Eurographics Symposium on Parallel Graphics and Visualization (2021)
M. Hadwiger, M. Larsen, F. Sadlo (Editors)

Evaluation of PyTorch as a Data-Parallel Programming API for GPU Volume Rendering

N. X. Marshak[1], A. V. P. Grosset[2], A. Knoll[3], J. Ahrens[2], and C. R. Johnson[1]

[1]SCI Institute, University of Utah
[2]Los Alamos National Laboratory, [3]Intel Corporation

Abstract

Data-parallel programming (DPP) has attracted considerable interest from the visualization community, fostering major software initiatives such as VTK-m. However, there has been relatively little recent investigation of data-parallel APIs in higher-level languages such as Python, which could help developers sidestep the need for low-level application programming in C++ and CUDA. Moreover, machine learning frameworks exposing data-parallel primitives, such as PyTorch and TensorFlow, have exploded in popularity, making them attractive platforms for parallel visualization and data analysis. In this work, we benchmark data-parallel primitives in PyTorch, and investigate its application to GPU volume rendering using two distinct DPP formulations: a parallel scan and reduce over the entire volume, and repeated application of data-parallel operators to an array of rays. We find that most relevant DPP primitives exhibit performance similar to a native CUDA library. However, our volume rendering implementation reveals that PyTorch is limited in expressiveness when compared to other DPP APIs. Furthermore, while render times are sufficient for an early "proof of concept", memory usage acutely limits scalability.

CCS Concepts

• Human-centered computing → Scientific visualization; • Computing methodologies → Parallel computing methodologies; Rendering;

1. Introduction

Data-parallel programming (DPP) has been explored for many parallel visualization applications [MAGM11, MAPS12, LLN*15, LMNC15, SM15, MSU*16]. This is because DPP provides powerful abstractions that allow parallel algorithms to be expressed not only concisely but also independently of the underlying hardware architecture. It is thus desirable to have an API for DPP that enjoys wide adoption and that is likely to continue to improve in the future. In addition, a number of visualization researchers and domain scientists have gravitated towards tools like Python and its GPGPU extensions because they can reduce the amount of labor required for software implementation. Finally, machine learning frameworks like PyTorch [PGM*19] and TensorFlow [ABC*16] expose powerful data-parallel primitives, feature both GPGPU and distributed capabilities, and have exploded in popularity in recent years.

Therefore, the time is ripe to evaluate PyTorch as both as an API for DPP, and as a way to implement data-parallel scientific visualization algorithms. In this paper, we present the following contributions:

- Preliminary evaluation of PyTorch as a data-parallel API.
- Application of DPP in PyTorch to implement a "proof of concept" volume renderer for scientific visualization.
- Performance comparison of two distinct data-parallel formulations of volume rendering, implemented in PyTorch, to each other and to off-the-shelf scientific visualization tools.

2. Related Work

DPP has its roots in the seminal work of Blelloch [Ble90], which presented a programming paradigm in which algorithms are expressed in terms of *data parallel primitives*, that by definition can apply an operation to a size-N array in $O(\log N)$ time, assuming an unlimited number of processors. Such primitives are intended to "abstract away" implementation details such as scheduling, synchronization, and communication that are specific to each parallel/distributed architecture being targeted.

The well-known Thrust library for CUDA provides data-parallel primitives and is heavily influenced by the DPP paradigm [BH12]. Within visualization, the VTK-m library exposes a data-parallel C++ API in order to facilitate large-scale distributed rendering [MSU*16]. Copperhead, a DPP GPGPU library for Python [CGK11], is unfortunately no longer actively maintained. In this work, we focused on PyTorch [PGM*19] because of its massive popularity, high-level interface to the GPU (via Python), built-in data-parallel operators, and volume sampling routines. In

DOI: 10.2312/pgv.20211041

theory, the same algorithms could be implemented using any other library that exposes the same data-parallel primitives.

