From Soup to Bricks: Fast Clustering of Fine-Grained AMR Hierarchies for Rendering on GPUs

Stefan Zellmann*
University of Cologne

Ingo Wald†
NVIDIA.

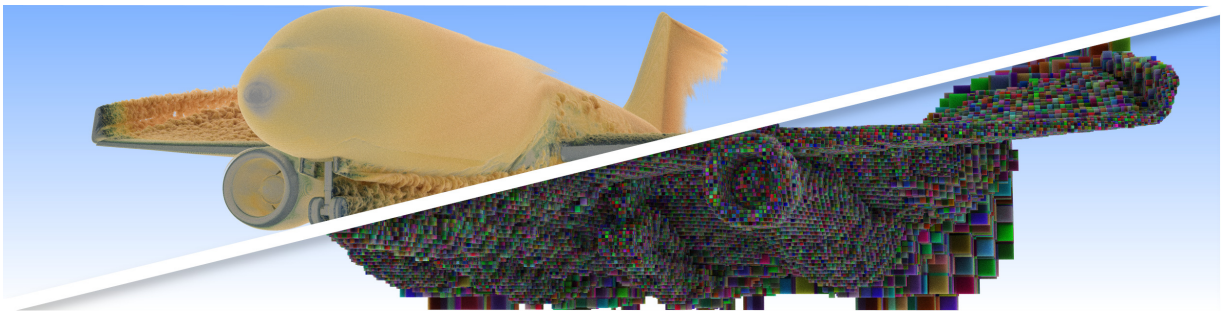


Figure 1: NASA's exajet data set. Top-left: OptiX path tracing using volumetric scattering. Bottom-right: brick partitioning (only the non-empty bricks) used for cell location as generated by our clustering algorithm.

ABSTRACT

We present two algorithms for building bricks over AMR cells on the GPU. Both algorithms focus on construction performance and produce output that allows for efficient cell location using NVIDIA ray tracing cores. In an effort to determine potential trade-offs between construction and sampling performance, we evaluate these algorithms on rendering methods common in scientific visualization.

1 INTRODUCTION

Scientific visualization and rendering applications on Cartesian adaptive mesh refinement (AMR) data incorporate optimized data structures that allow for fast random access and sampling on GPUs. As AMR software frameworks are deemed one of the critical components towards exacale computing [5], it is important that such data structures primarily aimed at post-processing run efficiently on modern GPUs that are omnipresent in today's supercomputers.

For AMR rendering the main operation is sampling, which depending on the algorithm used (ray marching, free-flight distance sampling, etc.) requires fast random access. As AMR is weakly structured, and to avoid exhaustive linear searches over cell lists, hierarchies are employed that allow for cell location in $O(\log(n))$ time.

AMR can come in different flavors, examples being block-structured AMR, forest-of-octree, or other variants that already bring their own hierarchy. In some cases these hierarchies can be directly used for sampling, by applying simple transformations and bringing them into a GPU-readable format. This is not generally the case though, as for example they might be excessively deep and too costly to traverse, or they do not fit in the GPU memory as they branch out to the individual leaf cell. Then there are cases where the AMR data does not even bring its own hierarchy since the simulation code does not write it out to its plot files.

In such cases the software needs to build homogeneous clusters (also referred to as "bricks" in this paper) itself that fulfill criteria such as convexity and shallowness of the hierarchy for efficient traversal. While early works used kd-trees with the clusters at the leaves [8], more recent works use AMR data structures optimized for hardware traversal with ray tracing cores [28]. In the latter case building these data structures becomes a two-stage process of first identifying the clusters of cells, but not the whole hierarchy. Instead, the bricks/clusters later become the primitives at the leaf nodes of a hardware bounding volume hierarchy (BVH), by using a ray tracing framework like OptiX [19], Vulkan-RT [11], or similar. BVH traversal is initiated by tracing rays, but traversal is performed by the hardware, and the sampling code only ever interacts with the bricks themselves.

While this results in highly optimized sampling data structures, efficient construction has so far been glossed over by prior works and the sampling data structures had to be built in an offline process. This allowed those works to concentrate on traversal performance exclusively, but resulted in the step of finding clusters of same-level AMR cells being costly in general, as performance had never been the focus of those works.

Making the build process more dynamic or perhaps even interactive could well result in trade-offs similar to those in building acceleration structures for ray/surface intersections, namely that a hierarchy optimized for build performance might result in inferior traversal performance, and vice versa. As this spectrum has so far been unexplored by prior works, we aim to identify exactly that sweet spot between construction and sampling performance for clustered AMR cell hierarchies, should those sweet spots exist.

To that end our contribution is centered around two algorithms we propose that are optimized for construction performance, both generating output generally compatible with Wald et al.'s ExaBricks structure [28]:

- The first algorithm clusters AMR cells on a space filling curve; bricks are not allowed to overlap so the final RT core hierarchy can be quite deep;
- The second algorithm generates bricks that can contain empty cells, which is an extension to ExaBricks and allows for a simpler construction algorithm.
- We evaluate those algorithms on different data sets using different flavors of AMR, and use different algorithms that require sampling that are typical for post processing in scientific visualization.

With that we aim to answer the question how to build good clusters for AMR sampling on GPUs equipped with ray tracing cores (RT cores) that are optimized for construction as well as iterating and sampling.

*e-mail: zellmann@uni-koeln.de
†e-mail: iwald@nvidia.com

2 Related Work

We review prior work on voxel-based data structures for rendering and post-processing. We found that works in this field place less emphasis on construction than on access or sampling performance. We are not aware of any related work on time-dependent AMR in particular. As our work shares similarity with construction algorithms for surface BVHs we also include relevant ones in our review.

2.1 Voxel Data Structures and Clustering

In volume rendering, clustering algorithms are primarily used for empty space skipping. The algorithms do not directly cluster the voxels themselves, but build kd-trees [31] or bounding volume hierarchies [27] on the GPU that bound empty and non-empty space. Fundamentally such data structures can for example also be used for unstructured data [16], as they at the beginning transition from processing voxels to using a proxy representation.

Voxel data structures are also used for quasi surface rendering where triangle meshes are voxelized and then hierarchically organized for fast access. An example of such data structures are sparse voxel octrees (SVO) [12]. An out-of-core builder for SVOs has been proposed by Baert et al. [1]. Their algorithm uses Morton codes to generate the octree, and a subdivision into smaller grids to realize the out-of-core component of the algorithm. Building octrees on the GPU from given inputs can be efficiently done with the algorithm by Karras [10]. The VDB sparse voxel representation by Museth [17, 18] provides a hierarchical voxel data structure. The OpenVDB framework comes with efficient tools for authoring and for sampling on GPUs, but is not optimized for construction performance.

2.2 AMR Rendering Data Structures

AMR renderers concentrate primarily on direct volume rendering [26] or isosurface extraction [25, 29], both of which are operations that require first order interpolants when accessing the data. As AMR data is often cell-centric that poses problems like T-junctions at level boundaries that have been addressed by prior work using different strategies [26, 29]. One such strategy is to reconstruct with a tent-shaped basis function whose footprint is proportional to the AMR level [26] (see Eq. (1) for details). These works concentrated on the interpolation problem, but did not account for memory consumption or efficient memory accesses. Locating single cells, as proposed by these works, was done using kd-trees. When built over single cells, these can have excessive memory requirements, making this approach, if not optimized, unsuitable for GPU rendering.

Another source of inefficiency when implementing ray marchers on AMR data is that cell location requires costly traversal of tree structures over the cells. Original work by Kähler et al. [8, 9] assumed that the data was vertex-centric (or the user would accept sampling artifacts at level boundaries), which allowed them to implement efficient OpenGL-based renderers computing primary visibility using ray marching. The traversal step was implemented on the CPU using a kd-tree over AMR cell clusters. Ray marching was implemented in a fragment shader that on each execution traversed a single coherent cluster. Obvious shortcomings of this approach are artifacts with cell-centric data (the AMR data sets that are available to us are all cell-centric) and inflexibility when it comes to more advanced integration methods that require tracing incoherent rays. Traversing the kd-tree with secondary rays to compute shadows or ambient occlusion was not possible with Kähler's framework as execution was tied to the fragment shader. Wald et al. [28] proposed a more flexible framework using NVIDIA OptiX that built upon a similar data structure to Kähler's.

