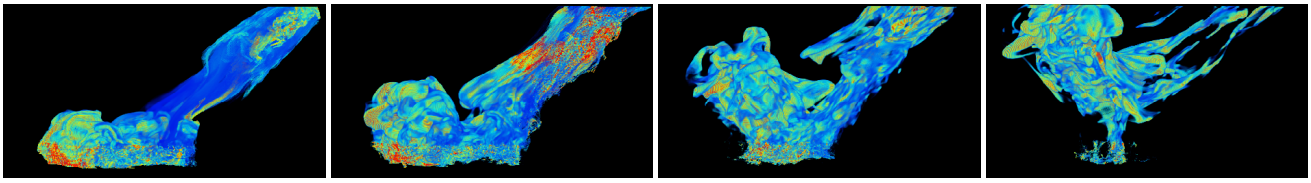
Eurographics Symposium on Parallel Graphics and Visualization (2023)
R. Bujack, D. Pugmire, G. Reina (Editors)

A GPU-based out-of-core architecture for interactive visualization of AMR time series data

W. Alexandre-Barff[1] , H. Deleau[1] , J. Sarton[3] , F. Ledoux[2] , L. Lucas[†1]

[1] Université de Reims Champagne-Ardenne, LICIIS, LRC DIGIT, France
[2] CEA, DAM, DIF, LRC DIGIT, F-91297 Arpajon, France
[3] Université de Strasbourg, ICube, UMR-CNRS 7357, France



(a) *Time step 60* (b) *Time step 120* (c) *Time step 180* (d) *Time step 240*

Figure 1: *AMR time series visualization of the scalar volume fraction of an asteroid at four time steps. The data are averaged at 14 frames per second (fps) and come from the original Deep Ocean Water time sequence dataset [PG17]. It consists of 269 volumes, with each volume containing 460 × 280 × 240 voxels encoded on 128 bits (about 31 million cells). One time step requires 470MB, and for a public or high-end GPU with respectively 4GB or 48GB, it will only be able to store 8 to 104 time steps at most (less than half of the dataset). Therefore, we propose an out-of-core system to navigate freely and interactively in the temporal and spatial dimensions during visualization.*

Abstract

This paper presents a scalable approach for large-scale Adaptive Mesh Refinement (AMR) time series interactive visualization. We can define AMR data as a dynamic gridding format of cells hierarchically refined from a computational domain described in this study as a regular Cartesian grid. This adaptive feature is essential for tracking time-dependent evolutionary phenomena and makes the AMR format an essential representation for 3D numerical simulations. However, the visualization of numerical simulation data highlights one critical issue: the significant increases in generated data memory footprint reaching petabytes, thus greatly exceeding the memory capabilities of the most recent graphics hardware. Therefore, the question is how to access this massive data - AMR time series in particular - for interactive visualization on a simple workstation. To overcome this main problem, we present an out-of-core GPU-based architecture. Our proposal is a cache system based on an ad-hoc bricking identified by a Space-Filling Curve (SFC) indexing and managed by a GPU-based page table that loads required AMR data on-the-fly from disk to GPU memory.

CCS Concepts

• ***Human-centered computing*** → ***Scientific visualization;***

1. Introduction

Numerical simulation is mathematical processing of real-world complex temporal phenomena using dedicated models. It can enhance scientific understanding of and provide insights into the interpretation and diagnosis of these models. Among all space discretizations, AMR representation is the most suitable for time-dependent evolutionary phenomena studies [dlAC17, KWAH06]. The AMR principle combines the simplicity of structured grids with the advantages of local refinement to obtain a multi-resolution hierarchy of cells. Still, interactive visualization of numerical simulation data highlights one critical issue: the significant increases in generated data memory footprint can now reach petabytes, thus vastly exceeding the memory capabilities of the most recent graph-

† The LRC DIGIT, a joint lab between CEA-DAM Ile de France and URCA, supports this research work. Authors would also like to thank J. Patchett from LANL and B. Levy from INRIA for providing the dataset.

DOI: 10.2312/pgv.20231080

ics hardware. Therefore, it is necessary to design scalable methods to handle large-scale data, such as i) data reduction methods at the expense of lowering visualization quality, or ii) parallel/distributed rendering methods on HPC clusters with different load-balancing strategies like [YMW04] or based on a kd-tree partitioning [LVI*13]. On a single GPU workstation, it is common practice to stream each time step of the dataset one by one, as proposed by Zellmann *et al.* in [ZWS*22]. However, this limits our ability to visualize the time series as we can only view it as an "*animation*" from beginning to end, whereas we would like to freely navigate through the spatial and temporal dimensions into the GPU cache. Our goal is to achieve high-quality interactive visualization of large-scale AMR time series on a simple workstation using an out-of-core approach that can handle massive datasets from high-performance computing devices such as current GPUs.

Contributions. In this paper, we present a cache system based on an ad-hoc bricking scheme identified by SFC indexing. The cache system is managed by a GPU-based page table that loads the required AMR data on-the-fly from disk to GPU memory for AMR time series visualization.

The remainder of this paper is organized as follows. In section 2, we briefly review the state of the art in AMR visualization and representation, scalable methods, time-varying AMR data management, and SFC. The background of the data caching system we use is discussed in section 3, before we describe our contribution in section 4. In section 5 we briefly discuss the early rendering approach, while in section 6 we compare our approach to existing methods. Experimental results are then presented and discussed in section 7. Finally, the conclusion and future work are given in section 8.

2. Related Work

AMR data visualization and representations

Many works have focused on the visualization of AMR data. There are a few studies that are based on the use of an object-based rendering method, such as [PBS02] that uses splatting, but the most recent methods are all based on ray-tracing approaches. In this context, we can find different methods of data representation used for the visualization. Several approaches are based on the dual of the mesh, such as [WHJ*01, WKL*01], which rely on a dual grid for continuous interpolation. This approach was later generalized by [ME11]. Wald [Wal20] also uses a dual mesh representation for surface visualization. On the other hand, working directly on the primal mesh, [KCP*02] proposed to divide the AMR hierarchy into equal resolution. In contrast, [WMU*20] used a hierarchical data representation, specifically an octree, which is well suited for parallel ray traversal on multi-CPUs. More recently, [WZU*21, ZWS*22] proposed to transform the AMR hierarchy into a set of non-overlapping bricks, called exabrick, for GPU volume raytracing. Regarding the time-varying AMR representation, Kaehler *et al.* [KPHH05] first introduce an intermediate grid hierarchy by merging the grids of refinement levels. They then use a clustering algorithm to induce a nested grid, and finally map the intermediate hierarchy using linear interpolation. However, their approach is limited to remotely accessible data with a sufficiently fast network and the ability to store resources for data processing. Next, Gosink *et al.* [GABJ08] proposed a nested hierarchy of grids with different resolutions for temporally concurrent visualization, displaying all variations of the scalar field through all time steps in a single image. Nevertheless, they acknowledge that their method is limited by the VRAM capacity, which requires finding a solution to overcome this limitation.

Scalable methods for static regular grid

Scalability issues related to the visualization of massive data are an active area of research, largely driven by various technological advances in computing. Beyer *et al.* [BHP15] review some of these developments in their report on state-of-the-art visualization, which typically require two essential components:

- a data representation: the decomposition of the original regular Cartesian grid into several independent sub-volumes to reduce the size of the data to be processed.
- an addressing data structure for navigating through the data representation: achieved through a tree traversal [CNLE09], or a multilevel multiresolution page table [HBJP12].

