Visualisations interactives haute-performance de données volumiques massives : une approche out-of-core multi-résolution basée GPUs

Jonathan Sarton

Université de Reims Champagne-Ardenne

École Doctorale Sciences du Numérique et de l'ingénieur

THÈSE

pour obtenir le grade de

Docteur de l'Université de Reims Champagne-Ardenne

Discipline : informatique

présentée et soutenue publiquement

par

Jonathan Sarton

le 28 Novembre 2018

Titre :

Visualisations interactives haute-performance de données volumiques massives : une approche out-of-core multi-résolution basée GPUs

JURY

M. Bruno Lévy	Directeur de Recherche, INRIA Nancy Grand Est	*Président*
M. Bruno Raffin	Directeur de Recherche, INRIA Grenoble Rhône-Alpes	*Rapporteur*
M. Pere-Pau Vázquez	Titular d'Universitat, Université Polytechnique de Catalogne	*Rapporteur*
Mme. Sophie Robert	Maître de Conférence, Université d'Orléans	*Examinatrice*
M. Pierre-Franck Piserchia	Ingénieur de Recherche, CEA DAM Ile-de-France	*Invité*
M. Laurent Lucas	Professeur, Université de Reims Champagne-Ardenne	*Co-directeur*
M. Yannick Remion	Professeur, Université de Reims Champagne-Ardenne	*Co-directeur*

Résumé

Les besoins en visualisation de données volumiques sont courants dans plusieurs domaines scientifiques et en particulier en imagerie médicale et bio-médicale. En effet, plusieurs types d'appareils d'acquisition fréquemment utilisés, génèrent des champs scalaires ou vectoriels, représentés sous forme de grille régulière 3D, qu'il est important de pouvoir visualiser de manière interactive pour en extraire des informations, ou pour valider des résultats expérimentaux. L'accroissement de la précision d'acquisition de ces appareils modernes engendre cependant une hausse exponentielle de la quantité des données générées. Les algorithmes de visualisation doivent, non seulement, faire face à cette problématique en s'adaptant à la volumétrie des données qu'ils manipulent, mais aussi à cette évolution rapide. L'utilisation de cartes accélératrices de type GPU est particulièrement bien adaptée à la nature des données volumiques et aux algorithmes de visualisation généralement associés. Les environnements de calcul haute performance se tournent aujourd'hui vers des solutions qui utilisent un grand nombre de ces cartes. Ceux-ci sont, par leur nature massivement parallèle, de bons candidats pour proposer des solutions de visualisation haute performance. La quantité de mémoire des GPUs est cependant très limitée, et bien moins importante que les données brutes des volumes à manipuler. Une solution est alors de concevoir des algorithmes "out-of-core", dont l'unité de calcul est dissociée de l'unité de stockage des données.

Dans ces travaux de thèse, nous proposons un pipeline complet permettant de visualiser de manière interactive sur GPU, de très grands volumes de données dépassant les capacités physiques de la mémoire du GPU et du CPU de la machine sur laquelle est réalisé le rendu. Nous étudions pour cela, un modèle de gestion "out-of-core", basé sur un principe de virtualisation de la mémoire, particulièrement bien adapté à de très grands volumes. Nous proposons une approche qui comprend une structure d'adressage virtuel, entièrement gérée sur GPU. Nous nous intéressons également à la compatibilité de ce modèle pour différents types d'applications de visualisation de données volumiques. Une première s'appuie sur le principe de microscope virtuel pour proposer une visualisation 3D autostéréoscopique de piles d'images ultra haute résolution. Une seconde propose un rendu volumique direct interactif par une approche "ray-guided", en montrant les capacités d'utilisation de notre modèle de gestion "out-of-core" dans des environnements de calcul haute performance hybrides, multi-GPUs, multi-CPUs.

Abstract

The needs for volume data visualization are common in several scientific fields, in particular in bio-medical imaging. Indeed, several types of frequently used acquisition devices generate scalar and vectorial fields represented as a 3D regular grid. It is important to be able to visualize interactively these volumes, in order to extract information or to validate experimental results. However, the increase in acquisition accuracy of these modern devices induces an exponential growth of the amount of data. To deal with this problem, visualization algorithms must be adapted, both, to the volume of data they handle and its steady growth. The use of GPU accelerator cards is particularly well suited to the nature of volume data and the associated visualization algorithms. High-performance computing environments are now turning to solutions that uses a large number of such cards. These are, by their massively parallel nature, good candidates to offer high-performance visualization solutions. However, the amount of memory in GPUs is very limited, and is much less important than the size of the raw data of the volumes to be handled. One solution is to design out-of-core algorithms, where the computing unit is dissociated from the data storage unit.

In this thesis work, we propose a complete pipeline for interactive visualization on the GPU of very large volumes of data exceeding the physical capacities of the GPU and the CPU memory independently of the machine used for the rendering. For this purpose, we study an out-of-core management model, based on a memory virtualization principle, particularly well adapted to very large volumes. We propose an approach that includes a virtual addressing structure, fully managed on the GPU. We are also interested in the compatibility of this model for different types of volume data visualization applications. We propose a first application that uses a virtual microscope principle to provide autostereoscopic 3D visualization of ultra-high resolution image stacks ; a second one that offers interactive direct volume rendering with a ray-guided approach, showing the usability of our out-of-core management model in hybrid, multi-GPUs, multi-CPUs high-performance computing environments.

Remerciements

Je remercie tout d'abord mes directeurs de thèse, Laurent Lucas et Yannick Remion de m'avoir aidé et soutenu tout au long de ces trois années de thèse ainsi que pour la confiance et la liberté qu'ils m'ont accordées dans mes travaux de recherche.

J'ai le plaisir d'adresser mes remerciements à Bruno Raffin et Pere-Pau Vázquez pour avoir accepté d'être rapporteurs de mon mémoire de thèse. Je remercie Bruno Lévy pour avoir présidé mon jury, ainsi que Sophie Robert et Pierre-Franck Piserchia pour avoir participé à mon jury de thèse, respectivement en qualité d'examinatrice et d'invité. Enfin, pour tous les membres de mon jury de thèse, je voudrais vous remercier pour votre travail et pour l'ensemble des remarques que vous m'avez adressées.

Je tiens également à remercier le laboratoire CReSTIC de l'université de Reims Champagne Ardenne, dans lequel j'ai passé ces quelques années sur ma thèse, ainsi que le département informatique de l'IUT de Reims dans lequel j'ai enseigné pendant cette période. Je remercie les membres de l'équipe RVM, ainsi que tous mes collègues de l'étage. Je souhaite remercier en particulier Nicolas, Ulysse, Joël, Pierre, Joris, Fransisco, Diana, Exavérine, Jimmy, Erwann, David, les personnes avec qui j'ai partagé mon bureau mais aussi beaucoup de très bons moments. J'adresse également des remerciement particuliers aux personnes avec qui j'ai étroitement travaillé, Hervé pour son aide régulière et très précieuse, Florent pour sa collaboration très enrichissante et Nicolas bien sûr, mon plus proche collègue (et mentor en C++) avec qui j'ai passé beaucoup de temps à développer, rédiger, me former, débbogger, voyager en conférence, courrir au squash ...

Mes remerciements vont maintenant à tous mes amis de longue date avec qui j'ai un peu perdu le contact régulier pendant la période de mon doctorat mais qui sont encore très importants pour moi, à eux, merci pour votre soutien. Je me tourne maintenant évidement vers ma famille, qui a été le vecteur principal de ma réussite générale. Je tiens à dire un grand merci à mes parents qui m'ont permis de faire l'ensemble de mes études dans de très bonnes conditions et qui m'ont apporté un cadre général me permettant d'aller aussi loin dans mes projets. Merci à l'ensemble de ma famille et de ma belle-famille pour leur compréhension et pour l'interet qu'ils ont portés sur mon travail et ma situation. Pour finir, je tiens à remercier tout particulièrement ma compagne, Laureen, et mon fils, Basile, pour tout le bonheur qu'ils m'apportent au quotidien et sans qui cette période aurait été beaucoup plus difficile.

Publications & communications

Les travaux présentés dans ce manuscrit de thèse ont fait l'objet de plusieurs publications et communications nationales et internationales qui sont listées ci-dessous :

Publications en conférence internationale

. Jonathan Sarton, Nicolas Courilleau, Anne-Sophie Hérard, Thierry Delzescaux, Yannick Remion et Laurent Lucas – Virtual Review of Large Scale Image Stack on 3D Displays – *IEEE* International Conference on Image Processing (ICIP) 2017. Beijing Chine.

. Jonathan Sarton, Nicolas Courilleau, Yannick Remion et Laurent Lucas – Towards an Interactive Navigation in Large Virtual Microscopy Images on 3D Displays – *IEEE* International Conference on 3D (IC3D) 2016. Liège Belgique.

Publication en cours dans un journal

. Jonathan Sarton, Nicolas Courilleau, Yannick Remion et Laurent Lucas – Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-Of-Core Approach – Soumission en révision mineure à *IEEE* Transactions on Visualization and Computer Graphics (TVCG).

Communication internationale

. Jonathan Sarton, Nicolas Courilleau – A Fully GPU-Based Out-Of-Core Approach to Handle Large Volume Data – GPU Technology Conference Europe (GTC)2018. Munich Allemagne – Présentation 45 minutes.

Communications nationales

. Jonathan Sarton, Nicolas Courilleau, Florent Duguet, Yannick Remion et Laurent Lucas – A fully GPU-based out-of-core approach to handle large volume data – Journées Française d'Informatique Graphique (J.FIG) 2017. Rennes France – Présentation 20 minutes – (2^{nd} prix du "Best paper awards").

. Jonathan Sarton, Nicolas Courilleau, Florent Duguet, Yannick Remion et Laurent Lucas – Visualisation augmentée de grandes masses de données par une approche out-of-core entièrement basée GPU – Journée scientifique VISION 2017. Reims France – Présentation 20 minutes.

. Nicolas Courilleau, Jonathan Sarton, Florent Duguet, Yannick Remion et Laurent Lucas – Une approche out-of-core entièrement basée GPU pour la manipulation de gros volumes de données. – Journée scientifique ROMEO 2017. Reims France – Présentation 20 minutes.

. Jonathan Sarton, Nicolas Courilleau, Florent Duguet, Yannick Remion et Laurent Lucas – Une approche out-of-core entièrement basée GPU pour la manipulation de gros volumes de données – Visu 2017 : Journée annuelle du GT Visualisation du GdR IG-RV. Paris France – Présentation 20 minutes.

. Jonathan Sarton, Yannick Remion et Laurent Lucas – Computer Graphics and High Performance Computing for Visual Big Data – Journées Française d'Informatique Graphique (J.FIG) 2016. Grenoble France – Poster.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

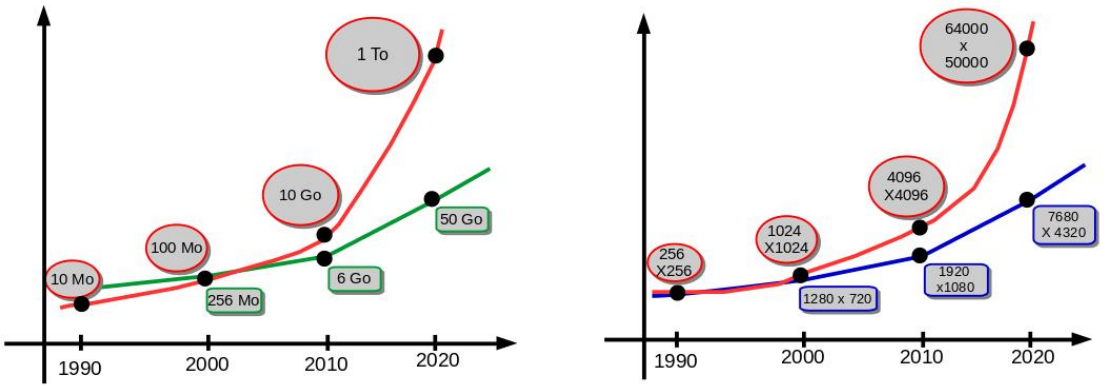
Sommaire

1.1 Contexte scientifique

La visualisation est un outil qui se démocratise de plus en plus, et qui est devenu aujourd'hui indispensable dans tous les domaines scientifiques. Elle est intégrée à la plupart des chaines d'expérimentation, à différents niveaux. On utilise des outils de visualisation aussi bien pour analyser des jeux de données ou en extraire des informations, afin de guider la modélisation de phénomènes, pour valider ou invalider des modèles, que comme outil d'apréciation de résultats expérimentaux. Parmi la quantité de types de données impliqués dans ces visualisations scientifiques, il en est un, la grille régulière 3D, qui se démarque des autres. Ses caractéristiques différenciantes tiennent à la volumétrie des données, à la sémantique quasi-directe en terme spatial et spatio-temporel, ainsi qu'à la spécificité des algorithmes de visualisation qui doivent rendre compte d'informations de surface comme de profondeur, en jouant sur l'occultation et la transparence. Cette thèse se focalise sur ce type de données courante dans plusieurs domaines applicatifs, et en particulier dans le monde biomédical. On y fait référence sous le terme de données volumiques.

On remarque une rapidité dans l'accroissement de la taille des données pour ce genre de représentation. Les appareils modernes d'acquisition biomédicale sont capables de générer des données ultra haute résolution permettant d'obtenir des informations à l'échelle microscopique. En revanche, cette précision engendre des données brutes de très grandes tailles, pouvant dépasser plusieurs teraoctets. La croissance forte de la volumétrie des données brutes générées n'est malheureusement pas comparable à l'évolution bien plus lente de la mémoire des GPU (voir figure 1.1(a)) En conséquence, l'écart croissant entre volume de données et mémoire de travail impacte directement et durablement les algorithmes de visualisation utilisés. L'évolution des écrans, supports d'affichage de ces outils de visualisation, n'a pas suivi la même tendance d'évolution en terme de définition (voir figure 1.1(b)). Cet antagonisme ne permet pas d'utiliser directement les algorithmes classiques en visualisation, et implique de devoir les adapter. Cela nous interroge sur l'évolution de ces outils de visualisation, qui doivent être revisités pour répondre à ces nouvelles contraintes. L'importance de visualiser ces données volumiques massives de manière interactive, afin de pouvoir "naviguer" de manière fluide dans celles-ci, conduit à un domaine actif de recherche sur le thème de la visualisation volumique haute-performance.

Les environnements de calcul haute performance, de part leur définition, semblent être une solution intuitive pour aborder la problématique sous-jacente. Ces environnements de calcul, tels que les super-calculateurs, ou les serveurs de calcul plus modestes, sont de plus en plus nombreux et performants. Un grand nombre de ces plateformes se tournent aujourd'hui vers des solutions matérielles hybrides, équipées à la fois de processeurs standards et de processeurs graphiques utilisés pour l'accélération des calculs. Venant appuyer cela, les évolutions récentes de ces cartes accélératrices de type GPU (Graphics Processor Unit) ont favorisé l'arrivée de méthodes basées sur ce type d'architecture. Ce sont aujourd'hui les principales solutions proposées pour l'exploration interactive de données volumiques, aussi bien sur PC grand public que dans de tels environnements. Leur architecture parallèle étant particulièrement adaptée à ces problématiques de visualisation, du fait de l'organisation régulière des pixels à l'écran, et de la cohérence spatiale de la contribution de ceux-ci par rapport aux données elles-mêmes. L'architecture massivement parallèle des GPUs et les configurations matérielles multi-GPUs dans des serveurs dédiés, semblent donc être des solutions naturelles pour un outil de visualisation haute-performance. Cependant, la capacité mémoire restreinte des GPUs, dont l'évolution est moins rapide que celle de la précision et



(a) (b)

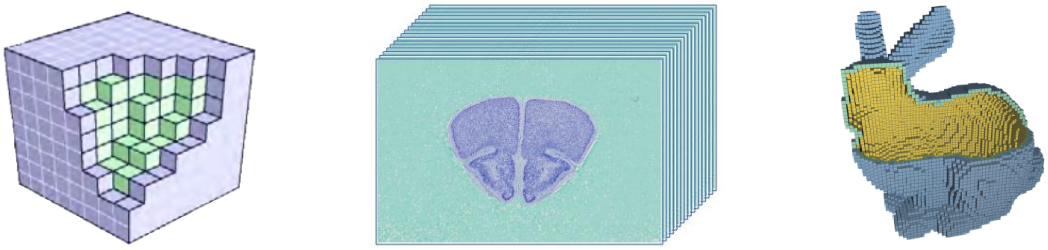
FIGURE 1.1 – **Tendances d'évolution.** Ces courbes représentent la tendance de l'évolution au cours de ces 30 dernières années (a) de la quantité mémoire nécessaire pour stocker les données volumiques (courbe rouge), confrontée à la quantité mémoire physique disponibles sur les GPUs (courbe verte), et (b) des dimensions de ces données (courbe rouge), confrontées aux dimensions des écrans (courbe bleue).

du volume des données, est un facteur limitant pour la manipulation de grands volumes de données. C'est pourquoi une des approches permettant de gérer des données massives pour la visualisation sur GPU, se tourne vers des solutions dites "out-of-core". Dans ce contexte, il s'agit de séparer l'unité de calcul de l'unité de stockage des données. Les méthodes classiques de rendu sur GPU utilisent des données qui sont entièrement présentes dans leur unité de mémoire à accès rapide. Comme il n'est pas possible de maintenir des données à l'échelle du teraoctet dans cette mémoire restreinte, celles-ci doivent donc être acheminées à la demande, depuis des unités de stockage plus lentes comme des disques durs. Il est alors important pour l'utilisateur de garantir un retour graphique cohérent et fluide, sans se soucier de l'emplacement mémoire des données qu'il manipule.

La perception des données visualisées est aussi un facteur important à prendre en compte. En particulier pour l'analyse de données numériques issues d'acquisition d'objets physiques, habituellement observés avec un microscope par exemple. Nous proposons de nous intéresser à la visualisation de données ayant différentes cohérences spatiales :

— en 2D : images ultra-haute résolution issues d'une pile d'images composant un volume 3D, généralement observées comme une lame avec un microscope électronique ;
— en 3D : structures dans un volume considéré comme une entité 3D à part entière.

La visualisation volumique haute-performance de données massives a déjà été largement étudiée durant cette dernière décennie dans le cadre du rendu volumique direct par lancer de rayon. Dans cette thèse, nous proposons de nous intéresser à différents types de visualisations, répondant ainsi aux différents besoins venant de plusieurs types d'acquisitions biomédicales. Ceci dans un contexte de gestion out-of-core des données, aussi bien sur une machine standard avec une visualisation locale, qu'avec une solution de visualisation distante permettant de profiter pleinement de la puissance de calcul de serveurs de rendu multi-GPUs déportés.



(a) (b) (c)

FIGURE 1.2 – **Grille régulière de voxels** Les grilles régulières de voxels, illustrées à gauche (a), peuvent provenir directement de l'acquisition, comme avec cette pile d'images biomédicales au milieu (b), ou encore de la voxélisation d'un maillage, ici à droite (c) (source: Schwarz et Seidel [SS10]).

1.2 Visualisation volumique haute performance

Cette section nous permet d'introduire les concepts de bases utiles à la compréhension des techniques avancées de visualisation et de gestion de données décrites dans la suite de ce manuscrit. Nous abordons notamment la représentation volumique des données en grille régulière 3D de voxels, une introduction à la problématique de la visualisation de données volumiques massives, une présentation de la programmation sur GPU et l'utilisation d'environnements de calcul haute performance (HPC en anglais pour High Performance Computing).

1.2.1 Représentations volumiques

Un jeu de données volumiques est représenté sous forme d'une grille de voxels en trois dimensions. Cette grille peut être régulière ou non, mais dans cette thèse, nous nous intéresserons uniquement aux grilles régulières de voxels. Un voxel (pour "Volume ELement" en anglais) est une extension d'un pixel en trois dimensions, et peut être vu comme une cellule 3D parallélépipédique. La grille devient alors une juxtaposition ordonnée de cellules 3D sur lesquelles les données scalaires ou vectorielles sont déployées. Ces données peuvent être interprétées comme valeurs homologues de chaque cellule 3D ou bien comme valeurs ponctuelles des sommets de ces cellules. La première interprétation induit un champ 3D discret alors que la seconde tend à fournir par interpolation, un champ compact. En pratique, c'est la deuxième représentation qui est utilisée. Une grille regulière 3D composée de voxels, modélise l'ensemble d'un objet, sa surface ainsi que son "intérieur" (voir Figure 1.2a).

On retrouve ce type de représentation dans plusieurs domaines. Dans l'industrie du cinéma par exemple, souvent pour modéliser des effets de fumée ou de nuages comme on peut le voir sur la Figure 1.3b. Les jeux vidéos exploitent également ce genre de représentation volumique (voir Figure Figure 1.3a). On retrouve également leur utilisation dans beaucoup de disciplines scientifiques. Une grille régulière 3D peut être obtenue directement par l'empilement d'une série d'images 2D (voir Figure 1.2b), comme c'est souvent le cas en imagerie biomédicale, ou alors après une étape de voxélisation d'un modèle 3D ((voir Figure 1.2c)). Cette étape consiste à transformer un objet représenté sous forme de maillage de triangles par exemple, en un ensemble de voxels.

(a) (b)

FIGURE 1.3 – **Utilisation des voxels dans le divertissement.** La représentation voxels est utilisée pour le divertissement, (a) dans l'industrie du jeux vidéo (ici avec Minecraft), ou encore dans le cinéma et l'animation (b) ici avec un nuage provenant de Disney Animation Studios.

Imagerie biomédicale

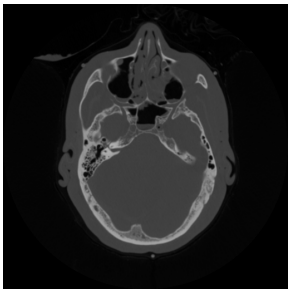
La représentation de données sous forme de grille régulière 3D est particulièrement courante en imagerie biomédicale. Il existe plusieurs types d'appareils d'acquisition permettant de générer des images 2D dans ces disciplines. Nous présentons ci-dessous une classification des principales techniques d'acquisition :

— Par tomographie – Ce type d'acquisition est utilisé en particulier par les scanners CT (Computed Tomography), qui envoient une série de rayons X sur un objet physique 3D depuis plusieurs points de vue (voir Figure 1.4a). Une paire de capteurs permettant d'émettre et de réceptionner un rayon X, tourne tout autour de l'objet à scanner, et se déplace sur toutes les "tranches" de cet objet. L'imagerie IRM (Imagerie par Résonance Magnétique) quand à elle, utilise la résonnance magnétique nucléaire en plaçant l'objet à scanner dans un champ magnétique puissant et stable (voir Figure 1.4b).

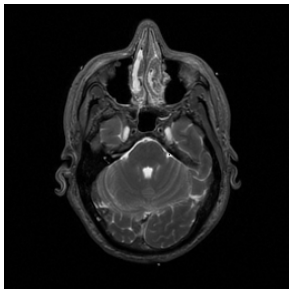
— Par coupes sériées – pouvant être déclinées en deux catégories : réelles ou virtuelles. La première consiste à numériser des lames biologiques contenant des "tranches" physiques d'un modèle découpé au préalable. Cette technique est notamment utilisée pour l'étude de coupes en histopathologie (voir Figure 1.4d). On parle plutôt de tranchage optique pour la deuxième catégorie. Ce principe est utilisé en microscopie confocale ou à feuillet de lumière (voir Figure 1.4e) en captant le signal émis par l'échantillon, lorsque ce dernier est excité par une source laser sur une longueur d'onde donnée.

Ces différentes méthodes d'acquisition d'images médicales génèrent une séquence d'images d'un même objet (voir Figure 1.4c), alors disposées sous forme de pile d'images qui composent ainsi une grille régulière 3D, un ensemble de voxels.

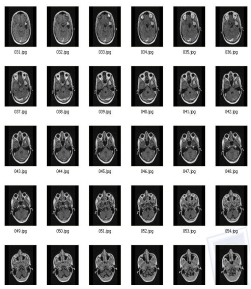
La problématique de la réprésentation des ces données sous forme de grille régulière 3D se situe dans leur forte volumétrie. Les tendances actuelles d'évolution des appareils d'acquisition provoquent d'une part, la multi-modalité des différents jeux de données et d'autre part, l'amélioration de la précision d'aquisition engendrent des données de plus en plus volumineuses. La conjonction de ces deux aspects fait naturellement croître la difficulté de pouvoir visualiser et/ou traiter ces volumes en temps réel.



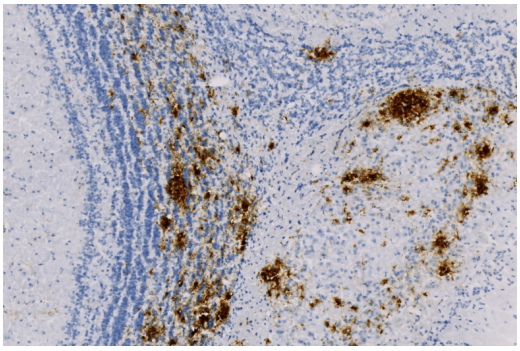
(a) CT-scan



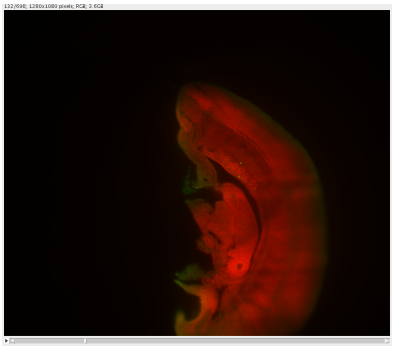
(b) IRM



(c) Séquence d'image CT-scan



(d) Lame histologique



(e) Microscope à feuillet de lumière

FIGURE 1.4 – **Exemple d'acquisition d'images médicales** (source: (a, b, c) Visible Human Project, (c, d) CEA Direction des Sciences du Vivant – Projet 3DNeuroSecure).

1.2.2 Visualisation de données volumiques massives

Il existe plusieurs techniques de visualisation volumique répondant chacune à différents besoins et à différentes natures physiques des données à visualiser. Les méthodes de rendu qui nous intéressent seront étudiées dans le chapitre 2. Comme nous le verrons plus en détails dans la suite de ce manuscrit, la visualisation interactive de grilles de voxels se fait aujourd'hui principalement sur GPU. Le pipeline classiquement utilisé consiste alors à transférer le ou les jeux de données que l'on souhaite visualiser, en mémoire vidéo par une seule copie explicite émise par le système central (CPU + RAM). Les données sont donc entièrement présentes en mémoire GPU durant tout le processus de visualisation. Il est alors possible d'accéder directement à celles-ci depuis l'unité de calcul du GPU, en profitant de la rapidité d'accès à sa mémoire interne. Ce principe repose directement sur la taille initiale du jeu de données à visualiser. La visualisation de données massives implique de concevoir un pipeline adapté à la gestion de cette grande quantité de données à manipuler. On parle alors d'algorithme sensible à la sortie ("output-sensitive" en anglais). L'idée est ici que l'algorithme de rendu ne dépende pas de la taille de l'entrée (jeu de données), mais plutôt de la taille de la sortie, c'est à dire, la résolution de l'écran sur lequel on produit l'affichage. Le pipeline correspondant doit alors intégrer une structure permettant la gestion out-of-core des données. Ainsi, la problématique est de maintenir une efficacité du rendu, tout en fournissant une abstraction de l'emplacement mémoire des données à différents niveaux. La solution de visualisation doit, elle aussi, s'adapter à cette problématique. Il est nécessaire de construire l'espace de travail utile parmi l'ensemble du jeu de données global et de le maintenir au fur et à mesure des besoins de l'étape de la visualisation interactive.

1.2.3 Utilisation des GPUs et des environnements HPC

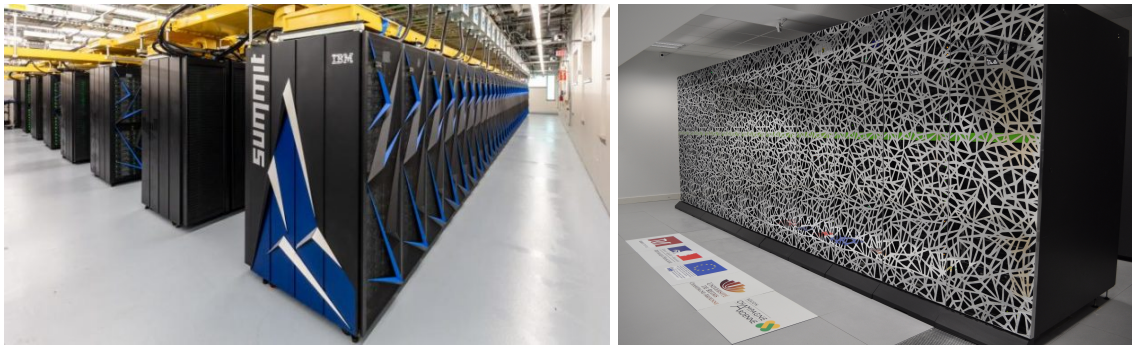
L'utilisation des GPUs fait entièrement partie des travaux de recherche présentés dans cette thèse. C'est pourquoi, nous présentons ici l'évolution de leur architecture, ainsi que leur modèle de programmation GPGPU actuel. Nous discuterons également des environnements de calcul haute performance qui utilisent de plus en plus ce genre de matériel pour l'accélération des calculs. Deux constructeurs occupent aujourd'hui principalement le marché des cartes accélératrices de type processeurs graphiques : AMD et NVIDIA. Ce dernier ayant été consulté lors de la phase préparatoire du projet, les acteurs associés au consortium 3DNeuroSecure se sont orientés vers leurs technologies. Ce choix est fait en particulier avec l'utilisation du centre de calcul *ROMEO* de l'Université de Reims Champagne-Ardenne, qui est fortement équipé de tels processeurs graphiques. C'est pourquoi, dans cette thèse, nous nous concentrerons uniquement sur l'étude et l'utilisation des GPUs Nvidia.

Architectures des GPUs modernes.

