Eurographics Symposium on Parallel Graphics and Visualization (2022)
R. Bujack, J. Tierny, F. Sadlo (Editors)

Design and Evaluation of a GPU Streaming Framework for Visualizing Time-Varying AMR Data

S. Zellmann[1], I. Wald[2], A. Sahistan[3], M. Hellmann[4] and W. Usher[5]

[1]Bonn-Rhein-Sieg University of Applied Sciences [2]NVIDIA [3]Bilkent University [4]University of Cologne [5]Intel

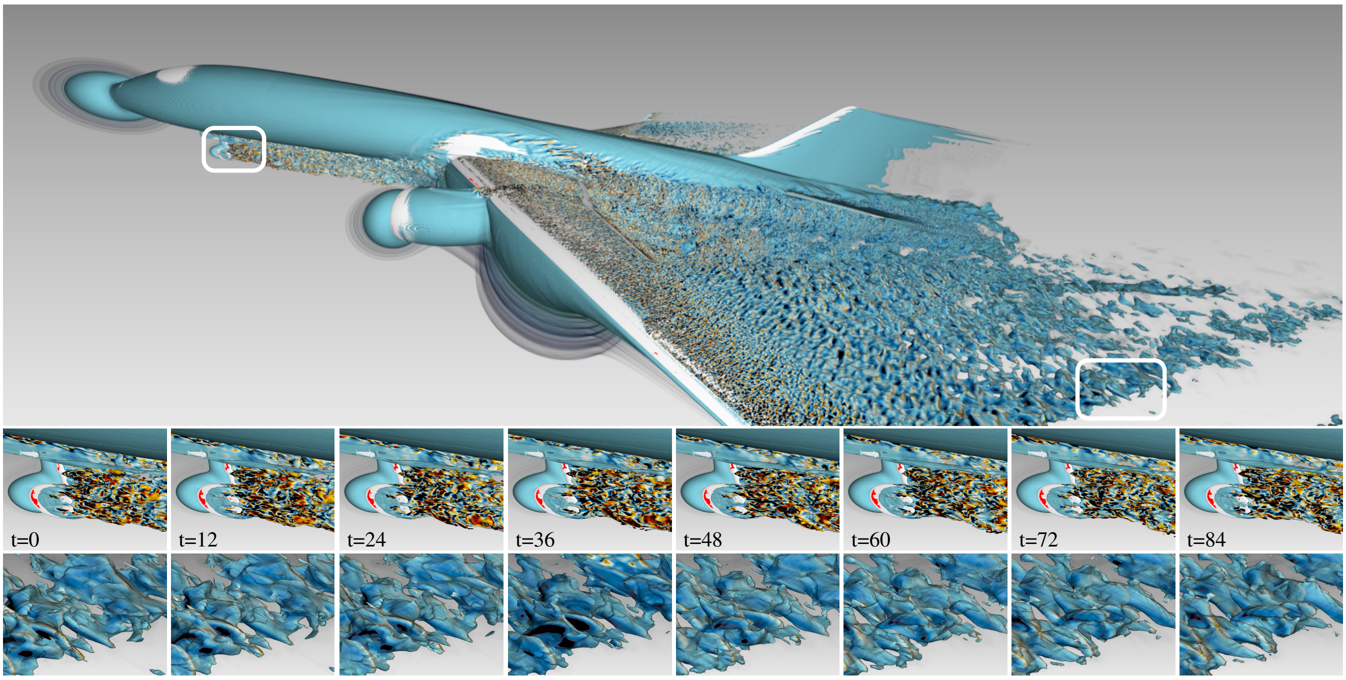


Figure 1: *The NASA Exajet serves as the motivating use case for our study. The large computational fluid dynamics data set was computed using an adaptive mesh refinement (AMR) code and consists of 656 million cells and 423 time steps. Each time step stores 2.5 GB of data per scalar field. At four scalar fields for density and X/Y/Z velocity components, the full time series occupies over 4 TB. Data sets such as these pose significant challenges for interactive visualization on current GPU workstations. We present and evaluate a prototypical framework targeting GPU workstations that asynchronously streams and renders such data sets at interactive rates and with high quality.*

Abstract

We describe a systematic approach for rendering time-varying simulation data produced by exa-scale simulations, using GPU workstations. The data sets we focus on use adaptive mesh refinement (AMR) to overcome memory bandwidth limitations by representing interesting regions in space with high detail. Particularly, our focus is on data sets where the AMR hierarchy is fixed and does not change over time. Our study is motivated by the NASA Exajet, a large computational fluid dynamics simulation of a civilian cargo aircraft that consists of 423 simulation time steps, each storing 2.5 GB of data per scalar field, amounting to a total of 4 TB. We present strategies for rendering this time series data set with smooth animation and at interactive rates using current generation GPUs. We start with an unoptimized baseline and step by step extend that to support fast streaming updates. Our approach demonstrates how to push current visualization workstations and modern visualization APIs to their limits to achieve interactive visualization of exa-scale time series data sets.

1. Introduction

In times of exa-scale computing, simulation codes such as those from computational fluid dynamics (CFD) or from domains like astrophysics have to cope with an increasing disparity between compute power and memory bandwidth. Although memory bandwidth has increased over the past few years, the increase in compute power has far outpaced it, resulting in a widening gap on massively parallel high performance computing architectures [KP21].

DOI: 10.2312/pgv.20221066

As such, physical simulations work to distribute the available memory bandwidth to high-frequency regions of the domain, by using either general finite elements [SSC16], or making use of hierarchical mesh topologies such as Octrees, forests of Octrees, and similar structures, referred to collectively as *adaptive mesh refinement* (AMR) techniques [BO84, BC89]. The Exajet shown in Fig. 1 is one example of the scale and complexity of these large-scale state-of-the-art AMR CFD simulations. A particular property of the Exajet that we exploit in our framework, which slightly simplifies the overall problem, is that the data set's AMR topology does not change with time, only the scalar data changes each step.

A major challenge in visualizing cell-centered AMR data is to interactively render the data at high quality, as the T-junction problem arises when performing smooth interpolation of the data at level boundaries. Furthermore, locating individual cells for reconstruction is a costly operation in itself [WBUK17]. Only recently have researchers addressed some of these issues, proposing data structures optimized for GPUs [WZU*20], though the presented frameworks concentrated on single time steps only. Even if prior frameworks supported animation, data sets at the scale of Exajet do not fit into DDR memory—let alone GPU memory—and would incur significant performance penalties due to the operating system constantly swapping data in and out from mass storage.

To efficiently visualize such large data sets requires a streaming approach that transfers scalar fields from storage over main memory to the GPU, and that in-between efficiently processes the data at different stages to update or rebuild auxiliary data structures used to accelerate rendering. Our experience demonstrates that this is neither cheap nor trivial, as there are multiple different bottlenecks encountered that can easily compound to degrade performance.

In this paper, we describe our experiences with taking the ExaBricks framework by Wald et al. [WZU*20] and step by step extending it to support interactive streaming updates to the scalar field. Our goal is to analyze how much impact optimizations to the streaming subsystem will have on visualizing time-varying AMR data on GPUs. We focus on interactive, high quality rendering, where single frames take on the order of 100 to 200 milliseconds to converge. While aiming for efficient streaming of consecutive frames, whenever possible, we prefer design decisions that allow for random jumps through the animation without excessive setup costs. We describe and analyze the individual challenges and bottlenecks encountered in this process, and present and evaluate a set of techniques to mitigate these issues. The system we target uses a low-latency solid-state drive that competes for PCIe 4.0 bandwidth with the graphics card. Step by step, as we improve the given baseline system we reveal bottlenecks in the reference data structure and imposed by the hardware, and systematically describe how to address them.