Outside of the DPP field, there exists a large amount of work on scientific visualization in high level languages, most of which focuses on binding GPU rendering to languages like Python. For instance, VTK [SMLK06], ParaView [AGL05], and `yt` [TSO*10] include GPU volume renderers, and have a Python interface. There have also been a number of attempts to integrate GPU-based rendering into Jupyter Notebook. For instance, VisPy [CKL*15], ipygany/K3D-jupyter [Qua, KTGK] and itkwidgets [Ins] bind OpenGL, WebGL, and VTK to Jupyter, respectively.

This paper is most closely related to computer vision papers that introduce data-parallel volume rendering in either PyTorch or TensorFlow. Lombardi et al. [LSS*19] use the "array of rays" (AoR) formulation presented in Section 4.2, whereas Henzler et al. [HMR19] and Mildenhall et al. [MST*20] employ the "scan+reduce" (SR) approach described in Section 4.1. Similar techniques are currently implemented in the latest release of the PyTorch3D library [RRN*20]. In addition, Liu et al. [LGL*20] explore sparse voxel octree acceleration. However, these papers do not explore applications to scientific visualization, and provide only a cursory presentation, if any, of their data-parallel formulation. Furthermore, to our knowledge, the computer vision community has not benchmarked the rendering performance of AoR against SR, nor have they compared against off-the-shelf volume renderers for visualization.

Within scientific visualization, Schroots and Ma [SM15] and Larsen et al. [LLN*15] use data-parallel primitives for direct volume rendering, built on top of the Dax [MAGM11] and EAVL [MAPS12] frameworks, respectively. These efforts continue as part of the VTK-m library [MSU*16]. Larsen et al. focus on unstructured data, whereas Schroots and Ma use structured volumes. Although we also use a structured grid, we could not use their particular data-parallel formulation because PyTorch does not support a `map<functor>` operation that can apply arbitrary user-defined functions (with encapsulated data) to each element of an array.

3. Background

3.1. Data-parallel operations

Data-parallel operations used in this paper include vectorized arithmetic, scan (a.k.a. "prefix sum"), and reduction. The latter two are defined over a binary operator, e.g., `+`, `*`, `max` or `min`. For the purposes of this paper, let `sum_reduce`, `mult_reduce`, and `mult_scan` denote what Blelloch calls `+-reduce`, `*-reduce`, and `*-scan`, respectively [Ble90].

3.2. Volume rendering

We evaluate the following approximation of the volume rendering integral [EHK*06], where L_0 is the background radiance, and $L(D)$ is the radiance received by the eye. Note that $g_i = g(x_i)\Delta x$ is an approximation of $\int_{x_{i-1}}^{x_i} g(x)\mathrm{d}x$, where $g(s) = L_e(s)\kappa(s)$, κ is the extinction, and L_e is the volumetric emission term.

$$L(D) \approx L_0 \prod_{i=1}^{N}(1-\alpha_i) + \sum_{i=1}^{N} g_i \prod_{j=i+1}^{N}(1-\alpha_j) \tag{1}$$

We define alpha as $\alpha_i := 1 - T_i$, where $T_i = \exp(-\int_{x_{i-1}}^{x_i} \kappa(x)\mathrm{d}x) \approx \exp(-\kappa_i \Delta x)$ denotes the transmittance of the i-th segment.

4. Method Overview

In this work, we elucidate and compare two ways to implement the above via data-parallel primitives:

1. Using scan and reduction (Section 4.1). Denoted as "SR".
2. Using vectorized math operators to parallelize over an array of rays (Section 4.2). Denoted as "AoR".

In computer vision, [HMR19] and [MST*20] use the former approach, and [LSS*19] employs the latter. We provide standalone source code that implements both formulations as part of the supplemental material that accompanies this paper. Unlike the CV use case, we must apply a transfer function, and currently this is performed as a preprocessing step.

