processes. While compression has been utilized in existing distributed rendering compositors to reduce the data being sent over the network, this compression tends to occur after the raw images are transferred from the GPU to main memory. In this paper, we present work that leverages OpenGL / CUDA interoperability to compress raw images on the GPU prior to transferring the data to main memory. This approach can significantly reduce the device-to-host data transfer time, thus enabling more efficient compositing of images generated by distributed rendering applications.

Index Terms: Computing methodologies—Computer graphics—Image compression; Computing methodologies—Parallel computing methodologies—Parallel algorithms—Massively parallel algorithms

1 Introduction

Data sets produced by simulations and collected from digital instruments and sensors are often too large to analyze on a single computer. Instead, researchers tend to leverage remote High-Performance Computing (HPC) centers where work can be distributed amongst many nodes of a computational resource. One common analysis task that researchers leverage to gain an understanding of the underlying data is three-dimensional (3D) visualization [31, 33].

Creating large-scale visualizations depends on distributed rendering applications that are capable of splitting geometric data between multiple GPUs (usually on multiple nodes of a graphics cluster). Multiple approaches to distributing rendering on parallel nodes exist [18], but the most fitting for large-scale HPC use an image compositing technique [37]. In this approach, partial images are rendered independently on each process in the distributed rendering task. The partial images are then composited into the final visualization (example shown in Fig. 1). When using real-time graphics algorithms, this compositing phase is typically the bottleneck in distributed rendering applications for two reasons: 1) rendered partial images must be copied from GPU memory to main memory, and 2) images now in main memory of each rendering process must be sent between rendering processes via network interconnect. This overhead can significantly increase overall job time and inhibit the ability to achieve interactive frame rates when streaming for real-time viewing and interaction [7, 21].

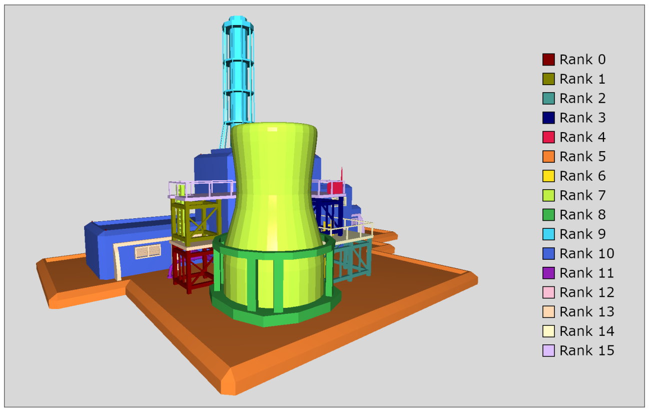

Figure 1: Composited rendering of a nuclear power station using 16 processes. Each process has drawn its portion of the model using a unique color.

While a few different distributed image compositors exist, IceT [19, 21] serves as an industry standard and has been integrated into widely used distributed rendering applications such as ParaView [25] and VisIt [35]. Prior to our work, IceT did leverage a form of compression for sending image data between rendering processes, but it still required copying uncompressed color and depth data from each rendering process's GPU to main memory. Our work aimed to implement image compression on the GPU, which yielded two benefits: 1) faster compression by leveraging the parallel architecture of GPUs, and 2) smaller data transfer size between a GPU and its host's main memory.

After covering prior related work on distributed rendering and compositing, this paper describes the GPU implementation of IceT's *Active Pixel* image compression format – a form of run-length encoding that accounts for image depth in addition to color. Next, it covers updates made to the IceT library to support modern OpenGL applications and GPU-based image compression. After describing implementation details, a series of performance measurements are provided in order to demonstrate the impact of GPU-based image compression on compositing in distributed rendering applications. Finally, the paper will conclude with limitations and of this work and future improvements to be made.

2 Related Work

As we aimed to integrate GPU-based image compression into a distributed rendering compositor, we investigated prior work done on distributed rendering and on image compression using GPUs.

2.1 Distributed Rendering and Compositing

Compositing images in distributed rendering applications occurs in one of two ways – two-dimensional (2D) image space compositing (i.e. stitching sub-images together) or three-dimensional (3D) image

[*]e-mail: lipi8497@stthomas.edu
[†]e-mail: morelandkd@ornl.gov
[‡]e-mail: papka@anl.gov
[§]e-mail: tmarrinan@stthomas.edu

space compositing (i.e. blending overlapping pixels). In the former method, the output image is partitioned into 2D regions. Geometry is sent to the processor responsible for that region of image space and rendered locally. These regions are then stitched together for the final image. In the latter method, the geometry is partitioned among processors, and each processors renders a full image with its portion of the geometry. These full images of partial geometry are blended together using a per-pixel operation that either blends colors or uses depth information to find the "front" color.

The image stitching approach is a natural fit for tiled displays, which partition the screen space with hardware [10]. Several rendering systems such as WireGL [12], CGLX [4], and ClusterGL [23] take advantage of this fact by intercepting OpenGL calls and streaming them to the appropriate nodes. Although these systems scale well with the size of the display, they suffer from potential load balancing problems with respect to the geometry.For example, if a model is only visible on two tiles, then only the two nodes responsible for rendering those tiles would receive the draw commands for that model.

To improve the load balancing of screen partitions, Samanta et al. [30] dynamically repartition the screen space. Areas of the screen with more geometry are partitioned into smaller regions so that each region takes roughly the same amount of time to render. These screen partitions then need to be read back, redistributed, and re-stitched to the physical display. The idea is periodically rediscovered by other rendering systems [5, 27]. Although this stitching approach only needs to handle color data, there is one major drawback – geometric models can cross sub-image boundaries and therefore need to be loaded on multiple rendering nodes.

In contrast, the image compositing approach where a stack of images, each generated by a different process, are blended together pixel-by-pixel has a high initial overhead – including reading back render buffers, transferring pixel data, and blending pixels – but scales very well with respect to the size of data. The approach is shown to work well at the largest scales [3, 21]. Much of the work on image compositing focuses on the reduction network for the images [15, 16, 20, 24, 26, 40], but there have been multiple other optimizations including amortizing costs across multiple renderings [14], interlacing images for better load balancing [32], improved gathering [9], and of course compression [1, 22, 39], which is the focus of this paper.

Some rendering systems are designed for multiple modes of parallel rendering. Chromium [13] extends the functionality of WireGL [12] to customize the streaming behavior of OpenGL calls, insert new behavior in the rendering system, and provide "out of band" communication to blend image pixels. More recently, Equalizer [6, 7] serves as a full fledged distributed rendering framework that incorporates dynamic load balancing of geometry and supports advanced rendering techniques such as stereoscopy for CAVE virtual reality displays. For the image compositing portion of the distributed rendering task, Equalizer leverages region of interest detection and basic image compression to reduce the data sent between rendering processes. This enables distributed rendering applications based on Equalizer to achieve interactive frame rates.

