A Comparative Evaluation of Three Volume Rendering Libraries for the Visualization of Sheared Thermal Convection

Jean M. Favre[a,*], Alexander Blass[b]

[a] *Swiss National Supercomputing Center (CSCS), Via Trevano 131, CH-6900 Lugano, Switzerland*
[b] *Physics of Fluids Group, Max Planck Center for Complex Fluid Dynamics, J. M. Burgers Center for Fluid Dynamics and MESA+ Research Institute, Department of Science and Technology, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands*

Abstract

Oceans play a big role in the nature of our planet. About 70% of our earth is covered by water [1]. Strong currents are transporting warm water around the world and therefore do not only make life possible, but also allow us to harvest its power producing energy. Yet, oceans also carry a much more deadly side. Floods and tsunamis can easily annihilate whole cities and destroy life in seconds. The earth's climate system is also very much linked to the circulation in the ocean due to its large coverage of the earth's surface. Deep ocean currents can be simulated by means of wall-bounded turbulent flow simulations. To support these very large scale numerical simulations and enable the scientists to interpret their output, we deploy an interactive visualization framework to study sheared thermal convection. The visualizations are based on volume renderings of the temperature field. To address the needs of supercomputer users with different hardware and software resources, we evaluate different implementations supported in the ParaView [2] environment: two GPU-based solutions with Kitware's native volume mapper or NVIDIA's IndeX library, and a software-only OSPRay-based implementation.

Keywords:
Scientific Visualization, High Performance Computing, Navier-Stokes Solver, Direct Numerical Simulation, Computational Fluid Dynamics

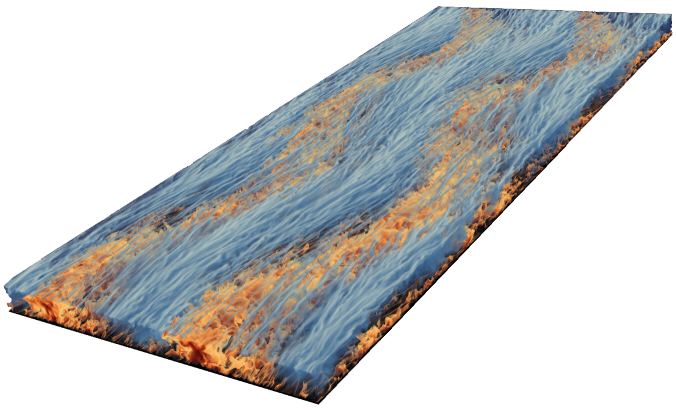

Figure 1: Snapshot of the three-dimensional temperature field of sheared thermal convection at $Ra = 4.6 \times 10^6$ and $Re_w = 6000$ [3].

1. Introduction

Thermohaline ocean circulation [4] is vital for the heat budget of our earth. Manabe and Stouffer [5] observed that it can contribute to a heat increase of up to $\sim$ 10°C on the yearly averaged mean surface temperatures in the North Atlantic region. Marshall and Schott [6] investigated a vast variety of scales in ocean dynamics and stated that deep convection can be related to mixing layers everywhere in the ocean. Since there are many complex three-dimensional events happening in large-scale fluid bodies such as oceans, it is vital to visualize the three-dimensional and temporal features of such flow simulations.

We study these large-scale bodies of fluids which are sheared by winds or currents and influenced by temperature differences in the flow. A fundamental setup of this natural mechanism is sheared thermal convection. Many processes in nature are based on heat and momentum transfer and therefore interaction between buoyancy [7, 8] and shear [9, 10]. Rayleigh-Bénard convection, the flow in a box heated from below and cooled from above is a paradigmatic system for thermal convection. We present the use of three different rendering libraries available in ParaView [2] to build a time-dependent volume rendering of thermal convection. The deployment, hardware and software

requirements, and the performance of these libraries are discussed. In the accompanying video [11] we are able to display the previously two-dimensionally presented flow structures in a three-dimensional motion. The reader is led through a presentation of one specific flow case with sheared thermal convection and can experience the dynamics of the thermal structures while being informed about different flow parameters.