Sarton *et al.* [SCRL20] introduce a dynamic data structure based on a caching strategy with a virtual memory addressing system coupled with efficient parallel management on the GPU to allow efficient access to regular grids in interactive time. These out-of-core approaches have already been proposed for regular Cartesian grids with explicit hierarchical representation, without time management considerations. Furthermore, while there exists a direct and natural relationship between the regular Cartesian grids and the data structure in this particular case, applying this model to time-varying AMR data is no less trivial.

Time-varying AMR data management

For time-varying AMR, Shih *et al.* [SZM*14] first propose a prefetch strategy by storing as much time step data as possible in RAM and VRAM during visualization. Although their solution involves out-of-core methods, it requires at least one full step to be completely stored in VRAM at a time, thus leaving the VRAM capacity bottleneck unresolved. Subsequently, Zellmann *et al.* [ZWS*22] present a streaming approach for GPU visualization of time-varying AMR data. Their approach is limited to static AMR hierarchy only, and by their sequential static methods of loading an entire single time step of data that fits in GPU memory at the time, which limits the freedom to navigate between different time steps. However, the approach of Zellmann *et al.* strengthens our preference for out-of-core approaches to handle large-scale datasets.

Using an out-of-core approach in this context, one must be able to address any part of the entire spatial representation of a 3D volume time series. SFCs generally have the property of mapping multi-dimensional (3D in this study) data into one-dimensional. Panda *et al.* [PMM16] present in their survey some typical applications to SFC, mainly Hilbert curves. The methods that come closer to our needs are, Sasidharan *et al.*'s [SDS15] approach using recursive generalized SFC to partition structured and unstructured 2D meshes. Their applications of SFC shows potential for AMR partitioning, although it is limited to 2D. Then, Zhou *et al.* [ZJW21]

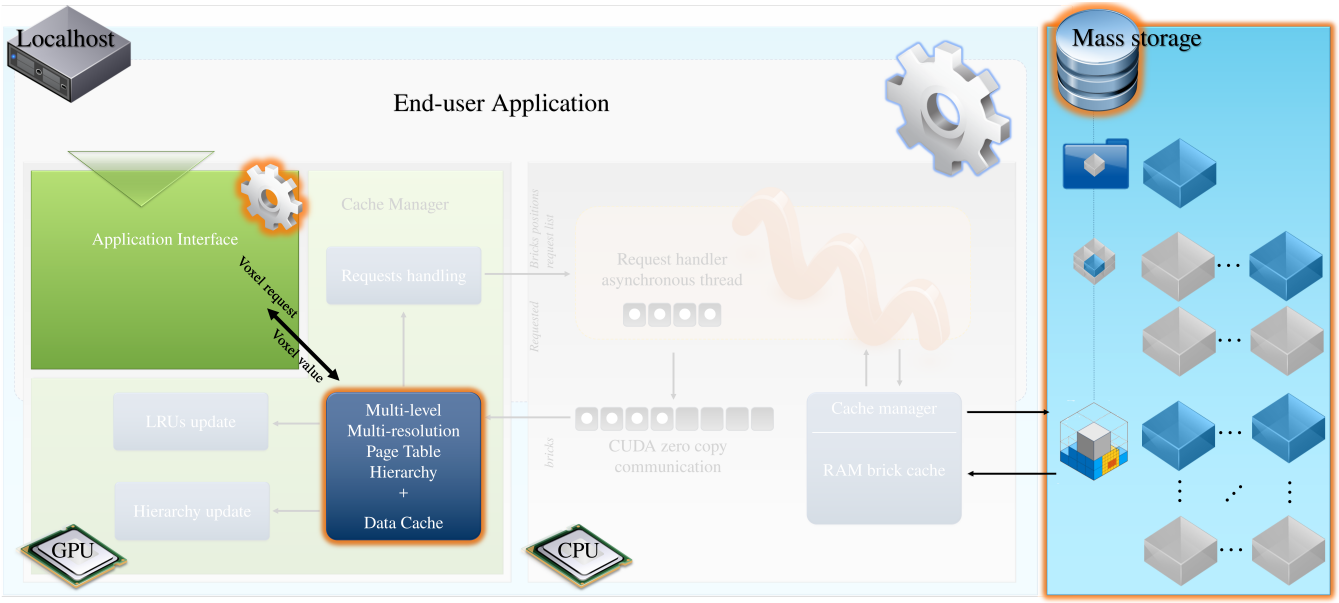


Figure 2: *Pipeline overview. The transparent parts are similar to the work of Sarton* et al. *scalable approach [SCRL20]. The parts highlighted in red (preprocessing and mass storage, time sequence brick addressing table, and the visualization application) reflect our adaptations to manage AMR spatiotemporal datasets.*

apply their solution of data-driven SFC methods to multiscale data by finding a Hamiltonian path. However, they only consider multiscale data in an octree format. Wang *et al.* [WGLS05] use error-guided load balancing based on SFC traversal applied to a hierarchical wavelet tree for parallel multiresolution volume rendering. Their use of Hilbert curves is similar to ours, but for a distributed workload across multiple GPUs. Finally, Kumar *et al.* [KEB*14] propose to improve an existing I/O multiresolution format in a parallel framework for AMR simulations by merging the AMR hierarchy into a single array. Although their methods provide excellent spatial and hierarchical localization for AMR data, their approach does not extend to time series AMR data.

In this paper, we propose to build on the work of Sarton et al. [SCRL20] by adapting the direct virtual addressing approach of static regular grid to time-varying AMR data. We propose a GPU implementation of ad-hoc SFC indexing coupled with a 2D addressing data structure that loads the AMR data from disk to GPU memory on-demand. We apply a different SFC path per AMR refinement level, which is used through a GPU data structure that allows addressing all refinement levels regardless of the time step. We use this approach with a visualization method based on Fogal et al.'s ray-guided volume rendering [FSK13], since it can be implemented in our out-of-core architecture with little to no effort.

3. Background: GPU-based out-of-core approach

The volume ray tracing used for regular grid visualization or AMR relies on a data sampling step inside the volume while traversing it. On GPU, this step, which can be linear or hierarchical, requires access to all the cells of the volume in texture memory. When the whole volume does not fit in GPU memory, one possibility is to rely on an out-of-core approach with a cache system to store the data. In this kind of system, data addressing is usually done by traversing a hierarchical structure that allows data virtualization. Here, we propose to rely on the system presented by Sarton *et al.* [SCRL20] for static regular grids. This method uses a multilevel multiresolution page table structure introduced by [HBJP12] with an entirely GPU-based cache management to ensure data queries and structure update in parallel, similar to Crassin *et al.* [CNLE09] for an octree. The modifications we make to the [SCRL20] pipeline in our contribution are highlighted in red in Figure 2 and concern the management of time-varying AMR data. Sarton *et al.*'s virtual addressing steps are as follows:

1. Assume an **end-user application** that handles large-scale regular voxel grids and requests specific data at a 3D position within a normalized volume defined as a pair (3D position, *Level Of Details* (or LOD)).
2. virtual memory addressing is triggered, which consists of looking at the address pointed to by the pair (3D position, LOD) in the GPU **addressing data structure** to know if the requested data is in the GPU data cache or not, leading to only three possible cases:

   i) EMPTY: The desired data does not exist, meaning that the end-user application is trying to reach a 3D position where no actual data exists.
   ii) MAPPED: This means that the requested data is in the GPU data cache, so it immediately returns to the end-user application with the requested data, and the LRU array is next updated.
   iii) UNMAPPED: This means that the desired data is not in the GPU data cache, which triggered the creation of a requested

