Eurographics Symposium on Parallel Graphics and Visualization (2008)
J. Favre, K. - L. Ma, and D. Weiskopf (Editors)

Multi-GPU Sort-Last Volume Visualization

Stéphane Marchesin and Catherine Mongenet and Jean-Michel Dischler

LSIIT, UMR CNRS-ULP 7005, Louis Pasteur University, Strasbourg, France

Abstract

In this paper, we propose an experimental study of an inexpensive off-the-shelf sort-last volume visualization architecture based upon multiple GPUs and a single CPU. We show how to efficiently make use of this architecture to achieve high performance sort-last volume visualization of large datasets. We analyze the bottlenecks of this architecture. We compare this architecture to a classical sort-last visualization system using a cluster of commodity machines interconnected by a gigabit Ethernet network. Based on extensive experiments, we show that this solution competes very well with a mid-sized PC cluster, while it significantly improves performance compared to a single standard PC.

1. Introduction

Thanks to the advent of dedicated graphics hardware, parallel architectures have been widely used to solve high-scale, large dataset graphics problems. Lately, commodity clusters are being used in the visualization field as well, and can lead to interactive performance even with very large datasets. These clusters make use of multiple machines, each having its own CPU and GPU interconnected by a network, usually gigabit Ethernet, Myrinet or Infiniband. However, because they require communication between multiple machines, visualization clusters add complexity and cost both on the hardware and on the software front. Furthermore, the interconnection network is often a performance bottleneck of such clusters, especially when high resolution pictures, such as those required for immersive environments, are to be produced.

The purpose of this paper is to resolve the problem of sort-last volume rendering for large datasets using a simple, inexpensive, off-the-shelf architecture that takes advantage of multiple graphics cards in a single machine instead of a full cluster of PCs. To the extent of our knowledge, no such off-the-shelf system has been explored in the sort-last context before, nor has a specific pipeline for such a hardware setup been proposed. We show that architectural differences between the multi-GPU system and the cluster lead to different system bottlenecks and therefore impact the resulting performance. We propose to adapt the sort-last volume rendering pipeline commonly used on clusters onto this multi-GPU architecture. Through a series of benchmarks, we show the influence of different parameters (such as the brick size or the rendering method) on the global rendering speed. This allows us to identify optimal parameters. Using a 1 GB dataset (e.g. too large to be visualized without degradation on a single GPU) we compare visualization performance on the multi-GPU system and the cluster. It shows that both result in similar performance, while the multi-GPU system is simpler, cheaper and easier to program. This study demonstrates that such a system is a promising solution for volume visualization of large datasets.

In the next section, we introduce related works. Section 3 details the sort-last pipeline for cluster-based volume visualization and points out the architectural differences of a multi-GPU system. This pipeline is described in Section 4. Section 5 is dedicated to implementation and experimental results, and compares the behaviour of our multi-GPU system to that of a similar visualization cluster. Finally, conclusions and future works are given in Section 6.

2. Related works

Large data visualization 3D texture-based volume visualization remains one of the most efficient techniques for direct volume rendering, but the memory limitations of current GPUs represent a serious hurdle that many methods attempt to circumvent, often by lowering the visual quality.

The first kind of technique uses simplification-based methods by decomposing the data into a number of equally-sized parallelepiped bricks (this is commonly called bricking). Each of these bricks is handled separately, and it is

© The Eurographics Association 2008.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

therefore possible to discard invisible bricks, or use multiresolution techniques. More generally, this provides a finer data manipulation granularity in the whole volume rendering pipeline. In this context, Weiler *et al.* [WWH*00] and Lamar *et al.* [LHJ99] use 3D textures at different levels of detail to achieve large dataset visualization without using too much texture memory. In both approaches the algorithm decides which level to use for each brick depending on the brick contents, its distance to the observer and other parameters. Guthe *et al* [GS04] achieve large volume dataset visualization by compressing the data offline using wavelets. The data is then reconstructed on-the-fly at different detail levels depending on the viewing conditions, and the rendering makes use of advanced techniques such as occlusion culling and empty space skipping to gain further speed up. Lamar *et al.* [LHJ03] propose a technique which allows efficient error calculation in the context of level of detail volume rendering. By decomposing the data into bricks and storing the histogram of each brick, they are able to determine the visibility of the brick as well as the approximation error by looking at the histogram only, thereby make interactive transfer function changes together with bricking possible. However, some of these techniques incur data degradation, and in order to handle large datasets without degrading the quality of the pictures, one has to resort to parallel systems.

