Volume Renderings 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

We present visualizations of sheared thermal convection in large scale wall-bounded turbulent flow simulations. The visualization is 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[1] 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, Naviers-Stokes Solver, Direct Numerical Simulation, Computational Fluid Dynamics

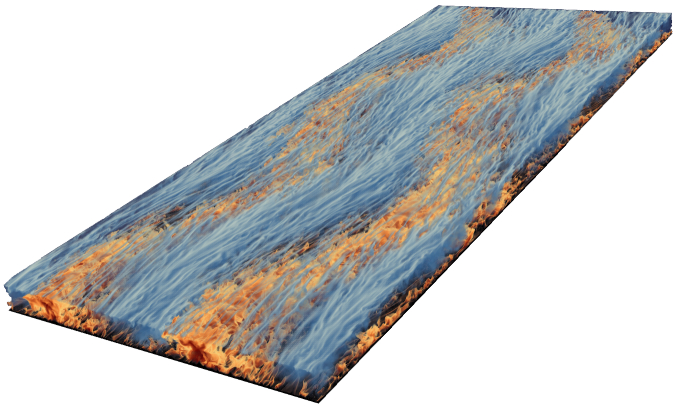

Figure 1: Instantaneous snapshot of the three-dimensional temperature field of sheared thermal convection at $Ra = 4.6 \times 10^6$ and $Re = 6000$ [2].

1. Introduction

Oceans play a big role in the nature of our planet. About 70% of our earth is covered by water. Strong currents are transporting warm water around the world and therefore don't only make life possible, but also allow us to harvest its power producing energy. But humanity tends to easily forget that oceans also yield 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. One has to treat the ocean with respect and carefulness, but it is unambiguously clear that humanity and its science is just attempting to reach the first step on an oceanic staircase to understanding nature.

2. Numerical Simulations

A more fundamental setup of this natural mechanism is sheared thermal convection. Many processes in nature can be based on heat and momentum transfer and therefore interaction between buoyancy and shear. Direct numerical simulations (DNS) were performed with the second-order finite-difference code *AFiD* [3] 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 sourced finite-difference Navier-Stokes solver *AFiD* [3] was written in Fortran 90 to study large-scale wall

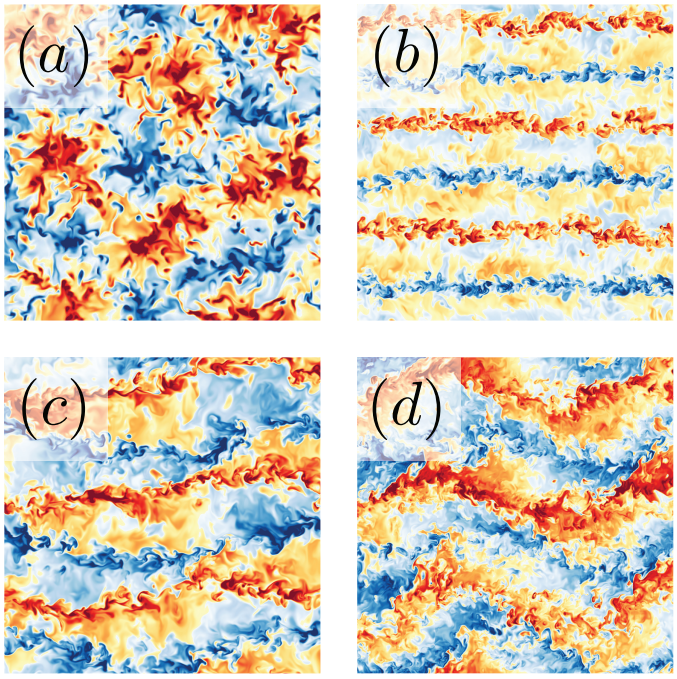

Figure 2: Zoomed instantaneous snapshots of temperature fields of a sheared and thermally forced flow transitioning through all flow regimes for $Ra = 2.2 \times 10^6$ and (a) $Re = 0$, (b) $Re = 2000$, (c) $Re = 3000$, (d) $Re = 6000$.

bounded turbulent flow simulations. In collaboration with NVIDIA, USA, the code was ported in its newest version to a GPU setting using a MPI and CUDA Fortran hybrid implementation optimized to run and solve large flow fields [4].

For this visualization showcase, data from Blass et al. [2] 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 (Eqn. 4), and the wall velocity as the strength of the shear forcing, non-dimensionalized as the wall shear Reynolds number (Eqn. 5), while keeping the Prandtl number (Eqn. 6), which is the ratio of thermal and viscous viscosity of a fluid, at unity. The equations read:

$$Ra = \frac{\beta g h^3 \Delta T}{\kappa \nu}, \tag{4}$$

$$Re_w = \frac{h u_w}{\nu}, \tag{5}$$

$$Pr = \frac{\nu}{\kappa}, \tag{6}$$

where β is the thermal expansion coefficient of the fluid, g the gravitational acceleration, κ the thermal diffusivity, ν the kinematic viscosity of the fluid, h the distance between the plates and u_w the wall velocity.