4.1. Volume rendering using parallel scan and reduction (SR)

Equation 1 can be evaluated as follows, where α_1 is assumed to be zero, $\odot$ denotes an elementwise product (see Section 3.1 for definitions of `mult_reduce`, `sum_reduce`, and `mult_scan`):

$$L(D) \approx L_0 * \texttt{mult\_reduce}\left(\begin{bmatrix} 1-\alpha_n \\ 1-\alpha_{n-1} \\ 1-\alpha_{n-2} \\ \vdots \\ 1-\alpha_2 \\ 1-\alpha_1 \end{bmatrix}\right) + \texttt{sum\_reduce}\left(\begin{bmatrix} g_n \\ g_{n-1} \\ g_{n-2} \\ \vdots \\ g_2 \\ g_1 \end{bmatrix} \odot \texttt{mult\_scan}\left(\begin{bmatrix} 1-\alpha_1 \\ 1-\alpha_n \\ 1-\alpha_{n-1} \\ 1-\alpha_{n-2} \\ \vdots \\ 1-\alpha_2 \end{bmatrix}\right)\right) \tag{2}$$

We obtain the input arrays, containing the required values of α_i and g_i, by sampling the volume using PyTorch's `grid_sample` function. A sampling grid is precomputed by creating a 3D array of uniformly spaced coordinates, and then applying a "perspective warp" via scaling the sample positions by a factor proportional to their camera space z-coordinate. (See source code included in the supplemental material for details.) Whenever the viewpoint is changed, the sample positions must be updated accordingly by applying an appropriate transformation matrix.

4.1.1. Tiled SR (TSR)

In order to reduce memory consumption, works like [MST*20] split the image into tiles and do SR sequentially over each tile. In this work, we use "TSR" to denote a "tiled" SR implementation that divides the image into four equal-sized quadrants. In the current implementation, the "perspective warp" of the sampling grid is applied once per quadrant at render time, as opposed to being a precompute step like in Section 4.1.

Algorithm 1 Volume rendering for an $m \times n$ image using AoR (Section 4.2). $\mathbb{0}^{m\times n}$, $\mathbb{1}^{m\times n}$, $\mathbf{e}$, $\odot$, and Δx denote an $m \times n$ array of zeros, $m \times n$ array of ones, eye position, elementwise product, and step size, respectively.

$L \leftarrow$ Background radiance
$D \leftarrow$ Ray directions
$T_{min}, T_{max} \leftarrow$ Ray-AABB intersect results
$T_{eval} \leftarrow T_{min}$
$R_{done} \leftarrow \mathbb{0}^{m\times n}$
while $R_{done} \neq \mathbb{1}^{m\times n}$ **do**
    $P \leftarrow \mathbf{e} + T_{eval} \odot D$
    $G \leftarrow g_i$ values (emissive contrib.) at sample positions P
    $A \leftarrow$ alpha values at sample positions P
    $L \leftarrow G + (1-A) \odot L$
    $T_{eval} \leftarrow T_{eval} + \Delta x$
end while

4.2. Volume rendering using "array of rays" (AoR)

The AoR approach, summarized in 1, and utilized by Lombardi et al [LSS*19], resembles a traditional back-to-front compositing loop for volume ray casting. Unlike GLSL or CUDA, however, PyTorch has no notion of individual fragments or thread IDs. Instead, we perform our operations over entire multidimensional arrays. Please refer the supplemental material for additional details.

5. Results and Discussion

Tests were performed on a desktop PC with a NVIDIA RTX 2070 GPU with 8GB of memory, and/or a "headless" server with an RTX 3090 GPU with 24GB of VRAM. Timing in PyTorch was performed using Python's `timeit.timeit` function, and execution time for native CUDA code was benchmarked by using `std::chrono::system_clock` to record "wall clock" timestamps for the start and end of execution. Peak GPU memory usage was measured by monitoring the output of the `nvidia-smi` utility. Finally, using the same benchmarking tools, we compared the above results to those obtained from VTK's OpenGL volume renderer, which was called from its Python API. VTK results were measured on the RTX 2070 only, since our test script must create a window and thus requires a GUI.

5.1. Performance of data-parallel primitives in PyTorch

We first evaluated `sum_scan` and segmented `sum_reduce`, which are standard data-parallel operators (Section 3.1). Segmented reduce was performed over eight equally sized array segments. Addition was the binary operator assumed for both scan and reduce. Finally, while not strictly a DPP primitive, we benchmarked sorting of single-precision floating point numbers. For all above operations, PyTorch was compared against native CUDA implementations provided by the Thrust library [BH12].