Now allowing for secondary ray traversal, as the rendering integrator was fully implemented inside an OptiX ray generation program, Wald solved the interpolation problem by extending the bricks at the leaf nodes by half a cell's width in each direction to accommodate a reconstruction filter kernel; instead of using those bricks directly, the regions where they overlap were identified, giving the so called *active brick regions*. These are tessellated so each brick region itself becomes an axis-aligned box. By storing pointers to the original bricks, and traversing the brick regions using an OptiX BVH, it is now possible to compute basis functions for interpolation even at the boundaries. The resulting *ExaBricks* data structure proved not only useful for ray marching, but also for flow visualization [32] and path tracing [35]. The latter work also evaluated different ways of locating cells using the OptiX BVH, by using a BVH over brick regions directly, or a BVH over extended bricks that overlap. Zellmann et al. [33] later proposed a streaming framework based on ExaBricks, yet the topology of the cells did not change over time so the authors concentrated on streaming the data and rebuilding only the BVH but not the clusters.

Another way of rendering cell-centric AMR is by transforming the grids to their duals where the interpolation becomes vertex-centric. This requires adding unstructured stitching elements [15]. As the majority of dual AMR cells are still voxels though, building clusters over these is still important to keep memory consumption low. The work by Zellmann et al. [34] addressed this by clustering voxels and stitching cells into different GPU data structures. They also proposed a construction algorithm for the clusters of dual voxel cells that inspired our projection-based algorithm. This unoptimized builder executes fully on the CPU though, is not parallelized, and uses a C++ `std::map` as underlying data structure. Commonalities to our projection-based algorithm are speculative activation of macrocells on AMR level grids, and the introduction of empty cells, which in their case was necessary to cover boundary regions with unstructured stitching cells.

2.3 Dynamic Construction of Triangle BVHs

In the field of triangle ray tracing a trade-off exists for favoring either bounding volume hierarchy (BVH) construction or ray/triangle intersection performance. Classical algorithms [24] optimize BVHs using the surface area heuristic (SAH). These algorithms exhibit relatively poor thread utilization as they use top-down construction. Potential for multi-threading or vectorization lies in the binning operation, but processing of multiple tree nodes at once is only possible once a significant portion of the tree has been processed. SAH leads to superior tree quality though.

Classical works in the field of fast construction use the split-middle heuristic that can be computed without prior knowledge of parent nodes and hence in a bottom-up fashion [13]. With this algorithm called LBVH, the middle split is derived from a Morton curve built over triangle centroids. Fast construction on GPUs can be realized using Karras's algorithm [10]. The LBVH algorithm forms the basis of numerous follow-up publications on Morton code-based builders [20, 23] and has also been applied to voxel data, yet not to build clusters but an empty space skipping data structure [30].

Parallelized locally-ordered clustering (PLOC) [14] is a variant of agglomerative clustering that seeks to build SAH-optimized BVHs from the bottom up. It shares similarities with our Morton-based cluster builder. PLOC uses a Morton curve over centroids as an acceleration structure to perform neighbor search on. Beginning at the bottom where each primitive bounding box forms its own cluster, it seeks to find other clusters as candidates to merge with, and does so by finding neighbors on the Morton curve. This is done in parallel starting at the cluster itself, searching both to the left (sweep-left) and to the right (sweep-right) within a fixed radius determining the number of neighbors to visit. Pairs of clusters that identified each other as the closest neighbor are merged; the algorithm continues iteratively until the full hierarchy is built. Extensions to this algorithm have later been proposed to merge and by that reduce the number of GPU kernel calls [2].

3 Background

We provide some background on Wald et al.'s ExaBricks data structure [28] that we use in this paper. We refer to [28, 33, 35] for a more complete review. The input to the data structure and rendering algorithms based on that are what we refer to as "soup of cells": one array with cells, each comprised of its voxel coordinates $(x,y,z) \in \mathbb{N}^3$, and its AMR level $l \in \mathbb{N}$ (smaller levels indicate finer cells). A second array of same length as the cell array holds the scalar data. The algorithm to build this data structure proceeds in multiple phases of which only the first one is relevant to this paper: building non-overlapping yet coherent bricks of cells. For that, only the cell input is necessary, not the scalars. Given those bricks, ExaBricks proceeds to perform cell reconstruction while guaranteeing C^0 smoothness at level boundaries. We use the *extended bricks* variant of ExaBricks described in [35]. Extending them results in the bricks overlapping even if their cells are on different AMR levels.

To reconstruct a sample, the tent basis reconstruction method [26] is used giving the sample value as

$$B(p) = \frac{\sum_{C_i} \hat{H}_{C_i}(p) C_{v_i}}{\sum_{C_i} \hat{H}_{C_i}(p)}. \quad (1)$$

This method associates each cell C of width C_w (which is proportional to the AMR level) with a tent basis:

$$\hat{H}_C(p) = \hat{h}\left(\frac{|C_{p_x} - p_x|}{C_w}\right)\hat{h}\left(\frac{|C_{p_y} - p_y|}{C_w}\right)\hat{h}\left(\frac{|C_{p_z} - p_z|}{C_w}\right), \quad (2)$$

where $\hat{h}(x) = \max(1-x, 0)$. To evaluate Eq. (1) the bricks must be extended by half a cell's width in each direction. Evaluation is performed using a neighborhood search realized with NVIDIA OptiX [19]. We use an intersection program that tests for overlap with all the (extended) bricks and the ray origin that was chosen to be aligned with the sample point (the direction is irrelevant and the ray effectively has zero length). The intersection program will trigger on each brick domain that overlaps; the addends going into the two sums of Eq. (1) are iteratively added. When the ray finished traversing the hierarchy the fraction is evaluated, which yields the final sample value. We chose this variant of ExaBricks for its simplicity and because of its minimal requirement for pre-processing, which lends itself to interactive construction.

An alternative variant of ExaBricks does not use the extended bricks directly, but further tessellates them into boxes called the *active brick regions* (ABR). This tessellation is finer, yet the boxes do not overlap, so once an ABR was located, locating the bricks to perform cell location is done by iterating over a linear list.

Extended bricks (or ABRs) are generated only once, when new data is available, e.g., because another time step was loaded from the simulation, etc. In addition to their bounds, the bricks also store min/max cell ranges which need to be computed on brick generation. Given these, the OptiX BVH can be rebuilt as required, i.e., potentially more often than clusters/bricks are generated. This can be done by testing the ranges against an alpha transfer function, culling empty bricks by emitting empty bounding boxes for which OptiX does generate leaf nodes. The resulting data structure can, e.g., be used for ray marching [28] or path tracing [35]. Our focus is to make this whole pipeline interactive, with the main bottleneck so far being the brick builder at the beginning. The whole pipeline is summarized in Fig. 2.

4 Problem Statement

The problem we address in this work is constructing homogeneous bricks over cells on the same AMR level. We aim for compatiblity with Wald et al.'s ExaBricks [28], i.e., the bricks will form the leaf nodes of a BVH that we build with NVIDIA OptiX. Generating the full hierarchy is optional and therefore not supported by the algorithms we propose. The bricks must either form a full space decomposition (as in prior work) or, as a necessary condition, fully cover all the cells present on the respective AMR level. There may be different reasons for building such bricks. Some codes, such as PowerFLOW [4] used for the Exajet from Fig. 1, do not write out any hierarchy other than the cells themselves. In other cases, for example when ray marching the spatial field, clusters or bricks coming with the data, and their hierarchy, are overly fragmented. Octree structures popular in AMR are an example where only a few cells share a leaf, but neighboring leaves in adjacent branches may contain cells of the same level, causing ray marchers to traverse over and over even in regions that are (topologically) homogeneous.