In Fig. 2 we present instantaneous 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).

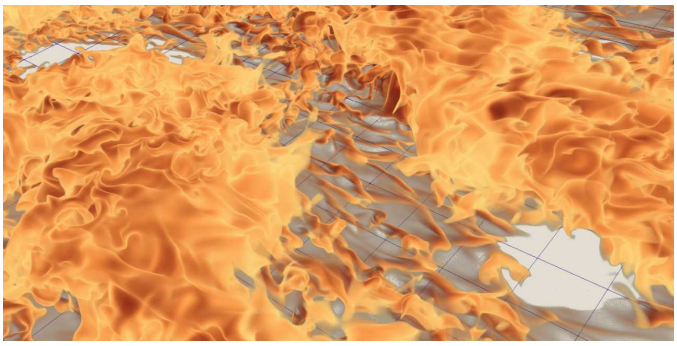

Figure 3: Zoom of an instantaneous snapshot of the temperature field at $Ra = 2.2 \times 10^6$ and $Re = 6000$.

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

$$Nu = \frac{Qh}{\kappa \Delta \theta} \tag{7}$$

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 dynamic interaction of turbulent structures with each other.

To have the chance to observe the flow while evolving and transitioning through different regimes is a great chance not only to statically observe different flow states at fixed locations in space, but to 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 [5]. So it is very important to also observe the emergence of structures close to the boundary layer. The detailed visualizations presented in this paper 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).

The mesh used in our simulations is made of $6912 \times 3456 \times 384$ grid points. The temperature scalar field stored as *float32* takes 36Gb 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. We use ParaView

v5.5.2, 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.

3. Volume rendering libraries and setup

Visualization of 3D scalar fields is a very mature field. Many techniques are available to get some sense of the 3D nature of the data, and its variations through the volume. Surface-based renderings with isosurface thresholds or slicing planes have a great appeal in that they are easy to use, but volumetric renderings, first made popular in medical applications, are also a great fit for scalar visualizations, especially in the realm of time-dependent outputs.
ParaView's "Smart" VolumeMapper is an image mapper that will delegate to a specific volume mapper based on rendering parameters and available hardware. It is a convenient wrapper to test multiple options.
The largest partition of the Piz Daint supercomputer nodes is equipped with one Intel Xeon E5-2690 (12 cores, 64 Gb RAM) and one NVIDIA Tesla P100 GPU (16 Gb RAM), so our priority is to evaluate the GPU-based implementations. ParaView "Ray Cast Only" mode is also available but we have found its performance far below the other options, and we discarded its use. We aim to test ParaView's GPU ray casting implementation, against an NVIDIA-developed custom library called IndeX, and against a CPU-only solution, to give a valid option to users of supercomputers not equipped with GPUs. Our performance evaluation is based on ParaView's benchmarking Python source code[1].
For all methods used, 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-centric, 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 automatized testing with a careful management of double buffering, saving images from the back buffer to bypass driver optimizations.

In the two GPU-based methods evaluated, we use an EGL-based rendering layer, relaxing 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 compute power, they are limited by the available memory (16Gb 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.

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

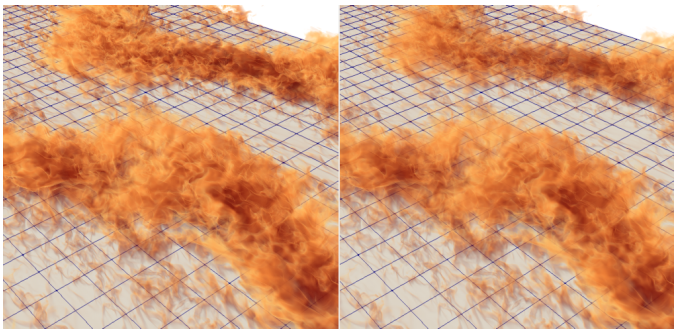

Figure 4: Volume renderings of temperature with ParaView's Smart Shader (left), and with NVIDIA IndeX (right)

3.1. ParaView's GPU Ray Casting

ParaView's most efficient mapper - if the GPU hardware supports it - is a volume mapper that performs ray casting on the GPU using fragment programs. Care must be taken when evaluating and comparing the different approaches. We highlight the fact that in this preferred mode, Paraview converts the 32-bit float data to 16-bit integer. In Fig. 4 (left), we note very small differences of illumination with the NVIDIA IndeX-rendered image at full 32-bit resolution, but no degradation due to down-converted data.

3.2. NVIDIA IndeX

NVIDIA IndeX[6] is a commercial 3D volumetric visualization SDK developed to enable the visualization of massive datasets. 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 v2.1 with the core library NVIDIA IndeX 1.5 (build 299900.2384). In Fig. 4 (right), an NVIDIA IndeX rendering is done with identical color and opacity transfer functions as used in 3.1. Small differences can be accounted for a non uniform interpretation of the sampling rates for the scalar opacity unit distance as coded in the three proposed implementations.

3.3. Intel OSPRay

OSPRay[7] 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.

In the case of OSPRay, 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 v1.6.1 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[8] 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 propriatary compositing library, so for the IndeX tests only, we disable ParaView's default image compositor.