Parallel rendering Molnar *et al.* [MCEF94] classify parallel rendering systems according to the placement of the sorting phase in the parallel graphics pipeline, and derive three categories: sort-first, sort-middle and sort-last. When sorting is done prior to both primitive transforming and rasterization, the approach is of the sort-first kind. When sorting is done between those phases, it is of the sort-middle kind. Finally, if sorting is at the end of the pipeline, after rasterizing the primitives, the approach is called sort-last. Among these approaches, we will focus on sort-last, since it is the most suited for large dataset visualization tasks.

For volume visualization, sort-last algorithms allow visualizing large datasets, as demonstrated by Wylie *et al.* [WPLM01]. Ma *et al.* [MPHK94] propose the binary swap technique, which is a highly scalable compositing algorithm for sort-last rendering. Stompel *et al.* [SML*03] present a parallel image compositing algorithm minimizing the amount of composited data and scheduling the compositing tasks on the processors of a cluster. Strengert *et al.* [SMW*04] propose an efficient hierarchical sort-last volume rendering technique, and report interactive results on a Myrinet interconnection network. Roth *et al.* [RR06] optimize the sort-last pipeline by splitting the screen into tiles and taking advantage of occlusion and full transparency of tiles in that context.

In order to achieve good performance and scalability with sort-last volume visualization, one also has to load balance the volume data between the nodes. This is not easily achieved, since transfer function or viewpoint changes can both result in unbalancing a previously balanced data distribution. Therefore, Marchesin *et al.* [MMD06] and Müller *et al.* [MSE06] use a hierarchical decomposition of the dataset into a KD-tree which is mapped onto the cluster nodes. The tree is then rebalanced in real time according to the node's respective load values. Even though parallel visualization machines can easily handle large datasets, they often incur significant additional complexity both on the hardware side, since an interconnection network is required, and on the software side, since code to implement the data communication must be developed. Using multiple graphics accelerators in a single machine would avoid these issues, at a lower cost.

Multicard Humphreys *et al.* [HHN*02] introduce Chromium, which is a framework for cluster-based rendering. This framework allows both sort-first and sort-last, and can distribute an application over a cluster of machines without requiring changes to it. Bhaniramka *et al.* [BRE05] introduce the SGI multipipe SDK. This SDK allows multiple card rendering to be used in common applications. However, this API is designed for expensive high-end SGI workstations and requires hardware composition for maximum efficiency. NVIDIA introduced SLI [nvib] and Quadro Plex [nvia] which transparently distributes the rendering workload to multiple cards in a sort-first fashion. However, such setups are limited to sort-first configurations and therefore do not scale well with the data size. Furthermore, it is limited to a number of specific NVIDIA cards only. Penner *et al.* [PSC] implement a drop-in replacement for the Direct3D library that parallelizes all the Direct3D applications over multiple graphics cards and multiple screens on a single system. This allows transparent rendering over multiple displays. Again, this technique is limited to sort-first situations. Unlike the previously described multicard-based methods, our technique implements a sort-last visualization algorithm on a single machine. Using sort-last as opposed to sort-first is known to allow better scalability when increasing the input data size. Since it does not replicate the data, sort-last is the most appropriate algorithm for large dataset visualization on parallel systems.

3. Comparing sort-last pipelines: cluster vs. multi-GPU

In this section, we detail our new sort-last multi-GPU pipeline. Figure 1 depicts the differences between sort-last on a two-node cluster and sort-last on a single multi-GPU machine with two GPUs. Each of the rendering nodes (in a cluster) or GPUs (in a multi-GPU machine) is called a client, and the node in charge of the compositing is called the server. The blue stages are done by the CPU and the green stages are done by the GPU, and the memory buffers are shown in yellow. Since a single machine differs from a cluster at an architectural viewpoint, one has to adapt the commonly used sort-last volume visualization methods to such an architecture. The classical sort-last pipeline is shown on the left of the figure. This pipeline works as follows. The