Next, consider a scalar c; two vectors $\mathbf{x}$, $\mathbf{y}$; and the operation $\mathbf{y} \leftarrow \mathbf{y} + c\mathbf{x}$, denoted as a "multiply-add". A PyTorch "multiply-add" implementation, based on vectorized arithmetic, was compared against a Thrust implementation that uses `thrust::transform`.

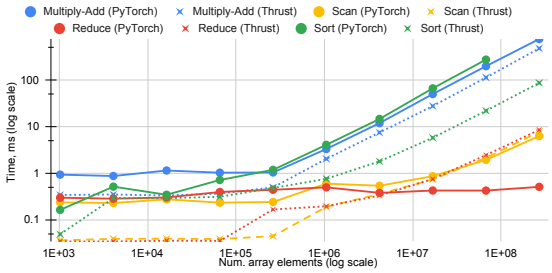

Figure 1: *Time for scan, segmented reduce (8 segments), sort, and 50 repeated multiply-adds. Plot uses a log-log scale. Benchmarks were done on the RTX 2070. Solid lines indicate PyTorch results, while dotted lines indicate Thrust results. A sort at the largest array size could not be performed with PyTorch due to memory limitations.*

Benchmark results can be found in Figure 1. For multiply-add, scan, and segmented reduce, PyTorch performance appears comparable to Thrust at nontrivial array sizes, and this is not surprising because PyTorch calls Thrust implementations of `scan` and `reduce`. In contrast, sort is approximately an order of magnitude slower, and could not be performed at the largest array size.

5.2. Volume Rendering Benchmarks

Most benchmark data came from a simulation of Rayleigh-Taylor instability [CCM04], which is represented as a 1024^3 `float32` uniform voxel grid. We downsampled this volume to produce 512^3, 256^3, and 128^3 versions in order to test scaling with volume size. Since we apply the transfer function as precompute step, we did not measure its execution time. However, the *space* cost of the resulting RGBA volume is included. We compared AoR (Section 4.2), SR (Section 4.1), TSR (Section 4.1.1), and VTK's OpenGL backend, as called from Python.

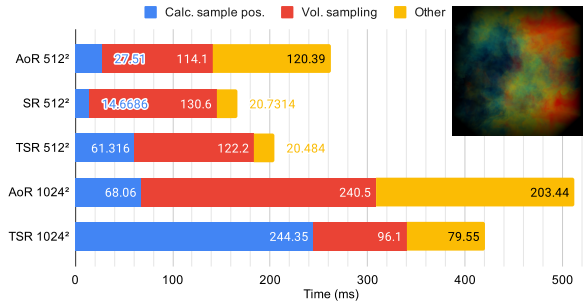

Figure 2: *Performance profiling results for AoR, SR, and TSR, as applied to volume rendering of a 512^3 version of Rayleigh-Taylor instability data (upper right) on the RTX 3090. "Calc sample pos." refers time per frame to compute sample positions, and "vol. sampling" refers to time to query the volume at the necessary sample positions, including time for trilinear interpolation. SR could not be performed at 1024^2 due to memory limitations.*

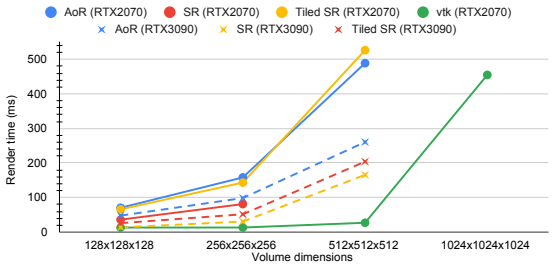

Figure 3: *Dimensions of `float32` uniform grid vs. render time for Rayleigh-Taylor instability data, rendered at 512x512 (Figure 2). SR results could not be recorded at* 256^3 *and above on the 2070, and none the PyTorch-based approaches could render the* 1024^3 *volume due to memory limitations (Figure 4). VTK measurements are for the 2070 only.*