*jfavre@cscs.ch

2. Numerical Procedure

Direct numerical simulations (DNS) were performed with the second-order finite-difference code *AFiD* [12], in which the three-dimensional non-dimensional Navier-Stokes equations with the Boussinesq approximation on a staggered grid are solved:

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} \boldsymbol{u} = -\boldsymbol{\nabla} P + \left(\frac{Pr}{Ra}\right)^{1/2} \nabla^2 \boldsymbol{u} + \theta \hat{z}, \tag{1}$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0, \tag{2}$$

$$\frac{\partial \theta}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} \theta = \frac{1}{(PrRa)^{1/2}} \nabla^2 \theta. \tag{3}$$

We use $\boldsymbol{u} = u(\boldsymbol{x}, t)$ as the velocity vector with streamwise, spanwise and wallnormal components. θ is the non-dimensional temperature ranging from $0 \leq \theta \leq 1$. The simulations are performed in a computational box with periodic boundary conditions in streamwise and spanwise directions and confined by a heated plate below and a cooled plate on top. The shearing of the flow is implemented by a Couette flow setting where both top and bottom plates of the flow are moved in opposite directions with the speed u_w keeping the average bulk velocity at zero and therefore minimizing dissipation errors. The domain size is $(L_x \times L_y \times L_z) = (9\pi h \times 4\pi h \times h)$ using a grid of $(n_x \times n_y \times n_z) = (6912 \times 3456 \times 384)$ which is homogeneously distributed in the streamwise and spanwise directions and clustered with an error-function $\eta(\xi) = erf(2\xi)/erf(2)$ in wallnormal direction.

The open source finite-difference Navier-Stokes solver *AFiD* [12] was written in Fortran 90 to study large-scale wall bounded turbulent flow simulations. In collaboration with NVIDIA, USA, the code was ported in its newest version to a GPU setting using an MPI and CUDA Fortran hybrid implementation optimized to run and solve large flow fields [13].

For this visualization showcase, data from Blass et al. [3] was used, where a parameter study over different input parameters was conducted to study the influence of such to the flow field. Control parameters were the temperature difference between the top and bottom plates as the strength of the thermal forcing, non-dimensionalized as the Rayleigh number Ra, and the wall velocity as the strength of the shear forcing, non-dimensionalized as the wall shear Reynolds number Re_w, while keeping the Prandtl number Pr, which is the ratio of thermal and viscous viscosity of a fluid, at unity [3].

In Fig. 2 we present snapshots of temperature fields at mid height in different flow regimes. It can be observed that the flow passes from a thermally dominated regime with large plumes (Fig. 2a) into a regime where the mechanical forcing is dominant. Here, large-scale meandering structures can be observed (Fig. 2d). To transition between the regimes, the flow has to pass a transitional stage, in which the thermal plumes get stretched into large streaks (Fig. 2b). If the shearing is further increased, these streaks become instable and start meandering into the final flow state (Fig. 2c).

In turbulent flows it is very important to research how certain characteristic parameters are influenced by the flow. In thermal convection, the heat transfer, non-dimensionally defined through the Nusselt number Nu is a good indicator if changing flow structures have a supporting effect or may disrupt a previously transport-favorable flow situation.

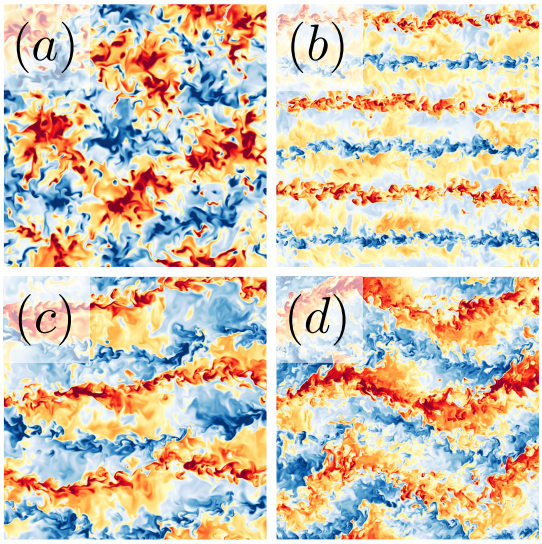

Figure 2: Zoomed snapshots of temperature fields of a sheared and thermally forced flow transitioning through all flow regimes for $Ra = 2.2 \times 10^6$ and (a) $Re_w = 0$, (b) $Re_w = 2000$, (c) $Re_w = 3000$, (d) $Re_w = 6000$, ranging from θ_{min} (blue) to θ_{max} (red).

While two-dimensional visualizations are very helpful in understanding the behavior of the large-scale structures, they don't show the complete scientific picture. They give a good indication of the flow behavior, but to understand thermal turbulence, it is vital to see the whole flow field and the dynamic interaction of turbulent structures with each other. The opportunity to observe the flow evolving and transitioning through different regimes is a great chance to not only statically observe different flow states at fixed locations in space, but to also actually follow the flow on its path to develop thermal plumes, streaks and meandering structures.

It has been previously shown in thermal convection that the large thermal plumes can be traced until very close to the heated and cooled plates [14]. So it is very important to also observe the emergence of structures close to the boundary layer. The detailed visualizations we presented allow us to not only follow the large-scale structures, but also the interaction of small-scale structures much closer to the plates (Fig. 3).