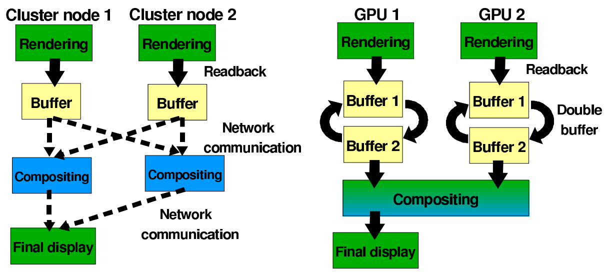


Figure 1: *The sort-last rendering pipeline on a cluster (left) and on a multi-GPU machine (right). The green stages run on the GPU, while the blue stages run on the CPU. Memory buffers are shown in yellow. Notice that we tried compositing on the CPU and on the GPU for the multi-GPU case.*

data is initially partitioned across the client nodes. For each frame, each client first renders its own data and then reads back the rendered images to system memory. The next phase consists in compositing the images together. In a typical sort-last system, this is done using the direct send algorithm [Hsu93]. To achieve composition, the screen is first partitioned into as many areas as there are clients, and each client is then in charge of compositing one of these areas. The relevant pictures for each area are sent to the corresponding client node during a communication phase as shown on the left of Figure 1. Once the client has received all the pictures for its area, it composes them together to form a part of the final picture. Finally, those composited pictures are gathered on the server node for final display. The case of direct send where multiple nodes are in charge of the compositing allows better scalability with an increasing number of nodes than the case where a single server node does the whole compositing itself.

In the multi-GPU case, we can notice a number of differences on the hardware which have implications on the pipeline of the parallel rendering algorithm. Let us now review the main stages of the sort-last volume rendering pipeline as depicted on Figure 1 and compare them in both situations:

Rendering This stage is in charge of rendering the data in a distributed fashion. In order to handle large scale datasets, the data is split into bricks, and visible bricks are determined and rendered. Level of detail techniques have been implemented and tested, but the benchmarks presented in this paper do not include such techniques, which makes the results more easily reproducible. Volume rendering itself is achieved using a classical 3D texture-based approach that slices the volume into multiple polygons or using raycasting. However, when using a slice-based approach, only one CPU will compute and send the slices to all the GPUs in the multi-GPU case as opposed to a cluster where each CPU computes and sends its vertex data to only one GPU. Therefore, this stage can become a bottleneck and we have to optimize it carefully as detailed in Section 4.

Readback The bricks that were previously determined to be visible are projected, and their footprint is read back. However, in the case of a multi-GPU machine, all reads are done to the same system memory, which could result in bandwidth starvation. We have experimented a number of different techniques to optimize this stage as detailed in the next section.

Compositing In the case of a visualization cluster, the compositing stage requires a communication phase to gather pieces of partial images to the compositing nodes. However, in the case of the multi-GPU machine, no communication phase has to take place. Although this might sound like an advantage at first, one has to keep in mind that all the compositing will be done on a single CPU and through a single memory bus. Therefore, in order to spare memory bandwidth, we have to reduce memory pressure as much as possible. An alternative is to have the composition done on the GPU.

Final display Once the image is ready, it is sent to the screen for final display. This stage presents no difference on a multi-GPU machine or on a visualization cluster.

From the previous qualitative comparison, one can notice that there are major differences between the classical cluster-based sort-last visualization pipeline and our multi-GPU system, which we will address in the next section.

4. Multi-GPU sort-last pipeline

Based on the previous comparative study, this section describes our modified multi-GPU sort-last pipeline and its implementation.

Rendering The rendering phase takes place first, in which each client process renders its own bricks. The rendering is done using either a 3D texture-based slicing approach or a GPU-based raycasting approach. In order to achieve good scalability, we have to minimize the overhead of sending the vertex data to the card. A solution to this problem is to use bigger bricks. Since each brick has to be sliced separately and therefore generates its own set of polygons, the more bricks there are, the more polygons must be sent to the card. However, increasing the brick size also reduces the granularity at which invisible data is culled, and reduces culling efficiency. Therefore, additional improvements can be obtained without changing the brick size by making use of OpenGL extensions for efficient vertex submission. We have experimented three ways of sending the vertex data to the OpenGL API and tested their respective performance: immediate mode, vertex arrays and vertex buffers objects. The first technique generates a single OpenGL call for each vertex, whereas the two latter techniques generate calls in batches, thereby reducing the overhead. Since the CPU has to send vertex data to all the cards at once, minimizing the overhead of such calls is of primary importance. For that reason, we have also implemented a GPU-based raycaster that avoids computing and sending slices altogether.