IceT [21] is an image compositing library that differentiates itself from other systems in a couple ways. First, when compared to other pixel blending systems, it reduces the time spent compositing by leveraging the fact that rendered sub-images often have significant empty space. This allows IceT to remove pixels in empty regions from inter-process communication, thus making better use of available network bandwidth. It also allows IceT to ignore the empty regions when compositing sub-images together, which reduces the computation time. Second, unlike Equalizer, it decouples itself from the rendering. This makes it more flexible so that it can work with a variety of rendering interfaces (including, but not limited to OpenGL).

2.2 GPU-based Image Compression

The primary purpose of most image compression algorithms is to reduce the image's data size while maintaining image quality. There are two types of image compression – lossless and lossy. Lossless compressed images can be fully recovered to their original uncompressed state, whereas lossy compressed images cannot. In either case, compression speed is often left as a secondary goal at best. Thus many compression algorithms are not well suited for real-time distributed rendering and compositing.

One form of lossless image compression that is computationally efficient is run-length encoding (RLE) [28]. RLE images group sequences of adjacent pixels with the same color by simply specifying the number of pixels in a run and the color for that run. RLE compression works particularly well when there are large swaths of uniform color in an image. Rutter [29] created a parallel implementation of RLE compression that could be computed on a GPU. Although the computational complexity is increased from the serial version, the overall computation time is significantly reduced due to the massive parallelism provided by GPUs.

Another lossless compression scheme is Huffman coding [11]. Huffman coding works with any generic data type and is used in common image formats such as JPEG and PNG. A binary tree is created based on the frequency that each symbol occurs in a data set. Symbols that occur more frequently are at a higher level in the tree whereas symbols that occur more rarely are at a lower level in the tree. Variable-length keys are then created for each symbol, corresponding to their location in the tree. The final compressed data contains both the tree and the sequence of keys that match the input data. Huffman coding works particularly well when there are patterns that repeat frequently (whether sequentially or not). Yamamoto et al. [38] demonstrate a method for efficiently parallelizing Huffman coding for execution on a GPU. Results from their work show that the parallel compression algorithm is computationally efficient – compute time on the order of a few milliseconds per GB of input data and about 100x faster than serial execution on the CPU.

Since GPUs are designed for computer graphics, texture-based compression schemes, such as DXT1, are also well suited for parallel encoding. DXT1 is a lossy compression scheme that always results in an 8:1 compression ratio for raw color images. For the DXT1 compression scheme, 4×4 blocks of pixels are independently compressed, thus making the compression algorithm embarrassingly parallel and well suited for computing on a GPU [2, 8]. Each block contains four colors – two colors are explicitly defined using RGB565 format (i.e. 5 bits for red, 6 bits for green, and 5 bits for blue), while the other two are implicitly defined as linear interpolations between the first two. Each of the 16 pixels in the 4×4 block then uses 2 bits to index which of the four possible colors is closest to its original uncompressed color. One other benefit of texture compression formats such as DXT1 is that they do not need to be decompressed prior to rendering since they are natively supported by most graphics interfaces.

2.3 Compression for Distributed Rendering Compositing

Unlike most situations where compression is used, the main concern in a distributed rendering application is speed as opposed to data size. As Makhinya et al. write in their paper about fast compositing for cluster-parallel rendering, "... data reduction can be achieved using image compression. However, this must meet demanding requirements, as its overhead has to be strictly smaller than any transmission gainings, which can be difficult to achieve." [17]. Since our work focused on performing image compression on the GPU, we ensured that compression compute time + device-to-host memory transfer time of compressed images was strictly less than device-to-host memory transfer time of the raw images.

3 Active Pixel Encoding

Rendering pipelines leverage a framebuffer for converting three-dimensional geometry to a two-dimensional grid of pixels. This framebuffer contains both a color and a depth value for each pixel. Color is most often represented using four unsigned 1-byte values that represent the red, green, blue, and alpha (RGBA) intensity for a given pixel, where alpha is the opacity. Depth is most often represented using a single 4-byte floating point value $[-1.0, 1.0]$, where -1.0 is near the virtual camera and 1.0 is far away from the virtual camera.

IceT leverages Active Pixel encoding to compress RGBA and depth information for compositing. Active Pixel encoding is a specialization of RLE that creates runs of *active pixels* (pixels that contain rendered geometry), and *inactive pixels* (pixels that do not contain rendered geometry and therefore simply store a background color value).

To determine whether or not a pixel is an active pixel, the depth value can be checked. If the depth value for a given pixel is 1.0, that means no geometry in the view volume projects to that pixel, and therefore it is inactive. If the depth value for a given pixel is anything other than 1.0, that means rendered geometry does project to that pixel, and therefore it is active.

Each run in the Active Pixel format is defined as follows: number of inactive pixels (32-bit integer), number of active pixels (32-bit integer), list of RGBA and depth values for each active pixel. The IceT API also has application developers define the desired background color so that when decompressing a final composited image, the remaining inactive pixels can be appropriately colored. Prior to our work, Active Pixel encoding was performed on the CPU, which required raw RGBA and depth buffers to be copied off of the GPU when using hardware accelerated rendering.

3.1 CPU-based Compression

Active Pixel encoding was selected for use with IceT because it exhibits the following properties: 1) *Fast encoding* – each pixel only needs to be visited once (i.e. run time $\mathcal{O}(n)$); 2) *Free decoding* – compositing can be done directly on encoded images (i.e. no decoding necessary until display of final result); 3) *Faster Blending* – pixels in inactive pixel regions are easily skipped; 4) *Effective compression* – often a significant number of inactive pixels resulting in good data size reduction; 5) *Good worst case behavior* – no image will even add more than 8 bytes to the uncompressed data size (i.e. image with no inactive pixels will add two 4-byte integers for the inactive/active run counts) [19].

An RGBA depth image can be encoded in the Active Pixel format by using a single loop to iterate over each pixel. For each pixel, its depth value is checked to determine whether it is inactive or active. Only active pixels need to copy their RGBA and depth values into the compressed image buffer. Additionally, a counter is employed to keep track of how many consecutive pixels are inactive / active. These run lengths are also copied into the compressed image buffer.

3.2 GPU-based Compression

While Active Pixel encoding on the CPU is quite fast, copying raw uncompressed RGBA and depth information from GPU memory to main memory can be relatively time consuming. Therefore, when using hardware accelerated rendering, it becomes advantageous to perform compression on the GPU prior to copying data to main memory. A secondary benefit is that the massively parallel architecture on a GPU can lead to even faster computation for Active Pixel encoding.