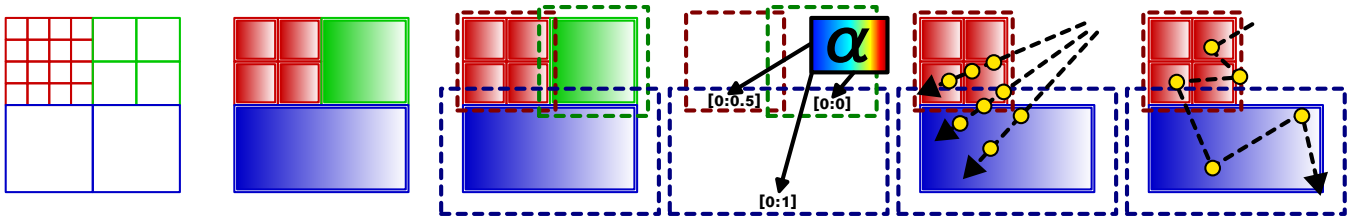
Given a brick partition, we use the *extended brick* variant of ExaBricks [35] to assemble the sampling data structure: the originally non-overlapping bricks are entered into the OptiX BVH not with their bounds, but with their *filter domain bounds* which are half a cell wider in each direction than the actual brick bounds—and consequently, these brick domains overlap.

The overarching question now becomes what is a "good" brick partitioning both in terms of construction as well as sampling performance. Multiple factors play a role and might be detrimental to one or the other:

- **Brick size and shape:** larger bricks result in shallower hierarchies, potentially reducing the number of traversal steps. Smaller bricks might allow the rendering or visualization algorithm to better utilize caches due to their smaller strides when accessing individual cells.
- **Split heuristic:** how the AMR is subdivided is determined based on a heuristic. In neighboring fields such as ray/triangle intersection with BVHs the split heuristic is one variable to optimize for either construction or traversal performance. For RT core-based AMR data structures the influence of the split heuristic has not been evaluated by the literature yet.
- **Presence of empty cells:** the original publication on ExaBricks [28] did not allow bricks to contain cells to skip over. This can result in very finely tessellated regions of space incurring high traversal costs. Allowing for empty cells may result in fewer bricks, yet at the expense of added costs when evaluating and storing the empty cells.
- **Access type:** performance depends on whether the sampling algorithm performs random access or if the accesses are coherent. If, e.g., several coherent samples are taken along a ray from the same brick, the costs of that can be amortized across those samples.

We propose two algorithms to partition the AMR data into bricks. In regard to the brick shape these algorithms incorporate simple heuristics such as middle-split when partitioning cells where the original ExaBricks paper used a variant of the surface area heuristic (SAH) for voxel cells. While split-middle and similar strategies allow for faster construction, the impact on sampling AMR cells is unknown.

We chose to evaluate two fundamentally different algorithms optimized for fast construction on GPUs: one that draws inspiration from Meister and Bittner's PLOC algorithm [14]. This algorithm reorganizes the cells on a space filling curve, expands them and then tries to merge with neighboring clusters. The second algorithm starts with a given macrocell partitioning that guides the brick creation, and then projects the AMR cells into those macrocells. While macrocells that are empty can be removed, other macrocells are shrunk (and by that become bricks) but still contain empty cells. Instead of laboriously trying to remove these we extended ExaBricks to skip over them at runtime, incurring a potential performance hit, which might however be compensated by a shallower sample BVH. The



(a) (b) (c) (d) (e) (f)

Figure 2: Problem addressed in this paper. (a) Starting from a given set of unordered AMR cells, we calculate coherent bricks in (b). Fast builders for that are the main contribution of this paper. Our sci-vis renderers build a BVH for sampling over extended ExaBricks using OptiX (c). This BVH is rebuilt whenever the alpha transfer function changes (d). In this example the green brick becomes invisible as per the alpha transfer function, and its BVH leaf gets invalidated. Renderers can exhibit coherent accesses as in volume ray marching (e) or incoherent accesses as in Monte Carlo path tracing (f). Though our main focus is on fast builders, we aim to make this entire pipeline interactive.

concept of allowing empty cells in brick data shares commonalities with the CPU builder by Zellmann et al. [34]; there, inserting empty cells was motivated by overlapping stitching cells. We describe these two algorithms in detail in the following sections.

To determine the trade-off between construction and sampling performance we perform an evaluation using rendering algorithms with different access characteristics. We compare those for the brick configurations generated with our algorithms, as well as with the original ExaBricks. This is described in detail in Section 6.

5 GPU Clustering Algorithms

In the following we present two algorithms for efficient AMR cell clustering. The input to both algorithms is a list of AMR cells e explicitly represented by their voxel position $vp \in \mathbb{N}^3$, their AMR level l, and an integer offset into a scalar field to obtain the voxel value. The output is a list of clusters (or *bricks*) that store axis-aligned bounding boxes, AMR level, and implicitly the AMR cells as a 3D array in memory. As an implementation detail that array can either contain scalar values directly, or IDs in case we support multiple scalar fields.

5.1 Algorithm 1: Morton Builder

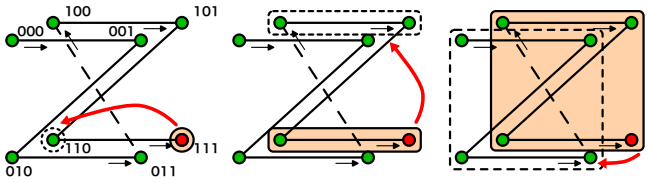
The first algorithm we propose (Algorithm 1) sorts the AMR cells on a Morton curve and then tries to merge them while maintaining convexity. For brevity the pseudo code assumes that the input is already sorted, giving us AMR cells e_l^z where l is the AMR level and z the Morton index.

Initially the cells each form their own cluster and trivially fulfill the convexity requirement. In a next, parallel step the clusters are expanded along the Morton curve as outlined in Section 5.1.1. Then, clusters are identified that overlap each other, and finally merged as described in Section 5.1.2.

To increase utilization in a GPU-parallel implementation, we maintain a global array of cells for all levels. In the respective sublists the cells are sorted by their Morton code. When we perform any of the operations described in the following, we first determine the cell range for the level we are currently on. This is achieved using a prefix array giving us `begin` and `end` sentinels per level. Given these we can execute the algorithms for each cell, regardless of their AMR level, but need to make sure to clip any loop counters, like those for the left and right sweeps below, against these sentinels.

5.1.1 Convex Z-Order Subsets

An integral part of this algorithm is tracking runs on the Morton curve starting at each cell and trying to continue the runs as long as all the cells on the run form a convex set. All cells on a run become cluster candidates that we later merge and prune to form a space decomposition. The maximum length of the runs is determined by the desired brick size; we aim for bricks of maximum size $8 \times 8 \times 8$ cells (for both algorithms), so the maximum run length is 8^3.



(a) (b) (c)

Figure 3: Operations performed during the "sweep-left" phase of our run tracking algorithm. If we are on a right vertex on the horizontal, we search for the neighboring left vertex to form a convex set (a); this is likewise done for pairs of edges (b) and faces (c). The corresponding sweep-right phase searches in the opposite direction and performs equivalent operations.

Let $\mathscr{Z}^{(i)}$ be the Morton code of the cell at i. We start tracking runs at that cell i, first in the right and then in the left direction, aiming to find the longest run within the given range.

When tracking runs from left to right we check for the invariant that $\mathscr{Z}^{(i+1)} = \mathscr{Z}^{(i)} + 1$; and likewise, when tracking runs from right to left the invariant must hold that $\mathscr{Z}^{(i-1)} = \mathscr{Z}^{(i)} - 1$; i.e., we track runs along the Morton curve until we find a gap, or the maximum run length is reached. We refer to this as the *Morton-completeness property*, where only vertices, horizontal edges, or vertical faces adjacent on the Morton curve can form a cluster.