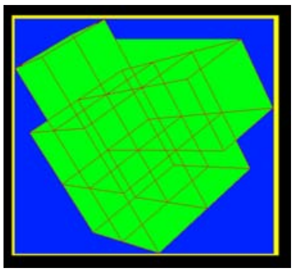


Figure 2: *Readback techniques: the 10 visible bricks are shown in red wireframe, span based readback is depicted in green, the screen-aligned bounding box is shown in blue and the same box aligned over a 32 pixel alignment is shown in yellow. Each of these areas is a super-set of the previous one.*

Readback Once the data has been rendered, the produced pictures have to be read back from video memory. We have implemented three techniques in order to reduce the amount of data to be read back from video memory as shown on Figure 2. These techniques are:

- Projecting the visible bricks, and using the screen-aligned bounding rectangle as a readback area as shown in blue on Figure 2.
- Projecting the visible bricks, and using the screen-aligned bounding rectangle as a readback area, as depicted in yellow on Figure 2. Aligning the bounding rectangle width over powers of two will help the subsequent CPU-based compositing phase, since this aligns memory access to each pixel. We have tested alignments of 2, 4, 8, 16 and 32.
- Projecting the visible bricks, and turn their footprint into single-line spans (in green on Figure 2). This results in more readback operations, but in a smaller readback area.

Since all GPUs read back their contents to a common memory area, the memory bandwidth can easily become a limiting factor. Therefore, we use a Unix System V shared memory buffer to exchange image data between the clients and the server, which results in a copy-less system between the clients, thereby reducing the strain on the memory bandwidth (obviously copying still happens from GPU memory to system memory for readback, and from system memory to GPU memory for final display).

Compositing We have experimented with two ways to compose the intermediate pictures into a final image. The first way is to use the CPU, in which case the composition is achieved by blending the pictures in a back-to-front order using the OVER operator as defined by Porter and Duff [PD84]. In that case, the intermediate picture from each GPU is read to system memory, the CPU does all the compositing, and the final picture is sent back to the GPU used for display (called the target GPU). The second way is to use the GPU for composition and take advantage of the fact that one of the pictures is already residing on the target GPU. To do so, we first read all the intermediate pictures except the one from the target GPU into system memory. These pictures are then sorted, and sent to the target GPU. We compose the pictures that are behind the target GPU's picture in a front-to-back fashion using the UNDER operator, and then those that are in front of the target GPU's picture in a back-to-front order using the OVER operator. In that case, all but one of the intermediate pictures have to be read to system memory, and those pictures must also be sent to the target GPU. As shown on the right of Figure 1, we use a double buffering scheme for communication between the clients and the server, and therefore we can overlap the final display and the rendering computation.

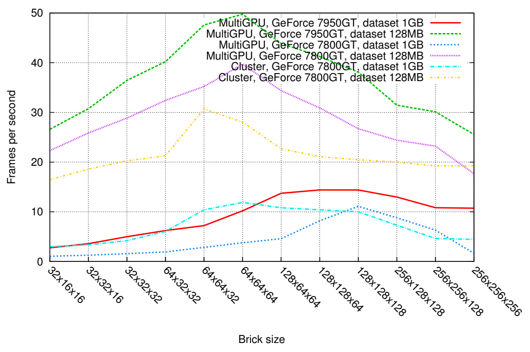
Final display Once the final picture is produced, it is sent to the screen for final display, similarly to a cluster-based sort-last visualization system. However, instead of sending the result to a server node, one of the GPUs is reused. We have measured that doing so has minimal impact on the volume rendering performance of this GPU since the cost of displaying a 2D picture is low (we measured it to be approximately 3% of the GPU time for a 1024×768 screen).

5. Implementation and results

This section presents our implementation, shows benchmarks for each stage of our sort-last pipeline, both using our multi-GPU sort-last volume rendering approach and the classical cluster-based approach, and discusses these results.

The commonly accepted solution for parallel visualization is to use a cluster of machines. We compared our architecture to a 9 node off-the-shelf visualization cluster running Linux (consisting of 8 client nodes and one server node). Each cluster node is equipped with an Athlon X2 4200+ processor, 2GB of memory and a GeForce 7800GT graphics card with 256MB of memory. The interconnection network used is gigabit Ethernet. This cluster runs a direct send sort-last volume visualization algorithm where the readback, communication and compositing phases work on the footprint of the data. An alternative would be to use binary swap, but experiments showed that direct send was faster in our case. For scalability tests, we run this cluster either as a $8+1$, $4+1$ or $2+1$ setup (n clients + 1 server). The multi-GPU machine used for these tests is equipped with a motherboard that supports 4 PCI Express slots, all at $8\times$ speed. The processor is a Pentium-D at 3.4 GHz and has 4GB of memory. Tests were conducted both with the same graphics cards as the cluster (GeForce 7800GT 256MB) in order to compare the architectures, and with better cards (GeForce 7950GT 512MB). We have implemented our multi-GPU sort-last volume visualization algorithm under Linux. In order to access the different GPUs independently, we configure the X server with exactly one X screen per card, even though no physical screen is actually connected to the card.