For our work, we selected Thrust [34] (a C++ template library for CUDA) as a General-Purpose computing on Graphics Processing Units (GPGPU) library. This choice was made since Thrust includes a number of parallel algorithms that are useful for RLE-based compression, such as inclusive / exclusive scan (new list that contains the running total of all elements in an input list) and reduce-by-key (summing or counting elements in an input list grouped by a common key). In order to access image data already on the GPU, we leverage OpenGL / CUDA interoperability. This enables Thrust-based algorithms to read directly from OpenGL textures without any need for uploading data from main memory or copying textures to a separate GPU buffer. While our implementation uses OpenGL and Thrust, we note that the algorithms are not language or library dependent and could be ported for use with other rendering and GPGPU APIs (e.g. Vulkan-based rendering and compute [36]).

Parallel encoding of a raw RGBA and depth image into the Active Pixel format is split into five steps (where each step leverages the massive parallelism available on a GPU):

1. For each pixel, determine whether or not it *is active* and determine whether the pixel starts a *new run* of active/inactive pixels. Results output to two integer buffers (where 'no' = 0 and 'yes' = 1).

2. Perform an inclusive scan on the *new run* buffer (output from Step 1). This will output a buffer whose values are such that pixels in the same run will have the same value, and thus serves as a *run ID*.

3. Perform a reduce-by-key with the *run ID* buffer (output from step 2) as the keys and a constant value 1 for performing the sum operation. This will essentially count the number of pixels in each run and thus outputs a buffer that contains *run lengths*.

4. Perform an exclusive scan on the *is active* buffer (output from Step 1). This will output a buffer whose values represent its *active index* (how many active pixels exist before each pixel).

5. Write the compressed image. For each pixel, if it is active, determine the correct offset in the encoded image buffer (using Equation 1) and copy RGBA and depth values to that location. Additionally, if the pixel starts a new run, copy run lengths for the current active run and the preceding inactive run at the location just before the offset for the RGBA and depth values for this pixel in the encoded image. Finally, if the pixel is the final pixel in the image, it should output the total size of the compressed image and, if the final pixel is inactive, copy inactive run length along with an active run of length 0 at the end of the encoded image. For Equation 1, note that p_2 will be the same as p_1 for active pixels. When computing offset for a final pixel that is inactive, r_{p_2} will be the ID of the active run that precedes p_1.

$$\text{ApOffset}(a_{p_1}, r_{p_2}) = 8 * \left(a_{p_1} + \frac{r_{p_2} - 1}{2} + 1\right) \tag{1}$$

where:

$$a_{p_1} = \text{number of active pixels prior to pixel } p_1,$$
$$r_{p_2} = \text{ID of run that pixel } p_2 \text{ belongs to}$$

Pseudocode for our parallel Active Pixel encoding is shown in Algorithm 1. While the parallel implementation reduces run-time, it is worth noting that it is more complex than the serial implementation (requires iterating over the pixels more than once). It also has a larger memory footprint since intermediate results must be stored prior to writing the final compressed image. Since the *new run* and *is active* buffers only store 0s and 1s, they can be arrays of 1-byte integers. The *run ID*, *run length*, and *active index* buffers all need to be arrays of 4-byte integers. Therefore, there is an additional memory footprint of 1.75x raw image size for storing the intermediate results.

However, in our implementation, Thrust enables data transformation on-the-fly during their inclusive/exclusive scan and reduce-by-key algorithms. Therefore determining whether or not a pixel is active or starts a new run does not need to be precomputed. This reduces the memory footprint to 1.5x raw image size. Open source code for parallel image compression using Thrust is available at `https://github.com/tmarrinan/pari-compression`.

Algorithm 1 Parallel Active Pixel Encoding

```
Input: $width, height, color, depth$
Output: $ap\_image, ap\_size$
 1: $new\_run, is\_active \leftarrow$ PIXELPROPERTIES($width$,
        $height$, $depth$)
 2: $run\_id \leftarrow$ INCLUSIVESCAN($new\_run$)
 3: $run\_len \leftarrow$ REDUCEBYKEY($run\_id$, 1)
 4: $active\_idx \leftarrow$ EXCLUSIVESCAN($is\_active$)
 5: $ap\_image, ap\_size \leftarrow$ COMPRESSIMAGE($width$,
        $height$, $color$, $depth$, $is\_active$, $new\_run$,
        $run\_id$, $run\_len$, $active\_idx$)

 1: procedure PIXELPROPERTIES
    input: $width, height, depth$
    output: $is\_active, new\_run$
 2:     $is\_active[0] \leftarrow (depth[0] \neq max\_depth)$
 3:     $new\_run[0] \leftarrow 1$
 4:     parfor $i \leftarrow 1, (width * height)$ do
 5:         $is\_active[i] \leftarrow (depth[i] \neq max\_depth)$
 6:         $p\_active \leftarrow (depth[i-1] \neq max\_depth)$
 7:         $new\_run[i] \leftarrow (is\_active[i] \neq p\_active)$
 8:     end parfor

 1: procedure COMPRESSIMAGE
    input: $width, height, color, depth, is\_active, new\_run$,
        $run\_id, run\_len, active\_idx$
    output: $ap\_image, ap\_size$
 2:     parfor $i \leftarrow 0, (width * height)$ do
 3:         if $is\_active[i]$ then
 4:             $pos \leftarrow \mathrm{ApOffset}(active\_idx[i], run\_id[i])$
 5:             $\mathrm{Memcpy}(ap\_image + pos, color[i], 4)$
 6:             $\mathrm{Memcpy}(ap\_image + pos + 4, depth[i], 4)$
 7:             if $new\_run[i] = 1$ then
 8:                 $inactive \leftarrow 0$
 9:                 if $run\_id[i] > 1$ then
10:                     $inactive \leftarrow run\_len[run\_id[i] - 2]$
11:                 $active \leftarrow run\_len[run\_id[i] - 1]$
12:                 $\mathrm{Memcpy}(ap\_image + pos - 8, inactive, 4)$
13:                 $\mathrm{Memcpy}(ap\_image + pos - 4, active, 4)$
14:     end parfor
15:     $i \leftarrow (width * height) - 1$
16:     $active\_run \leftarrow run\_id[i] + is\_active[i] - 1$
17:     $pos \leftarrow \mathrm{ApOffset}(active\_idx[i], active\_run)$
18:     $ap\_size \leftarrow pos + 8$
19:     if not $is\_active[i]$ then
20:         $inactive \leftarrow run\_len[run\_id[i] - 1]$
21:         $active \leftarrow 0$
22:         $\mathrm{Memcpy}(ap\_image + pos, inactive, 4)$
23:         $\mathrm{Memcpy}(ap\_image + pos + 4, active, 4)$
```

4 ICET INTEGRATION

Prior to our work, IceT provided an interface for compositing partial images rendered by legacy OpenGL applications. IceT also provided a generic compositing interface that was rendering framework agnostic. Since IceT's OpenGL interface only supported legacy applications, those using modern OpenGL (OpenGL 3.0+ using GLSL shaders) were relegated to using the more complex generic compositing interface.

In order to perform compression on the GPU, images need to be rendered to a texture rather than onscreen. Therefore, we have updated IceT in two ways: 1) creating a modern OpenGL interface that supports a programmable rendering pipeline, outputting both color and depth to textures on the GPU, and 2) using Thrust to compress image data stored in OpenGL textures on the GPU prior to device-to-host memory transfer.