In the following we provide a definition of vertices, edges, and planes adjacent on the Morton curve. Given a position $p \in \mathbb{R}^3$, its corresponding Morton code comprises 3-tuples of bits:

$$\mathscr{Z}_0 = x_{n-1}y_{n-1}z_{n-1}\cdots x_0y_0z_0,$$

where $x_i, y_i, z_i \in [0,1]$. We say that $\mathscr{Z}_0$ is a vertex on Morton level 0. Likewise,

$$\mathscr{Z}_1 = x_{n-1}y_{n-1}z_{n-1}\cdots x_1y_1z_1000$$

(the three least significant bits zeroed out) is a vertex on Morton level 1, and so on. Logically, a vertex represents one octant of the octree induced by the Morton curve, on a chosen octree level. We define edges to be comprised of two neighboring vertices and, without loss of generality, are only interested in those edges that are horizontal. A face comprises four vertices that are adjacent and share the same x_i and y_i bits. The entities highlighted in orange in Fig. 3 are examples of vertices, edges, and faces.

Morton-complete runs are a necessary condition for finding bricks; another condition is that those bricks are convex, i.e., if we have for example found an edge, finding the vertex adjacent to that edge when continuing the run is not sufficient. Instead we must find the two vertices on the run that form the neighboring edge. Similarly, if we have found the first four vertices on the Morton curve within a run that form a front face (or back face) we must find the neighboring four vertices forming either the back face (to the right) or the front face (to the left) of the face we found.

Consequently, while tracking runs, we also need to keep track whether we currently search for a neighboring vertex, edge, or face.

Algorithm 1 uses run tracking to grow and merge clusters while staying convex and becoming no larger than $8 \times 8 \times 8$ in size.

Input: list of sorted cells $E = e_0^0, e_0^1, .., e_0^z, e_1^0, e_1^1, .., e_L^{\mathscr{Z}}$
Output: list of clusters/bricks $c \in C$

$C = E$ ▷ Start with each cell being a cluster
▷ `Expand`: grow clusters to max size boxes
for $e \in C$ **do in parallel**
    EXPAND(e)
end parfor
    ▷ `Find overlap`: identify which cluster overlaps which
for $e \in C$ **do in parallel**
    parent[e] = e
    for e_2 =CLUSTERSSURROUNDING(e) **do**
        if e_2.CONTAINS(e) and $e_2 >$ parent[e] **then**
            ▷ Assign biggest clusters we find as parents
            ▷ We will later only keep the parents
            parent[e] = e_2
        else if $e_2 = e$ and e_2.LEFTOF(e) **then**
            ▷ Found same cluster: designated one wins
            parent[e] = e_2
        else
            ▷ Cannot get here, all clusters are on Morton curve
        end if
    end for
end parfor
▷ `Merge`: keep only the largest clusters
for $e \in C$ **do in parallel**
    ▷ In parallel, move children into parents
    $n = e$
    while parent[n] $\neq$ parent[parent[n]] **do**
        parent[n] = parent[parent[n]]
        parent[e] = parent[n]
    end while
end parfor
for $e \in C$ **do in parallel**
    ▷ Mark all other clusters for deletion
    if $e \neq$parent[e] **then**
        $e = \emptyset$
    end if
end parfor

We start our runs at the original cell and first perform a sweep right and then a sweep left. We first *snap* to a position where we can immediately check for the two neighboring vertices on the horizontal if they are Morton-complete. If they are not the original cell forms a one-cell cluster. We perform this snap operation to give us a starting position simplifying the control flow used during run tracking.

Generally, on the sweep right, if we found a left vertex we test if we also find the corresponding right one; if we are on a top edge we search the corresponding bottom edge, etc. For the left direction we perform the equivalent operations. Finding out which vertex, edge, or face we are on can be done by inspecting the bit pattern of the Morton index where the current run halted. A graphical explanation of this search algorithm is shown in Fig. 3.

Finding the neighboring vertices, edges, or faces, where required, guarantees that the runs we track correspond to a convex set of level cells. When performing this run tracking algorithm we would however only be able to find bricks of size $2 \times 2 \times 2$ because once we found that brick, adding another vertex, edge, or face on the Morton curve would result in the set no longer being convex.

Instead, when we found a $2 \times 2 \times 2$ brick of cells we interpret that brick as a vertex on the next higher level of the octree induced by the Morton curve. We then repeat the same algorithm from before, tracking runs comprised of vertices, edges and faces, yet on that next higher level. An edge on that level, e.g., will be comprised of $4 \times 2 \times 2$ cells. This procedure can be recursively repeated up to the maximum depth that is also given by the maximum run length.

The corresponding implementation does not use recursion though, but instead keeps track of run lengths and the current level we are on. For bricks of size $8 \times 8 \times 8$, e.g., we simply hardcode the maximum number of Morton levels to be $\log(8) = 3$. Encoding the neighbor relationship of what are the next Morton indices we expect can be implemented using a lookup table.

5.1.2 Cluster Merging

After the run tracking phase (denoted `Expand` in the pseudocode) we end up with each cell still corresponding to one cluster, but those clusters are now expanded and overlap or even have the exact same spatial extent. Also note that momentarily the clusters are represented by ranges on the Morton curve indicating which cells will become part of the cluster. In the next step we have to identify which ranges overlap and contain each other and then choose which ones to keep and which ones to reject.

We first observe that due to the Morton property, the clusters and corresponding ranges cannot overlap arbitrarily; due to the Morton-completeness property, clusters rather overlap if their extents are identical, or if one cluster is on a higher level than the other and is then hierarchically a parent to it. In either case, if one cluster is fully contained, and if one extent is larger than the other we keep the bigger cluster and discard the other one. If two clusters overlap and have the same extent, we have to ensure that these do not cancel due to circular dependency, so we arbitrarily only keep the left one.

With these two rules we can iterate over each range and perform a search over all its neighbors within the 8^3 run length, comparing with all the other ranges and deciding if we keep or discard the current one. This can be done in parallel, but comes at a relatively high cost per cell because a linear search is necessary, as we do not know in advance the sizes of the up to 8^3 neighboring ranges. A potential optimization could be to build a search data structure at this point, but we did not explore this as it would come with other cost constraints.

While iterating over the cells and performing neighbor lookups, we must be careful to avoid race conditions as all the steps involved are performed in parallel GPU kernels. Instead of physically removing ranges we instead just update their state using a *parent list*. If a range has a parent that is not the range itself, the other range has higher precedence than the current one according to the rules from before.

Given the spatial arrangement, clusters will in general have multiple ancestors, e.g., one cluster being contained in another, which is again contained in a cluster that is bigger, and so on. We resolve this by traversing for each cell the parent list up to the final ancestor, which will be associated with the cluster with the largest volume, and keep only this one. After this operation, clusters that are not contained in other clusters will have themselves as parent.

Consequently we only keep those latter clusters that are guaranteed to be the largest ones. Based on this criterion, deleting the ranges that have a parent that is not themself is trivially done in another compute kernel (a final compute kernel forming bricks we omitted from Algorithm 1 for simplicity).

5.2 Algorithm 2: Cell-to-Grid Projection

Our second algorithm (Algorithm 2) projects cells onto a given macrocell partition per AMR level. Starting with this partition, the macrocells will eventually form our bricks, but we must first prune them, and shrink them to their final size, and remove empty bricks where this is possible. Pruning and shrinking is done in a single logical step—in the code referred to as `Resize`—but we illustrate the two concepts separately, in Fig. 4 and Fig. 5.

Algorithm 2 partitions each level into macrocells of size $8 \times 8 \times 8$, then projects the AMR cells into those macrocells, and then shrinks the macrocells to fit to obtain the final clusters.