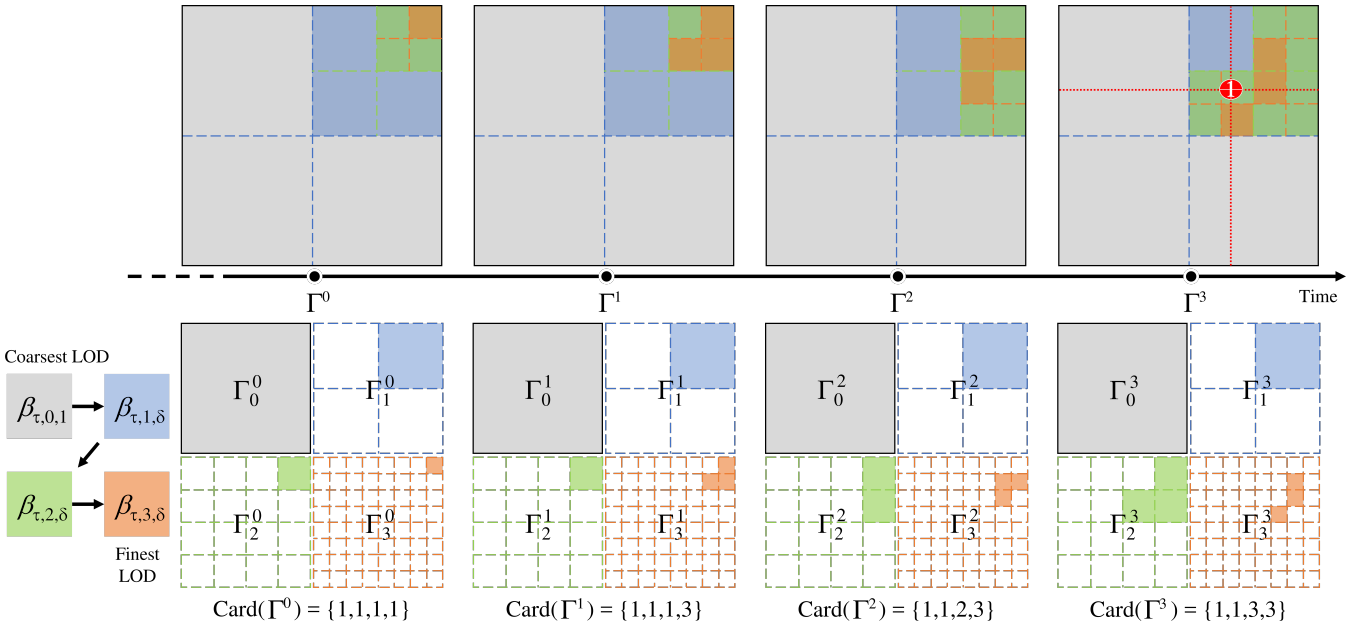


Figure 3: *2D representation of a Block-Structured AMR (BS-AMR) sequence. It consists of four time steps, each defined on four Levels Of Detail (LOD). The BS-AMR hierarchy varies throughout the sequence, which is illustrated by the different topologies of the AMR bricks. Each AMR brick is represented according to its LOD from coarsest to finest. In the fourth time step, a 2D normalized position is requested and displayed in red. In this figure, Γ_i^τ stands for a BS-AMR at time step τ and LOD i, $\beta_{\tau,i,\delta}$ is an AMR brick identify by its time step τ, its LOD i and, its SFC index δ and finally Card(Γ) determines the decomposition of a BS-AMR hierarchy per LOD. These terms are described in detail in the section 4.1 and 4.2.*

data list form of the id of the desired data from the GPU to the CPU and is handled asynchronously to avoid bottlenecks.

Throughout the entire visualization process, a single CPU thread processes the asynchronous request list to look up the mirroring addressing data structure that manages the CPU cache to know if the desired data exists. By following the same three possible cases as the GPU counterpart, we do not change anything except the data flow. If the desired data is in the CPU cache, it is transferred to the GPU, and the CPU's LRU is updated. In contrast, if the data is not present in the CPU cache, it is loaded directly from the mass storage. In order to load the desired data from the **data representation** obtained during the preprocessing step, we first check if the CPU cache is full. If it is, we search the CPU's LRU to find the last requested data and swap it with the newest data. If the CPU cache is not full, we store the newest data from the CPU cache before sending it to the GPU. Once all the cache system mechanisms are complete, we update the GPU's LRU with the requested data. If necessary, we swap the last requested data with the newest data from the CPU if the GPU's data cache is already full.
The approach proposed by Sarton *et al.* is general-purpose and allows the cache update strategy to be adapted to the needs of the end-user application, both for rendering and processing. Moreover, the virtual memory addressing mechanism ensures that the end-user application always requests the necessary voxel data and avoids unnecessary loading. This approach is modular, agnostic, and sufficiently generic to be adapted to any data type by first adapting **the data representation** module, then **the addressing data** structure, and finally **the end-user application** (see red highlighted parts in Figure 2). In the next section, we discuss how we adapted these modules for AMR time series data.

4. Adaptations to AMR time series data

In this section, we detail our contributions. After defining fundamental concepts, we specify how the preprocessing (Hilbert's indexing) and data addressing steps work.

4.1. Definition

We define a *Block-Structured* AMR (BS-AMR) [BC89, MA15] as a nested hierarchy of structured grids, denoted by Γ. A grid of this hierarchy represents one refinement level (LOD) and is denoted by $\Gamma_{i=0\ldots(n-1)}$, where n is the number of LODs, ranking from the coarsest Γ_0 to the finest Γ_{n-1}. By definition, and for any LOD $i > 0$, a BS-AMR satisfies the relation:

$$\Gamma_i \subseteq \Gamma_{i-1} \subseteq \Gamma\,.$$

By extension, a single BS-AMR of a time series (see Figure 3) is defined by Γ^τ, where τ is the time step. A refinement level for a given time step is denoted by Γ_i^τ. For simplicity, in the following definitions we will focus on a single time step of the sequence, i.e., a single BS-AMR. Therefore, except for Γ_0, each subsequent structured grid is nested within a coarser grid with a refinement ratio of 2. Therefore, for a single AMR brick belonging to a coarser

grid, there are at most 2^3 AMR bricks of the next finer grid nested within it.

Moreover, we call an AMR brick (brick for short), denoted by β_i, a group of voxels (scalar data) contained in and belonging to any structured grid, where each voxel has a spatial resolution corresponding to its respective LOD. This brick definition is cell-centered, which implies that a single brick belonging to any coarser grid will store the average voxels among the nested bricks of the next finer grid. Thus, each brick in each LOD contains the same number of voxels, meaning it has the same memory footprint. Also, due to the cell-centered property, the coarsest grid Γ_0 is a single brick at the top of the structured grid hierarchy that covers the entire BS-AMR domain, such as:

$$\Gamma_0 = \beta_0 \, .$$

This property guarantees that we always have a return value during the voxel request for the end-user application.

To describe the topology of our BS-AMR definition, i.e., the AMR hierarchy, we introduce the following notation: $\mathrm{Card}(\Gamma_i)$ denotes the number of bricks within a given structured grid. Similarly, we denote by $\mathrm{Card}(\Gamma)$ an array of integers representing the number of bricks within each structured grid of a BS-AMR from the coarsest to the finest LOD, such that:

$$\mathrm{Card}(\Gamma) = \{\mathrm{Card}(\Gamma_0), ..., \mathrm{Card}(\Gamma_{n-1})\} \text{ with } \mathrm{Card}(\Gamma_0) = 1.$$

4.2. Preprocessing

This step is responsible for converting regular Cartesian grids into a BS-AMR sequence that represents each individual time step of the dataset. Our preprocessing step generated BS-AMR with a set of bricks that always contained the same number of voxels with different spatial resolutions but the same memory footprint. This ensured a constant algorithmic time to load the bricks into the GPU memory.

