# Une approche out-of-core entièrement basé GPU pour la manipulation de gros volumes de données

A fully GPU-based out-of-core approach to handle large volume data

Jonathan Sarton, Nicolas Courilleau, Florent Duguet, Yannick Remion & Laurent Lucas

**English Abstract**—3D datasets production capabilities in bioimaging has widely evolved in recent years notably with a rapid increase in the raw size of these ones. As a result, many large-scale applications including visualization problems has become challenging to address, particularly when the available main or GPU memory - even more limited - hardly ever exceeds the data size for a processing requiring the entire volume to be present in memory for instance. The solution to this issue lies in providing out-of-core algorithms specifically designed to handle datasets larger than memory. We propose in this article a new approach based on joint works of Hadwiger *et al.* [10] and Crassin [5]. This pipeline was designed to manage data as regular grids regardless of the underlying application. It relies on a caching approach with a virtual memory addressing system fully managed by the GPU. It allows any visualization or processing application to leverage of the flexibility of its structure by managing multi-modalities datasets in any way. We also discuss about implementation details in particular the hierarchical cache levels approach. Furthermore, we present some results on a single node PC cluster before to provide a detailed performance analysis of our solution. All examples are rendered at interactive rate while respecting the limited GPU memory budget.

**Index Terms**—GPU, Caching system, Out-of-core, Large data

## 1 Introduction

The need of visualizing and/or processing large volume data has become common today in different fields such as biomedical or even entertainment. Navigating inside such volumes in real-time involves designing efficient out-of-core data management algorithms to address entire massive datasets from high-performance computing devices such as current GPUs.

In this paper, we introduce a complete out-of-core pipeline from disk to GPU to access very large volumes exceeding GPU or CPU memory in real-time. We base our work on modern methods already known in this field, such as the design of output-sensitive algorithm, the on demand-paging and data streaming, the bricking and multi-resolution representation, the use of a brick pool as a cache on the GPU, the virtual addresses translation and more. However, we propose a system that allows access to multi-modal volumes from any type of end-user application with a GPU interface connected to our out-of-core pipeline.

This work aims to propose a solution which takes the maximum advantage of the multi-threaded environment of the GPUs in order to carry out as much as possible operations in parallel, while leaving enough free GPU occupancy time for an end-user application that could be very consuming in computing resource (such as volume ray-casting or convolutional processing on the volume for instance).

This paper is structured as follow. In section 2, a brief state of the art of recent advances in external memory management is given. In the next sections (from 3 to 5), our contribution is presented and discussed in terms of data representation first, out-of-core management next and ultimately implementation. Some preliminary results are then presented in section 6. Finally, our future plans are exposed in section 7 before we conclude.

**Context of research**
This work is part of the ***3DNeuroSecure*** project ("Intensive Computing and Numerical Simulation" call of PIA2 for the Development of the Digital Economy). The aim of this project is to propose a collaborative solution to process and interactively visualize massive multi-scale data from ultra-high resolution 3D imaging (e.g. light sheet microscopes and histological slide scanners). This secure solution also aims at breaking therapeutic innovation by allowing the exploitation of 3D images and complex data of large dimensions as

- *J. Sarton: Université de Reims Champagne-Ardenne, CReSTIC, France*
  *E-mail: jonathan.sarton@univ-reims.fr.*
- *N. Courilleau: URCA, CReSTIC, France & Neoxia, France*
  *E-mail: nicolas.courilleau@neoxia.fr.*
- *F. Duguet: URCA, CReSTIC, France & Altimesh, France*
  *E-mail: florent.duguet@altimesh.com.*
- *Y. Remion: URCA, CReSTIC, France*
  *E-mail: yannick.remion@univ-reims.fr.*
- *L. Lucas: URCA, CReSTIC, France*
  *E-mail: laurent.lucas@univ-reims.fr.*

part of applications framework linked to the neurodegenerative diseases like Alzheimer.

## 2 Related Works

External memory data management [12], [13] also called out-of-core data management, defines the set of techniques used to handle data that are too large to fit entirely into the main memory of the unit in charge of processing these data. The development of these methods for real-time visualization of regular voxel grids has been motivated by volume ray-casting on GPU [11] of large datasets and has been widely used during the last decade.

Gobbetti *et al.* [9] in 2008, are the first to offer a complete, out-of-core, multi-resolution volume renderer. Then, in 2009, Crassin *et al.* [6] propose a more efficient system with Gigavoxel, a ray-guided streaming of opaque voxelized surfaces for entertainment purpose. In [7], Engel presents a framework for scientific visualization of tera-voxels, improving previous works by optimizing the GPU to CPU communications.

