Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-Of-Core Approach

Jonathan Sarton, Nicolas Courilleau, Yannick Rémion, Laurent Lucas

Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-Of-Core Approach

Jonathan Sarton, Nicolas Courilleau, Yannick Remion and Laurent Lucas

Abstract—In a wide range of scientific fields, 3D datasets production capabilities have widely evolved in recent years, especially with the rapid increase in their size. As a result, many large-scale applications, including visualization or processing, have become challenging to address. A solution to this issue lies in providing out-of-core algorithms specifically designed to handle datasets significantly larger than memory. In this article, we present a new approach that extends the broad interactive addressing principles already established in the field of out-of-core volume rendering on GPUs to allow on-demand processing during the visualization stage. We propose a pipeline designed to manage data as regular 3D grids regardless of the underlying application. It relies on a caching approach with a virtual memory addressing system coupled to an efficient parallel management on GPU to provide efficient access to data in interactive time. It allows any visualization or processing application to leverage the flexibility of its structure by managing multi-modality datasets. Furthermore, we show that our system delivers good performance on a single standard PC with low memory budget on the GPU.

Index Terms—GPU, Caching system, Out-of-core data management, Large data, Interactive visualisation, On-demand processing

1 Introduction

The needs for visualizing and/or processing large volume data are common today in different scientific fields and entertainment. Navigation inside such high-resolution volumes in real-time involves designing efficient out-of-core data management algorithms which address entire massive datasets from high-performance computing devices such as current GPUs.

The interactive addressing problem of any part of a data volume that exceeds the amount of GPU memory has been addressed by several methods. Nonetheless, these approaches have focused on visualization tools and more particularly volume rendering. Recent out-of-core proposed methods integrated into a fully visualization-driven pipeline are not adapted to manage efficiently on-demand processing of non-visible data during the visualisation stage. In many contexts, however, it can be crucial to visualize and interact with data during visualization through processing. This involves providing out-of-core methods adapted to on-the-fly data modification by image processing algorithms applied during interactive visualization.

The proposed solution is a complete out-of-core pipeline from disk to GPU designed to access very large volumes exceeding GPU or CPU memory in interactive time. We base our work on modern methods already known in this field, such as the design of an output-sensitive algorithm with on-demand paging and data streaming, bricking and multi-resolution representation, the use of a brick pool as a cache on GPU texture memory and a virtual address translation mechanism.

This work aims to propose a solution which takes advantage of the many-core environment of the GPUs. This environment allows to carry out as many operations as possible in parallel while preserving enough GPU resources for an end-user application that could require significant computing resources (such as volume ray casting or convolution processing on the volume for instance).

In the following, a state of the art of recent advances in external memory management is given in section 2. Section 3 gives an overview of the proposed pipeline. Our solution is presented and discussed regarding the data representation in section 4 and out-of-core management in section 5. Section 6 describes the impacts of on-demand processing during interactive visualization. In section 7, we introduce a comparison of our approach with the closest previous ones. Our method will be evaluated in section 8 before discussing some specific points in section 9. Our perspective works are exposed in section 10.

Contributions. In this paper, we introduce an efficient out-of-core data management solution suitable for both interactive visualization and on-demand processing on visible or non-visible data of large 3D regular grids from GPU. Our method is based on the principle of virtual memory to address voxels through a GPU page table (PT) hierarchy introduced in [17]. In previous approaches, the cache miss management methods for accessing any part of a large volume were organized in the screen space and therefore directly bounded to the displayed data. This does not allow to handle efficiently data requests for non-visible or not currently visualized data. Moreover, these cache miss reports imply large memory transfers between the device and the host. They can become a very strong constraint in a context of visualisation on very high-definition displays and even worse in a multi-view environment. In this article, we propose an approach that addresses these issues while providing an efficient parallel GPU implementation of the out-of-core addressing data structure. Our method is designed to limit the communications between the CPU and the GPU to their strict minimum while ensuring not to overload the GPU occupancy. In addition, our system is created for a general-purpose context and provides an API available from the GPU for any applications such as standalone or combined visualization and processing of large volume data.

2 Related Works

External memory data management [4, 28, 32] also called out-of-core data management, defines the class of techniques used to handle data that are too large to fit entirely into the main memory of the unit in charge of their processing. There is a vast body of literature on the use of this kind of methods for visualization on different data types, other than volume data [9, 16, 25, 31]. The development of out-of-core methods more specifically for real-time visualization of regular voxel grids has been motivated by volume ray casting [21] of large datasets on GPU and has been widely used during the last decade.

In 2008, Gobbetti *et al.* [15] were the first to offer a complete, out-of-core, multi-resolution volume renderer on GPU. Crassin *et al.* [11] proposed the year after, a more efficient system with Gigavoxel, a ray-guided streaming of opaque voxelized surfaces for entertainment purposes. In [12], Engel presented a framework for scientific visualization of tera-voxels, improving previous works by optimizing the GPU to CPU communications.

- *J. Sarton, N. Courilleau, Y. Remion and L. Lucas are with the Université de Reims Champagne-Ardenne, 51100 Reims, France. E-mail: jonathan.sarton, nicolas.courilleau, yannick.remion, laurent.lucas@univ-reims.fr.*
- *N. Courilleau is also with Neoxia, 75008 Paris, France. E-mail: nicolas.courilleau@neoxia.com.*
- *The first two authors contributed equally to this work.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

In all previously mentioned works, a tree structure is used to address out-of-core bricked data (an octree or a generalized N^3-tree in Gigavoxel) with a kd-restart algorithm to go through the tree on the GPU. The basic principle is the use of a brick pool in GPU texture memory as a cache to store small bricks of voxels with, in the case of [11, 12], a node pool to store the tree nodes. Cache misses are reported for the bricks that are not present in the video memory. These pools are updated at each frame to insert requested data by replacing unused ones if needed; all managed with a simple Least Recently Used (LRU) mechanism. While Gobbetti *et al.* used visibility information for culling, the others introduce a full ray-driven streaming that only loads visible data. Brix *et al.* [8] approach relies on concepts described in [11] and adapts it to the specific needs of multi-channel microscopy on standard computers. More recently, Hoetzlein [18] presented GVDB, a voxel database structure particularly suitable for large volumes with dynamic topology changes to render simulations with a ray tracer on a sparse hierarchy of grids. He also used a particular tree with their own short stack ray tracer. Moreover, several works for storage optimization or efficient construction have been proposed to improve tree structures in this area [4, 20, 22].

We can find many approaches based on a tree structure that are used to address out-of-core data (KD-tree [27, 34], binary space partitioning (BSP) tree [33] or octree [11, 12, 15]) by linking the data through the nodes of the tree. However, Hadwiger *et al.* [17] presented a new virtual memory approach to address several petabytes of biomedical data. They focused on electron microscopy volumes with a continuous stream of data. They compared their approach with tree traversal and observed that it scales better to extremely large volume sizes. Their work was extended to the visualization of segmented electron microscopy volumes [6]. In this extension, the data are segmented beforehand and then stored in an archive before being sent to the GPU on-demand. The segmentation is not performed during the interactive visualization stage on the data cached on the GPU. Jeong *et al.* [19] proposed a volume renderer with on-the-fly processing for noise removal, but only on the currently visible part of the volume. More recently, a visibility-driven method for on-the-fly filtering of 4D ultrasound data have been presented by Solteszova *et al.* [29, 30].

Lastly, for a detailed analysis of out-of-core ray-guided volume rendering, one can refer to Fogal *et al.* [14]. They present a study about the optimal brick subdivision, I/O disk access with or without compression and other characteristics analysis. A complete state of the art can be found in [7]. The above-mentioned works are presented with a complete description of different methods of data representation and storage or comparison of address translation approaches. In addition, some works focused on the use of out-of-core methods on clusters or supercomputers, to distribute the data on several nodes [13].

Although these different solutions address most of the out-of-core data management issues, they do not mention all the needs in a versatile way regardless of the end-user application. For instance, BSP and octree-based approaches may need intermediate structures to be built and thus updated for any change in the data stored in leaves. With animated scenes or scenes where content is unknown in advance, the whole hierarchy needs to be updated. Previous modern visualisation-driven approaches proposed cache miss management that is not adapted to algorithms other than visualization. Hadwiger and Fogal, for example, use a set of hash tables to report missing elements organized in the screen space in a ray-guided context. This does not allow to generate cache misses for bricks that are not part of the working set required for visualization.

Besides the previously mentioned approaches, it is important to mention the NVidia's unified memory technology with CUDA [3]. This technology has been improved since the Pascal architecture and CUDA 8.0 and now allows to allocate buffers larger than the physical memory available on GPUs. By using this system with the address translation support (ATS) and the POSIX *mmap*, it is possible to address an entire volume stored on disk from the GPU. In that case, the page faults are managed automatically by the GPU and the OS. However and conversely to our method, this is strictly restricted to the use of the latest NVidia V100 cards with NVlink2, combined to a UNIX system with the latest kernel (4.16) on a Power 9 architecture [1]. In addition, this technology does not allow to have a control on the page faults and does not allow to use the GPU texture memory (which is mandatory, for instance, in a volume ray casting renderer to take advantage of the interpolation during the sampling).