FIGURE 1.5 – **Architecture des cartes Nvidia Volta V100.** Organisation des différents coeurs de calcul et niveaux de mémoire au sein d'un multi-processeur. (source: Nvidia – http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf).

Les GPUs ont beaucoup évolué durant les deux dernières décennies. Après l'arrivée et l'évolution des modèles programmables de shaders depuis le début des années 2000, c'est l'évolution des micro-architectures pour la programmation GPGPU qui a marqué ces dernières années. Aujourd'hui, le modèle de programmation et les évolutions matérielles sont surtout encouragés par les besoins croissants dans les domaines de l'intelligence artificielle ou encore de l'informatique graphique. Les dernières architectures et leurs couches logicielles proposent en particulier l'utilisation d'unités de calcul appellées "tensor cores" dédiées au calcul matriciel, adaptées à l'apprentissage profond ("deep learning" en anglais).

Les architectures des GPUs sont conçues comme un environnement massivement parallèle. En effet, celles-ci sont organisées en un ensemble de multi-processeurs (voir Figure 1.5), composés d'un très grand nombre de coeurs de calcul (5120 coeurs sur une carte Nvidia V100, la carte la plus récente au jour où ce manuscrit est rédigé). Le paradigme d'implémentation sous-jacent repose sur un modèle de programmation SIMT ("Single Instruction, Multiple Thread"). Ce principe permet d'opérer un même jeu d'instructions en parallèle sur un grand nombre de données distinctes, et il est particulièrement bien adapté aux méthodes de rendu volumique modernes. L'API CUDA maintenue par Nvidia, permet d'utiliser directement les instructions du GPU au travers de langages haut niveau comme le C++. Cette API évolue en même temps que les modèles d'architecture des cartes, et propose aujourd'hui des fonctionnalités très intéressantes. Il est important de prendre en compte



(a) (b)

FIGURE 1.6 – **Environnement de calcul haute performance.** Illustration d'environnements de calcul haute performance équipés de solution parallèles hybrides avec des noeuds de calculs multi-CPUs et multi-GPUs. (a) Le calculateur américain *SUMMIT* classé premier au TOP500 à sa sortie en Juin 2018. (b) Le calculateur du méso-centre français *ROMEO*, hebergé à l'Université de Reims Champagne-Ardenne et classé 249^{e} au TOP500 à sa sortie en Juin 2018 .

ces évolutions matérielles qui ont un lien direct avec les solutions proposées dans la littérature, même dans un contexte de recherche purement académique. Bien que les GPUs offrent une importante puissance de calcul exploitable dans la conception d'algorithmes de visualisation interactive, ceux-ci imposent des contraintes dans la gestion des données, d'un point de vue de la mémoire et des communications.

Mémoire GPU

A l'instar des systèmes "classiques" équipés de CPUs, les processeurs graphiques possèdent plusieurs niveaux de mémoires, avec des accès plus ou moins rapides. On retrouve d'abord la mémoire globale, qui offre la plus grande capacité mais qui, en contrepartie, est celle dont l'accès est le plus lent. Cette zone mémoire propose plusieurs types de stockage dont la mémoire texture qui offre plusieurs avantages. Elle permet, entre autre, de stocker des données avec une cohérence spatiale 1D, 2D ou 3D avec des patterns d'accès optimisés pour ce genre de représentation. Elle permet également des accès normalisés avec interpolation trilinéaire matérielle et possède son propre système de cache. Il existe ensuite plusieurs niveaux de caches à accès rapide, le cache L2 et le cache L1, ainsi qu'un certain nombre de registres. Les caches sont gérés automatiquement, et il n'est pas possible d'avoir de contrôle dessus. Depuis l'architecture Pascal, les cartes Nvidia proposent une gestion de virtualisation de la mémoire avec le principe de mémoire unifiée (voir section 2.4.2). Cependant, ce système n'est aujourd'hui pas assez complet et n'offre pas assez de contrôle sur la gestion des défauts de page. Il ne peut pas être utilisé comme unique solution à la gestion de gros volumes de données dépassant la quantité mémoire GPU et CPU. Bien que la capacité mémoire des GPUs soit de plus en plus importante, celle-ci est très restreinte et reste ainsi un facteur limitant. Cet aspect est au coeur des travaux que nous présentons ici puisqu'il induit directement la problématique de cette thèse.

Environnements de calcul haute performance

Les clusters de calcul, super-calculateurs et autres serveurs offrant d'importantes ressources de calculs, proposent aujourd'hui de plus en plus des environnements équipés de

processeurs graphiques en grand nombre pour l'accélération matérielle (voir Figure 1.6). Ces environnements multi-GPUs offrent alors un autre niveau de parallélisation. L'exploitation de tels environnements implique de concevoir des applications avec plusieurs niveaux de paradigmes de calcul parallèle, aussi bien sur CPU que sur GPU, ainsi que plusieurs couches de communications reliant l'ensemble des ressources. Les échanges de données entre les différentes unités de calcul impliquent une surcouche logicielle pour les communications inter- et intra-noeuds, qui reposent souvent sur les APIs OpenMP et MPI respectivement. Ces communications apparaissent à plusieurs niveaux dans un environnement hybride multi-GPUs, multi-CPUs, puisqu'elles peuvent intervenir entre n'importe quelles paires de ces appareils. Les solutions de visualisation développées pour ce type d'environnement doivent fournir une répartition efficace des données et des calculs sur les différentes unités. Cela permet une mise à l'échelle du point de vue des calculs, comme de celui de la mémoire.

1.3 Motivations et objectifs

Les travaux de cette thèse s'inscrivent dans le cadre du projet **3DNeuroSecure**. Dans cette section, nous allons présenter le projet dans son ensemble avant de détailler les objectifs de cette thèse dans ce contexte.

1.3.1 Projet 3DNeuroSecure

Le projet **3DNeuroSecure** s'inscrit dans la deuxième phase du Programme d'Investissement d'Avenir (PIA2), dans le domaine du calcul intensif et du secteur médical. Il vise à proposer une solution collaborative, sécurisée et haute performance, pour l'innovation thérapeutique dans le monde biomédical. La solution doit notamment permettre la navigation interactive dans des données visuelles massives, ayant pour cadre applicatif l'imagerie biomédicale 3D ultra-haute résolution (de l'ordre du micron) possiblement multi-modale. La preuve de concept de ce projet est le développement des molécules contre de nouvelles cibles thérapeutiques identifiées dans la maladie d'Alzheimer.

Les données numériques biomédicales servant au sein du projet, sont obtenues au moyen de différents types d'appareils d'acquisition permettant de générer des données 3D très détaillées, à partir de modèles d'animaux. Ces acquisitions se font principalement sur des cerveaux de rongeurs ou de primates, soit à partir d'un scanner de lames, soit au moyen d'un microscope à feuillet de lumière. Le premier étant utilisé pour l'étude de coupes histologiques avec des marqueurs permettant de mettre en évidence certains élements. Ce type d'appareil génère des images avec une précision de l'ordre du micron dans le plan, pour une distance entre coupes beaucoup plus importante, ce qui provoque une très forte anisotropie. Le deuxième permet une acquisition d'images en fluorescence dévoilant ainsi certaines structures dans un volume en trois dimensions.

Un des aspect important du projet est l'exploitation d'environnements de calcul haute performance. La solution proposée doit pouvoir offrir un déport des calculs en environnements HPC, avec une visualisation distante sur différents périphériques, allant du PC de bureau classique aux terminaux légers comme des tablettes 3D sans lunettes. La Figure 1.7 illustre l'ensemble des aspects qui entrent en jeu dans ce projet.

FIGURE 1.7 – **Illustration du projet 3DNeuroSecure**

Ce projet est organisé autour d'un consortium composé de six partenaires issus de l'industrie et du secteur académique de la recherche :

— Neoxia (porteur du projet) : Cabinet de conseil en technologies numériques, qui se positionne dans ce projet comme le pilote et le coordinnateur de l'ensemble des équipes du projet.
— CEA : Commissariat à l'énergie atomique et aux énergies alternatives, qui s'intègre dans ce projet avec deux de ces départements : les équipes de la Direction des Applications Militaire, qui travaillent dans le domaine RD sur la simulation et le calcul scientifique, la surveillance de l'environnement et le respect des traités internationaux, ainsi que sur l'électronique et la physique nucléaire ; et la Direction des Sciences du Vivant, qui a pour mission de développer et d'évaluer des stratégies de recherche en réponse aux problèmes de santé publique et de fournir une plateforme d'imagerie préclinique dédiée à l'étude des maladies neurodégénératives.
— Tribvn : PME française spécialisée dans l'acquisition et l'interprétation des images de microscopie.
— ESIEA/CNS : Le laboratoire Confiance Numérique et Sécurité (CNS) aborde au sein de l'ESIA, l'ensemble des domaines liés à la cyberdéfense et à la sécurité de l'information et des systèmes.
— Archos : Leader dans l'électronique grand public, qui propose aujourd'hui des tablettes, des smartphones ainsi que des objets connectés au niveau mondial.
— URCA/CReSTIC : Le Centre de Recherche en STIC (CReSTIC) et la Maison de la Simulation (MaSCA) de l'université de Reims Champagne Ardenne mobilisent leurs compétences en simulation sur architecture HPC accélérée et en imagerie numérique en 3D relief auto-stéréoscopique. Le Centre de Calcul ROMEO fournit pour le projet le plus puissant supercalculateur "vert" de France.

Dans ce contexte, deux thèses sont proposées en colaboration avec le laboratoire CReSTIC (Centre de Recherche en STIC) de l'Université de Reims Champagne-Ardenne : la thèse présentée dans ce manuscrit et celle de Nicolas Courilleau. Ces deux thèses ont pour but de collaborer en partie, et de se compléter d'un point de vue des apports scientifiques. La section 1.4 identifie les contributions propres à cette présente thèse et celles qui sont communes avec Nicolas Courilleau.

1.3.2 Objectifs

Cette thèse a pour objectif de prendre en charge des données biomédicales massives à des fins notamment de visualisation interactive. Ces données sont caractérisées par :

— leur nature : les données manipulées ont une représentation volumique possiblement multi-modales (voir Section 1.2.1) ;

— leur niveau de précision : les volumes haute résolution générés engendrent des données brutes de plusieurs gigaoctets à quelques teraoctets.

L'utilisation de cartes accéleratrices de type GPU (Graphics processing Unit) se justifie dans ce contexte en vue de leur architecture massivement parallèle, et de leur efficacité pour traiter des problèmes de rendu temps-réel. Ces deux dernières décennies, l'évolution de ce type de matériel a permis de rendre leur pipeline plus flexible, et ainsi de pouvoir manipuler efficacement des données volumiques pour leur visualisation ou leur traitement. En revanche, le facteur limitant dans l'utilisation de GPU est leur capacité mémoire restreinte.

Ainsi, les objectifs de la thèse présentée dans ce manuscrit sont les suivants :

1. Proposer une structure de données capable de manipuler des données volumiques massives en temps-réel depuis un GPU. Cette structure, en plus d'être adaptée à la représentation volumique des données, devra permettre d'adresser un volume de très grande dimension dans un contexte out-of-core. Elle devra également être conçue de manière à pouvoir être utilisée dans un environnement HPC hybride multi-CPU et multi-GPU. Finalement, cette structure sera adaptée à la visualisation d'une part, et aux traitements d'autre part, des données qu'elle manipule. Ce développement est mené en commun avec Nicolas Courilleau.
2. Proposer plusieurs solutions de visualisation interactive dans un pipeline complet intégrant la structure de données précédement décrite. Les approches retenues doivent permettre de naviguer de manière fluide dans les données et proposer plusieurs manières différentes de visualisation, en 2D ou en 3D relief. Il doit ainsi être possible de visualiser une pile d'images de données généralement observées au microscope électronique, aussi bien qu'un volume en 3D avec des méthodes de rendu volumique direct.

Même si cette thèse se concentre sur la visualisation des données, la structure de données retenue doit être adaptée à la réalisation de traitements sur ces données. Les traitements sont prévus dans le cadre de la deuxième thèse sur ce projet (celle de Nicolas Courilleau), qui s'est déroulée en partie sur la même période que la thèse décrite dans ce manuscrit. Les deux thèses se sont accordées sur une solution de gestion de gros volumes de données volumiques permettant la visualisation, et le traitement à la demande en temps interactif. La structure proposée ainsi que l'ensemble du pipeline dans lequel elle s'intègre devait donc être adaptée aux problématiques de ces deux thèses.

1.4 Contributions

Dans ces travaux de thèse, nous présentons une méthode capable de manipuler efficacement n'importe quelle grille régulière 3D depuis un ou plusieurs GPUs afin de pouvoir visualiser et / ou traiter ces données de manière interactive. L'approche proposée est en particulier capable de gérer des volumes de données dépassant la quantité mémoire disponible sur GPU ou sur CPU dans un cadre applicatif très général. Pour cela, nous proposons un pipeline complet, du disque dur au GPU, qui inclut une structure d'adressage out-of-core des données entièrement intégrée et gérée sur GPU. Cette structure d'adressage virtuel est une hiérarchie de table de pagination multi-niveaux et multi-résolution. Une telle structure est utilisée de manière à pouvoir virtualiser de très grands volumes multi-résolution accessibles via une API depuis le GPU, sans ce soucier, au niveau applicatif, de l'emplacement mémoire physique de la donnée. Cette approche utilise également un système de caches sur GPU, permettant de stocker les données nécessaires au fur et à mesure des demandes de l'application.

Dans ce contexte, nos contributions portent plus particulièrement sur la gestion d'une telle structure d'adressage et des caches entièrement sur GPU. De plus, cette gestion a été pensée de manière très générique, afin d'être adaptée à n'importe quel système de visualisation interactive et / ou de traitement à la demande, sur des données utiles à chacun des process traités. Les contributions présentées sur cet aspect font l'objet d'une collaboration à part égale avec les travaux de thèse de Nicolas Courilleau.

Nous proposons également deux applications de visualisation qui utilisent ce pipeline :

— Un système de microscope virtuel permettant de visualiser une pile d'images 2D ultra haute résolution. Nos contributions portent ici sur l'utilisation de notre structure out-of-core pour la navigation interactive dans un volume 3D, avec la possibilité de visualiser des coupes 2D sur un dispositif d'affichage auto-stéréoscopique multi-vues. Pour ce faire, nous proposons la génération interactive d'images multi-vues afin de reproduire une information de profondeur obtenue par l'observation de données avec un microscope classique. Les contributions présentées sur cet aspect font l'objet d'une collaboration à part égale avec les travaux de thèse de Nicolas Courilleau.

— Un lancer de rayon volumique out-of-core sur GPU. Dans ce contexte, nous proposons la distribution de notre structure d'adressage out-of-core dans un environnement multi-GPUs avec des calculs déportés sur un serveur de rendu pour un affichage distant sur des clients légers (PC, tablette 3D). Les contributions présentées sur cet aspect sont propres à cette thèse et n'entrent pas dans le cadre d'une collaboration avec celle de Nicolas Courilleau.

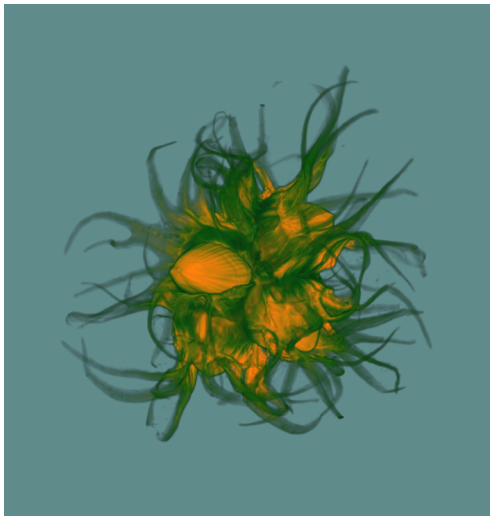
En résumé, nous introduisons un pipeline complet pour une approche de gestion out-of-core de données volumiques capable de manipuler de très grands volumes de données en temps interactif sur GPU. Notre méthode permet de tirer pleinement parti de l'architecture massivement parallèle des GPUs dans un cadre applicatif général. De plus, elle est adaptée à un environnement de visualisation multi-vue et utilisable dans des environnements hautes performances multi-GPUs. Ces points sont validés par deux développements applicatifs.

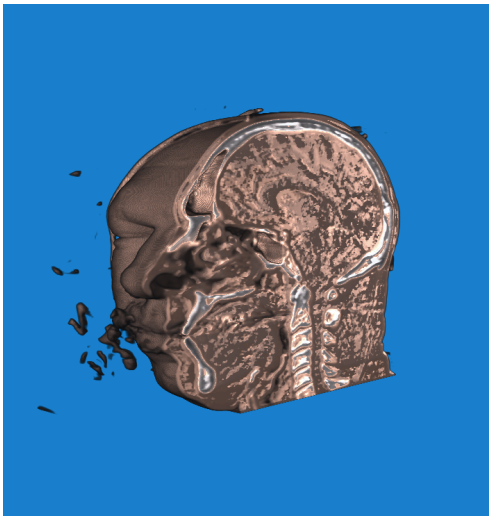
1.5 Plan de thèse

Ce manuscrit de thèse est structuré en 5 chapitres en complément de cette introduction et de la conclusion. Le chapitre 2 fournit un état de l'art des différents aspects traités dans cette thèse. Nous nous intéresserons ainsi à l'étude des méthodes existantes dans la littérature scientifique, et à l'introduction des concepts généraux. Cette étude portera sur les techniques de visualisation scientifique de données volumiques, le rendu volumique en temps-réel sur GPU et la gestion de données out-of-core pour le rendu volumique de grandes dimensions. Nous présenterons ensuite, dans le chapitre 3, notre solution de gestion out-of-core de données sur GPU en exposant l'ensemble du pipeline que nous proposons, puis en détaillant la structure d'adressage virtuel utilisée et sa gestion complète sur GPU. Le chapitre 4 introduit un microscope virtuel adapté à la visualisation de coupes biologiques ultra-haute résolution sur écran multi-vues auto-stéréoscopique. Le chapitre 5 présente un système de visualisation distante par lancer de rayon volumique multi-GPUs tirant parti de la puissance de calcul de serveur de rendu. Nous présenterons ensuite les résultats obtenus pour évaluer l'ensemble de ces travaux dans le chapitre 6. Nous terminerons ensuite ce manuscrit par le chapitre 7 qui propose une conclusion sur les travaux réalisés dans le cadre de cette thèse ainsi que quelques perspectives.

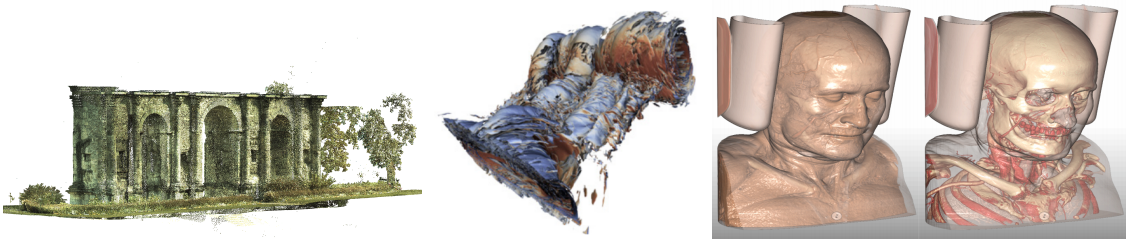
Chapitre 2

Etat de l'art





Sommaire



(a) (b) (c)

FIGURE 2.1 – **Méthodes de visualisation scientifique** (a) Nuage de points, (b) Rendu surfacique, (c) Rendu volumique (source : (a) illustration issue de la plateforme de visualisation MINT du CReSTIC, (b) [WJA+17], (c) [CM11]).

2.1 Introduction

Dans ce chapitre, nous présentons un état de l'art spécifique aux aspects techniques et scientifiques traités dans le cadre de cette thèse. Les travaux abordés s'étendent sur plusieurs domaines de recherche : la visualisation scientifique, le calcul haute performance sur GPU, le rendu volumique temps-réel et la gestion de mémoire out-of-core. Le dernier aspect est sans doute le plus caractéristique des travaux de recherche de cette thèse. Cependant, il est nécessaire d'introduire certaines notions de visualisation au préalable, afin de pouvoir discuter ensuite des méthodes de gestion out-of-core de la mémoire, associés à ce contexte. Nous commencons donc par discuter dans la section 2.2, de visualisation scientifique en se concentrant plus particulièrement sur les méthodes de visualisation 2D et 3D de données biomédicales représentées de manière volumique sous forme de grilles 3D. Nous nous intéressons ensuite, dans la section 2.3, aux méthodes de rendu volumique en temps-réel, basées GPUs. Nous abordons enfin les problématiques de représentation et de stockage out-of-core dans la section 2.4, ainsi que les techniques d'adressage, en temps-réel à la demande, de données volumiques massives depuis le GPU.

2.2 Visualisation scientifique de données volumiques

Dans divers domaines des sciences, il est nécessaire de pouvoir visualiser des ensembles de données qui ont été mesurés ou simulés afin de les analyser, de les comprendre et d'en tirer des conclusions. On peut citer par exemple l'analyse de structure moléculaire en biochimie [KKF+17], de modèle 3D ou de simulation de phénomènes physiques, ou encore de l'imagerie médicale [PB13]. La visualisation scientifique est un domaine de recherche qui regroupe un ensemble de problèmatiques décrites par Chris Johnson en 2004 [Joh04] qui sont encore valables aujourd'hui.

Il est très important de pouvoir interagir avec les données que l'on visualise. Ainsi, la visualisation scientifique peut être vue comme une succession d'observation et d'interaction proposant à l'expert une observation active dans une boucle perception/action. Cependant, la visualisation de données implique de devoir calculer une image ou un ensemble d'images par ordinateur. Le calcul de ces images de synthèse varie énormément selon le type des données et les techniques de visualisation. Il peut être résumé au calcul de la projection d'un modèle ou d'une scène 3D dans l'espace du support de visualisation, classiquement,

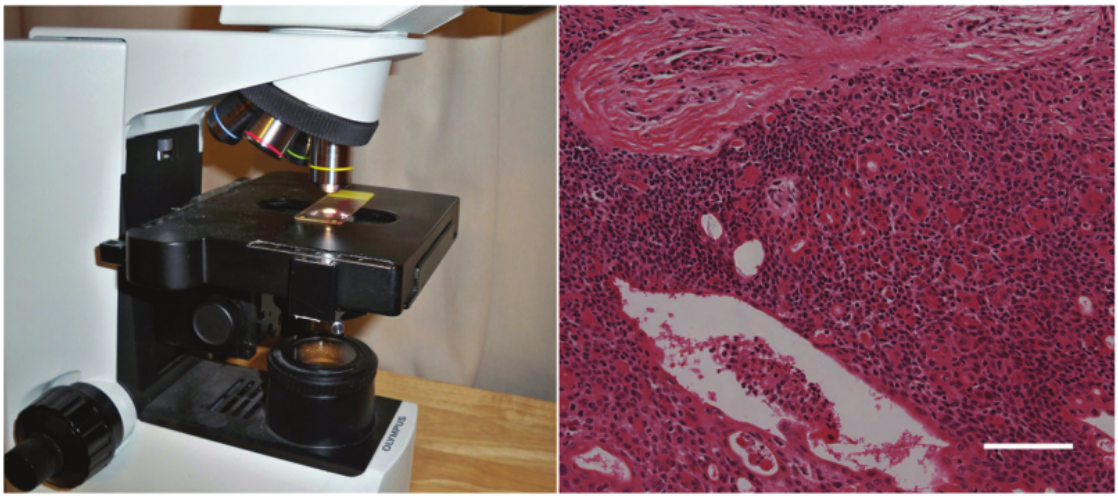


FIGURE 2.2 – **Visualisation de piles d'images 2D.** Microscope virtuel pour la visualisation de coupes biologiques. (source : [JST+10]).

un écran 2D. Les capacités d'interactions pendant la visualisation sont directement liées au coût de calcul de ces images.

Il existe une grande variété de méthodes et de représentations utilisées en visualisation scientifique. Le choix d'une de ces méthodes dépend de la nature du modèle ou du phénomène à visualiser d'une part (la nature des données), et de la représentation visuelle que l'on souhaite obtenir pour l'analyse de ces données d'autre part. Nous pouvons retrouver des méthodes de visualisation par nuage de points [HP07], en rendu surfacique ou encore en rendu volumique (voir Figure 2.1). Nous nous concentrons dans cette section, sur les techniques de visualisation 2D et 3D de données volumiques et plus particulièrement à celles qui sont adaptées aux données biomédicales. Le rendu volumique est une approche couramment utilisée dans le domaine de la visualisation en médecine ou en biologie de part la représentation native en grille 3D des données engendrées par des techniques d'acquisitions usuelles.

2.2.1 Microscopie virtuelle 2D

Un jeu de données volumiques en imagerie biomédicale est assimilé à un ensemble d'images 2D générées par un appareil d'aquisition de type scanner CT, IRM ou encore par microscopie électronique. Les volumes associés sont alors composés de plusieurs coupes qui forment une pile d'images, construisant un champ scalaire ou vectoriel 3D. Ces données sont bien représentées de manière volumique et chaque image de la pile est considérée comme une coupe (biologique) qui peut être visualisée seule, tel qu'il peut être fait avec un microscope classique pour une coupe physique. Dans le cadre de la visualisation de ces coupes numériques, on peut parler de microscopie virtuelle. Le but est de simuler le principe d'un microscope [CBC+03] de manière à observer et à naviguer dans ces coupes virtuelles. Nous renvoyons le lecteur vers un état de l'art [RGI07] permettant de faire le point sur les différentes méthodes de visualiseur en microscopie virtuelle.

Les images manipulées sont en général de grande dimension [ALB+13] et l'amélioration

dans la précision des appareils permet de générer des coupes de plus en plus précises et ainsi, d'observer des détails intéressants pour les pathologistes. Cependant, cette précision engendre des données de plus en plus lourdes en mémoire. C'est pourquoi il est nécessaire de développer des méthodes permettant de traiter cela. Les problématiques sont similaires à celles de la navigation dans des images haute-résolution, dites gigapixel, de tout domaine confondu [KUDC07].

On retrouve plusieurs travaux de recherche autour de la visualisation de coupes virtuelles de grandes dimensions [JST+10, WHH15, MBT+16], ainsi que des logiciels commerciaux. On retrouve également des outils disponibles sur internet pour visualiser ces images haute-résolution [Ope, Dee]. La technique de base qui est très souvent employée pour gérer ces grandes images est une représentation pyramidale multi-résolution, tuilée (voir Figure 2.3). Le principe est de représenter une coupe à différents niveaux de détails et de construire ainsi une pyramide multi-résolution dont chaque niveau est une image représentant la coupe à un certain niveau de résolution. Chaque niveau de la pyramide est ensuite découpé en un ensemble de tuiles régulières 2D qui décomposent l'image en un ensemble de sous-parties. Le niveau de détail est ensuite adapté à la distance de l'objet visualisé et seules les tuiles visibles sont chargées en mémoire. Plusieurs approches se sont concentrées sur des techniques efficaces de compression des données [CIAR13].

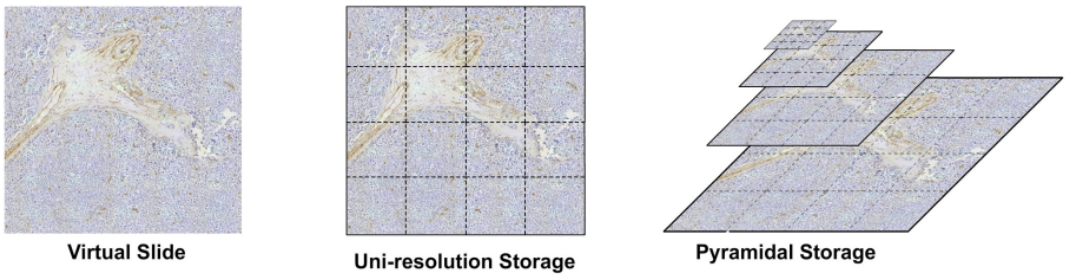
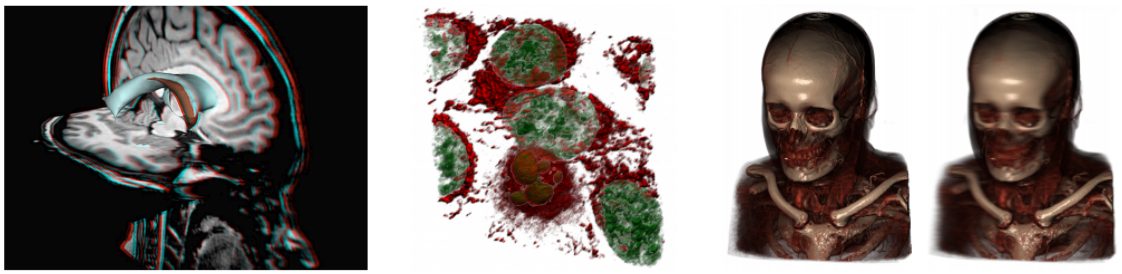


FIGURE 2.3 – **Pyramide multi-résolution tuilée.** Représentation de coupe virtuelle de grande dimension. (source : [RGI07]).

Les approches que l'on peut trouver dans la littérature en microscopie virtuelle se concentrent sur la visualisation interactive de coupes haute résolution individuelles. L'aspect de la navigation interactive entre plusieurs coupes dans une pile d'images est rarement abordé [JST+10]. Pourtant, les objets observés sont souvent en 3D et les données générées sont alors représentées sous forme de piles d'images pouvant s'apparenter à un volume régulier. En 2013, Hortsh [Hor13] reporte un inconvénient dans l'utilisation d'un tel système de visualisation pour l'étude de lames virtuelles. En effet, il rapporte que les utilisateurs de ces outils étaient frustrés de n'avoir qu'un seul plan de focalisation et de perdre la perception tridimensionnelle.