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 [6], [7], a node pool to store the tree nodes. 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.* [4] rely on concepts described in [6] and adapt it to their specific needs of multi-channel microscopy on standard computers.

Whereas we can find many approaches based on the use of an octree, Hadwiger *et al.* [10] present a new virtual memory approach to address several petabytes of biomedical data. They focused on Electron Microscopy volumes with continuous stream of data. They compare their approach with octree traversal and note that it scales better to extremely large volume sizes.

For a detailed analysis, one can refer to Fogal *et al.* [8]. They present a study about the bricking and the optimal brick size, the I/O disk access with or without compression and other characteristics analysis. In addition a more complete state of the art can be found in [3], we can find the above mentioned works with a complete description on different methods of data representation and storage or comparison of address translation approaches.

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.

**Contribution**

Our contribution based on Hadwiger *et al.* [10] and Cyril Crassin's Phd Thesis [5] proposes an hybridization of these two works in order to introduce a generic caching system embedded and managed on the GPU. To achieve this, we first borrow the virtual memory approach of [10] to translate virtual to physical brick addresses with a multi-level, multi-resolution page table (PT) hierarchy. Such a choice is motivated by both, the construction and the management constraints of an octree and the better scalability for very large volumes. In contrast to Hadwiger *et al.* who manage the content of GPU caches from the CPU, with per-frame read-back for bricks usage and requests, we next use the cache mechanism entirely managed on the GPU proposed by [5].

## 3 System overview

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

- multi-resolution 3D mipmap pyramid subdivided into small bricks for each level and stored in a mass storage, obtained in a pre-precessing step;
- large CPU RAM brick cache;
- multi-resolution multi-level PT hierarchy and a data cache on GPU;
- cache manager entirely on GPU to manage the maintenance of caches and offer an efficient management of cache miss.

We propose to implement a multi-level multi-resolution page table hierarchy, introduced by Hadwiger *et al.* [10], to address a whole volume using virtual memory scheme. This hierarchy is composed of several levels of virtualization, with a PT for each level, in order to address very large volumes. The levels of page table as well as the bricks of voxels are cached in the GPU and managed individually by LRUs. While Hadwiger *et al.* manage the entire content of the GPU caches from the CPU, we propose a full management on the GPU in order to take advantage of the multi-threaded architecture and to limit CPU reads back.

This management can be summarized in two main tasks:

1) The updates of the cached bricks usage information and the LRU caches updates.
2) The raise of cache misses.

On their side, Hadwiger *et al.* use a *use* bit for each brick present in the cache to report bricks usages. In addition, to report cache misses, they use one hash table on the GPU for each $N \times N$ pixels subset of the display. For $N = 64$ with a full HD display for instance, they use 510 hash tables with atomic