2. Related Work

Our paper builds on three fields that we discuss in this section: optimized file I/O and data transfers, scientific visualization of large-scale time-varying data, and research on AMR visualization.

The first field we touch upon is concerned with the hardware/-software interface of streaming of data from mass storage over DDR memory to the GPU. In particular, we are interested in *non-volatile memory express* (NVMe) solid-state drives that are connected directly to the CPU's fabric. The operating system's storage stack is the main bottleneck for accesses to such low-latency storage devices [KLK16]. Work by Koh et al. [KJL*19] evaluated different I/O completion methods and their influence on NVMe and the even newer *ultra-low latency* (ULL) drives, which are directly plugged into a server's memory bus. Accesses to NVMe drives that we concentrate on potentially collide with host-to-device data transfers from the CPU to the GPU. An overview of bandwidth and performance considerations for these types of accesses are discussed by LeBeane et al. [LeB18]. Performance measurements for overlapping data transfers with GPU compute is discussed by Bastem et al. [BUZ*17].

There exists a wealth of literature on time-varying visualization of volumes that are not AMR. Papers from this field concentrate on multiresolution approaches [WGLS05] paired with out-of-core approaches [DCS09]. Ko et al. [KLW*08] used video-based compression to compress an Octree structure in a preprocess and decompressed it on-the-fly to render it on the GPU. Marton et al. [MAG19] proposed a framework that used a multiresolution pyramid to compress and page rectilinear, time-varying volumes into GPU memory for rendering. In contrast to our approach, which begins from an existing spatial structure, prior work has *built* various *structured* spatial indices over rectilinear volumes specific for time-series visualization.

In the field of AMR visualization on GPUs, earlier work by Kähler et al. [KSH03, KWAH06] focused on data structures for efficient rendering. However, this work did not address smooth interpolation of cell-centered data at level boundaries, which is made challenging by the T-junction problem. Reconstruction issues like these have however been addressed on the CPU, e.g., by Weber et al. [WCM12], or by Wald et al. [WBUK17] who used tent-shaped basis function interpolation and cell location via kd-trees, or by Wang et al. [WWW*19] through an octant-based high-quality interpolant. Later work by Wang et al. [WMU*20] focused on CPU-based rendering of Octree AMR data, such as the Exajet. The ExaBricks data structure by Wald et al. [WZU*20] is focused on high-quality GPU rendering and is discussed below in Section 3. This data structure was later extended by Zellmann et al. [ZSM*22] to support steady flow visualization.

The system presented by Shih et al. [SZM*14] is most related to our framework, as it targets similar hardware configurations with time-varying AMR data. Their streaming approach uses an asynchronous prefetch queue that is updated in a background thread. In contrast to our framework, the proposed system does however not focus on high-quality interpolation.

In general, research on time-varying AMR has primarily focused on data sets where the grid topology changes over time. Kähler et al. [KPHH05] proposed to compute the union of several key frame grids to represent a whole collection of timesteps. Meyer et al. [MGA*08] focused on query-driven, multitemporal visualization, where features from all timesteps queried by the user are conveyed in a single image. To do so, Meyer at al. built a composite data structure derived from the temporal AMR grid hierarchy.

In contrast to prior work on time-varying AMR visualization,

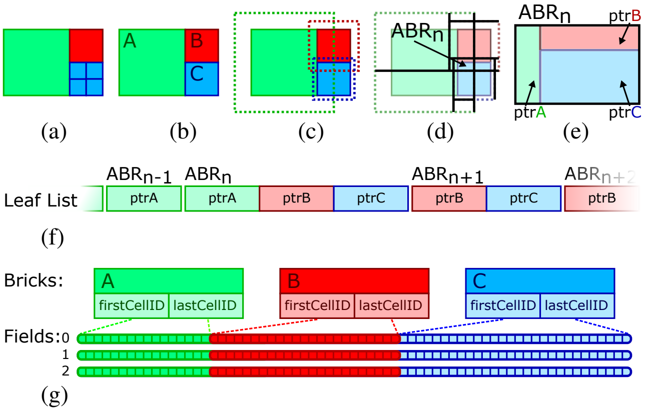


Figure 2: *The ExaBricks data structure. Same-level cells from different subgrids are flattened (a) to form bricks (b), whose filter support regions (c) are spatially subdivided into active brick regions (ABRs) (d). The pointers from (e) of all ABRs form the leaf list (f) that is used by integrators to quickly enumerate the ABRs' bricks. Bricks contain offsets into the scalar fields (g) which are ordered by cell ID.*

our work focuses on AMR simulations where the grid topology is stationary over time. The focus of our paper lies on streaming for time-varying, non-trivial volume data, and on accommodating data processing along the pipeline without introducing undesirable synchronization that disrupts the data flow and degrades performance.

3. Background: AMR and ExaBricks Data Structure

This section provides a review of the ExaBricks data structure by Wald et al. [WZU*20], which forms the basis of our proposed framework. ExaBricks is loosely based off previous work by Kähler et al. [KSH03], who reorganized the AMR grid into bricks of same-level cells using a kd-tree. These bricks are typically more spatially coherent and larger than the original AMR subgrids, as they are optimized using a kd-tree splitting heuristic—e.g., based on the surface area or volume—making them more amenable to rendering on GPUs.

The problem with this type of data structure is that, while vertices line up at level boundaries, cell centers generally do not, and hence the data structure is not suitable for trilinear interpolation of the cells at level boundaries using GPU texture units because of the T-junction problem [WBUK17].

To support the smooth interpolants for cell-centered AMR that were proposed by Wald et al. [WBUK17] or by Wang et al. [WWW*19], ExaBricks builds an additional data structure on top of Kähler's brick data structure that allows for quickly finding all the bricks whose *filter support* overlaps a given position inside the volume. In the case of the tent-shaped basis functions proposed by Wald et al. [WJA*17] the support region amounts to a single cell's width beyond the brick boundary. ExaBricks therefore introduces the concept of *active brick regions* (ABR), which are collections of pointers to bricks whose support overlaps a certain region of space; integrators that reconstruct the scalar value at a position x inside the volume can use the ABRs to quickly enumerate the bricks that contribute to the scalar reconstruction at a point.

Fig. 2 provides an overview of the data structure. In Fig. 2a we are given an AMR volume with three different subgrids at three different refinement levels. In Fig. 2b, same-level cells from separate subgrids are merged into bricks, whose filter support regions are depicted in Fig. 2c. The overlap of these regions are general L- and T-like shapes, etc., and thus not useful for rendering. ExaBricks therefore constructs a spatial partition over the support regions, shown in Fig. 2d, forming the rectilinear boxes called *active brick regions* (ABRs) by Wald et al. [WZU*20], which are more amenable to be rendered on GPUs.

