Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs and T. Kuhlen (Editors)

Parallel Rendering on Hybrid Multi-GPU Clusters

S. Eilemann[†1], A. Bilgili[1], M. Abdellah[1], J. Hernando[2], M. Makhinya[3], R. Pajarola[3], F. Schürmann[1]

[1]Blue Brain Project, EPFL; [2]CeSViMa, UPM; [3]Visualization and MultiMedia Lab, University of Zürich

Abstract

Achieving efficient scalable parallel rendering for interactive visualization applications on medium-sized graphics clusters remains a challenging problem. Framerates of up to 60hz require a carefully designed and fine-tuned parallel rendering implementation that fits all required operations into the 16ms time budget available for each rendered frame. Furthermore, modern commodity hardware embraces more and more a NUMA architecture, where multiple processor sockets each have their locally attached memory and where auxiliary devices such as GPUs and network interfaces are directly attached to one of the processors. Such so called fat *NUMA processing and graphics nodes are increasingly used to build cost-effective* hybrid *shared/distributed memory visualization clusters. In this paper we present a thorough analysis of the asynchronous parallelization of the rendering stages and we derive and implement important optimizations to achieve highly interactive framerates on such hybrid multi-GPU clusters. We use both a benchmark program and a real-world scientific application used to visualize, navigate and interact with simulations of cortical neuron circuit models.*

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—Distributed Graphics; I.3.m [Computer Graphics]: Miscellaneous—Parallel Rendering

1. Introduction

The decomposition of parallel rendering systems across multiple resources can be generalized and classified, according to Molnar et al. [MCEF94], based on the sorting stage of the image synthesis pipeline: Sort-first (2D) decomposition divides a single frame in the image domain and assigns the resulting image tiles to different processes; Sort-last database (DB) decomposition performs a data domain decomposition across the participating rendering processes. A sort-middle approach cannot typically be implemented efficiently with current hardware architectures, as one needs to intercept the transformed and projected geometry in scan-space after primitive assembly. In addition to this classification, frame-based approaches distribute entire frames, i.e. by time-multiplexing (DPlex) or stereo decomposition (Eye), to different rendering resources. Variations of sort-first rendering are pixel or subpixel decomposition as well as tile-based decomposition frequently used for interactive raytracing.

Hybrid GPU clusters are often build from so called *fat* render nodes using a NUMA architecture with multiple CPUs and GPUs per machine. This configuration poses unique challenges to achieve optimal distributed parallel rendering performance since the topology of both the individual single node as well as the full cluster has to be taken into account to optimally exploit locality in parallel rendering. In particular, the configuration of using multiple rendering source nodes feeding one or a few display destination nodes for scalable rendering offers room for improvement in the rendering stages management and synchronization.

In this paper we study in detail different performance limiting factors for parallel rendering on hybrid GPU clusters, such as thread placements, asynchronous compositing, network transport and configuration variation in both a straightforward benchmark tool as well as a real-world scientific visualization application.

The contributions presented in this paper not only consist of an experimental analysis on a medium-sized visualization cluster but also introduce and evaluate important optimizations to improve the scalability of interactive applications for large parallel data visualization. Medium-sized visualization clusters are currently among the most practically relevant configurations as these are cost efficient, widely avail-

† stefan.eilemann@epfl.ch, ahmet.bilgili@epfl.ch, marwan.abdellah@epfl.ch, jhernando@fi.upm.es, makhinya@ifi.uzh.ch, pajarola@acm.org, felix.schuermann@epfl.ch

able and reasonably easy to manage. Most notable, we observed that asynchronous readbacks and regions of interest can contribute substantially towards high-performance interactive visualization.

2. Related Work

A number of general purpose parallel rendering concepts and optimizations have been introduced before, such as parallel rendering architectures, parallel compositing, load balancing, data distribution, or scalability. However, only a few generic APIs and parallel rendering systems exist which include VR Juggler [BJH*01] (and its derivatives), Chromium [HHN*02], OpenSG [VBRR02], OpenGL Multipipe SDK [BRE05], Equalizer [EMP09] and CGLX [DK11] of which Equalizer is used as the basis for the work presented in this paper.

For sort-last rendering, a number of parallel compositing algorithm improvements have been proposed in [MPHK94, LRN96, SML*03, EP07, PGR*09, MEP10]. In this work, however, we focus on improving the rendering performance by optimizations and data reduction techniques applicable to the sort-first and the parallel direct-send sort-last compositing methods. For sort-first parallel rendering, the total image data exchange load is fairly simple and approaches $O(1)$ for larger N. Sort-last direct-send compositing exchanges exactly two full images concurrently in total between the N rendering nodes. Message contention in massive parallel rendering [YWM08] is not considered as it is not a major bottleneck in medium-sized clusters.

To reduce transmission cost of pixel data, image compression [AP98, YYC01, TIH03, SKN04] and screen-space bounding rectangles [MPHK94, LRN96, YYC01] have been proposed. Benchmarks done by [SMV11] show that exploiting the topology of Hybrid GPU clusters with NUMA architecture increases the performance of CUDA applications.