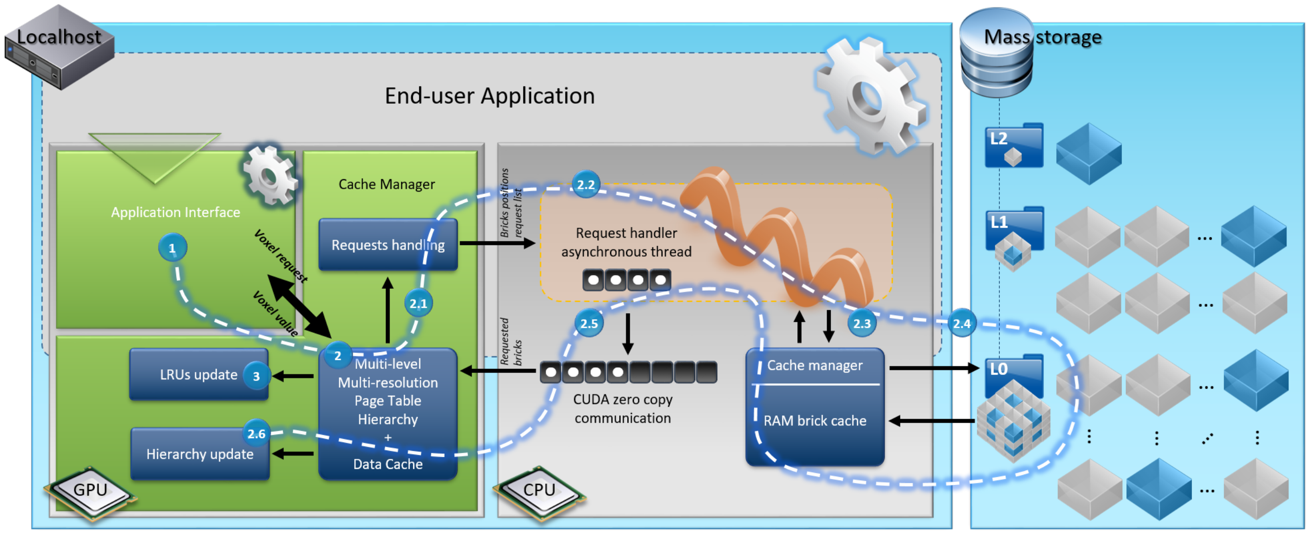

Fig. 1. **Complete out-of-core pipeline**. An end-user application is connected to the GPU interface of our pipeline. This application can require voxels on the GPU. The navigation through the volume is virtualized and the voxels are stored in small bricks and cached on the GPU. The cache is entirely managed on the GPU to handle bricks usage and requests. A small request list for non present bricks is sent to the CPU processed by an asynchronous thread that will require for these bricks to a CPU cache manager. If a brick is not present in the CPU cache, we read it from a mass storage where a bricked multi-resolution representation of the volume is stored. The brick is sent back to the GPU and the page table hierarchy is updated.

adds to report brick requests. Those hash tables are then sent to the CPU to handle the requests. As we have already mentioned, we draw our inspiration from [5] for a full management of the caches on the GPU. For (1) we proceed in the exact same way, so we will briefly explain the procedure as a reminder in the section 5.2). However, one notable difference compared to Crassin is that he uses a tree structure with nodes as page tables to address the data bricks. In his system, the page table entries (nodes of the octree) are requested by the GPU and involve communications between the central memory and the video memory. Thus, for (2) we propose a system slightly different.

The main feature of our pipeline is to propose a GPU interface for any type of 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 applicafion a voxel can reside in several memory places (mass storage, CPU or GPU). The voxel addressing is done in the application as one uniform address space regardless of the physical place of the voxel. 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 ($p \in [0, 1]^3$). When the application requires a voxel (Fig.1-1), the entire pipeline is triggered as follow: *i)* The first step is to check in the page table hierarchy if the brick containing the required voxel is present or not in the GPU data cache (see Sec.5.1 & Fig.1-2). *ii)* In the case where this brick is present in the data cache, the cache manager can directly provide the required voxel to the application. Then it will report the usage of that brick (see Sec.5.2). *iii)* In the other case – when a cache miss occurs – a brick request is reported by the cache manager (see Sec.5.2).

The end-user application needs to launch explicitely the update of our cache manager. For instance, with a ray-caster it is generally performed after each frame creation. This action will lead, on one hand, to the update of the LRUs from the previous use of bricks (Fig.1-3) and, on the other hand, will create 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) where an asynchronous thread will request the bricks to the CPU cache manager (Fig.1-2.3). If needed the brick is read from the mass storage and written into the CPU cache (Fig.1-2.4). Then, the asynchronous thread will write the requested bricks in a buffer accessible by the GPU (Fig.1-2.5). When the brick can be written in the data cache, we will update the page table hierarchy accordingly (see Sec.5.4 & Fig.1-2.6).

Compared to [10], our pipeline is simpler because we do not have the constraint of a constant stream of data coming from a microscope. Therefore, we have an *a priori* knowledge of the data set and we store directly small bricks (e.g. $16^3 - 128^3$ voxels) rather than storing 2D tiles that requires registrations and a step of 3D construction of bricks at run-time.

In our system, the only communications from host to device are operated to transfer the bricks containing the voxels. The communications in the other direction (device to host) are limited to the small request lists containing the pair $(l, p)$ 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 could come from physical model acquisition (e.g. biomedical scans, MRI, etc.) or from the voxelization of any kind of data as long as they may be represented as a scalar 3D grid (see Sec. 6.3). In order to use these volumes in our system they need to be preprocessed. In a first step, we create a pyramid of resolution from the given volumes, it basically consists of creating a 3D mipmap pyramid of the volumes. In addition, applying different down-sampling ratios along each axis gives the opportunity to correct the raw data anisotropy.

Once a level is mipmapped, a second step, called "bricking", is performed. It is an object space decomposition method and consists in taking each volume of the pyramid and divide it into small independent blocks, previously called "bricks". Using independent bricks gives the ability to get access to data in a constant time. 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 to generate unnecessary empty bricks like most approaches that manipulate only power of two volume sizes.

In contrast to Hadwiger *et al.* [10], the data acquisition is not in real-time and has been already performed. It means that treatments could be performed on the data before using them in the system. As we are talking about 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.