3. Volume Rendering Libraries and Setup

We use ParaView v5.6.0, a world-class, open source, multi-platform data analysis and visualization application installed on Piz Daint. Piz Daint, a hybrid Cray XC40/XC50 system, is the flagship supercomputer of the Swiss National HPC service. We have deployed and tested several solutions within ParaView where parallelism is expressed at different degrees: data-parallel visualization pipelines with GPU-based renderings, or multi-threaded parallelism for software renderings.

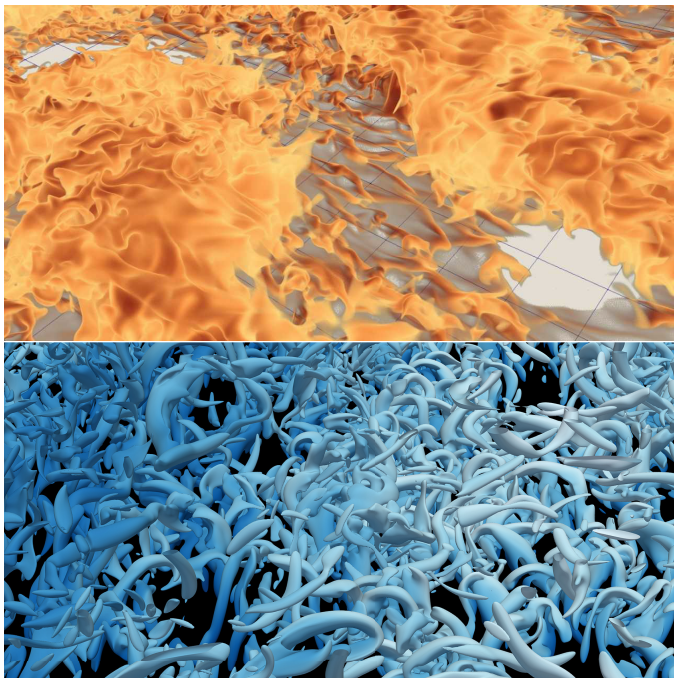

Figure 3: Zoom of an snapshot of the temperature field (top) and the vorticity structures (bottom) at $Ra = 2.2 \times 10^6$ and $Re_w = 6000$.

The computational domain used for our simulations is made of $6912 \times 3456 \times 384$ grid points. The temperature scalar field stored as *float32* takes 36 GB of memory, an overwhelming size to handle on a normal desktop. Using different parallel programming paradigms has enabled us to provide an engaging environment to promote interactive tuning of visualization options and high productivity for movie generation.

Visualization of three-dimensional scalar fields is a very mature field. Many techniques are available to make some sense of the three-dimensional nature of the data, and its variations throughout the volume. Surface-based renderings with isosurface thresholds or slicing planes have a great appeal in that they are easy to use, and provide unambiguous representations based on clearly defined numerical values. Volume renderings, early applied to medical applications, are also a great fit for scalar visualizations, especially in the realm of time-dependent outputs. They are, however, much more difficult to use. Volume rendering is based on the principle of converting a 3D scalar field onto an RGB (color) volume and an Opacity volume. Transfer functions, often defined in an ad-hoc manner, convert scalar values to colors, and classify the data into regions of different opacities. A volume can then appear as clouds with varying density and color. Their interpretation remains subjective to the user's taste and practice. We refer readers to other sources [15] to dive more deeply into the principles of Volume Rendering.

Volume Rendering can be implemented in different manners. ParaView was chosen because it offers a testbed for several state-of-the-art implementations which can be selected based on rendering parameters and available hardware.

The largest partition of the Piz Daint supercomputer has nodes equipped with one Intel Xeon E5-2690 (12 cores, 64 GB RAM) and one NVIDIA Tesla P100 GPU (16 GB RAM, OpenGL driver 396.44). Thus our priority is to evaluate the GPU-based implementations. ParaView's default installation enables also a software ray caster for rendering volumes but we have found its performance far below the other options. The lack of advanced parameter settings in the Graphical User Interface (GUI) of ParaView also led us to abandon its evaluation. We tested ParaView's native GPU ray casting implementation, against IndeX, an NVIDIA library, as well as OSPRay, a software-based library developed by Intel. Doing so, provides a valid option to users of supercomputers not equipped with GPUs. Our performance evaluation is based on ParaView's benchmarking Python source code[1].

In all cases, we have ignored disk-based I/O costs. There is often quite a bit of variability when running on a large distributed file system shared by hundreds of users. Our motivations are rendering-centered, and two-fold: evaluate the memory cost and resources (CPU, GPU) required to get a first image on the screen, and see if color/opacity transfer function editing, as well as other image tuning can be done in real time, using any of the three methods proposed. In the evaluation of performance costs, ParaView's benchmark code enables fully automated testing with a careful management of double buffering, by saving images from the back buffer to bypass driver optimizations.