We then use an SFC path to address them uniquely and individually. Defined in a unit cube, this 3D SFC allows us to convert - as a mapping function - the central 3D position of the brick into a single SFC index, denoted by δ, which is related to the order of its SFC traversal. We use Hilbert's curve here because of its property of preserving the locality distance between nearby points [DCOM00]. Since Hilbert's path is sensitive to domain resolution (see Figure 4 for a 2D indexing), we apply a different Hilbert ordering path according to each LOD i for $i = 1...(n-1)$, such that each brick belonging to any Γ_i has an index SFC δ_i:

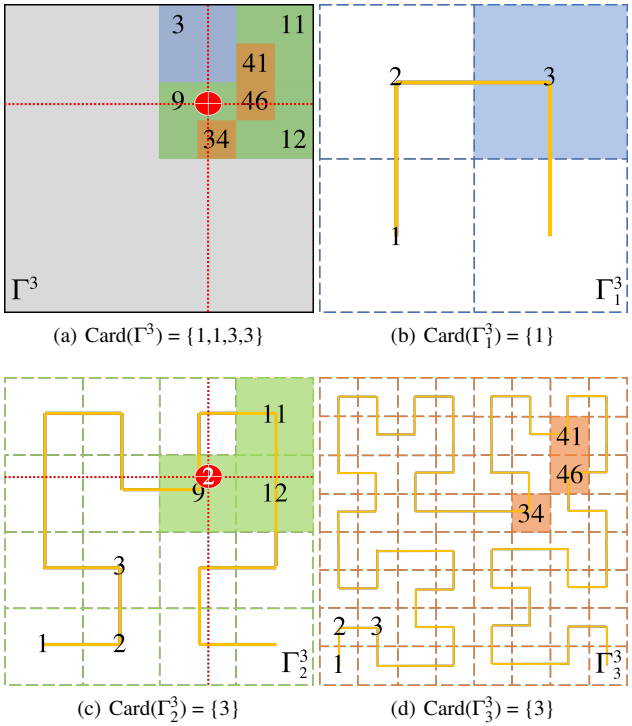
$$\delta_i \in \{1, ..., 2^{3*i}\} \qquad \text{with} \qquad \delta_0 = 1.$$

In our study, we focus only on the BS-AMR sequence, where each BS-AMR has the same spatial dimension throughout the sequence. Therefore, the same Hilbert ordering path is used for the whole sequence.

As a result of this operation, a single brick β is identified by the expression:

$$\beta_{\tau,i,\delta} \, ,$$

where τ indicates the time step, i the LOD, and δ the SFC index.



(a) $\mathrm{Card}(\Gamma^3) = \{1,1,3,3\}$ (b) $\mathrm{Card}(\Gamma^3_1) = \{1\}$

(c) $\mathrm{Card}(\Gamma^3_2) = \{3\}$ (d) $\mathrm{Card}(\Gamma^3_3) = \{3\}$

Figure 4: *2D Space-Filling Curve (SFC) indexing of a single BS-AMR. (a) The result of SFC's indexing for the fourth BS-AMR of the sequence from Figure 3, where we have applied (b) a first, (c) then a second, (d) and a third order Hilbert curve for subsequent nested structured grids according to their LOD along their respective topology. The 2D normalized position is always requested at the finest available LOD, shown in red.*

At the end of our preprocessing step, we store each brick scalar value into individual file I/Os identified by their respective time step, LOD, and SFC index. It is these file contents that are asynchronously uploaded to the CPU all the way up to the GPU during our out-of-core mechanism. Thus completing our **data representation** for AMR time series data.

4.3. Addressing data structure

In this section, we present the atomic form of our **addressing data structure**, then as a whole, before giving a step-by-step example of virtual addressing.

Brick addressing table architecture

The main component of our addressing data structure is the *Brick Addressing Table* (BAT), as shown in Figure 5. Noted $\mathcal{B}^\tau$, a BAT is defined as an array of bricks obtained from a single BS-AMR Γ^τ, with a size equal to the number of bricks found per LOD i.

Data in $\mathcal{B}^\tau$ are naturally ordered in ascending order from coarsest to finest LOD i. In addition, all bricks are organized similarly by their respective SFC index δ such that $\forall \beta_{i=0...(n-1)} \in \mathcal{B}^\tau$ and

$\forall \delta \in \{1, ..., 2^{3*i}\}$ with,

$$\beta_{\tau,i} \prec \beta_{\tau,i+1}$$

$$\beta_{\tau,i,\delta} \prec \beta_{\tau,i,\delta+1} \, .$$

Once we create a $\mathcal{B}^\tau$ for each time step τ of the BS-AMR sequence, we use an upper level of our **addressing data structure** to handle the collection of $\mathcal{B}^\tau$ along with the data cache management.

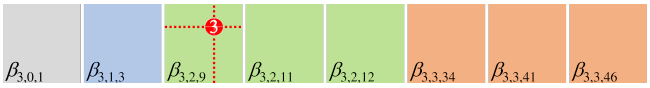


Figure 5: *Brick addressing table (BAT) of a single BS-AMR. The corresponding BAT for the fourth BS-AMR of the sequence in Figure 3 (denoted by $\mathcal{B}^3$). Each AMR brick is sorted by its LOD, with the coarsest value first, and then by its SFC index, with the smallest value first. This BS-AMR representation allows us to reduce any AMR hierarchy to a one-dimensional array. The 2D normalized position is converted into 2D coordinates in the corresponding AMR brick, shown in red.*

Time step brick addressing table architecture

Similarly, we call our virtual memory **addressing data structure** the *Time Step Brick Addressing Table* (TS-BAT), as shown in Figure 6. We can think of the TS-BAT as an upper level of the data cache. We define the TS-BAT architecture as a 2D array of entries of time-ordered $\mathcal{B}^\tau$ on the Y-axis, such as $\mathcal{B}^\tau \prec \mathcal{B}^{\tau+1}$, and on the X-axis, such as:

$$\max_{\tau}(\sum_{i=0}^{n} \text{Card}(\Gamma_i^\tau)) \, .$$

Each entry in the TS-BAT represents a single brick $\beta_{\tau,i,\delta}$ in the entire BS-AMR sequence and stores a pair $(cachePos, flag)$, where *cachePos* is a 3D position pointing to the address of the brick's voxels $\beta_{\tau,i,\delta}$ stored in the data cache and $flag \in \{\text{MAPPED}, \text{UNMAPPED}, \text{EMPTY}\}$. These *flag* indicate the following states:

- If $flag = \text{MAPPED}$, the requested brick is already in the data cache. We send the voxel back to the **end-user application** before updating the LRU.
- If $flag = \text{UNMAPPED}$, the asynchronous request buffer is sent to the CPU to load the requested brick from the system or mass memory.
- Otherwise, the **end-user application** tries to reach a position without scalar data.

Virtual addressing

To obtain the value of a given voxel, the **end-user application** sends a 3D normalized position in the data for a given time step to our *virtual memory* **addressing data structure**. Any 3D normalized position at any time step belonging to the BS-AMR sequence will go through the same following steps, here illustrated in 2D according to the red spot from Figure 3 to Figure 6.