Finally, the bricks are stored on a large storage device in raw data or in compressed format. Using a compressed bricks format gives two advantages; the volumes will take less memory space on the storage and the most important one, because the bricks will be lighter it will take less time to load the bricks from the storage to the CPU (see Sec. 6.2). Nevertheless, we are targeting a real-time application using medical images, it means the compression algorithm needs to be lossless and has to provide fast time decompression. To perform this we opted for an LZ4[1] compression scheme.

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

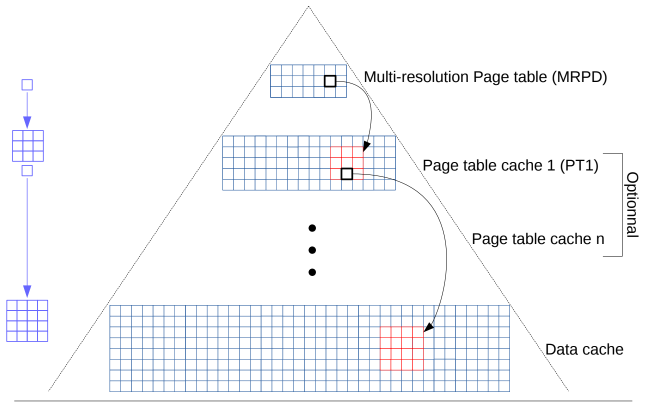

Fig. 2. Multi-resolution, multi-level page table hierarchy virtualization.

## 5 Out-of-core data management

### 5.1 Virtualization

In order to address all the bricks of a large volume from the GPU, we use the approach of the multi-level multi-resolution page table hierarchy. This can be seen as a pyramid where each level virtualize a set of entries in the level below it (see 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, for all level of resolution. This is the only level which is not virtualized and entirely present on the GPU with a low memory impact (see Sec. 6.2). Below the latter, there may be $N$ intermediate levels of PT with a cache for each of them. Finally, the data cache containing the bricks is located at the bottom of the hierarchy. As Hadwiger *et al.* show, only $N = 2$ intermediate PT levels allow to address volumes of several petabytes. This approach of virtualization offer a complexity of $\mathcal{O}(n)$, with $n$ the number of level, to address any voxels, regardless of the desired LOD.

Our implementation of this hierarchy has been planned in a generic way in order to be able to dynamically create as much intermediate PT level table as necessary to address any volume size.

### 5.2 Implementation

The MRPD, the PT caches and the data cache are all implemented with the CUDA surface API [1] in order to read and write elements into 3D texture memory. In the case of a graphic end-user application, the spatial coherent access patterns of the texture memory is a significant benefit. All the caches are managed by an LRU implemented as a simple device vector with the Thrust parallel template library [1].

The data cache is templated in order to store any type of voxels from *GRAY8* to *RGBA32* and more (see Sec. 6.3). Conversely, an entry of PT is always represented with four 32-bits integers to store the

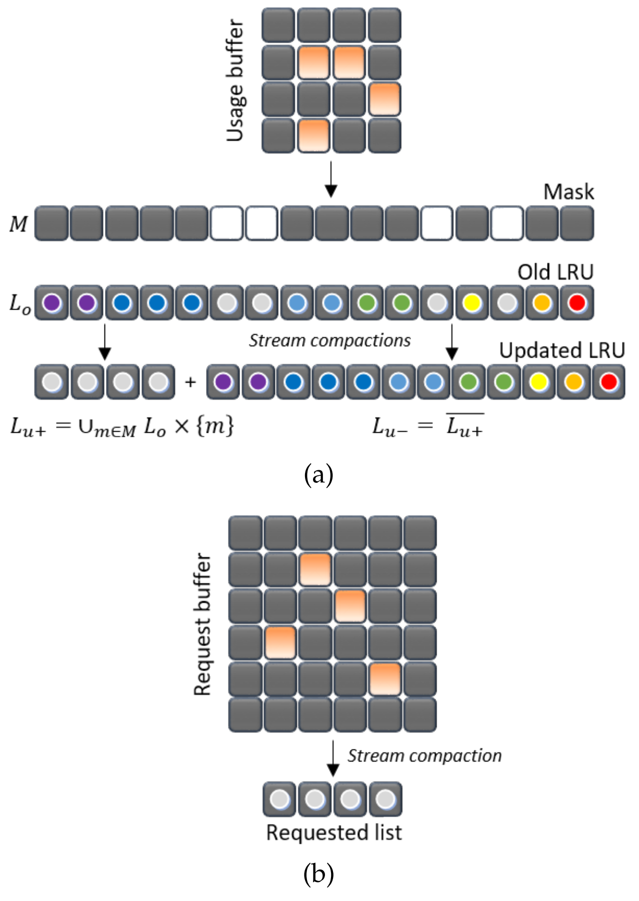

Fig. 3. **Parallel cache manager mechanisms on the GPU** (a) Cache usage update: a single pass to update LRU with double stream compaction (Cartesian products) from a shared buffer to report bricks usage. (b) Bricks request list creation: a single stream compaction from a shared request buffer to create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

3D position of the first element virtualized by this entry in the next level of the hierarchy and a flag representing the state of the virtualized element. This flag can be either *empty*, *mapped* or *unmapped* (not present on the GPU).