4.1 IceT for Modern OpenGL Applications

The first release of IceT occurred in 2001 when OpenGL was primarily a fixed-function pipeline [22]. Consequently, the initial design assumed rendering would occur on a GPU and the compositing work would happen on the CPU. This meant that before compositing could start, pixels needed to be transferred from the GPU to main memory. IceT works to minimize the region of pixels transferred, but raw pixels are transferred nonetheless.

A generic interface for IceT that does not rely on OpenGL (or any other rendering system) has long existed. However, in this case as well, image data was transferred to IceT through memory buffers on the CPU. Thus, the generic interface did not get around the issue of transferring raw pixels.

Since IceT's inception, graphics hardware has changed dramatically. The ability to compile software to run directly on the GPU makes it possible to move some of the IceT functionality there. To make this possible, we first update IceT to use a more modern version of OpenGL. The reason for this is twofold. First, OpenGL 1.1 provides no convenient mechanism to access framebuffers in GPU programming environments like CUDA. Second, it is unreasonable to expect applications to be using such an outdated version of OpenGL. Our new rendering interface layer is built on top of OpenGL 3.0, which is the first version to contain the GLSL capabilities needed for our implementation but is new enough to be forward compatible with more recent versions of OpenGL.

Our OpenGL 3 layer to IceT first creates textures to hold the color and depth values of rendered images, and then builds a framebuffer object (FBO) with these textures. The FBO identifier is made available to the distributed rendering application so that it can render its image to this framebuffer. Once the rendering function completes, the textures of the framebuffer can be copied to main memory or used for further image processing on the GPU.

4.2 OpenGL / CUDA Interop

Graphics data can be accessed directly by general purpose GPU code through the use of OpenGL / CUDA interoperability. One complication that occurs though is that the OpenGL depth texture cannot be accessed directly by CUDA because of number format incompatibility (internally stored as 24-bit unsigned integer, but accessed as 32-bit float). Instead, the data has to be first copied to a single component floating-point texture. This is done by implementing an additional rendering pass. A single fullscreen quad is drawn that uses an OpenGL GLSL shader to read pixel values from the original 24-bit unsigned integer depth texture and output pixel values to the 32-bit float texture. Although this essentially adds a memory copy, it is all done on the GPU and therefore occurs very fast.

Once rendered images are stored in the appropriate format, compression can be performed on the GPU. A texture cannot be accessed simultaneously by OpenGL and CUDA. Therefore it is important to unbind the framebuffer with the color and depth textures prior to mapping them in the CUDA code. Similarly it is important to unmap the textures in the CUDA code after performing the compression but prior to transferring control back to the rendering system.

Since IceT supports compositing for tiled displays, the rendered image may need to be compressed into multiple lower resolution

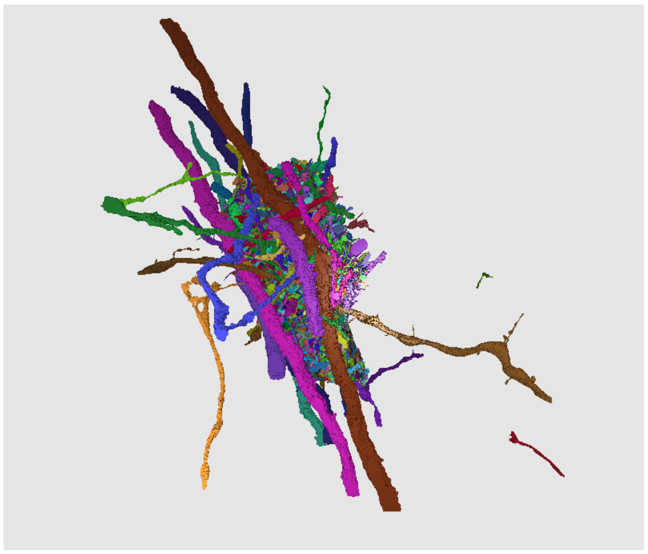

Figure 2: Model of over 1600 neurons reconstructed from an electron microscopy scan of a mouse brain.

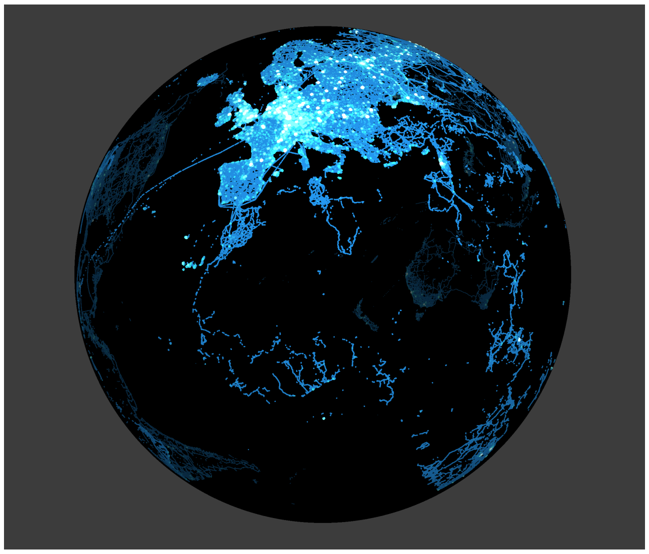

Figure 3: Point cloud of approximately 55.3 million GPS coordinates collected by OpenStreetMap.

images – one per tile. Therefore, we implemented a slight modification to the Active Pixel encoding as described in Sect. 3. The modification takes into account two factors specific to IceT: 1) dimensions of the rendered image and compressed image used for compositing may have different resolutions, and 2) due to region of interest detection, certain areas of the image are ignored (i.e. treated as inactive regardless of pixel depth values). The region of interest in the rendered texture and Active Pixel compressed image will be of the same dimensions, but the location (pixel row and column offset) may be different. Equation 2 shows how to calculate which pixel in the rendered image corresponds to a given pixel in the Active Pixel encoded image. The *render_viewport* is the offset into rendered image for bottom-left corner of where the region of interest is located. The *target_viewport* is the offset into Active Pixel encoded image for bottom-left corner of where the region of interest should be stored.

$$\begin{aligned} render_x &= target_x - target\_viewport_x + render\_viewport_x \\ render_y &= target_y - target\_viewport_y + render\_viewport_y \end{aligned} \quad (2)$$

In addition to performing the compression computation on the GPU, we also use *pinned memory* – memory that cannot be paged out (i.e. will always be located in physical memory). This is a technique that enables higher bandwidth data transfer between a GPU and its host. The combination of direct access to rendered image data, performing Active Pixel encoding using a parallel algorithm, reducing the amount of data needing to be transferred from the GPU to main memory, and using pinned memory for fast data transfer leads to a significant reduction in time for image data compression and memory transfer.