In the volume ray casting rendering context, Hadwiger *et al.* [17] introduced a data structure different from a tree to avoid maintaining and traversing it. Their choice was motivated by the major constraint of a constant flow of data coming from a microscope inducing the non-prior knowledge of the entire dataset. Moreover, their structure is interesting outside its application framework and scales better than octrees for extremely large volumes. The access time to the data is the same regardless of the resolution level, unlike a tree that requires a depth traversal to access the higher resolution data. Nonetheless, they use it only for visualization with a specific volume ray caster. Our method proposes to extend their approach in a general-purpose framework allowing to visualize and process volumes interactively on the GPU. In addition, Hadwiger proposed to manage the content of the caches present in its structure from the CPU, with per-frame read-back for brick usages and requests. To overcome this shortcoming, we take our inspiration from Cyril Crassin's work realized in another context and described in his PhD thesis [10]. We adapt his method used to maintain an octree to propose a parallel version on the GPU of the management of the virtual addressing structure proposed by Hadwiger. Finally, we present our understanding of the implementation proposed by Hadwiger and give a detailed description of our own.

3 System overview and context

The system we propose is illustrated in Figure 1. It is composed of:

- Multi-resolution 3D mipmap pyramid subdivided into small bricks of voxels for each level and stored in a mass storage, obtained in a preprocessing step;
- Large and simple CPU RAM brick cache and a brick loader interface between the disk / CPU and the GPU;
- Virtual memory architecture: multi-level, multi-resolution PT hierarchy and a brick cache on GPU;
- Cache manager entirely on GPU to maintain the caches and offer an efficient cache miss management.

We propose to implement a multi-level, multi-resolution page table hierarchy to address a full volume using a virtual memory management scheme. This hierarchy is composed of several virtualization levels, with a page table for each level, to address very large volumes. The page table levels (except the root) as well as the bricks of voxels are cached in the GPU and managed individually by LRUs. We propose a full management of the GPU caches from the GPU in order to take advantage of the multi-threaded architecture and to limit communications and synchronization between CPU and GPU.

Caches management. This management can be summarized in two main tasks:

1. the updates of the usage information of cached bricks and the updates of LRU caches;
2. the cache miss reports.

Pipeline details. The main feature of our pipeline is to propose a GPU interface for any application that manipulates very large volumes represented as a regular grid of voxels. The navigation inside the volume is performed in a virtual normalized volume. At any time of the run-time application, a voxel can reside in several memory places (mass storage, system memory, or GPU memory). The voxel addressing is done as one uniform address space regardless of its physical storage location. The access to a voxel is determined by a pair (l, p) with l, the desired level of details (LOD) and p, the 3D normalized floating position in the virtual volume ($p \equiv [x, y, z] \in [0, 1)^3$). When the application requires a voxel (Fig. 1.(1)), the entire pipeline is triggered as follows:

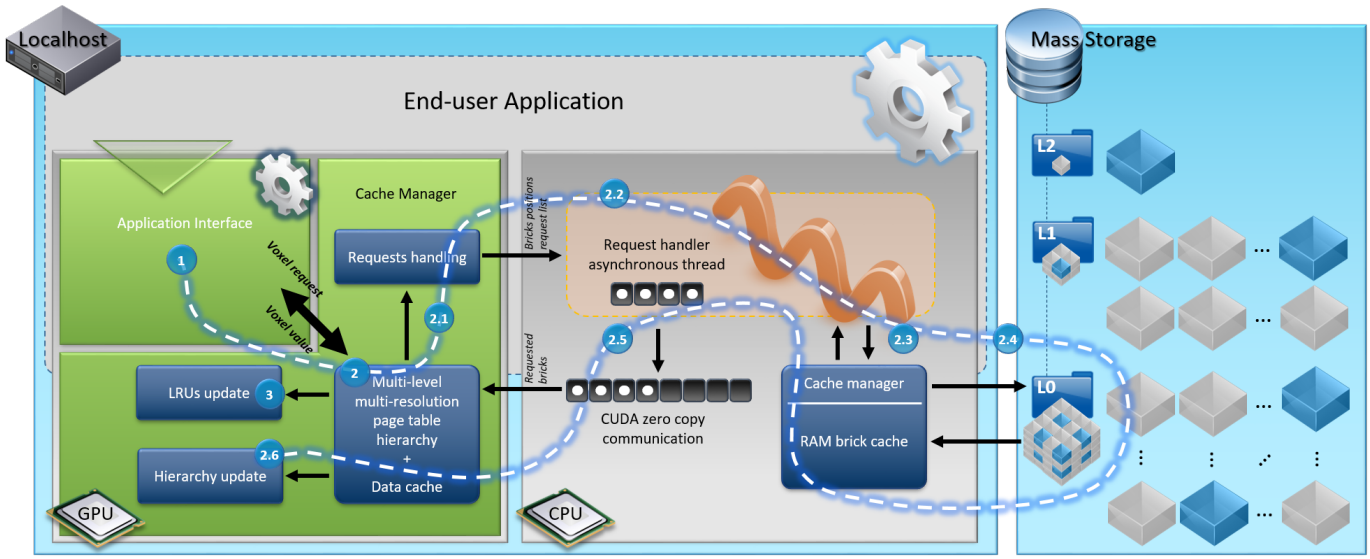


Fig. 1. **Overall out-of-core pipeline**. An end-user application (1) is connected to the GPU interface of the pipeline and can require voxels (2) from the caching system. This system is entirely managed on the GPU to handle brick usage (3) and requests (2.1). When bricks are missing, a small request list is sent to the CPU (2.2) to load them asynchronously from the CPU cache manager (2.3) or from a mass storage device (2.4) where a bricked multi-resolution representation of the volume is stored. The bricks are sent back to the GPU (2.5) and the page table hierarchy is updated (2.6).

i) (Lookup) The first step is to check in the page table hierarchy if the brick containing the requested voxel is present or not in the GPU data cache (Sec. 5.1 & Fig. 1.(2)). *ii) (Hit)* In the case where this brick is present in the brick cache, the cache manager can directly provide the requested voxel to the application. Then it reports the usage of that brick (Sec. 5.2). *iii) (Miss)* In the other case — when a cache miss occurs — a brick request is reported by the cache manager (Sec. 5.2).

Our implementation targets general usage. The optimal cache update strategy depends on the application and will surely be different for ray casting, volume data processing or slice-based volume visualization. On one hand, the cache update will trigger the LRU update from the previous use of bricks (Fig. 1.(3)) and, on the other hand, it will generate a complete request list with all the reported cache misses (Fig. 1.(2.1)). Then a small request list is sent to the CPU (Fig. 1.(2.2)) for an asynchronous processing. An asynchronous thread will request the bricks to the CPU cache manager (Fig. 1.(2.3)). If needed, the bricks are read from the mass storage and written into the CPU cache (Fig. 1.(2.4)). Then, the requested bricks will be written in a buffer accessible by the GPU (Sec. 5.3 & Fig. 1.(2.5)). When the bricks can be written in the GPU data cache, the page table hierarchy is updated accordingly (Sec. 5.6 & Fig. 1.(2.6)). The application efficiently manages the voxel requests, by only asking for the necessary bricks, so as never to load unnecessary ones (for instance, adopting a ray-guided method for volume ray casting).

This pipeline is relatively similar to that proposed by Hadwiger *et al.* [17]. However, its design differs in several aspects because the data accessed is not a constant stream output by a microscope. Therefore, we have an *a priori* knowledge of the dataset, and we store the bricks directly (e.g. $32^3 - 256^3$ voxels) rather than storing 2D tiles that require registrations and a step of 3D construction of bricks at run-time.

Communications. In our system, the communications from CPU to GPU are limited to the strict minimum. They consist of sending the bricks of voxels and a Boolean flag for each of them (for an empty brick information). The communications from GPU to CPU are restricted to a small request list containing (l, p) pairs of each requested bricks. All other mandatory actions of the out-of-core data management are performed inside the GPU, taking advantage of its computing power and limiting costly communications with the host.

4 Data Representation

The input data may come from a physical model acquisition (e.g. biomedical scans, MRI, etc.) as well as from the voxelization of any data as long as they can be represented as a regular 3D grid (Sec. 9).

Multi-resolution. Raw data volumes are preprocessed in order to create a 3D mipmap pyramid which represents the volume at different LODs. Data anisotropy in sampling, which often occurs in medical imaging, can be drastically reduced by applying different downsampling ratios along each axis in this multi-resolution representation.

Bricking. Each level of the multi-resolution pyramid is then divided into small independent blocks of voxels, called "bricks". All bricks have the same size, regardless of their resolution level. This object space decomposition method allows for the manipulation of small amounts of data rather than the whole large volume. The brick shape does not necessarily have to be cubic; different edge sizes can be used. In any case, when the volume is not sufficient to complete all bricks, the last bricks are simply completed with empty voxels. Our approach avoids generating unnecessary empty bricks like most approaches that manipulate only power of two volume sizes. In addition, an overlapping area can be created, between neighbouring bricks, to ensure coherence of future treatments on the voxels (interpolation, convolution, *etc.*).

Our system is designed for the use of volumes whose entire dataset is already defined and known. It means that treatments could be performed on the data before using them in the system. As we are dealing with large volumes of data, both steps mipmapping and bricking take a substantial time to be done; therefore it is interesting to perform these steps outside the stream.

Compression. Finally, the bricks are stored on a large storage device in raw data or compressed format. Using a compressed brick format has two advantages: the volumes will take less memory space on the storage; and it will reduce the loading time of the bricks from the mass storage to the CPU (Sec. 8.4). Nevertheless, we are targeting a real-time application using scientific images; this means the compression algorithm needs to be lossless and has to provide a fast time decompression. To this end, we have opted for an LZ4[1] compression scheme.