4. Volume rendering of the thermal convection

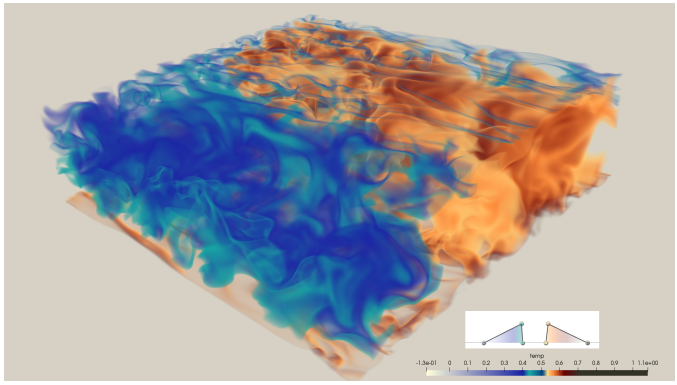

Figure 5: Example of a color and opacity transfer functions to highlight hot and cold plumes.

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.

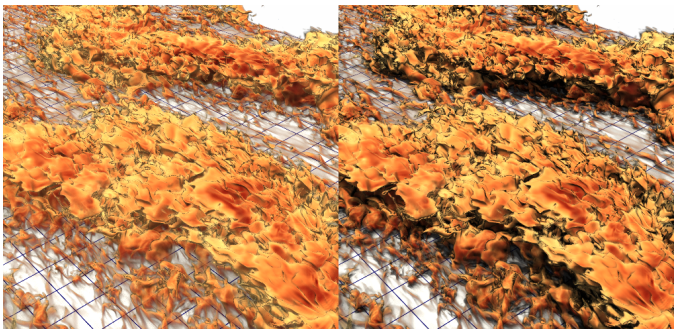

Figure 6: Volume rendering with shading based on gradient estimation (left), and with OSPRay-enabled shadows (right)

4.1. Visual effects

When presented with multiple visualizations including different illumination and shading, the scientists actually 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. Rendering costs 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.
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. To be able to measure the memory cost on a single node, we restricted our test sample to a quarter-size domain of the original grid, i.e. 2.28G voxels ($1730 \times 3455 \times 384$).

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

Rendering library	Startup	pvserver task
OSPRay	2.1 secs	27.6Gb
ParaView GPU Mapper	7.7 secs	25.8Gb
NVIDIA IndeX	15.7 secs	21.1Gb

The high initial setup cost incurred with the NVIDIA IndeX library is due to higher volume transfer between CPU and GPU, a cost that increases further when in parallel mode, as the current implementation of IndeX will trigger re-execution of the

data I/O because of larger than usual ghost layer requests. Work is in progress[2] to minimize this impact in a future version of the plugin.
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 indeed, and wew are not able to load a half-size domain on a single GPU node. Indeed, both the 64Gb RAM on the node and the 16Gb 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 128Gb of RAM. We measured again the startup cost for the first image 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 2.1, 4.0, and 8.0 seconds, respectively. The associated cost in RAM is also linear, at 27.6Gb, 55.1Gb and 110 Gb, respectively.

After the first frame has been built, our experience is that renderings are done at interactive speed with all three libraries tested. Color and opacity transfer functions editing is also interactive and very intuitive. We ran scripted navigation scenarios through a static domain loaded in memory at HD (1920x1080 pixels) and 4K UDH (3840x2160 pixels) resolutions.

4.3. 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. A nearly 9G voxels dataset just 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 2 summarizes our findings. As noted earlier, the NVIDIA IndeX library sents the full 32-bit resolution data to the GPU. This leads to the application crashing when using 4 nodes only. Thus, a minimum allocation must contain 8 nodes/GPUs. Conservatively, we may say that 1 giga-voxel per GPUs is a good rule of thumb to do a resource allocation if using NVIDIA IndeX library.

Table 2: Initial cost for the first frame and frames/sec for the full size domain

Rendering library	Startup (secs)	frames/sec
ParaView 4 GPU Mapper	6.3 secs	0.19
ParaView 8 GPU Mapper	3.2 secs	0.30
NVIDIA 4 IndeX	n/a	n/a
NVIDIA 8 IndeX	49 secs	1.48

Finally, we note that with 8 GPUs, the NVIDIA IndeX implementation offers a faster frame/sec rate. The startup cost however remains a big bottleneck. Discussion with the NVIDIA developers are on-going and our hopes is that this will be much improved in future versions of the SDK.

[2]personal communication with NVIDIA Dev. team

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 made available by 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 filter 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 fitted 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 was two-fold. Have 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 we achieved high productivity with a data-parallel execution path. Movie production was done on small subsets of GPU nodes with ParaView's native volume renderer.

Acknowledgments

Alexander Blass was financially supported by the Dutch Organization for Scientific Research (NWO-I) and conducted his simulation runs at the Swiss National Supercomputing Center, under compute allocations s713 and s802. 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.

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

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

[8] K. Moreland, W. Kendall, T. Peterka, J. Huang, An image compositing solution at scale.