5 Performance Measurements

In order to evaluate the impact that performing Active Pixel encoding on the GPU has on distributed rendering applications, we measured both compute and memory transfer times as well as overall frame rate. We compared three of IceT's compositing methods: 1) generic compositing interface, 2) OpenGL 3 compositing interface, and 3) OpenGL 3 compositing interface with GPU compression. When using the generic compositing interface, the entire rendered framebuffer is provided to IceT and Active Pixel encoding occurs on the CPU. When using the OpenGL 3 compositing interface, region of interest detection is leveraged to reduce raw data memory transfer size, but Active Pixel encoding still occurs on the CPU. When using the OpenGL 3 compositing interface with GPU compression, Active Pixel encoding occurs on the GPU (taking into account the region of interest) and compressed image data is transferred to main memory.

It is also worth noting that rendering with IceT's generic interface can be configured to use region of interest detection. However, we wanted to be able to measure the relative benefit of GPU-based compression compared to that of region of interest detection. Additionally, we feel that the burden of configuring region of interest detection using the generic compositing interface is non-negligible and therefore application developers may not implement it in their distributed rendering applications.

We used three different data sets in our experiments – a simple model of a nuclear power station, reconstructed neurons from an electron microscopy scan of a mouse brain, and latitude/longitude coordinates collected from GPS devices by OpenStreetMap (shown in Figs. 1, 2, and 3 respectively). The nuclear power station is stored as 34 separate OBJ files and contains 9,096 triangles in total. The neuron data set is stored as 1632 separate OBJ files and contains just over 91 million triangles in total. The GPS data set contains approximately 2.77 billion latitude/longitude coordinates, of which 2% were preprocessed for use with our experiments. The resulting ~55.3 million latitude/longitude coordinates were mapped onto a globe in 3D and rendered using an imposter sphere point cloud visualization with both size and color mapped to density.

For each data set and compositing method, we performed strong scaling tests at three different resolutions – HD (1920×1080), 4K (3840×2160), and 8K (7680×4320). For the nuclear power station and neuron data sets, OBJ files were distributed amongst rendering processes. For the GPS data set, points were distributed based on latitude/longitude with an equal number of points per process. The nuclear power station data set was rendered using 2, 4, 8, 16, and 32 processes. The neuron and GPS data sets were rendered using 6, 12, 24, 48, 96, and 192 processes.

The nuclear power station data set intentionally has a low polygon count, which leads to quick render times even with a small number of processes. This results in compositing as the main bottleneck in the distributed rendering pipeline. The neuron and GPS data sets are intentionally larger-scale. When using a small number of processes, data is either too large to fit in GPU memory or leads to non-interactive render times. However, when using a large number of processes, data easily fits in GPU memory and can be rendered quickly. Therefore, as more processes are used, these distributed rendering applications move from having rendering as their bottleneck to having compositing as their bottleneck.

All tests were run on Argonne National Laboratory's Cooley system – a 126-node cluster with an NVIDIA Tesla K80 dual GPU in each node and a 56 Gbps FDR InfiniBand interconnect. Each test used two processes per node (with each using a separate GPU). Visualizations of each data set positioned the models to fill the center of the screen. The models were rotated 360° over 180 frames. Our main goal was to speed up the overall distributed rendering pipeline. Therefore the primary dependent variable we were interested in observing was frame rate, measured in average frames per second (FPS). Secondarily, we also were interested in observing compression compute time and device-to-host data transfer time.

5.1 Frame Rate Comparison

The main goal of our work was to improve distributed rendering time, thus leading to higher frame rates. Results from rendering the nuclear power station, reconstructed neurons, and GPS point cloud data sets are visualized in the left column of Fig. 4. The two compositing techniques using the new OpenGL 3 compositing interface for IceT were able to achieve substantially higher frame rates than the