Let us assume we have n GPUs available. On startup, the



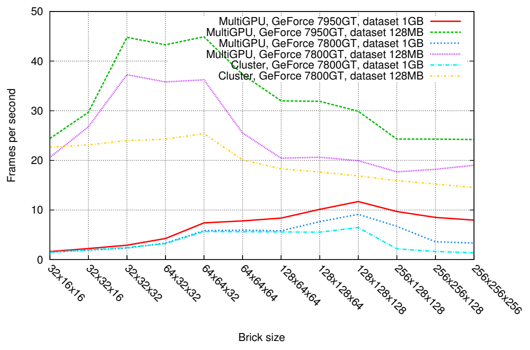


Figure 3: *Influence of the brick size using CPU-based slice computation (top) and GPU-based raycasting (bottom).*

server spawns n processes (in the CPU compositing case) or $n-1$ processes (in the GPU compositing case). Each of these processes opens a connection to a different X screen, and creates an OpenGL pbuffer. All rendering is then done through this pbuffer. Therefore, each client is able to explicitly access its own graphics card. The multi-GPU sort-last volume visualization implementation used is the same as that of the cluster, except that the communication stage is removed. It is therefore possible to directly compare the performance of the two architectures. The benchmarks have been conducted with multiple datasets: one is the 128MB (512^3 voxels) Christmas tree dataset and the other one is a 1GB (1024^3 voxels) geological core dataset (respectively seen on Figure 9). The volume rendering implementation uses bricking, brick-based empty space skipping and pre-integration [EKE01], both for slice-based and raycasting-based rendering. Unless specified, a 1024×768 viewport is used. All the datasets were sampled at 1.5 voxel's width for rendering, both for slice-based rendering and for raycasting.

Rendering The first stage of the pipeline is the rendering stage. In order to find out the best parameters for this stage, we compare results on a $4+1$ nodes cluster with the multi-GPU system using its 4 GPUs. Figure 3 shows the influence of the choice of the brick size on the rendering speed, using respectively a slicing-based rendering approach (top) and a raycasting-based one (bottom). Since bricks overlap by one voxel in order to achieve rendering

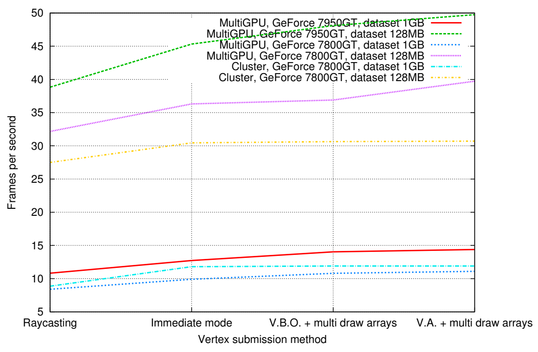


Figure 4: *Performance impact of the vertex submission techniques.*

continuity, there is a trade-off to make between the brick size and the memory overhead. These figures show that using GPU-based raycasting is not as efficient as using slice-based volume rendering. Indeed, for an optimal brick size, the CPU-based slicing method produces a framerate increase over the GPU raycasting approach: about 10% for the smaller dataset, and approximately 20% for the larger dataset. One can notice that the optimal brick size depends on the dataset and the rendering technique used. On the multi-GPU machine, the 1GB dataset seems to perform better with $128 \times 128 \times 128$ bricks with both rendering approaches. On the same machine the 128MB dataset has better framerates with $64 \times 64 \times 64$ bricks for the slicing-based approach, and $32 \times 32 \times 32$ bricks for the slicing-based approach. On the cluster, the optimal brick size for the 128MB dataset remains $64 \times 64 \times 32$ for both rendering techniques, whereas the optimal brick size for the 1GB dataset depends on the rendering approach: $64 \times 64 \times 64$ for CPU-based slicing and $128 \times 128 \times 128$ for raycasting. In order to reduce the per-vertex overhead of our system, we have tried different rendering techniques. Figure 4 shows three slicing-based approaches, namely vertex arrays combined with the GL_EXT_multi_draw_arrays OpenGL extension, vertex arrays, and vertex buffer objects, and one raycasting-based approach, namely GPU-based raycasting which should avoid the computation and sending of the vertices by the CPU altogether. The figure demonstrates that vertex arrays combined with the GL_EXT_multi_draw_arrays OpenGL extension result in the best performance, improving the framerates by more than 10% over the baseline in the case of a Multi-GPU machine with GeForce 7950GT cards. Although it requires less work to be done on the CPU and less data to travel over the bus, the GPU-based raycasting approach is not globally faster. This is due to the locality of texture access in the shader which is lower than with bare texturing. However, the pictures produced using GPU raycasting have slightly better quality especially when using a small sampling step, thanks to the intermediate computations being done in GPU registers at full 32-bit floating point accuracy inside each brick.