[1] http://lz4.github.io/lz4/

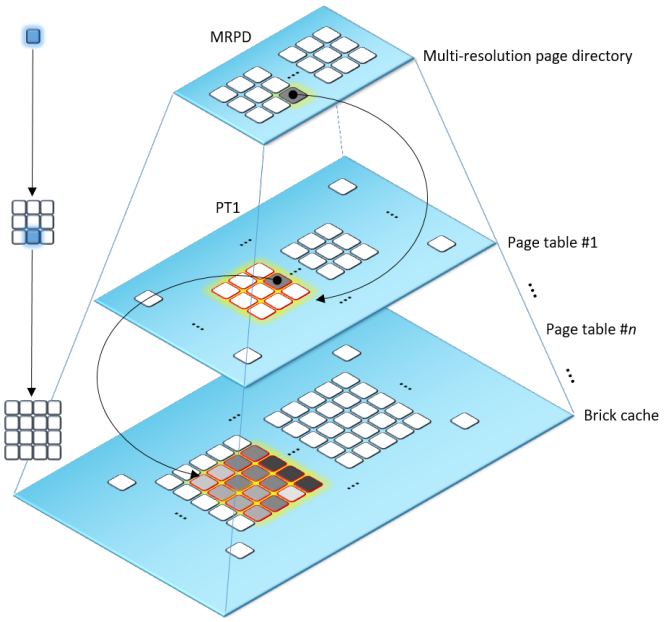


Fig. 2. **Multi-resolution, multi-level page table hierarchy virtualization principle.** A 2D representation of our 3D addressing structure. In this example, only one intermediate virtualization level is used (PT1). One entry of the MRPD addresses a PT block of 3×3 entries in the PT1 cache and one entry in the PT1 cache addresses a brick of 4×4 voxels in the data cache.

5 Out-of-core data management

We detail in this section the out-of-core addressing data structure used to access all the dataset. We present its implementation on GPU and the parallelization strategy of its associated cache manager. We also propose our implementation version and technical details for the virtual hierarchy management and update.

5.1 Memory virtualization

In order to address the bricks of a large multi-resolution volume from the GPU, we adopt an approach of multi-level, multi-resolution page table hierarchy. This structure is based on a large page table organized on several levels, which consists of mapping the virtual address of each voxel with its physical address. This can be seen as a pyramidal structure where each level virtualizes a set of entries (e.g. $16^3 - 64^3$) organized into 3D blocks in the level below (Fig. 2). The top of the hierarchy is composed of the multi-resolution page directory (MRPD) which corresponds to the starting point of the virtual addressing of any voxels, at any resolution level. This is the only level which is not virtualized and entirely present on the GPU with a low memory impact (Sec. 8.4). For very large volumes of data, the MRPD itself is too large to be held in GPU memory. The virtualization principle can therefore be applied to the page table itself. Hence, below the MRPD, there may be N intermediate PT levels with, for each one, a cache containing PT blocks. Adding intermediate levels increases the amount of virtualization and reduces the size of the MRPD. Finally, the data cache containing the bricks is located at the bottom of the hierarchy. The theoretical complexity of the algorithm to access data, given its tree-based data structure, is $\mathcal{O}(\log(b))$, where b is the number of bricks at leaf level. However, the constant of this complexity is very small, and in practice, with two intermediate levels, petabytes of data can be addressed. In addition, the access time does not increase with LOD (unlike an octree).

Our implementation of this hierarchy has been planned generically in order to dynamically create as much intermediate PT level as necessary to address any volume size. This allows the use of smaller bricks and therefore a finer data addressing.

Note, it is important not to confuse the multi-resolution representation volume and its virtualization used for the address indexing on the GPU. While, the former refers to the resolution levels l (LODs), the latter refers to the virtualization levels.

5.2 Implementation

Each hierarchy level is implemented with the CUDA surface API [2] in order to read and write elements into 3D texture memory with hardware accelerated 3D indexing. In the case of a graphic end-user application, the spatial coherent access patterns of the surface memory have a significant benefit. All the caches are managed by an LRU implemented as a simple device vector with the Thrust parallel template library [2].

The data cache implementation is templated in order to store any type of voxels from *GRAY8* to *RGBA32* and others (Sec. 9). Conversely, a page table entry is represented with four 32-bit integers to store the 3D coordinates c of the block virtualized by this entry in the next hierarchy level and a flag representing the state of this block. This flag can be either *empty*, *mapped* or *unmapped* (not present on the GPU).

The MRPD is represented in memory as a buffer containing one 3D subgrid per LOD added next to each other on the x-axis. This representation is illustrated in figure 4. The size of the MRPD is equal to the sum of the x-size of each subgrid and the size of the largest subgrid in the y- and z- axes. This representation implies that a part of the buffer will not be used.

A cache manager is used to maintain the bricks and page tables usage as well as to manage the brick requests. These operations are done in parallel on the GPU with data parallel primitive stream compaction operations.

LRU update. Updating the used bricks is necessary for caches management. This is assured by an LRU algorithm on a list for each cache. All the GPU threads report the use of elements by marking the corresponding elements, in a shared 3D buffer (called *usage buffer*) on global GPU memory, with a global 32-bit integer timestamp. This timestamp is incremented after each update; it is not necessary to use atomic writes to handle concurrent threads access because each thread will write the same timestamp integer in the corresponding entry. There is one such buffer for each cache and it contains as many entries as elements present in its corresponding cache (one to one 3D mapping). Then, a mask is created and filled with a Boolean value that indicates if the corresponding usage buffer entry contains the current timestamp or a previous one. The stream compaction ensures the sorting of the lists by moving the most recently used elements to the beginning while maintaining the order of the others (Fig. 3.(a)).

Cache misses. Figure 3.(b) illustrates the creation of the requested brick list. When a requested brick is not present in the GPU (flag set to *unmapped*), a cache miss occurs and triggers a request for this brick. In the same way as the element usage information, the requested bricks are maintained in a list through a stream compaction operation from a buffer (called *request buffer*) shared by all the CUDA threads. This buffer contains as many entries as bricks in the whole multi-resolution volume, and each thread will tag the corresponding entries to the required bricks with the global integer timestamp. The resulting request list is limited to a small size in order to limit the number of bricks loading at each update. This list contains the requested bricks IDs. These IDs are guaranteed to be unique and are constructed from the spatial position in the volume and the resolution level of the bricks. The proposed cache miss management strategy is entirely generic. It makes it possible to request any brick at any resolution level and at any time without being oriented towards a specific application type.

Comparison with an octree. With an octree approach, the PT entries of the virtual memory management are the octree nodes themselves (stored in a node pool). This implies to transfer these nodes from the CPU to the GPU memory by raising PT entry requests (node requests). In the proposed method, the PT entries are updated directly in the GPU, thus avoiding transfers to the device. The single communications between the central memory and the GPU texture memory are data (bricks) transfers. However, our request buffer has to be sized to the number of total bricks. For instance, let us consider a large volume of 16384^3 voxels RGBA (which is about

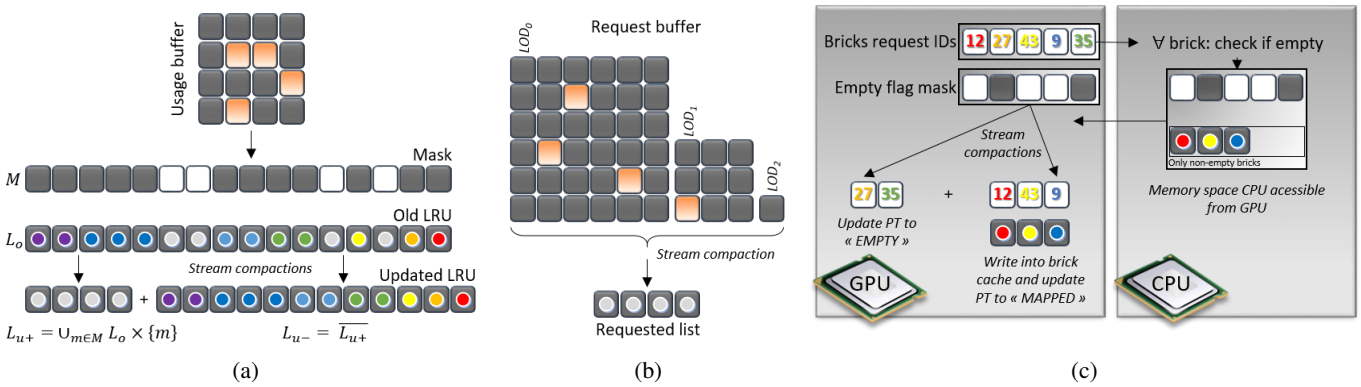


Fig. 3. **Parallel cache manager mechanisms on the GPU** (a) Cache usage update: a single pass to update LRU with double stream compaction (Cartesian products) from a shared buffer to report bricks usage. (b) Bricks request list creation: a single stream compaction from a shared request buffer to create a small request list. (c) Empty bricks management strategy at loading.

17.6 TB) with a ratio of two between two consecutive LODs and a brick size of 64^3 voxels. Since the request buffer is represented in memory in the same way as the MRPD, it will require, with nine LODs $(16384+8192+4096+2048+1024+512+256+128+64) \times \frac{1}{64}$ entries on the x-dimension, $\frac{16384}{64}$ entries on the y-dimension, and $\frac{16384}{64}$ entries the z-dimension. With a timestamp on 4 bytes, the total size of the request buffer will be around 134 MB. With a data cache of 4 GB (which is correct for modern GPUs), it represents less than 3.5% overhead in memory.