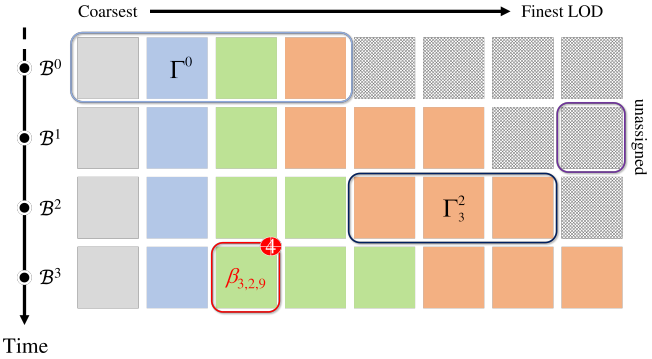


Figure 6: *Time step brick addressing table (TS-BAT) of a BS-AMR sequence. The TS-BAT corresponds to the sequence shown in Figure 3, where each row represents the BAT of each time step in ascending order from top to bottom, and the entries are sorted according to their respective BAT order in a given row. This TS-BAT has four rows, corresponding to the number of time steps in the sequence, and as many columns as the largest size of all BATs ($\mathcal{B}^3$ in this example). The remaining entries of all other smaller rows are not assigned, resulting in a sparse matrix architecture. The AMR brick corresponding to the 2D normalized position requested is shown in red.*

1. Virtual addressing starts from a 2D normalized position requested during a given time step (the fourth time step $\tau = 3$ in this example), shown in Figure 3.
2. We then compute the finest LOD available for the corresponding time step ($\tau = 3$) to always ensure the finest possible value. Otherwise, we send the corresponding value in the coarsest LOD to ensure that we always find a minimum value. Once the LOD is found ($i = 2$ in this example), we then compute the SFC index of the corresponding brick ($\delta = 9$ in this example), shown in Figure 4.
3. We search in the BAT of the corresponding time step ($\tau = 3$) to get the offset (2 on the X-axis) and compute the 2D coordinate in the brick to get the corresponding voxel coordinate shown in Figure 5.
4. Finally, we look in the TS-BAT at the corresponding row for the given time step (fourth row) and the computed offset (third column) to retrieve the entry corresponding to the brick ($\beta_{3,2,9}$ in this example) that contains the requested voxel shown in Figure 6.

Once all these steps are done, we either return the voxel value from the data cache if the entry is MAPPED, or send an asynchronous request list to the CPU to load the missing brick if the entry is UNMAPPED. Except for the steps mentioned above, the rest of the original out-of-core mechanism [SCRL20] remains the same. From the asynchronous request sent to the CPU, to the CPU memory management mirrored by the GPU counterpart. We then identify the requested brick file I/O within our **data representation** by its time step, LOD, and SFC index (which in this example are $\tau = 3$, $i = 2$, and $\delta = 9$) before sending it back to the GPU.

However, unlike [SCRL20], we deal with hierarchical multiresolution data, which affects direct virtual addressing. Thus, we use

Hilbert's code to bypass the direct addressing approach. Furthermore, we reduce the AMR hierarchy to a one-dimensional array with our BAT architecture to navigate the temporal dimension in our 2D TS-BAT architecture.

5. Rendering

Our **end-user application** provides a classic ray marching loop to compute the absorption and emission model, based on Max's equation [Max95]. It's a simple *Direct Volume Rendering* (DVR) program using CUDA to solve the *absorption and emission* equation. The rendering algorithm works *front-to-back*, allowing *early-ray-termination* when the ray's opacity exceeds a given threshold. We cast one ray per pixel on the image plane intersecting our volume data, and for each ray, we compute the intersection pair (t_0,t_1) with Γ_0. Then, we evaluate each 3D position within the volume, starting from t_0 and stepping by samples, until we reach t_1 or an opaque computed opacity according to the *early-ray-termination*. To evaluate each 3D position in the volume, our rendering method adopts the scalable GPU-based ray-driven volume rendering approach of Fogal *et al.* [FSK13]. The Fogal approach is based on the observation that the data required for visualization is usually smaller than the full data set. Therefore, it provides an efficient way to reduce the working set of large datasets by exploiting optimal brick sizes. Brick size has a significant impact on empty space skipping methods, especially those based on domain decomposition into bricks. Smaller brick sizes improve *empty space skipping* while avoiding I/O exchanges. Fogal *et al.* found that a 64^3 voxel brick size provides an optimal tradeoff [FSK13]. Their approach can be summarized as follows:

1. calculate the LOD of the 3D position along the ray
2. identify which brick to fetch from the GPU
3. look-up in the addressing data structure the status of the requested brick:
   - if empty, skip the brick and continue ray-casting;
   - else if non-empty and present, we can ray-cast directly with that data;
   - else (if non-empty and absent), we report the misses to the GPU addressing data structure to trigger the fetch from the CPU side. Then we return the coarser value of the desired brick.

Our scalable approach is easily integrated into the overall process. It fits seamlessly into the out-of-core architecture by replacing the addressing data structure with ours and prioritizing the computation of the highest resolution LOD. Moreover, during the sampling phase, our approach does not require normalized access to the GPU texture to fetch voxel data from the data cache, since our BS-AMR format is cell-centered. In fact, direct access to the 3D position inside the brick is fast, coupled with the use of a voxel coverage of one in each direction for each brick to avoid brick boundary artifacts. The focus of this paper is to demonstrate the viability of our framework for handling large-scale AMR time series data as a rudimentary visualization *end-user application*. Therefore, its data traversal and sampling optimization are unrelated to the validation of our primary contribution. Second, although we have chosen a visualization purpose, this rendering module can also be integrated with other processing modules. Finally, this rendering approach can be added to scientific visualization tools such as *VTK's Paraview*, as well as other APIs such as *OSPRay*, *OptiX*, or *IndeX* to provide *hardware-accelerated ray tracing* support. In the long run, integration with *Omniverse* is feasible, but deployment on such a platform is beyond the scope of this paper.

6. Comparison with existing method

Zellmann *et al.*'s [ZWS*22] GPU streaming framework for visualizing time-varying AMR data is the closest approach to ours. Indeed, our solutions visualize large-scale time-varying AMR data with a scalable approach. Both of our approaches enable interactive time visualization with good overall visual quality.

In particular, the management of GPU memory allocation is different between our approaches. Zellmann et al. ensure that the data for each time step fits completely into GPU memory, while we load only the data necessary for sample rendering into GPU memory, resulting in a much smaller memory footprint. In addition, Zellmann's approach is limited to static AMR hierarchies and focuses only on changes in scalar values throughout the sequence, while our approach can handle dynamic AMR hierarchies using our 2D addressing data structure and unique brick addressing. Finally, due to the streaming method, Zellmann is limited to navigating between temporally coherent data, while our solution allows non-coherent AMR data to be loaded side-by-side.

We limit our comparison with the solution in [ZWS*22] to a methodological point of view, since both methods deal with the same type of data, but with a different focus on memory management. Zellmann *et al.* optimize the GPU/disk exchange, while we focus on a more dynamic management of the data from a spatial and temporal point of view, without considering the exchange optimization. Furthermore, to the best of our knowledge, there are no other methods besides Zellmann *et al.* that deal with exactly the same type of large-scale data for interactive visualization on a single GPU.

7. Experimental results

All testing was performed on an NVidia RTX A4000 Mobile with 8GB of VRAM, an 8-core 11th Gen Intel Core i7-11850H 64-bit CPU, with 32GB of RAM, and CUDA 11.7 to render in a Full HD 1920×1080 viewport.

We used two datasets defined as a sequence of:

- Dataset 1: 269 volumes of $460 \times 280 \times 240$ voxels encoded on 128 bits (4 scalar fields - material Id, temperature, volume fraction of water, and volume fraction of asteroid). It represents the study of asteroid impacts in deep ocean water [PG17]. This sequence has been refined with a 4 level of refinement, and each brick has 64^3 voxels.
- Dataset 2: 311 volumes of $1000 \times 1000 \times 1000$ binary voxels representing a free-surface fluid simulated by a semi-discrete partial optimal transport algorithm [Lév22]. This sequence has been refined up to 5 levels to ensure the same brick size as Dataset 1, i.e. 64^3 voxels each.