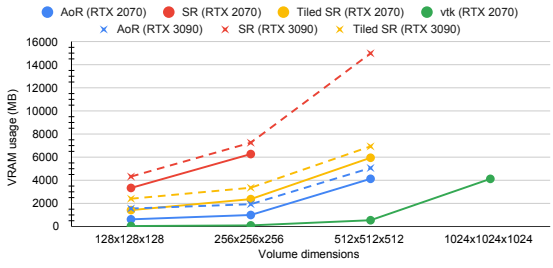

Figure 4: *Grid dimensions vs. memory usage. See Figure 3 caption for benchmark parameters, which are the same here. SR exceeds the 2070's 8GB of memory at* 512^3*, and all PyTorch implementations exceed the 3090's 24GB memory limit at* 1024^3*.*

First, we kept the render resolution fixed to 512x512, while benchmarking render time and memory usage at different volume sizes. It is quite apparent that the PyTorch-based approaches are not memory-efficient (see Figure 4 and caption for details). In fact, none of the PyTorch-based approaches could render the original 1024^3 volume due to memory saturation. By contrast, VTK had no difficulty with this, even on the test GPU with less memory.

We then called the `cProfile` profiler from Python in order to attempt to identify performance bottlenecks when rendering the 512^3 version of the volume, and the results are presented in Figure 2. In particular, our PyTorch implementations tend to spend most of their time either calculating sample positions (which must be performed whenever the viewpoint changes) or sampling the volumes at those positions, with the latter usually being dominant. Interestingly, when using TSR to render at 1024x1024, calculation of sample positions becomes the larger bottleneck.

We performed two additional experiments. First, kept the volume size fixed to 512^3, and varied render resolution. Second, we repeated the `cProfile` measurements, but for the well-known "Magnetic Reconnection" dataset [GLDL14]. Please see the supplemental material for both sets of results.

6. Evaluation and Limitations

Our experiments in (Section 5.1) indicate that individual DPP primitives tend to perform well. (Note that sort does not execute as quickly as Thrust's implementation, and is not as memory efficient.) As a DPP API, PyTorch is somewhat lacking in expressiveness. Recall from the end of Section 2 that we could not adopt the approach of Schroots and Ma [SM15] for unstructured volumes due to PyTorch's lack of a `map<functor>` operator. More broadly speaking, unlike Thrust, VTK-m, and Jax, PyTorch lacks the ability to pass user-defined operators as inputs to data-parallel primitives, precluding straightforward implementation of unstructured DPP volume rendering as proposed by Larsen et al. [LMNC15]. This may also increase the number of required intermediate arrays and/or reads and writes to them, which is one of the main reasons why Thrust supports composition of DPP primitives with user-defined operators [NVI21].

When performing volume rendering with PyTorch's DPP primitives, render times that are within about 1.5 orders of magnitude of the "baseline" set by VTK's volume renderer can be achieved. While this is far from ideal, it is sufficient for a "proof of concept", given that VTK and OpenGL are mature, deliberately engineered solutions.

More problematic, however, is the high memory consumption of all of our PyTorch-based approaches. When rendering an image using SR, we must take a minimum number of samples per pixel *and* per cell. (In this work, we take one sample per pixel and roughly one per cell, but a rigorous implementation would sample at a rate that is strictly above the Nyquist frequency.) Assuming an image size of MxM and volume size of NxNxN, the array of samples given to the `scan` and `reduce` contains $O(M^2N)$ elements. Therefore, it is unsurprising that SR cannot scale beyond small volumes and image sizes.

Using TSR (Section 4.1.1) can reduce memory consumption, but as can be seen in Figures 3, this comes at the cost of performance, since each tile is rendered in serial. AoR's VRAM usage appears to scale better with volume dimension (Figure 4), which is sensible because AoR requires an array of sample positions and a framebuffer, each of which contain $O(M^2)$ elements, to be held in memory in order to render an MxM image. Unlike with SR and TSR, the size of these arrays is independent of the dimension of the volume. However, AoR still consumes far more memory than VTK's OpenGL backend, in part because preclassification forces us to hold an RGBA volume in memory. Furthermore, Figure 3 shows that AoR is not as fast as SR, and this may be due in part to the fact that it does back-to-front compositing in serial, and does not exploit parallelism in the *depth* dimension. Thus, while TSR and AoR consume less memory than SR, they are slower, and only reduce memory consumption so much - recall from Section 5.2 that neither approach is scalable enough to render the original 1024^3 volume.