In the two GPU-based methods evaluated, we use an EGL-based rendering layer [16] to overcome the need to have a server-side X-Windows server running on the compute node. This enables headless, offscreen rendering with GPU acceleration. We note however that although the GPUs provide phenomenal rendering power, they are limited by the available memory (16 GB on our NVIDIA's Pascal GPUs). For the full size of our simulations outputs, we are actually forced to use data-parallel pipelines on multiple nodes to use the aggregate memory of the different GPUs.

Our third option, however, uses OSPRay and software rendering and does not suffer from a memory hard limit. Compute nodes are easily and often found with large memory banks. We use Piz Daint's high memory nodes with 128 GB of RAM, where our mesh of over 9 billion voxels can be rendered easily on a single node.

3.1. ParaView's GPU Ray Casting

When GPU hardware is present, ParaView's most efficient mapper is a volume mapper that performs ray casting on the GPU using vertex and fragment programs [17]. The core ray-tracing algorithms are coded in GLSL and require a graphics driver supporting at least OpenGL version 3.2 [18]. The data is stored into a vtkVolumeTexture which manages the OpenGL volume texture, its type and internal format. Although this class supports streaming data into separate blocks to make it fit the GPU memory, we have not used this option which imposes a performance trade-off, artificially going over the fixed GPU

[1] source code found in ./Wrapping/Python/paraview/benchmark/

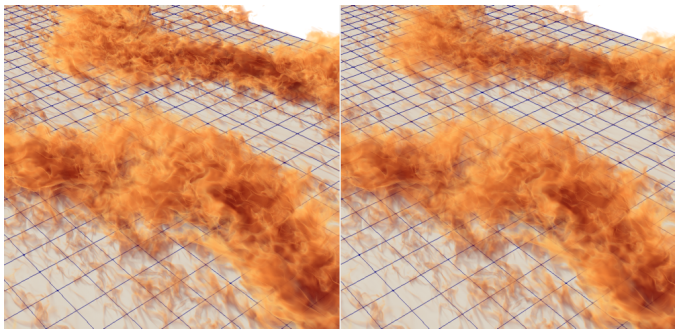

Figure 4: Volume renderings of temperature with ParaView's OpenGL GPU RayCastMapper (left), and with NVIDIA IndeX (right).

memory limit. Block streaming, sometimes called data bricking may also suffer from artifacts at the block boundaries where gradient computations are done to support shading. ParaView's OpenGL VolumeRayCastMapper binds the 32-bit float scalar field array to a three-dimensional texture image of type GL_FLOAT with a call to glTexImage3D(). An explicit Texture object is created, transfering data from CPU memory to GPU memory. The GPU's memory and the CPU to GPU transfer bus set the maximum achievable performance.

3.2. NVIDIA IndeX

NVIDIA IndeX [19] is a three-dimensional volumetric visualization SDK developed to enable the visualization of massive data sets. NVIDIA has worked in tandem with Kitware to bring an implementation of IndeX to ParaView, and we have enjoyed the benefits of a close partnership between the Swiss National Supercomputing Center (CSCS) and NVIDIA, to be able to use IndeX in a multi-GPU setting. We use the ParaView plugin v2.2 with the core library NVIDIA IndeX 2.0.1 (build 305900.4300.3163). NVIDIA IndeX Accelerated Compute (XAC) interface integrates the core surface and volume sampling programs written in CUDA [20]. For this showcase, we have used the generic programs provided by IndeX, without custom programming. In Fig. 4 we show side-by-side renderings done with the two GPU-based libraries. The NVIDIA IndeX rendering (right) is done with identical color and opacity transfer functions and sampling rates as used in 3.1. ParaView's GPU Ray Casting image (left) is used as reference. Differences of illumination are barely noticable to the human eye.

3.3. Intel OSPRay

OSPRay [21] is a ray tracing framework for CPU-based rendering. It supports advanced shading effects, large models and non-polygonal primitives. OSPRay can distributes "bricks" of data as well as "tiles" of the framebuffer, although in our current implementation, we only use brick subdivisions. The Texas Advanced Computing Center has developed a ParaView plugin which enables us to test the possibility of using a ray-tracing based rendering engine for volumetric rendering. This is the best solution for clusters where no GPU hardware is available.