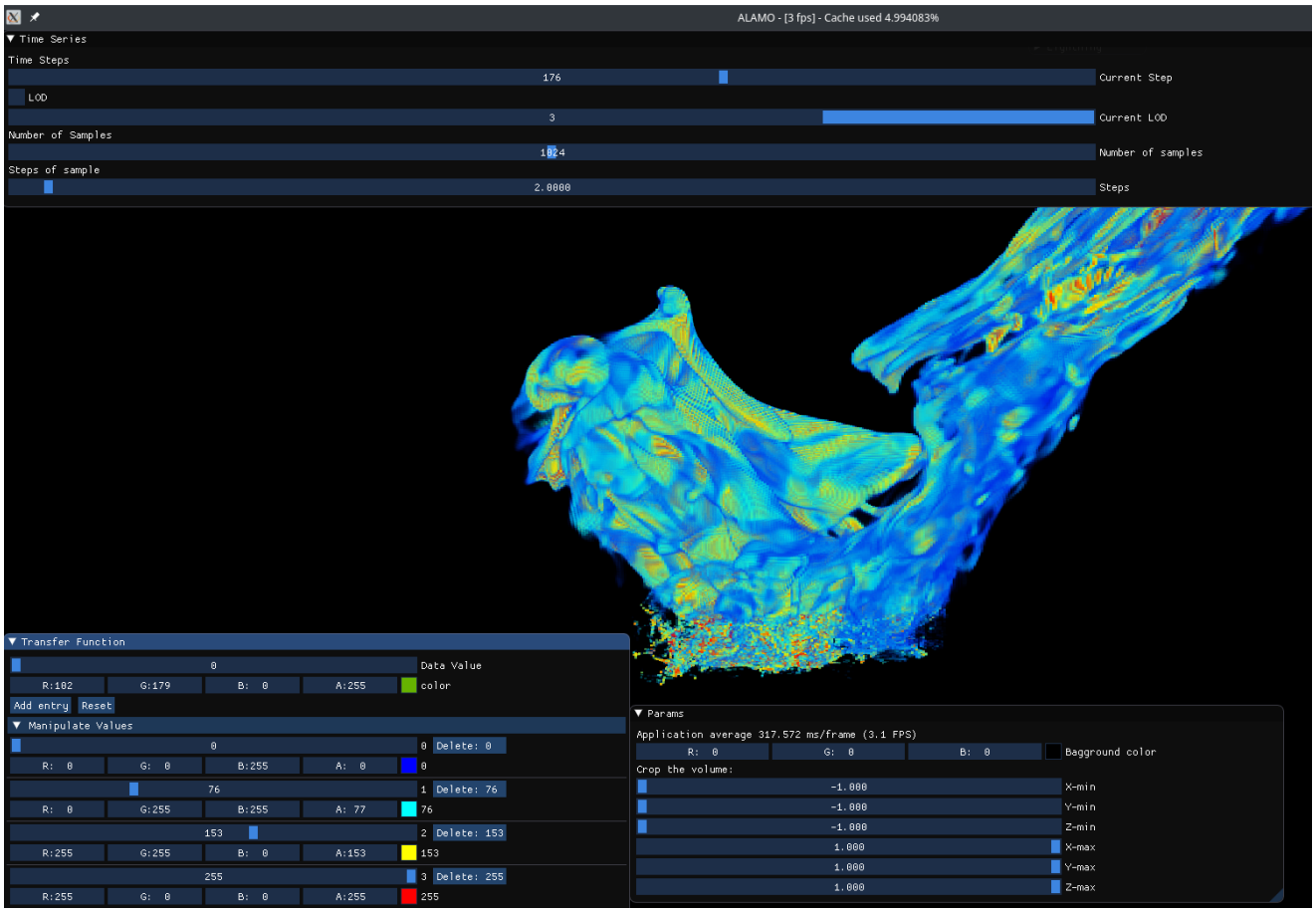
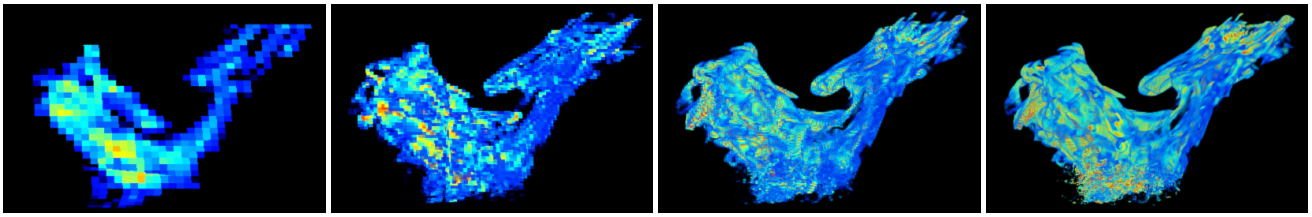


Figure 7: *Visualization Interface from Dataset 1.*



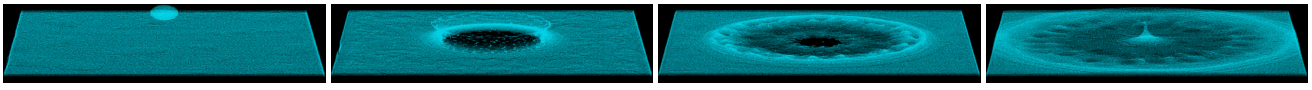
(a) Γ_0^{180} *(lowest resolution LOD) at 1000 fps*

(b) Γ_1^{180} *at 606 fps*

(c) Γ_2^{180} *at 250 fps*

(d) Γ_3^{180} *(highest resolution LOD) at 17 fps*

Figure 8: *Visualization of* Γ^{180} *at different resolution (from coarse to fine) from Dataset 1.*



(a) Γ_2^{50} *at 62 fps*

(b) Γ_2^{100} *at 71 fps*

(c) Γ_2^{150} *at 71 fps*

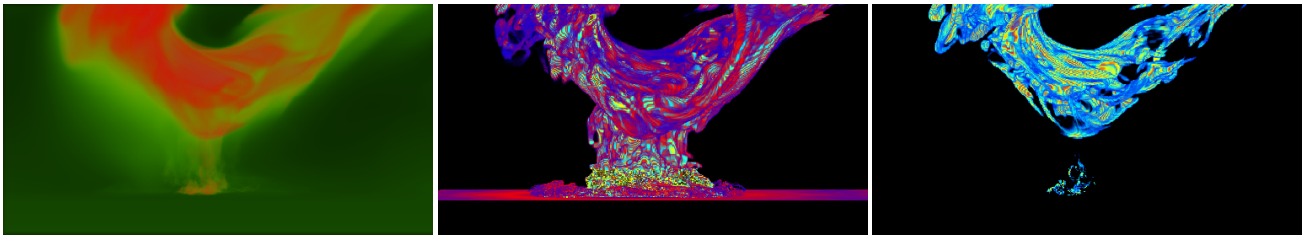
(d) Γ_2^{200} *at 62 fps*

Figure 9: *Visualization of a sequence of 4 time steps from the binary Dataset 2 in the most visible resolution.*

Figure 7 shows the user interface of our visualization application. We chose to display essential information such as the current rendering's fps, *cache used* percentage to track the GPU's data cache usage, and the application's *average time to render* a single