Input: list of cells $e \in E$
Output: list of clusters/bricks $c \in C$

▷ Generate list of macrocells for all levels L
MC_L =FROMBOUNDS(L)
INVALIDATE(MC_L)
▷ `Resize`: find macrocell sizes via projection
for $e_l \in E$ **do in parallel**
    $mc \leftarrow$FINDMC(MC_L, e_l)
    INSERTINTO(mc, e_l)
end parfor
▷ `Init arrays`:
▷ Array to map cells before/after compation
$Map \leftarrow \emptyset$
for $mc \in MC_L$ **do in parallel**
    $Map[mc.ID] \leftarrow mc.ID$
end parfor
▷ Array of cell offsets
$O \leftarrow \emptyset$
for $mc \in MC_L$ **do in parallel**
    $O[mc] \leftarrow$ VOLUME(mc)
end parfor
REMOVEIFZERO(O)
$O \leftarrow$ EXCLUSIVESCAN(O)
▷ Allocate list of clusters based on offsets
ALLOC(C, O)
▷ `Populate`: fill clusters with cells
for $e_l \in E$ **do in parallel**
    $mc \leftarrow$FINDMC(MC_l, e_l)
    $c \leftarrow C[Map[mc.ID]]$
    ADDCELLTO(c, e_l)
end parfor

Our method can result in empty cells at the boundary even if we shrink the macrocell bounding boxes to minimal size (cf. Fig. 5); for this algorithm we allow for empty cells inside the partition. This is conceptually similar to [34] only that in their case the empty cells would cover generally unstructured dual AMR cells used for stitching.

Let the *logical grid* for level l the *dense* Cartesian grid that would contain all the cells on l. Cells on the logical grid over l are represented by their *logical grid coordinates* $X \in \mathbb{N}^3$. To transform these cells into the global voxel space and back one would multiply and divide their sides by 2^l. This allows us to treat these grids independently. With this representation it becomes irrelevant that our data is AMR. As before, we maintain all the data on each level in global arrays and use range sentinels to avoid out-of-bound accesses.

We first compute the logical grid bounds for each level. That gives us a spatial domain (still in logical grid coordinates) per level that we can subdivide into macrocells. It is possible that the AMR cells inside this domain have empty space inbetween, or at the boundary, so all the macrocells are candidates to become bricks, and in general some will be removed and others be shrunk so they will not retain their original size. As all macrocells start out empty, filling them with AMR cells "activates" them. In general, the set of AMR cells inside a macrocell—even after shrinking the bounding boxes—is not convex. We therefore mark the empty cells using a negative index into the spatial field associated with the data set (cf. Fig. 5a).

The macrocells themselves do not require much memory if just represented by their bounding boxes. But if we were to allocate them to speculatively hold all the AMR cells they could possibly hold—regardless whether they eventually turn out empty or not—

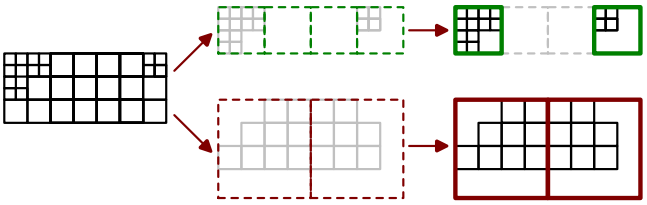


Figure 4: `Resize` phase of Alg. 2: macrocell activation and pruning. Starting with the AMR grid on the left, we subdivide the logical grid of each level into macrocells (middle, macrocell size 4×4). The dashed outline signifies these macrocells are candidates to become clusters. We then activate the macrocells that hold AMR cells (right, solid outline for active, dashed/gray for inactive). Before allocating memory for cell IDs, the active macrocells are shrunk (cf. Fig. 5).

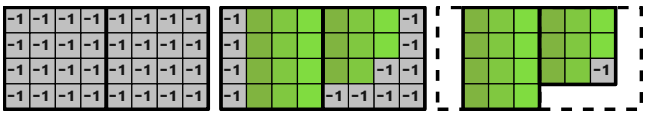


Figure 5: `Resize` phase of Alg. 2: box shrinking. (a) all AMR cells are initially empty (`-1`, these cells are not represented in memory) and macrocell bounds marked invalid. (b) we project all AMR cells to their macrocell slot. (c) the original macrocells are marked by a dashed outline; the clusters are defined by the axis-aligned bounding box containing all the cells (including potentially empty ones).

we would use more scratch memory than is available on the GPU. Therefore, we subdivide the part of the algorithm where macrocells get populated into two phases.

The first phase is comprised of identifying the macrocells that become active by projecting AMR cells (cf. Fig. 5b) and shrinking them to their final size (Fig. 5c). This gives us the brick bounds of the final partition. With that representation we also transition from an implicit grid (of macrocells) to a compact list (of bricks).

We compute the axis-aligned bounding boxes of each macrocell by projecting all the AMR cells on all levels in parallel. Given those bounding boxes we compute an offset array initially containing the number of AMR cells in each macrocell, computed using the previously determined bounding boxes. This offset array we compact and scan, and while doing so keep track of which original macrocells go to which new slots in the compact list using another array for mapping.

In the second phase we fill the macrocells that have now become bricks with AMR cells. For that we first allocate the memory to hold all the AMR cells given the previously computed prefix sum over the compact offset array. Note that the memory allocated also includes all the empty cells. In another parallel `for` loop we project the AMR cells again, only this time, given the preallocated memory buffer and the array mapping macrocells to bricks we can now write the AMR cells directly to their respective location in memory. By initializing the cell buffer to -1, those locations that do not receive a cell via projection automatically become empty.

This gives us our final brick representation in memory. We again omit kernels to convert to a format compatible with ExaBricks; in the implementation-dependent case where the ExaBricks store the scalars directly instead of cell IDs, we also convert negative IDs associated with empty cells to a special floating value, such as `NAN`, that the renderer can later identify during sampling.

5.3 Implementation Details

We implement both algorithms with CUDA. The source code is included in the additional material of this paper. We first copy the AMR cells E to the GPU by allocating device memory and filling it from pinned host memory to be sure to use the fastest possible CUDA runtime functions for this. As we target a GPU rendering implementation the final output clusters C stay in device memory. In

the pseudoce (Algorithms 1 and 2) all parallel `for` loops correspond to CUDA kernels.

Both algorithms start by a common initialization phase (omitted in the pseudo code) to compute the min/max AMR level of the whole data set, and for each level the bounds of the logical grid. Algorithm 1 requires this information to transform cells to object space before computing Morton codes. Algorithm 2 requires it to allocate a conservative number of bricks for the macrocell grids. We implement this by anticipating the maximum level as 64 and allocate constant size arrays for the bounds, allowing us to use shared memory atomics to implement this phase. Only if the maximum level turns out to exceed 64 we rerun this phase using a more general kernel without this optimization, which is significantly more costly. This does not happen for any of the data sets we use for the evaluation though, and we consider the AMR level maximum of 64 to be quite conservative.

For the sorting phase in Algorithm 1 we use CUB and sort the cells in a single sweep, by using 64-bit primitive references into the array of cells. We encode the Morton code in the lower 56 and the level in the upper 8 bits of the primitive reference, allowing us to sort the cells per level and in space at once. In Algorithm 2 we use device atomics to resize the macrocells; here we cannot use shared memory atomics as this is (logically) a `scatter` operation.

Both algorithms have a phase where the bricks are written out to memory in their final layout. This brick assembly phase is more distinctive in Algorithm 1 though as it requires to copy cell IDs to their new memory location; while in Algorithm 2 the phase that projects the cells to their slot in the compact macrocell grid already accounts for this.

6 Evaluation

We evaluate if our builders are fast on the GPU. At the same time we are interested in the trade-off between construction and sampling performance. For sampling it will make a difference if the memory accesses are coherent or incoherent. We proposed two different algorithms, one that starts with the cells and iteratively transforms them into clusters; and a second one that starts with a tentative configuration of (overlapping) clusters and prunes and shrinks those until minimally bounding the cells. The two algorithms differ in that one uses a costly neighbor search and the other performs a scatter operation using global atomics. While Algorithm 1 will result in a fine tessellation, and with all bricks aligned on a Morton curve, Algorithm 2 will result in overlapping boxes that are less strictly aligned. All these variables will contribute to the execution time of the algorithm and quality of the resulting clusters.

6.1 Data Sets

We use the data sets in Fig. 6. These were also used by prior work on AMR volume rendering using efficient GPU data structures [28, 35] and represent a variety of different Cartesian AMR types. For comparability we reuse cameras and transfer functions from prior work.