OSPRay can use its own internal Message Passing Interface (MPI) layer to replicate data across MPI processes and composite the image. This would result in linear performance scaling and supports secondary rays used in ParaView's *path-tracer* mode, but would be prohibitive in terms of communication costs. In this study, we rely on a different parallel computing paradigm. The emphasis is no more on data parallelism, but rather on multi-threaded execution. A complete *software-only* ParaView installation was deployed with an LLVM-based and OpenGL Mesa layer. We used Mesa v17.2.8, compiled with LLVM v5.0.0, and the OSPRay v1.7.2 library to provide a very efficient multi-threaded execution path taking advantage of Piz Daint's alternate partition of compute nodes. These nodes are built with two Intel Broadwell CPUs (2x18 cores and 64/128 GB RAM). We run ParaView with SLURM's option "–cpus-per-task=72 –ntasks-per-core=2", effectively taking full advantage of the multi-threading exposed by the LLVM and OSPRay libraries.

3.4. Parallel Image Compositing

ParaView's default mode of parallel computing is to use data-parallel distribution, whereby sub-pieces of a data grid are processed through identical visualization pipelines. To combine the individual framebuffers of each computing nodes, ParaView uses Sandia National Laboratory's IceT [22] compositing library. We use it in its default mode of operation doing sort-last compositing for desktop image delivery. We note here that NVIDIA's IndeX uses a proprietary compositing library, so for the IndeX tests only, we disable ParaView's default image compositor.

4. Volume Rendering of the Thermal Convection

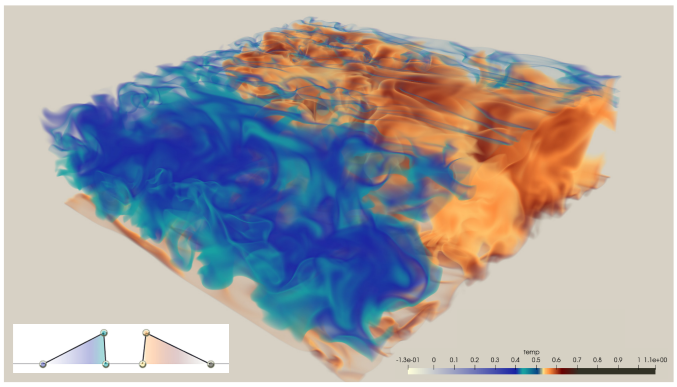

Figure 5: Example of a color and opacity transfer functions to highlight hot and cold plumes.

In visualizing the temperature field, we seek to highlight the turbulence which is best shown by clearly differentiating between cold and hot regions to see how they interact with each other, as seen in Fig. 5. Our movie animation shows an initial phase where region of blue tint is superposed on top of the hotter region. Plumes emerging from the bottom and mixing into the cold regions highlight this phenomenon.

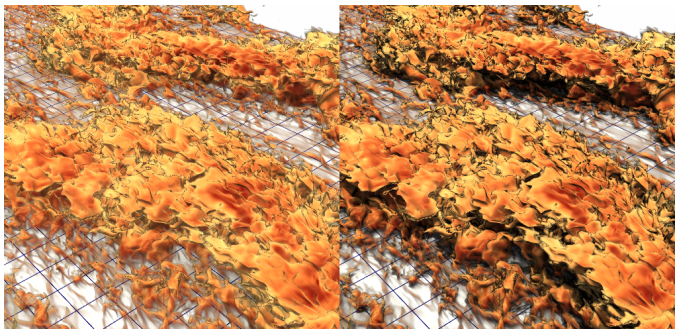

Figure 6: Volume rendering with shading based on gradient estimation (left), and with OSPRay-enabled shadows (right).

4.1. Visual Effects

When presented with multiple visualizations including different illumination and shading, we preferred the renderings which emphasize the amorphous nature of the field data. As can be seen in Fig. 6, shading based on gradient estimation offers little improvement because our data does not have strong gradients, and the use of shadows which at first might seem more appealing, produces images with a strong *surface-like* look, which we discarded upon further analysis.

4.2. GPU-based rendering on a Single Node

Volumetric rendering at this scale of grid has a non-significant cost which we briefly document here. Creating the first frame after data has been read in memory, i.e., the startup cost has a great impact in having users adopt a particular implementation. In a *post-hoc* visualization, data would be read from disk; in an *in-situ* scenario, data might have to be converted to VTK data structures. Thus, we measure performance after the time ParaView has collected all the data and created a bounding-box representation. This startup cost for the first image is also of paramount importance in a movie-making scenario, where data are read from disk, a single image is computed, and the whole visualization pipeline and hardware resources are flushed to visualize the next timestep.

Unlike ParaView's native GPU ray caster implementation which does not enable block streaming, the NVIDIA IndeX library processes data by chunks. However, it does so by bringing volume sub-extents *incrementally* into the GPU memory. Early volume chunks are rendered properly as long as the GPU memory is not exhausted. When memory runs out, late chunks actually corrupt the final image. Our attempts to render a 4 billion voxels dataset on a single node did not succeed with NVIDIA IndeX. We observe failures to allocate 64^3 voxel cubes and the final images are corrupted.