5.3 Data fetch

The brick requests are asynchronously handled by the CPU to fetch the data. As shown by Crassin [10], it is more efficient to use CUDA zero-copy [2] in order to load the bricks into GPU memory. It ensures optimal uses of the hardware rather than transferring those manually using copy operations triggered by the CPU. To achieve this, we use a *requested buffer* allocated in CPU memory with CUDA; this buffer is pinned on a physical address region for the PCI Express controller to use directly. A dedicated CPU thread takes care of fetching bricks from the CPU cache or the mass storage and signals the GPU that bricks are available in this buffer. The writing of the requested bricks from the requested buffer to the GPU data cache is performed with a single pass in a CUDA kernel with one thread per voxel per brick.

Empty bricks. Empty brick information makes possible to distinguish the bricks that do not participate in the running algorithm because they are considered as empty, transparent or without any important information. It is important to never load such bricks on the GPU to avoid cluttering the data cache. The access to a brick whose flag is set to *empty* in the virtual addressing structure does not generate a cache miss. However, a brick flagged to *unmapped*, which generates a cache miss, may potentially be empty. Figure 3.(c) shows the adopted strategy to efficiently deal with this behaviour. For each request of brick, we test on the CPU whether it is empty or not (regardless of which criteria). Only non-empty bricks are added in the requested buffer. A Boolean vector indicating for each requested brick whether it is empty or not is transferred to the GPU. With this information, it is possible for the GPU to identify the bricks to transfer and to write inside the data cache. Thus, a dual stream compaction operation is used on the initial brick request list from the empty information mask. This step generates a sorted list to discriminate non-empty from empty bricks. The former will be added to the data cache and referenced as *mapped* in the hierarchy. The latter will be referenced with the *empty* flag.

5.4 Voxel access

This section is a reminder of the method presented by Hadwiger *et al.* [17]. For the sake of reproducibility, we adopt a formal way of presenting the involved concepts. In particular, we present the virtual volume representation in section 5.5 and the update of the virtualization hierarchy in section 5.6.

Using figure 4, a voxel is addressed from a LOD l and a position $p \in [0,1)^3$, with the following two equations:

$$PDentry(l,p) = PageDirBase[l] + \lfloor p \odot v[l]_0 \rfloor \in \mathbb{N}^3 \quad (1)$$

$$Centry_n(l,p) = Cbase_n(l,p) + \lfloor p \odot v[l]_{n+1} \rfloor \bmod b_n \in \mathbb{N}^3 \quad (2)$$

with

$$Cbase_n(l,p) = \begin{cases} \text{MRPD}[PDentry(l,p)].c & \text{if } n=1 \\ \text{PTCache}_{n-1}[Centry_{n-1}(l,p)].c & \text{if } n>1 \end{cases} \quad (3)$$

Equation (1) returns the 3D address to look at in the MRPD. In this formula, $PageDirBase[l]$ corresponds to the 3D position of the beginning of the subgrid for the resolution level l in the MRPD and $v[l]_0$ its number of entries (3D size). The operator $\lfloor\ \rfloor$ is the element-wise vector floor, *i.e.* $\lfloor [x,y,z] \rfloor = [\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor]$. The operator $\odot$ is the element-wise vector multiplication, *i.e.* $[x,y,z] \odot [a,b,c] = [xa,yb,zc]$. If the flag contained in the entry at the coordinates $PDentry(l,p)$ of the MRPD is set to *mapped*, we can continue to equation (2). This second gives the 3D address to look at in all different caches (PT and data caches). The operator modulo is defined on vectors as $[x,y,z] \bmod [a,b,c] = [x \bmod a, y \bmod b, z \bmod c]$. The value n corresponds to the virtualization level from $n=0$ (MRPD) to $n=n_{max}$ (data volume) and passing through intermediate cache levels ($0<n<n_{max}$). Since equation (1) computes the case $n=0$, n goes from $n=1$ (the first cache after the MRPD) to $n=n_{max}-1$ (the data cache) in equations (2) and (3). Hence, as shown in the equation (3), $Cbase_n(l,p)$ is a vector containing 3D cache coordinates c. It is stored in the MRPD for $n=1$ or in the cache just above otherwise. This vector will indicate the 3D position of the beginning of the block in the cache below. $v[l]_n$ and b_n are respectively: the virtualized volume size of LOD l according to the virtualization level n and the size of a PT block, for the PT cache of level n; or the volume size in voxels and the size of a brick, for the data cache (Sec. 5.5).

For instance, the access to the highlighted voxel (voluntarily restricted to the plane $z=0$) of figure 4.(a), using the configuration of figure 4.(b), will be:

$$PDentry(0, \left[\frac{1}{8}, \frac{1}{4}, \frac{0}{1}\right]) = \begin{cases} x = 0 + \lfloor \frac{1}{8} \times 4 \rfloor = 0 \\ y = 0 + \lfloor \frac{1}{4} \times 2 \rfloor = 0 \\ z = 0 + \lfloor \frac{0}{1} \times 1 \rfloor = 0 \end{cases}$$

It is important to note that even if p is composed of real values, the results are given as integer values which is required to

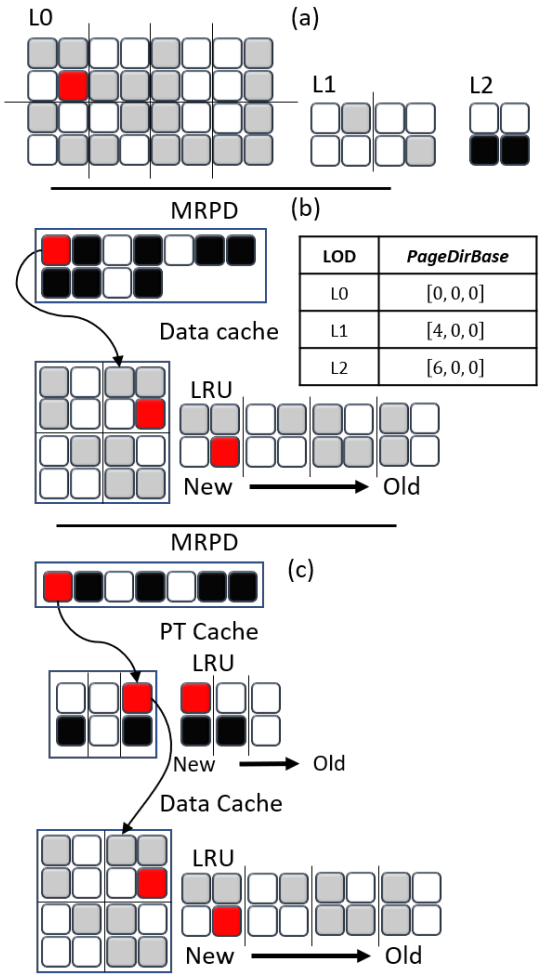


Fig. 4. **2D representation of voxel addressing in the virtual hierarchy.** (a) Volume used as an example, represented with three LODs cut in bricks of size of 2×2 voxels. It contains 8×4 voxels at the highest resolution level (L0), 4×2 voxels at the intermediate LOD (L1) and 2×1 voxels at the coarsest LOD (L2) filled with empty voxels (black) to get one entire brick. (b) Data structure to address the voxels using one virtualization level. One entry of the MRPD addresses one brick. (c) Data structure to address the voxels using two virtualization levels. One entry in the MRPD addresses one PT block of 1×2 entries in the intermediate PT cache. One entry in this PT cache addresses one brick in the data cache. In both cases (b) and (c), we try to get the value of the voxel positioned at $l = 0$ and $p = [\frac{1}{8}, \frac{1}{4}, 0]$. In the MRPD and the PT cache, white means the flag is set to *mapped* and black means it is set to *unmapped*.

match with the positions in the caches. The entry located at $PDentry(0, [\frac{1}{8}, \frac{1}{4}, \frac{0}{1}]) = [0,0,0]$ in the MRPD contains the beginning of the entry (here a brick) in the data cache and a flag set to *mapped*. In our case the entry will be $[x,y,z] = [2,0,0]$. We can now calculate the coordinate of the voxel in the data cache:

$$Centry_1(0, \left[\frac{1}{8}, \frac{1}{4}, \frac{0}{1}\right]) = \begin{cases} x = 2 + \lfloor \frac{1}{8} \times 8 \rfloor \bmod 2 = 3 \\ y = 0 + \lfloor \frac{1}{4} \times 4 \rfloor \bmod 2 = 1 \\ z = 0 + \lfloor \frac{0}{1} \times 1 \rfloor \bmod 1 = 0 \end{cases}$$

By accessing the coordinates $[x,y,z] = [3,1,0]$ in the data cache, we can read the values of the requested voxel. The behaviour will be the same independently of the number of intermediate caches (Fig. 4.(c)) with equation (1) for the MRPD and equation (2) for all other caches.

5.5 Real *vs.* virtual volume representations

The virtualization hierarchy may introduce changes in the size of the volume. These changes are induced by the successive block sizes b_n between the different virtualization levels n. Hence, a volume of a given size may be virtually larger than its real size. To express these differences we note the size of the real volume $r[l]_n$ and the size of the virtual volume $v[l]_n$.