7. Conclusion and Future Work

In this work, we set out to conduct a preliminary evaluation of PyTorch as a data-parallel API, and to compare two different DPP formulations of volume rendering, as implemented within PyTorch.

We found that the relevant DPP primitives are, for the most part, comparable to Thrust in performance for nontrivial array sizes, with the notable exception of sort. PyTorch is limited in expressiveness because it lacks operations like `map<functor>` that enable composition of user-defined functors and/or binary operators with data-parallel primitives. Furthermore, while rendering speed is reasonable for a "proof of concept", our approach cannot scale to large volumes due to its high memory usage.

Future work includes investigating PyTorch performance in greater detail - e.g., why the bottleneck changes from volume sampling to sample position calculation when TSR rendering resolution is increased from 512x512 to 1024x1024 (Figure 2).

In addition, we could investigate more ways to perform volume bricking/tiling. While we split the image into four tiles, there are clearly more ways in which this could be done, e.g., with more tiles. At least in theory, there could be as many tiles as there are pixels! Some previous authors also "brick" in the z-dimension, as opposed to tiling in image space [LMNC15, SM15], which we did not investigate. We leave a rigorous asymptotic analysis that is aware of the fact that tiles are processed serially, as well as supporting experiments, to future work.

In the future, we may also investigate a broader range of DPP primitives (e.g. scatter and gather), visualization algorithms (e.g. isosurfacing), and programming frameworks (e.g. Jax and Julia).

References

[ABC*16] ABADI M., BARHAM P., CHEN J., CHEN Z., DAVIS A., DEAN J., DEVIN M., GHEMAWAT S., IRVING G., ISARD M., KUDLUR M., LEVENBERG J., MONGA R., MOORE S., MURRAY D. G., STEINER B., TUCKER P., VASUDEVAN V., WARDEN P., WICKE M., YU Y., ZHENG X.: Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283. 1

[AGL05] AHRENS J., GEVECI B., LAW C.: Paraview: An end-user tool for large data visualization. In *The Visualization Handbook*, Hansen C., Johnson C., (Eds.). Academic Press, 2005, pp. 717–731. 2

[BH12] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, Hwu W.-m. W., (Ed.). Elsevier, 2012, pp. 359–371. 1, 3

[Ble90] BLELLOCH G.: *Vector Models for Data-parallel Computing*. AI Series. MIT Press, 1990. 1, 2

[CCM04] COOK A. W., CABOT W., MILLER P. L.: The mixing transition in rayleigh–taylor instability. *Journal of Fluid Mechanics 511* (2004), 333–362. 3

[CGK11] CATANZARO B., GARLAND M., KEUTZER K.: Copperhead: Compiling an embedded data parallel language. In *Principles and Practices of Parallel Programming (PPoPP)* (2011), pp. 47–56. 1

[CKL*15] CAMPAGNOLA L., KLEIN A., LARSON E., ROSSANT C., ROUGIER N. P.: VisPy: Harnessing The GPU For Fast, High-Level Visualization. In *Proceedings of the 14th Python in Science Conference* (Austin, Texas, United States, July 2015), Huff K., Bergstra J., (Eds.). 2

[EHK*06] ENGEL K., HADWIGER M., KNISS J., REZK-SALAMA C., WEISKOPF D.: *Real-Time Volume Graphics*. CRC Press, 2006. 2

[GLDL14] GUO F., LI H., DAUGHTON W., LIU Y.-H.: Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Phys. Rev. Lett. 113* (Oct 2014), 155005. 4