We summarize in Table 1 the time from when volumetric rendering options are enabled, triggering the building of internal structures until the first frame appears. In order to measure the memory cost of all three libraries under evaluation on a single node, we restricted our test sample to a quarter-size domain of the original grid, i.e., 2.28G voxels ($1730 \times 3456 \times 384$), to fit the available GPU RAM. The GPU memory usage[2] settles at 9.1 GB for ParaView native raycaster, and 12.3 GB for NVIDIA IndeX.

Table 1: Initialization and memory costs for a quarter-size domain on one node.

Rendering library	Startup	ParaView task
OSPRay	1.34 secs	18.4 GB
ParaView GPU Mapper	6.17 secs	27.2 GB
NVIDIA IndeX	11.84 secs	39.2 GB

We note both a much higher memory consumption on the application side of ParaView and on the GPU memory side for the NVIDIA IndeX implementation. The high initial setup cost incurred by the NVIDIA IndeX library is due to higher volume transfer between CPU and GPU, a cost that increases further when in parallel, as the current implementation of IndeX triggers re-execution of the data I/O due to larger than usual ghost layer requirements. Work is in progress[3] to minimize this impact in a future version of the plugin.

4.3. CPU-based rendering on a Single Node

If memory costs are substantial, more nodes, and/or more GPUs will be required, increasing the run-time cost of the visualization. Our data domain is quite large, and we are not able to load a half-size domain on a single GPU node. Indeed, both the 64 GB RAM on the node and the 16 GB RAM on the GPU are hard limitations. The OSPRay-based software rendering is one way to alleviate this problem. We can load the full size domain on a single node of the multi-core partition of Piz Daint with dual-Xeon chips and 128 GB of RAM. We measured again the startup cost for the first image at full HD resolution (1920x1080 pixels), using 72 execution threads and found them to increase linearly with grid dimensions. We tested the quarter-size, half-size and the full domain and report the delivery of the first image in 1.07, 1.50, and 2.33 seconds, respectively. The associated cost in RAM is also linear, at 18.4 GB, 36.5 GB and 73 GB, respectively. Of great interest is OSPRay's management of memory. OSPRay volumes can be stored in two different manners. The first variant named *shared structured volume* matches ParaView's data layout. Version 5.6 of ParaView is the first version where this zero-copy access pattern is used and it provides both a faster startup time and a much lower memory footprint, as compared to previous work. Indeed, we reported earlier on the use of OSPRay's alternate implementation called *block bricked volume* whereby data locality in memory is increased by arranging voxel data in smaller chuncks. This came however at a higher cost, doubling the memory footprint on the CPU [23].

After the first frame has been built, our experience is that smooth interaction is possible with all three libraries tested. In fact, ParaView supports acceleration shortcuts for lower precision renderings during interactive navigation, enabling a comfortable user experience for mouse-driven interaction, with little degradation of quality. Color and opacity transfer functions editing is also interactive and very intuitive.

[2] GPU memory usage is measured with the nvidia-smi diagnostic tool

[3] personal communication with NVIDIA Dev. team

Movie quality renderings on the other hand are done with all level-of-details optimizations turned off and we tested the rendering speed of that particular mode, in a batch production test. We created an OSPRay-based benchmark test to mimic a navigation fly-through in a full resolution domain, starting from an overall view of the full grid, zooming in, rotating the view-point, and finally zooming in to immerse the viewer in the volume. From an initial pixel coverage of about 80%, we then move into the scene such that the viewport is completely covered by active pixels, i.e., all pixel rays hit the volume. Using the OSPRay library which is a CPU-only implementation, rendering times are of course dependent on frame buffer pixel resolution. We rendered this short movie segment at three different pixel resolution, WXGA (1280x800 pixels), Full HD (1920x1080 pixels) and 4K Ultra HD (3840x2160 pixels), to evaluate the impact of pixel resolution on rendering costs. We also evaluated the use of hyper-threading to further boost performance. Table 2 summarizes our average rendering time per frame for 300 frames of navigation.

Table 2: Average rendering for the full size domain at different pixel resolution

Pixel Resolution vs. # of threads	36 threads	72 threads
WXGA (1280x800 pixels)	2.85 secs	1.69 secs
FHD(1920x1080 pixels)	3.37 secs	1.90 secs
4K UHD (3840x2160 pixels)	4.81 secs	2.73 secs

Our compute nodes are featuring two sockets with eighteen cores each. We note the clear benefit of using hyper-threading to spawn up to seventy-two threads for an increased throughput. We also note that increasing frame buffer resolution to very large sizes is not a showstopper.