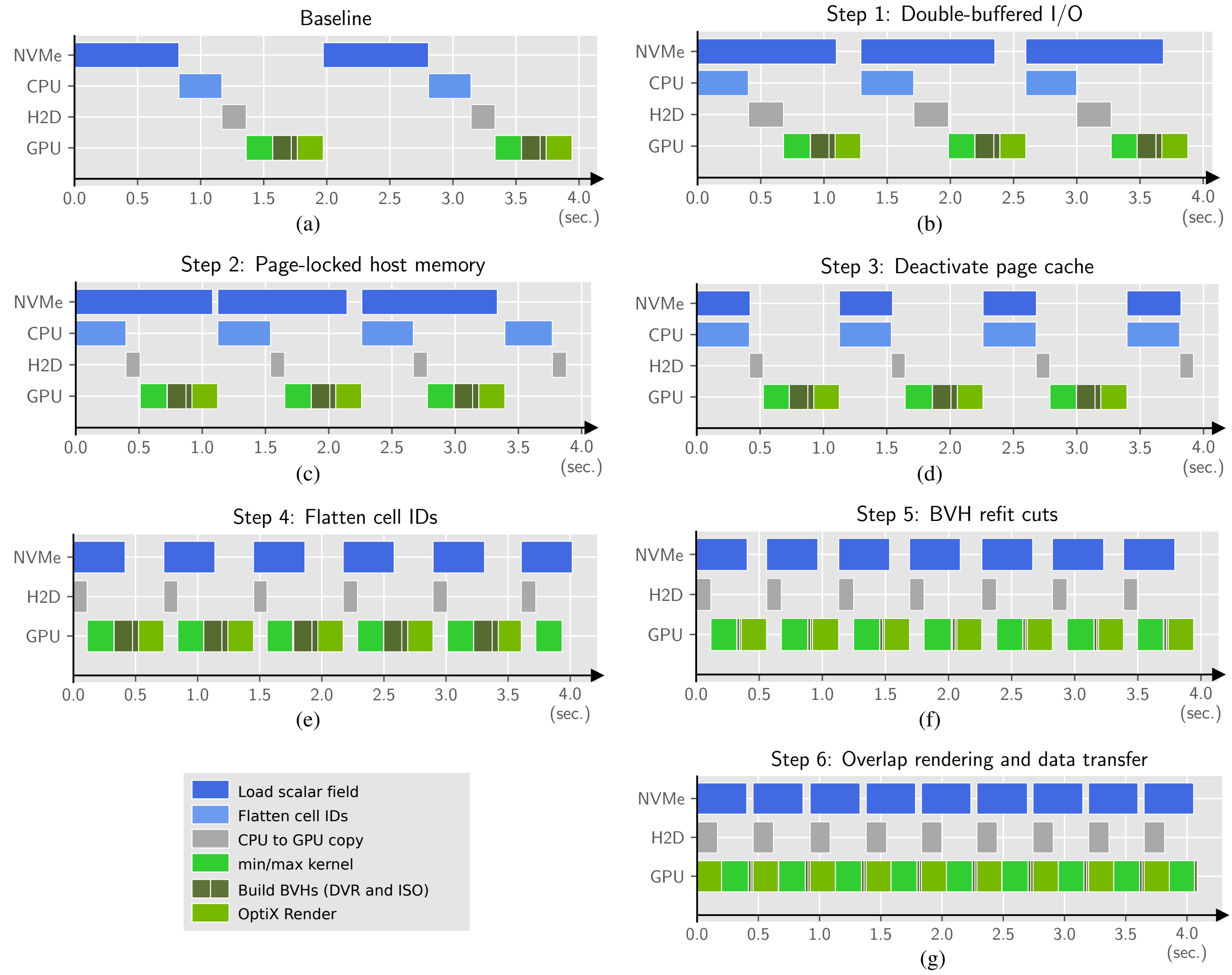
The ABRs store pointers to the bricks they overlap (cf. Fig. 2e); an integrator can then simply iterate over the list of bricks in a region to reconstruct the scalar field inside the ABR. Technically, this is realized by the ABRs storing offsets into a list of pointers mapping each ABR to its respective bricks (cf. Fig. 2f). Knowing the mapping of ABRs to their set of bricks, and a sample position to reconstruct a value at, integrators can locate the cells with overlapping support using cell ID offsets (Fig. 2f) into the lists of scalar fields (Fig. 2g) that were previously flattened in a pre-process and are hence ordered by cell IDs. Wald et al. [WZU*20] proposed to build OptiX BVHs over the ABRs to accelerate DVR and iso-surface rendering with empty space skipping and adaptive sampling. The DVR BVH must be rebuilt whenever the transfer function changes, by classifying the ABRs as empty or not empty based on pre-computed min/max ABR value ranges [PSL*98]; similarly, the iso-surface BVH is rebuilt when an iso value changes.

The ExaBricks data structure can render large AMR data sets like Exajet and supports smooth interpolation at level boundaries. The data structure—with its various hierarchies that are involved at the different phases—is however not particularly designed for fast updates or rebuilds. It also does not support the type of AMR data for example supported by Kähler, where subgrids can overlap and effectively form a level-of-detail hierarchy.

4. Hardware/System

In our study we focus on workstations where a top-shelf RTX GPU and *non-volatile memory express* (NVMe) NAND solid-state drive (SSD) are connected via the PCIe 4.0 interconnect. Modern-days NVMe storage devices are directly connected to the fabric via PCIe 4.0 and allow for high-bandwidth and low-latency accesses due to the drive being close to the CPU cores [Kou18].

We conduct our study on a GPU workstation with an NVIDIA Ampere A6000 graphics card with 48 GB GDDR6X memory, an 11th Gen Intel Core i7-11700KF, 3.60 GHz, eight core CPU, 64 GB DDR memory, and a 2 TB Samsung 980 PRO NVMe M.2 SSD. We base our study off the ExaBricks framework [WZU*20], which is written in C++ and CUDA and uses OptiX [PBD*10] to utilize NVIDIA's hardware ray tracing extensions (RTX). Some of the choices we make for our study are focused on this particular system with high-bandwidth access to the file system of our Linux distribution.



Baseline

(a)

Step 1: Double-buffered I/O

(b)

Step 2: Page-locked host memory

(c)

Step 3: Deactivate page cache

(d)

Step 4: Flatten cell IDs

(e)

Step 5: BVH refit cuts

(f)

Step 6: Overlap rendering and data transfer

(g)

Figure 3: *Timeline view and profiling results for the step-by-step outline of our framework as presented in Section 5.*

5. Step-by-Step Implementation and Analysis of an Optimized Streaming Prototype

In this section we develop step by step a highly optimized streaming-enabled AMR visualization system based on the ExaBricks framework [WZU*20]. We start by defining the ingredients for a naïve baseline that runs the necessary steps fully synchronously, and through developing the prototype analyze trade offs and considerations made when employing the optimizations we propose. The steps taken, from the baseline system to the highly optimized final solution, are summarized in Fig. 3. While laying out our approach step by step, we will continuously refer to Fig. 3 to illustrate the overall performance impact of each change.

5.1. Naïve Baseline

A naïve implementation of a rendering system targeting large AMR data sets would treat each time step as its own data set. In fact, this is exactly what the unmodified version of ExaBricks [WZU*20] does. In the following we detail the data movements and transformations that are performed by ExaBricks when used to render multiple consecutive time steps of the Exajet data set. The flow of execution is comparable to that of other systems targeted at AMR volume rendering.

5.1.1. File I/O and Data Movements

First, the data associated with the current animation frame are loaded from storage to DDR memory. This data is comprised of the scalar fields associated with the animation frame. For the Exajet we visualize the 2.5 GB density field. Any extra field, such as the three velocity components, would add another 2.5 GB of scalar data.