**LRU update.** We use the data parallel primitive stream compaction provided with Thrust to update each LRU. This ensures to sort the LRUs by moving the most recently used elements to the beginning of the list while keeping the order of the other ones (Fig. 3-a). To manage the caches, we maintain, on the global GPU memory, *usage buffers* where each buffer contains as many entries as elements present in its respective cache. A global 32-bits integer timestamp is created inside the cache manager and incremented after each update. This timestamp is used to tag, in the *usage buffers*, all the entries of the hierarchy accessed when voxels are requested by CUDA kernels. 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. Then, a mask is created and filled with a boolean flag that indicates if the corresponding usage buffer entry contain the current timestamp or an old one. Finally, this mask plus the old LRU are given as input to the stream compaction operation (Fig. 3-a) to obtain the updated LRUs.

**Cache misses.** Figure 3-b shows the creation of a request list. When a requested brick is not present in the GPU brick cache, a cache miss occurs. This induces a request for this brick. In the same way as the usage buffer, a request buffer is maintained in the global GPU memory. This buffer contains as many entries as bricks in the whole volume and the values are set to the current timestamp to the missing bricks. Then, by using stream compaction on this request buffer, we obtain a complete request list. This list is then limited to a small size in order to limit the number of brick loading requests at each update.

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 form the CPU to the GPU memory by raising PT entries requests (node requests). The PT entries are managed in the same way as the bricks in the brick pool. In our method, the PT entries are updated directly in the GPU avoiding tranfers between the CPU and the GPU. The singles 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, a large volume of $16384^3$ voxels and $64^3$ voxels bricks implies a request buffer of ~77 MB. With a data cache of ~4 GB (which is correct for modern GPUs), it represents less than 2%, which is negligible.

## 5.3 Data fetch

Once a brick request is received by the CPU, its cache handles it. From this point, two scenarios are possible. In the worst case, the bricks are loaded from the storage device, then decompressed and finally added to the CPU cache. Conversely, in the best case the requested bricks are already presents in the CPU cache and can be directly used. Depending on the size of the bricks and the number of bricks to load this steps could take a significant amount of time. To overcome this problem, the load request is forwarded to another CPU thread and frees computing time to process other treatments.

As shown by Crassin, it is more interesting to use Cuda zero copy [1] to load the bricks from the GPU. It ensures optimum uses of the hardware rather than transfering them manually using copy operations triggered from the CPU. To achieve this, we use a *requested buffer* allocated by CUDA in CPU memory. The dedicated asynchronous CPU thread takes care of