3. Scalable Rendering on Hybrid Architectures

3.1. Parallel Rendering Framework

We chose Equalizer for our parallel rendering framework, due its scalability and configuration flexibility. The architecture and design decisions of Equalizer are described in [EMP09] and the results described in this paper build upon that foundation.

However, the basic principles of parallel rendering are similar for most approaches, and the analysis, experiments and improvements presented in this paper are generally applicable and thus also useful for other parallel rendering systems.

In a cluster-parallel distributed rendering environment the general execution flow is as follows (omitting event handling and other application tasks): *clear, draw, readback, transmit* and *composite*. Clear and draw optimizations are largely ignored in this work. Consequently, in the following we focus on the readback, transmission, compositing and load-balancing stages.

3.2. Hybrid GPU Clusters

Hybrid GPU clusters use a set of fat rendering nodes connected with each other using one or more interconnects. In contrast with traditional single-socket, single-GPU cluster nodes, each node in a hybrid cluster has an internal topology, using a non-uniform memory access (NUMA) architecture and PCI Express bus layout with multiple memory regions, GPUs and network adapters. Figure 1 shows a typical layout, depicting the relative bandwidth through the thickness of the connections between components.

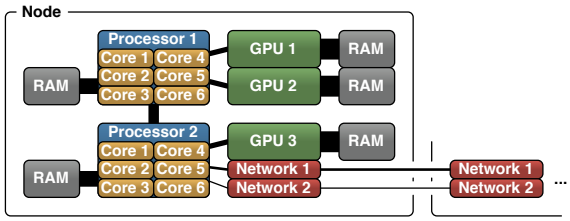


Figure 1: *Example hybrid GPU cluster topology.*

Future hardware architectures are announced which will make this topology more complex. The interconnect topology for most visualization clusters uses typically a fully non-blocking, switched backplane.

3.3. Asynchronous Compositing

Compositing in a distributed parallel rendering system is decomposed into readback of the produced pixel data (1), optional compression of this pixel data (2), transmission to the destination node consisting of send (3) and receive (4), optional decompression (5) and composition consisting of upload (6) and assembly (7) in the destination framebuffer.

In a naive implementation, operations 1 to 3 are executed serially on one processor, and 4 to 7 on another. In our parallel rendering system, operations 2 to 5 are executed asynchronously to the rendering and operations 1, 6 and 7. Furthermore, we use a latency of one frame which means that two rendering frames are always in execution, allowing the pipelining of these operations, as shown in Figures 2(a) and 2(c). Furthermore we have implemented asynchronous readback using OpenGL pixel buffer objects, further increasing the parallelism by pipelining the rendering and pixel transfers, as shown in Figure 2(b). We did not implement asynchronous uploads as shown in Figure 2(d), since it has a minimal impact when using a many-to-one configuration but complicates the actual implementation significantly.

In the asynchronous case, the rendering thread performs only application-specific rendering operations, since the overhead of starting an asynchronous readback becomes negligible.

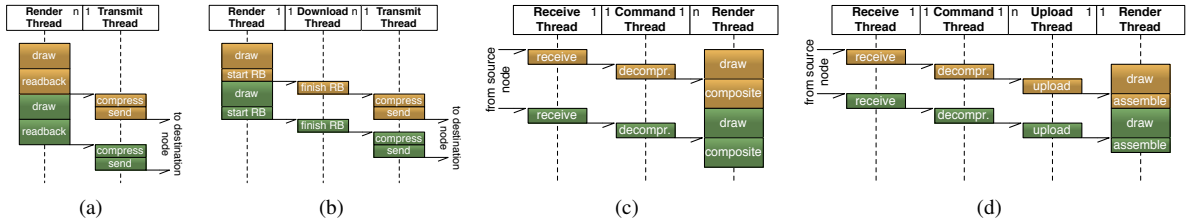


Figure 2: *Synchronous (a) and asynchronous (b) readback as well as synchronous(c) and asynchronous (d) upload for two overlapping frames.*

Equalizer uses a plugin system to implement GPU-CPU transfer modules which are runtime loadable. We extended this plugin API to allow the creation of asynchronous transfer plugins, and implemented such a plugin using OpenGL pixel buffer objects (PBO). At runtime, one rendering thread and one download thread is used for each GPU, as well as one transmit thread per process. The download threads are created lazy, when needed.

3.4. Automatic Thread and Memory Placement

In a parallel rendering system, a number of threads are used to drive a single process in the cluster. In our software we use one main thread (main), one rendering thread for each GPU (draw) and one to finish the download operation during asynchronous downloads (read), one thread for receiving network data (recv), one auxilary command processing thread (cmd) and one thread for image transmission to other nodes (xmit).