After the scalar fields are fully present in DDR memory, the data is transformed so that it can be rendered. First, element indices that are stored along with the data are flattened by constructing direct fields where addresses are implicit. The scalar fields are 1D arrays of type `float`. A flat list of bricks provides start and end offsets into those scalar arrays to retrieve them during rendering. To remove the element index indirection, the ExaBricks software allocates extra temporary storage per field to store the flattened scalars in, giving us the storage buffers depicted in Fig. 2g. The flattened scalars are finally copied to the GPU with CUDA using a single (blocking) host-to-device `memcpy` command.

5.1.2. ExaBricks Data Structure Construction

Next, the ExaBricks data structure is constructed. This comprises building *active brick regions* (ABRs) and OptiX BVHs (cf. Section 3). During this step the min/max value range of each ABR is computed for use in empty space skipping. This requires another pass over all scalar fields, which our implementation performs in a CUDA kernel. The min/max kernel clips each ABR's filter support region against the filter support regions of its bricks, and then iterates over the contained cells to compute the min/max range.

Finally, the OptiX BVHs are built, and must be rebuilt whenever the transfer function or iso values change, to remove empty ABRs for space skipping. The system described so far will only ever load a single animation frame that is then rendered and interacted with.

The steps described here are executed sequentially and data is only processed as soon as it is available in full at the respective logical or execution unit.

5.1.3. Adding Support for Time-Varying Data

Although not supported out of the box, with the components described above it is easy to implement a visualization and rendering system with support for time-varying data. The tasks that need to be performed *per time step* are: *file I/O*, *cell ID flattening* and *host/device* copies, *min/max computation* for empty space skipping, *BVH construction*, and finally *rendering*.

Since the AMR hierarchy of the data we focus on is fixed and does not change over time, the ABRs must only be built once at program start. However, we must still recompute the per-ABR min/max ranges when a new scalar field is loaded to update the OptiX BVH for empty space skipping on the new data.

With this naïve baseline animation system, and the target hardware presented above, we conducted a first performance study to analyze the bottlenecks that are encountered per frame. A timeline analysis for two consecutive animation frames can be found in Fig. 3a. We can see that the majority of the time to load, process, and render an animation frame with one scalar field is spent on file I/O, but that other major bottlenecks exist as well, such that the time to image for this naïve approach is on the order of two seconds.

In the following sections we describe the steps necessary to transform this system to make optimal use of the available bandwidth to render time series data sets like Exajet interactively. Our aim is to achieve smooth animation by approaching the limits imposed by the available bandwidth, while retaining interactive rendering performance. We note that these two are potentially opposing goals, as discussed below.

5.2. Step 1: Hiding File I/O Through Double Buffering

When analyzing our naïve baseline implementation, the first observation we make is that the system is currently limited by file I/O latency. Thus, our first step is to hide the roughly 0.8s of I/O latency for loading a single scalar field by preemptively loading the next one in the sequence while simultaneously executing the remaining processing steps for the first field. We implement this with double buffering as shown in Fig. 4. Here, we show a reduced version of the animation loop—that would in reality be distributed across a number of functions so that the animation can be paused, or the animation frame be incremented with a stride other than +1. When entering the loop, we wait until the first animation frame was loaded, then preemptively preload the next frame into the back buffer, while simultaneously performing the remaining operations necessary to render the current one. After rendering the single frame, the animation continues immediately. Profiling results of this approach can be seen in Fig. 3b.

```
int frontID=0; // id of front buffer
future fileIOFuture;
Fields scalarFields[2];
float *dFields[NumFields]; // device ptrs

for (int t=frameBegin; t<frameEnd; ++t) {
  // wait for pending preemptive fetch
  if (fileIOFuture.valid())
    fileIOFuture.wait();
  else {
    // The very first (blocking) read from SSD
    foreach(field: scalarFields[frontID])
      field->loadFile(t);
  }

  // Preemptively fetch t+1 into *CPU*
  // back buffer in the background
  fileIOFuture = async({
    foreach(field: scalarFields[!frontID])
      field->loadFile((t+1)%frameEnd);
  });

  // Flatten current field's cell IDs
  foreach(field: scalarFields[frontID]) {
    field->flatten(t);
  }

  // Upload from CPU front buffer to GPU buffer
  foreach(field,ID: scalarFields[frontID]) {
    cudaMemcpy(dFields[ID],field,
               cudaMemcpyHostToDevice);
  }

  // Swap buffers
  frontID = !frontID;

  // Compute min/max per region on the GPU
  regionsMinMax(dFields);

  buildBVHs();
  render();
}
```

Figure 4: *Double-buffered animation loop to overlap file I/O with the ensuing per-frame operations.*

By hiding I/O latency, our system performs simultaneous memory transfer operations to upload a time step t to the GPU, and to the

NVMe drive to preload the next time step $t+1$ from the file system. Both of these transfers go through PCIe 4.0. Comparing the latency to that imposed by our synchronous baseline implementation, we observe an increase in both file I/O as well as CPU to GPU latency, with I/O going up from 0.8-0.9s to 1.0-1.1s, and GPU copies going up from 190ms to 280ms. This is however made up for by an increase in throughput.

5.3. Step 2: Increasing DDR to GDDR Transfer Performance

Next, we optimize the host-to-device data transfer using `cudaMemcpy` by using page-locked memory for the staging area as proposed by Bastem et al. [BUZ*17]. The new execution flow now loads the data from storage to DDR, and when flattening the cell IDs, directly stores the flattened result in paged-locked memory that was allocated using `cudaMallocHost`. We use the `cudaMemcpyAsync` API call, but immediately synchronize the CUDA stream that the operation is executed on, for comparability with ordinary CUDA host-to-device copies. Results for the adapted experiment can be found in Fig. 3c.

We also made an interesting observation regarding the two simultaneous data transfers for file I/O and host-to-device copy over PCIe 4.0. With the synchronous baseline we achieved roughly 190 ms for the host-to-device copy of the 2.5 GB scalar field; and in Section 5.2 the switch to asynchronous I/O increased the CPU to GPU transfer time to 280 ms. However, when switching to use page-locked memory for CPU to GPU transfers, the transfer took about 110 ms, regardless of whether synchronous or asynchronous I/O was used. This indicates that simultaneous PCIe 4.0 transfers were competing for bandwidth on the bus or other resources provided by the host's memory subsystem, but that this issue can be mitigated by using page-locked host memory for the host-to-device copy's staging buffers.

5.4. Step 3: Increasing Read Performance from the SSD

We have now analyzed and optimized how SSD to DDR and DDR to GDDR transfers affect each other, and through this optimized CPU to GPU transfer time. The biggest bottleneck remaining in our system at this point remains I/O access to the SSD. A limiting factor of NVMe SSDs is the I/O stack, including synchronized operations that are performed on the page cache before the actual data transfer happens. While approaches such as the one by Lee et al. [LSS*19] propose an optimized I/O stack where those operations are performed asynchronously, for our study we resort to a pragmatic solution and just use I/O that bypasses the page cache altogether. On a Linux system this is achieved by replacing calls to the `fread` library function with calls to the `read` system call, and by creating the associated file descriptor using `open` and the `O_DIRECT` flag.

Apart from being pragmatic, completely bypassing the page cache also allows for better control and more deterministic results. Using Intel's Storage Performance Development Kit (SPDK) [YHW*17], it would be possible to design an I/O system without much effort that supports fast accesses to the SSD, while still benefiting from the file system's cache. We note that to report consistent results in our experiments we cleared the page cache manually and ensure that scalar fields are not reused after being loaded. However, this may not be representative of how users typically explore their data sets. We therefore report the performance of cached and uncached file I/O using `fread` to load the data from memory—compared to file I/O that bypasses the page cache—in Fig. 5. By bypassing the page cache, we can optimize file I/O to 6 GB/s (cf. Fig. 3d), which gets us near the maximum bandwidth of our system.