2.2.2 Stéréoscopie et multiscopie

En visualisation scientifique, la stéréoscopie est utilisée pour améliorer la perception 3D afin d'avoir une impression de profondeur. Le principe est de calculer plusieurs points de vues de projection, de manière à simuler un effet de relief. Cela permet d'enrichir la visualisation afin de mieux appréhender les relations spatiales d'un jeu de données. Ce type de visualisation est directement lié aux dispositifs d'affichage (écrans) et éventuellement au



(a) (b) (c)

FIGURE 2.4 – **Exemple de visualisation stéréoscopique** (a) Anaglyphe , (b) Stéréoscopie, (c) Multi-vues (source : (a) [GDHRR+16]), (b, c) illustrations issues de la plateforme de visualisation MINT du CReSTIC.

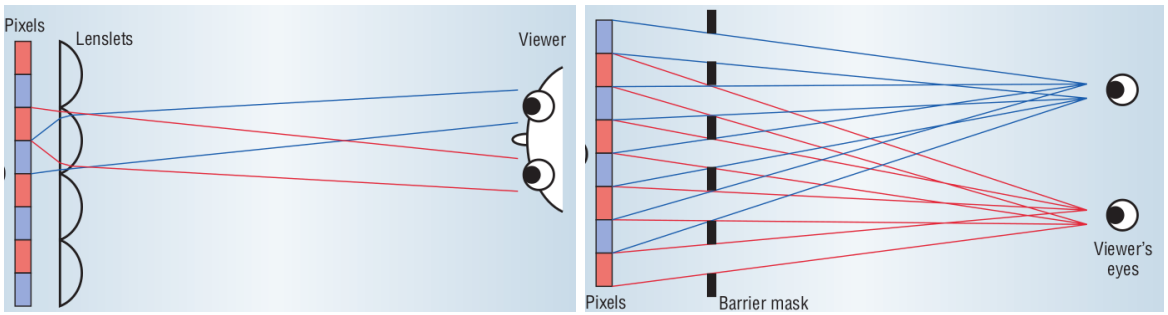
matériel complémenataire (lunettes) qui permettent la différenciation des images proposées à chaque œil. Il existe plusieurs types d'écrans permettant un affichage multi-vue, certains nécessitant une paire de lunette et d'autres, dits auto-stéréoscopiques, qui permettent un affichage relief sans accessoire. Nous présentons dans cette section les différentes méthodes de projection stéréoscopique. Dans tous les cas, leurs principes reposent sur la vision humaine binoculaire qui prend en compte la vue venant de nos deux yeux séparés par une certaine distance appelée distance inter-occulaire. Une image stéréoscopique est généralement composée de deux points de vue (voir Figure 2.4b). Lorsque l'on compose une image avec un nombre plus important de points de vue, on parle alors de multiscopie (voir Figure 2.4c).

La conception d'une image de synthèse stéréoscopique nécessite de construire un modèle multi-caméra adapté, permettant de calculer tous les points de vue qui entrent en jeu. On retrouve plusieurs travaux qui introduisent ces concepts dans la littérature [DF93, LLR13]. L'implémentation d'un modèle de caméra stéréoscopique peut être difficile à paramétrer et les images générées peuvent alors produire des erreurs génant l'observateur dans sa perception de profondeur et de mouvement. Jones *et al.* [JLHE01] étudient les différents paramètres physiques qui entrent en jeu dans la conception d'une image stéréoscopique correcte et proposent une méthode qui permet une expérience de visualisation confortable pour l'utilisateur. Ils notent en particulier que la disparité doit être limitée pour être en accord avec le système de vision humaine. La convergence des yeux et l'accomodation sont liées et il faut prendre en compte cette relation pour l'affichage sur des écrans stéréoscopiques.

La composition d'une image de synthèse multi-vue générée par les méthodes précédement décrites, peut alors se faire de plusieurs manières. L'anaglyphe est la technique de visualisation stéréoscopique la plus anciennement utilisée. Il s'agit d'utiliser des lunettes anaglyphes, composées de deux filtres, un pour chaque œil, qui associent à chacun une couleur complémentaire différente (en pratique, parmi le rouge, le vert et le bleu). L'image à afficher est alors une superposition de deux vues filtrées selon les deux couleurs correspondantes aux dispositifs permettant de la visualiser (voir Figure 2.4a). Chaque filtre exclut la vue correspondante dans la paire stéréoscopique de l'image et donne ainsi la séparation entre une image visible par l'œil gauche et une image visible par l'œil droit [DF93]. Il existe également des méthodes de visualisation d'image "side-by-side". Dans ce cas, l'image finale est composée des deux vues l'une à coté de l'autre et non pas superposées (voir Figure 2.4b).

Auto-stéréoscopie

Les dispositifs auto-stéréoscopiques permettent un affichage 3D en relief, à partir de N vues distinctes, ne nécessitant pas le port de lunettes. Il existe deux types de technologies d'écrans qui le permettent *1)* les écrans à réseau lenticulaire et *2)* les écrans à barrière de parallaxe (voir figure 2.5). Dans les deux cas, le principe est de contrôler la lumière qui sort de l'écran de manière à ce que l'observateur ne voit qu'une seule colonne de pixels sur N pour n'importe quel angle de vue. La première catégorie citée assure cela en se reposant sur un réseau de lentilles cylindriques permettant de diriger la lumière, alors que la deuxième catégorie utilise une barrière servant de masque pour cacher une partie des pixels de l'écran. Ces deux principes sont expliqués dans les travaux de Dodgson [Dod02, Dod05] puis par Holliman *et al.* [HDFP11] en 2011.



(a) (b)

FIGURE 2.5 – **Ecrans auto-stéréoscopique à reseau lenticulaire et à barrière de parallaxe.** (a)Reseau lenticulaire , (b)Barrière de parallaxe (source : [Dod05]).

Il est important de noter qu'il est nécessaire de calculer une image pour chacun des N points de vue qui compose l'écran. On obtient alors une image multi-vue dont le temps de calcul est multiplié par N par rapport à une image 2D classique.

2.2.3 Rendu volumique direct

Après avoir évoqué la visualisation de coupes 2D dans la section 2.2.1, nous allons maintenant aborder l'approche de rendu volumique direct (noté DVR en anglais pour Direct Volume Rendering). Cette technique est opposée au rendu volumique indirect (noté IVR pour Indirect Volume Rendering) qui passe par une étape intermédiaire, consistant à extraire une géométrie à partir de la représentation volumique d'un jeu de données de manière à en visualiser des isosurfaces [LC87].

Le DVR considère le volume de données comme un milieu participatif dans lequel un ensemble de particules peuvent émettre, diffuser ou absorber de la lumière. Il s'agit donc de calculer l'interaction de ces particules avec les propriétés physiques de la lumière qui traversent ce milieu.

Modèle Emission/Absorption

Plusieurs modèles optiques ont été définis relativement à la précision de rendu souhaité. Pour le rendu volumique dans le contexte de la visualisation scientifique, le modèle optique est généralement limité au modèle local d'émission et d'absorption sans prendre en compte

de sources d'illumination extérieures (indirectes) ou les effets de diffusion pour lesquels, la lumière émise se propage aux particules voisines. Le modèle classique d'émission/absorption est décrit par Williams & Max [WM92] en 1992 et reprit par Max [Max95] en 1995. Ce modèle optique décrit l'absorption de la lumière d'une part et la contribution sous forme de couleur d'autre part, par les particules rencontrées dans le milieu traversé par un ensemble de rayons provenant de l'arrière plan, et allant jusqu'à l'observateur.

Nous pouvons alors définir l'équation de rendu volumique, par l'intégration le long d'un rayon de lumière à partir d'un point de départ $s = s_0$ jusqu'à un point final $s = D$, dans laquelle k est le coefficient d'opacité locale et $q(s)$ est la densité d'émission (couleur) des données à la position s le long du rayon [EHMK+06] :

$$I(D) = I_0 \; e^{-\int_{s_0}^{D} k(t)dt} + \int_{s_0}^{D} q(s) \; e^{-\int_{s}^{D} k(t)dt} \, ds \tag{2.1}$$

Le résultat $I(D)$ correspond à l'intensité de lumière qui atteint l'observateur en sortant du volume après l'intégration dans le milieu. L'état I_0 décrit l'intensité lumineuse à la position initiale $s = s_0$ à l'arrière plan. L'équation 2.1 peut être décomposée en deux termes qui représentent respectivement, *1)* l'atténuation de la lumière en traversant le volume depuis l'arrière plan et *2)* l'intégration de l'émission de lumière en tout point, prenant en compte le facteur d'atténuation du milieu (la transparence).

La transparence peut être écrite de la manière suivante :

$$T(s_1, s_2) = e^{-\tau(s_1, s_2)} \tag{2.2}$$

avec

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} k(t)dt \tag{2.3}$$

le terme caractérisant la profondeur optique entre deux points s_1 et s_2, qui indique la longueur physique du chemin que la lumière parcourt avant d'être totalement absorbée par le milieu.

On peut ainsi simplifier l'équation 2.1 en l'écrivant avec la forme suivante :

$$I(D) = I_0 \; T(s_0, D) + \int_{s_0}^{D} q(s) \; T(s, D) \, ds \tag{2.4}$$

Discrétisation

Le modèle théorique continu précédemment décrit doit être évalué de manière numérique pour obtenir une solution approximée de l'intégrale. Généralement, on utilise une approximation par une somme de Riemann, découpant le domaine d'intégration de s_0 à $s_n = D$ en n intervalles égaux s_i (voir Figure 2.6). On obtient alors l'intensité lumineuse à la position s_i par l'équation :

$$I(s_i) = I(s_{i-1}) \; T(s_{i-1}, s_i) + \int_{s_{i-1}}^{s_i} q(s) \; T(s, s_i) \, ds \tag{2.5}$$

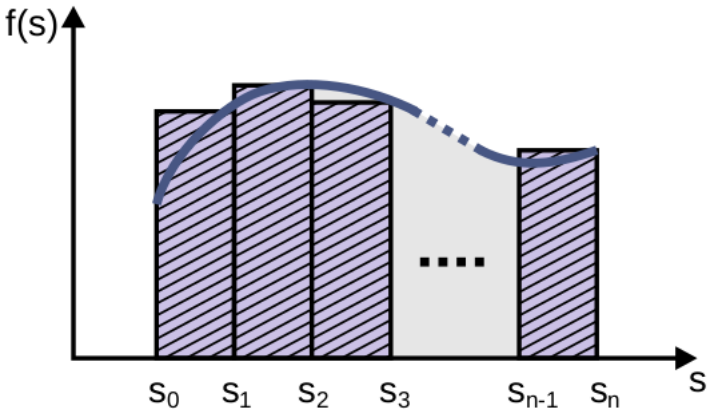


FIGURE 2.6 – **Approximation de l'intégrale de rendu volumique par la somme de Riemann.** (source : [EHMK+06]).

En évaluant cette fonction pour tous les points de l'intervalle, nous obtenons alors une version discrète de l'intégrale de rendu volumique. En notant respectivement T_i, la transparence et c_i, la contribution colorimétrique du i^{me} intervalle avec :

$$T_i = T(s_{i-1}, s_i), \qquad et \quad c_i = \int_{s_{i-1}}^{s_i} q(s)\ T(s, s_i)\, ds \tag{2.6}$$

On obtient alors :

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j, \qquad avec \quad c_0 = I(s_0) \tag{2.7}$$

Composition

L'équation 2.7 peut être évaluée itérativement par composition de la couleur C (généralement dans l'espace RGB) et de l'opacité A (alpha) par *alpha blending*. Il existe deux sens de composition : d'arrière en avant (*back-to-front*) et d'avant en arrière (*front-to-back*). La deuxième méthode est généralement utilisée car elle permet d'optimiser les calculs en arrêtant l'itération le long d'un rayon lorsqu'un certain seuil d'opacité est atteint (early ray termination). On utilise alors l'opérateur *OVER* défini par Porter & Duff en 1984 [PD84] :

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \tag{2.8}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \tag{2.9}$$

Au cours des itérations, on accumule la couleur C' et l'opacité A' à une position i en prenant en compte l'accumulation C'_{i-1} et A'_{i-1} à l'itération précédente et les valeurs C_i et A_i à la position courante. Dans ces équations, la couleur C est pré-multipliée par la valeur A correspondante dans le quadruplet $RGBA$ (opacity-weighted colors) [WMG98].

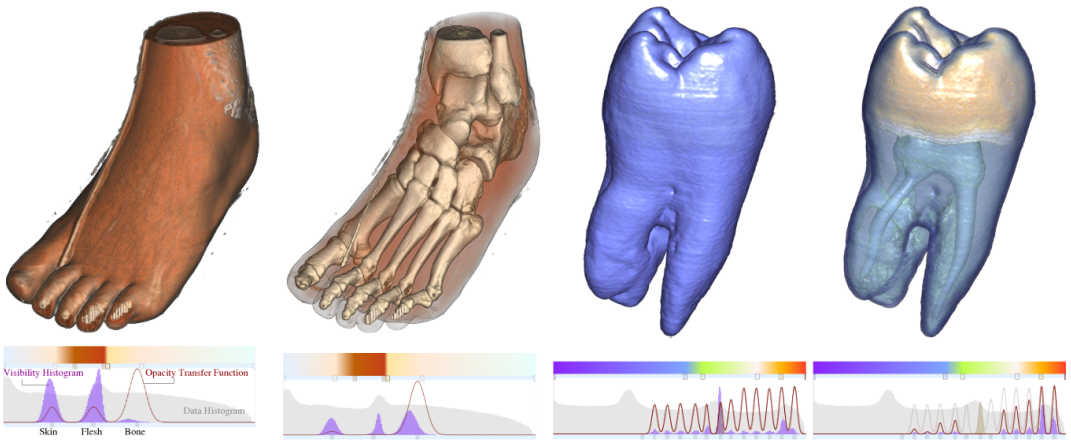


FIGURE 2.7 – **Fonction de transfert pour la classification en DVR.** Utilisation de deux fonctions de transferts 1D différentes, représentées en bas, avec le rendu correspondant pour le volume associé au dessus. (source : [CM09]).

Classification par Fonction de Transfert

La classification dans le rendu volumique direct permet d'appliquer les propriétés optiques aux échantillons provenant d'un jeu de données volumiques. On utilise pour cela une fonction de transfert qui permet de faire correspondre une intensité à un quadruplet $RGBA$, déterminant ainsi une couleur et une opacité pour chaque valeur. La fonction de transfert permet de définir à la fois ce que l'on souhaite rendre invisible dans un volume, mais également la manière de représenter ce qui, au contraire, est visible pour l'observateur (voir figure 2.7).

On retrouve beaucoup de travaux autour des fonctions de transfert pour le rendu volumique. En 2010, Arens et Domik [AD10] proposent une classification des fonctions de transfert en six catégories : 1D basée donnée, 2D basée gradient, ou encore, basée sur la courbure, sur la taille, sur la distance et enfin, basée texture. Drebin *et al.* [DCH88] présentent le concept de fonction de transfert comme une classification des matériaux avec une probabilité basée sur la valeur scalaire. La forme la plus commune est la fonction de transfert 1D. Cependant, cette forme simple n'est souvent pas adaptée pour l'imagerie médicale car les valeurs d'intensité des tissus peuvent se recouvrir fortement, comme décrit par Lundström *et al.* [LLY06]. Certaines approches utilisent des fonctions de transfert multi-dimensionnelles, comme Kniss *et al.* [KKH02] qui présentent une fonction 2D basée sur le mapping de la valeur et de l'intensité du gradient. Le design de fonction de transfert est généralement manuel, complexe et coûteux en temps. Quelques approches proposent une génération automatique. Kindlmann et Durkin [KD98] proposent une méthode semi-automatique pour générer des fonctions de transfert adaptées à la visualisation, qui permettent de faire ressortir les éléments caractéristiques, en prenant en compte l'histogramme 3D du volume. Correa et Ma [CM09, CM11] proposent de contrôler la visibilité directement, en calculant des histogrammes de visibilité représentant la visibilité des valeurs d'échantillons pour un point de vue donné. Nous pouvons nous référer à un état de l'art récent et complet, proposé par Ljung *et al.* [LKG+16] en matière de fonction de transfert. Certains travaux proposent des alternatives aux fonctions de transfert traditionnelles, comme Quan *et al.* [QCJJ18] qui introduisent un système intelligent d'apprentissage par convolution 3D.

2.3 Rendu volumique temps-réel sur GPU

Les premières approches de rendu volumique ont été implémentées sur les processeurs centraux (CPU) [Bli82, KVH84]. Les avancées technologiques liées aux développements des cartes accéleratrices de type GPU (Graphics processing Unit) ont permis d'améliorer grandement les performances en rendant ainsi possible une visualisation interactive en temps réel de jeux de données volumiques [KW03]. Il existe plusieurs approches de rendu volumique sur GPU et nous allons discuter dans cette section des plus populaires. Nous commencons par présenter les méthodes basées textures par calcul de géométrie de proximité ("proxy geometry" en anglais) à partir des données scalaires 3D. Nous évoquerons ensuite une méthode plus moderne, basée image, très utilisée de nos jours, opposée aux méthodes précédentes, basées objet. Tandis que les premières utilisent le pipeline fixe du matériel permettant la rastérisation de polygones, la deuxième est rendue possible depuis l'évolution, ces deux dernières décennies, de ce même pipeline dont une partie est maintenant rendue assez programmable, par l'implémentation de shaders, pour contourner son utilisation de base. Nous nous intéresserons ensuite aux méthodes d'accélération du rendu dans des environnements multi-GPUs.

2.3.1 Méthodes basées texture

Le principe de ces méthodes est d'utiliser les textures implémentées par les cartes graphiques. Ce sont des tableaux à une, deux ou trois dimensions qui offrent des avantages d'accès aux données, en particulier par une interpolation matérielle. Les textures des GPU possèdent leur propre cache et il est possible d'accéder à une coordonnée dans une texture de manière normalisée.

Géométrie de proximité

Ces approches utilisent le pipeline de rastérisation matérielle tel qu'il a été conçu à la base, c'est à dire en manipulant des polygones. Ces approches basées objet doivent donc transformer préalablement les données volumiques représentées sous forme de grille régulière 3D (le plus communément) en les décomposant en primitives géométriques, puis en effectuant le rendu de ces primitives sur le GPU. Ces primitives sont stockées en un ensemble de textures 2D ou alors dans une texture 3D.

Textures 2D

Une première approche consiste à stocker le volume comme une série de textures 2D qui composent des coupes dans le volume 3D. La géométrie de proximité utilisée est alors un ensemble de polygones alignés dans le volume ou, autrement dit, alignés selon les axes (object-aligned ou axis-aligned slices) (voir figure 2.8). Il est nécessaire de stocker trois piles de coupes pour chacun des trois axes du volume. Cette approche pose problème car on ne peut pas utiliser d'interpolation trilinéaire pour échantillonner les données dans le volume 3D dû à l'utilisation de textures 2D. De plus, elle peut générer des artefacts visuels selon l'orientation de la vue, avec des changements abruptes lorsque la pile de coupes 2D est orientée à 45 degrés.

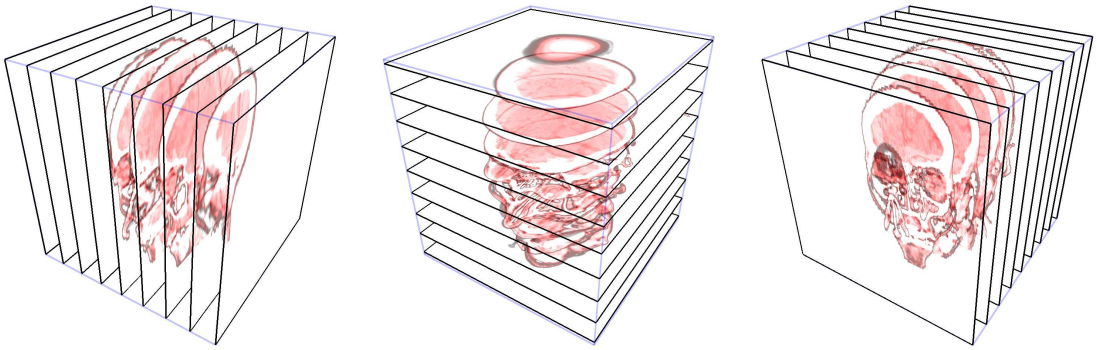


FIGURE 2.8 – **Rendu volumique basé sur des textures 2D.** Un ensemble de textures 2D est utilisé pour stocker une pile de polygones orientés dans l'objet. (source : [BBJL05]).

Textures 3D

Une alternative à la méthode précédente, est d'utiliser une seule texture 3D. Dans ce cas, les coupes sont orientées perpendiculairement au plan de la vue (voir figure 2.9). Cette approche permet une interpolation trilinéaire des données et produit donc une meilleure qualité de rendu. Cullip et Neumann [CN93] en 1993 et Cabral *et al.* [CCF94] en 1994 sont les premiers à utiliser une texture 3D pour stocker une géométrie de proximité pour le rendu volumique. Ils utilisent également un mélange selon le canal alpha ("alpha-blending" en anglais) pour la composition. Westerman et Ertl [WE98] ont significativement étendu cette approche en introduisant un algorithme multi-passes pour afficher des isosurfaces.

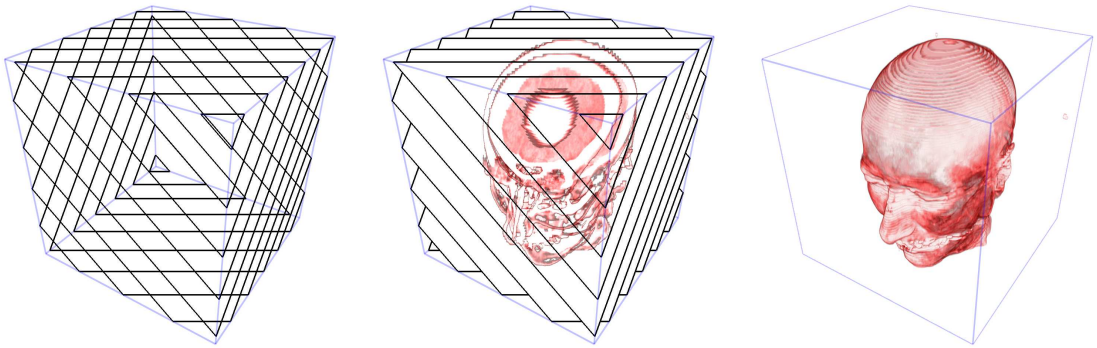


FIGURE 2.9 – **Rendu volumique basé sur l'utilisation d'une texture 3D.** Une texture 3D permet de stocker la géométrie de proximité, alignée dans le plan image. (source : [BBJL05]).

Ces méthodes basées texture ont été très utilisées dans la littérature. Elles ne requièrent pas de matériel avancé et permettent d'obtenir de bonnes performances, car elles utilisent le pipeline de rastérisation, qui est très efficace, et la bande passante entre la rastérisation et la mémoire texture, qui offre un débit très rapide. En revanche, elles génèrent des artefacts visuels et offrent peu de flexibilité. De plus, elles sont peu adaptées à de gros volumes de données car leur complexité dépend directement du jeu de données en entrée (approches basées objet).

2.3.2 Lancer de rayon

Les méthodes de lancer de rayon sont des approches basées image, contrairement aux méthodes basées objet vues précédemment. Le lancer de rayon volumique est l'approche la plus directe et intuitive pour évaluer l'intégrale du rendu volumique présentée par l'équation 2.1. On retrouve plusieurs appellations dans la littérature pour faire référence à cette approche. On utilise les termes anglais de "volume ray-casting" qui fait référence au lancer d'un rayon primaire adapté au rendu volumique, ou encore de "ray-marching" afin de caractériser une marche le long d'un rayon à travers un volume. Cette méthode est très utilisée aujourd'hui sur GPU car elle se prête très bien à leur architecture massivement parallèle et elle permet d'obtenir une bonne qualité de rendu en temps-réel. Le principe est de lancer virtuellement un rayon depuis l'observateur (une caméra virtuelle) à travers un volume de données dans une scène 3D pour chaque pixel qui compose l'image. Le volume est échantillonné suivant des positions discrètes le long de chaque rayon, et les valeurs scalaires correspondantes sont intégrées à l'intérieur du volume après l'application des propriétés optiques, puis composées de manière à calculer la couleur du pixel correspondant (voir figure 2.10). Elle a été introduite puis améliorée par Levoy [Lev88, Lev90] utilisant une implémentation sur CPU. Plusieurs approches sur CPU ont ensuite été reprises par la suite [DH92].

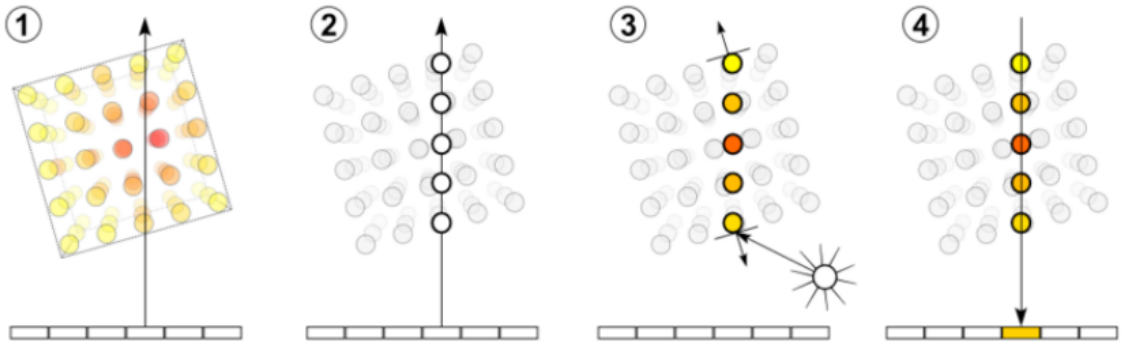


FIGURE 2.10 – **Lancer de rayon volumique.** La première étape consiste à lancer un rayon à travers la grille 3D. La second étape représente l'échantillonnage dans cette grille, le long du rayon. La troisième étape correspond à l'affectation des propriétés optiques aux échantillons. La quatrième étape montre le principe d'accumulation qui prend en compte la contribution de chaque échantillon le long d'un rayon.

Accéleration matérielle

L'adaptation de cet algorithme sur GPU a été introduit plus tardivement par Röttger *et al.* [RGW+03], suivi de Krüger et Westermann [KW03]. Tandis que ces approches reposent sur un rendu multi-passe, Stegmaier *et al.* [SSKE05] ou encore Scharsach [Sch05], proposent un lancer de rayon en une seule passe. Ces travaux, plus complets, se basent sur les améliorations des GPUs en terme de branchements dynamiques et de boucles.

Le lancer de rayon volumique peut facilement se paralléliser car il utilise des rayons primaires qui sont indépendants les uns des autres. Il est alors possible d'utiliser l'environnement multi-threads des GPU en attribuant un rayon (un pixel de l'image) par thread (voir figure 2.11). De plus, les données scalaires peuvent être stockées en mémoire texture 3D afin de profiter de leurs avantages matériels et d'utiliser les techniques de "texture map-

ping". Cette configuration permet d'implémenter directement le comportement des rayons avec des programmes de "fragment shaders" en GLSL par exemple [Khra, RLKG+09] ou avec les langages de programmation GPGPU (General Purpose GPU), tels que CUDA [Nvi] ou OpenCL [Khrb]. Des illustrations de rendus obtenus par lancer de rayon volumique sur GPU sont montrés sur la figure 2.12.

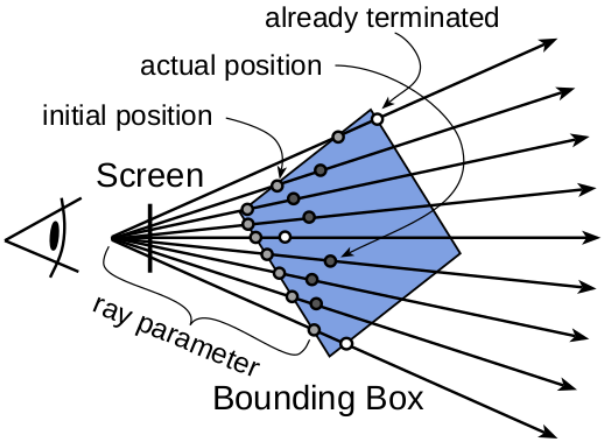


FIGURE 2.11 – **Lancer de rayon en parallèle.** Tous les rayons sont lancés simultanément sur le GPU avec un rayon par thread. (source : [RGW+03]).

Techniques d'optimisation

Les méthodes les plus modernes concernent les techniques de rendu en une seule passe et implémentant des méthodes d'accéleration de différents types. La technique de terminaison précoce des rayons ("early ray termination" en anglais) permet de stopper le parcours d'un rayon si l'accumulation de l'opacité le long de celui-ci a atteint un certain seuil. Au-delà de ce seuil, on considère que les données ne participent plus à l'intégration car elles sont occultées. Cela est rendu possible uniquement avec une traversée de rayon d'avant en arrière (front-to-back). Cette technique permet alors éventuellement de ne pas parcourir l'ensemble d'un rayon et ainsi d'améliorer les performances en temps de calcul [SSKE05, Sch05]. L'échantillonnage adaptatif, introduit par Röttger *et al.* [RGW+03] permet d'adapter le pas d'échantillonnage le long d'un rayon selon les caractéristiques des régions du volume qui sont traversées. Il est alors possible de réduire ce pas aux frontières de différentes régions et de l'augmenter pour de grandes régions uniformes. Ceci peut être fait indépendamment pour chaque rayon. Cette méthode repose sur le principe de pré-intégration, utilisé par Engel *et al.* [EKE01] dans un contexte de rendu basé texture. Il existe également plusieurs méthodes de saut d'espace vide afin de ne pas tenir compte de certaines régions complètement vides au sein d'un volume. Ces méthodes sont appellées "empty-space skipping" ou encore "empty-space leaping". On peut citer les travaux de Klein *et al.* [KSSE05] pour s'y référer.