Thread placement is critical to achieve optimal performance. Due to internals of the driver implementations and the hardware topology, running data transfers from a 'remote' processor in the system has severe performance penalties. We have implemented automatic thread placement by extending and using the hwloc [MPI12] library in Equalizer using the following rules:

- Restrict all node threads (main, receive, command, image transmission) to the cores of the processor local to the network card.
- Restrict each GPU rendering and download thread to the cores of the processor close to the respective GPU.

Figure 3 shows the thread placement for the topology example used in Figure 1. The threads are placed to all cores of the respective processor, and the ratio of cores to threads varies with the used hardware and software configuration. Furthermore, many of the threads do not occupy a full core at runtime, especially the node threads which are mostly idle.

The memory placement follows the thread placement when using the default 'first-touch' allocation scheme, since all GPU-specific memory allocations are done by the render threads.

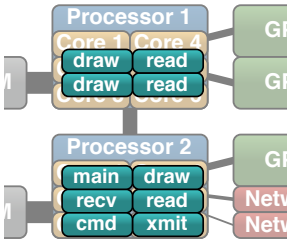


Figure 3: *Mapping threads within a* fat *cluster node.*

3.5. Region of Interest

The region of interest (ROI) is the screen-space 2D bounding box enclosing the geometry rendered by a single resource. We have extended the core parallel rendering framework to transparently use an application-provided ROI to optimize load balancing and image compositing.

Equalizer uses an automatic load-balancing algorithm similar to [ACCC04] and [SZF*99] for 2D decompositions. The algorithm is based on a load grid constructed from the tiles and timing statistics of the last finished frame. The per-tile load is computed from the draw and readback time, or compress and transmit, whichever is higher. The load-balancer integrates over this load grid to predict the optimal tile layout for the next frame. In Equalizer, the screen is decomposed into a kd-tree where in [ACCC04] a dual-level tree is used, first decomposed into horizontal tiles and then each horizontal tile into vertical sub-tiles.

We use the ROI of each rendering source node to automatically refine the load grid, see also Figure 4. In cases where the rendered data projects only to a limited screen area, this ROI refinement provides the load-balancer with a more accurate load indicator leading to a better load prediction.

The load-balancer has to operate on the assumption that the load is uniform within one load grid tile. This leads naturally to estimation errors, since in reality the load is not uniformly distributed, e.g., it tends to increase towards the center of the screen in Figure 4. Refining the load grid tile

size using ROI decreases this inherent error and makes the load prediction more accurate.

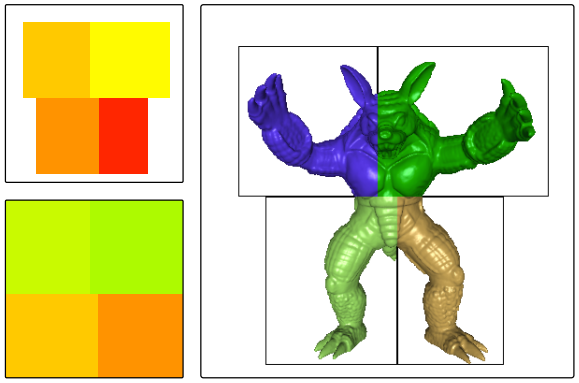


Figure 4: *Load distribution areas with (top) and without (bottom) using the ROI of the right-hand 2D parallel rendering.*

Furthermore, the ROI can also be used to minimize pixel transport for compositing [MEP10]. This is particularly important for spatially compact DB decompositions, since the per-resource ROI becomes more compact with the number of used resources.

3.6. Multi-threaded and Multi-process Rendering

Equalizer [EMP09] supports a flexible configuration system, allowing an application to be used with one single process per GPU (referred to as multi-process), with one process per node and one rendering thread for each GPU (referred to as multi-threaded). This flexibility and the comparison of the two configurations allows us to observe the impact of memory bandwidth bottlenecks in a multi-threaded application versus increased memory usage in a multi-process application, as explained in Section 5.5. Furthermore, a multi-process configuration is representative of typical MPI-based parallel rendering systems, which are relatively common.

4. Visualization of Cortical Circuit Simulations

4.1. Neuroscience Background

RTNeuron [HSMd08] is a scientific application used in the Blue Brain Project [Mar06] to visualize, navigate and interact with cortical neuron circuit models and corresponding simulation results, see also Figure 5. These detailed circuit models accurately model the structure and function of a portion of a young rat's cortex and the activity is being mapped back to the structure [LHS*12]. The cells are simulated using cable models [Ral62], which divide neuron branches in electrical compartments. The cell membrane of these compartments is modeled using a set of Hodgkin and Huxley equations [HHS*11]. Detailed and statistically varying shapes of the neurons are used to layout a cortical circuit and establish synapses, the connections between neurons. Eventually, the simulation is run in a supercomputer and different variables, such as membrane voltage and currents, spike times etc. are reported.