Dataset	Protocol	Data Cache	Mean	Median	Range	CV
1: 460 × 280 × 240 × 269 voxels encoded on 128bits of 4 scalar fields Study of asteroid impacts in deep ocean water Total size: 124GB (31 million cells per time step)	1	COLD	124.61ms	114.38ms	112.93ms	27.57%
		HOT	81.43ms	84.54ms	79.53ms	21.02%
	2	COLD	151.62ms	147.14ms	89.65ms	12.46%
		HOT	150.72ms	145.24ms	59.30ms	9.35%
	3	COLD	1.27ms	1.17ms	7.32ms	60.14%
		HOT	0.58ms	0.57ms	1.50ms	43.59%
2: 1000 × 1000 × 1000 × 311 binary voxels Free-surface fluid simulated by a semi-discrete partial optimal transport algorithm Total size: 1.13TB (1 billion cells per time step)	1	COLD	970.90ms	981.08ms	984.12ms	18.71%
		HOT	850.10ms	799.80ms	467.43ms	15.01%
	2	COLD	5377.04ms	5383.19ms	923.28ms	3.69%
		HOT	5377.18ms	5429.16ms	1137.92ms	4.83%
	3	COLD	1.83ms	1.48ms	3.00ms	51.65%
		HOT	0.88ms	0.84ms	4.16ms	34.69%

Table 1: *Performance Evaluation Table. These values represent the time to load AMR data in milliseconds (ms) from disk to GPU memory in three protocols, the first (1) evaluates the time to load a full time step of data, the second (2) the time to load a potential single frame from a single time step, and (3) the time to load a single AMR brick. Each protocol is evaluated from two data cache states, (COLD) an empty data cache and (HOT) a full data cache. The notation* CV *stands for coefficient of variation expressed as a percentage.*



(a) *Temperature at 47 fps* (b) *Volume fraction water at 52 fps* (c) *Volume fraction of asteroid at 43 fps*

Figure 10: *Visualization of* Γ^{268} *with 3 different scalar values from Dataset 1.*

frame. We also added dedicated windows to manipulate *time series* rendering, *transfer function* modification, and finally *Params* to handle various parameters such as background color and crop volume. The *time series* windows allow us to navigate through the sequence all at once or in small steps during the visualization. Thanks to our out-of-core mechanism to handle on-the-fly loading of the AMR data, coupled with our unique identification of each brick $\beta_{\tau,i,\delta}$ by the triplet (timestep, LOD, SFC index), we ensure dynamic navigation within the time series. Our application allows the visualization of a BS-AMR sequence by time steps, as shown in Figure 1, which shows the volume rendering results and performance for Dataset 1 on a sample of 4 time steps. We observe different rendering performances between different time steps mainly because the AMR hierarchy is not static throughout the sequence. Therefore, if there are fewer bricks in a given time step, the rendering algorithm has fewer samples to compute, and the more bricks in a time step, the more samples are needed. Furthermore, as shown in Figure 8, we can select the LOD to be rendered for each time step and improve rendering performance at the expense of render quality by selecting a coarser resolution, and vice-versa. Similarly, we can visualize a sequence from binary Dataset 2 on a sample of 4 time steps at the most visible resolution possible in Figure 9. Figure 10 shows different scalar values of Γ^{268} from Dataset 1. We achieved different rendering performances because the resulting AMR hierarchy of each scalar is related to its refinement threshold during the preprocessing step. Therefore, having different Card(Γ^{268}) per scalar value significantly affected the rendering performance for the same refinement level of a given numerical simulation sequence data with a static spatial resolution. For a coherent comparison, we maintain the same level of refinement across different scalar values to ensure the same brick size at the end of the preprocessing step. We have implemented three evaluation protocol tests. Each of them was performed both on a COLD data cache (i.e., starting from an empty one) to measure the time to load AMR data from memory disk to GPU memory, and on a HOT data cache (i.e., already full) to evaluate the same loading time along with cache system updates (deleting AMR data already in the data cache and replacing it with new AMR data). We chose to evaluate our BS-AMR sequence with a brick size of 64^3 with respect to the optimal block of voxels related by [FSK13] to compare both datasets on a common basis, since they do not share the same spatial resolution.

1. The first protocol evaluates the time it takes our system to load the entire AMR data from a single time step by loading a random full time step.
2. The second protocol evaluates the time to render a potential single frame from a single time step by requesting a random million 3D positions in the volume from a single time step. With this configuration, we measure independently of the camera viewpoint and with spatially consecutive 3D positions.
3. The last protocol evaluates the time to load a single AMR brick

from disk to GPU memory by loading a random single 3D position at random time steps. With this configuration, we measure the time to load arbitrary AMR data into GPU memory side by side with absolutely no spatial or temporal coherence (voxels that are not consecutive in either space or time).

The results of our performance evaluation results are shown in Table 1. They suggest several remarks:

- The first thing that stands out is the noticeably better performance of all three protocols when evaluated on a HOT data cache, except for Dataset 2 + Protocol 2, where the random number generator may have improved COLD over HOT. This is partly explained by the fact that the cache system is already warmed-up when the data cache is full, so subsequent calls are already optimized for GPU access memory. To avoid further outliers, we will measure the same 3D random positions for both cases in the future.
- The second point, regarding the first protocol results, we immediately see that Dataset 1 is on average faster to fully load a single time step in the data cache compared to Dataset 2. This is mainly due to the higher refinement level of Dataset 2 compared to Dataset 1, which means that many more bricks are loaded into the GPU memory by a factor of 2^3.
- The third point, regarding the second protocol results, we see that it takes ten times longer to load a brick with spatial coherence for Dataset 2 than for Dataset 1. This is partly explained by the difference in the spatial dimension between the two datasets, where Dataset 1 is defined in a 512^3 cubic size, and Dataset 2 is defined in a 1024^3 cubic size.
- The final point regarding the third protocol result is that with the COLD data cache, Dataset 1 takes slightly less time to load a single random brick with no spatial and temporal coherence than Dataset 2. This is again explained by the larger data structure traversal for the more refined Dataset2 compared to Dataset 1.

8. Conclusion and future works

We extended the out-of-core approach for static regular voxel grids to hierarchical AMR time series data. After analyzing our solution loading performance, we concluded that our approach is optimal when a BS-AMR sequence has the same number of refinement levels per scalar value at each time step to ensure the maximum spatial and temporal coherence between them. Therefore, the limitation of our approach is that each time step of a BS-AMR sequence must have the same spatial dimension throughout the entire time series. Nevertheless, our solution allows dynamic navigation through a BS-AMR sequence consisting of a dynamic AMR hierarchy between timesteps. We achieve this through efficient GPU parallelism of the virtual addressing of our 2D data structure, coupled with our unique addressing of each AMR brick by the triplet (timestep, LOD, SFC index). In this paper, we propose a scalable approach for large-scale AMR time series data. Our approach is defined as a cache system based on ad-hoc bricking identified by Hilbert's code indexing and managed by a 2D addressing data structure. This addressing data structure loads the required AMR data on the fly from disk to GPU memory. Our global pipeline starts with a preprocessing step that identifies each AMR brick by its Hilbert code. Then, we reduce the AMR hierarchy of a single BS-AMR to a one-dimensional array to navigate through the temporal dimension within our 2D TS-BAT architecture. However, at this stage we still lack a virtual memory mechanism and a preloading method based on an AI approach.