[HMR19] HENZLER P., MITRA N. J., , RITSCHEL T.: Escaping plato's cave: 3d shape from adversarial rendering. In *Computer Vision (ICCV), 2019 IEEE International Conference on Computer Vision* (2019). 2

[Ins] INSIGHT SOFTWARE CONSORTIUM: itkwidgets. URL: https://github.com/InsightSoftwareConsortium/itkwidgets. 2

[KTGK] KOSTUR M., TRZĘSIOK A., GANDOR T., KAŚKOSZ F.: K3d-jupyter. URL: https://k3d-jupyter.org/. 2

[LGL*20] LIU L., GU J., LIN K. Z., CHUA T.-S., THEOBALT C.: Neural sparse voxel fields. *NeurIPS* (2020). 2

[LLN*15] LARSEN M., LABASAN S., NAVRÁTIL P., MEREDITH J., CHILDS H.: Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (Cagliari, Italy, May 2015), pp. 53–62. 1, 2

[LMNC15] LARSEN M., MEREDITH J., NAVRÁTIL P., CHILDS H.: Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium* (Hangzhou, China, Apr. 2015), pp. 279–286. 1, 4, 5

[LSS*19] LOMBARDI S., SIMON T., SARAGIH J., SCHWARTZ G., LEHRMANN A., SHEIKH Y.: Neural volumes: Learning dynamic renderable volumes from images. *ACM Trans. Graph. 38*, 4 (July 2019), 65:1–65:14. 2, 3

[MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.-L.: Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *2011 IEEE Symposium on Large Data Analysis and Visualization* (2011), pp. 97–104. 1, 2

[MAPS12] MEREDITH J. S., AHERN S., PUGMIRE D., SISNEROS R.: EAVL: The Extreme-scale Analysis and Visualization Library. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), Childs H., Kuhlen T., Marton F., (Eds.), The Eurographics Association. 1, 2

[MST*20] MILDENHALL B., SRINIVASAN P. P., TANCIK M., BARRON J. T., RAMAMOORTHI R., NG R.: Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV* (2020). 2

[MSU*16] MORELAND K., SEWELL C., USHER W., LO L.-T., MEREDITH J., PUGMIRE D., KRESS J., SCHROOTS H., MA K.-L., CHILDS H., LARSEN M., CHEN C.-M., MAYNARD R., GEVECI B.: Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications 36*, 3 (2016), 48–58. 1, 2

[NVI21] NVIDIA CORPORATION: *Thrust Quick Start Guide*, 2021. URL: https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf. 4

[PGM*19] PASZKE A., GROSS S., MASSA F., LERER A., BRADBURY J., CHANAN G., KILLEEN T., LIN Z., GIMELSHEIN N., ANTIGA L., DESMAISON A., KOPF A., YANG E., DEVITO Z., RAISON M., TEJANI A., CHILAMKURTHY S., STEINER B., FANG L., BAI J., CHINTALA S.: Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems* (2019), Wallach H., Larochelle H., Beygelzimer A., d'Alché-Buc F., Fox E., Garnett R., (Eds.), vol. 32, Curran Associates, Inc. 1

[Qua] QUANTSTACK: ipygany. URL: https://github.com/QuantStack/ipygany. 2

[RRN*20] RAVI N., REIZENSTEIN J., NOVOTNY D., GORDON T., LO W.-Y., JOHNSON J., GKIOXARI G.: Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501* (2020). 2

[SM15] SCHROOTS H. A., MA K.-L.: Volume rendering with data parallel visualization frameworks for emerging high performance computing architectures. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing* (New York, NY, USA, 2015), SA '15, Association for Computing Machinery. 1, 2, 4, 5

[SMLK06] SCHROEDER W., MARTIN K., LORENSEN B., KITWARE I.: *The Visualization Toolkit: An Object-oriented Approach to 3D Graphics*. Kitware, 2006. 2

[TSO*10] TURK M. J., SMITH B. D., OISHI J. S., SKORY S., SKILLMAN S. W., ABEL T., NORMAN M. L.: yt: A multi-code analysis toolkit for astrophysical simulation data. *The Astrophysical Journal Supplement Series 192*, 1 (2010), 9. 2