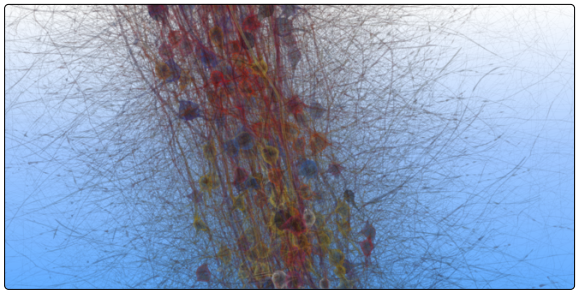


Figure 5: *RTNeuron example rendering of the 1,000 innermost cells of a cortical circuit.*

The outcome of these simulations is analyzed in several ways, including direct visualization of the simulator output. Interactive rendering of cortical circuits presents several technical challenges, the geometric complexity being the most relevant for this study. A typical morphological shape representing a neuron consists, on average consisting of more than 4,200 branch segments, translates to a mesh of 140,000 polygons. A typical circuit consist of 10,000 neurons chosen from hundreds of unique morphologies which leads hundreds of million geometric elements to be rendered in each frame.

4.2. RTNeuron Implementation

There are two geometric features of a cortical circuit that make its visualization difficult. Neurons have a large spatial extent but exhibit a very low space occupancy and they are spatially interleaved in a complex manner, making it practically impossible to visually track a single cell. This also causes aliasing problems and makes visibility and occlusion culling hard.

To address the raw geometric complexity, level-of-detail (LOD) techniques can be used [LRC*03]. These techniques are applied on a single workstation scenario, however for the purpose of this paper we have focused on the higher quality and scalability constraints of fully detailed meshes.

For scalability, we exploit parallel rendering using 2D (sort-first) and DB (sort-last) task decomposition, as well as combinations of both. Time-multiplexing can increase the throughput but does not improve the interaction latency, and pixel decomposition does not improve scalability for this type of rendering. Subpixel decomposition for anti-aliasing does increase visual quality but not rendering performance.

For 2D decomposition, the application is required to implement efficient view frustum culling to allow scalability with smaller 2D viewports. For this, our application uses the culling techniques presented in [Her11]. The view frustum culling algorithm used is based on a skeleton of capsules

(spherically-capped cylinders) extracted from the neuronal skeletons. The frustum-capsule intersection test that determines the visibility is performed on the GPU. Then, for each neuron the visibility of the capsules is used to compute which intervals of its polygon list need to be rendered. This culling operation is relatively expensive for large circuits, and its result is cached if the projection-modelview matrix is unchanged from the previous frame. This is typically the case during simulation data playback, as the user observes the time series from a fixed viewpoint. The simulation data, typically voltage information, is applied using a color map on the geometry.

To address both memory constraints and rendering scalability for large data, DB decomposition is the most promising approach. We have implemented and compared two different decomposition schemes, a round-robin division of neurons between rendering resources and a spatially ordered data division using a kd-tree.

The round-robin decomposition is easy to implement and leads to a fairly even geometry workload for large neuron circuits. However, it fails to guarantee a total depth-order between the source frames to be assembled. This requires the usage of the depth buffer for compositing, and precludes the use of transparency due to the entanglement of the neurons.

The spatial partition DB decomposition approach is based on a balanced kd-tree constructed from the end points of the segments that describe the shape of the neurons (the morphological skeletons). For a given static data set and fixed number of rendering source nodes, a balanced kd-tree is constructed over the segment endpoints such that the number of leaves matches the number of rendering nodes. The split position in each node is chosen such that the points are distributed proportionally to the rendering nodes assigned to the resulting regions.

The kd-tree leaf nodes define the spatial data region that each rendering node has to load and process. Each leaf occupies a rectangular region in space, and clip planes restrict fragments to be generated for the exact spatial region. Therefore, this DB decomposition can be composited using only the *RGBA* frame buffer, and solves any issues with alpha-blending since a coherent compositing order can be established for any viewpoint.

Both DB decompositions modes are difficult to load balance at run-time. In particular, dynamic spatial repartitioning and on-demand loading have been omitted because of their implementation challenges. Furthermore, the readback, transmission and compositing costs are higher for DB than for 2D decomposition.

A possible way to improve load balancing in a static DB decomposition is to combine it with a nested 2D decomposition. This way, a group of nodes which is assigned the same static data partition can use a dynamic 2D decomposition to balance the rendering work locally.

5. Experimental Analysis

Any parallel rendering system is fundamentally limited by two factors: the rendering itself, which includes transformation, shading and illumination; as well as compositing multiple partial rendering results into a final display, i.e. image. While an exceeding task load of the former is the major cause for parallelization in the first place, the latter is often a limiting bottleneck due to constrained image data throughput or synchronization overhead.

In our experiments we investigate the image transmission and compositing throughput of sort-first (2D) and sort-last (DB) parallel rendering. For DB we study direct-send (DS) compositing, which has shown to be highly scalable for medium-sized clusters [EP07]. For benchmarking we used two applications, *eqPly* and *RTNeuron*. The first is a textbook parallel triangle mesh rendering application allowing to observe and establish a baseline for the various experiments, while the second is a real-world application used by scientists to visualize large-scale HPC simulation results.