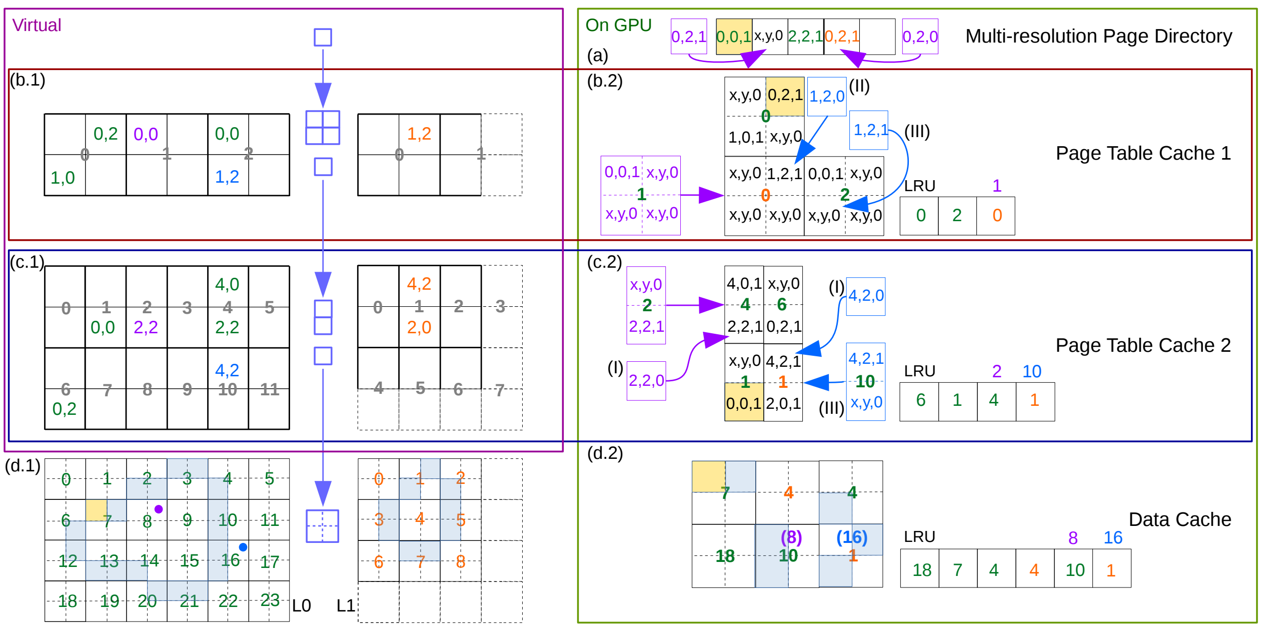

Fig. 4. **Hierarchy updates.**: This 2D example shows the virtualization of a *volume* sized of $12 \times 8$ voxels (d.1) with bricks sized of $2 \times 2$ using an MRPD and 2 levels of PT caches. One entry of the MRPD addresses $2 \times 2$ PT1 cache entries. One entry of PT1 cache addresses $1 \times 2$ PT2 cache entries. One entry of the PT2 cache addresses a brick of $2 \times 2$ voxels. The MRPD and the 3 caches are stored on the GPU with one LRU for each cache. To get access to the voxel [2, 2] of L0 we follow the yellow path as explain by [10]. The entries stored in the MRPD and the PT caches are [X, Y, F] with [X, Y] the coordinate of the beginning of the entry in the next cache and F the flag of this entry (see Sec. 5.2) The accesses to the blue and purple pixels raise a request and a hierarchy update (see Sec. 5.4).

writing the bricks to load in it and then indicates to the GPU when the bricks are available in this buffer. Finally the *write* of all the requested bricks from the *requested buffer* to the data cache is performed with a single pass in a CUDA kernel with one thread per voxel per brick.

### 5.4 Hierarchy updates

The update of the hierarchy follows 3 sequential steps: *i)* removing the references of the bricks that will be removed from the data cache (if the cache is already full), *ii)* writing the new bricks in the data cache and updating its LRU and then *iii)* referencing these new bricks in the hierarchy.

Based on the example given in the figure 4, the purple and blue pixels (Fig. 4-d.1) raise a brick request respectively for the bricks number 8 and 16 of the level L0, written L0.8 and L0.16. Given these 2 bricks to add, we take the last 2 bricks in the LRU of the data cache (L1.1 and L0.10) and remove the link to these bricks from the PT2 cache by switching their respecting linking flags to 0 (Fig. 4-c.2.I). Once this step is done, the bricks are isolated and are not reachable anymore from the MRPD. They are then replaced by the new bricks, the brick L1.1 is replaced by the brick L0.16 and the L0.10 by the L0.8 (Fig. 4-d.2).

Finally, the new bricks need to be referenced in the hierarchy. This steps is performed in 3 sub-steps. We first need to know to which point the new bricks are referenced. From the MRPD an entry to the brick L0.16 exists in the PT1 cache (Fig. 4-b.2). In the PT1 cache the flag for the entry to the brick is set to 0, it means for all caches from this cache to the data cache, no PT entries exist to reference this brick. In a second step, for these caches we take the last PT entry in the LRUs (that has not been added during the current update) and remove its reference in its parent. In our case the last entry in PT2 cache is the entry L1.1 and its reference is removed from the PT1 cache (Fig. 4-b.2.II) by setting its flag to 0. This is done in order to dereference the last entry of the cache and being able to replace it with the new entry. Finally for the same caches we add new PT entries to reference the new brick (Fig. 4-c.2.III) and reference it in its parent (Fig. 4-b.2.III). When done, these 3 final steps are performed once again for the brick L0.8. In general case, these steps are performed sequentially one brick after another (see Sec. 6.3).