References

[BC89] BERGER M. J., COLELLA P.: Local adaptive mesh refinement for shock hydrodynamics. *J. of Computational Physics 82*, 1 (May 1989), 64–84. doi:10.1016/0021-9991(89)90035-1. 4

[BHP15] BEYER J., HADWIGER M., PFISTER H.: State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum 34*, 8 (Dec. 2015), 13–37. doi:10.1111/cgf.12605. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. of the 2009 Symp. on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, Association for Computing Machinery, pp. 15–22. doi:10.1145/1507149.1507152. 2, 3

[DCOM00] DAFNER R., COHEN-OR D., MATIAS Y.: Context-based space filling curves. *Computer Graphics Forum 19* (05 2000). doi:10.1111/1467-8659.00413. 5

[dlAC17] DE LA ASUNCIÓN M., CASTRO M. J.: Simulation of tsunamis generated by landslides using adaptive mesh refinement on GPU. *J. of Computational Physics 345* (Sept. 2017), 91–110. doi:10.1016/j.jcp.2017.05.016. 1

[FSK13] FOGAL T., SCHIEWE A., KRÜGER J.: An analysis of scalable gpu-based ray-guided volume rendering. In *2013 IEEE Symp. on Large-Scale Data Analysis and Visualization (LDAV)* (2013), pp. 43–51. doi:10.1109/LDAV.2013.6675157. 3, 7, 9

[GABJ08] GOSINK L. J., ANDERSON J. C., BETHEL E. W., JOY K. I.: Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE Trans. on Visualization and Computer Graphics 14*, 6 (Dec. 2008), 1715–1722. doi:10.1109/TVCG.2008.157. 2

[HBJP12] HADWIGER M., BEYER J., JEONG W.-K., PFISTER H.: Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Trans. on Visualization and Computer Graphics 18*, 12 (2012), 2285–2294. doi:10.1109/TVCG.2012.240. 2, 3

[KCP*02] KAHLER R., COX D., PATTERSON R., LEVY S., HEGE H.-C., ABEL T.: Rendering the first star in the universe - a case study. In *IEEE Visualization, 2002. VIS 2002.* (2002), pp. 537–540. doi:10.1109/VISUAL.2002.1183824. 2

[KEB*14] KUMAR S., EDWARDS J., BREMER P.-T., KNOLL A., CHRISTENSEN C., VISHWANATH V., CARNS P., SCHMIDT J. A., PASCUCCI V.: Efficient i/o and storage of adaptive-resolution data. In *SC '14: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), pp. 413–423. doi:10.1109/SC.2014.39. 3

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

[KWAH06] KAEHLER R., WISE J., ABEL T., HEGE H.-C.: *GPU-Assisted Raycasting for Cosmological Adaptive Mesh Refinement Simulations*. The Eurographics Association, 2006. doi:10.2312/VG/VG06/103-110. 1

[Lév22] LÉVY B.: Partial optimal transport for a constant-volume lagrangian mesh with free boundaries. *J. of Computational Physics 451* (2022), 110838. doi:10.1016/j.jcp.2021.110838. 7

[LVI*13] LEAF N., VISHWANATH V., INSLEY J., HERELD M., PAPKA M., MA K.-L.: Efficient parallel volume rendering of large-scale adaptive mesh refinement data. pp. 35–42. doi:10.1109/LDAV.2013.6675156. 2

[MA15] M. ADAMS P. C.: *Chombo Software Package for AMR Applications - Design Document*. Report LBNL-6616E, Lawrence Berkeley National Laboratory Technical Report, 2015. 4

[Max95] MAX N.: Optical models for direct volume rendering. *IEEE Trans. on Visualization and Computer Graphics 1*, 2 (1995), 99–108. 7

[ME11] MORAN P., ELLSWORTH D.: Visualization of amr data with multi-level dual-mesh interpolation. *IEEE Trans. on Visualization and Computer Graphics 17*, 12 (2011), 1862–1871. doi:10.1109/TVCG.2011.252. 2

[PBS02] PARK S., BAJAJ C., SIDDAVANAHALLI V.: Case study: Interactive rendering of adaptive mesh refinement data. pp. 521–524. doi:10.1109/VISUAL.2002.1183820. 2

[PG17] PATCHETT J., GISLER G.: Deep water impact ensemble data set. *URL:https://sciviscontest2018.org/wp-content/uploads/sites/19/2017/09/DeepWaterImpact Ensemble-DataSet_Revision1.pdf 4* (2017). 1, 7

[PMM16] PANDA S., MISHRA S., MISHRA S.: Study and analysis of multidimensional hilbert space filling curve and its applications - a survey. *Int. J. of Computer Science, Engineering and Information Technology 6* (04 2016), 01–09. doi:10.5121/ijcseit.2016.6201. 2

[SCRL20] SARTON J., COURILLEAU N., REMION Y., LUCAS L.: Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-of-Core Approach. *IEEE Trans. on Visualization and Computer Graphics 26*, 10 (Oct. 2020), 3008–3021. doi:10.1109/TVCG.2019.2912752. 2, 3, 6

[SDS15] SASIDHARAN A., DENNIS J. M., SNIR M.: A general space-filling curve algorithm for partitioning 2d meshes. In *2015 IEEE 17th Int. Conf. on High Performance Computing and Communications* (2015), pp. 875–879. doi:10.1109/HPCC-CSS-ICESS.2015.192. 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 Symp. on Large Data Analysis and Visualization (LDAV)* (Nov 2014), pp. 93–100. doi:10.1109/LDAV.2014.7013209. 2

[Wal20] WALD I.: A simple, general, and gpu friendly method for computing dual mesh and iso-surfaces of adaptive mesh refinement (amr) data. *ArXiv abs/2004.08475* (2020). 2

[WGLS05] WANG C., GAO J., LI L., SHEN H.-W.: A multiresolution volume rendering framework for large-scale time-varying data visualization. In *4th Int. Workshop on Volume Graphics, 2005.* (2005), IEEE, pp. 11–223. 3

[WHJ*01] WEBER G., HAMANN B., JOY K., LIGOCKI T., MA K.-L., SHALF J.: Visualization of adaptive mesh refinement data. vol. 4302, pp. 121–132. doi:10.1117/12.424922. 2

[WKL*01] WEBER G., KREYLOS O., LIGOCKI T., SHALF J., HAMANN B., JOY K., MA K.-L.: High-quality volume rendering of adaptive mesh refinement data. *Vision, Modeling & Visualization 522* (01 2001), 121–128. 2

[WMU*20] WANG F., MARSHAK N., USHER W., BURSTEDDE C., KNOLL A., HEISTER T., JOHNSON C.: CPU Ray Tracing of Tree-Based Adaptive Mesh Refinement Data. *Computer Graphics Forum 39* (June 2020), 1–12. doi:10.1111/cgf.13958. 2

[WZU*21] WALD I., ZELLMANN S., USHER W., MORRICAL N., LANG U., PASCUCCI V.: Ray tracing structured amr data using exabricks. *IEEE Trans. on Visualization and Computer Graphics 27*, 2 (2021), 625–634. doi:10.1109/TVCG.2020.3030470. 2

[YMW04] YU H., MA K.-L., WELLING J.: I/O Strategies for Parallel Rendering of Large Time-Varying Volume Data. In *Eurographics Workshop on Parallel Graphics and Visualization* (2004). doi:10.2312/EGPGV/EGPGV04/031-040. 2

[ZJW21] ZHOU L., JOHNSON C. R., WEISKOPF D.: Data-driven space-filling curves. *IEEE Trans. on Visualization and Computer Graphics 27*, 2 (2021), 1591–1600. doi:10.1109/TVCG.2020.3030473. 2

[ZWS*22] ZELLMANN S., WALD I., SAHISTAN A., HELLMANN M., USHER W.: Design and Evaluation of a GPU Streaming Framework for Visualizing Time-Varying AMR Data. In *Eurographics Symp on Parallel Graphics and Visualization* (2022). doi:10.2312/pgv.20221066. 2, 7