Techniques d'illuminations (shading)

Bien que cette thèse n'aborde pas particulièrement les méthodes d'éclairage en rendu volumique, il est important de citer quelques travaux qui s'y rapportent. En effet, ces travaux s'intègrent à la suite des études précédement décrites dans le domaine du lancer de rayon volumique sur GPU et permettent d'améliorer la qualité du rendu. Hadwiger *et al.* [HKSB06] proposent une technique d'ombrage selon le point de vue de la lumière. Les

travaux de Hernell *et al.* [HLY07] portent sur le calcul d'occlusion ambiante en lançant des rayons secondaires dans une sphère pour chaque voxel. On peut aussi retrouver une comparaison de différentes méthodes d'ombrage faites par Ropinski *et al.* [RKH].

(a) (b) (c)

FIGURE 2.12 – **Exemple de rendu par lancer de rayon volumique sur GPU** (source : [HLSR08], [KSSE05]).

Les travaux plus récents dans le contexte du lancer de rayon volumique sur GPU se sont intéressés à la gestion de jeux de données de grande dimension, mais cet aspect sera abordé plus en détail dans la section 2.4.

2.3.3 Rendu distribué

La quantité de calcul nécessaire au processus de rendu est conséquente, et cela peut devenir un facteur limitant pour des jeux de données complexes ou de grande dimension. Pour garantir un rendu de bonne qualité tout en fournissant une interactivité avec un rafraichissement rapide des images, il est parfois nécessaire de se tourner vers des méthodes de parallélisation et de distribution des calculs. Il existe aujourd'hui une grande variété d'architectures parallèles et d'environnements distribués. Nous pouvons classer ces architectures en deux catégories, selon qu'elles possèdent un seul noeud de calcul ou alors un ensemble de noeuds interconnectés. Dans les deux cas, un noeud de calcul peut être composé de plusieurs CPU, de plusieurs GPU ou d'un environnement hybride qui regroupe les deux. La première catégorie peut se retrouver dans un PC standard ou alors sur un serveur de calculs dédié. La deuxième catégorie fait référence aux clusters de calculs ou supercalculateurs. Par la suite, nous utiliserons le terme d'"unité de calcul" au sens large pour faire référence aussi bien à un CPU, un GPU, ou encore à un noeud de calcul d'un cluster. Les méthodes de rendu haute-performance utilisent les architectures distribuées précédemment décrites, en répartissant la charge de calcul d'un jeu de données sur plusieurs unités de calcul.

Nous allons d'abord présenter dans cette section, les méthodes de rendu distribué générales. Ces méthodes ne sont pas spécifiques à un type de rendu en particulier, ni à une architecture précise. Nous nous intéressons ensuite aux approches plus spécifiques de lancer de rayon volumique sur des clusters multi-GPU.

Méthodes de rendu distribué

En 1994, Molnar *et al.* [MCEF94] proposent une classification des méthodes de rendu parallèle. Celle-ci comprend trois catégories, qui sont appellées respectivement *1)* "sort-first", *2)* "sort-middle" et *3)* "sort-last". La différence entre ces catégories se rapportant à l'étape à laquelle les données sont distribuées dans le pipeline de rendu. Dans le cadre du rendu volumique, les deux approches possibles sont le rendu "sort-first" et le rendu "sort-last".

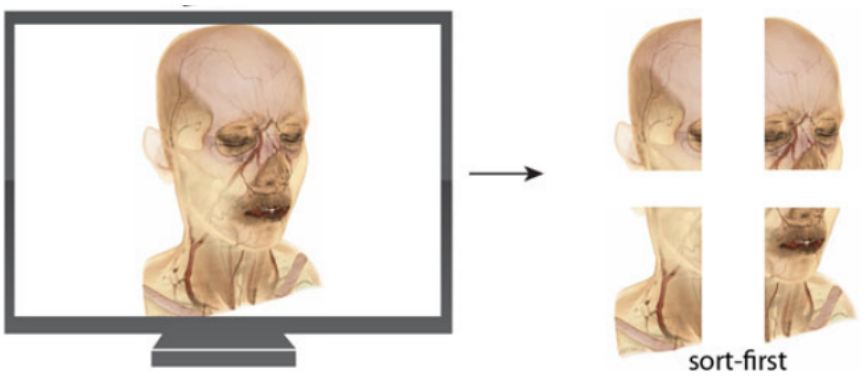


FIGURE 2.13 – **Méthode de rendu distribué "Sort-first".** La répartition des calculs sur les différentes unitées de rendu se fait selon une distribution du domaine dans l'espace image. (source : [BHP15]).

La première est une méthode de décomposition dans l'espace image (voir Figure 2.13). Chaque unité de calcul est en charge d'une partie de l'image à afficher. La répartition de la charge des calculs est alors directement liée au point de vue et peut impliquer une forte redistribution des données sur les différents noeuds lors d'un changement de caméra.

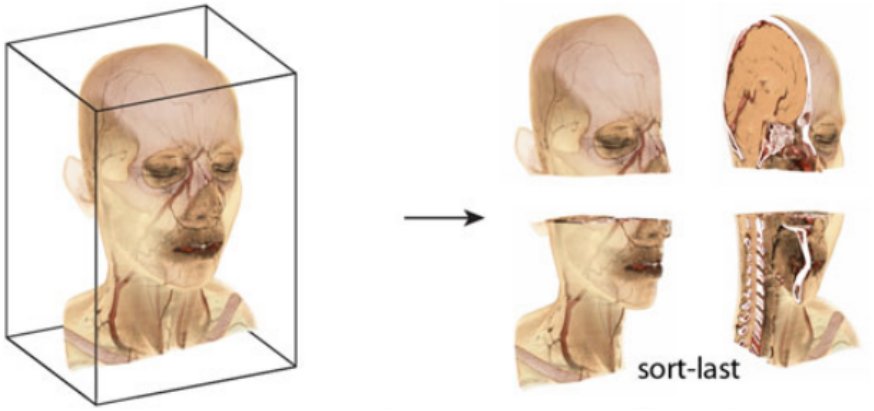


FIGURE 2.14 – **Méthode de rendu distribué "Sort-last".** La répartition des calculs sur les différentes unitées de rendu se fait selon une distribution du domaine dans l'espace de l'objet. (source : [BHP15]).

La deuxième approche consiste à décomposer les données dans l'espace objet (voir Figure 2.14). C'est alors le volume qui est découpé et réparti sur les différentes unités de calculs. Chaque unité de calcul est alors chargée d'effectuer le rendu sur le sous-domaine des données qu'elle gère. Dans ce cas, il est nécessaire de réaliser une étape suplémentaire de recomposition des pixels de l'image finale à afficher à l'écran. Cette étape, dites de "com-

positing" a fait l'ojet de plusieurs études [MPHK94, YWM08, PGR+09]. Les approches sort-last sont plus intéressantes pour de gros volumes de données car elles permettent une mise à l'échelle d'un point de vue de la gestion de la mémoire. Il est important de prendre en compte une bonne stratégie de répartition des calculs sur les différentes unités. Cette problématique de "load-balancing" a été adressée par plusieurs travaux dans la littérature, dont notamment ceux de Marchesin *et al.*[MMD06].

Le lecteur peut se référer à des travaux reprenant l'ensemble des méthodes et problématiques en rendu parallèle et en rendu distribué [BSS00, ZSJ+05].

Certaines approches se basent sur du rendu volumique parallèle sur CPU. Howison *et al.* [HBC10] utilisent dans leur travaux, une méthode de rendu hybride (méthode basée objet combinée à un lancer de rayon) sur des systèmes massivement multi-coeur CPU. Cette approche est ensuite étendue pour une étude comparative d'un lancer de rayon volumique parallèle avec une implémentation sur un système multi-coeur CPU et une implémentation GPU [BCH12, HBC12]. Ils traitent différents type de mémoire et de paradigme de parallélisation tel que MPI, OpenMP, Pthread et CUDA. Nous allons maintenant nous intéresser à des approches qui utilisent des environnements multi-GPU.

Lancer de rayon volumique multi-GPUs

Marchesin *et al.* [MMD08] présentent une étude des performances d'une approche sort-last sur un seul noeud multi-GPU, comparé à un cluster multi-noeud composés d'un seul GPU chacun. Alors que beaucoup d'approches se sont concentrées sur du rendu sort-last, Moloney *et al.* [MAWM11] montrent qu'il est intéressant d'utiliser une approche sort-first qui se montre efficace sur un cluster de GPU.

Tous les travaux cités ici sont des approches qui manipulent des jeux de données tenant en entier en mémoire GPU. Les techniques de rendu distribué sont en particulier utilisées pour du rendu de grande masse de données volumiques. Cela nous permet de faire le lien avec la section suivante qui va s'intéresser en détail aux techniques out-of-core. Nous retrouverons alors d'autres travaux de recherche plus récents, en rendu distribué out-of-core sur GPU.

2.4 Gestion de données out-of-core pour le rendu volumique

Dans cette section, nous allons aborder plus spécifiquement l'aspect de la visualisation volumique de grande masse de données. L'évolution des appareils d'acquisition en imagerie biomédicale par exemple, rend aujourd'hui possible l'acquisition de données très détaillées qui engendrent des volumes pouvant atteindre plusieurs téraoctets. La capacité mémoire des GPUs, qui augmente également avec le temps, ne suit pas pour autant la même courbe d'évolution. Les GPUs modernes, spécifiques à une utilisation professionnelle et scientifique, peuvent atteindre aujourd'hui jusqu'à 24 Go pour les plus haut de gamme. Les GPUs destinés au grand public quant à eux, possèdent généralement entre 2 et 8 Go de mémoire interne. Les techniques de visualisation ont donc dû faire face à ce phénomène afin de pouvoir continuer à manipuler de telles quantités de données. Cela a donné lieu à l'élaboration de méthodes dites "out-of-core", c'est à dire, dont l'unité de calcul est dissociée de l'unité de stockage des données. Nous allons nous concentrer ici sur une étude des techniques utilisées par les approches modernes de rendu volumique, adaptées à de grandes masses de données. L'ensemble des concepts évoqués dans cette section peuvent être retrouvés dans un état de l'art complet, proposé par Beyer *et al.* [BHP15] en 2015.

2.4.1 Représentation et stockage des données

Bricking

Une des approches de base utilisée dans le rendu out-of-core de larges volumes de données, est appellée "briking" ou encore "blocking". Cette méthode consiste à subdiviser un jeu de données volumiques trop gros pour tenir en mémoire, en un ensemble de petites briques de voxels ("brick" ou "block" en anglais dans la littérature). Ce principe émane du fait qu'un volume entier est trop important pour être manipulé comme une seule entité et qu'il est alors nécessaire de le découper afin de répartir la complexité du problème (voir Figure 2.15). Les briques, généralement d'une taille moyenne de l'ordre de 32^3 voxels, sont alors manipulées indépendamment les unes des autres au fur et à mesure de la demande de l'étape de visualisation. Les approches qui utilisent cette méthode proposent un système de cache sur le GPU ("brick pool" en anglais) en mémoire texture, pour gérer ces briques pendant le rendu. Il est également utile de tenir compte de la visibilité des données pour déterminer les briques néces-

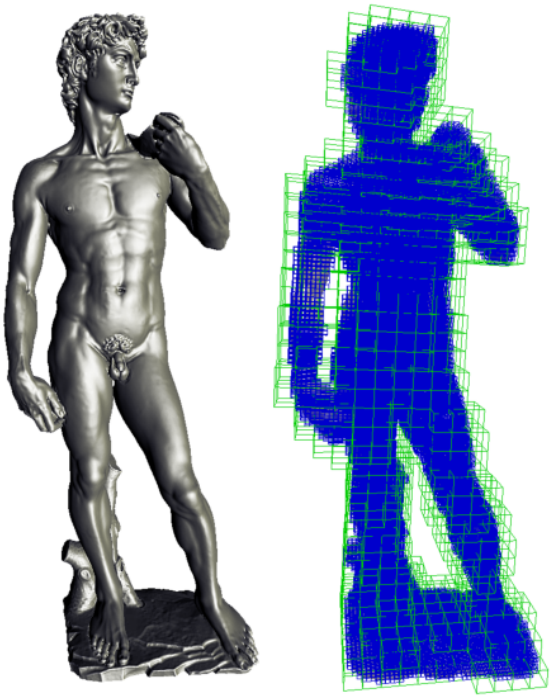


FIGURE 2.15 – **Bricking : méthode de subdivision d'un volume.** Un volume trop gros pour tenir en mémoire est alors découpé en un ensemble de petites briques contenant un petit nombre de voxels. (source : [HSS+05]).

saires (appelé "working set" en anglais) à chaque image. Certains traitements nécessitent des interactions entre les voxels et leur voisinage, il faut alors porter une attention particulière pour les voxels aux frontières des briques. Typiquement, l'interpolation texture tri-linéaire, dans l'échantillonnage pour le lancer de rayon volumique, requiert ce genre de traitements. Il est alors nécessaire de stocker pour chaque brique, les voxels qui se trouvent aux frontières des briques voisines (appellé "ghost voxels" [ILC10]). Nous pouvons retrouver une étude sur la taille optimale des briques dans les travaux de Fogal *et al.* [FSK13]. Les auteurs indiquent alors qu'il est plus intéressant de manipuler des briques de tailles importantes (128^3 - 256^3 voxels) pour optimiser le stockage et les transferts vers le GPU, mais à l'inverse, il est plus judicieux de traiter de petites briques (16^3 - 32^3 voxels) pour le rendu sur le GPU. Ils proposent alors un système de redécoupage dynamique des briques à la volée.

Scharsach [Sch05] et Hadwiger *et al.* [HSS+05] dans leurs travaux de 2005, font partie des premiers à utiliser cette stratégie de décomposition du domaine en découpant leurs données en briques pour du rendu d'isosurfaces. Ils utilisent ensuite un lancer de rayon en une seule passe sur GPU à partir des briques qu'ils stockent dans un cache. Beyer *et al.* [BHWB07] utilisent cette approche de bricking pour du rendu multi-volumes appliqué à la neurochirurgie. Rujiters et Vilanova [RV06] proposent une approche de bricking avec l'utilisation d'un octree par brique pour un lancer de rayon multi-passe.

Multi-resolution

Alors que les méthodes précédemment citées proposent une représentation classique qui ne prend en charge qu'un seul niveau de résolution, plusieurs travaux se sont interéssés au rendu multi-résolution. Une telle représentation inclue plusieurs niveaux de détails (appelés "LOD" dans la littérature pour "Level Of Detail" en anglais) pour un jeu de données. Il est alors possible d'adapter le niveau de détail à la résolution de l'écran et à la distance du volume visualisé par rapport à la caméra. En plus d'éviter des artefacts d'aliasing dû au sur-échantillonnage, cette approche permet de réduire la quantité de données à accéder et est donc très utile pour des jeux de données de grande taille. On retrouve plusieurs méthodes multi-résolutions dans la littérature.

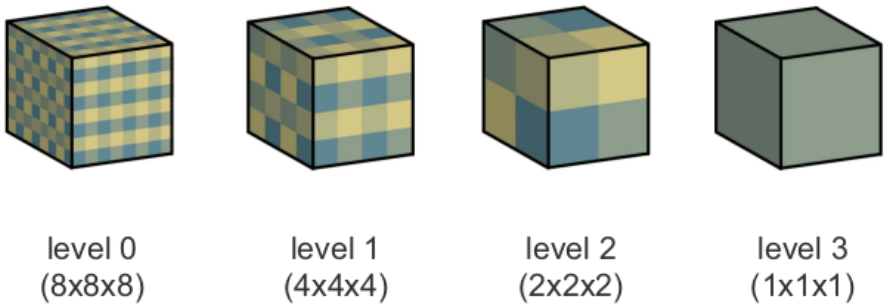
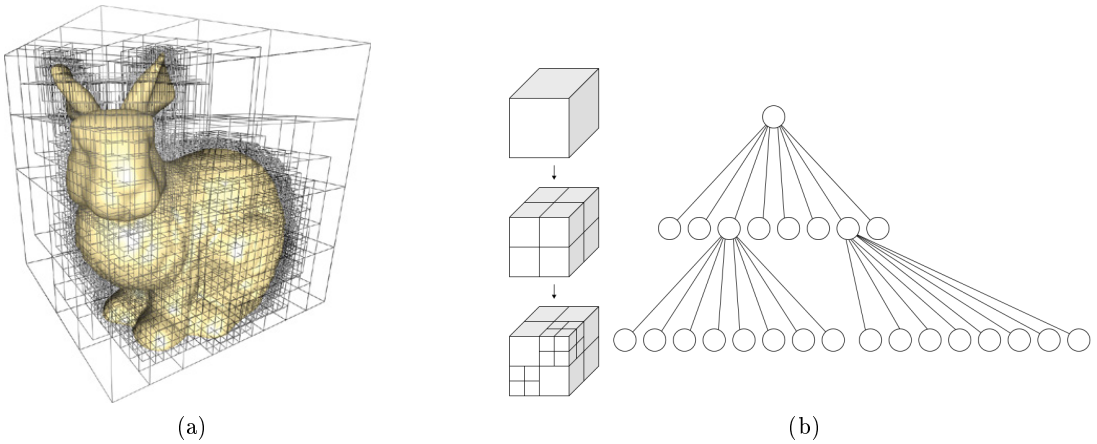


FIGURE 2.16 – **Multi-résolution : Mipmap 3D briqué.** Représentation multi-résolution dont chaque niveau représente le volume à un certain niveau de détail et découpé en un ensemble de petites briques. (source : [BHP14]).

L'utilisation de "mipmaps" a été introduite assez tôt pour du rendu multi-résolution basé textures [Wil83]. Cette structure est une pyramide qui comprend tous les niveaux de détails d'un jeu de données, du plus résolu au plus grossier. Elle peut être utilisée en

combinaison avec une approche de bricking et est alors constituée d'une grille hiérarchique dont chaque niveau de résolution est découpé en un ensemble de petites briques [JBH+09, HBJP12, FSK13] (voir figure 2.16). On obtient alors une pyramide qui est un mipmap 3D découpé en briques ou alors un mipmap 2D découpé en tuiles si l'on ne considère pas le volume 3D dans son ensemble (voir section 2.2.1). Hadwiger *et al.* [HBJP12] utilisent des ratios de sous-échantillonnage différents selon les trois axes du volume et pour chaque niveau de résolution. De cette manière, ils peuvent corriger une éventuelle anisotropie des données qui peut arriver fréquemment sur des jeux de données biomédicales.



(a) (b)

FIGURE 2.17 – **Multi-résolution : structure d'octree** (a) Un modèle 3D entouré de l'octree associé (b) Représentation d'un octree pour un volume multi-résolution (source : [GPU05, wik18]).

Beaucoup de travaux en rendu volumique direct utilisent une structure d'arbre comme représentation multi-résolution. On peut retrouver l'utilisation de BSP-trees ("Binary Space Partition" en anglais, pour "partition binaire de l'espace") [XYL+16] ou encore de kd-trees [SZM+14, ZSL18]. Cependant, la structure d'arbre la plus fréquente dans ce contexte est l'octree [Kno06] (voir figure 2.17). Un octree découpe l'espace en sous-région, de manière récursive, en insistant sur les zones plus détaillées. Ceux-ci sont plus utilisés que les kd-trees car ils sont plus adaptés à une grille régulière (données volumiques). Cette structure de données utilisée en rendu volumique direct est efficace et intuitive pour implémenter des méthodes de saut d'espace vide. Nous pouvons citer plusieurs travaux qui utilisent des structures d'arbres pour du rendu volumique multi-passes qui présument que le volume entier tient en mémoire (approches "in-core") [LHJ00, WWH+00, BNS01]. Les SVOs pour "Sparse Voxel Octree" ont beaucoup été utilisés, plus récemment, en informatique graphique pour l'animation ou l'industrie des jeux vidéo en particulier. Plusieurs travaux proposent des optimisations du stockage, comme Laine et Karras [LK11] en 2010, ou alors de la construction efficace [KSA13, BLD14].

Pendant le rendu, il est nécessaire de déterminer quel niveau de détail doit être utilisé. La sélection de LOD regroupe plusieurs familles d'approches : dépendantes de la vue, basées sur une estimation d'erreur, ou encore, basées sur la fonction de transfert. La première catégorie se base sur des mesures de distance avec l'observateur [LHJ00, GWGSs02, BD02, CNLE09]. Il s'agit de calculer la taille projetée d'un voxel dans l'espace écran et de changer de niveau de détails si un voxel occupe plus d'un pixel dans cet espace. La deuxième catégorie se base sur le calcul de l'erreur dûe à l'utilisation d'un niveau de détail moins résolu que le volume à sa résolution maximum. Cette erreur peut se calculer, par exemple,

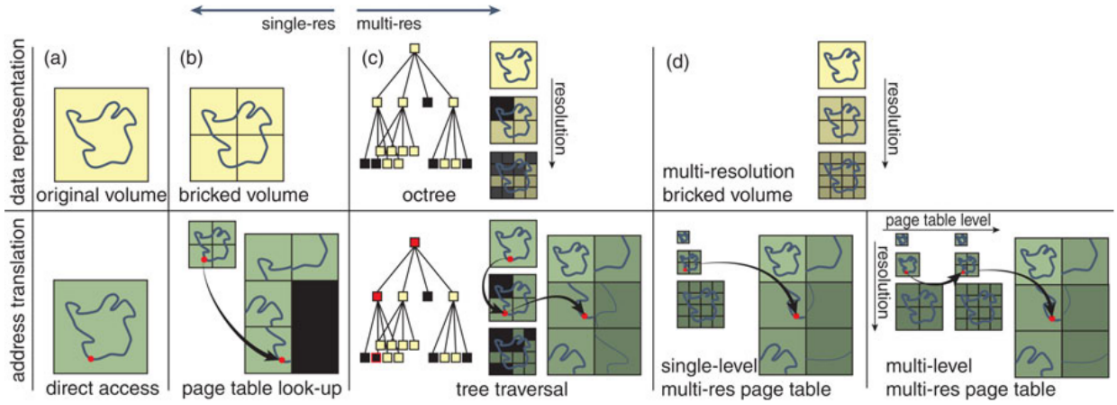


FIGURE 2.18 – **Méthodes de traduction d'adresse pour le rendu volumique out-of-core.** Nous retrouvons ici toutes les formes d'accès à un volume selon sa représentation. L'accès direct, sur un volume représenté de manière classique sert de référence aux méthodes d'adressage out-of-core pour des volumes briqués avec un ou plusieurs niveaux de résolution. (source : [BHP15]).

par une méthode de Root-Mean-Square-Error en pré-traitement. On sélectionne ensuite un niveau de détails en essayant de minimiser cette erreur [BNS01, GWGSs02]. Le problème avec ces méthodes qui se basent sur l'erreur liée aux données, tient au fait que les données échantillonnées dans le pipeline de rendu volumique direct sont ensuite associées à une valeur venant d'une fonction de transfert arbitraire. Il est donc plus pertinent de travailler dans le domaine de la fonction de transfert [LHJ03, GS04, GH06].

2.4.2 Structures d'adressage out-of-core

Les méthodes modernes de rendu volumique de grandes masses de données s'appuient sur les techniques de représentation et de stockage décrites dans la section précédente. Dans ce contexte, elles proposent des approches out-of-core de manière à manipuler efficacement des volumes qui dépassent la capacité mémoire physique disponible sur les GPUs. Les méthodes de lancer de rayon-volumique en une seule passe, sur des volumes de grande dimension, utilisent une représentation multi-résolution briquée, qui nécessite d'utiliser des approches d'adressage virtuel des données. Toutes les briques qui composent le volume n'ont pas besoin d'être présentes en mémoire GPU pour le rendu. Le principe est d'utiliser des méthodes de traduction d'adresse entre les adresses virtuelles et les adresses physiques de l'emplacement mémoire des briques. Dans ce cas, l'algorithme de rendu peut accéder à n'importe quel échantillon d'un grand volume, sans ce soucier de l'emplacement physique de celui-ci. Nous nous intéressons, dans cette section, aux différentes méthodes d'adressage out-of-core proposées pour ce contexte dans la littérature. La figure 2.18 reprend les différentes formes d'adressage d'un volume selon sa représentation et la structure utilisée.

Accès direct

De manière classique, et lorsque le volume tient en entier en mémoire GPU, celui-ci est stocké entièrement dans une seule texture 3D. La méthode d'accès à un échantillon dans ce volume se fait alors de manière directe à la position correspondante dans la texture. On ne parle pas dans ce cas de méthode out-of-core. La plupart des méthodes citées dans

les sections précédentes et se rapportant à une approche de lancer de rayon volumique sur GPU, utilisent cet accès texture direct lors de l'échantillonnage du volume.

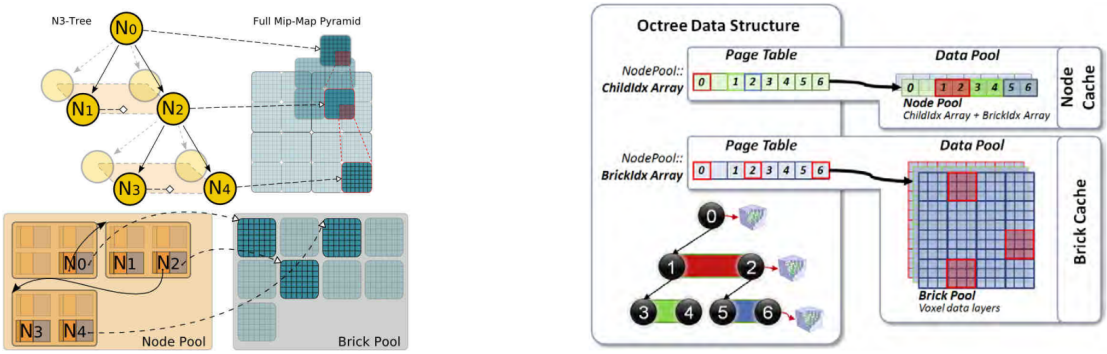
Traduction d'adresses

Nous allons maintenant présenter les différentes structures utilisées pour adresser un volume briqué pour du rendu volumique out-of-core, avec un ou plusieurs niveaux de résolution. Dans tous les cas, l'approche généralement employée est la suivante : les briques sont stockées dans un grand cache en mémoire texture 3D avec des accès normalisés permettant l'interpolation. Dans la littérature, on retrouve les termes anglais de "brick/data cache/pool" pour y faire référence. La navigation dans les données se fait alors à l'intérieur d'un volume virtuel sur le GPU, puisque celui-ci n'est pas entièrement physiquement présent dans sa mémoire. Une structure d'adressage est alors employée pour traduire l'adresse virtuelle utilisée par l'algorithme de rendu, en une adresse physique dans la texture 3D qui contient les briques. La structure qui permet cette traduction d'adresse est également stockée en mémoire texture mais avec des accès linéaires qui sont moins coûteux. La stratégie de remplacement des données dans les caches, la plus communément utilisée, est une stratégie de type LRU ("Least Recently Used") [GMG08, CNLE09, FSK13]. Elle consiste à remplacer les plus anciennes données utilisées. Cette stratégie est sûrement la plus adaptée à un contexte de visualisation interactive. Cependant, Hadwiger *et al.* [HBJP12] proposent une startégie hybride entre une LRU et une MRU ("Most recently used"), utilisée pour réduire les remplacements de cache dans le cas ou toutes les briques nécessaires à un rendu ne tiennent pas en entier dans celui-ci.

Structure d'arbre

Comme nous l'avons vu à la section précédente, les structures d'arbres ont été employées pour gérer l'aspect multi-résolution dans le cadre du rendu volumique depuis le début des années 2000. Elles sont également utilisées comme structures permettant d'adresser les briques d'un volume dans un contexte out-of-core. Dans ce cas, chaque noeud de l'arbre correspond à une brique et permet d'accéder à l'adresse texture de celle-ci. La représentation multi-résolution briquée du volume est intégrée dans la structure d'arbre de la manière suivante : le noeud racine pointe sur une brique qui correspond au volume entier à la résolution la plus faible ; une feuille de l'arbre pointe sur une des briques du volume à la plus haute résolution ; un noeud intermédiaire pointe sur une brique d'une résolution intermédiaire. Les noeuds sont généralement également stockés dans un cache. Afin d'accéder à une brique, l'arbre est parcouru depuis sa racine. Le parcours efficace de kd-trees ou d'octree a beaucoup été étudié dans le domaine du ray-tracing. On retrouve des approches avec ou sans pile ("stack-based" ou "stackless" en anglais). Les approches "stackless" [PGSS07] sont plus couramment utilisées dans ce contexte car elles sont plus adaptées au calcul sur GPU. En effet, il peut être complexe de stocker et de maintenir une pile efficacement sur GPU. Il existe cependant des alternatives aux approches "stack-based", dites "short-stack", qui exploitent la proximité spatiale des noeuds voisins et qui réduisent ainsi la taille de la pile nécessaire [HSHH07]. L'algorithme du "kd-restart" [FS05] est sûrement la méthode la plus utilisée. C'est une approche "stackless" qui consiste en un parcours en profondeur jusqu'aux feuilles, qui peut recommencer depuis la racine selon que le rayon est arrété, ou qu'il continue.

Gobbetti *et al.* [GMG08] ont été parmi les premiers à utiliser un octree avec un lancer de rayon en une seule passe, proposant un parcours d'arbre avec un algorithme sans pile sur GPU. Les noeuds de leur octree stockent des pointeurs sur leurs huit noeuds enfants,



(a) (b)

FIGURE 2.19 – **Adressage out-of-core avec un N^3-tree.** Cette structure est utilisée dans Gigavoxel pour l'adressage out-of-core des données depuis le GPU [CNLE09] (a) Une structure d'arbre N^3-tree qui fait le lien avec l'ensemble des briques du volume multi-résolution, (b) Représentation des caches de briques et de noeuds liés à la structure de données. (source : [CNLE09, Cra11]).

sur les six noeuds voisins et sur les données (la brique) associées. Crassin *et al.* [CNLE09] proposent une structure d'arbre plus générale, un N^3-tree, qui est équivalente à un octree pour $N = 2$. (voir Figure 2.19a). Le sous-arbre nécessaire au rendu courant est mis à jour et géré dans un cache de noeud appellé "node pool" et les données sont stockées sous forme de briques dans un cache appellé "brick pool" (voir Figure 2.19b). Les N^3 enfants d'un noeud sont toujours stockés ensemble, il est alors possible de ne maintenir qu'un seul pointeur pour y faire référence dans leur noeud parent. Ils se basent ensuite sur un algorithme "kd-restart" pour le parcours de l'arbre sur le GPU. Engel [Eng11] utilise également une structure d'octree multi-résolution avec un parcours kd-restart appliqué à la visualisation scientifique. Plus récemment enfin, Hoetzlein [Hoe16] introduit GVDB, une structure de données dédiée à la gestion dynamique (changement de topologie) de grands ensembles de voxels, pour du rendu de simulation associé à un "ray tracer" basé sur une hiérarchie de grilles creuses. Il utilise également une structure d'arbre particulière avec sa propre méthode "short-stack" pour "ray tracer".