## 6 Results

### 6.1 Use case

In order to test our system, we developed a simple 2D slices visualizer. This application allows to navigate

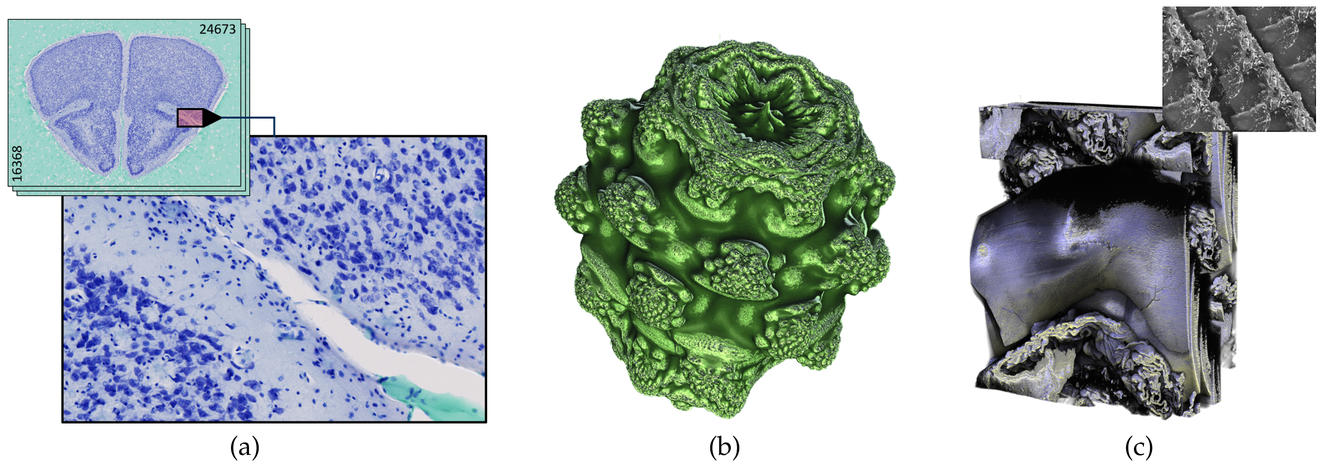

Fig. 5. **Datasets used for system validation and performance analysis.** (a) render with our simple 2D slices visualizer (see Sec. 6.1), (b) & (c) render with our own OptiX-based DVR solution.

(pan and zoom) in a stack of images with 2D multi-resolution rendering. Thus, it allows to graphically validate the coherence, the reactivity of the navigation and the performance of the entire out-of-core data management. We opt for a strategy that promotes the quality of the visual feedback of the user. In this sense, when a query is lifted for a brick of resolution level $l$, the cache manager provides to our renderer a lower resolution brick (if there is one in the cache) the time that the brick of level $l$ arrived in the data cache.

To illustrate the results, we used three datasets (see Fig. 5):

- (a) A stack of 8 histological slides with a resolution of $61856 \times 46383$ RGBA pixels (88 GB).
- (b) A 3D fractal of $2560^3$ RGB voxels (50 GB) generated with Mandelbulb3D.
- (c) A $2160 \times 2560 \times 1072$ volume with grayscale 16bits voxels (11 GB) from a light sheet microscope.

All the tests were carried out on an NVIDIA Grid K2 with 4 GB of VRAM, a CPU Intel Xeon 2.60 GHz with 8 cores and 32 GB of RAM, and an SSD. We used CUDA 8.0 and openGl 4.5 interoperability to render on a $1920 \times 1080$ viewport.

Due to general lack of very high-resolution voxel datasets, we will not test here a hierarchy with multiple levels of page table caches.

## 6.2 Performance & Memory Analysis

We present here the average times of the various stages of maintenance of the caches and of the hierarchy as well as the loading of the bricks on the GPU. These times are calculated on the three datasets by performing the following navigation scenario: a zoom to the most detailed level of resolution, then a navigation in the slices of the image stack and then a 2D displacement in a slice.

We can see that the average loading time of a brick in GPU cache is about 200 µs. The additional cost of the cache manager is about 1500 µs. The latter consists of the management of the LRUs and the management of the brick requests for 1300 µs which is carried out at each update (after each creation of a frame for the application proposed here) and the update of the virtualization hierarchy for 200 µs, which is performed only when a set of bricks are added to the cache.

Moreover, we can note that the rendering is carried out with a rate of about 200 frame per second by our application. This rate is high because the calculation performed for the rendering of a frame is low in the case of the proposed application, and constant since the loading of bricks is performed asynchronously.

Despite the fact that the entire management of the caches and the hierarchy of virtualization are realized on the GPU, its occupancy is between 4% and 6%. This allows to leave a lot of occupancy for an application that requires heavy processing.

The memory occupation of the MRPD is 1.4 GB for (a), 772 kB for (b) and 2 MB for (c).