All tests were carried out on a 11 node cluster with the following technical node specifications: dual six-core 3.47GHz processors (Intel Xeon X5690), 24GB of RAM, three NVidia GeForce GTX580 GPUs and a Full-HD passive stereo display connected to two GPUs on the head node; 1Gbit/s, 10Gbit/s ethernet and 40Gbit/s QDR InfiniBand. The GPUs are attached each using a dedicated 16x PCIe 2.0 link, the InfiniBand on a dedicated 8x PCIe 2.0 link and the 10 Gbit/s Ethernet using a 4x PCIe 2.0 link.

Using *netperf*, we evaluated a realistic achievable data transmission rate of $n = 1100MB/s$ for the 10 GBit interconnect. Using *pipeperf*, we evaluated the real-world GPU-CPU transfer rates of $p = 1000MB/s$ for synchronous and asynchronous transfers. Note that in the case of asynchronous transfers almost all of the time is spent in the finish operation, which is pipelined with the rendering thread. Thus, unless the rendering time is smaller than the combined readback and transmission time, the readback has no cost for the overall performance. Thus for a frame buffer size s_{FB} of 1920×1080 ($6MB$) we expect up to 160 fps during rendering.

All 2D tests where conducted using a load balancer. All DB tests used an even static distribution of the data across the resources. For the eqPly tests, we rendered four times the David 1 mm model consisting of 56 million triangles with a camera path using 400 frames, as shown in Figure 6. For the RTNeuron tests we used a full neocortical column using 10,000 neurons (around 1.4 billion triangles).

5.1. Thread Placement

We tested the influence of thread placement by explicitly placing the threads either on the correct or incorrect processor. We found that this leads to a performance improvement of more than 6% in real-world rendering loads, as shown in Figure 7. Thread placement mostly affects the GPU-CPU

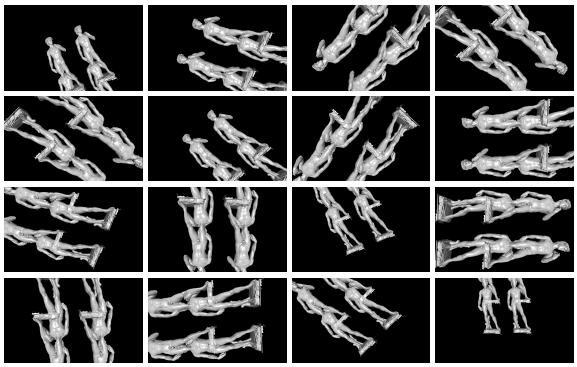


Figure 6: *Screenshots along the eqPly camera path.*

download performance, and therefore has a small influence on the overall performance, with a growing importance at high framerates as the draw time decreases.

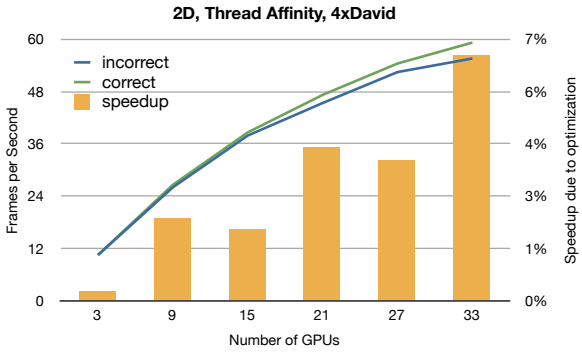


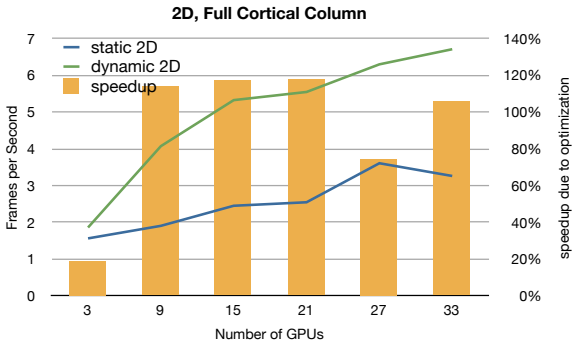
Figure 7: *Influence of thread affinity on the rendering performance*

5.2. RTNeuron scalability

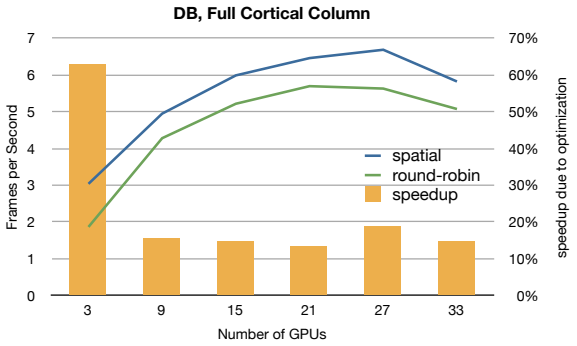
We tested the basic scalability of RTNeuron using a static and a load-balanced 2D compound as well as one direct send DB compound with the round-robin and spatial DB decomposition, as shown in Figure 8.