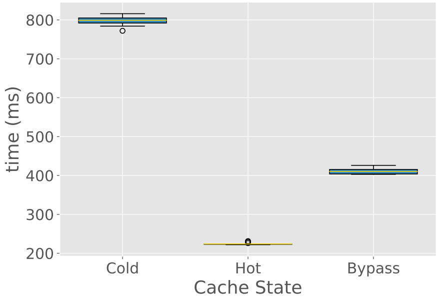


Figure 5: *Comparing `fread` performance for loading a 2.5 GB scalar field, with the page cache being empty ("Cold"), the page cache being filled ("Hot"), and the page cache being bypassed using system calls ("Bypass").*

5.5. Step 4: Flattening Cell IDs in a Pre-Process

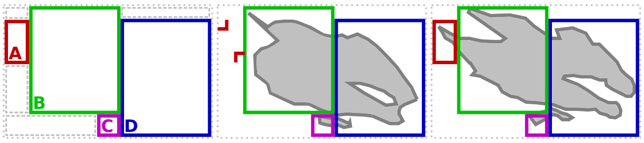
Our next step is to completely remove the flattening phase by observing that, with the type of AMR that ExaBricks supports, the mapping from cell IDs to scalars is always 1:1 and subgrids are not allowed to overlap. Hence, we can simply reorder the scalar fields by cell ID and store them in this new order to eliminate this step. We again note that this optimization is specific to data sets like Exajet, where the topology does not change over time.

The result from this optimization can be observed in Fig. 3e. We further note that after this change, the data can be directly streamed from storage through DDR memory to the GPU without any processing performed on the CPU. At this point the CPU is only responsible for initiating data movements or invoking GPU compute kernels, and does not perform any costly per-frame computations. Although not the focus of our paper, this would allow for storing compressed scalar fields, streaming them directly to the GPU, and only decompressing them on the GPU.

5.6. Step 5: Refitting to Improve BVH Build Performance

At this point we are no longer limited by bandwidth but instead by processing time on the GPU. This is in contrast to a large number of applications that are typically bandwidth limited. Thus, we turn our focus to reducing processing costs on the GPU.

As Exajet's AMR topology never changes, neither does the topology of the ABRs, nor does the topology of the OptiX BVH change significantly on transfer function changes. Recall that Wald et al.'s [WZU*20] ABRs are a flat list of boxes that form a spatial decomposition of the domain, and the BVH over them is used by ray integrators to quickly traverse from ABR to ABR. Each ABR



(a) (b) (c)

Figure 6: *Refit cuts allow us to use BVH refitting to build high-quality empty space skipping accelerators. (a) The initial OptiX BVH is a full spatial decomposition over the whole volume, induced by the ABRs. In this example we are only interested in the colored, not the dotted boxes. (b) The BVH that is valid for a certain time step, the red box labelled A is culled but remains in the BVH as a dummy node. (c) Another time step, this time the configuration is different and box A has become visible.*

stores the min/max range of the cells it overlaps; when the transfer function changes, an ABR can become invisible and be culled.

In the presence of animation frames and a stationary topology, we can eliminate the BVH rebuild step for each new timestep, and instead only refit the BVH. When a new animation frame is loaded, the ABR min/max value ranges are recomputed and the BVH nodes corresponding to invisible ABRs culled by inserting inverted boxes. However, as the mesh topology does not change, there is a bounding box associated with each ABR *before* culling, even if the ABR volume is currently invisible. This set of the bounding boxes of all ABRs forms a full spatial decomposition of the volume.

Before ever loading a scalar field, we build a BVH *once* over *all* these boxes, producing a tree whose leaves form the full spatial decomposition created by the ABRs (see Fig. 6a). We then use this BVH as a template, and never rebuild a new one from scratch. The template BVH provides the *topology* for all subsequent configurations, and is only adjusted via refits on visibility changes, e.g., loading a new scalar field or changing the transfer function.

When we load a scalar field, we compute the min/max ranges for each ABR and use OptiX to refit the BVH. By classifying min/max ranges in the bounds program as before, OptiX will compute a cut through the BVH based on the current visibility setting, thereby culling invisible ABRs and making newly visible ones active for ray intersections. Two such cuts can be seen in Fig. 6b and Fig. 6c. In Fig. 6b, the red box labeled A is culled, but is still present in the BVH as a dummy node. Traversing such BVHs—particularly when done in hardware—is as fast as traversing a BVH that was just rebuilt, while refitting reduces BVH update time by an order of magnitude (see Fig. 3f). We use the same approach for the iso-surface BVH, where visibility is driven by the iso-value.

5.7. Step 6: Overlapping Host-to-Device Copies with Compute

In a final step, we hide the host-to-device copy latency behind GPU compute. As outlined in Section 5.2, we already use asynchronous copy operations, but (apart from removing the synchronization barrier) need to take measures so the operations can overlap.

First, we must decide which GPU operations to overlap the copy with. As seen in Fig. 3f, the possible kernels to overlap with are the min/max range computation kernel or rendering with OptiX. Based on knowledge of the work performed in each kernel, we decide to overlap the host to device copy of frame $t+1$ with rendering frame t; our reasoning here is that min/max incurs more pressure on the memory subsystem, while rendering is largely compute-bound, and memory-intensive operations such as large data transfers are best hidden behind compute.

```
- ~~float *dFields[NumFields]; // device ptrs~~
+ float *dFields[NumFields][2]; // device ptrs
```

```
    foreach(field,ID: scalarFields[frontID]) {
-     ~~cudaMemcpy(dFields[ID],field,~~
-                ~~cudaMemcpyHostToDevice);~~
+     cudaMemcpyAsync(dFields[ID][!frontID],
+       field,cudaMemcpyHostToDevice,copyStream);
    }
```

```
    // Compute min/max per region on the GPU
-   ~~regionsMinMax(dFields);~~
+   cudaDeviceSynchronize();
+   regionsMinMax(dFields[frontID]);
```

```
    buildBVHs();
-   ~~render();~~
+   renderAsync(frontID);
```

Figure 7: *Extending the CPU-side double buffer from the listing in Fig. 4 to allow for asynchronous host to device buffer uploads that overlap with the* `render` *compute kernel.*

To enable overlapping the operations, we must make sure that the rendering and data copy operations are performed in separate CUDA streams. The host memory involved in the data transfer is also required to be page-locked, as otherwise the operations will not overlap. With this in mind, we extend the double buffering approach from Section 5.2 to use two separate GPU device buffers.

Finally, when min/max regions are computed, we have to synchronize so that the buffer we just uploaded is available on the device. For simplicity, we synchronize the whole device, i.e., the min/max computation will only start when the GPU front buffer was fully updated *and* the previous frame was rendered. As a consequence, frames that render faster than the host to device copy will have to wait for the latter to finish. For maximum throughput, we could extend our pipeline to also overlap with the min/max computation, and possibly the BVH builds, thereby relaxing the synchronization. However, this would require also double buffering the min/max value ranges, the ABRs, and the BVHs. We instead assume that the host to device copy can be completely, or at least mostly, hidden behind rendering to reduce overall complexity.