## 6.3 Discussion

*Weaknesses.*

**Visualization.** Using this system for the visualization implies some limitations regarding the race conditions. The thread that is loading and preparing the bricks cannot load them in the cache and update the hierarchy without inducing possible collision with the GPU.

**Hierarchy updates.** There are 2 main limitations during the update of the hierarchy. The first one occurs when a PT entry is remove from the cache. In figure 4.c.2 when the entry L0.10 overwrite the entry L1.1 we lose the link to the brick that this entry was referencing. Losing the link to the brick L1.1 in 4.d.2 is not a problem because it will be overwritten but we will automatically lose the link to the bricks L1.4. This scenario means we could have bricks present in the data cache but they will not be referenced anymore. If the next update requires and loads the brick L1.4, it will be present twice in the data cache. This problem may occur at any level of the hierarchy and higher it happens more potential references are lost. The second limitation is directly related to the way the referencing of the new bricks is performed in the update itself. Despite this step is computed on the GPU, it cannot use its power of parallelism. Assume that in figure 4 the bricks L0.16 and L0.22 were requested, if the referencing was performed in parallel two table entries L0.10 could be added in PT2 cache. To avoid this scenario, each brick needs to be referenced one by one. For this example, the entry L0.10 would have been added only once and when the brick L0.22 will be referenced, it will modify the just added entry. It implies that more bricks will be loaded for during one update more time this step will take. Therefore, some improvement may be required (See Sec. 7).

### *Strengths.*

With this data structure, the system get access to the data in a constant time which is in $O(n)$ where $n$ is a the number of PT cache plus the MRPD when the data are in the GPU 5.1.

**Image processing.** In contrast to the image visualization, the bricks fetch time for image processing could be optimized. Once a brick is consumed by the processing it is not required to keep it. Thus, the system can work on a just-in-time basis, while it consumes bricks it load an others.

**Data Type.** With this out-of-data management, it is possible to virtualize any kind of data as long as the data used can be represented as a regular 3D grid of scalar values. It uses the CUDA texture memory that can store up to 4 channels with a maximum of 32 bits each. However it is possible to use the global memory instead of the texture, it allows the storage of any kind of data (double type or more than 4 channels). In addition it is possible to virtualize really large volume of data [10].

**Scalability.** This system is be easily scaled in a high performance computing environment. The cache managers deployed on each node are independent and can work on their own process on their side.

## 7 Conclusion & Perspectives

We proposed an out-of-core caching system fully managed on GPUs to address very large volume 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. A simple 2D slice visualizer tools allow us to validate the functioning of our system by allowing an interactive navigation in volumes exceeding the amount of GPU and CPU memory with a very high rendering rate.

**Hierarchy updates.** Actually, the PT hierarchy updates are performed on the GPU but in a sequential context (a one thread kernel). It avoids to load twice the same bricks or to add multiple time the same PT entry but it does not take the advantages of the parallelism of the GPUs. It could be interesting to consider a multi-threaded parallel strategy in order to improve the time spent on this step.

**Transfer times.** The main bottleneck in the system is the bricks loading time from the storage device to the GPU. Data compression addresses a part of this problem, but it could be more interesting to be able to decompress the bricks directly on the GPU. This could take advantages of the multi-threaded environment to implement an efficient decompression algorithm and reduce the amount of transferred data to the GPU. 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, for example by prioritizing requests. With a simple counter, we can determine which bricks have been the most requested. Thus, these bricks have a higher priority for the loading step.

## Acknowledgments

This work is supported by the French national funds (PIA2′program) under contract No. P112331-3422142 (3DNS project). We would like to thank the three clusters (Cap Digital, Systematic and Medicen) that support this project.

We would like to thank Thierry Delzescaux and the Mircen team (CEA, France) for the datasets (a) & (c) Fig.5

## References

[1] Nvidia cuda programming guide 8.0. https://docs.nvidia.com/cuda/index.html. [Online; accessed 2017-April-27].

[2] 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, July 2013.

[3] 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. 1 - état de l'art : rendu volumique.

[4] 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.

[5] C. Crassin. *GigaVoxels : un pipeline de rendu basé Voxel pour l'exploration efficace de scènes larges et détaillées*. phdthesis, Université de Grenoble, July 2011.

[6] 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*, pages 15–22. ACM, 2009.

[7] 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)*, pages 123–124, Oct. 2011.

[8] 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)*, pages 43–51, Oct. 2013.

[9] E. Gobbetti, F. Marton, and J. A. I. Guitián. 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, June 2008.

[10] 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.

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

[12] C. Silva, Y.-j. Chiang, W. Corrêa, J. El-sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *In Visualization'02 Course Notes*, 2002.

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