LANL Impact. A meteor impact simulation [21] generated with xRage [7]. We use time steps $t = 5,700$, $t = 20,060$, and $t = 46,112$ of the *tev* field for the evaluation. The number of cells per time step are: 283 M, 158 M, and 26.8 M cells, across four AMR levels each.

TAC Molecular Cloud. The data set was generated with the "SILCC-Zoom" simulation [22], which uses FLASH [6]. The output is a forest of octrees with $16 \times 16 \times 16$ leaf nodes. The data consists of 80.6 M AMR cells distributed across four AMR levels.

NASA Landing Gear. This data set was generated with Chombo [3], which produces block-structured AMR. The spatial arrangement of the time step we have access to causes massive "teapot in a stadium" issues as large boundary cells overlap heavily. 262 M cells, 13 AMR levels.

Data set	ExaBricks	Alg. 1	Alg. 2
LANL 5k	14.71 sec	132.2 ms (103.3 ms)	52.10 ms (23.53 ms)
LANL 20k	89.32 sec	672.6 ms (506.1 ms)	231.2 ms (69.71 ms)
LANL 46k	189.5 sec	1181. ms (883.7 ms)	407.3 ms (112.5 ms)
Cloud	33.71 sec	340.4 ms (255.2 ms)	121.4 ms (58.39 ms)
Gear	354.3 sec	1029. ms (754.3 ms)	379.5 ms (105.3 ms)
Exajet	551.0 sec	2931. ms (2283. ms)	884.5 ms (189.9 ms)

Table 1: Direct construction performance, our two proposed algorithms compared to a (TBB-parallel) version of the original ExaBricks builder. We present the total time including copying the cell data from CPU main memory to the GPU, and the bare compute time once the data is on the GPU, in parentheses.

NASA Exajet. This is the biggest data set we use, comprising 656 M cells across four AMR levels. The PowerFLOW simulation code [4] generates forest of octree AMR; the plot files written out by the simulation do not contain any hierarchy information other than the cells themselves.

All original data sets are comprised of multiple time steps, but for the cloud and landing gear data we only have access to a single one. Exajet is special in that only the scalar data changes over time, but not the cells. In an in-situ scenario, e.g., one would build the bricks only once, though subject to different scalar data per time step the generated *extended brick* partition and BVH derived from that constantly change. That does not affect cluster construction though.

6.2 Brick Construction

Our primary goal in this work is to significantly improve the construction performance of building (Exa-)bricks to address this bottleneck in the overall pipeline. We first focus the evaluation on brick construction performance, comparing the two algorithms we presented in Section 5. To evaluate we use a system equipped with an NVIDIA RTX 4090 GPU. The CPU used is an Intel Core i7-14700K with 64 GB main memory.

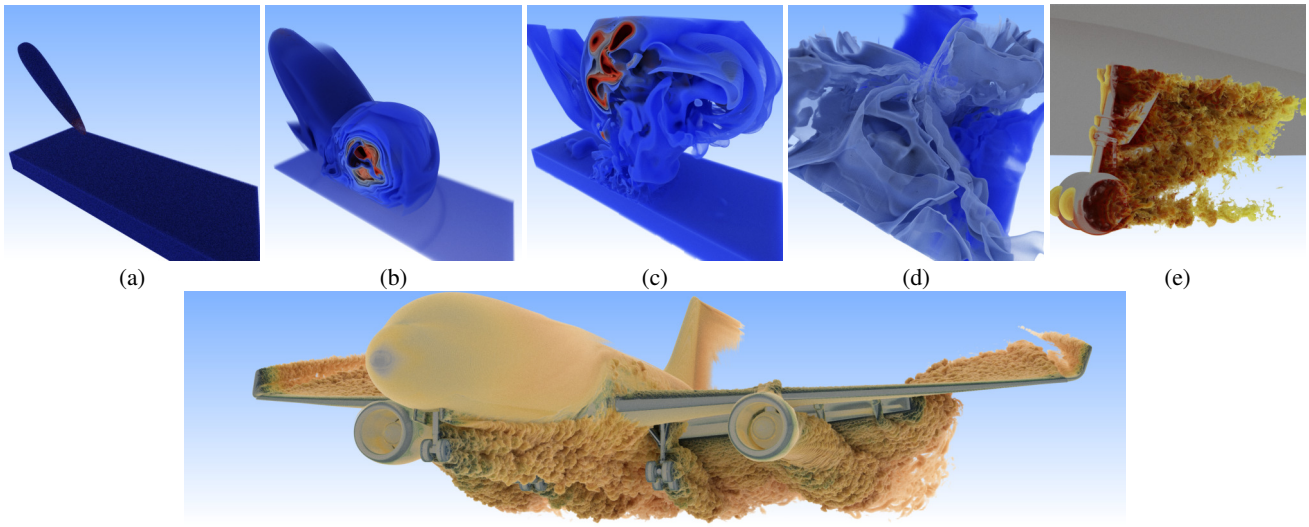
Direct Performance Comparison

We first compare the performance of our algorithms against the original ExaBricks clustering algorithm [28], which uses a heuristic similar to SAH to build clusters in a top-down fashion. This algorithm is by no means optimized, as that was not the focus of the original publication. It is written in C++ for the CPU. It does not use (explicit) vectorization. Portions of the code that distribute cells to leaf nodes are parallelized with TBB, but due to the top-down nature the utilization of this algorithm is rather low.

We present results for building clusters with the three algorithms (ExaBricks, and our two new ones) in Table 1. For the GPU algorithms we report timings including host-to-device copies (using `cudaMallocHost` and `cudaMemcpy`), but also report the pure compute times in parantheses. This acknowledges that in scenarios like in-situ the data might already be on the GPU.

Performance by Phases of Algorithm 1

Algorithm 1 consists of multiple phases. The first phase ("level bounds") is shared by both algorithms and computes min/max levels and bounds. The second phase sorts the AMR cells by level and Morton code ("sort"). The third phase ("expand") finds maximal runs on the Morton curve. The fourth phase ("find overlap") performs neighbor searches to identify which clusters overlap, and keeps tally of the biggest overlapping clusters in a parent list. In phase five ("merge") the parent list is pruned so that all clusters either have the biggest cluster, or themselves as parent. Clusters that overlap with a bigger (parent) cluster are removed, yielding the final partition. Finally the bricks are assembled and cells copied to their memory location ("assembly"). We report the time for this algorithm broken down by individual phases in Table 2.



(a) (b) (c) (d) (e) (f)

Figure 6: Data sets used for the evaluation. (a-c) LANL meteor impact data set at time steps t=5,700, t=20,600, and t=46,112. (d) TAC molecular cloud, (e) NASA Landing Gear, (f) NASA Exajet. The images were rendered with the path tracing benchmark.

	LANL 5k	LANL 20k	LANL 41k	Cloud	Gear	Exajet
GPU copy	28.26	164.6	294.3	84.02	272.1	640.6
Level Bounds	10.53	13.46	17.69	12.98	16.22	23.46
Sort	13.13	71.13	124.3	37.44	115.0	441.5
Expand	13.16	66.74	125.6	30.16	90.28	339.6
Find overlap	62.82	338.2	584.8	162.3	495.1	1400.
Merge	1.382	7.491	13.26	3.831	12.40	29.79
Assembly	2.229	9.037	18.09	8.483	25.32	49.14
Σ Compute	103.3	506.1	883.7	255.2	754.3	2283.
Σ Total	132.2	672.6	1181.	340.4	1029.	2931.

Table 2: Alg. 1 timings broken down per phase, in milliseconds.

	LANL 5k	LANL 20k	LANL 41k	Cloud	Gear	Exajet
GPU copy	28.33	164.8	294.5	84.22	272.5	684.6
Level Bounds	10.01	12.71	18.87	1.111	17.09	26.17
Resize	9.237	44.61	81.88	21.63	79.03	151.3
Init Arrays	3.234	3.767	3.169	3.378	3.265	3.527
Populate	1.054	5.087	8.625	1.867	5.898	15.28
Σ Compute	23.53	69.71	112.5	27.99	105.3	189.9
Σ Total	52.10	231.2	407.3	112.2	379.5	884.5

Table 3: Alg. 2 timings broken down per phase, in milliseconds.