The changes required to our pipeline to overlap the host-to-device copy and rendering are shown in Fig. 7. We extend the listing to use two device buffers that are fully coupled with the host-side double buffers. While the CPU's front buffer serves as a staging area for the host-to-device data transfer, the GPU's front buffer is used for rendering. When swapping one, we at the same time also swap the other.

Although the ABRs are not double-buffered, we must take care when recomputing the min/max regions for BVH refits to use the scalars from the current front buffer on the device. Thus, we must

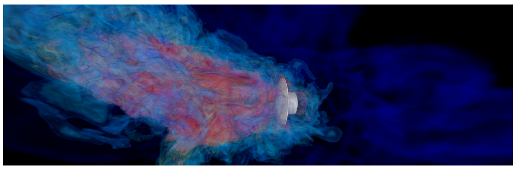
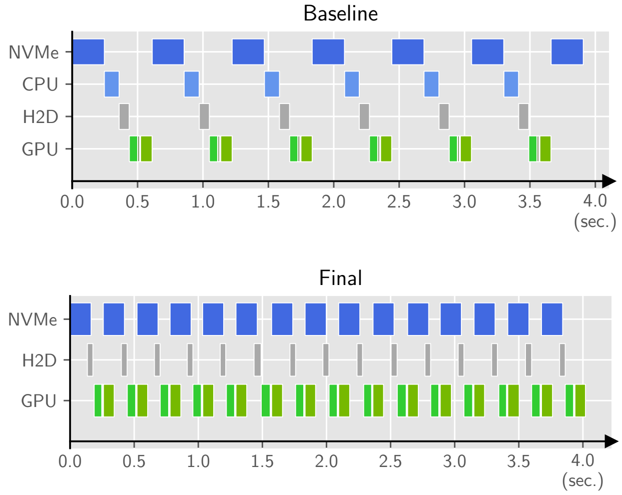
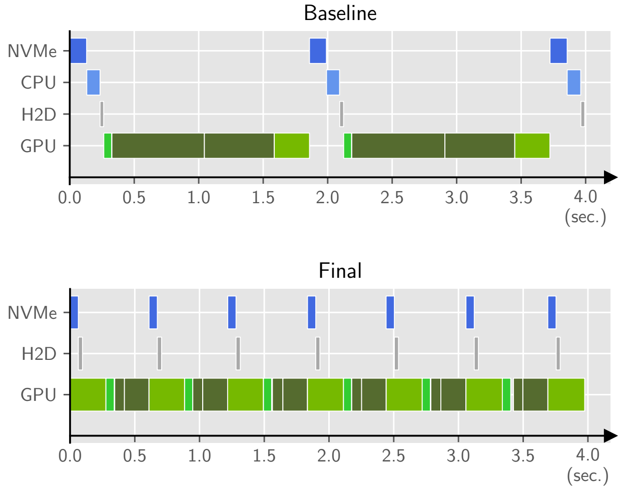
NASA Landing Gear	NASA Mars Lander
	
Cells: 262M (1 GB) Bricks: 55K ABRs: 1.6M Speedup: 2.3×	Cells: 100M (382 MB) Bricks: 3.7M ABRs: 127M Speedup: 3.5×
	

Table 1: *Evaluation of our framework on other data sets. We evaluate on the NASA landing gear, which is a block-structured AMR data set, and the NASA Mars lander, which was resampled from generally unstructured finite elements to forest of Octrees AMR.*

perform the synchronization mentioned before. Here we can use a full device synchronization instead of just synchronizing a single GPU stream, as the only other compute operation running simultaneously is the rendering kernel, which we synchronize on, too.

After min/max computation and BVH refit, we asynchronously render from the frame buffer, allowing the operation to overlap with the host-to-device transfer of a new time step.

With this optimization, rendering still takes the same 200 ms as before, but host-to-device transfer performance goes *down* from 110 ms to roughly 160 ms (cf. Fig. 3g). This still allows us to hide the copy operation completely, so that we effectively win another 110 ms over the synchronous version. We note that in situations where rendering is much faster than the data transfer overlapping the two operations may have an adverse effect.

6. Evaluation on Other Data Sets

We evaluate our framework on two other data sets with different characteristics to make sure that it will generalize. The first data set is the NASA landing gear that was also used by Wald et al.'s ExaBricks paper [WZU*20]. The second data set is a resampled version of the NASA Mars lander, an originally unstructured data set consisting of billions of finite elements [WMZ22]. The landing gear is a block-structured AMR data set with 1 GB per scalar field, which our brick and ABR builders convert into 55K bricks and 1.6M ABRs. We resampled the lander on a forest of Octrees with 100M cells, resulting in a scalar field of size 382 MB. For the landing gear we only have access to a single time step that we duplicate on the SSD to make sure the data set is not preloaded in the page cache. The Mars lander consists of ten time steps that were all resampled on the same AMR hierarchy.

We present the results of our evaluation in Table 1. The color coding used by the graphs is the same that we used in Fig. 3. We observe that again, we are able to completely hide the I/O and memory transfer latency—particularly as both data sets are smaller in size than our reference data set, the Exajet. We see overall speedups of 2.3× and 3.5×. With the landing gear, we observe that rendering is faster than data transfer and hence encounter the issue mentioned above that, as the ABRs are not double-buffered, operations are synchronized on the rendering kernel. The highly complex lander is limited by BVH builds, where the refit cut algorithm is particularly effective. In both cases, we are GPU compute bound, as we observed before with the Exajet. We observe—although we are now not limited by this operation—that min/max computation is relatively costly: in fact, the compute kernel's running time is of the same order as the host-to-device transfer to upload the scalar field.

7. Analysis of Remaining Bottlenecks

The framework we presented optimizes throughput and can successfully hide I/O operations behind GPU computation. This allows us to stream and then render the 2.5 GB Exajet at more than two frames per second (FPS)—including all the necessary data movement and data wrangling—achieving a speedup of about 4× over the 0.5 FPS achieved with the naïve baseline. Compared to rendering only a single frame, without streaming and pre-processing, the theoretical limit we could achieve when *only* rendering the data set

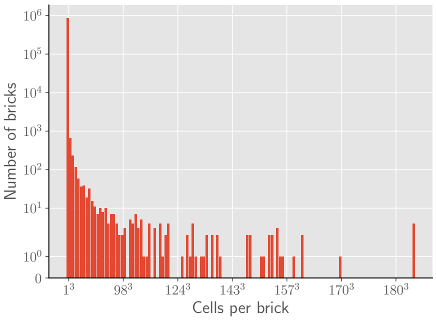


Figure 8: *Histogram (log scale) over number of cells per brick, for Exajet. A majority of bricks contains only a single cell.*

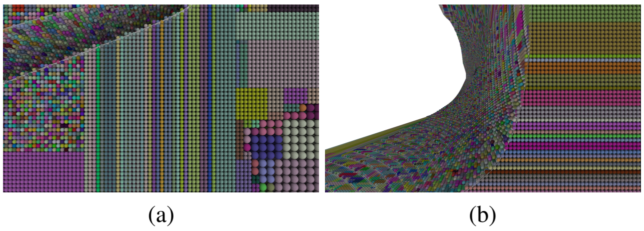


Figure 9: *A visualization of bricks at mesh and level boundaries that cannot be merged using ExaBricks's builder. Cells are represented as spheres that are sized according to their AMR level and colored by their brick ID.*

would be roughly 5 FPS. In this section, we briefly analyze what we find to be the root cause of this gap and provide directions for future improvements.