In the static 2D and both DB modes RTNeuron can cache the results of the cull operation, as it is the case in the typical use case of this application. Nevertheless the load-balanced 2D decompositions provides a better performance than the static decomposition. Due to the culling overhead, this mode does not scale as well as it does in the more simple eqPly benchmark.

For the DB rendering modes, the spatial decomposition delivers better performance due to the reduced compositing cost of not using the depth buffer and increasingly sparser region of interest. The spatial mode delivers about one frame per second better performance than the round-robin DB decomposition throughout all configurations, resulting in a speedup of 12-60% at the observed framerate.



(a)



(b)

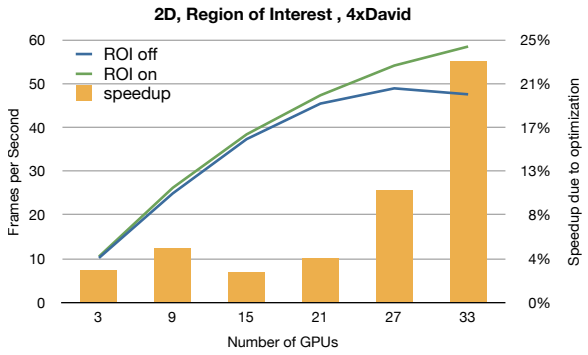
Figure 8: *RTNeuron scalability for 2D (a) and DB (b) decomposition*

5.3. Region of Interest

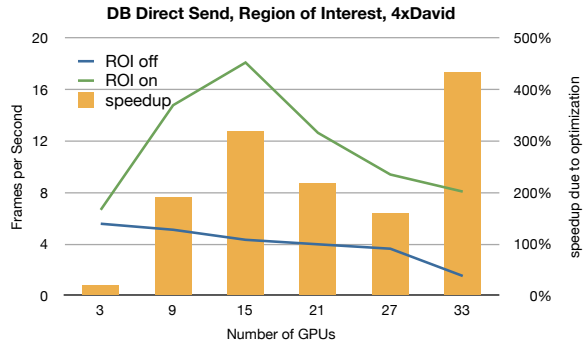
Applying ROI for 2D compounds has a small, constant influence when a small number of resource is used, as shown in Figure 9(a). With our camera path and model, the improvement is about 5% due to the optimized compositing. As the number of resources increases, the ROI becomes more important since load imbalances have a higher impact on the overall performance. With ROI enabled we observed performance improvements when using all 33 GPUs, reaching 60hz. Without ROI, the framerate peaked at less than 50hz when using 27 GPUs.

For depth-based DB compositing the ROI optimization during readback is especially important, as shown in Figure 9(b). The readback and upload paths for the depth buffer are not optimized and incur a substantial cost when compositing the depth buffer at full HD resolution. Furthermore, the messaging overhead on TCP causes significant contention when using 21 GPUs or more.

For the round-robin DB mode in RTNeuron, adding a region of interest has predictably very little influence, since each resource has only a marginally reduced region with this decomposition, as shown in Figure 10(a). On the other hand, the spatial DB mode can benefit of more than 25% due to the

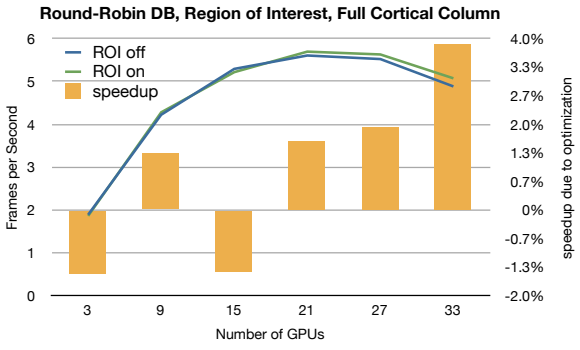


(a)

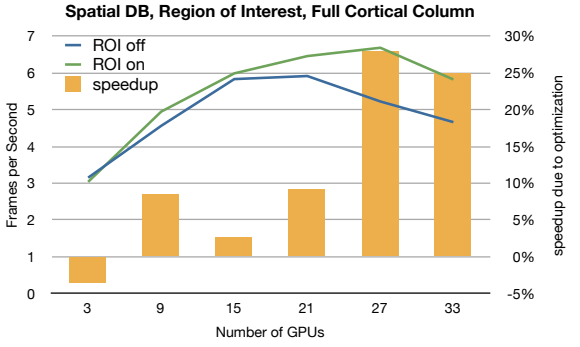


(b)

Figure 9: *Influence of ROI on eqPly 2D (a) and DB (B) rendering performance*



(a)



(b)

Figure 10: *Influence of ROI on RTNeuron round-robin (a) and spatial (b) DB rendering performance*

increasingly spare image regions as resources are added, as shown in Figure 10(b).