Table de pagination

Pour un volume briqué avec un seul niveau de résolution, il est possible d'adresser l'ensemble des briques via une table d'indexation qui peut être vue comme une table de pagination. Les tables de pagination sont utilisées dans le principe de virtualisation de la mémoire employée par les systèmes d'exploitation. Elles permettent de faire le lien entre une adresse physique en RAM et une adresse virtuelle utilisée par un programme. Dans notre contexte, cette grande table contient autant d'entrées qu'il existe de briques dans le volume et chaque entrée permet d'adresser une brique, qu'elle soit présente ou non dans le cache GPU. Cette approche est utilisée par Hadwiger *et al.* [HSS+05] pour du rendu d'isosurfaces par lancer de rayon en chargeant dynamiquement les briques intersectées par l'isosurface dans un cache texture. Scharsach *et al.* [SHN+06] en 2006 et Beyer *et al.* [BHMF08] en 2008 utilisent aussi ce système pour du rendu volumique direct en déterminant quelles briques sont actives par rapport à la fonction de transfert. Dans le cas de représentation briquée multi-résolution, il est possible d'utiliser une table de pagination par niveau de résolution [JST+10]. Après avoir déterminé le niveau de détails désiré pour un échantillon, il suffit d'utiliser la table de pagination correspondant à ce niveau de résolution. Hadwiger

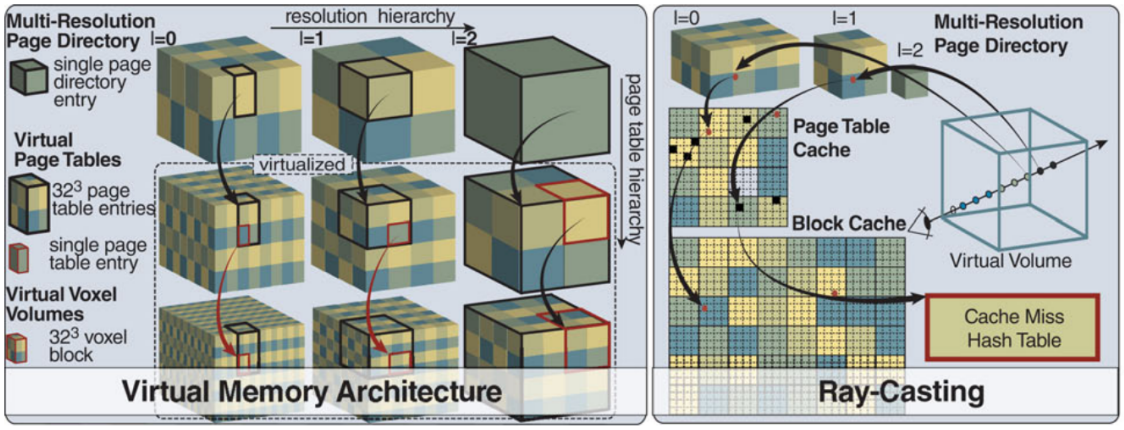


FIGURE 2.20 – **Hiérarchie de tables de pagination multi-niveaux, multi-résolutions.** En plus d'utiliser une table de pagination par niveau de résolution, cette structure propose une hiérarchie de table de pagination permettant d'augmenter la quantité de données à virtualiser. (source : [HBJP12]).

et al. [HBJP12] utilisent plutôt une approche avec une table de pagination multi-niveau pour adresser de très grands volumes multi-résolutions. En effet, ils montrent que pour de très grands volumes avec de petites briques, la table de pagination devient trop grande pour tenir en mémoire GPU. Ils appliquent donc le principe de virtualisation à la table de pagination elle-même afin d'augmenter la quantité virtualisable. On se retrouve dans ce cas avec une hiérarchie de table de pagination en plus d'avoir une table par niveaux de résolution (voir Figure 2.20). Ils utilisent également un cache pour chaque niveau de virtualisation (de table de pagination) et seul le premier niveau (le plus petit) de cette structure est entièrement physiquement présent sur le GPU. Les mises à jour de la structure et la gestion de l'ensemble des caches sont cependant fait sur CPU, impliquant ainsi des transferts entre les deux precesseurs. Une telle structure de table de pagination multi-niveaux, multi-résolution, a été utilisée pour de la visualisation de très grands volumes de données en microscopie électronique [BHJ+11, HBJP12, BHAA+13].

En pratique, un arbre peut être vu comme une table de pagination pour laquelle un noeud est une entrée de cette table. A l'inverse, une hiérarchie multi-niveaux de table de pagination peut être considérée comme une structure d'arbre. Hadwiger *et al.* [HBJP12] proposent une comparaison entre l'utilisation d'un octree multi-résolution pour l'adressage out-of-core de grands volumes de données avec leur hiérarchie de table de pagination multi-niveaux, multi-résolution. Ils montrent que cette dernière est plus adaptée qu'un octree pour la gestion de très grands volumes de données. Ils démontrent que la profondeur d'un arbre, et donc la complexité d'adressage, est directement liée à la représentation multi-résolution. En effet, le ratio de sous-échantillonnage choisi entre deux niveaux de résolution impacte la forme de l'arbre. Une hiérarchie de table de pagination permet de dissocier complètement ces deux aspects, et ainsi, maintenir un "saut" peu important entre deux niveaux de résolution, tout en ayant une structure peu profonde. De plus, avec une telle structure, la compléxité d'accès à un élement est complétement indépendante du niveau de résolution, alors qu'avec un octree, plus l'on désire accéder à un niveau de résolution élevé, plus le parcours de l'arbre est long. Pour finir, une structure d'arbre nécessite un pré-traitement pour calculer celui-ci et un chargement de sous-arbres sur le GPU pour garantir l'accès des briques.

Défauts de cache et requêtes de briques

Si une brique n'est pas présente dans le cache GPU dédié, un défaut de cache est levé ("cache miss" en anglais) et il faut alors charger cette brique jusqu'au cache texture sur le GPU. Cyril Crassin, dans sa thèse [Cra11], détaille l'utilisation d'un buffer de requêtes utilisé pour calculer une liste de requêtes en une passe avec un kernel sur le GPU. Cette courte liste contient les identifiants des briques non présentes en cache et nécessaires au lancer de rayon (voir Figure 2.21).

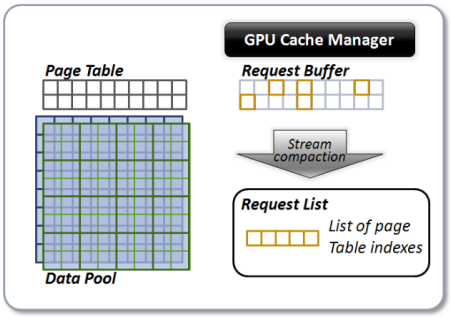


FIGURE 2.21 – **Création d'une liste de requêtes à partir des défauts de cache.** Cyril Crassin décrit l'utilisation d'un buffer de requêtes lié à la table de pagination pour créer une liste des briques à charger sur le GPU. (source : [Cra11]).

Il utilise ce même système pour charger les noeuds servant alors de table de pagination, des sous-arbres correspondants dans l'octree associé. Hadwiger *et al.* [HBJP12] utilisent des tables de hachages pour gérer les requêtes de briques. Ces tables sont organisées dans l'espace écran et reportent ainsi les défauts de cache provoqués par chaque rayon. Ils limitent le nombre de requêtes par rayon afin de répartir la charge sur plusieurs images. Fogal *et al.* [FSK13] utilisent également le même système, avec une version compressée des tables de hachages afin de réduire les communications entre le GPU et le CPU. Dans toutes les approches, la gestion des requêtes de briques proposée est intrinséquement liée à la visualisation. Ceci peut poser problème pour utiliser ce système dans une approche qui voudrait effectuer par exemple des traitements sur des données qui ne sont pas visibles à l'écran mais qui pourront l'être par la suite.

Généralement, les approches proposées partent du principe que toutes les briques nécessaires à chaque instant de la visualisation interactive, tiennent en entier dans le cache GPU. Certains travaux changent cependant la stratégie de rendu dans le cas contraire, en utilisant un rendu multi-passes au lieu d'un rendu en une seule passe [Eng11, FSK13].

Avancées des technologies matérielles

Les méthodes existantes se basent sur une solution logicielle proposant un algorithme complet de gestion out-of-core des données depuis le GPU, en passant par le CPU et éventuellement jusqu'au disque pour certaines. L'évolution matérielle est également à prendre en compte. En particulier, l'implémentation CUDA comprend un principe appelée mémoire unifiée, qui permet de ne voir qu'un seul espace d'adressage depuis le GPU et qui ne nécessite pas de transfert explicite depuis le CPU. Alors qu'il a d'abord été limité à des allocations ne dépassant pas la quantité de mémoire physique des GPUs, ce principe a été amélioré à partir de l'architecture Pascal des cartes Nvidia et de la version 8.0 de CUDA [Nvi]. Il est désormais possible d'allouer des buffers d'une taille égale à la capacité mémoire de la RAM CPU et les défauts de pages sur GPU sont alors gérés automatiquement. Au cours de l'année 2018, à la toute fin de cette thèse, est apparu un système qui pourrait permettre de compléter encore ce principe avec le support du service de traduction d'adresse (Address Translation Support) à travers la technologie de communication NVLink, présent dans les versions 7.0 du "compute capability" de CUDA. Cela permet au GPU d'accéder directement aux tables de pagination du CPU via un système de requêtes (Address

Translation Request) qui fait alors le lien avec le système de virtualisation du système d'exploitation. Avec l'utilisation de l'instruction unix "mmap", il serait alors possible d'avoir un pipeline complet du GPU jusqu'à un fichier sur disque de n'importe quelle taille (limité cette fois par la quantité de l'espace adressable), qui gérerait les défauts de pages à tous les niveaux nécessaires. Il ne serait alors plus nécessaire d'implémenter et de maintenir notre propre cache de briques sur le GPU. Cependant, ceci est rendu possible uniquement sur les cartes Volta V100 de Nvidia avec la technologie NVLink et CUDA 9.2, en association avec une architecture CPU Power9 [STKS17] et un système linux uniquement avec le dernier noyau disponible (4.16). De plus, cette méthode n'offre pas de contrôle suffisant sur les défauts de pages et sur le cache, en particulier, cela ne permettrait pas l'utilisation de la mémoire texture du GPU, qui joue pourtant un rôle majeur dans l'accélération matérielle de l'échantillonage pour le rendu volumique.

2.4.3 Détermination de la visibilité

Une partie importante de la visualisation interactive de grandes masses de données est de limiter la quantité de mémoire nécessaire à son strict minimum, tout en garantissant un rendu visuellement correct. L'approche de bricking vue à la section 2.4.1 est une prémisse de cette notion. Il reste alors à déterminer quelles briques sont réellement nécessaires à la visualisation à un instant donné, pour un certain point de vue. L'ensemble de ces briques est généralement appelé "espace de travail" ("working set" an anglais) dans la littérature. Les approches les plus anciennes se basent sur des méthodes d'abattage ("culling" en anglais). Celles-ci consistent, en partant d'un jeu de données complet, à retirer des données inutiles si celles-ci ne répondent pas à certains critères de visibilité. Ces méthodes nécessitent un calcul explicite de la visibilité des briques. Les méthodes out-of-core plus modernes utilisent plutôt une approche de lancer de rayon avec une détermination intuitive des briques au fur et à mesure de l'échantillonage le long des rayons. A l'inverse des méthodes de "culling", l'espace de travail est vide au départ et on y ajoute des briques visibles au fur et à mesure. Ces méthodes sont appelées "ray-guided" ou encore "visibility-driven". Tous ces aspetcs sont présentés dans la suite de cette section.

Méthodes de "culling"

Il existe plusieurs méthodes d'abattage. Ces approches ont été introduites en informatique graphique pour du rendu de géométrie (maillages de triangles ou de polygones) afin de limiter le nombre de primitives à afficher à l'écran. Cependant, ces approches peuvent s'adapter à l'abattage de briques dans le contexte de rendu volumique.

- Tout d'abord, l'abattage du cône de vision ("view frustum culling" an anglais), est la forme la plus basique. Elle consiste à déterminer quelles briques sont à l'intérieur du cône de vision défini par le système de caméra. Les briques en dehors de ce cône de vision ne sont effectivement pas visibles à l'écran et il est alors inutile de les charger en mémoire GPU. Pour déterminer cela, il est nécessaire de réaliser un test d'intersection des plans du cône de vision avec la boîte englobante de chacune des briques du volume [AM00].
- Une autre approche consiste à déterminer quelles parties, présentes dans le cône de vision, sont occultées. On parle ici d'"occlusion culling". En rendu volumique, une brique peut être occultée par un ensemble de briques opaques qui se trouveraient devant elle dans l'axe de vision. Li *et al.* [LMK03] introduisent une méthode de ce type pour éviter le rendu de parties inutiles d'un volume dans une approche

basée texture. Marchesin et Ma [MM10], présentent un pipeline de rendu volumique parallèle qui utilise un calcul d'occlusion propagé entre les différents noeuds de rendu.

— Nous pouvons aussi retrouver une famille de méthodes d'abattage basée sur certaines propriétés ou attributs. Il est possible par exemple de déterminer quelles briques sont visibles par rapport à la fonction de transfert. Il est alors nécessaire de stocker une valeur minimum et une valeur maximum des voxels de chaque brique et de les comparer ainsi à la fonction de transfert pour déterminer si oui ou non, tous les voxels d'une brique sont considérés comme totalement transparents [HSS+05]. On retrouve aussi des approches de rendu volumique sur des objets segmentés qui retirent certaines parties d'un volume selon la segmentation en activant ou désactivant ces zones [BHWB07].

Gobbetti *et al.* [GMG08] utilisent un calcul de visibilité hybride. Ils utilisent des "occlusion queries" sur GPU [BWPP04], basées sur un partitionnement de l'écran pour détecter les parties visibles d'une subdivision de l'espace, basé sur une structure d'arbre. Cette approche nécessite des échanges GPU-CPU coûteux. Ils uilisent ensuite des méthodes de "culling" classiques sur CPU.

Approches "ray-guided"

Les méthodes "ray-guided" construisent l'espace de travail pendant le parcours des rayons à l'intérieur du volume. De cette manière, seules les briques réellement rencontrées par au moins un rayon sont chargées sur le GPU. Crassin *et al.* [CNLE09] proposent un lancer de rayon totalement "ray-guided" en 2009 avec Gigavoxel, avec du rendu de scènes opaques très détaillées pour des applications de divertissement. Engel [Eng11] utilise également un pipeline "ray-guided" appliqué à de la visualisation scientifique sur des données Teravoxels. Il améliore les communications entre CPU et GPU par rapport à l'approche de Crassin *et al.* [CNLE09] et gère dynamiquement des changements de fonction de transfert en temps réel. Hadwiger *et al.* [HBJP12] introduisent en 2012 une méthode de virtualisation de la mémoire avec une hiérarchie de tables de pagination multi-niveaux multi-résolutions, combinée à un pipeline "ray-guided" pour du rendu de très grands volumes issus de flux continus provenant de microscopes électroniques. Fogal *et al.* [FSK13] proposent une analyse des méthodes de lancer de rayon out-of-core "ray-guided" en proposant leur propre pipeline. Ils donnent également une étude détaillée de plusieurs aspects spécifiques propres à ces méthodes.

2.5 Conclusion

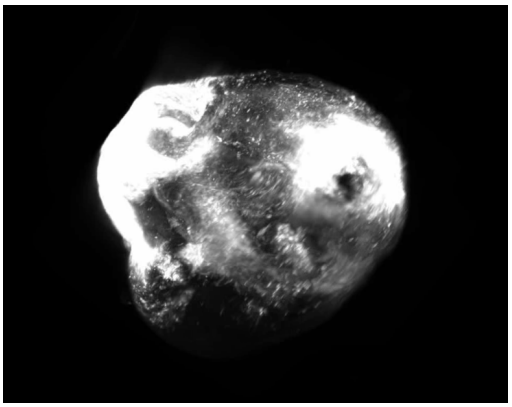
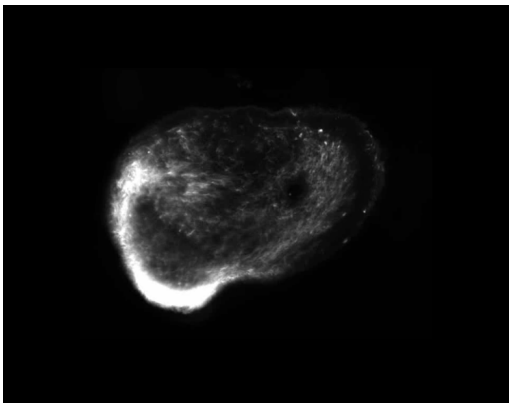
Dans ce chapitre, nous avons présenté différentes méthodes de la littérature, propres aux principaux champs que nous abordons dans cette thèse, regroupés en un domaine d'étude : la visulisation temps-réel out-of-core sur GPUs, pour de grandes masses de données volumiques. Dans ce contexte, les avancées dans la littérature nous permettent de nous positionner sur des méthodes servant de base à un grand nombre de travaux en terme de représentation et de gestion des données. Les grandes idées sur lesquelles reposent les méthodes modernes de lancer de rayon volumique out-of-core sur GPU sont déjà établies et il est alors intéressant de les utiliser pour développer notre propre approche, répondant aux objectifs de cette thèse. En particulier, les méthodes de "bricking", de représentation multi-résolution ou encore de l'utilisation d'un système de cache en mémoire texture 3D sur GPU pour stocker des données acheminées à la demande. Comme nous l'avons vu, les structures d'adressage out-of-core proposées sont souvent basées sur une structure d'arbre (BSP-trees, kd-trees et surtout octrees). En 2012, Hadwiger *et al.* [HBJP12] introduisent une nouvelle approche de virtualisation de la mémoire avec une hiérarchie de tables de pagination multi-niveaux multi-résolutions. Cette méthode est plus adaptée à de très grands volumes de données et ne nécessite ni construction, ni maintien et parcours d'arbre potentiellement profond sur GPU. Cette méthode nous semble donc intéressante, en dehors de son cadre applicatif spécifique au lancer de rayon pour du rendu volumique direct sur des données issues d'un flux continu de microscope électronique. Bien que toutes les méthodes présentées adressent la plupart des problèmes de la gestion out-of-core de données dans ce contexte, elles ne mentionnent pas tous les besoins d'un point de vue applicatif. En effet, elles sont souvent spécifiques à un type de visualisation et sont même fortement orientées par ce principe même.

Notre approche

Dans ce contexte, nous proposons un pipeline out-of-core complet, du disque dur jusqu'au GPU pour visualiser et/ou traiter de très grands volumes de données efficacement en temps interactif. Notre méthode tire pleinement partie de l'architecture massivement parallèle des GPUs modernes et elle permet de limiter les communications entre CPU et GPU à leur strict minimum. L'approche proposée n'est pas spécifique à un type de visualisation en particulier et a été conçue de manière à pouvoir visualiser et/ou traiter des données de différentes manières. Ainsi nous montrons dans un premier temps, son efficacité dans le cadre d'une visualisation de type microscopie virtuelle. Dans ce contexte nous proposons une extension à de la visualisation 3D en relief sur écran auto-stéréoscopique afin de garantir une bonne perception tridimentionnelle des données. Deuxièmement, nous utilisons notre pipeline pour du rendu volumique direct en temps interactif sur GPUs, en environnement HPC ("High Performance Computing" en anglais pour "Calcul Haute Performance"). Nous montrons ainsi la possibilité de distribution d'une telle approche afin d'exploiter des serveurs de rendu performants. Cet aspect permet de répartir la charge de calcul, aussi bien d'un point de vue algorithmique pour la visualisation, que pour la gestion des transferts de données qui présentent généralement un goulot d'étranglement dans ce genre d'approche.

Chapitre 3

Virtualisation de très grands volumes de données entièrement gérée sur GPU

Sommaire

3.1 Introduction

L'ensemble des aspects évoqués dans ce chapitre fait l'objet d'une collaboration à part égale avec les travaux de thèse de Nicolas Courilleau.

Nous allons présenter dans ce chapitre, l'ensemble du pipeline que nous avons conçu et implémenté. Celui-ci permet de répondre efficacement à des besoins de visualisation interactive et de traitement à la demande sur des grilles 3D de très grands volumes dépassant la quantité mémoire physique disponible sur GPU et sur CPU. Après avoir donné une vue d'ensemble de notre pipeline et de ces avantages par rapport aux travaux précédents, nous décrirons la représentation des données que nous proposons. Nous décrirons ensuite le concept de virtualisation de la mémoire utilisée pour adresser de grandes quantités de données out-of-core depuis le GPU. Nous utilisons une structure de données représentée sous forme d'une hiérarchie de tables de pagination multi-résolutions combinée à un mécanisme de caches sur GPU, dont nous détaillerons aussi le fonctionnement dans ce chapitre.

3.2 Pipeline général

L'approche proposée comprend l'implémentation d'un pipeline complet du disque dur jusqu'au GPU, en passant par le CPU pour manipuler de très grandes quantités de données volumiques pour n'importe quel type d'application de visualisation interactive ou de traitement à la demande. Ce pipeline, présenté dans la figure 3.1, comprend :

1. Une représentation multi-résolution briquée et compressée des données, stockée sur un espace de stockage de grande capacité. Cette représentation est une pyramide mipmap 3D.
2. Un grand cache de briques dans la mémoire RAM centrale du CPU, accompagné d'un système de chargement de briques à l'interface du disque d'un coté et du GPU de l'autre.
3. Un système d'adressage virtuel sur GPU, accompagné d'une structure de données de tables de pagination multi-niveaux, multi-résolutions et d'un cache de briques en mémoire texture 3D.
4. Un gestionnaire de cache entièrement sur GPU pour maintenir les caches à jour, et offrir une gestion efficace des requêtes de données.

Nous proposons d'implémenter une table de pagination multi-niveau, multi-résolution, pour adresser un volume complet en utilisant un système de gestion de mémoire virtuelle. Cette structure est une hiérarchie pouvant être composée de plusieurs niveaux de virtualisation, comprenant alors chacun une table de pagination, afin d'être capable d'adresser de très grands volumes. Dans ce cas, chaque niveau de virtualisation, sauf la racine, stocke alors ses pages dans un cache dédié. La structure contient également un cache de briques utilisé pour stocker les données elles-mêmes. Chaque cache est stocké dans une texture 3D sur le GPU, et ils sont gérés individuellement par une LRU. Nous proposons une gestion de ces caches GPU entièrement sur GPU, afin de profiter de leur architecture massivement multi-thread d'une part, et de limiter les communications et les synchronisations entre le CPU et le GPU d'autre part. La structure utilisée est entièrement créée sur le GPU et reste uniquement présente sur celui-ci tout au long de l'exécution.

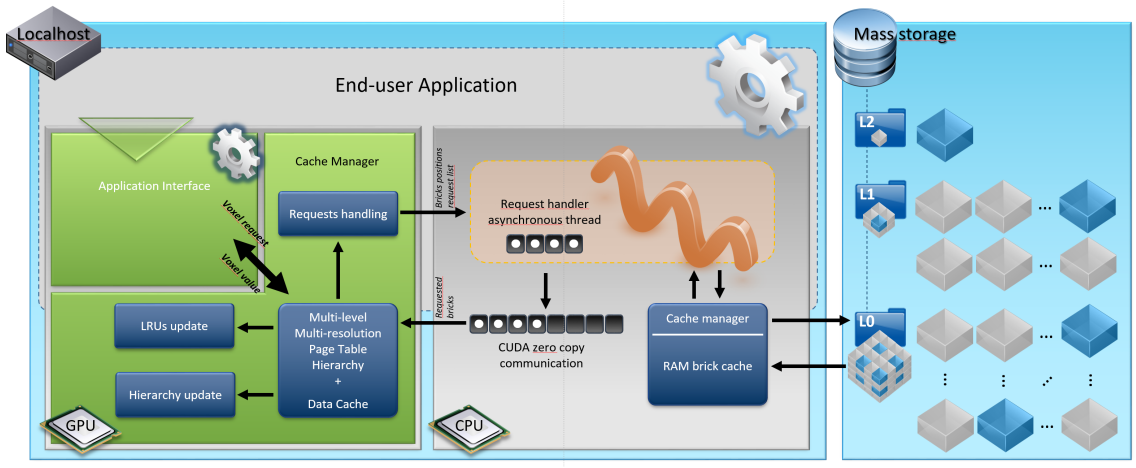


FIGURE 3.1 – **Pipeline complet.** Ce pipeline présente toutes les étapes mises en place pour permettre à une application sur le GPU d'accéder, à la demande, à n'importe quelle partie d'une grille de voxels, dépassant la capacité mémoire du GPU et du CPU.

Gestion des caches GPU

La gestion des caches GPU peut se résumer en deux tâches principales :

1. La mise à jour de l'utilisation des éléments présents dans les caches et la mise à jour des LRUs associées.
2. La gestion des défauts de cache.

Pipeline détaillé

Notre pipeline propose une interface pour n'importe quel type d'application qui a la nécessité de manipuler un très grand volume de données, représenté sous forme d'une grille régulière 3D de voxels. La navigation dans le volume multi-résolution depuis le GPU se fait dans un volume normalisé. A tout moment de l'exécution de l'application, un voxel peut se trouver à différents emplacements mémoire (disque, mémoire CPU ou mémoire GPU). L'adressage d'un voxel se fait cependant dans un seul espace d'adressage uniforme, peu importe l'emplacement mémoire physique de celui-ci. Afin d'accéder à un voxel, l'application détermine une paire (l, p) avec l, le niveau de détail désiré dans la représentation multi-résolution, et p, la position 3D flottante dans le volume normalisé ($(p \equiv [x, y, z] \in [0, 1[^3)$). Lorsque l'application veut accéder à un voxel, l'ensemble du pipeline est alors déclenché de la manière suivante :

— **Lookup** La première étape est de vérifier dans la structure de table de pagination hiérarchique, si la brique qui contient le voxel en question est présente ou non dans le cache de briques sur le GPU.

— **Hit** Dans le cas où cette brique est bien présente dans le cache, le gestionnaire de cache peut directement fournir le voxel requêté à l'application. Il va ensuite signaler l'usage de la brique correspondante.

— **Miss** Dans le cas où la brique n'est pas présente dans le cache, un défaut de cache ("cache miss" en anglais) est levé. Le gestionnaire de cache va alors signaler la requête de cette brique.

Notre implémentation vise une utilisation de ce pipeline pour différents types d'application. La stratégie optimale de mise à jour des caches dépend de l'application et est certainement différente, selon que l'on soit dans un contexte de lancer de rayon volumique, ou encore de visualisation volumique basée sur des coupes 2D. Cette mise à jour des caches consiste, d'un coté, à mettre à jour les LRUs à partir des utilisations des éléments du cache correspondant, et d'un autre coté, à répertorier tous les caches miss provoqués par le cache de briques (pour chaque brique manquante). A partir de cette deuxième information, une courte liste des briques requêtées est créée puis envoyée au CPU pour être gérée de manière asynchrone par celui-ci. Un thread CPU dédié effectue alors les requêtes auprès du gestionnaire de cache de briques CPU. Si nécessaire, les briques sont lues depuis le disque puis écrites dans le cache CPU. Toutes les briques requêtées sont ensuite transférées dans un buffer accessible par le GPU. Quand les briques pourront être écrites dans le cache de briques GPU, la table de pagination et les LRUs seront mises à jour en conséquence.

En utilisant ce pipeline, l'application peut gérer efficacement les requêtes de voxels en ne demandant que les briques nécessaires, de manière à ne jamais charger les briques inutiles. Par exemple, une application de lancer de rayon volumique peut assurer cela en utilisant une approche "ray-guided", de manière à orienter le calcul de visibilité des briques selon le parcours des rayons dans le volume, et ne demander que les briques rencontrées le long de ceux-ci (voir section 2.4.3).

Communications

Dans le système que l'on propose, les communications entre le système central et le GPU sont réduites à leur strict minimum (voir Figure 3.2). Ainsi, les transferts initiés par le GPU et envoyés au CPU sont limités à une simple liste des identifiants des briques requêtées. Les échanges dans l'autre sens sont limités aux données elles-mêmes, les briques contenant les voxels, ainsi qu'un Boolean, pour chaque brique requêtée, indiquant si celle-ci est considérée comme vide ou non (donc envoyée au GPU ou non). Toutes les autres actions nécessaires à la gestion des données out-of-core sont effectuées par le GPU, tirant ainsi partie de sa puissance de calcul et limitant les communications coûteuses avec le CPU.

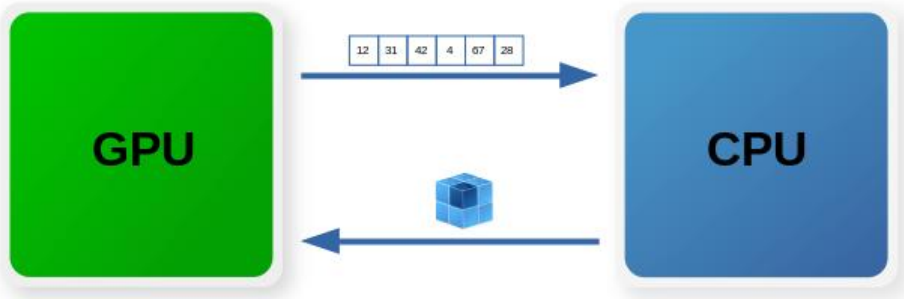


FIGURE 3.2 – **Communications entre le GPU et le système central dans le pipeline proposé.** Les communications sont réduites à leur strict minimum afin d'éviter les goulots d'étranglement : les identifiants des briques requêtées d'une part et les briques de voxels elles-mêmes d'autre part. En particulier, il n'y a aucun transfert de pages ou d'une quelconque partie de la structure de données.