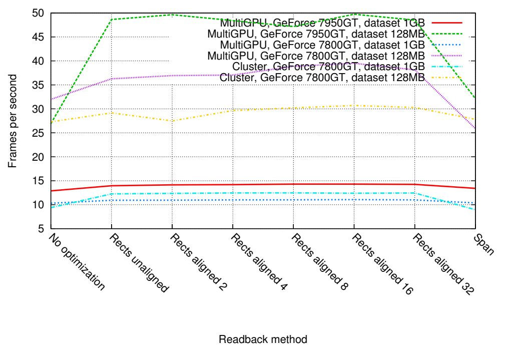


Figure 5: *Readback optimization techniques.*

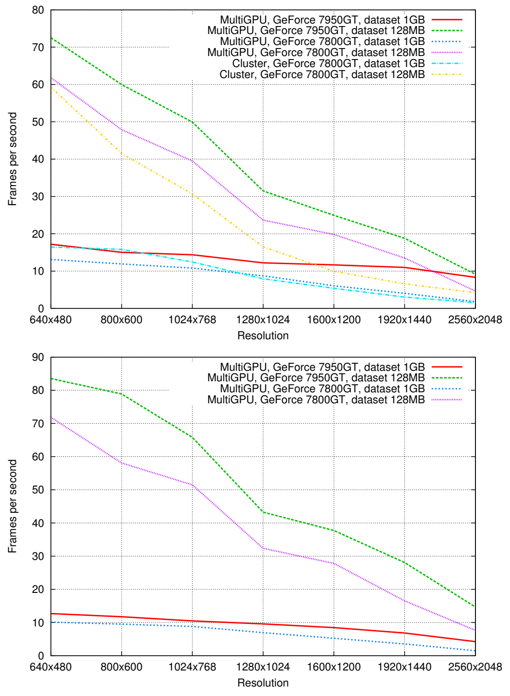


Figure 6: *Scalability with the screen resolution using CPU composition (top) and GPU composition (bottom).*

Readback The second stage of a sort-last rendering system is the readback of intermediate pictures from the cards to system memory. Figure 5 shows the influence of the readback optimization techniques we tried. In particular, this figure outlines that using pixel spans as the readback primitive does not result in performance increase, but instead degrades the framerate, both on a cluster and on the multi-GPU machine. Indeed, numerous small readbacks result in a smaller global readback bandwidth, and therefore cause a slowdown. On the other hand, projecting the data bounding box and aligning the boundary of this box over a multiple of 16 pixels results in the best performance in all cases.

Compositing The next stage of the pipeline is the composition stage. Figure 6 shows the global rendering speed when

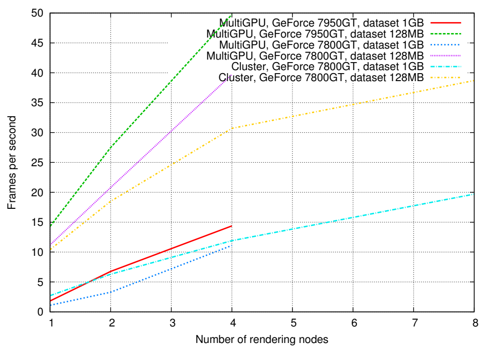


Figure 7: *Multi-GPU vs. cluster scalability with the number of GPUs.*

using CPU-based (top) and GPU-based composition (bottom) with two different datasets using the optimal brick size as previously found. These figures show that GPU-based compositing is more interesting when the number of composited pixels per second is high (that is using the smaller dataset), while CPU-based compositing prevails for a lower number of pixels (that is when using the bigger dataset). We have measured that GPU-based compositing (including the readback from screen and display of the final picture operations) can compose up to 146 millions of pixels per second, while CPU-based compositing using SSE assembly can only achieve 105 millions of pixels per second. Notice that our multi-GPU system scales well with the screen resolution. In particular, it achieves 28 frames per second when viewing a 128MB dataset on a single multi-GPU machine with a 1600×1200 viewport and GeForce 7800GT cards. In contrast, a cluster with the same graphics hardware achieves approximately only 10 frames per second with the same resolution and dataset.