5.4. Asynchronous Readback

Asynchronous readbacks are, together with ROI, one of the most influential optimizations. When being mostly rendering bound, pipelining the readback with the rendering yields a performance again of about 10%. At higher framerates, when the rendering time of a single resource decreases, asynchronous readback has even a higher impact, of over 25% in our setup, as shown in Figure 11.

5.5. Multi-threaded versus Multi-process

Using three processes instead of three rendering threads on a single machine yields a slight performance improvement of up to 6% for eqPly, as shown in Figure 12. This is due to decreased memory contention and driver overhead with multi-threaded rendering, but comes at the cost of using three times as much memory for 2D compounds.

5.6. Finish after Draw

During benchmarking we discovered that a *glFinish* after the draw operation dramatically increases the rendering performance in DB decompositions, contrary to common sense. Consequently, we extended our implementation to insert a *glFinish* after the application's draw callback. The finish might allow the driver to be more explicit about the synchronization, since it enforces the completion of all outstanding rendering commands. We could not investigate this issue further due to the closed nature the OpenGL drivers used. Figure 13 shows a speedup of up to 200%.

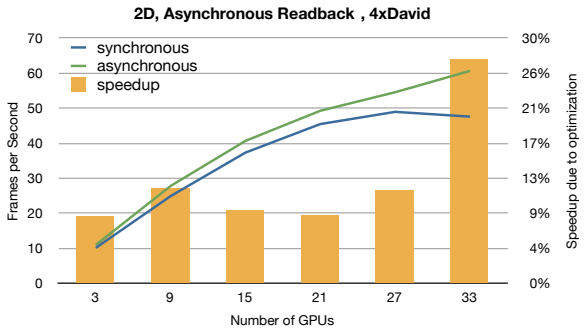


Figure 11: *Difference between synchronous and asynchronous readback*

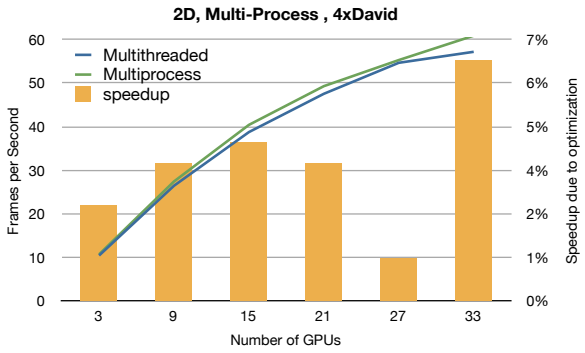


Figure 12: *Difference between multi-threaded and multi-process rendering*

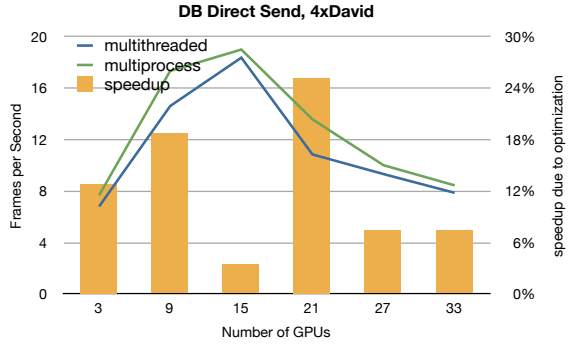


Figure 13: *Influence of glFinish on DB decomposition performance*

6. Conclusions and Future Work

This paper presents, analyses and evaluates a number of different optimizations for interactive parallel rendering on medium-sized, hybrid visualization clusters. We have shown that to reach interactive frame rates with large data sets not only the application rendering code has to be carefully studied but also that the parallel rendering framework requires careful optimization.

Apart from implementing and evaluating known optimizations such as asynchronous readbacks and automatic thread placement in a generic parallel rendering framework, we presented a novel algorithm for the optimization of 2D load-balancing re-using region of interest information from the compositing stage for refined load distribution.

The study of the impact of the individual optimizations provides valuable insight into the influence of the various features on real-world rendering performance. Our performance study on a hybrid visualization cluster demonstrates the typical scalability and common roadblocks towards high-performance large data visualization.

We want to benchmark the use of RDMA over InfiniBand, in particular for DB decompositions which have a significant network transport overhead when using TCP over 10 Gbit ethernet.

Regarding RTNeuron, our tests showed that there is room for improvement in several areas. The view frustum culling implementation has been identified as one of the major obstacles for scalability in some configurations. We will evaluate if culling overhead can be reduced by using a different algorithm, such as clustering the cell segments and using an independent capsule skeleton for each cluster. Also, the current visualizations target circuit models where the cells geometries are not unique. The simulation is moving towards completely unique morphologies, which is a complete paradigm shift with new challenges to be addressed.

We want to further analyse the use of subpixel compounds to improve the visual quality for RTNeuron, which requires strong anti-aliasing for good visual results.