Performance by Phases: Algorithm 2

Algorithm 2 can also be broken down into multiple phases, the first of which ("level bounds") is shared with Algorithm 2. The "resize" phase is the most work-intensive one. It performs pruning and shrinking of macrocells using global atomic scatter. The "init arrays" phase creates the cell offset array that will at the end hold the cell IDs, and the mapping array from old to new macrocell memory locations. The final ("populate") phase projects the AMR cells to their final location, by that also filling the offset arrays. We report timings for this in Table 3.

6.3 Sampling

To evaluate the quality and sampling performance of the resulting clusters, we use two different renderers (cf. Fig. 7): a path tracer with incoherent memory accesses, and a ray marchers that (theoretically) benefits from clusters being coherent because of coherent accesses both along single as well as neighboring rays in a CUDA warp.

Data set	ExaBricks	Alg. 1	Alg. 2
LANL 5k	12.61	6.592	8.880
LANL 20k	35.75	32.14	34.60
LANL 46k	27.81	24.25	25.19
Cloud	27.62	24.17	24.15
Gear	7.894	6.370	4.653
Exajet	5.527	5.428	5.496

Table 4: Path tracing with extended bricks and DDA, on 1K images. Frame rates reported in frames/sec.

Data set	ExaBricks	Alg. 1	Alg. 2
LANL 5k	51.23	14.00	21.66
LANL 20k	12.38	9.968	14.33
LANL 46k	21.36	14.90	19.63
Cloud	27.36	27.31	28.29
Gear	18.46	20.92	14.54
Exajet	4.741	10.22	12.03

Table 5: Ray marching with active brick regions (ABR) on 4K images. Frame rates reported in frames/sec.

6.3.1 Path tracing

Our first benchmark uses the path tracing renderer from [35], with the extended bricks sampler and a uniform grid for adaptive sampling based on spatially varying majorants. We modify this renderer to skip over empty cells as generated by Algorithm 2. The renderer uses multi-scattering, ambient lighting, and Russian Roulette with a hard cutoff at 1024 bounces (not reached by our benchmarks). The choice in transfer function will influence the overall rendering performance. The transfer functions we use (cf. Fig. 6) exhibit comparably high absorption. This still gives a meaningful amount of randomness though, which is both due to scattering, as well as due to the Monte Carlo nature of the Woodcock free-flight distance sampling method the path tracer uses. We report frame rates for this experiment, for rendering single samples per pixel (1 SPP) of size 1024×1024 in Table 4.

6.3.2 Ray marching

For this benchmark we use the same visualization algorithm from the original ExaBricks paper [28] computing active brick regions and marching inside those along coherent camera rays. These regions are traversed coherently, and samples are taken directly by identifying the clusters active in the region, i.e., sampling is a sequence of

Data set	# bricks			∅ cells/brick			# single-cell bricks			# cells (largest brick)			∅ aspect ratio			# empty cells		
	Exa	Alg. 1	Alg. 2	Exa	Alg. 1	Alg. 2	Exa	Alg. 1	Alg. 2	Exa	Alg. 1	Alg. 2	Exa	Alg. 1	Alg. 2	Exa	Alg. 1	Alg. 2
LANL 5k	53 K	309 K	62 K	504	87	480	9.4 K	18 K	46	750 K	512	512	0.63	0.65	0.95	n/a	n/a	2.9 M
LANL 20k	1.2 M	3.3 M	444 K	132	47	400	171 K	639 K	560	649 K	512	512	0.58	0.76	0.83	n/a	n/a	19 M
LANL 46k	996 K	3.6 M	670 K	284	79	448	120 K	545 K	668	1.1 M	512	512	0.56	0.75	0.90	n/a	n/a	17 M
Cloud	17 K	157 K	157 K	4.9 K	512	512	175	0	0	619 K	512	512	0.49	1.00	1.00	n/a	n/a	0
Gear	12 K	512 K	512 K	21 K	512	512	50	0	0	980 K	512	512	0.47	1.00	1.00	n/a	n/a	2 K
Exajet	1.1 M	3.6 M	1.4 M	553	177	483	95 K	463 K	549	1.2 M	512	512	0.36	0.79	0.95	n/a	n/a	11 M

Table 6: Statistics for the brick partitionings generated with the original ExaBricks software, and with our two proposed algorithms.

Data set	Alg. 1		Alg. 2	
	peak	final	peak	final
LANL 5k	1.47 GiB	562 MiB	1.28 GiB	576 MiB
LANL 20k	5.49 GiB	1.13 GiB	3.84 GiB	1.13 GiB
LANL 46k	9.23 GiB	1.60 GiB	6.09 GiB	1.59 GiB
Cloud	3.09 GiB	762 MiB	2.38 GiB	784 MiB
Gear	8.49 GiB	1.43 GiB	5.68 GiB	1.44 GiB
Exajet	20.4 GiB	3.00 GiB	13.3 GiB	3.00 GiB

Table 7: GPU memory consumption, measured with `nvidia-smi`. We report peak consumption during algorithm execution, and final memory after all but the bricks and cell IDs were released.

Figure 7: Left: path tracing benchmark, 1024×1024px (4.6–7.9 frames/sec.). Right: ray marching benchmark, on the landing gear data set at 4096×4096px (14.5–20.9 frames/sec.).

lookups, but does not require extra tree traversal other than for locating the active brick region in the first place. The algorithm should benefit from taking as many steps as possible once inside a brick region, to avoid traversing the BVH over and over. We also note though that leaf nodes that are too big could impact cache utilization during sampling, as the cells are stored in positional order in memory, which, depending on the viewpoint could lead to large strides taken through the data. For better comparability we render images of size 4096×4096 and report results in Table 5.

6.4 General Statistics

We finally report statistics about the brick partitionings in Table 6 to identify possible correlations. These include the number of generated bricks, the average number of cells per brick, the number of single-cell bricks, the size of the largest brick, the average aspect ratio (of the shortest to longest side of the bricks), and the number of empty cells (only applies to Algorithm 2). We note that for the cloud data set Algorithm 1 and Algorithm 2 create the same partitioning. We attribute that to the forest-of-octree structure used by FLASH that produces AMR cells on a Morton curve. The generated bricks exactly happen to exactly match the leaf nodes of the original FLASH hierarchy. We also report the memory consumption of our two algorithms, measured with `nvidia-smi`. Since not only the final memory, but also the peak consumption during construction is important—high peak memory consumption could result in the algorithm not executing at all even if the cells and final data fit into memory—we also report this in Table 7.

7 Discussion and Conclusion

We proposed two GPU-accelerated algorithms that outperform the (unoptimized) CPU reference implementation by several orders of magnitude. We now discuss several noteworthy findings. We first observe that Algorithm 2 generally outperforms Algorithm 1 in almost every regard. We primarily attribute this to shallower hierarchies built by Algorithm 2, which is evident from the significantly lower number of single-cell bricks generated. Those single-cell bricks require excessive traversal through the BVH, which although accelerated in hardware can become costly. We also observe that both algorithms achieve about the same interactivity during sampling as when using the original bricks.

With Algorithm 2 in particular, the build performance is so high that the compute operations are faster than the time it takes to copy the input data from the host to the device (using optimized CUDA transfer routines). The mere compute time to build bricks for our largest data set, the 656 M cell exajet, takes on the order of 0.2 sec. (excluding data transfer), which we consider interactive, when, e.g., used in an in-situ scenario. Though building smaller bricks, in general, even the ray marcher shows similar performance although requiring more traversal operations when bricks are smaller. Padding bricks with empty cells seems generally superior than very small bricks that finely tessellate the volume domain.