Scalability Figure 7 compares the global performance of our multi-GPU approach with that of a similar visualization cluster, with both datasets and a 1024×768 resolution. CPU-based composition is used for these tests. These results show good scalability for both the 128MB and the 1GB datasets: using 4 GPUs, we are able to achieve a speedup factor of 3.5 with the 128MB dataset, and a speedup factor of 8 with the 1GB dataset, thanks to the increase of available texture memory. These tests also show that our system is consistently faster than a similarly equipped cluster for the smaller 128MB dataset, and is a little slower or reaches similar performance levels for the 1GB dataset. The fact that our multi-GPU setup performs almost as well as a similarly-equipped $4+1$ node visualization cluster with the 1GB dataset is very promising as its cost is significantly lower, since it is based on a single machine and does not require an efficient and therefore expensive interconnection network. If one considers a 1.5 voxels width sampling distance, an approximation of the number of vertices to be handled per second can be obtained by muliplying the optimal brick size's width with the number of bricks and the number of frames per second. For the 1024^3 dataset rendered at 12 frames per second and 128^3 bricks (which was shown experimentally to have no discarded bricks), this gives us $128 \times 8^3 \times 12 = 786432$ polygons per second. Considering a cluster or a multi-GPU setup with n GPUs, the CPU of a given cluster node (in charge of computing the slicing) is dedicated to a single card, and therefore computes $1/n$ of that an amount of slices. In the case of our multi-GPU machine, the CPU is shared between the cards and has to compute all the slices. Also, one has to keep in mind that the CPU from the multi-GPU machine is slower (3.4 GHz Pentium-D dual core) than the one in a single cluster node (Athlon X2 4200+ dual core). Furthermore, if one considers that a plane slicing a cube has 4 vertices on average, and that each vertex is 36 bytes (each vertex carries 3 3-component floating point attributes), that is 108MB per second of data to be sent to the cards. While the PCI Express bus for the cluster nodes operates at 16$\times$ speed, the the bus in the multi-GPU machines operates at 8$\times$ because of technical limitations of the motherboard used. In the case of the 128MB dataset, our multi-GPU system outperforms the cluster because the cluster becomes communication-limited by the bandwidth of the Ethernet network, while the multi-GPU machine does not require this time-costly communication phase. When switching to the GeForce 7950GT cards, the multi-GPU setup sees higher performance. This shows that our system remains scalable with improvements on the graphics hardware side. This is promising, as future improvements in graphics hardware will thus warrant related improvements in the performance of our sort-last volume visualization system. One last thing we noticed during our tests is that our system does not introduce any additional latency because of the communication phase, and no jittering was observed, both of which are commonly seen on clusters when low-cost interconnection networks such as Ethernet are used.

Time breakup Finally, Figure 8 shows how the workload is distributed among the different pipeline stages and among the nodes on both architectures. Using the 1GB dataset, a CPU-based compositing and slicing approach and the optimal brick size as computed previously, we have measured the time taken by each pipeline stage when rendering to a 1024×768 frame. On the multi-GPU machine, the rendering time prevails over the other stages. This is also the case on the cluster where it overlaps with the communication and compositing stages. This figure shows that the readback times differ significantly between both platforms. This suggests that the use of a PCI Express 8$\times$ bus on the multi-GPU machine partly accounts for the performance difference between these platforms.

6. Conclusions and future works

In this paper, we have introduced an architecture for sort-last volume rendering based on a multi-GPU setup. As opposed

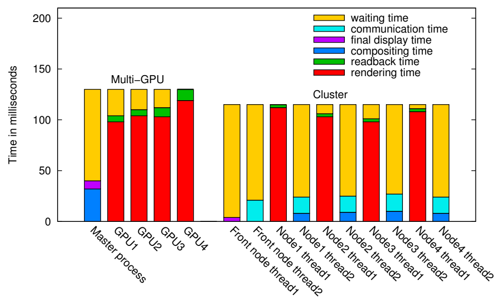


Figure 8: *Breakup of the times of the pipeline stages.*

to a cluster, this architecture does not require the use of multiple machines or an interconnection network. It is therefore much simpler, cheaper and easier to realize. The performance that we achieved by adapting the rendering pipeline to this new setup demonstrates that our parallel solution represents a highly competitive alternative to graphics clusters for large volume visualization tasks. Indeed, our system achieves interactive rendering of 1GB datasets at very large resolutions on a single machine, which is not possible on a single GPU, unless the data is degraded.