Acknowledgments

The authors would like to thank and acknowledge the following institutions and projects for providing 3D test data sets: the Digital Michelangelo Project and the Stanford 3D Scanning Repository. This work was supported in part by the Blue Brain Project, the Swiss National Science Foundation under Grant 200020-129525 and by the Spanish Ministry of Science and Innovation under grant (TIN2010-21289-C02-01/02) and the Cajal Blue Brain Project.

We would also like to thank github for providing an excellent infrastructure hosting the Equalizer project at http://github.com/Eyescale/Equalizer/.

References

[ACCC04] ABRAHAM F., CELES W., CERQUEIRA R., CAMPOS J. L.: A load-balancing strategy for sort-first distributed rendering. In *IN PROCEEDINGS OF SIBGRAPI 2004* (2004), IEEE Computer Society, pp. 292–299. 3

[AP98] AHRENS J., PAINTER J.: Efficient sort-last rendering using compression-based image compositing. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization* (1998). 2

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: VR Juggler: A virtual platform for virtual reality application development. In *Proceedings of IEEE Virtual Reality* (2001), pp. 89–96. 2

[BRE05] BHANIRAMKA P., ROBERT P. C. D., EILEMANN S.: OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization* (2005), pp. 119–126. 2

[DK11] DOERR K.-U., KUESTER F.: CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics 17*, 2 (March 2011), 320–332. 2

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics* (May/June 2009). 2, 4

[EP07] EILEMANN S., PAJAROLA R.: Direct send compositing for parallel sort-last rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2007). 2, 5

[Her11] HERNANDO J. B.: *Interactive Visualization of Detailed Large Neocortical Circuit Simulations*. PhD thesis, Facultad de Informática, Universidad Politécnica de Madrid, 2011. 4

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21*, 3 (2002), 693–702. 2

[HHS*11] HAY E., HILL S., SCHÜRMANN F., MARKRAM H., SEGEV I.: Models of neocortical layer 5b pyramidal cells capturing aăwide range of dendritic and perisomatic active properties. *PLoS Comput Biol 7*, 7 (07 2011), e1002107. 4

[HSMd08] HERNANDO J. B., SCHÜRMANN F., MARKRAM H., DE MIGUEL P.: RTNeuron, an application for interactive visualization of detailed cortical column simulations. *XVIII Jornadas de Paralelismo, Spain* (2008). 4

[LHS*12] LASSERRE S., HERNANDO J., SCHÜRMANN F., DE MIGUEL ANASAGASTI P., ABOU-JAOUDÂŐ G., MARKRAM H.: A neuron membrane mesh representation for visualization of electrophysiological simulations. *IEEE Transactions on Visualization and Computer Graphics 18*, 2 (2012), 214–227. 4

[LRC*03] LUEBKE D., REDDY M., COHEN J. D., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, San Francisco, California, 2003. 4

[LRN96] LEE T.-Y., RAGHAVENDRA C., NICHOLAS J. B.: Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (July-September 1996), 202–217. 2

[Mar06] MARKRAM H.: The Blue Brain Project. *Nature Reviews Neuroscience 7*, 2 (2006), 153–160. http://bluebrain.epfl.ch. 4

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32. 1

[MEP10] MAKHINYA M., EILEMANN S., PAJAROLA R.: Fast compositing for cluster-parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 111–120. 2, 4

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap image composition. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 59–68. 2

[MPI12] MPI O.: Portable Hardware Locality. http://www.open-mpi.org/projects/hwloc/, 2012. 3

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A configurable algorithm for parallel image-compositing applications. In *Proceedings ACM/IEEE Conference on High Performance Networking and Computing* (2009), pp. 1–10. 2

[Ral62] RALL W.: Theory of Physiological Properties of Dendrites. *Annals of the New York Academy of Science 96* (1962), 1071–1092. 4

[SKN04] SANO K., KOBAYASHI Y., NAKAMURA T.: Differential coding scheme for efficient parallel image composition on a pc cluster system. *Parallel Computing 30*, 2 (2004), 285–299. 2

[SML*03] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: SLIC: Scheduled linear image compositing for parallel volume rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 33–40. 2

[SMV11] SPAFFORD K., MEREDITH J. S., VETTER J. S.: Quantifying numa and contention effects in multi-gpu systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2011), GPGPU-4, ACM, pp. 11:1–11:7. 2

[SZF*99] SAMANTA R., ZHENG J., FUNKHOUSER T., LI K., SINGH J. P.: Load balancing for multi-projector rendering systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 1999), HWWS '99, ACM, pp. 107–116. 3

[TIH03] TAKEUCHI A., INO F., HAGIHARA K.: An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing 29*, 11-12 (2003), 1745–1762. 2

[VBRR02] VOSSG., BEHR J., REINERS D., ROTH M.: A multi-thread safe foundation for scene graphs and its extension to clusters. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2002), EGPGV '02, Eurographics Association, pp. 33–37. 2

[YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings IEEE/ACM Supercomputing* (2008). 2

[YYC01] YANG D.-L., YU J.-C., CHUNG Y.-C.: Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *Journal of Supercomputing 18*, 2 (February 2001), 201–22–. 2