A surprising observation is that the min/max value range kernel has an extraordinarily high running time compared to the other GPU operations. To understand what this kernel is limited by, we briefly review the operation. The kernel executes over each ABR and clips the ABR's bricks' filter supports against the ABR's filter support itself. The resulting clip regions will be aligned on the logical coordinate system of the ABR's brick with the smallest size. The kernel then executes three nested loops whose start and end offsets are determined by the clip regions.

Potential caveats here are that we a) run a significant amount of sequential operations per GPU thread, and b) if the clip ranges differ a lot, we will also suffer from warp and general execution divergence. This is in harsh contrast to the framework of Marton et al. [MAG19] who also compute per-brick min/max value ranges in connection with streaming their data to the GPU. Their bricks are regular and their spatial index more structured than our *given* spatial index, so that they do not suffer from as much divergence.

We spent significant effort attempting to optimize this kernel, trying schemes such as restructuring the nested for loop, or using a brick-to-ABR index list and parallelizing not over ABRs but over that list and using atomic operations similar to the voting operation of Zellmann et al. [ZHL19]. Zellmann et al. [ZHL19] compute per-block value ranges in a CUDA *shared memory* atomic; after this operation finished, a single thread updates an atomic counter in global memory for a uniformly sized macro cell. However, in our case we cannot use shared memory atomics as the number of threads allocated to an ABR is not known in advance. Thus, we must resort to global memory atomics, which slow down the computation significantly.

Processing the Exajet specifically is made more challenging by the fact that the builder creates a large number of single-cell bricks (cf. Fig. 8). These single-cell bricks are caused by the many neighboring different-level cells near the hull and wing of the jet, where the simulation mesh boundary is finely tessellated (cf. Fig. 9). As the cells are on different levels, they cannot be merged into single bricks by our builders. In the future we would have to accommodate such meshes by extending our builders, for example, by allowing them to promote cells onto the next finer AMR level by duplicating the cell.

Our observations lead us to the conclusion that AMR data structures that organize same-level cells into bricks potentially suffer from the limitations described here. We particularly point out that the different-level cells that cannot be merged are encountered on the brick- and not the ABR level, so that this observation is not limited to data structures such as the one by Wald et al. [WZU*20] that support smooth interpolation.

Adapting the data structure to accommodate the issue described points to important future work, but is outside the scope of this paper. Measures taken to improve the quality of the acceleration data structure, such as clustering, are likely to negatively impact construction performance, which would need to be accounted for. Our observations about the technical aspects of data streaming, overlap of operations, and the effect of simultaneous transfer operations on each other stand on their own and can probably even be translated to other large data visualization problems.

8. Discussion

We have presented a highly optimized framework that supports streaming very large AMR data sets from an NVMe drive through CPU DDR memory to the GPU. The type of volume data set we consider is non-trivial; a major challenge with this endeavor is that the visualization requires us to process the data at several stages along the streaming and rendering pipeline. We presented our experience with designing such a framework.

The pipeline we present is static: rather than preloading a significant number of frames into a dynamic data structure (see for example Shih et al. [SZM*14]) we only preload but a single frame in a double buffer. Ultimately, this is due to the fact that supporting smooth interpolation via data structures such as the ABRs makes streaming updates much more complicated, and we found static data structures using a fixed number of buffers more flexible here. Depending on the ratio between memory transfer and compute operations, which in similar systems might be different, the framework could be extended to use triple or quad buffering.

However, the design we ended up with does not reflect our initial intuition. What we initially had in mind was a typical streaming system reminiscent of those used for video on demand (VOD) [KLW*08] With VOD, the situation is quite different in that one is typically bound by network bandwidth, while decompression and single frame processing are comparably cheap and in fact typically performed on multiple frames at once.

In our use case, of rendering non-trivial data at high-quality, we are generally limited by GPU operations, and can completely hide I/O and memory transfer behind them. This encourages a completely different design such as ours, where the various pipeline operations execute in lockstep. We found that there are ample advantages to this type of design. For example, the initialization interval of our pipeline—i.e., the number of operations required when restarting the animation by resetting the loop counter and jumping through the data set—is controllable and totally deterministic. In our case, it is bound by preloading a single frame, after which the animation can be played smoothly. A design such as ours is also easier to maintain than, for example, one using a dynamic buffer that is filled in the background.

9. Limitations and Future Directions

One clear limitation of our approach is that it can currently only handle data sets where the AMR hierarchy itself does not change over time. For some applications, this will be acceptable, while for others, it will not. This clearly means that more work will be required, and that eventually we also need techniques that can rebuild the hierarchy.

However, even future techniques that will ultimately allow for rebuilding the hierarchy must still solve the problems covered in this paper, thus we believe our approach to be a first step in that direction. Furthermore, having a technique that works for some applications is arguably still better than not having a better one that works for all. This is, in fact, reminiscent of how support for dynamic scenes evolved in ray tracing: early techniques could only do refitting, while full arbitrary rebuilds were only solved later on [Wal07]. In this analogy, refitting was not the final solution, but served as an important stop-gap to move the field forward until better techniques could be found. Another interesting observation from this analogy is that even today refitting is still widely used because it is cheaper than full rebuilds, and thus preferable for those applications where it is applicable.

Our analysis also revealed bottlenecks in the overall AMR pipeline, and a more careful construction or some lightweight clustering or collapsing steps might significantly improve the quality of the ExaBricks data structure, pointing us to important future work. Ultimately, when optimizing the compute bottlenecks, we may find ourselves in a situation where we are again limited by bandwidth rather than compute. In that case, a natural pathway for further optimization would be to store and load compressed data, that is only decompressed once loaded on to the GPU, e.g., using `zfp` [Lin14]. Our framework would be ideally suited to such a scenario, as it already defers all computation to the GPU, and uses the CPU only to initiate data transfers and GPU compute kernels.

10. Conclusions

We presented a highly optimized streaming and high-quality visualization framework for non-trivial, large-scale volumes with AMR hierarchies that do not change over time. Step by step extending the ExaBricks data structure, we share our experience and analyze the various bottlenecks and potential sources of contention with a system where both file I/O and GPU data transfers share the same high bandwidth, low latency interconnect. We finally arrive at a point where throughput is increased by up to 4.5× for the 2.5 GB scalar field of the Exajet. In contrast to our initial intuition, our system is limited by GPU compute and not by bandwidth; we discussed and also indicate significant future work to improve on the ExaBricks data structure; with the current framework, we are able to successfully hide all the latency from file I/O and data transfer and give a guide how to efficiently use the various system components in order to achieve this.

Acknowledgments

We acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—grant no. 456842964. We also thank the Ministry of Culture and Science of the State of North Rhine-Westphalia for supporting the work through the PROFILBILDUNG grant PROFILNRW-2020-038C. The Landing Gear was graciously provided by Michael Barad, Cetin Kiris and Pat Moran of NASA. The Exajet was made available by Exa GmbH and Pat Moran.