3.3 Positionnement par rapport aux travaux précédents

Ce pipeline est proche, pour certaines parties, de ceux généralement proposés dans des contextes similaires. En revanche, il diffère pour plusieurs aspects qui en font un pipeline efficace et générique. Les approches modernes qui ont proposé un pipeline complet pour traiter le même genre de problématiques se sont concentrées sur le développement de pipelines dit "visualisation-driven" en anglais. Cette appellation est due au fait que c'est l'étape de visualisation qui dirige l'ensemble du pipeline pour répondre à ses besoins. Comme nous travaillons dans un contexte plus général que la visualisation uniquement, nous pourrions alors décrire notre système comme un pipeline "déclenché à la demande", ou "on-demand triggered" en anglais.

Les principaux travaux les plus proches des nôtres, dans le développement d'un tel pipeline complet à des fins de gestion out-of-core de grands volumes de données, sont ceux de Gobbetti *et al.* [GMG08] en 2008, Crassin *et al.* [CNLE09] en 2009, Hadwiger *et al.* [HBJP12] en 2012 et Fogal *et al.* [FSK13] en 2013.

Cependant, Gobbetti *et al.* et Hadwiger *et al.* proposent une gestion de leur structure (un octree pour les premiers et une hiérarchie de tables de pagination multi-niveaux, multi-résolutions pour les deuxièmes) sur CPU. De plus, bien que Hadwiger *et al.* utilisent une structure de données très semblable à la nôtre, leur pipeline comprend plusieurs étapes qui le complexifient. En effet, ils construisent des blocs 3D seulement à la demande, à partir des données stockées sous forme de tuiles 2D, directement générées par un flux continu provenant d'un microscope électronique, après une étape de recalage des images.

Crassin *et al.* proposent une gestion entièrement sur GPU, mais dans le cadre de l'utilisation d'un octree comme structure d'adressage d'un volume multi-résolution, qui diffère donc sur plusieurs points. En effet, leur "pages" sont stockées dans les noeuds de l'arbre et ceux-ci doivent être transférés depuis le CPU en plus des briques de voxels. En plus des données, leur structure est également gérée de manière out-of-core et doit être acheminée et maintenue sur GPU au fur et à mesure, sous forme de sous-arbres. De plus, leurs travaux sont à la base destinés à manipuler des volumes qui tiennent en mémoire centrale.

Fogal *et al.* proposent eux aussi une gestion sur GPU et pour une structure qui semble proche de celle que l'on propose. Cependant, ils ne donnent pas de détail sur celle-ci, et très peu sur la manière de la gérer efficacement sur GPU. De plus, ils utilisent un système de gestion des caches misses (comme dans tous les autres travaux mentionnés ici) fortement dirigé par la visualisation, et donc directement lié à la visibilité des données.

Dans ce contexte, nous proposons un pipeline qui corrige tous les problèmes précédement identifiés dans cette section. Ainsi, la structure d'adressage que l'on propose est entièrement présente sur le GPU, et les pages utilisées pour l'accès virtuel aux données sont directements créées à la volée sur celui-ci (et non transférées depuis le CPU). Cette structure n'est pas dupliquée sur le CPU, ni transférée depuis celui-ci et est entièrement gérée sur le GPU. Nous réduisons ainsi les communications à leur minimum. De plus, sa gestion est pensée de manière générique afin de pouvoir être utilisée pour n'importe quel type d'application de la visualisation volumique interactive au traitement à la demande, pendant la visualisation, sur des données visibles ou non visibles.

3.4 Données : Pyramide 3D multi-résolution briquée

Nous utilisons une représentation multi-résolution briquée (voir section 2.4.1) pour la grille régulière 3D de voxels que nous devons manipuler dans le pipeline précédement décrit. Cette représentation peut être vue comme un mipmap 3D découpé en petites briques, comme illustré sur la figure 3.3. Pour rappel, cette approche apporte deux avantages majeurs. Premièrement, l'aspect multi-résolution, en plus de corriger des problèmes d'aliasing, permet de réduire la quantité de données à afficher et donc également à stocker en cache. Deuxièmement, le bricking permet de diviser un très gros volume qui ne peut pas être manipulé comme une seule entité, en un ensemble de petits volumes légers et ainsi répartir la complexité.

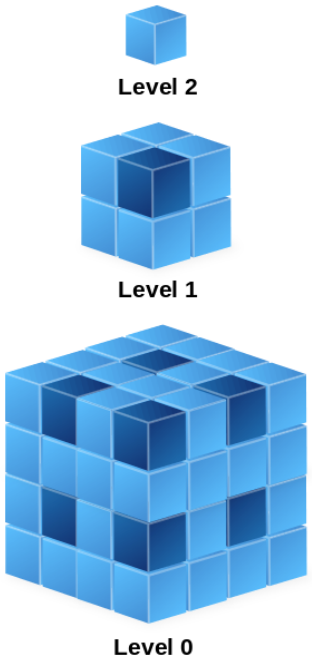


FIGURE 3.3 – **Illustration de la représentation muli-résolution briquée d'une grille régulière 3D de voxels.**

Le volume original, le plus résolu (noté $L0$) est décliné en plusieurs versions de moins en moins détaillées (trois niveaux de détails en tout sur la figure 3.3). Chaque niveau est ensuite découpé en petites briques de voxels avec une taille constante (selon le niveau de résolution) et paramétrable (selon les volumes). Ces étapes sont entièrement réalisées en pré-traitement et n'impacte donc pas l'interactivité du pipeline décrit dans la section précédente. Toutes les briques sont directement accessibles sur le disque sans aucun traitement (sauf une éventuelle décompression comme nous le verrons par la suite), et cela pour n'importe quel niveau de résolution, pendant toute l'exécution en temps-réel.

3.4.1 Mipmapping

L'obtention d'un volume à un niveau de résolution L_i se fait à partir du niveau L_{i-1}, en appliquant une moyenne sur tout le volume par groupes 3D de pixels. La taille de ce groupe est paramétrable et peut être différent sur chacun des trois axes et pour chaque niveau de résolution. Cependant, en pratique, il est souvent choisi avec $x = 2$, $y = 2$ et $z = 2$ pour tous les niveaux. Le fait de pouvoir choisir un ratio différent sur un des trois axes permet de réduire un effet d'anisotropie souvent présent après acquisition de données biomédicales. Lors de l'acquisition d'un ensemble de coupes biologique 2D, la précision d'acquisition est souvent plus importante entre 2 pixels dans le plan, que la distance entre deux coupes consécutives. C'est cela qui engendre une anisotropie dans les données. Ceci peut être progressivement corrigé en paramétrant un ratio de sous-échantillonage plus grand sur l'axe qui cause cette anisotropie.

D'un point de vue implémentation, nous proposons une méthode out-of-core sur GPU pour cette étape de "mipmapping". Nous utilisons un buffer de pixel alloué à partir d'une certaine taille en octet N, paramétrable selon les capacités mémoire du GPU. Il s'agit ensuite de transférer progressivement tout le volume sur le GPU étape par étape. Chaque

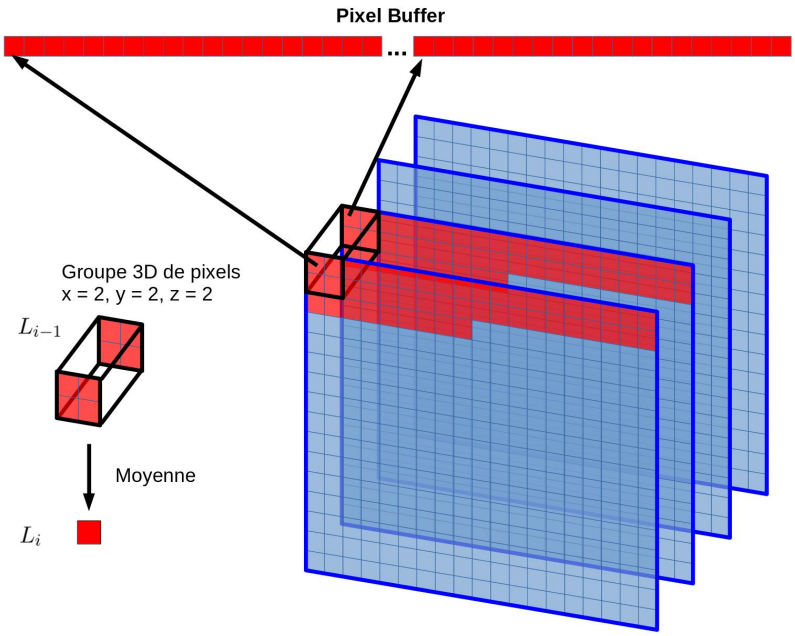


FIGURE 3.4 – **Sélection des voxels pour le mipmaping out-of-core** Sur cette illustration, le ratio de sous-échantillonage choisi est de $x = 2$, $y = 2$ et $z = 2$. La taille du buffer de pixels à envoyer au GPU est de $N = 94$ octets. Un pixel est encodé sur $E = 1$ octet. Le nombre de voxels à sélectionner par coupe est donc $nb_{vox} = \frac{N}{E \times n} = \frac{94}{1 \times 2} = 47$. La taille du groupe 3D de pixels à moyenner est de $x \times y \times z = 2 \times 2 \times 2 = 8$

étape consiste à transférer autant de voxels que ce qui est possible de faire tenir dans le buffer. Le schéma de sélection des voxels est le suivant : on sélectionne progressivement dans une coupe, ainsi que dans les $n-1$ coupes suivantes sur la 3ème dimension (avec n égale au ratio de sous-échantillonage choisi pour z, donc généralement $n = 2$), $nb_{vox} = \frac{N}{E \times n}$, avec E le nombre d'octet d'encodage d'un pixel. Une fois le buffer en mémoire GPU, nous utilisons un kernel CUDA avec autant de threads que nécessaire pour appliquer une moyenne par groupe 3D de pixels sur tous les pixels contenus dans ce buffer, en une seule passe. Un buffer de sortie est ensuite envoyé au CPU pour écriture sur disque de cette partie du volume sous-échantilloné. Ces opérations se répètent ainsi séquentiellement jusqu'à ce que l'ensemble du volume soit sous-échantillonné. A nouveau, ce processus est repété autant de fois que nécessaire pour chaque niveau L_i de la pyramide multi-résolution. Ce nombre de niveaux dépend des ratios de sous-échantillonnage choisis et de la taille des briques. Le but étant de n'avoir plus qu'une seule brique pour le niveau le moins détaillé.

3.4.2 Bricking

L'étape de bricking est quant à elle entièrement réalisée sur CPU, car elle ne se prête pas particulièrement bien à l'utilisation d'une carte accélératrice. En effet, elle ne consiste "que" en un grand nombre de lectures et d'écritures sur disque. La complexité de cette étape repose plutôt dans les schémas d'accès 3D au voxels à partir d'une représentation 1D dans le fichier raw.

Taille et forme des briques

Une première chose à noter est la régularité des briques en fonction du niveau de résolution. En effet, les briques ont toutes la même taille, peu importe le niveau de détail du volume dans la pyramide multi-résolution. Cette taille est exprimée en voxels et non pas comme une "taille spatiale". Le nombre de briques décroit alors très fortement au fur et à mesure des niveaux de la pyramide jusqu'au dernier niveau, le moins résolu, qui contient ainsi une seule brique représentant l'ensemble du volume. Plusieurs travaux se sont déjà intéressés à l'étude de la taille optimale des briques de voxels (voir section 2.4.1). En suivant ces travaux, nous optons pour une taille de brique allant de 32^3 à 256^3 voxels selon le volume. De plus, comme Hadwiger *et al.* [HBJP12] et Fogal *et al.* [FSK13], nous utilisons un système de redécoupage dynamique des briques. Cela nous permet d'optimiser le stockage et les transferts avec des briques de grande taille (128^3 - 256^3 voxels) d'une part, et d'optimiser leur manipulation sur le GPU avec une forme plus petite (32^3 - 64^3 voxels) d'autre part. Ce système nous permet également d'éviter un autre problème. En effet, nous avons constaté que pour de très grands volumes, composés donc d'un très grand nombre de briques, le disque et l'OS n'arrivent pas à gérér la quantité d'opérations pour un très grand nombre de très petits fichiers. Il est donc nécessaire à ce niveau, d'utiliser des briques de grande taille pour éviter un fort ralentissement de cette étape de bricking. Certaines approches n'utilisent que des grilles de briques en puissance de 2. Cela provoque une surcharge de la quantité de données si la taille du volume lui même n'est pas une puissance de 2. En effet, il faut alors compléter le volume avec autant de briques "vides" que nécessaire sur les trois dimensions. A l'inverse, notre système est capable de gérer des volumes de n'importe quelle taille et nous complétons seulement les dernières briques qui "dépassent" du volume en ajoutant du "vide". Le processus de découpage des briques peut également prendre en compte un chevauchement de celles-ci pour assurer une cohérence spatiale pour certains traitements. Nous avons choisi de laisser la taille de chevauchement paramétrable de manière à satisfaire les besoins de différents traitements. En effet, un lancer de rayon volumique sur GPU se contentera d'un chevauchement d'un voxel pour garantir une bonne interpolation dans l'étape d'échantillonnage dans la texture 3D. En revanche, il serait plus interéssant d'avoir un chevauchement plus grand afin de pouvoir appliquer par exemple un masque de convolution (par exemple de taille $5 \times 5 \times 5$) sur les données pour d'éventuels traitements.

Compression

Après découpage, une brique peut être compressée de manière à réduire la représentation mémoire pour son stockage. Nous utilisons une compression LZ4 [1] qui repose sur un algorithme de compression par dictionnaire LZ77. Cet algorithme de compression sans perte, très rapide, permet de compresser et surtout de décompresser les données en temps réel, tout en ayant de bons taux de compression. Les briques peuvent donc être stockées sous forme compressée, puis être décompressées à la volée sur le CPU pendant l'exécution.

L'arborescence sur disque est composée d'un fichier raw (compréssé) par brique, tous organisés dans un répertoire par niveau de résolution L_i.

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

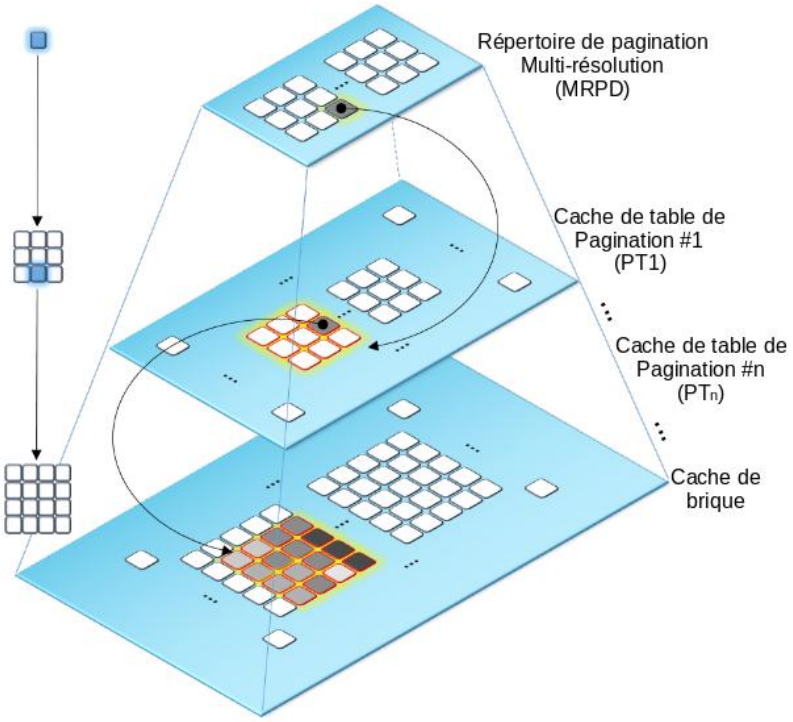


FIGURE 3.5 – **Structure d'adressage out-of-core : hiérarchie de tables de pagination multi-niveaux, multi-résolution.** Une représentation 2D de notre structure d'adressage 3D. Dans cet exemple, on utilise un niveau intermédiaire de virtualisation (PT1). Une entrée de la *MRPD* adresse un *bloc PT* de 3×3 entrées dans le cache PT1 et une entrée dans le cache PT1 adresse une brique de 4×4 voxels dans le cache de briques.

3.5 Adressage : Structure de pagination

3.5.1 Présentation de la structure

Nous allons détailler ici la structure de données que l'on propose d'utiliser pour adresser l'ensemble des briques d'un grand volume multi-résolution depuis le GPU dans un contexte out-of-core. Cette structure permet un adressage virtuel efficace, à l'aide d'une hiérarchie de table de pagination multi-niveaux, multi-résolutions. Elle est basée sur une grande table de pagination organisée sur un ou plusieurs niveaux, qui consiste à traduire l'adresse virtuelle de chaque voxel du volume multi-résolution, en une adresse physique. Elle peut être vue comme une structure pyramidale où chaque niveau virtualise un ensemble d'entrées, organisées en blocs 3D dans le niveau en dessous (voir Figure 3.5). Le principe de cette structure d'adressage virtuel a été introduit par Hadwiger et al. [HBJP12] en 2012 (voir Section 2.4.2).

Le premier niveau de la hiérarchie, est constitué du répertoire de pagination multi-résolution (que l'on appelera *MRPD*, pour le terme anglais "Multi-Resolution Page Directory"). Celui-ci est le point d'entrée de tout parcours de la hiérarchie pour l'adressage virtuel d'un voxel, il permet d'accéder à n'importe quelle partie du volume multi-résolution. C'est une table de pagination composée de pages organisées en autant de grilles 3D qu'il y a de niveaux dans la réprésentation multi-résolution du volume. La *MRPD* est le seul niveau

qui est toujours entièrement physiquement présent sur le GPU. Pour de très grands volumes de données, la *MRPD* devient elle-même trop grande pour tenir en mémoire GPU. Le principe de virtualisation qu'elle implémente peut alors être appliqué à elle-même. Ainsi, nous pouvons avoir N niveaux intermédiaires de table de paginaton, en dessous de la *MRPD*. Ces niveaux sont alors des caches, chacun contenant des entrées de table de pagination (des pages), organisées en blocs appelés *bloc PT*, dont chacun adresse un ensemble contiguë de briques dans l'espace. Le fait d'ajouter des niveaux intermédiaires augmente la quantité de virtualisation et permet ainsi de réduire la taille de la *MRPD* et par extension, d'adresser de plus gros volumes. Finalement, dans le dernier niveau de la hiérarchie, on retrouve le cache de brique destiné à contenir les briques de voxels.

La compléxité théorique de l'accès aux données dans une telle structure est $\mathcal{O}(\log_c(b))$, avec b, le nombre de briques au dernier niveau et la base logarithmique c, qui correspond à la taille des *blocs PT*. En pratique, seuls deux niveaux intermédiaires de virtualisation permettent d'adresser une très grande quantité de données (de l'ordre de plusieurs pétaoctets). Cette structure peut être vue comme un arbre, mais la profondeur de celui-ci est très limitée par rapport à une véritable structure d'arbre, comme l'utilisation d'un octree pour gérer la représentation d'un grand volume multi-résolution briqué. De plus, contrairement à un octree, le temps d'accès à une brique n'augmente pas en fonction de son niveau de résolution.

Structure d'arbre et table de pagination

Dans notre contexte, une structure d'arbre peut être vue comme une table de pagination et la structure de table de pagination utilisée ici peut être vue comme un arbre. Les deux structures ont le même rôle, mais des représentations et des fonctionnements différents. En revanche, la structure proposée ici permet une meilleure mise à l'échelle pour la gestion de très grands volumes et permet de garantir une meilleure qualité de rendu pour un accès aux données plus rapide (comme nous l'avons vu dans la section 2.4.2). Elle a également l'avantage de ne pas avoir à construire une structure d'arbre au préalable et de ne pas avoir à maintenir cette structure avec une gestion out-of-core (transfert des noeuds des sous-arbres) nécessaire pour de très grands arbres. Enfin, contrairement à une structure d'arbre classique utilisée pour adresser de grands volumes multi-résolution, la profondeur de la structure proposée ne dépend pas du nombre de niveaux de résolution dans la représentation multi-résolution du volume.

Niveau de résolution vs niveau de virtualisation

Afin de faciliter la compréhension, il est important de noter qu'il ne faut pas confondre la structure que l'on décrit ici et la représentation multi-résolution du volume ; même si celles-ci sont liées, du fait que la première intègre la représentation de la deuxième. Les deux peuvent être vues comme une structure pyramidale, mais nous manipulons deux concepts différents de "niveaux". La représentation multi-résolution utilise plusieurs niveaux de détails du volume, que l'on note L_i. La structure d'adressage, quant à elle, en plus du cache de briques qu'elle intègre, est également composée de plusieurs niveaux qui sont des niveaux de virtualisation. Nous les appellons *MRPD* pour le premier, puis PT_n pour les suivants. Enfin, pour résumer la hiérarchie de notre structure et reprendre tous les termes associés : le premier niveau appelé *MRPD* est une table de pagination permettant d'accéder à tous les voxels du volume multi-résolution. Il est le seul entièrement physiquement présent en mémoire GPU. Tous les autres niveaux de la hiérarchie sont des caches. Le dernier niveau est un cache de voxels stockés sous forme de blocs 3D appelés briques. Il peut également

y avoir N niveaux intermédiaires entre la *MRPD* et le cache de brique. Ces niveaux sont également des caches (appelés *caches PT*) mais qui eux, contiennent des pages, stockées par blocs 3D appelés *blocs PT*, dont chacun adresse un ensemble contiguë de briques dans l'espace.

Dans la suite de ce manuscrit, nous faisons souvent référence à la fois à une brique et à un *bloc PT* par le terme plus générique de *bloc*. Cela représente l'élément de base dans un cache, peu importe que ce soit le cache de brique ou un *cache PT*. Cela nous permet d'adresser les deux concepts en même temps pour des actions qui peuvent être faites aussi bien sur les uns que sur les autres.

3.5.2 Implémentation GPU

Notre structure d'adressage virtuel est entièrement implémentée sur GPU. Chaque niveau de la hiérarchie est stocké dans une texture 3D avec un accès aux éléments par un mode indexé, direct, sans conversion. Cependant pour le cache de briques, il est possible d'avoir un accès flottant normalisé aux éléments de la texture, selon les besoins de l'application. En effet, un tel mode d'accès est par exemple nécessaire pour une application de rendu volumique direct afin de profiter de l'interpolation trilinéaire matérielle proposée par ce type de texture. Alors que le cache de briques contient des voxels, tous les autres niveaux contiennent des entrées de la table de pagination. Une page est représentée par seulement quatre entiers, stockés ensemble avec le type vectorisé *uint4* de CUDA permettant ainsi des accès optimisés. Les trois premiers entiers d'une page nous servent à stocker l'adresse 3D du début du bloc (bloc PT ou brique) sur lequel elle pointe dans le niveau en dessous le sien. Cette adresse correspond à des coordonnées 3D dans une texture. Le quatrième entier est un "flag" permettant de décrire le statut du bloc en question. Ce statut peut prendre trois états différents, *présent*, *non-présent* ou *vide*, selon que ce bloc est présent ou non dans le niveau suivant (dans le cache) ou que ce bloc est vide. La figure 3.6 illustre la représentation mémoire d'une telle page.

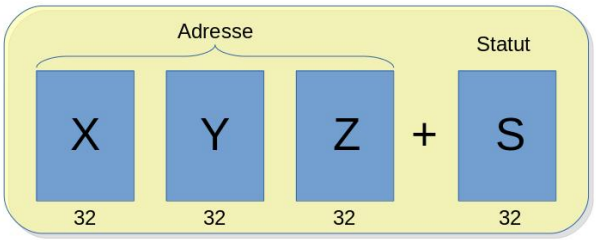


FIGURE 3.6 – **Représentation d'une entrée de notre table de pagination.** Une page est composée de trois entiers 32 bits correspondants à l'adresse de la position 3D entière du début du bloc qu'elle adresse dans le niveau en dessous dans la hiérarchie + un "flag" indiquant le statut de ce bloc : *présent*, *non-présent* ou *vide*

LRUs

Chaque cache (c'est à dire tous les niveaux de notre structure sauf la *MRPD*) est géré par une LRU. Chacune étant implémentée comme un vecteur sur le GPU, qui contient

autant d'entrées qu'il y a d'éléments (de blocs) dans le cache correspondant. Une entrée de la LRU stocke trois informations qui nous sont utiles pour faire fonctionner les mécanismes internes de mise à jour de notre structure (voir section 3.7). On y retrouve d'abord la position 3D du début du bloc dans le cache, stockée avec un vecteur de trois entiers 32 bits. Il y est aussi stocké une paire (l, p) correspondant au niveau de résolution et la position 3D dans le volume normalisé (voir section 3.6). Il est possible de stocker une telle information pour n'importe quel bloc, car ceux-ci ont bien une correspondance spatiale dans le volume, soit juste une brique (pour le cache de brique), soit un ensemble de briques (pour un *cache PT*). Cette information est stockée avec un entier 32 bits pour l et un vecteur de 3 flottants 32 bits pour la position p.

Niveaux de virtualisation et granularité d'adressage

Notre implémentation de la hiérarchie a été faite de manière générique afin de pouvoir créer dynamiquement autant de *caches PT* intermédiaires que nécessaire, pour adresser n'importe quelle taille de volume. Comme nous l'avons déjà vu, en pratique, seuls deux de ces niveaux de virtualisation intermédiaires permettent d'adresser plusieurs pétaoctet de données. Cependant, le fait de pouvoir ajouter plus de niveaux de virtualisation permet d'adresser plus de briques et donc d'utiliser des briques plus petites pour un même volume, ayant ainsi un adressage plus fin des données. Cette granularité peut être particulièrement utile pour certains types d'applications.

Mise à jour des textures

Pendant longtemps, la mémoire texture n'était pas faite pour des opérations d'écritures depuis les threads dans un kernel CUDA. Depuis quelques générations de micro-architecture, il est maintenant possible d'utiliser l'API *surface* offrant la possibilité de lire et d'écrire dans une texture 3D depuis l'exécution parallèle d'un kernel. Nous utilisons cette fonctionnalité pour manipuler l'ensemble de nos textures afin de pouvoir mettre celles-ci à jour, aussi bien pour l'écriture des briques et des *blocs PT*, respectivement dans le cache de briques et dans les caches PT_n, ainsi que pour mettre à jour une page dans n'importe quel niveau de la table de pagination. Nous utilisons également l'API *object* plus récente, qui offre une meilleure gestion des textures en les manipulant comme des pointeurs et qui présente de bien meilleures performances dans un contexte multi-GPUs, comparée à l'ancienne API *reference* [2].

Représentation mémoire des caches et de la *MRPD*

Excepté le premier niveau de la hiérarchie, tous les autres niveaux sont des caches. Ils sont représentés comme une grande zone mémoire 3D qui comporte des blocs 3D (briques ou *blocs PT*) les uns à coté des autres, sans cohérence particulière les uns par rapport aux autres. En revanche, à l'intérieur de ces blocs, on retrouve une cohérence spatiale des voxels dans les briques et des pages dans les *blocs PT*. Dans la table de pagination, (dans la *MRPD* ou dans les *blocs PT* des caches intermédiaires) les entrées sont organisées de manière régulière tout comme la représentation de la grille 3D multi-résolution de briques qu'elles adressent. Une entrée correspond spatialement à l'ensemble qu'elle virtualise. Dans le niveau juste au-dessus du cache de brique (que ce soit la *MRPD* ou un *cache PT*), chaque entrée adresse une et une seule brique du volume multi-résolution (présente ou non dans le cache de briques). La *MRPD*, quant à elle, est représentée en mémoire de la manière

2. https://devblogs.nvidia.com/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility/

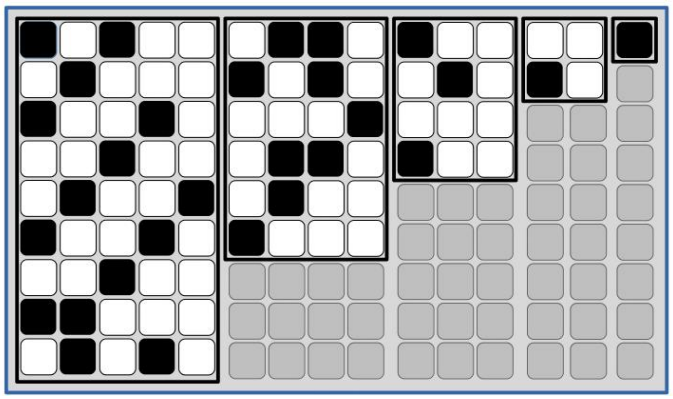


FIGURE 3.7 – **Représentation mémoire du répertoire de pagination multi-résolution.** Cette figure est une illustration en 2D de la texture (normalement en 3D) qui contient la *MRPD*. Il contient ici 5 sous-grilles pour adresser un volume avec 5 niveaux de résolution. Les cellules grises ne sont pas utilisées. Les cellules noires représentent les entrées avec le statut *présent* et les cellules blanches, celles avec le statut *non-présent*.

suivante : c'est également une grande zone mémoire 3D, qui elle, peut être décomposée en autant de sous-grilles 3D qu'il y a de niveaux de détails dans la représentation multi-résolution du volume. Ces sous-grilles sont positionnées les unes à coté des autres sur l'axe des X. L'intérieur d'une sous-grille est composé de pages qui possèdent une cohérence spatiale, identique à celle des briques ou des *blocs PT*. La figure 3.7 illustre un exemple de *MRPD* pour un volume avec cinq niveaux de résolution. La dimension de la texture 3D de la *MRPD* est égale à la somme des dimensions des sous-grilles sur l'axe des X, et la dimension de la plus grande sous-grille sur l'axe des Y et l'axe des Z. Cette représentation implique qu'une partie de la texture allouée ne serve pas (comme on peut le voir avec les cases grises de la figure 3.7). Une possibilité pour remédier à ce problème serait d'allouer autant de textures 3D que de niveaux de résolution. Chaque sous-grille serait alors stockée dans sa propre zone mémoire texture. Cette solution serait en revanche sûrement moins efficace du fait que le GPU passerait du temps à "changer" de texture et arriverait sûrement moins à optimiser les schémas d'accès dans ses caches dédiés.