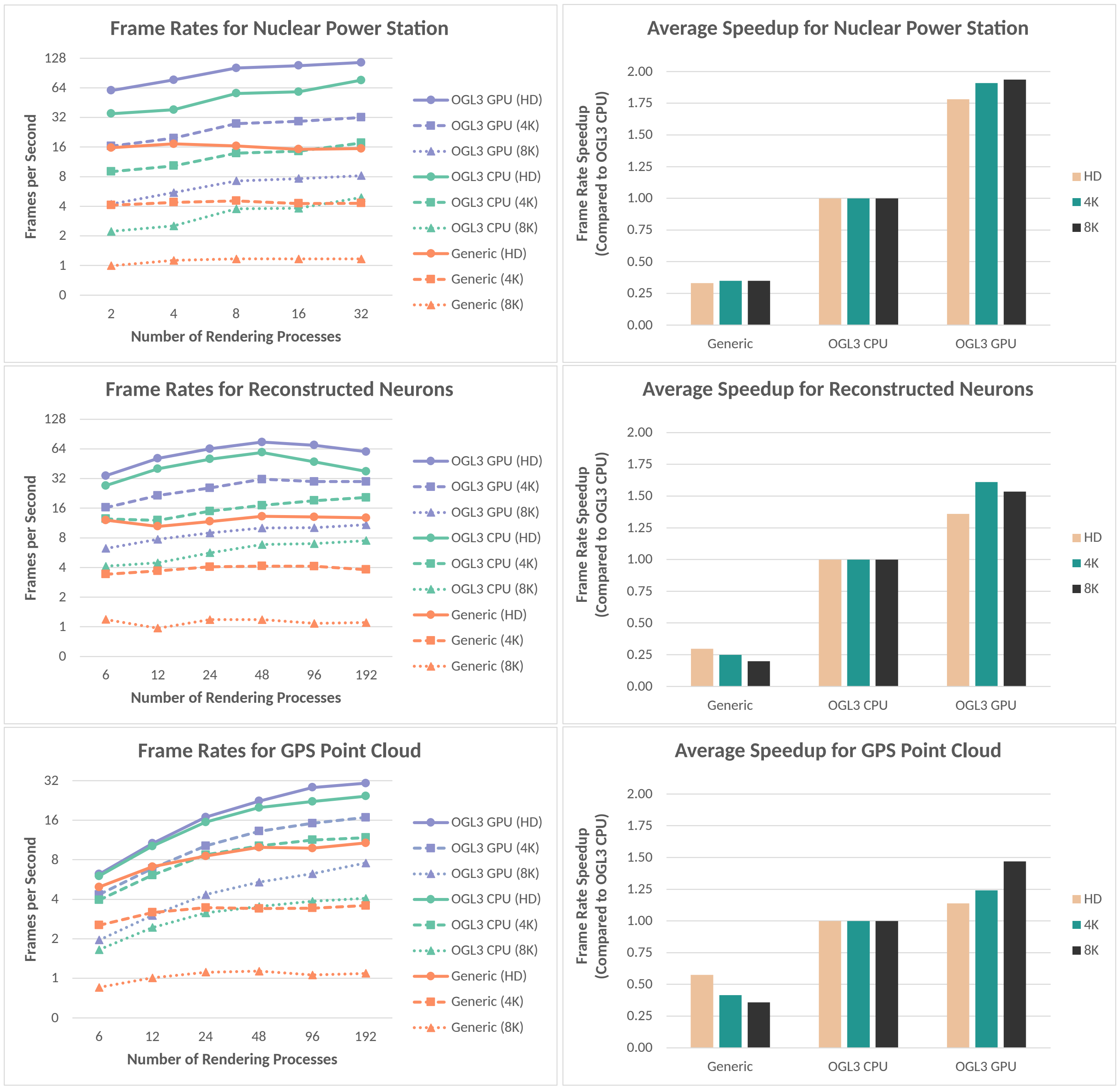

Figure 4: Frame rate results for nuclear power station, reconstructed neurons, and GPS point cloud data sets rendered at HD, 4K, and 8K using the three compositing techniques. Left column shows strong scaling results when measuring overall frames per second. Right column shows relative frame rate speed (with the regular OpenGL 3 compositing interface as the reference).

generic compositing interface, which falls in line with prior experiments showing that region of interest detection can greatly improve compositing efficiency. Additionally, the OpenGL 3 compositing interface with GPU-based compression resulted in achieving even higher frame rates than when performing the compression on the CPU. Results also show that the OpenGL 3 compositing interface with GPU-based compression scales better as the improved frame rates become more pronouced at higher process counts.

We observe that the frame rate for each data set plateaus (or even decreases) at some point. Adding processes benefits the distributed rendering pipeline by reducing the amount of geometry each process is responsible for rendering and shrinking the region of interest each process uses during compositing. However, extra processes also add more communication for the compositing step. As with any strong scaling setup, at some point the benefits of reduced computation are outweighed by increased overheads.

Our three data sets show three different behaviors. The nuclear power station receives almost no boost to rendering performance as the number of processes increase since there is so little geometry to begin with. Distributed rendering frame rates increase nonetheless for both compositing methods that use the OpenGL 3 interface due to the fact that the region of interest for each rendering process is shrinking. The reconstructed neurons also receives little to no boost in rendering performance as the number of processes increase. While this data set is larger, it is already being split into 6 parts at the smallest tested scale. We were however able to split the data amongst a higher number of processes and therefore observe the point when adding more rendering processes became detrimental to the overall performance of the application. The GPS point cloud applications shows nice scaling as the number of rendering processes increase. Even the generic compositing interface (which does not use region of interest detection) sees an increase in frame rate as the application uses more rendering processes. However, this application also sees diminishing returns after a certain point, with frame rates typically only slightly increasing between the largest two runs.

In addition to looking at raw frame rates, we compared average frame rates across all scales to the regular OpenGL 3 compositing interface for each data set / resolution combination. This metric gave us an average distributed rendering speedup achieved by each compositing technique when compared to the prior state-of-the-art of using region of interest detection, but copying raw pixels off the GPU and performing compression on the CPU. Results for the nuclear power station, reconstructed neurons, and GPS point cloud data sets are visualized in the right column of Fig. 4. Beyond the fact the using GPU-based compression led to a speed up of 1.1x-1.9x for the whole distributed rendering process, it is worth noting that the benefits typically became more pronounced as resolution increased. This is likely due to the fact that for our test applications rendering time is primarily impacted by geometry, not resolution. However, compositing time is primarily impacted by resolution, not geometry. Therefore, as image resolution increases, a larger portion of the distributed rendering pipeline is spent in the compositing phase, thus making the GPU-based compression approach more impactful.

5.2 Data Transfer and Compression Time Comparison

In order further investigate the impact that compositing technique has on distributed rendering applications, we compared GPU to main memory image transfer time and computation time for performing the Active Pixel compression. Table 1 shows results for the nuclear power station, reconstructed neurons, and GPS point cloud data sets when run using a small number of rendering processes (2, 6, and 6 respectively) and a large number of rendering processes (32, 192, and 192 respectively).

Results show that region of interest detection can lead to a substantial decrease in both memory transfer time and compression computation time. When comparing the regular OpenGL 3 compositing interface to the generic compositing interface across all data sets and resolutions, the regular OpenGL 3 compositing interface led to a 5.8x-18.8x reduction in memory transfer time at small scale and a 19.1x-109.0x reduction in memory transfer time at large scale. It also led to a 1.6x-4.6x reduction in compression computation time at small scale and a 6.1x-24.3x reduction in compression computation time at large scale.

We note that the gains of using the regular OpenGL 3 compositing interface over the generic compositing interface are more substantial at large scale. This is due to the fact that the regular OpenGL 3 compositing interface uses region of interest detection whereas the generic compositing interface always uses the entire rendered image. As more rendering processes are used, each process is responsible for rendering less geometry, and therefore the region of interest is likely to shrink. The smaller the region of interest, the less data there is to transfer and the fewer pixels there are to compress.

Results also show that performing compression on the GPU can lead to a substantial decrease in both memory transfer time and compression computation time. When comparing the OpenGL 3 compositing interface with GPU-based compression to the regular OpenGL 3 compositing interface across all data sets and resolutions, using GPU-based compression led to a 18.1x-205.6x reduction in memory transfer time at small scale and a 10.2x-688.2x reduction in memory transfer time at large scale. It also led to a 1.9x-6.4x reduction in compression computation time at small scale and a 0.4x-2.7x reduction in compression computation time at large scale.

Table 1: Timing results for performing GPU to main memory image transfer and computation for image compression. Results from smallest scale runs shown in blue. Results from largest scale runs shown in red.

	Nuclear Power Station				**Reconstructed Neurons**				**GPS Point Cloud**			
	Mem. Transfer (milliseconds)		Compression (milliseconds)		Mem. Transfer (milliseconds)		Compression (milliseconds)		Mem. Transfer (milliseconds)		Compression (milliseconds)	
HD (1920 × 1080)												
Generic	39.518	41.536	6.790	5.922	38.228	41.631	6.080	5.920	36.539	43.184	6.172	5.877
OGL3 CPU	6.819	0.488	4.344	0.259	2.535	0.689	1.388	0.280	4.140	2.256	2.584	0.964
OGL3 GPU	0.358	0.048	1.072	0.495	0.066	0.029	0.712	0.642	0.044	0.027	0.689	0.719
4K (3840 × 2160)												
Generic	152.554	158.988	26.992	23.724	156.545	158.900	24.224	23.562	143.465	159.966	24.118	23.466
OGL3 CPU	22.826	1.574	17.289	1.004	8.347	2.080	5.404	1.011	15.602	7.835	9.994	3.811
OGL3 GPU	1.261	0.120	2.768	0.819	0.253	0.042	2.522	1.379	0.097	0.031	1.809	1.717
8K (7680 × 4320)												
Generic	658.877	633.736	108.929	95.005	591.419	632.637	96.580	94.081	574.176	632.394	96.198	93.711
OGL3 CPU	105.717	5.816	69.041	4.048	32.841	7.046	21.055	3.879	61.075	29.594	39.376	15.239
OGL3 GPU	5.764	0.410	10.818	2.114	0.930	0.092	8.853	4.292	0.297	0.043	6.248	5.669

We note that since both techniques leverage region of interest detection, the reduction in memory transfer time can be attributed to the fact that compressed image data is being copied to main memory instead of raw pixel data. Performing compression on the GPU was faster in all but four cases – the large scale runs for the nuclear power station (HD) and the reconstructed neurons (HD, 4K, and 8K). This is likely because when rendering at lower resolution and/or using a high number of rendering processes, the region of interest may become quite small. Therefore, the extra complexity in the parallel implementation of Active Pixel compression ends up outweighing the benefits of parallelism. However, the reduced memory transfer time more than made up this extra computation time, so doing GPU-based compression still led to higher frame rates in these four instances.