The processing of the real and the virtual sizes ($r[l]_n$ and $v[l]_n$) is performed in two steps using the following equations:

$$r[l]_n = \lceil r[l]_{n+1} \oslash b_n \rceil \tag{4}$$

and

$$v[l]_n = \begin{cases} r[l]_n & \text{if } n = 0 \\ v[l]_{n-1} \odot b_{n-1} & \text{if } n \geq 1 \end{cases} \tag{5}$$

The first step uses equation (4) and computes the real dimensions of the volume for each virtualization level. Starting from the dimension of the volume in voxels $r[l]_{n_{max}}$, this allows us to obtain the dimension of the MRPD $r[l]_0$ according to the block size at each virtualization level n. The operator $\oslash$ is the element-wise vector division, *i.e.* $[x,y,z] \oslash [a,b,c] = [\frac{x}{a}, \frac{y}{b}, \frac{z}{c}]$. The operator $\lceil\ \rceil$ is the element-wise vector ceil, *i.e.* $\lceil [x,y,z] \rceil = [\lceil x \rceil, \lceil y \rceil, \lceil z \rceil]$. The second step, using equation (5), computes the virtual sizes $v[l]_n$ from the MRPD $v[l]_0 = r[l]_0$ to the volume $v[l]_{n_{max}}$ in voxels according to the different block size b_n.

Tables 1.(a, b) indicate the real and the virtual dimensions of the volume shown in section 5.4 and figure 4. The table 1.(b), especially for LOD L1 and L2, illustrates the fact that virtual sizes can be larger than real sizes, due to the ceiling operator in equation (4).

(a)

	$n = 2 = n_{max}$	$n = 1$	$n = 0$	$n = 1$	$n = 2 = n_{max}$
	Vol. voxels	**Vol. bricks**	**MRPD**	**Vol. bricks**	**Vol. voxels**
$L0$	$[8,4,1]$	$[4,2,1]$	$[4,2,1]$	$[4,2,1]$	$[8,4,1]$
$L1$	$[4,2,1]$	$[2,1,1]$	$[2,1,1]$	$[2,1,1]$	$[4,2,1]$
$L2$	$[2,2,1]$	$[1,1,1]$	$[1,1,1]$	$[1,1,1]$	$[2,2,1]$
b_n	–	$b_1 = [2,2,1]$	$b_0 = [1,1,1]$	$b_1 = [2,2,1]$	–
		$r[l]_n$		$v[l]_n$	

(b)

	$n = 3 = n_{max}$	$n = 2$	$n = 1$	$n = 0$	$n = 1$	$n = 2$	$n = 3 = n_{max}$
	Vol. voxels	**Vol. bricks**	**PT 1**	**MRPD**	**PT 1**	**Vol. bricks**	**Vol. voxels**
$L0$	$[8,4,1]$	$[4,2,1]$	$[4,1,1]$	$[4,1,1]$	$[4,1,1]$	$[4,2,1]$	$[8,4,1]$
$L1$	$[4,2,1]$	$[2,1,1]$	$[2,1,1]$	$[2,1,1]$	$[2,1,1]$	$[2,2,1]$	$[4,4,1]$
$L2$	$[2,2,1]$	$[1,1,1]$	$[1,1,1]$	$[1,1,1]$	$[1,1,1]$	$[1,2,1]$	$[2,4,1]$
b_n	–	$b_2 = [2,2,1]$	$b_1 = [1,2,1]$	$b_0 = [1,1,1]$	$b_1 = [1,2,1]$	$b_2 = [2,2,1]$	–
		$r[l]_n$			$v[l]_n$		

Table 1. **Real and virtual sizes of the volume illustrated on figure 4.(a).** Table (a) corresponds to the configuration of figure 4.(b). Table (b) corresponds to the configuration of figure 4.(c). The $r[l]_n$ and the $v[l]_n$ are the virtual representation of the volume at the virtualization level n. For a PT cache level, $v[l]_n$ corresponds to the set of PT entries mapped on the volume representation at the virtualization level n.

Position. The previously mentioned vector $p \in [0,1)^3$ represents the 3D normalized coordinates of a voxel in the virtual volume representation. Thereby, a position $p = [0.5,0.5,0.5]$ will refer to the center of the virtual volume which is not necessary the center of the real volume. To correct this, one needs to change its coordinate system as follow:

$$p_{real} = p_{virtual} \odot (r[l]_{n_{max}} \oslash v[l]_{n_{max}})$$

Block size. The choices of the different b_n are at the user's discretion. We chose small values b_n for this paper in order to simplify the

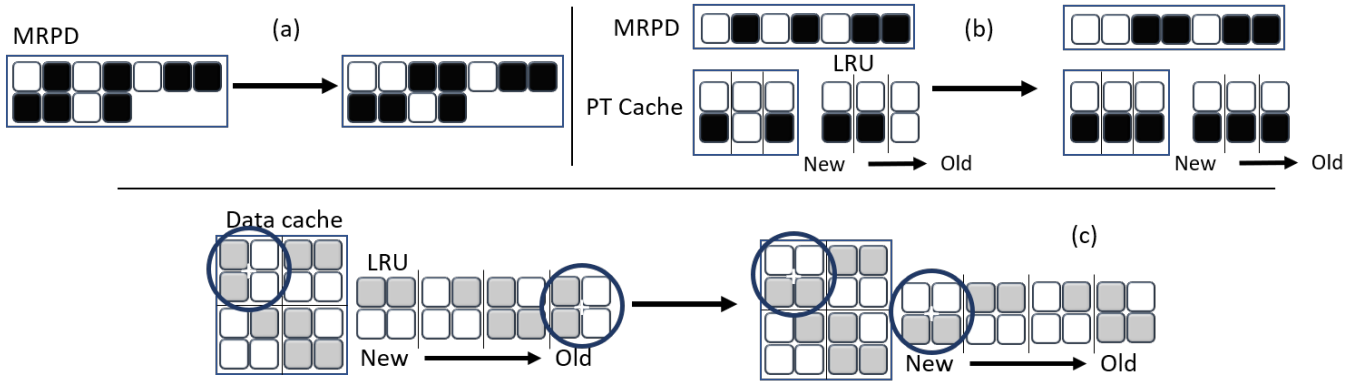


Fig. 5. **2D representation of the hierarchy update.** Update of the PT hierarchy when adding the brick containing the voxel positioned at $l = 0$ and $p = [\frac{2}{8}, \frac{0}{4}, \frac{0}{1}]$ in the configuration of figure 4.(a). The status before the update is shown on the left of the arrows and the status after the update on their right. (a) The update of the MRPD in the case of one virtualization level (according to figure 4.(b)). (b) The update of the MRPD and the intermediate PT cache (and its LRU) in the case of two virtualization levels (according to figure 4.(c)). (c) The update of the data cache (and its LRU) in both cases.

understanding. In real situations the b_n are commonly sized from 16^3 to 64^3. The greater these values, the lower the MRPD, however, larger blocks will require more memory space. A balance must be struck between the b_n, the size of the caches and the size of the MRPD.

5.6 Hierarchy updates

Figure 5 illustrates the update of the PT hierarchy according to the configuration of figure 4. This update consists to load new bricks into the data cache and to reference them. It follows three sequential steps: *i)* removing the references of the bricks that will be removed from the GPU data cache (if the cache is already full), *ii)* writing the new bricks in the GPU data cache and updating its LRU, and *iii)* referencing these new bricks in the PT hierarchy.

For instance, let us suppose we request the brick containing the values of the voxel positioned at $l = 0$ and $p = [\frac{2}{8}, \frac{0}{4}, \frac{0}{1}]$ from figure 4.(a). To add this new brick in the data cache illustrated in figure 5.(c), it is required to dereference the oldest brick given by the LRU (the one positioned at coordinates $[0,0,0]$ in the data cache). Using equation (1) with the coordinates of the oldest brick, it is possible to set the flag of its corresponding entry in the MRPD to *unmapped* (Fig. 5.(a)). The brick is then considered as not present in the cache. Once this first step is done, the new brick is loaded in place of the oldest one for the second step (Fig. 5.(c)). Finally, knowing the coordinates of the new brick and using equation (1), we set in the corresponding MPRD entry the coordinates of this brick in the data cache and its flag to *mapped* (Fig. 5.(a)).

In case of intermediate PT caches level (Fig. 5.(b)), the behaviour is the same. However, if we want to load the brick containing the voxel $l = 0$ and $p = [\frac{2}{8}, \frac{0}{4}, \frac{0}{1}]$, it will be necessary to add a new PT block in the PT cache in order to be able to reference the added brick. To add a new PT block, the oldest one (given by the LRU) is dereferenced from the level just above. Then a new one is generated with all flags set to *unmapped*. Finally the oldest one is replaced by this new one which is then reference in the level just above.

6 On-demand processing during visualization

The proposed approach allows the introduction to on-demand data processing algorithms on visible or not currently visible data during the visualization stage. A processing stage is initiated by an action of the user during the interactive visualization. It can be launched in a different CUDA stream in order to be able to run in parallel (kernel overlapping) and to be independent of the visualization algorithm. Both algorithms (visualization and processing) can request bricks at the same time using the shared *request buffer* without concurrency problem. Regardless of which step it is generated, these requests will be handled in the same way by the cache manager.