3.6 Adressage virtuel

Coté applicatif, sur le GPU, la navigation dans le volume multi-résolution se fait dans un volume normalisé. Pour accéder à un voxel à partir de celui-ci, on construit une adresse (l, p) avec l le niveau de résolution et $p \equiv [x, y, z] \in [0, 1[^3$, la position flottante normalisée (voir Figure 3.8). Cette navigation n'a pas besoin de se faire dans une représentation multi-résolution du volume. Le fait que la position p soit une position normalisée (qui ne dépend donc pas des dimensions du volume), combinée au niveau de détail désiré l permet de gérer cela implicitement. La représentation virtuelle utilisée est donc un unique volume normalisé qui permet de faire le lien avec n'importe quel niveau de résolution.

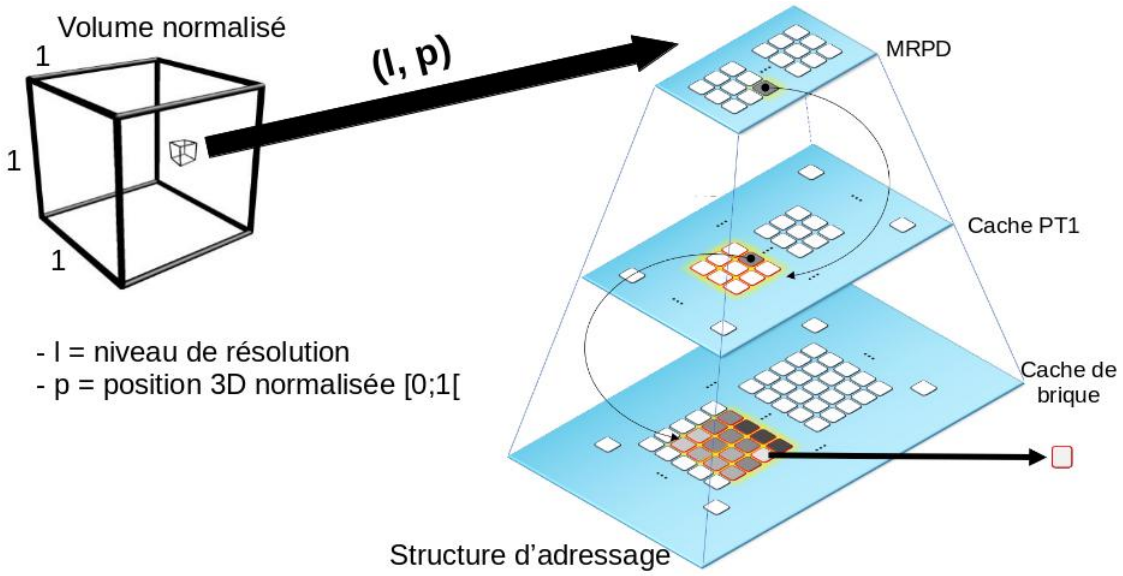


FIGURE 3.8 – **Navigation dans un volume normalisé pour l'adressage des voxels** Illustration de l'accès à un voxel via notre structure d'adressage pendant la navigation, dans un volume normalisé, sur GPU.

Le volume normalisé décrit ici est un concept différent de la représentation virtuelle dont nous allons discuter dans la suite de cette section. Alors que le premier est utilisé par l'application pour naviguer dans le volume multi-résolution, la deuxième est utilisée par le mécanisme interne de notre structure de données pour l'adressage out-of-core.

3.6.1 Représentation réelle et représentation virtuelle

La table de pagination employée pour l'adressage out-of-core du volume multi-résolution, utilise un système de virtualisation de ce volume. Nous avons donc une représentation réelle du volume et une représentation virtuelle de celui-ci, créée et utilisée par notre système interne d'adressage. Pour un niveau de virtualisation PT_n donné, tous les *blocs PT* possèdent les mêmes dimensions, notées $b_n \in \mathbb{N}^3$ (par exemple 16^3). En revanche, les *blocs PT* d'un autre niveau peuvent avoir des dimensions différentes (par exemple 32^3). Le choix de la taille des *blocs PT* pour chaque niveau de virtualisation peut avoir des conséquences sur les dimensions de la représentation virtuelle du volume à ce niveau.

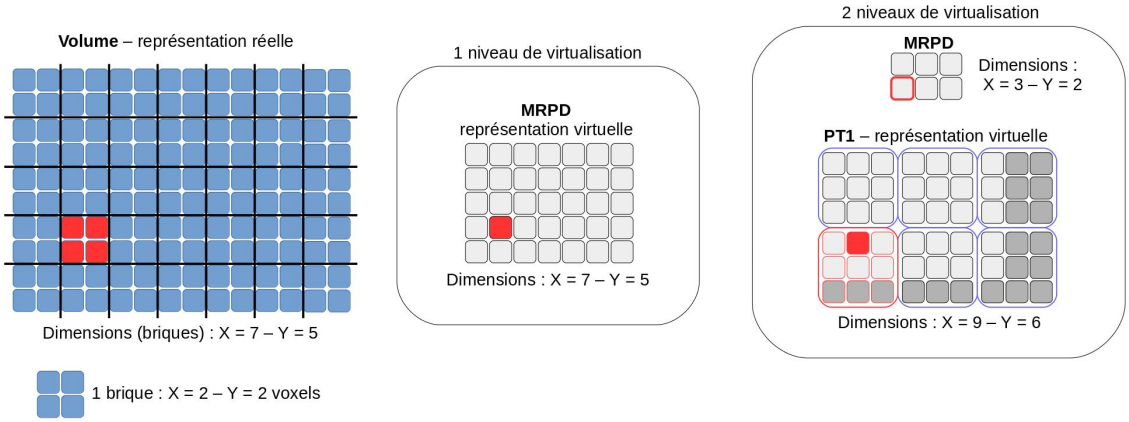


FIGURE 3.9 – **Exemple en 2D d'un volume et ses représentations virtuelles.** Cet exemple illustre la représentation réelle d'un volume avec un seul niveau de résolution et ses représentations virtuelles dans deux cas de figure : 1) avec un seul niveau de virtualisation et 2) avec deux niveaux de virtualisation et des *blocs PT* de dimensions 3×3. Les dimensions virtuelles du volume sont plus grandes que ses dimensions réelles dans le deuxième cas de figure.

En particulier, cela pourra engendrer des dimensions plus grandes dans la représentation virtuelle du volume multi-résolution, que ses dimensions réelles (voir les cases grisées sur la figure 3.9). Cela n'implique pas de surcharge de stockage mémoire puisqu'on parle ici de représentations virtuelles.

Nous noterons respectivement $r[l]_n$ et $v[l]_n$ (tous deux appartenant à l'ensemble $\mathbb{N}^3$) les tailles réelles et les tailles virtuelles d'un volume. Celles-ci dépendent du niveau de résolution l et du niveau de virtualisation n. Afin de calculer ces tailles, nous utilisons les deux équations suivantes [3] [4] [5] :

$$r[l]_n = \lceil r[l]_{n+1} \oslash b_{n+1} \rceil \tag{3.1}$$

et

$$v[l]_n = \begin{cases} r[l]_n & \text{if } n = 0 \\ v[l]_{n-1} \odot b_n & \text{if } n \geq 1 \end{cases} \tag{3.2}$$

En partant de la dimension en voxels $r[l]_{n_{max}}$ du volume et en utilisant itérativement l'équation (3.1) avec la taille des blocs à chaque niveau de virtualisation n, on obtient la taille de la *MRPD* $r[l]_0$. On peut ensuite obtenir la dimension virtuelle de chaque niveau de virtualisation n en utilisant itérativement l'équation (3.2) à partir de la dimension virtuelle de la MRPD, $v[l]_0 = r[l]_0$, jusqu'à la dimension en voxels $v[l]_{n_{max}}$ du volume.

3. L'opérateur $\oslash$ est la division vectorielle, élément par élément, ainsi, $[x, y, z] \oslash [a, b, c] = [\frac{x}{a}, \frac{y}{b}, \frac{z}{c}]$.
4. L'opérateur $\lceil\ \rceil$ est l'opération vectorielle d'arrondi à l'entier supérieur, élément par élément, ainsi, $\lceil [x, y, z] \rceil = [\lceil x \rceil, \lceil y \rceil, \lceil z \rceil]$.
5. L'opérateur $\odot$ est la multiplication vectorielle, élément par élément, ainsi, $[x, y, z] \odot [a, b, c] = [xa, yb, zc]$.

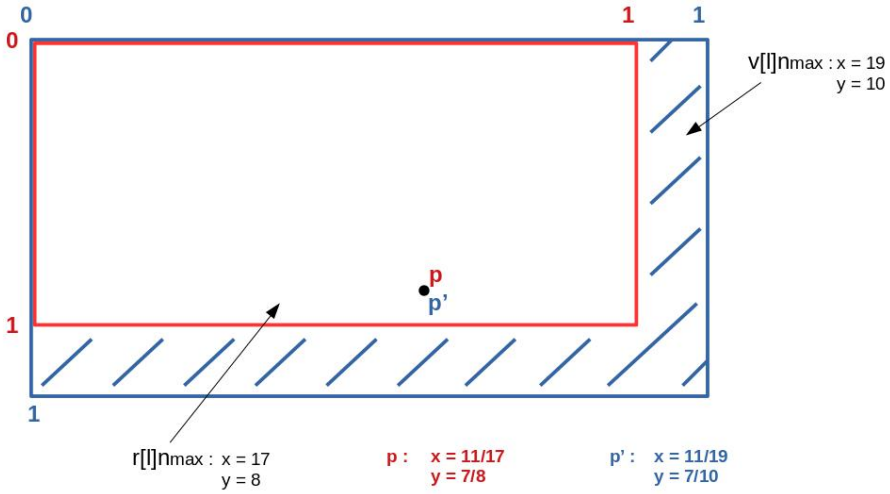


FIGURE 3.10 – **Représentation 2D de la navigation dans le volume normalisé.** La zone rouge correspond au volume normalisé dans lequel l'application navigue pour déterminer une adresse 3D flottante p. Celui-ci est calqué sur les dimensions réelles du volume $r[l]_{n_{max}}$. La zone bleue correspond à la représentation virtuelle du volume, utilisée par notre structure d'adressage out-of-core et calquée sur les dimensions virtuelles $v[l]_{n_{max}}$. C'est dans cette zone que l'on exprime la position normalisée p'. Graphiquement, dans ces deux volumes normalisés, la position p et la position p' pointent sur le même point. Cependant leurs valeurs peuvent être différentes si $v[l]_{n_{max}} \neq r[l]_{n_{max}}$. C'est le cas ici dans cet exemple.

Position virtuelle

La position normalisée $p \equiv [x, y, z] \in [0, 1[^3$ vue au début de cette section, est exprimée par l'application dans le volume normalisé calqué sur la représentation réelle du volume. Notre structure interne d'adressage out-of-core, elle, utilise une position normalisée dans la représentation virtuelle du volume (voir figure 3.10). En notant p la première et p' la deuxième, nous pouvons alors utiliser l'équation suivante afin d'effectuer ce changement de repère :

$$p' = p \odot (r[l]_{n_{max}} \oslash v[l]_{n_{max}})$$

3.6.2 Accès aux voxels

Nous allons maintenant décrire la manière d'accéder à n'importe quel voxel du volume multi-résolution en utilisant notre système d'adressage virtuel. Il est nécessaire pour cela, de parcourir la structure de données en partant de la racine (la *MRPD*), jusqu'à sa base (le cache de brique). Comme nous l'avons vu dans la section 3.5, en pratique, cette structure contient peu de niveaux. Généralement, seulement la *MRPD* et le cache de briques et éventuellement un, voir deux, niveaux de *caches PT* intermédiaires pour de très grands volumes. Cependant nous donnons ici la méthode de parcours général applicable pour n'importe quel nombre de niveaux de virtualisation.

Le parcours de la structure consiste pour chaque niveau, à calculer l'adresse 3D de

l'entrée dans le niveau suivant. A chaque fois que l'on recupère une telle entrée, on continue le parcours uniquement si le "flag" de cette entrée possède le statut *présent*, jusqu'à arriver au voxel désiré dans le cache de brique. Le parcours commence à partir de l'adresse (l, p') d'un voxel, établie dans le volume normalisé. A partir de celle-ci, on peut obtenir la position 3D de l'entrée correspondante dans la *MRPD* en utilisant l'équation suivante [6] :

$$PDentry(l, p') = PageDirBase[l] + \lfloor p' \odot v[l]_0 \rfloor \in \mathbb{N}^3 \tag{3.3}$$

Nous utilisons un tableau à une dimension noté $PageDirBase[l]$, qui permet de stocker les positions 3D du début de chaque sous-grille, correspondant à chaque niveau de résolution l du volume, dans la *MRPD*. Le premier terme de l'équation (3.3) permet alors de se déplacer dans la texture jusqu'à la sous-grille adressant le niveau l, alors que le deuxième terme permet de se déplacer dans cette sous-grille selon la position virtuelle $p' \equiv [x, y, z] \in [0, 1[^3$ du voxel et la dimension de la sous-grille.

Pour la suite du parcours, nous pouvons récupérer à chaque étape, l'adresse stockée dans l'entrée courante. Celle-ci est stockée sous forme de coordonnées 3D c. A l'instar du tableau $PageDirBase[l]$, ces coordonnées permettent d'accéder au début du bloc contenant l'entrée recherchée dans le niveau suivant, pour tous les niveaux de cache. On les note $Cbase_n(l, p')$ et elles se récupèrent avec l'équation suivante :

$$Cbase_n(l, p') = \begin{cases} \text{MRPD}[PDentry(l, p')].c & \text{si } n = 1 \\ \text{PTCache}_{n-1}[Centry_{n-1}(l, p')].c & \text{si } n > 1 \end{cases} \tag{3.4}$$

Alors que l'on note l'entrée dans la *MRPD*, $PDentry(l, p')$, nous noterons les entrées dans tous les niveaux de caches (*caches PT* ou cache de brique) $Centry_n(l, p')$. Celles-ci, en plus du niveau de résolution l et de la position p', dépendent du niveau de virtualisation n. Nous utilisons l'équation suivante pour y accéder [7] :

$$Centry_n(l, p') = Cbase_n(l, p') + \lfloor p' \odot v[l]_n \rfloor \bmod b_n \in \mathbb{N}^3 \tag{3.5}$$

La valeur n correspond au niveau de virtualisation, depuis $n = 0$, correspondant à la *MRPD*, jusqu'à $n = n_{max}$, correspondant au cache de brique, en passant par tous les niveaux de caches intermédiaires ($0 < n < n_{max}$). Dans ce contexte, l'équation (3.3) traite le cas où $n = 0$ et l'équation (3.5) gère tous les autres cas, de $n = 1$ à $n = n_{max}$. Pour rappel, dans l'équation (3.5), $v[l]_n$ et b_n sont respectivement :

- la taille du volume en voxel et la taille d'une brique – Si $n = n_{max}$ (cache de brique),
- la taille du volume, au niveau de résolution l, dans sa représentation virtuelle au niveau de virtualisation n et la taille d'un *bloc PT* dans le *cache PT* associé – Si $0 < n < n_{max}$ (niveaux intermédiaires de virtualisation),
- la taille du volume au niveau 0 ($v[l]_0 = r[l]_0$) et une valeur arbitraire pour b_0 car les entrées de la MRPD ne sont pas stockées par blocs – Si $n = 0$ (MRPD).

6. L'opérateur $\lfloor\ \rfloor$ est l'opération vectorielle de troncature à l'entier inférieur, élément par élément, ainsi, $\lfloor [x, y, z] \rfloor = [\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor]$.

7. L'opérateur modulo est définie sur les vecteurs comme $[x, y, z] \bmod [a, b, c] = [x \bmod a, y \bmod b, z \bmod c]$.

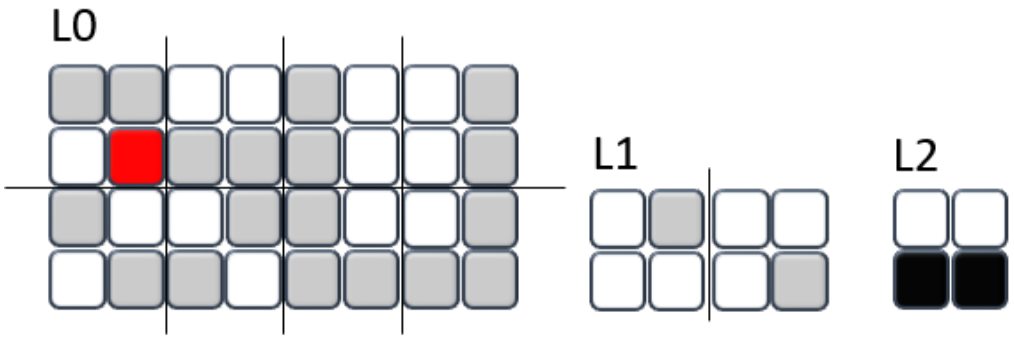


FIGURE 3.11 – **Exemple d'adressage virtuel d'un voxel dans un volume multi-résolution.** Illustration 2D d'un volume avec trois niveaux de résolution, découpés en briques de taille 2×2 sans chevauchement de voxels. Il contient 8×4 voxels à la plus haute résolution ($L0$), 4×2 voxels au niveau de résolution intermédiaire ($L1$) et 2×1 voxels à la plus basse résolution ($L2$), contenus dans une seule brique complétée par des voxels "vides" illustrés ici en noir.

Il est important de noter que, même si p' est exprimé avec des valeurs réelles ($p' \in \mathbb{R}^3$), les résultats des équations (3.3) et (3.5) sont des vecteurs entiers. En effet, les accès aux éléments dans ces textures se font par indexation avec des valeurs entières. Il y a tout de même une exception dans le cas où le cache de brique est implémenté avec une texture à accès flottant normalisé. Dans ce cas, pour l'accès à ce cache, l'utilisation de l'équation (3.5) se fait sans l'opérateur d'arrondi $\lfloor\ \rfloor$.

Prise en compte du chevauchement

Comme nous l'avons déjà évoqué dans la section 3.4.2 le découpage du volume en brique peut se faire en comprenant un chevauchement de voxels entre des briques voisines. Si c'est le cas, cet aspect doit être pris en compte pour la recherche d'un voxel dans le cache de brique (pour $n = n_{max}$) avec l'équation (3.5). Pour cela, il faut tenir compte de deux informations. Premièrement, la taille utile des briques, $b_{n_{max}}$, correspond à la taille de l'espace d'adressage interne sans chevauchement. Elle doit être égale à la taille de stockage de la brique dont on soustrait la taille du chevauchement $ch \in \mathbb{N}^3$, de part et d'autre de la brique. Par exemple, pour des briques de 32^3 voxels et un chevauchement de taille 1 sur chacune des trois dimensions, $b_{n_{max}} = (32 - 1 \times 2)^3 = 30^3$. Deuxièmement, les coordonnées c données par $Cbase_{n_{max}}(l, p')$ et amenant à la position 3D dans le cache, correspondent au début du stockage de la brique avec chevauchement. Pour accéder au début de la zone utile, il convient d'ajouter le chevauchement à $Cbase_{n_{max}}(l, p')$ dans l'équation (3.5). On obtient ainsi l'équation suivante :

$$Centry_n(l, p') = \begin{cases} Cbase_n(l, p') + \lfloor p' \odot v[l]_n \rfloor \bmod b_n \in \mathbb{N}^3 & \text{si } n < n_{max} \\ Cbase_n(l, p') + ch + \lfloor p' \odot v[l]_n \rfloor \bmod b_n \in \mathbb{N}^3 & \text{si } n = n_{max} \end{cases} \tag{3.6}$$

Exemples d'utilisation

Afin d'illustrer l'utilisation des équations précédentes, supposons la représentation 2D d'un volume, présenté sur la figure 3.11. Celui-ci possède trois niveaux de résolution et

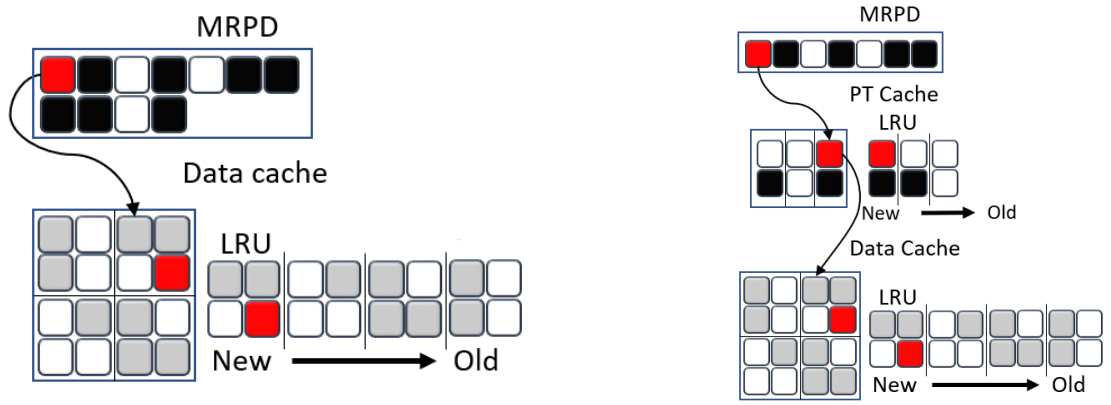


FIGURE 3.12 – **Exemples 2D de structure d'adressage pour le volume de la figure 3.11.** Cette figure sert d'exemple de structures d'adressage virtuel, utilisées pour l'accès aux voxels du volume multi-résolution de la figure 3.11. (a) Structure avec un seul niveau de virtualisation. Une entrée de la *MRPD* adresse une brique. (b) Structure avec deux niveaux de virtualisation. Une entrée de la *MRPD* adresse un *bloc PT* de 1×2 entrées dans le *cache PT* intermédiaire. Une entrée de ce cache adresse une brique dans le cache de brique. Dans les deux cas (a) et (b), on retrouve un cache de brique à accès direct indexé, d'une taille de 2×2 et la LRU associée à ce cache. Les cases noires représentent les entrées qui possèdent un "flag" avec le statut *non-présent*, et les cases blanches, celles avec le statut *présent*.

est découpé en briques de 2×2 voxels. Nous allons illustrer l'accès au voxel marqué en rouge sur cette figure, par deux configurations de virtualisation différentes (illustrées sur la figure 3.12). Dans les deux cas, le voxel auquel nous souhaitons accéder est présent en cache. Pour ces exemples, nous choisissons volontairement de restreindre le plan $z = 0$.

La première configuration, représentée sur la figure 3.12a ne contient qu'un seul niveau de virtualisation, la *MRPD*. Le cache de données, quant à lui contient 2×2 briques et est figé dans un certain état pour l'exemple. La table 3.1 donne le contenu du tableau $PageDirBase[l]$ associé à cette configuration. La table 3.2, donne les valeurs des dimensions réelles et virtuelles du volume pour chaque niveau de virtualisation n et chaque niveau de résolution l. Pour accéder au voxel rouge, nous utilisons en premier l'équation (3.3) de la manière suivante :

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

L'entrée corespondante dans la *MRPD* contient les coordonnées du début du bloc qu'elle adresse dans le niveau suivant. Ici, ce bloc est une brique dans le cache de brique. Dans notre exemple, ces coordonnées sont $MRPD[0,0,0].c = [2,0,0]$. Cette entrée contient également un "flag" qui possède ici le statut *présent*. Il est maintenant possible de continuer le parcours en calculant les coordonnées du voxel dans le cache de brique :

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

En accédant aux coordonnées $[x, y, z] = [3, 1, 0]$ dans le cache de brique, nous pouvons lire la valeur du voxel en question.

Considérons maintenant la configuration de la hiérarchie de virtualisation donnée sur la figure 3.12b par deux niveaux de virtualisation. En utilisant la table 3.3 pour les valeurs de $v[l]_n$ et de b_n, ainsi que la table 3.1, également valable dans cette configuration pour le contenu du tableau $PageDirBase[l]$. Nous pouvons alors calculer :

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

Les coordonnées contenues dans l'entrée sont $MRPD[0, 0, 0].c = [2, 0, 0]$. Le "flag" de l'entrée possède le statut *présent*. Ainsi :

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

Les coordonnées contenues dans l'entrée sont $PTCache_1[2, 0, 0].c = [2, 0, 0]$. Le "flag" de l'entrée possède le statut *présent*. Ainsi :

$$Centry_2(0, \left[\frac{1}{8}, \frac{1}{4}, \frac{0}{1}\right]) = \begin{cases} x = 2 + \lfloor \frac{1}{8} \times 8 \rfloor \bmod 2 = 3 \\ y = 0 + \lfloor \frac{1}{4} \times 4 \rfloor \bmod 2 = 1 \\ z = 0 + \lfloor \frac{0}{1} \times 1 \rfloor \bmod 1 = 0 \end{cases}$$

Nous retrouvons bien les coordonnées $[x, y, z] = [3, 1, 0]$ dans le cache de brique pour accéder au voxel rouge.

LOD	**PageDirBase**
L0	[0, 0, 0]
L1	[4, 0, 0]
L2	[6, 0, 0]

TABLE 3.1 – **Coordonnées 3D des sous-grilles des *MRPD* décrites sur la figure 3.12.**

		$n = 1 = n_{max}$	$n = 0$	$n = 1 = n_{max}$	
	Vol. voxels	**Cache brique**	**MRPD**	**Cache brique**	**Vol. voxels**
$L0$	$[8, 4, 1]$	$[8, 4, 1]$	$[4, 2, 1]$	$[8, 4, 1]$	$[8, 4, 1]$
$L1$	$[4, 2, 1]$	$[4, 2, 1]$	$[2, 1, 1]$	$[4, 2, 1]$	$[4, 2, 1]$
$L2$	$[2, 2, 1]$	$[2, 2, 1]$	$[1, 1, 1]$	$[2, 2, 1]$	$[2, 2, 1]$
b_n	–	$b_1 = [2, 2, 1]$	–	$b_1 = [2, 2, 1]$	–

$r[l]_n$ $v[l]_n$

TABLE 3.2 – **Dimensions réelles et virtuelles des différents niveaux dans le cas de l'exemple décrit sur les figures 3.11 et 3.12a.**

		$n = 2$ $= n_{max}$	$n = 1$	$n = 0$	$n = 1$	$n = 2$ $= n_{max}$	
	Vol. voxels	**Cache brique**	**PT 1**	**MRPD**	**PT 1**	**Cache brique**	**Vol. voxels**
$L0$	$[8, 4, 1]$	$[8, 4, 1]$	$[4, 2, 1]$	$[4, 1, 1]$	$[4, 2, 1]$	$[8, 4, 1]$	$[8, 4, 1]$
$L1$	$[4, 2, 1]$	$[4, 2, 1]$	$[2, 1, 1]$	$[2, 1, 1]$	$[2, 2, 1]$	$[4, 4, 1]$	$[4, 4, 1]$
$L2$	$[2, 2, 1]$	$[2, 2, 1]$	$[1, 1, 1]$	$[1, 1, 1]$	$[1, 2, 1]$	$[2, 4, 1]$	$[2, 4, 1]$
b_n	–	$b_2 =$ $[2, 2, 1]$	$b_1 =$ $[1, 2, 1]$	–	$b_1 =$ $[1, 2, 1]$	$b_2 =$ $[2, 2, 1]$	–

$r[l]_n$ $v[l]_n$

TABLE 3.3 – **Dimensions réelles et virtuelles des différents niveaux dans le cas de l'exemple décrit sur les figures 3.11 et 3.12b.**

3.7 Gestion de l'utilisation de la structure

Nous proposons de gérer l'ensemble de la structure proposée avec une implémentation entièrement sur GPU. Cette gestion comprend la mise à jour de l'utilisation des caches et donc des LRUs associées, ainsi que la mise à jour de la hiérarchie de table de pagination.

3.7.1 Mise à jour de la hiérarchie

La mise à jour de la structure hiérarchique est nécessaire lorsqu'une nouvelle brique doit être ajoutée dans le cache de brique GPU. Si ce cache est plein au moment d'un tel ajout, la nouvelle brique entrante doit alors prendre la place d'une brique déjà présente dans le cache. La brique à retirer est choisie par rapport à sa position dans la LRU associée au cache de brique (voir section 3.7.2). La mise à jour de la structure est alors constituée de deux ou trois étapes séquentielles, selon que le cache soit déjà plein ou non. Si le cache est effectivement plein, il faut alors d'abord déréférencer la brique qui doit être retirée pour laisser la place à la nouvelle brique. Ensuite, dans tous les cas, vient l'écriture de la nouvelle brique dans le cache et la mise à jour de la LRU associée (voir section 3.7.2). Finalement, il faut référencer cette nouvelle brique dans la hiérarchie de table de pagination.

Déréférencement

Nous utilisons le terme de déréférencement dans le sens où l'on retire la référence, le pointage d'une entrée de la table de pagination, qui pointait sur un bloc présent dans un des niveaux de cache de la structure. Cette action a pour but d'indiquer qu'un bloc a été retiré du cache. Concrètement, cela consiste à changer le "flag" de l'entrée, du statut *présent* au statut *non-présent*. Il n'est pas nécessaire de changer l'adresse contenue dans l'entrée car elle ne sera plus utilisée tant que le statut est *non-présent*. Celle-ci sera changée lorsque l'on rebasculera le statut sur *présent*, avec les coordonnées du nouvel emplacement dans le cache, du bloc adressé par l'entrée en question (référencement).