4.4. Rendering the Full Domain in Parallel

We have discussed the fact that the two GPU-based rendering solutions are limited by the available GPU memory, since our 9-billion voxels data set will not fit on a single GPU. To enable our visualization, we must use a parallel set of nodes. We measure again the initial cost for the first image (after all I/O has been done), and also the frame rate achieved in a scripted animation loop. Table 3 summarizes our results, with runs distributed amongst 4, 8 and 12 compute nodes.

Table 3: Initial cost for the first frame and frames/sec for the full size domain

Rendering library	Startup (secs)	frames/sec
4-node GPU RayCaster	8.2 secs	0.19
8-node GPU RayCaster	4.6 secs	0.27
12-node GPU RayCaster	2.9 secs	0.44
4-node NVIDIA IndeX	18.9 secs	0.56
8-node NVIDIA IndeX	12.4 secs	1.48
12-node NVIDIA IndeX	9.8 secs	1.80

As expected, startup times decrease almost linearly with the number of compute nodes, since less data is transfered over from CPU memory to GPU memory. Our animation benchmark loads a single timestep of data, thus, once the data has migrated to the GPU, there is hardly any CPU to GPU communication apart from a single frame buffer image, and we see *frames per second* rates grow somewhat linearly since there is less workload per GPU. In a movie production setting where all timestep outputs are read once, rendered once and are then discarded, the startup cost of any rendering library needs to be weighted against the I/O costs. Although our data I/O statistics show quite a bit of variation because of the high load of our multi user system with over 5000 compute nodes, our simulation data are read, *in average*, in about 32 seconds (resp. 16, and 25 seconds) on 4 nodes (resp. 8 and 12 nodes). We see that the initialization of the rendering sub-system has a greater impact than suspected, and that in an *in-situ* scenario, it would be the singlemost important barrier to performance. The initialization of the NVIDIA IndeX has the most significant bottleneck. Discussion with the NVIDIA developers is on-going and our hope is that this will be much improved in future versions of the SDK since the library is still a very new development.

5. Summary and Conclusion

We have discussed three implementations of volume rendering for a thermal convection simulation output of substantial size. Our time-dependent output is stored as a *float32* array of 36G bytes per timestep. This is a non-trivial size for the most common GPUs. This leaves the scientist with two options: 1) use a data-parallel visualization application with GPU-assisted rendering, or 2) use a *software-only* visualization environment which can fit on compute nodes where large memory banks are easily found. Our choice was to deploy a single application, the open-source ParaView, thanks to its support for different parallel execution paradigms, and for its ability to work with different off-screen and on-screen rendering backends. Having a single application, driven by fully automatized python scripts and a benchmarking suite of tools available in ParaView itself, enabled us to confront all possible implementations with reduced variability.

We tested two GPU-based rendering options. We first used ParaView's native volume rendering which has proved to offer the best compromise between startup time, and interactive performance; We also tested an alternative solution based on a new library in current development by NVIDIA. In our current setup, the IndeX library offers superior interactive rendering, however at non-negligible initialization costs.

We evaluated an implementation of volume rendering provided by the Intel OSPRay library, a software-based framework which can take remarkable advantage of a multi-threaded execution layer. This also fit very well on a subset of our available hardware, a dual-Xeon based compute node without GPU. Our experience is of interest for quite a few computer platforms around the world where graphics hardware is not available.

Our emphasis in creating the scientific visualization shown in the accompanying video [11] was two-fold. First, having an interactive environment enabling us to prototype the visualization with large scale data. The edition of color and opacity transfer functions is the most demanding step in deriving the

proper visualization, and we were able to provide an interactive setup using either hardware-based, or software-based volume rendering. Dealing with long time-dependent simulations outputs was the second requirement, and the path to achieve high productivity was to use parallel and scalable I/O routines. We used VTK's native XML partitioned file format convention for cartesian image data. This was primordial for a quick turnaround time. However, we were unable to mix the two parallel processing paradigms we tested. The ability to get multithreaded rendering with OSPRay is not available in conjunction with MPI-based ParaView's batch execution for faster I/O. The compromise for movie production was to use small subsets of GPU nodes with ParaView's native volume renderer.

The volume rendering framework deployed to analyse our large-grid simulations gives us a unique chance to observe sheared thermal convection in a very simple system with very complex impact to our future. Further, the visualizations allow us to have a very good first insight into the interplay between thermal convection and flow shearing by different kinds of wind and flow currents. We are now able to better understand the emergence and behavior of flow structures transporting heat through the system and affecting the flow dynamics. Going forward, it is the scientific duty of our community to further develop this approach and overcome more and more computational limitations on the pursuit to better understand the nature of our planet.