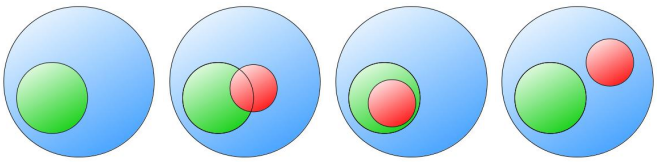
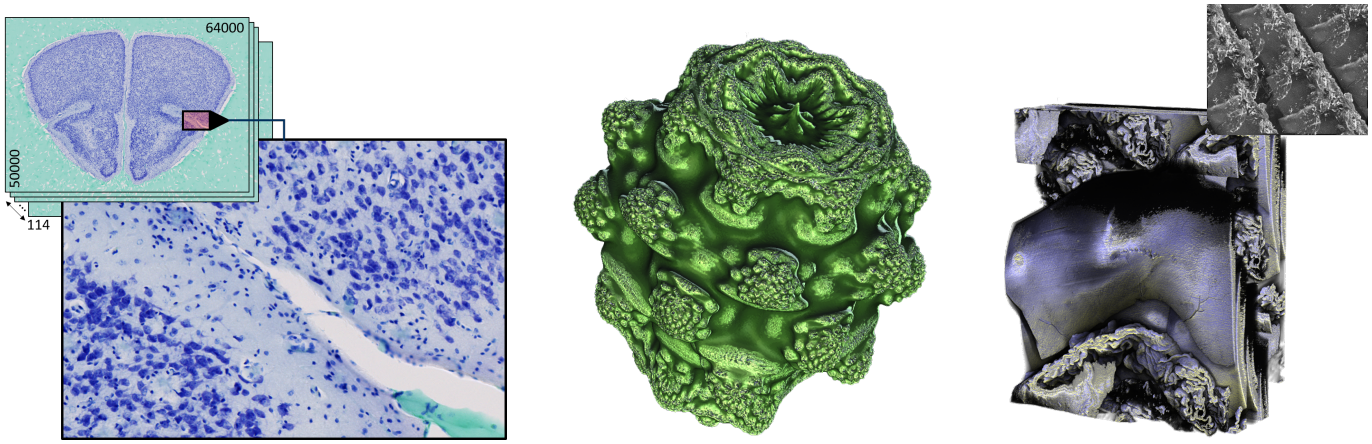


Fig. 6. **Working set configurations.** The blue set represents the whole multi-resolution volume data, the green and the red ones represent the visualization and/or processing working sets. (a) a unique application (visualization or processing), (b, c, d) on-demand processing during interactive visualization.

The processing step can impact the data in different ways. It can provide global information that does not modify the data itself (e.g. number or size of an element). It can indirectly modify a voxel by providing information about it (whether or not a voxel is present in a set, e.g. classification), which will be then interpreted by the visualization step to possibly treat this voxel differently (e.g. applying a certain color). Finally, it can modify the value of a voxel (e.g. filtering). This modification can be permanent: the new value of the voxel is written back in the cache and then propagated with a CPU read-back to be written on the storage; or not permanent: the treatment should be reapplied to this voxel if it is removed and put back later into the cache. If it is necessary to propagate the modification of the value of a voxel to all the LODs, it is up to the application to support this behaviour by requesting the corresponding brick in each LOD present in the multi-resolution volume representation. It is necessary to build as many addresses (l, p) as resolution levels l, keeping the same p (the normalized voxel position in the virtual volume).

Working set. Figure 6 shows the different configurations of the possible working set. An approach that only allows visualization or processing is represented by the first configuration (Fig. 6.(a)). The working set can be defined by one of the other representations in the context of on-demand processing during visualization. A processing could operate only on the data that are visualized during its initialization (Fig. 6.(c)). In that case, the data involved in the processing algorithm are already present in the GPU data cache (or already requested by the visualization stage). Conversely, if the processing algorithm sets up data that are not part of the visualization-specific working set (Fig. 6.(b, d)), this can lead to new brick requests. This may increase the risk of having a working set that is too large to be stored in the cache. In practice, the last configuration would probably be the least frequent. However, it may be found in a distributed environment with a subset of



(a) (b) (c)

Fig. 7. **Datasets used for system validation and performance analysis.** (a) A 1.459 TB histological slices stack render with our high-resolution 2D slices visualizer, (b, c) a 330 GB Mandelbulb fractal and a 11 GB primate hypocampus from a light sheet microscope, render with our own OptiX-based DVR solution [24].

the resources dedicated to the visualization and possibly another subset in charge of processing a part of the volume that is not visualized.

7 Comparison with other systems

By contrast to above-mentioned approaches (Sec. 2), our proposal is the first system that uses a virtual memory architecture based on a multi-level, multi-resolution page table hierarchy with a complete management on the GPU. In this way, in addition to taking advantage of its multi-threaded environment, we designed a system with communications between CPU and GPU that are reduced to their bare minimum. In the work of Hadwiger *et al.* and Fogal *et al.* [14, 17] the use of the same type of structure could be found but with a cache management carried out on the CPU. Hadwiger *et al.* attached a *use*-bit to each brick present in the cache to report brick usages. To report cache misses in their parallel volume ray casting application, they hosted on the GPU, one hash table per image tile to avoid contention and synchronization. For an image tile size of 64×64 with a full HD frame buffer, this sums up to 510 hash tables with expensive atomic operations to report brick requests. Those hash tables are then sent back to the CPU to handle the requests. It is possible to quantify the transfer of these hash tables. For this, it is necessary to note that they used a 32-bit or a 64-bit ID to report each cache misses, depending on the size of the volume. They also limit the number of requests per ray to a small number M in order to distribute cache misses over multiple frames. This represents a message set of about $1920 \times 1080 \times 4 \times 4 \approx 33$ MB (in the worst-case scenario) for an HD viewport with $M = 4$ and a 32-bit ID, at each frame. It reaches $3840 \times 2160 \times 4 \times 4 \approx 133$ MB for a 4K viewport with the same configuration. These values may have to be multiplied by N in a context of visualization with N views (stereo or multiscopic) [26]. For instance, a 4K screen with 11 views it represents $133 \times 11 \approx 1.4$ GB. These quantities are smaller in the case of Fogal *et al.* since they use a compressed format of lock-free hash tables. We propose to limit globally the request number instead of a per-ray limitation. This allows us to work in a general-purpose context and to be more scalable on the cache miss transfers. As already mentioned, we draw our inspiration from Crassin [10] for a comprehensive cache management on the GPU. For the brick usage, we proceed in the same way. However, one notable difference compared to his work is the use of a tree structure with nodes as page tables to address the data bricks. In his system, the octree nodes (equivalent to our page table entries) are requested by the GPU and induce communications between the central memory and the video memory. Moreover, in contrast to previous systems that are particularly focused on volume ray casting, our approach takes place in a general-purpose context.

8 Evaluation

In order to test our GPU cache system, we developed two kinds of visualization applications and a processing stage. The first is a 2D rendering virtual microscope with a navigation through an image stack [26] that provides easy access to high-resolution images. The second is a GPU-based volume ray casting [5]. Furthermore, to test our system with on-demand processing, we offer a convolutional image processing algorithm.

To illustrate the results, we used three datasets (Fig. 7):

- (a) A 114 histological slices stack of a mouse brain with a resolution of 64000×50000 RGBA pixels (1.459 TB);
- (b) A 3D Mandelbulb fractal of 4352^3 RGBA voxels (330 GB) generated with Mandelbulb3D;
- (c) A $2160 \times 2560 \times 1072$ volume with grayscale 16 bits voxels (11 GB) of a primate hypocampus from a light sheet microscope.

All the tests were carried out on an NVIDIA GeForce Titan X with 6 GB of VRAM, a CPU Intel i7 4790K 4 GHz, 32 GB of RAM. We used CUDA 8.0 and OpenGL 4.5 interoperability to render on a full HD viewport 1920×1080.

8.1 Use case 1 – Virtual microscope

The first developed application allows us to navigate (pan and zoom) in a stack of images with 2D multi-resolution rendering. The interest is to simulate the behaviour of a microscope to visualize and navigate into large slices. The principle is to compose a 2D texture from the bricks intersecting all or part of the plane perpendicular to the z-axis in the volume considering the 3D camera position in the volume. We opt for a strategy that promotes the quality of the visual feedback for the user. In this sense, when a request is lifted because a brick of resolution level l is missing, the cache manager provides our renderer with a lower resolution brick (if there is one in the cache) until the brick of level l arrives in the data cache.

8.2 Use case 2 – OptiX™ based volume rendering

In order to maximize the GPU load (cache and end-user application included), a ray casting volume renderer module based on the NVidia OptiX [24] engine was developed. This module is based on the proposed GPU volume ray casting of Kruger and Westermann [21] and uses a modern ray-guided system to ensure a visibility-driven rendering of multi-resolution volumes. The sample integration scheme simulates an emission-absorption model based on the equation presented by Max [23].