5.3 Image Compression Ratios

Both region of interest detection and Active Pixel encoding result in reducing the data size of rendered sub-images. Table 2 shows the relative data sizes (compared to full frame RGBA-depth) produced by each technique for all three tested data sets. Compression ratios are resolution independent since they correspond to the percentage of a frame that geometry or its bounding box projects to. As expected, data size reductions for both region of interest detection and Active Pixel encoding are more pronounced at larger scale since each rendering process is responsible for drawing less geometry. The compression ratio can vary greatly between rendering processes depending on how the geometry is distributed and the selected viewpoint, as demonstrated by comparing the min and max compression ratios for each data set.

While all three data sets tested had good compression ratios (especially when using a larger number of rendering processes), we do note that compression is highly application dependent. In the absolute worst case, geometry would completely fill the rendered viewport on every rendering process, resulting in no inactive pixels. Even in such a case, the result would be a negligible degradation in performance compared to not performing compression at all – the data size would grow by a mere 8 bytes (two integers specifying the length of inactive and active runs) and computation time would only add a maximum of a few milliseconds. This worst case is also extremely unlikely. Even if geometry projects to every pixel in the final image, each process will likely have some empty space once the geometry is distributed.

A more feasible bad case scenario is when the chosen view is zoomed in on a scene. This could result in most visible geometry existing on only a few rendering processes. In such a situation, some rendering processes would compress their sub-image into one containing only inactive pixels while others would compress their sub-image into one containing mostly or fully active pixels. This situation could be avoided by implementing dynamic load balancing in the application to redistribute geometry based on the view.

Table 2: Relative data size for using raw pixels with region of interest (ROI) detection and Active Pixel (AP) encoding. Minimum and maximum data sizes produced by any rendering process are provided. Results from smallest scale runs shown in blue. Results from largest scale runs shown in red.

	Min ROI Ratio	Max ROI Ratio	Min AP Ratio	Max AP Ratio
Nuclear Power Station	30.159% 0.010%	100.000% 42.732%	9.439% 0.009%	21.809% 16.508%
Reconstructed Neurons	3.791% 0.849%	48.933% 26.965%	1.382% 0.040%	5.306% 1.875%
GPS Point Cloud	15.452% 0.308%	75.262% 71.010%	0.185% 0.002%	2.433% 0.468%

6 Discussion

The work presented in this paper set out with the goal of improving just two of the many steps in the distributed rendering pipeline – transferring data from the rendering GPU to main memory and compressing images prior to sending data over the network between rendering processes. While our results provide a substantial improvement in maximum achievable frame rates, compositing still bottlenecks real-time rendering applications due to the transfer of image data between rendering processes. One way of addressing this would be to investigate alternate image compression formats that may yield a greater data reduction than Active Pixel encoding.

We would also like to note that our work has a couple of limitations. First, while the parallel algorithms are generic, our implementation is OpenGL and CUDA specific. This means that GPU-based compression can only be used by applications that use IceT's OpenGL 3 compositing interface and can only be enabled when run on machines that use NVIDIA GPUs. Additionally, the current implementation only supports framebuffers that use unsigned byte RGBA color components and a floating point depth component. While this represents the standard configuration for most rendering applications, IceT in general does support other color and depth formats. The second limitation is the fact that to perform compression on the GPU, additional GPU memory is required. GPU-based compression requires a copy of the depth texture, buffers for storing intermediate data when compressing the image, and a buffer for writing the final Active Pixel compressed image into. While this additional space is not overly large, it may hinder applications that require large amounts of GPU memory for storing application data.

Lastly, we also would like to mention that the impact that GPU-based compression has on overall frame rate is greatly impacted by rendering performance. Performing compression on the GPU typically saved a few milliseconds per frame. For applications that have short render times, a few millisecond difference can have a large impact on frame rate. But for applications with longer render time, the overall impact may be less noticeable. For example, if a distributed rendering application can produce a frame every 25 ms, it would result in a frame rate of 40.0 FPS. By reducing the frame time by 10 ms down to 15 ms, the frame rate would now jump to 66.7 FPS (67% faster). However, if a distributed rendering application takes 250 ms to produce a frame, it would result in a frame rate of 4.0 FPS. Reducing this frame time by 10 ms down to 240 ms, the frame rate would barely increase to 4.2 FPS (4% faster).

7 Conclusion

Our research set out with two main goals – develop a parallel algorithm for performing Active Pixel encoding on the GPU and integrate GPU-based image compression into the distributed rendering pipeline to improve overall frame rates. We have presented our parallel Active Pixel encoding algorithm as well as developed an implementation of that algorithm using CUDA's Thrust library. We also integrated GPU-based image compression into the IceT compositing library. This integration included developing a modern compositing interface for OpenGL applications, which should make building distributed rendering applications with IceT more developer-friendly. All of our work has been integrated into the official IceT repository (`https://gitlab.kitware.com/icet/icet/`). The IceT OpenGL 3 compositing interface can be configured with or without GPU-based compression so that it can be leveraged whether hardware and application constraints are met or not.

We also conducted a series of performance evaluations in order to determine the impact that GPU-based image compression has on the overall distributed rendering pipeline. Our results confirm earlier work done showing the benefits of region of interest detection. They also show that GPU-based compression can lead to further substantial improvements to achievable frame rates. While performing Active Pixel encoding in parallel typically was faster than the serial implementation, the main gain was due to reducing the amount of data being transferred from the GPU to main memory.

In the future, we would like to investigate whether leveraging the GPU for other compositing steps (e.g. image decompression, pixel blending, etc.) would make sense. We are also interested in investigating other compression schemes, including both lossless (e.g. Huffman coding) and lossy (e.g. DXT1) formats that would help reduce that data communicated between rendering processes. Finally, we are interested in creating another compositing interface for Vulkan-based rendering. This would enable applications to be developed using a graphics protocol aimed at high performance rendering to use IceT for compositing without falling back to the generic interface. It would also allow us to implement GPU-based image compression using Vulkan compute shaders, which would remove some of the hardware limitations with the current CUDA implementation.