Acknowledgments

Alexander Blass was financially supported by the Dutch Organization for Scientific Research (NWO-I) and conducted his simulations at the Swiss National Supercomputing Center, under compute allocations s713 and s802. We would also like to acknowledge the support from the Dutch national e-infrastructure of SURFsara, a subsidiary of the SURF cooperation, and the Priority Programme SPP 1881 Turbulent Superstructures of the Deutsche Forschungsgemeinschaft. The authors thank the ParaView development team at Kitware, USA, for fruitful discussions and motivational material. Dave DeMarle has been particularly helpful in discussion related to the OSPRay plugin. Mahendra Roopa at NVIDIA has also been extremely receptive to our feedback and instrumental in helping us get the best of the IndeX library in a multi-GPU setting. We also would like to thank Paul Melis from SURFsara for valuable input to our video [11].

References

[1] Intergovernmental Panel on Climate Change, Ocean Systems, 2014, pp. 411–484.

[2] J. Ahrens, B. Geveci, C. Law, Paraview: An end-user tool for large data visualization.

[3] A. Blass, X. Zhu, R. Verzicco, D. Lohse, R. J. A. M. Stevens, Direct numerical simulations of turbulent sheared thermal convection, in prep. for J. Fluid Mech.

[4] S. Rahmstorf, The thermohaline ocean circulation: A system with dangerous thresholds?, Climatic Change 46 (2000) 247–256.

[5] S. Manabe, R. J. Stouffer, Two stable equilibria of a coupled ocean-atmosphere model, J. Climate 1 (1988) 841–866.

[6] J. Marshall, F. Schott, Open-ocean convection: Observations, theory, and models, Rev. Geophys. 37 (1) (1999) 1–64.

[7] G. Ahlers, S. Grossmann, D. Lohse, Heat transfer and large scale dynamics in turbulent Rayleigh-Bénard convection, Rev. Mod. Phys. 81 (2009) 503.

[8] D. Lohse, K.-Q. Xia, Small-scale properties of turbulent Rayleigh-Bénard convection, Annu. Rev. Fluid Mech. 42 (2010) 335–364.

[9] A. J. Smits, B. J. McKeon, I. Marusic, High-Reynolds number wall turbulence, Ann. Rev. Fluid Mech. 43 (2011) 353–375.

[10] D. Barkley, L. S. Tuckerman, Mean flow of turbulent-laminar patterns in plane Couette flow, J. Fluid Mech. 576 (2007) 109–137.

[11] J. M. Favre, A. Blass, Volume renderings of sheared thermal convection [video file].
URL `https://youtu.be/yEj83O3hVv4`

[12] E. P. van der Poel, R. Ostilla-Mónico, J. Donners, R. Verzicco, A pencil distributed finite difference code for strongly turbulent wall-bounded flows, Computers & Fluids 116 (2015) 10–16.

[13] X. Zhu, E. Phillips, V. S. Arza, J. Donners, G. Ruetsch, J. Romero, R. Ostilla-Mónico, Y. Yang, D. Lohse, R. Verzicco, M. Fatica, R. J. A. M. Stevens, AFiD-GPU: a versatile Navier-Stokes solver for wall-bounded turbulent flows on GPU clusters, Comput. Phys. Commun. 229 (2018) 199–210.

[14] R. J. A. M. Stevens, A. Blass, X. Zhu, R. Verzicco, D. Lohse, Turbulent thermal superstructures in Rayleigh-Bénard convection, Phys. Rev. Fluids 3 (2018) 041501.

[15] Will Schroeder, Ken Martin, Bill Lorensen, The Visualization Toolkit, 2006, pp. 213–244.

[16] Egl eye: Opengl visualization without an x server, `http://tinyurl.com/ybmnzdtv`.

[17] Volume rendering improvements in vtk, `https://blog.kitware.com/volume-rendering-improvements-in-vtk`.

[18] Shaders in vtk, `https://www.vtk.org/Wiki/Shaders_In_VTK`.

[19] Nvidia index, `https://developer.nvidia.com/index`.

[20] R. Haas, P. Mosta, M. Roopa, A. Kuhn, M. Nienhaus, Programmable interactive visualization of a core-collapse supernova simulation, in: Conference on High Performance Computing Networking, Storage and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018.

[21] Ospray: a ray tracing based rendering engine for high-fidelity visualization, `http://www.ospray.org/index.html`.

[22] K. Moreland, W. Kendall, T. Peterka, J. Huang, An image compositing solution at scale, in: Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011, pp. 25:1–25:10. doi:10.1145/2063384.2063417.
URL `https://doi.org/10.1145/2063384.2063417`

[23] J. M. Favre, A. Blass, Volume renderings of sheared thermal convection, in: Conference on High Performance Computing Networking, Storage and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018.