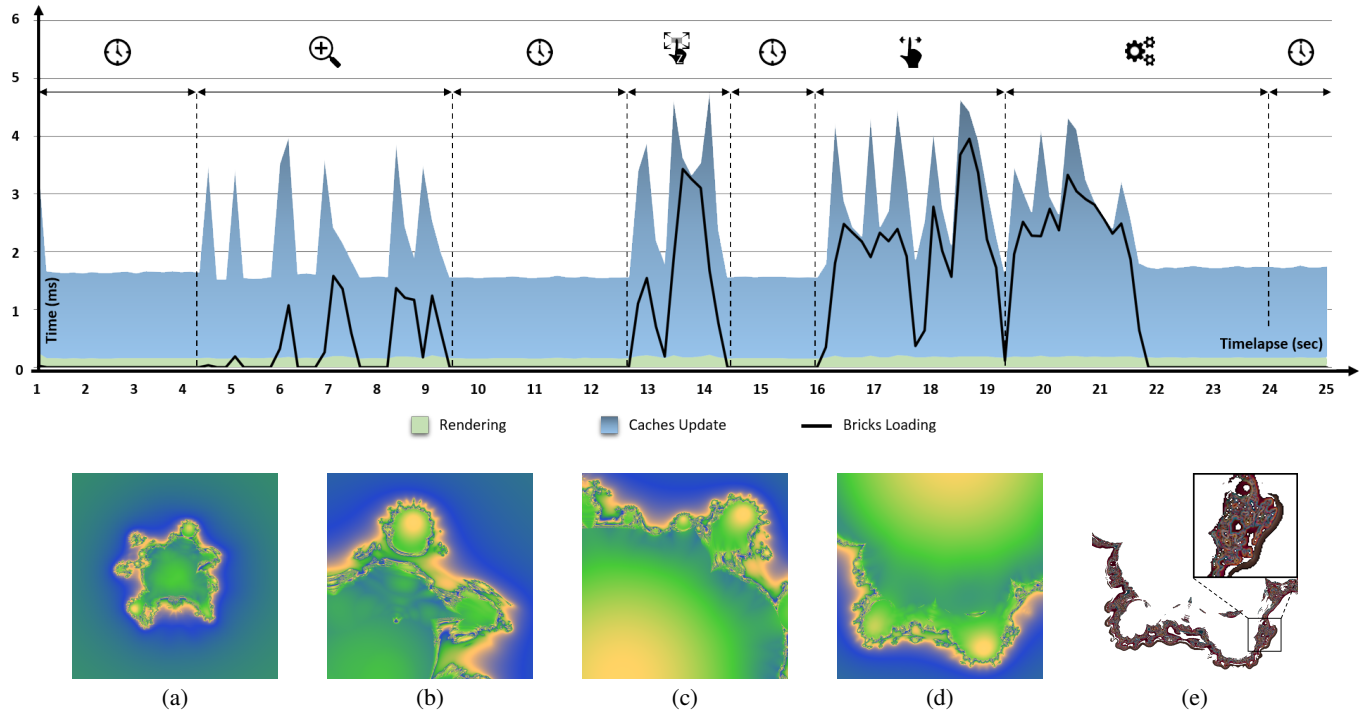


Fig. 8. **GPU usage and bricks loading time.** Comparison of the time spent (in ms) for the rendering, the caches management, and the bricks loading on a 25-second exploration sequence inside the Mandelbulb dataset (b), with the following scenario: a zoom-in, a z-axis navigation, a pan navigation and an edge detection algorithm as image processing. This timeline was obtained with the 2D visualizer described in section 8.1. The five images (a) to (e) are key snapshots of the previously described scenario.

8.3 On-demand image processing

In order to test the system with on-demand processing, we have developed a convolution image processing algorithm. This is an edge detection algorithm that is integrated into the main rendering loop. At any time, the user can initiate it by interacting with the visualized data. It is not relevant here to provide details about this algorithm which can be seen as a black box. This algorithm was used to validate the correct behaviour of an on-demand processing step during the interactive visualization phase, especially to test the sharing of out-of-core data request management of such a process with the rest of the application.

8.4 Performance analysis

Figure 8, obtained by benchmarking the application described in section 8.1, presents the comparison of the time spent on three operations: the rendering and the caches management, both performed on the GPU, and the bricks loading processed asynchronously on the CPU. The caches management operations cover: the LRUs updates and the data request management done at each iteration, the writing of the bricks in the data cache and the hierarchy updates done when new bricks are sent to the GPU. These times are calculated on the datasets (b) by performing the following navigation scenario: zooming from the lowest to the highest resolution level, navigating through the slices of the image stack along the z-axis, pan navigating within a slice (x-,y-axes), and an edge detection convolutional algorithm for image processing. A low amount of time without any interaction is deliberately maintained between each step in order to illustrate the null bricks loading time during these periods.

Throughout the entire scenario, the rendering time is constant and very low (~ 0.2 ms). Moreover, we note that our application allows the rendering to be carried out with a rate of about several hundreds of frames per second. Such high frame-rates can be achieved because the proposed application does not require significant resources at the scale of the data management. In addition, this time is constant because the loading of bricks is performed asynchronously and does not interfere with rendering. The caches update times are less than 2 ms with additional peaks according to the brick loading. These update times are reasonable and allow interactive navigation in the scene. Furthermore, in the case of an application that requires more computational resources, they become negligible.

This scenario allows us to validate an on-demand image processing step within an interactive visualization application. Thus, we can see in figure 8 some of the brick loading peaks that emanate from requests caused by this processing step. Once again, we can note that the rendering time is not affected by this step. As a consequence, these two steps can cohabit by using our out-of-core management system, both at the same time.

Despite the fact that the entire management of the caches and the virtualization hierarchy are realized on the GPU, its occupancy is around 5%. The bottleneck occurs at the data loading, between the mass storage and the CPU. The computation footprint on the GPU is small enough to allow heavy rendering workloads.

Bricks loading. We measured *i)* the average loading time for one brick, from its request by the GPU, to its rendering (or processing), and *ii)* the average time to render a complete correct view (with all the needed bricks addressed in the GPU cache). These two measurements, shown in the table 2, were performed from empty GPU caches. We studied two scenarii: firstly, a worst-case scenario where the bricks are only present on the mass storage. Secondly, when all the bricks are present in our intermediate CPU cache. In the first scenario, the given times indicate: *1)* the bricks requests management (request list creation and sending to the CPU), *2)* the disk access time, *3)* the LZ4 decompression, *4)* the CPU to GPU transfer and *5)* the update of the addressing hierarchy (page table referencing and LRU update). The second scenario only includes the steps 1, 4 and 5. Note that like Hadwiger [17], our worst-case scenario takes few seconds to generate a complete view.

Ray casting performance. The last row of table 2 shows the aver-

Dataset		Histological	Mandelbulb	LightSheet
Vol. resolution		$114 \times 64000 \times 50000$	4352^3	$2160 \times 2560 \times 1072$
Vol. size (GB)		1459	330	11
Brick size (voxels)		64^3	64^3	64^3
Voxel size (B)		4	4	2
MRPD (kB)		25.20	< 1	749.08
LRU(s) (kB)		94.08	94.08	160.16
Usage buffer(s) (kB)		13.44	13.44	22.88
Request buffer (kB)		12518	2550	187.27
Total (MB)		12.65	2.65	1.11
One brick	HDD to GPU cache (ms)	11.76	13.60	5.15
	CPU cache to GPU cache (ms)	2.80	2.90	2.12
One complete view	HDD to GPU cache (s)	6.57	7.43	1.97
	CPU cache to GPU cache (s)	1.64	1.62	0.83
FPS volume ray casting		–	45	47

Table 2. **Volumes description, memory occupancy and ray casting frame rates.** The first part of this table lists the characteristics of the 3 datasets used. The second part details the GPU memory footprint for all the resources needed in the out-of-core data management. The third part gives the average brick loading times for *i)* one brick and *ii)* all the bricks needed to a complete full HD view. The last row gives the frame rates for the volume ray casting rendering. The FPS measurement environment is described in section 8.4 and the virtualization parameters used to measure the amount of memory, depending on the datasets, are described in section 8.5.

age frame rates of the volume rendering using the application described in section 8.2. These measurements used a linear transfer function. We focused these measurements on the dataset (b) and (c) because there is no interest to visualize the dataset (a) in this way. These frame rates can be considered at least as efficient as the previous approaches similar to ours. Therefore, we do not lose in rendering performance using a volume ray caster while proposing a more general method allowing different types of visualization and / or processing.

8.5 Memory occupancy

The second part of the table 2 shows the required memory on the GPU for all the elements of our structure and its management (without the caches themselves). For (a) and (b), we used a PT cache storing 500 PT blocks in the second virtualization level and a data cache storing 2860 bricks (~ 3 GB). For (c), we use a data cache storing 5720 bricks (~ 3 GB). The memory occupancy is about 12.65 MB for the first dataset (a) with two virtualization levels, using a brick size of 64^3 and a PT block size of 32^3 voxels; 2.65 MB for the second (b) with two virtualization levels, using a brick and a PT block size of 64^3 voxels; 1.11 MB for the last (c) with only one virtualization level and a brick size of 64^3 voxels. The low costs of these values maintain a limited pressure rate between the memory allocated for the caches management and those necessary for a well-functioning state of the end-user application.

9 Discussion

The proposed approach offers benefits for a wide range of applications, but suffers a few limitations detailed here.

9.1 Limitations

Race conditions. This system implies some limitations regarding race conditions. Indeed, the thread in charge of loading and preparing the bricks cannot load them in the cache and update the virtual hierarchy without inducing possible collision with the GPU rendering (or processing) thread.

Page table hierarchy updates. There are two main limitations related to the update of the PT hierarchy. The first occurs when an entry is removed from a PT cache. In figure 5.(b) when the new PT block overwrites the oldest one, we lose the link to all the bricks that the last one was referencing. In essence, some data may still be present in the caches but not referenced (garbage that is not taken out). However, these orphan blocks/bricks will converge to the oldest part of the LRUs and will be replaced in future iterations. The second limitation is related to the way to reference, in the PT, the bricks recently added to the data cache. Although we propose an implementation of this step on the GPU, it is done sequentially for each brick. For instance, let us assume we want to add, at the same time, two spatially close bricks, both being referenced by a single PT block. A parallel system would add two PT blocks in the cache that would both reference a single brick while only one block may be required to reference these two new bricks. To avoid such a scenario, each brick needs to be referenced one by one.

Loss of virtual pages. The proposed approach of virtual memory architecture does not guarantee the meta-information perennity of pages inside the page table. For the possible intermediate virtualization levels (PT caches), there is a problem of information preservation. The pages are created on-the-fly on the GPU as needed and then cached. When a page is removed from the cache, all its contained information is lost. A typical example is the empty brick state indicated by the *empty* flag which will not be preserved. In contrast, this problem does not occur with the use of tree structures (like octrees) since the nodes of the tree are actually used as pages. These nodes are written and saved and then transferred to the GPU if necessary.