We observe that the brick partitioning generally influences sampling performance—though importantly the performance is always on the same order. But a clear correlation between size, shape, or spatial arrangement of bricks, and the sampling performance, is hard to identify. The original (Exa-)bricks were not limited in size, while the GPU builders create bricks of at most $8 \times 8 \times 8$. The GPU builders also create bricks with more uniform aspect ratio. We believe that more uniform bricks are more cache-friendly, as a smaller aspect ratio can, depending on the camera pose, lead to unoptimal memory access strides. This could explain why the ray marcher in some cases even benefits in sampling performance when using the newly proposed brick builders.

The two algorithms per design delegate the work of building the hierarchy to OptiX. This results in a very specialized data structure only applicable to AMR rendering with NVIDIA RT cores. We believe that the data structures and builders could however easily be adapted to support other data types such as Open- or NanoVDB, and would be easily extended to also generate hierarchies of high enough quality. This could, e.g., be accomplished by iteratively performing the clustering operation from Algorithm 1 until the data consists of one (root) cluster. An algorithm like that would be structurally similar to the PLOC algorithm by Meister and Bittner [14].

We conclude that clusters of AMR cells can be efficiently built on GPUs. Similar trade-offs exist in terms of quality vs. construction performance as can be observed for triangle acceleration structures, yet the differences are less pronounced, and in some cases the data structures can even outperform SAH-like builders as the one originally used by ExaBricks. Out of the two algorithms proposed we observe that it is favorable to build shallower hierarchies, even if this comes at the expense of introducing empty cells and some amount of overlap where some of the bricks do not carry valid data.

Acknowledgments

This project was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant no. 456842964.

References

[1] J. Baert, A. Lagae, and Ph. Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pp. 27–32. ACM, New York, NY, USA, 2013. doi: 10.1145/2492045.2492048

[2] C. Benthin, R. Drabinski, L. Tessari, and A. Dittebrandt. PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited. *Proc. ACM Comput. Graph. Interact. Tech.*, 5(3), July 2022. doi: 10.1145/3543867

[3] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen. Chombo Software Package for AMR Applications Design Document, 2000.

[4] PowerFLOW User's Guide 3.0, 1998.

[5] ExCALIBUR. AMReX at Exascale meeting and workshop. Available at `https://excalibur.ac.uk/amrex-at-exascale-meeting-and-workshop/`, Accessed: 27 Jun 2025.

[6] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal Supplement Series*, 2000.

[7] M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W. R.Oakes, D. Ranta, and R. Stefan. The RAGE Radiation-Hydrodynamic Code. *Computational Science & Discovery*, 2008.

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

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

[10] T. Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In C. Dachsbacher, J. Munkberg, and J. Pantaleoni, eds., *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association, 2012. doi: /10.2312/EGGH/HPG12/033-037

[11] Khronos Group. Vulkan 1.4.319 - A Specification (with all registered extensions). Available at `https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html#VK_KHR_ray_tracing`, Accessed: 25 Jun 2025.

[12] S. Laine and T. Karras. Efficient sparse voxel octrees – analysis, extensions, and implementation. NVIDIA Technical Report NVR-2010-001, NVIDIA Corporation, Feb. 2010.

[13] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009. doi: 10.1111/j.1467-8659.2009.01377.x

[14] D. Meister and J. Bittner. Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. *IEEE Transactions on Visualization and Computer Graphics*, 24(3):1345–1353, 2018. doi: 10.1109/TVCG.2017.2669983

[15] P. Moran and D. Ellsworth. Visualization of amr data with multilevel dual-mesh interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1862–1871, 2011. doi: 10.1109/TVCG.2011.252

[16] N. Morrical, W. Usher, I. Wald, and V. Pascucci. Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing. In *Proceedings of IEEE Visualization*, VIS '19, pp. 256–260, 2019.

[17] K. Museth. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.*, 32(3), jul 2013. doi: 10.1145/2487228.2487235

[18] K. Museth. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. In *ACM SIGGRAPH 2021 Talks*, SIGGRAPH '21. Association for Computing Machinery, New York, NY, USA, 2021. doi: 10.1145/3450623.3464653

[19] NVIDIA Corporation. NVIDIA OptiX Ray Tracing Engine. Available at `https://developer.nvidia.com/optix`, Accessed: 27 March 2022.

[20] J. Pantaleoni and D. Luebke. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, p. 87–95. Eurographics Association, Goslar, DEU, 2010.

[21] J. M. Patchett, F. J. Samsel, K. C. Tsai, G. R. Gisler, D. H. Rogers, G. D. Abram, and T. L. Turton. Visualization and Analysis of Threats from Asteroid Ocean Impacts. Technical report, Los Alamos National Laboratory, 2016.

[22] D. Seifried, S. Walch, P. Girichidis, T. Naab, R. Wünsch, R. S. Klessen, S. C. O. Glover, T. Peters, and P. Clark. SILCC-Zoom: the dynamic and chemical evolution of molecular clouds. *Monthly Notices of the Royal Astronomical Society*, 472(4):4797–4818, Dec 2017. doi: 10.1093/mnras/stx2343

[23] T. Viitanen, M. Koskela, P. Jääskeläinen, H. Kultala, and J. Takala. Mergetree: A fast hardware hlbvh constructor for animated ray tracing. *ACM Trans. Graph.*, 36(5), Oct. 2017. doi: 10.1145/3132702

[24] I. Wald. On fast Construction of SAH-based Bounding Volume Hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 33–40, 2007. doi: 10.1109/RT.2007.4342588

[25] I. Wald. A simple, general, and GPU friendly method for computing dual mesh and iso-surfaces of adaptive mesh refinement (AMR) data, 2020. doi: 10.48550/ARXIV.2004.08475

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

[27] I. Wald, S. Zellmann, and N. Morrical. Faster RTX-Accelerated Empty Space Skipping using Triangulated Active Region Boundary Geometry. In M. Larsen and F. Sadlo, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2021. doi: 10.2312/pgv.20211042

[28] I. Wald, S. Zellmann, W. Usher, N. Morrical, U. Lang, and V. Pascucci. Ray tracing structured AMR data using ExaBricks. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):625–634, 2021.

[29] F. Wang, I. Wald, Q. Wu, W. Usher, and C. R. Johnson. CPU Isosurface Ray Tracing of Adaptive Mesh Refinement Data. *IEEE Transactions on Visualization and Computer Graphics*, 2019.

[30] S. Zellmann, M. Hellmann, and U. Lang. A Linear Time BVH Construction Algorithm for Sparse Volumes. In *2019 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 222–226, 2019. doi: 10.1109/PacificVis.2019.00033

[31] S. Zellmann, J. P. Schulze, and U. Lang. Binned k-d Tree Construction for Sparse Volume Data on Multi-Core and GPU Systems. *IEEE Transactions on Visualization and Computer Graphics*, 27(3):1904–1915, 2021. doi: 10.1109/TVCG.2019.2938957

[32] S. Zellmann, D. Seifried, N. Morrical, I. Wald, W. Usher, J. A. P. Law-Smith, S. Walch-Gassner, and A. Hinkenjann. Point Containment Queries on Ray-Tracing Cores for AMR Flow Visualization. *Computing in Science & Engineering*, 24(2):40–51, 2022. doi: 10.1109/MCSE.2022.3153677

[33] S. Zellmann, I. Wald, A. Sahistan, M. Hellmann, and W. Usher. Design and Evaluation of a GPU Streaming Framework for Visualizing Time-Varying AMR Data. In R. Bujack, J. Tierny, and F. Sadlo, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2022. doi: 10.2312/pgv.20221066

[34] S. Zellmann, Q. Wu, K.-L. Ma, and I. Wald. Memory-efficient GPU volume path tracing of AMR data using the dual mesh. *Computer Graphics Forum*, 42(3):51–62, 2023. doi: 10.1111/cgf.14811

[35] S. Zellmann, Q. Wu, A. Sahistan, K.-L. Ma, and I. Wald. Beyond ExaBricks: GPU Volume Path Tracing of AMR Data. *Computer Graphics Forum*, 43(3):e15095, 2024. doi: 10.1111/cgf.15095