Acknowledgments

We would like to thank Mykhailo Ohorodnichuk and TurboSquid for creating and providing the "3 D Nuclear station Low-poly" model, Narayanan Kasthuri (Argonne National Laboratory and the University of Chicago) for providing the data for the reconstructed neuron models, and OpenStreetMap for collecting GPS coordinate data and making it publicly available. This research was supported in part by the Argonne Leadership Computing Facility and the Oak Ridge Leadership Computing Facility, which are U.S. Department of Energy Office of Science User Facilities operated under contracts DE-AC02-06CH11357 and DE-AC05-00OR22725 respectively.

References

[1] J. Ahrens and J. Painter. Efficient sort-last rendering using compression-based image compositing. In *Second Eurographics Workshop on Parallel Graphics and Visualization*, September 1998.

[2] I. Castaño. High quality DXT compression using CUDA. Technical report, NVIDIA, 2007.

[3] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. H. Weber, and E. W. Bethel. Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications*, 30(3):22–31, May/June 2010. doi: 10.1109/MCG.2010.51

[4] K.-U. Doerr and F. Kuester. CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):320–332, 2011. doi: 10.1109/TVCG.2010.59

[5] Y. Dong and C. Peng. Screen partitioning load balancing for parallel rendering on a multi-GPU multi-display workstation. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2019. doi: 10.2312/pgv.20191111

[6] S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization & Computer Graphics*, 15(03):436–452, may 2009. doi: 10.1109/TVCG.2008.104

[7] S. Eilemann, D. Steiner, and R. Pajarola. Equalizer 2.0–convergence of a parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 26(2):1292–1307, 2020. doi: 10.1109/TVCG.2018.2870822

[8] L. Emerson and T. Marrinan. Real-time compression of dynamically generated images for offscreen rendering. In *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 91–92, 2019. doi: 10.1109/LDAV48142.2019.8944381

[9] A. V. P. Grosset, M. Prasad, C. Christensen, A. Knoll, and C. Hansen. TOD-Tree: Task-overlapped direct send tree image compositing for hybrid mpi parallelism and gpus. *IEEE Transactions on Visualization and Computer Graphics*, 23(6):1677–1690, June 2017. doi: 10.1109/TVCG.2016.2542069

[10] M. Hereld, I. R. Judson, and R. L. Stevens. Introduction to building projection-based tiled display systems. *IEEE Computer Graphics and Applications*, 20(4):22–28, July 2000. doi: 10.1109/38.851746

[11] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi: 10.1109/JRPROC.1952.273898

[12] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, p. 129–140. Association for Computing Machinery, New York, NY, USA, 2001. doi: 10.1145/383259.383272

[13] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, p. 693–702. Association for Computing Machinery, New York, NY, USA, 2002. doi: 10.1145/566570.566639

[14] M. Larsen, K. Moreland, C. Johnson, and H. Childs. Optimizing multi-image sort-last parallel rendering. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, Oct. 2016. doi: 10.1109/LDAV.2016.7874308

[15] C.-F. Lin, S.-K. Liao, Y.-C. Chung, and D.-L. Yang. A rotate-tiling image compositing method for sort-last parallel volume rendering systems on distributed memory multicomputers. *Journal of Information Science and Engineering*, 20(4):643–664, 2004.

[16] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, July/August 1994. doi: 10.1109/38.291532

[17] M. Makhinya, S. Eilemann, and R. Pajarola. Fast compositing for cluster-parallel rendering. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV '10, p. 111–120. Eurographics Association, Goslar, DEU, 2010.

[18] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.

[19] K. Moreland. IceT users' guide and reference: Version 2.1. Technical report, Sandia National Laboratories, 2011.

[20] K. Moreland. Comparing binary-swap algorithms for odd factors of processes. In *Proceedings of the 8th IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, Oct. 2018. doi: 10.1109/LDAV.2018.8739210

[21] K. Moreland, W. Kendall, T. Peterka, and J. Huang. An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. Association for Computing Machinery, New York, NY, USA, 2011. doi: 10.1145/2063384.2063417

[22] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering on tile displays. In *Proceedings of IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 85–92, October 2001.

[23] B. Neal, P. Hunkin, and A. McGregor. Distributed OpenGL rendering in network bandwidth constrained environments. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2011. doi: 10.2312/EGPGV/EGPGV11/021-029

[24] J. Nonaka, K. Ono, and M. Fujita. 234Compositor: A flexible parallel image compositing framework for massively parallel visualization environments. *Future Generation Computer Systems*, 82:647–655, 2018. doi: 10.1016/j.future.2017.02.011

[25] ParaView. `https://www.paraview.org/`. Accessed: 2021-04-22.

[26] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, November 2009. doi: 10.1145/1654059.1654064

[27] N. R. Revanth and P. J. Narayanan. Distributed massive model rendering. In *Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing*, ICVGIP '12. Association for Computing Machinery, New York, NY, USA, 2012. doi: 10.1145/2425333.2425375

[28] A. Robinson and C. Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967. doi: 10.1109/PROC.1967.5493

[29] R. Rutter. Run-length encoding on graphics hardware. Master's thesis,

University of Alaska at Fairbanks, 2011.

[30] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load balancing for multi-projector rendering systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS '99)*, pp. 107–116, 1999. doi: 10.1145/311534.311584

[31] L. Stähli, D. Rudi, and M. Raubal. Turbulence ahead - a 3D web-based aviation weather visualizer. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, p. 299–311, 2018. doi: 10.1145/3242587.3242624

[32] A. Takeuchi, F. Ino, and K. Hagihara. An improvement on binary-swap compositing for sort-last parallel rendering. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pp. 996–1002, 2003. doi: 10.1145/952532.952728

[33] R. M. Taylor. Practical scientific visualization examples. *SIGGRAPH Comput. Graph.*, 34(1):74–79, Feb. 2000. doi: 10.1145/563788.604456

[34] Thrust. `https://docs.nvidia.com/cuda/thrust/index.html`. Accessed: 2021-04-22.

[35] VisIt. `https://wci.llnl.gov/simulation/computer-codes/visit`. Accessed: 2021-04-22.

[36] Vulkan. `https://www.vulkan.org/`. Accessed: 2021-04-22.

[37] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable rendering on PC clusters. *IEEE Computer Graphics and Applications*, 21(4):62–70, July/August 2001.

[38] N. Yamamoto, K. Nakano, Y. Ito, D. Takafuji, A. Kasagi, and T. Tabaru. Huffman coding with gap arrays for gpu acceleration. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3404397.3404429

[39] D.-L. Yang, J.-C. Yu, and Y.-C. Chung. Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. In *1999 International Conference on Parallel Processing*, pp. 200–207, 1999. doi: 10.1109/ICPP.1999.797405

[40] H. Yu, C. Wang, and K.-L. Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008. doi: 10.1109/SC.2008.5219060