References

[BC89] Berger M. J., Colella P.: Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics 82*, 1 (1989). 2

[BO84] Berger M. J., Oliger J.: Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics* (1984). 2

[BUZ*17] Bastem B., Unat D., Zhang W., Almgren A., Shalf J.: Overlapping data transfers with computation on GPU with tiles. In *2017 46th International Conference on Parallel Processing (ICPP)* (2017), pp. 171–180. doi:10.1109/ICPP.2017.26. 2, 6

[DCS09] Du Z., Chiang Y.-J., Shen H.-W.: Out-of-core volume rendering for time-varying fields using a space-partitioning time (SPT) tree. In *2009 IEEE Pacific Visualization Symposium* (2009), pp. 73–80. doi:10.1109/PACIFICVIS.2009.4906840. 2

[KJL*19] Koh S., Jang J., Lee C., Kwon M., Zhang J., Jung M.: Faster than flash: An in-depth study of system challenges for emerging ultra-low latency SSDs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)* (2019), pp. 216–227. doi:10.1109/IISWC47752.2019.9042009. 2

[KLK16] Kim H.-J., Lee Y.-S., Kim J.-S.: NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (Denver, CO, June 2016), USENIX Association. URL: https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim. 2

[KLW*08] Ko C.-L., Liao H.-S., Wang T.-P., Fu K.-W., Lin C.-Y., Chuang J.-H.: Multi-resolution volume rendering of large time-varying data using video-based compression. In *2008 IEEE Pacific Visualization Symposium* (2008), pp. 135–142. doi:10.1109/PACIFICVIS.2008.4475469. 2, 9

[Kou18] Koutoupis P.: Data in a Flash, Part I: the Evolution of Disk Storage and an Introduction to NVMe. *Linux Journal* (Dec 2018). 3

[KP21] Kogge P. M., Page B. A.: Locality: The 3rd wall and the need for innovation in parallel architectures. In *Architecture of Computing Systems* (Cham, 2021), Hochberger C., Bauer L., Pionteck T., (Eds.), Springer International Publishing, pp. 3–18. 1

[KPHH05] Kähler R., Prohaska S., Hutanu A., Hege H.-C.: Visualization of time-dependent remote adaptive mesh refinement data. In *VIS 05. IEEE Visualization, 2005.* (2005), pp. 175–182. doi:10.1109/VISUAL.2005.1532793. 2

[KSH03] Kähler R., Simon M., Hege H.-C.: Interactive volume rendering of large data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (2003), 341 – 351. doi:10.1109/TVCG.2003.1207442. 2, 3

[KWAH06] Kähler R., Wise J., Abel T., Hege H.-C.: GPU-Assisted Raycasting for Cosmological Adaptive Mesh Refinement Simulations. In *Volume Graphics* (2006), Machiraju R., Moeller T., (Eds.), The Eurographics Association. doi:10.2312/VG/VG06/103-110. 2

[LeB18] LeBeane M. W.: *Optimizing Communication for Clusters of GPUs*. PhD thesis, The University of Texas at Austin, 2018. 2

[Lin14] Lindstrom P.: Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2674–2683. doi:10.1109/TVCG.2014.2346458. 10

[LSS*19] Lee G., Shin S., Song W., Ham T. J., Lee J. W., Jeong J.: Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 603–616. URL: https://www.usenix.org/conference/atc19/presentation/lee-gyusun. 6

[MAG19] Marton F., Agus M., Gobbetti E.: A framework for GPU-accelerated exploration of massive time-varying rectilinear scalar volumes. *Computer Graphics Forum (Proceedings Eurographics/IEEE Symposium on Visualization, Eurovis 2019 38*, 3 (2019), 53–66. 2, 9

[MGA*08] Meyer M., Gosink L. J., Anderson J. C., Bethel E. W., Joy K. I.: Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1715–1722. doi:10.1109/TVCG.2008.157. 2

[PBD*10] Parker S. G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D., McAllister D., McGuire M., Morley K., Robison A., Stich M.: OptiX: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July 2010), 66:1–66:13. URL: http://doi.acm.org/10.1145/1778765.1778803. 3

[PSL*98] Parker S., Shirley P., Livnat Y., Hansen C., Sloan P.-P.: Interactive ray tracing for isosurface rendering. In *Proceedings Visualization '98 (Cat. No.98CB36276)* (1998), pp. 233–238. doi:10.1109/VISUAL.1998.745713. 3

[SSC16] Silva W. A., Sanetrik M. D., Chwalowski P.: Using FUN3D for aeroelatic, sonic boom, and aeropropulsoservoelastic (APSE) analyses of a supersonic configuration. In *15th Dynamics Specialists Conference* (January 2016). URL: https://arc.aiaa.org/doi/abs/10.2514/6.2016-1319, doi:10.2514/6.2016-1319. 2

[SZM*14] Shih M., Zhang Y., Ma K.-L., Sitaraman J., Mavriplis D.: Out-of-core visualization of time-varying hybrid-grid volume data. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)* (2014), pp. 93–100. doi:10.1109/LDAV.2014.7013209. 2, 9

[Wal07] Wald I.: On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), RT '07, IEEE Computer Society, pp. 33–40. 10

[WBUK17] Wald I., Brownlee C., Usher W., Knoll A.: CPU Volume Rendering of Adaptive Mesh Refinement Data. In *SIGGRAPH Asia 2017 Symposium on Visualization* (2017). doi:10.1145/3139295.3139305. 2, 3

[WCM12] Weber G. H., Childs H., Meredith J. S.: Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (AMR) data. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2012), pp. 31–38. doi:10.1109/LDAV.2012.6378973. 2

[WGLS05] Wang C., Gao J., Li L., Shen H.-W.: A multiresolution volume rendering framework for large-scale time-varying data visualization. In *Fourth International Workshop on Volume Graphics, 2005.* (2005), pp. 11–223. doi:10.1109/VG.2005.194092. 2

[WJA*17] Wald I., Johnson G., Amstutz J., Brownlee C., Knoll A., Jeffers J., Günther J., Navratil P.: OSPRay - a CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (Jan 2017), 931–940. 3

[WMU*20] Wang F., Marshak N., Usher W., Burstedde C., Knoll A., Heister T., Johson C. R.: CPU Ray Tracing of Tree-Based Adaptive Mesh Refinement Data. *Computer Graphics Forum* (2020). doi:10.1111/cgf.13958. 2

[WMZ22] Wald I., Morrical N., Zellmann S.: A memory efficient encoding for ray tracing large unstructured data. *IEEE Transactions on Visualization and Computer Graphics 28*, 1 (2022), 583–592. doi:10.1109/TVCG.2021.3114869. 8

[WWW*19] Wang F., Wald I., Wu Q., Usher W., Johnson C. R.: CPU Isosurface Ray Tracing of Adaptive Mesh Refinement Data. *IEEE Transactions on Visualization and Computer Graphics* (2019). doi:10.1109/TVCG.2018.2864850. 2, 3

[WZU*20] Wald I., Zellmann S., Usher W., Morrical N., Lang U., Pascucci V.: Ray tracing structured AMR data using exabricks. *IEEE Transactions on Visualization and Computer Graphics* (2020). 2, 3, 4, 6, 8, 9

[YHW*17] Yang Z., Harris J. R., Walker B., Verkamp D., Liu C., Chang C., Cao G., Stern J., Verma V., Paul L. E.: SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (Los Alamitos, CA, USA, dec 2017), IEEE Computer Society, pp. 154–161. URL: https://doi.ieeecomputersociety.org/10.1109/CloudCom.2017.14, doi:10.1109/CloudCom.2017.14. 6

[ZHL19] Zellmann S., Hellmann M., Lang U.: A linear time BVH construction algorithm for sparse volumes. In *Proceedings of the 12th IEEE Pacific Visualization Symposium* (2019), IEEE. 9

[ZSM*22] Zellmann S., Seifried D., Morrical N., Wald I., Usher W., Law-Smith J., Walch-Gassner S., Hinkenjann A.: Point containment queries on ray tracing cores for AMR flow visualization. *Computing in Science Engineering* (2022), 1–1. doi:10.1109/MCSE.2022.3153677. 2