Our experiments show that the optimal brick size seems to depend on the dataset characteristics (both its size and nature) and the rendering method used. We would like to investigate with more datasets what parameters determine this optimal brick size, and how to automatically find it.

To our knowledge this work represents the first study of a multi-GPU setup used in a sort last volume rendering context. It therefore opens the way for further research. The tight coupling of such an architecture should allow us to make intensive use of information exchange between the GPUs to improve performance, which is only hardly possible on clusters because of the network latency and limited interconnection bandwidth available. In the future, we would also like to experiment with more graphics cards to see how scalable this solution is. However, as of today, no motherboard able to host more than 4 PCI Express graphics cards is available. It is therefore not possible right now to further test scalability on a single machine. Instead, two different ways could be investigated. First, multi-core machines could help distribute the compositing load among more CPUs, or allow computing real time brick occlusion. Second, we would like to experiment with hybrid systems, i.e. clusters of multi-GPU machines. In particular, we would like to derive hybrid hierarchical compositing schemes (across multiple cards in a single machine, and across numerous multi-GPU machines over a network) that are suited to such a cluster. In fact, from a conceptual viewpoint this adds a new level of parallelism between the internal parallelism of the graphics card and the parallelism of the cluster. Such hybrid approaches could also make use of the high data locality within a single machine to increase the performance. Achieving good locality on such a system will also require smart data distribution, which we plan to investigate further.

7. Acknowledgment

The authors would like to thank Yves Géraud from the "Institut de Physique du Globe de Strasbourg" UMR 7516, for providing the geological dataset used in this paper.

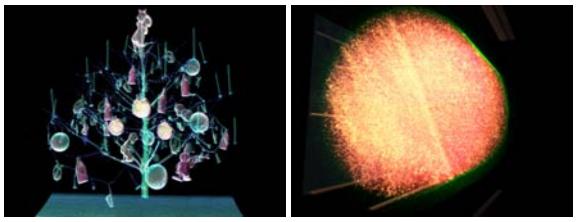


Figure 9: *The 128MB Christmas tree dataset (left) and 1GB geological dataset (right) rendered with our system.*

References

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

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001), ACM Press, pp. 9–16.

[GS04] GUTHE S., STRASSER W.: Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. *Computers & Graphics 28*, 1 (Feb. 2004), 51–58.

[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. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), ACM Press, pp. 693–702.

[Hsu93] HSU W. M.: Segmented ray casting for data parallel volume rendering. In *Proceedings of the symposium on Parallel rendering* (1993), pp. 7–14.

[LHJ99] LAMAR E., HAMANN B., JOY K. I.: Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of the IEEE Visualization conference* (1999), D. Ebert M. G., Hamann B., (Eds.), pp. 355–362.

[LHJ03] LAMAR E. C., HAMANN B., JOY K. I.: *Efficient Error Calculation for Multiresolution Texture-Based Volume Visualization*. Springer-Verlag, Heidelberg, Germany, 2003, pp. 51–62.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 23–32.

[MMD06] MARCHESIN S., MONGENET C., DISCHLER J.: Dynamic Load Balancing for Parallel Volume Rendering. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV06)* (2006), Eurographics Association, pp. 43–50.

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 59–68.

[MSE06] MÜLLER C., STRENGERT M., ERTL T.: Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV06)* (2006), Eurographics Association, pp. 59–66.

[nvia] http://www.nvidia.com/page/quadroplex.html.

[nvib] http://www.slizone.com/.

[PD84] PORTER T., DUFF T.: Compositing digital images. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), pp. 253–259.

[PSC] PENNER E., SCHMIDT R., CARPENDALE S.: A GPU cluster without the clutter: A drop-in scalable programmable-pipeline with several gpus and only one pc. In *ACM I3D 2006, Technical Poster.*

[RR06] ROTH M., REINERS D.: Sorted pipeline image composition. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV06)* (2006), Eurographics Association, pp. 119–126.

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

[SMW*04] STRENGERT M., MAGALLÓN M., WEISKOPF D., GUTHE S., ERTL T.: Hierarchical visualization and compression of large volume datasets using gpu clusters. In *EGPGV* (2004), pp. 41–48.

[WPLM01] WYLIE B., PAVLAKOS C., LEWIS V., MORELAND K.: Scalable rendering on PC clusters. vol. 21, IEEE Computer Society Press, pp. 62–70.

[WWH*00] WEILER M., WESTERMANN R., HANSEN C., ZIMMERMANN K., ERTL T.: Level-of-detail volume rendering via 3d textures. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization* (New York, NY, USA, 2000), ACM Press, pp. 7–13.