Processing. Our system can support any data processing if it is used alone. However, if it is initiated during an interactive visualization stage, it needs to be restricted. Indeed, an algorithm manipulating the whole volume will generate many cache flushes, which may severely impact the possibility to maintain the interactive visualization working set stored in the data cache. Thus, we must limit the processing to a lower working set size corresponding to a set of smaller local treatments.

9.2 Strengths

With this data structure, the system gets access to the data in $\mathcal{O}(\log(b))$ where b is the number of bricks. However, the complexity constant is such that in practice it dwarfs the access time to a very small value (two or three levels in practice), as detailed in section 5.1.

Communications. The communications between the CPU and the GPU are restricted in order to reduce to data flow. The information transferred is simply the requested brick coordinates for the GPU to CPU communications and the voxels themselves, organized in bricks, for the CPU to GPU communications. Restricting the communications with these two kinds of data prevents bottlenecks.

Data Type. With the proposed approach, it is possible to virtualize any kind of data as long as these data can be represented as a regular 3D grid. Our approach uses the CUDA texture memory that can store up to 4 channels with a maximum of 32 bits each. However the approach also applies to general data configuration with the use of global memory

instead of CUDA surfaces to extend to a more general purpose context; the same algorithms and the logic still apply.

Scalability. This system may be easily scaled in a high performance computing environment: cache managers deployed on each node are independent, each one handling a subworking set. The main difference would lie in the cache misses, where the nodes could communicate together in order to speed up the brick loading time.

10 Conclusion & Perspectives

We propose an out-of-core caching system fully managed on GPU to address very large volumes of data. Our pipeline can be used by any type of application that requires manipulating volumes represented as a regular grid of voxels, with the ability to manipulate multi-modal data. In particular, this allows us to introduce the possibility of on-demand processing of visible or not currently visible data during an interactive visualization stage. We demonstrate the validity of our implementation using a 2D high-resolution slice visualization tool with interactive navigation in volumes exceeding the amount of GPU and CPU memory with a very high rendering frame-rate and a low memory footprint on the GPU. We also show that our method performs at least as effectively as the previous ones in a context of volume ray casting application.

Hierarchy updates. In the current approach, the PT hierarchy updates are performed on the GPU but in a sequential context (one thread kernel). It prevents duplications in the load of bricks or race conditions in the PT entry updates but it does not take advantages of the parallelism of the GPUs. It could be interesting to consider a parallel strategy to improve performance of this update step.

Transfer times. The main bottleneck in the system is the brick loading time from the storage device to the GPU. Data compression addresses part of this problem, but it could be more interesting to be able to decompress the bricks directly on the GPU. This could take advantage of the multi-threaded environment to implement an efficient decompression algorithm and reduce the amount of transferred data to the GPU. As many bricks are loaded, there is a natural parallelism that can be exploited. Another solution would be to bring this system to a high performance computing environment. By properly sharing the work, the different nodes of such an environment could then distribute the transfers and thus reduce this bottleneck.

Cache misses. We could consider a strategy to improve the cache miss management system by prioritizing requests. For instance, with a counter we could determine which bricks have been the most requested. Such bricks would have a higher priority for the loading step. This could potentially improve visual quality of the rendering for the same overall performance level of the system.

Acknowledgments

This work is supported by the French national funds (PIA2'program "Intensive Computing and Numerical Simulation" call) under contract No. P112331-3422142 (***3D**Neuro**Secure*** project). The purpose of this project is to propose a collaborative solution to process and interactively visualize massive multi-scale data from ultra-high resolution 3D imaging. This secure solution also aims to break therapeutic innovation by allowing the exploitation of 3D images and complex data of large dimensions as part of applications framework linked to neurodegenerative diseases like Alzheimer's. We would like to thank all the partners of the consortium led by Neoxia, the three French clusters (Cap Digital, Systematic and Medicen), Florent Duguet for his participation and advice on the problems we encountered, Thierry Delzescaux and the Mircen team (CEA, France) for the datasets (a) & (c) Fig. 7 as well as NVidia for all their advice.

References

[1] Everything you need to know about unified memory. `http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf`. [Online; accessed 2018-July-12].

[2] Nvidia CUDA programming guide 8.0. `https://docs.nvidia.com/cuda/index.html`. [Online; accessed 2017-December-06].

[3] A. A. Awan, C.-H. Chu, H. Subramoni, X. Lu, and D. K. Panda. Can Unified-Memory support on Pascal and Volta GPUs enable Out-of-Core DNN Training? In *Proceedings of International Supercomputing Conference*, ISC '18, 2018.

[4] J. Baert, A. Lagae, and P. Dutr. Out-of-Core Construction of Sparse Voxel Octrees. *Computer Graphics Forum*, 33(6):220–227, 2014.

[5] B. Battin, G. Valette, J. Lehuraux, Y. Remion, and L. Lucas. A Premixed autostereoscopic OptiX-based Volume Rendering. In *2015 International Conference on 3D Imaging (IC3D)*, pp. 1–5, 2015.

[6] J. Beyer, M. Hadwiger, A. Al-Awami, W. K. Jeong, N. Kasthuri, J. W. Lichtman, and H. Pfister. Exploring the Connectome: Petascale Volume Visualization of Microscopy Data Streams. *IEEE Computer Graphics and Applications*, 33(4):50–61, 2013.

[7] J. Beyer, M. Hadwiger, and H. Pfister. State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum*, 34(8):13–37, 2015.

[8] T. Brix, J.-S. Praßni, and K. H. Hinrichs. Visualization of large volumetric multi-channel microscopy data streams on standard PCs. *CoRR*, abs/1407.2074, 2014.

[9] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens. Out-of-core Data Management for Path Tracing on Hybrid Resources. In *Computer Graphics Forum*, vol. 28, pp. 385–396, 2009.

[10] C. Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. phdthesis, Universit de Grenoble, 2011.

[11] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Symposium on Interactive 3D Graphics and Games*, pp. 15–22. ACM, 2009.

[12] K. Engel. CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs. In *2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 123–124, 2011.

[13] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher. Large Data Visualization on Distributed Memory multi-GPU Clusters. In *Proceedings of the Conference on High Performance Graphics*, HPG'10, pp. 57–66. Eurographics Association, 2010.

[14] T. Fogal, A. Schiewe, and J. Kruger. An analysis of scalable GPU-based ray-guided volume rendering. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pp. 43–51, 2013.

[15] E. Gobbetti, F. Marton, and J. A. I. Guitin. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008.

[16] P. Goswami, M. Makhinya, J. Bsch, and R. Pajarola. Scalable Parallel Out-of-core Terrain Rendering. In *EGPGV*, pp. 63–71, 2010.

[17] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, 2012.

[18] R. K. Hoetzlein. GVDB: Raytracing Sparse Voxel Database Structures on the GPU. In *Proceedings of High Performance Graphics*, HPG '16, pp. 109–117. Eurographics Association, 2016. doi: 10.2312/hpg.20161197

[19] W. K. Jeong, J. Beyer, M. Hadwiger, A. Vazquez, H. Pfister, and R. T. Whitaker. Scalable and Interactive Segmentation and Visualization of Neural Processes in EM Datasets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1505–1514, Nov. 2009. doi: 10.1109/TVCG.2009.178

[20] V. Kämpe, E. Sintorn, and U. Assarsson. High Resolution Sparse Voxel DAGs. *ACM Trans. Graph.*, 32(4):101:1–101:13, 2013.

[21] J. Kruger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pp. 38–. IEEE Computer Society, Washington, DC, USA, 2003.

[22] S. Laine and T. Karras. Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1048–1059, 2011.

[23] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995. doi: 10.1109/2945.468400

[24] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pp. 66:1–66:13. ACM, New York, NY, USA, 2010.

[25] R. Richter and J. Dllner. Out-of-core real-time visualization of massive 3d point clouds. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in*

Africa, pp. 121–128. ACM, 2010.

[26] J. Sarton, N. Courilleau, A.-S. Hérard, T. Delzescaux, Y. Remion, and L. Lucas. Virtual Review of Large Scale Image Stack on 3D Display. In *2017 International Conference on Image Processing (ICIP)*, 2017.

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

[28] C. Silva, Y.-j. Chiang, W. Corrła, J. El-sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *In Visualization02 Course Notes*, 2002.

[29] V. Solteszova, A. Birkeland, S. Stoppel, I. Viola, and S. Bruckner. Output-Sensitive Filtering of Streaming Volume Data: Output-Sensitive Filtering. *Computer Graphics Forum*, 36(1):249–262, Jan. 2017. doi: 10.1111/cgf.12799

[30] V. Solteszova, A. Birkeland, I. Viola, and S. Bruckner. Visibility-Driven Processing of Streaming Volume Data. p. 11, 2014.

[31] J. E. Stone, K. L. Vandivort, and K. Schulten. Immersive Out-of-Core Visualization of Large-Size and Long-Timescale Molecular Dynamics Trajectories. In *Advances in Visual Computing*, pp. 1–12. Springer, Berlin, Heidelberg, 2011.

[32] J. S. Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, 2008.

[33] J. Xue, J. Yao, K. Lu, L. Shao, and M. M. Rahman. Efficient volume rendering methods for out-of-Core datasets by semi-adaptive partitioning. *Information Sciences*, 370:463–475, 2016.

[34] S. Zellmann, J. P. Schulze, and U. Lang. Rapid k-d Tree Construction for Sparse Volume Data. *Eurographics Symposium on Parallel Graphics and Visualization*, p. 10, 2018.