Le déréférencement d'un bloc ne se fait que dans le niveau juste au-dessus de celui qui contient ce bloc. Un bloc doit être déréférencé lorsqu'il est retiré d'un cache. Un bloc doit être retiré d'un cache dans le cas où l'on doit ajouter un nouveau bloc dans ce cache et que celui-ci est déjà plein. Dans ce cas, le bloc à retirer est celui qui a été utilisé dans le cache, il y a le plus longtemps par l'application. Celui-ci est donné par la dernière entrée de la LRU du cache correspondant. Comme nous l'avons déjà vu (voir section 3.5.2) l'entrée de la LRU contient l'adresse (l, p) du bloc dans le volume. En utilisant les équations (3.3), (3.4) et (3.5) avec cette adresse, nous pouvons retrouver l'entrée correspondante dans la table de pagination, dans le niveau juste au dessus du cache contenant ce bloc. Il est alors possible de changer le "flag" de cette entrée. A ce moment là, le bloc n'est plus considéré comme présent dans le cache GPU.

Référencement

L'action de référencement consiste à mettre à jour les entrées, à tous les niveaux d'adressage, correspondantes à un bloc qui vient d'etre ajouté dans un cache. Cela permet de faire le lien avec ce nouveau bloc depuis la racine de la structure. Ce référencement revient à changer le "flag" des entrées, du statut *non-présent* au statut *présent* et à mettre à jour l'adresse qu'elles contiennent avec les coordonnées du bloc dans le niveau suivant. Il est

possible que le bloc ajouté fasse lui-même partie d'un bloc, dans un des niveaux au dessus de lui, qui est déjà référencé. A partir de l'adresse (l, p) du bloc et des équations de parcours de la structure, il est alors nécessaire de réaliser un premier parcours, de haut en bas, pour trouver le dernier niveau auquel le bloc est référencé. On peut ensuite compléter le référencement en procédant de bas en haut, à partir du niveau juste au-dessus du cache où a été ajouté le bloc, jusqu'au niveau indiqué par le premier parcours.

Ajout d'une brique et ajout d'un nouveau *bloc PT*

L'ajout d'une nouvelle brique dans le cache de brique se traduit par une écriture des valeurs des voxels qu'elle contient. Lors de l'ajout d'une nouvelle brique, à l'étape de référencement de celle-ci, il est possible de devoir ajouter également un *bloc PT* dans un niveau de *cache PT*. Cet ajout consiste plutôt en une réinitialisation de toutes les entrées du bloc dans le cache avec le statut *non-présent* et une adresse $[0, 0, 0]$. Après cette réinitialisation, nous pouvons mettre à jour l'entrée de ce *bloc PT*, correspondante à la brique pour laquelle nous l'avons ajouté, avec l'adresse de celle-ci et le statut *présent*. Si l'on ajoute un nouveau *bloc PT* au moment du référencement d'une nouvelle brique, celui-ci est référencé en même temps que la brique, cela n'implique pas de parcours supplémentaire de la structure.

Quand un nouveau *bloc PT* écrase un ancien, nous perdons alors éventuellement le lien vers toutes les briques qu'il référençait dans le cache de brique. Cela implique que certaines données peuvent encore être présentes en cache alors qu'elles ne sont plus référencées. Cependant, ces blocs vont très rapidement converger à la fin de la LRU et donc être très rapidement remplacés. Ceci, d'autant plus que si un *bloc PT* est écrasé, c'est que celui-ci se trouvait lui-même à la fin de la LRU du cache dans lequel il était. Il adressait donc des briques qui n'avaient pas été utilisées depuis longtemps et donc des briques qui elles-mêmes devaient se trouver vers la fin de la LRU correspondante.

L'approche proposée ne garantit pas la pérennité des méta-informations des pages dans la table de pagination. Les pages dans les niveaux intérmédiaires de virtualisaton (*caches PT*), sont créées à la demande et à la volée sur le GPU pour être mises en cache. Lorsqu'une telle page est "retirée" du cache, les informations qu'elle contenait sont également perdues. En particulier, le statut *vide* qui indique que la page adresse une brique (ou un ensemble de briques) vide, ne sera pas maintenue. A l'inverse, ce problème n'apparaît pas avec l'utilisation d'un octree qui utilise les noeuds de l'arbre comme pages. Ceci est l'avantage de leur inconvenient. Leur inconvénient étant que les noeuds sont gardés en mémoire CPU puis transférés à la demande sur GPU, cela a l'avantage de sauvegarder leur contenu même s'ils sont évincés du cache GPU.

Exemple de mise à jour

La figure 3.13 illustre la mise à jour des deux structures utilisées en exemple dans la section précédente et présentées par la figure 3.12. Cette figure décrit l'état de la structure (table de pagination et cache de brique) avant et après une mise à jour pour l'utilisation du voxel positionné à $p = [\frac{2}{8}, \frac{0}{4}, \frac{0}{1}]$ et au niveau de résolution $L0$ du volume exemple de la figure 3.11. Pour ajouter la nouvelle brique contenant ce voxel dans le cache, il faut en premier lieu déréférencer la brique utilisée il y a le plus longtemps par l'application. Celle-ci est donnée par la dernière entrée de la LRU du cache de brique. Il s'agit de la brique de coordonnées $[0, 0, 0]$ dans le cache de brique (voir figure 3.13(c)). En parcourant la structure avec ses coordonnées p' et son niveau de résolution l, nous pouvons retrouver

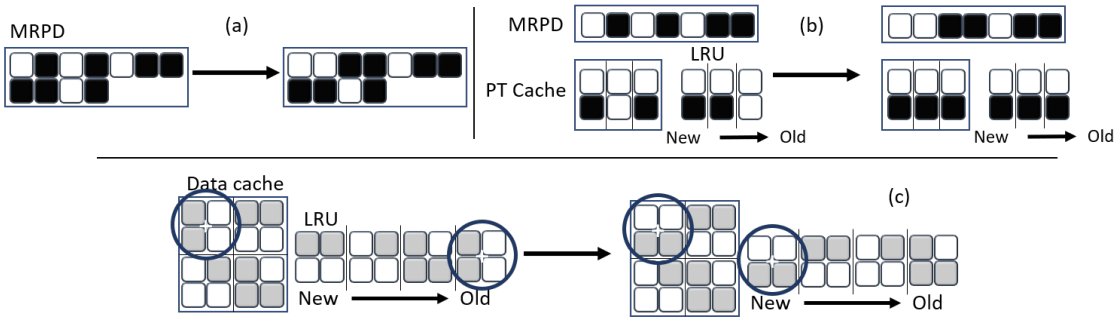


FIGURE 3.13 – **Mise à jour de la hiérarchie de table de pagination et du cache de brique.** Illustration d'un exemple de mise à jour de l'ensemble de la structure d'adressage virtuel correspondant au volume présenté sur la figure 3.11 avec (a) la structure avec un niveau de virtualisation, présentée sur la figure 3.12a, (b) la structure avec deux niveaux de virtualisations, présentée sur la figure 3.12b et (c) le cache de brique correspondant et sa LRU, illustrés sur la figure 3.12.

l'entrée correspondante dans la table de pagination, au niveau juste au-dessus du cache de brique. Dans la première configuration, cette entrée est celle à la position $[2, 0, 0]$ dans la *MRPD*. Il est alors possible de changer le "flag" de cette entrée avec le statut *non-présent* (voir figure 3.13(a)). Pour la deuxième configuration, il s'agit de l'entrée à la position $[1, 0, 0]$ dans le *cache PT* (voir figure 3.13(b)). Il est ensuite possible d'écrire le contenu de la nouvelle brique dans le cache de brique aux coodonnées de celle qui vient juste d'être déréférencée et de mettre à jour la LRU en conséquence (voir figure 3.13(c)). Finalement, avec les coordonnées et le niveau de résolution de la nouvelle brique, il est possible de référencer celle-ci dans tous les niveaux de la table de pagination. Il faut alors changer l'adresse et le flag de l'entrée à la position $[0, 1, 0]$ dans la *MRPD* pour la première configuration, et des entrées aux positions $[0, 1, 0]$ dans la *MRPD* et $[0, 1, 0]$ dans le *cache PT* pour la deuxième configuration (voir figure 3.13(a,b)). Cette dernière action dans le cas de la deuxième configuration, nécessite de remplacer un *bloc PT* dans le *cache PT* pour pouvoir adresser comme il faut la nouvelle brique. Ce cache étant déjà plein, il faut alors réinitialiser le bloc utilisé il y a le plus longtemps par l'application, donné par la LRU associé à ce niveau de cache. Il s'agit ici du bloc aux coordonnées $[1, 0, 0]$. Ce bloc est lui aussi déréférencé dans le niveau juste au-dessus, à la position $[2, 0, 0]$ dans la *MRPD*, puis remplacé avec les informations du nouveau bloc.

Implémentation GPU

Ces étapes entrant dans le processus de mise à jour sont entièrement implémentées sur le GPU avec un kernel CUDA pour chacune d'elles. En pratique, la mise à jour se fait pour l'ajout de plusieurs nouvelles briques en parallèle. Le kernel en charge du déréférencement possède autant de threads que de briques à déréférencer et effectue cette tache en une seule passe. Le kernel d'écriture des briques dans le cache possède autant de threads qu'il y a de voxels dans une brique. Il écrit chaque brique dans la texture correspondante en une seule passe en répétant l'opération pour toutes les briques dans une boucle interne au kernel. L'étape de référencement, quant à elle, s'effectue aussi sur le GPU mais avec un kernel ne possédant qu'un seul thread. Celui-ci réalise les parcours de la structure séquentiellement pour chaque nouvelle brique. Cette étape n'est pas faite en parallèle pour chaque brique, afin d'éviter des possibilités de duplication de *bloc PT*. En effet, un tel scénario peut se produire avec un système parallèle si l'on souhaite ajouter deux briques spatialement

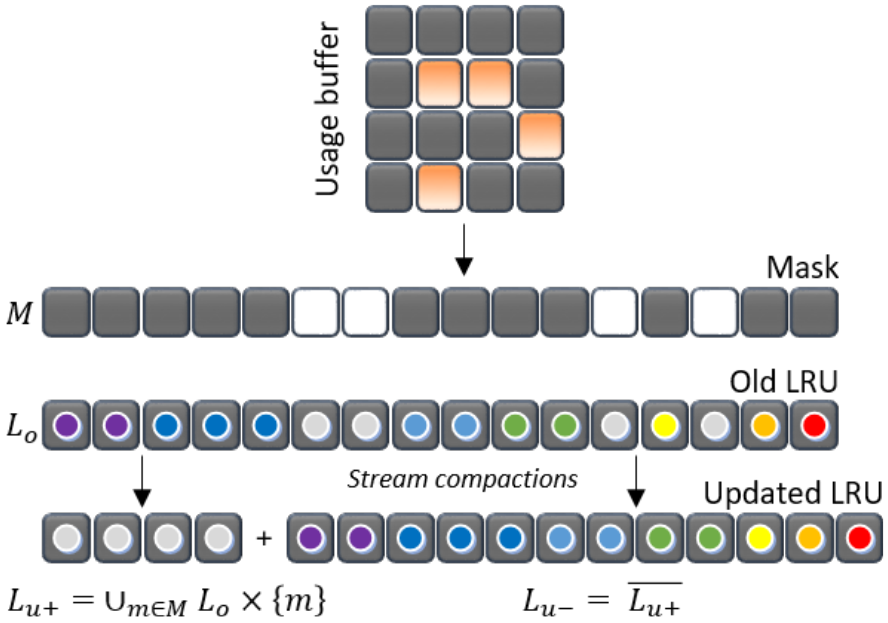


FIGURE 3.14 – **Mise à jour d'une LRU sur GPU par *stream compactions* à partir d'un *buffer d'usage*.** Cette figure contient une représentation 2D du *buffer d'usage* 3D utilisé pour marquer l'utilisation des éléments d'un cache au fur et à mesure du temps, en parallèle. A partir de celui-ci, on utilise un système de double *stream compactions* pour maintenir la LRU à jour.

proches en même temps. Dans ce cas, un algorithme parallèle pourrait ajouter deux *blocs PT* qui référenceraient alors tous les deux une même brique, alors qu'un seul de ces deux blocs aurait été nécessaire pour référencer les deux nouvelles briques en même temps.

3.7.2 Mise à jour des LRUs

Le processus de mise à jour des LRUs permet de maintenir l'ordre d'utilisation des éléments dans les caches. Ainsi, nous pouvons établir une politique de remplacement lors d'ajout de nouveaux éléments dans un cache plein. La stratégie établie par l'algorithme LRU étant de maintenir une liste triée avec les éléments les plus récemments utilisés au début, les moins récemments utilisés à la fin et de remplacer ces derniers en cas d'ajout. Nous proposons un système parallèle entièrement sur le GPU pour répondre à ce besoin (voir figure 3.14). Ce principe a été introduit dans les travaux de thèse de Cyril Crassin [Cra11] pour l'utilisation d'un octree multi-résolution comme structure d'adressage out-of-core.

Afin de reporter l'usage des éléments dans les caches, nous utilisons un système de buffer 3D en mémoire globale du GPU. Ces buffers sont nommés *buffer d'usage*. Chaque cache possède sa propre version d'un tel buffer. Ceux-si sont calqués sur le cache dont ils dépendent et possèdent une correspondance spatiale 3D exacte, avec une entrée par élément dans le cache (une brique ou un *bloc PT*). Parallèlement à cela, nous maintenons un compteur global à l'application (un timestamp) qui est incrémenté à chaque passe. Nous faisons une petite parenthèse ici pour parler d'une "passe". Pour une application

de visualisation interactive, cela peut faire référence à une itération du rendu. Pour une application de traitement des données, cela peut avoir un tout autre sens. Comme déjà discuté dans la section 3.2, c'est l'application qui se charge de décider du meilleur moment pour réaliser cette action.

Lorsqu'un thread GPU accède à un élément d'un cache, celui-ci se charge également de marquer l'entrée correspondante dans le *buffer d'usage* avec le timestamp courant. A chaque "passe", on peut alors mettre à jour les LRUs de la manière suivante : On crée d'abord un masque binaire M à partir du *buffer d'usage*. Une case de ce masque est initialisée à "vrai" si l'entrée correspondante dans le buffer possède un timestamp égale au timestamp courant et à "faux" sinon. On utilise ensuite une opération de primitive parallèle appellée *stream compaction*. Cette opération, permet, à partir d'un masque, d'une liste de même taille et d'un certain prédicat, d'obtenir en sortie tous les éléments de la liste dont les entrées correspondantes dans le masque répondaient au prédicat donné en paramètre. Nous utilisons alors en entrée, notre masque binaire M, l'ancienne LRU L_0 (tout deux de même taille), et un prédicat répondant uniquement aux cases marquées "vrai". Nous obtenons ainsi en sortie une liste L_{u+} contenant uniquement les éléments de la LRU dont les cases correspondantes dans le masque sont à "vrai", c'est à dire, ceux qui viennent d'être utilisés à la "passe" courante. En répétant cette action de *stream compaction* une deuxième fois avec le prédicat inverse, nous obtenons alors la liste L_{u-} de tous les éléments de la LRU qui n'ont pas été utilisés à la passe courante. Une concaténation de ces deux listes nous permet d'obtenir la nouvelle LRU triée dans le bon ordre. Nous obtenons ainsi les éléments venant d'être utilisés durant la dernière passe, en tête de la LRU, suivie par les autres éléments sans avoir modifié leur ordre. Nous pouvons ensuite incrémenter le timestamp.

Cette approche permet de garantir une mise à jour efficace des LRUs en parallèle. Les opérations de *stream compaction* sont réalisées avec la bibliothèque Thrust [Nvi]. Elle permettent de trier la liste en évitant un algorithme de tri classique qui serait coûteux sur GPU. Le fait d'utiliser un seul timestamp global permet de ne pas avoir à gérer les accès concurrents dans le *buffer d'usage*. En effet, si deux threads veulent reporter en même temps l'usage d'un élément du cache, il n'y a de conflit de concurrence du fait qu'ils écrivent tout les deux la même valeur. Nous utilisons un timestamp codé avec un entier non signé sur 32 bits. Pour une application de visualisation interactive offrant un rafraichissement de 30 images par secondes et réalisant une mise à jour à chaque image, cela nous permet de ne pas avoir à réinitialiser le *buffer d'usage* même si l'application tournait pendant plusieurs années. Avec un timestamp codé sur 16 bits, cela engendrerait une réinitialisation toutes les 36 minutes environ.

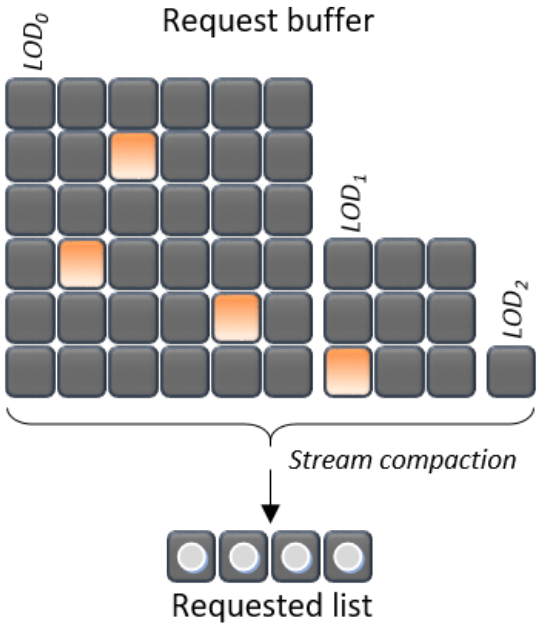


FIGURE 3.15 – **Gestion des défauts de caches en parallèle sur GPU par une opération de *stream compaction* à partir d'un *buffer de requêtes*.** Cette figure contient une représentation 2D du *buffer de requête* 3D utilisé pour marquer les défauts de cache, en parallèle. A partir de celui-ci, on utilise une opération de *stream compaction* pour créer une courte *liste de requêtes* contenant les indices des briques à acheminer jusqu'au cache GPU.

3.8 Requêtes des données

Nous présentons dans cette section, une méthode de gestion des requêtes de voxels initiées par l'application, sur le GPU, selon ses besoins. Cette approche se veut générique afin de répondre aux besoins de n'importe quel type d'application.

3.8.1 Report des défauts de cache

La gestion des requêtes de données dans notre contexte out-of-core comprend la manière de gérer les défauts de cache qui peuvent arriver sur notre cache de brique GPU. En effet, ce sont ces défauts de cache qui vont provoquer la requête des briques manquantes et déclencher l'ensemble du pipeline pour assurer le streaming de celles-ci depuis le disque ou depuis le CPU, jusqu'au cache de brique GPU.

A l'instar du *buffer d'usage* présenté dans la section précédente, nous utilisons un *buffer de requêtes* pour reporter les défauts de cache (voir figure 3.15). En revanche, contrairement au *buffer d'usage* qui existe en autant d'exemplaires qu'il y a de caches dans l'ensemble de la structure, le *buffer de requêtes* n'existe qu'en un seul exemplaire et sert uniquement aux défauts de cache propres au cache de brique. Effectivement, les pages ne font pas l'objet de requêtes dans le sens ou celles-ci sont créées directement sur le GPU. Le *buffer de requêtes* est un buffer 3D stocké en mémoire globale du GPU et accessible par tous les threads en

parallèle. Il a une correspondance spatiale exacte avec le volume multi-résolution entier, comprenant autant d'entrées qu'il y a de briques dans ce dernier. Il est représenté en mémoire de la même manière que la *MRPD* (voir section 3.5.2).

Lorsqu'un thread souhaite accéder à un voxel, il parcourt la structure pour atteindre celui-ci dans le cache de brique. Si une page, rencontrée durant ce parcours, possède un "flag" avec le statut *non-présent*, alors la brique qui contient ce voxel ne se trouve pas dans le cache GPU. Le parcours peut donc s'arrêter aussitôt et le thread va marquer la case correspondante à cette brique, dans le *buffer de requêtes* à l'aide de l'adresse (l, p) du voxel. Nous utilisons le timestamp global à l'application évoquée dans la section précédente, pour marquer les cases du *buffer de requêtes*. A la fin de chaque "passe", nous utilisons une opération de *stream compaction* simplifiée par rapport à celles utilisées pour la mise à jour des LRUs (voir section 3.7.2). Dans ce cas, l'entrée est limitée au buffer lui-même ; et le prédicat accepte uniquement les entrées dont la valeur est égale au timestamp courant. Nous obtenons en sortie, une liste contenant les indices 3D de toutes les entrées dans le *buffer de requêtes* qui viennent d'être marquées à cette "passe". Cette liste est ensuite limitée à une certaine taille N, donc éventuellement tronquée. Cela permet de répartir la quantité de données à charger, sur plusieurs "passes". Beaucoup d'approches contrôlent la taille de cette liste en limitant le nombre de requêtes par rayon (par thread). Nous utilisons plutôt une limite globale, nous permettant ainsi d'être plus efficace dans un contexte applicatif plus général (pas limité à une application de lancer de rayon volumique). La liste finale, appelée *liste de requêtes* contient donc N entrées, chacune contenant 4 entiers 32 bits vectorisés, correspondant à l'indice 3D, de la brique dans le volume et son niveau de résolution. C'est cette liste qui est envoyée au CPU pour que celui-ci gère la suite des requêtes à son niveau.

Ce buffer peut être vu comme un facteur limitant dans notre méthode dans le sens où il dépend directement de la taille de l'entrée (du volume multi-résolution). Cependant, la surcharge mémoire engendrée par ce buffer n'est que de quelques pourcents même pour de très grands volumes. Considérons par exemple un grand volume de 16384^3 voxels (soit 17.6 TB pour des voxels RGBA) découpés en briques de 64^3 voxels. Avec un ratio de sous-échantillonage de deux, entre deux niveaux de résolution consécutifs, cela génèrera un *buffer de requêtes* avec les dimensions suivantes : $(16384 + 8192 + 4096 + 2048 + 1024 + 512 + 256 + 128 + 64) \times \frac{1}{64}$ entrées en x, $\frac{16384}{64}$ entrées en y, et $\frac{16384}{64}$ entrées en z. Avec un timestamp représenté sur 4 octets, la taille totale de ce buffer sera de environ 134 MB. Avec un cache de brique de 4 GB (ce qui est tout à fait correct sur un GPU moderne), cela représente une surcharge en mémoire de moins de 3.5%.

3.8.2 Acheminement des données

Nous proposons un gestionnaire de requêtes efficace sur CPU, qui a pour rôle de receptionner et de traiter la *liste de requêtes* envoyée par le GPU. L'ensemble de la gestion de ces requêtes de briques sur le CPU est réalisée dans un thread dédié, créé en plus du thread principal. Cette stratégie permet d'avoir un comportement asynchrone par rapport aux autres actions du CPU et en particulier de ne pas bloquer les interactions avec le GPU pour d'autres tâches. Le gestionnaire de requêtes se charge de demander l'accès à chacune des briques de la *liste de requêtes* auprès du gestionnaire de cache CPU. Celui-ci récupère la brique, soit avec un accès direct dans son propre cache de brique, soit avec une lecture et décompression de cette brique depuis le disque, en cas de défaut de cache. Il peut ensuite fournir la brique au gestionnaire de requêtes qui va l'écrire dans un buffer

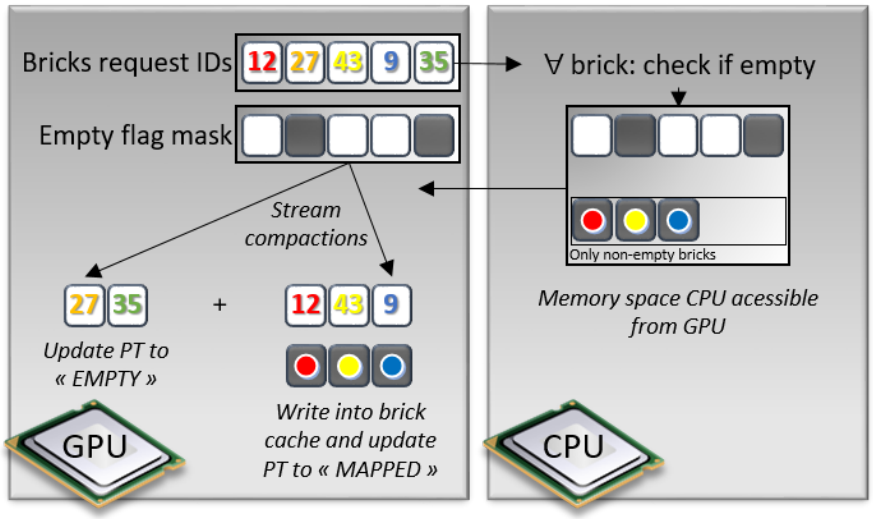


FIGURE 3.16 – **Système de gestion de l'information de brique vide pour les requêtes de briques.** Nous utilisons un vecteur de booléen qui permet d'indiquer au GPU si les briques d'une *liste de requêtes* sont vides ou non. Seules les briques non-vides sont chargées et les entrées de la table de pagination correspondantes à toutes les briques requêtées sont mises à jour avec le statut *présent* ou *vide* en fonction de l'état de ces briques.

alloué en mémoire CPU et accessible depuis l'espace d'adressage du GPU (voir figure 3.1 dans la section 3.2). Nous utilisons pour cela la technologie CUDA zero-copy [Nvi], permettant ainsi d'optimiser les transferts des briques en mémoire texture du GPU. En effet, ces transferts ne sont pas initiés par le CPU, mais ce sont les threads GPU qui peuvent directement y accéder depuis l'exécution d'un kernel CUDA sur le GPU et ainsi profiter d'une meilleur bande passante.

Le thread CPU chargé de cette gestion des requêtes, possède un *statut* qui permet d'indiquer s'il est déjà occupé ou non à traiter une *liste de requêtes*. Après chaque "passe", le thread principal du CPU, qui reste le "chef d'orchestre" de tout le pipeline, interroge ce statut afin de savoir s'il est nécessaire pour le GPU de créer, et de transférer une nouvelle *liste de requêtes* ou non. Si le thread en question n'a pas terminé de traiter les requêtes précédentes, le timestamp est incrémenté et le GPU continue sur la "passe" suivante. De la même manière, lorsque le thread a terminé de traiter toutes les requêtes d'une liste, en plus d'indiquer qu'il est à nouveau disponible pour en traiter une nouvelle, il indique que le buffer est plein, et que le GPU peut venir accéder aux briques pour les écrire dans son cache, ce qu'il fera à la fin de la prochaine "passe".

Gestion des briques vides

L'information de brique vide est importante dans notre contexte pour un grand nombre d'applications. Cela permet de distinguer certaines briques qui ne sont pas nécessaires pour les traitements de l'algorithme de l'application. Ces briques n'ont donc pas besoin d'être chargées en cache sur le GPU. Comme nous l'avons déjà évoqué, les pages de la table de pagination possèdent un "flag" qui indique le statut du bloc qu'elles adresse. Ce "flag" gère le statut *vide*, permettant d'indiquer que le bloc en question est vide, que les briques

correspondantes ne sont pas en cache GPU et qu'il n'est pas nécessaire de provoquer de requêtes pour celles-ci. Cenpendant, lorsqu'un thread GPU veut accéder à une brique qui n'est pas présente en cache (avec le statut *non-présent*), une requête est alors levée pour celle-ci alors qu'il est possible qu'elle soit vide.

La figure 3.16 montre le système que l'on propose pour gérer l'information de brique vide au moment de la gestion des requêtes envoyées par le GPU. Quand le CPU reçoit une *liste de requêtes* venant du GPU, celui-ci s'occupe alors de vérifier si les briques en question sont vides ou non. Il n'écriera que les briques non-vides dans le buffer accessible par le GPU. Il fournira également un vecteur de booléen qui possède autant d'éléments que dans la *liste de requêtes*. Chaque case de ce vecteur indiquant ainsi si la brique correspondante dans la liste est vide ou non. Avec ces deux informations, nous utilisons sur le GPU une double opération de *stream compaction* en parallèle, nous permettant de récupérer d'un coté les identifiants des briques qui sont vides et de l'autre, ceux des briques qui ne le sont pas. Tandis que les premières ne sont pas ajoutées au cache de brique GPU et que leurs pages sont mises à jour avec le statut *vide*, les autres sont écrites en cache et leurs pages sont mises à jour avec le statut *présent*.

Nous ne discutons pas ici de la manière de décider si une brique est vide ou non. C'est à l'application de le gérer. Cela peut être fait dynamiquement au cours de l'exécution, par exemple par rapport à une fonction de transfert interactive, dans le cadre d'une application de lancer de rayon pour du rendu volumique direct. Nous proposons également un système qui permet d'écrire des prédicats permettant de définir une liste de briques vides qui seront considérées comme telle pendant toute la durée de l'exécution (selon des critères colorimétriques par exemple). Ce traitement est effectué en pré-traitement en même temps que le découpage du volume en briques.

Découpage des briques à la volée

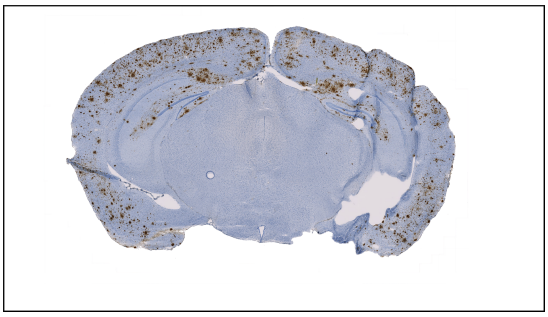
Comme nous l'avons déjà vu, nous utilisons un système qui permet de manipuler de grandes briques au niveau du stockage sur disque et dans le cache CPU, puis des briques plus petites pour l'application et notre structure d'adressage sur le GPU. Lorsqu'une petite brique est requêtée par le GPU, il y a alors deux stratégies possibles permettant de gérer cet aspect. Premièrement, le découpage de la grande brique est fait sur le CPU, et seule la petite brique réellement demandée par le GPU est ajoutée au buffer accessible par celui-ci pour l'écriture des briques dans son cache. Dans ce cas, on ne profite alors pas de l'avantage d'utiliser de grandes briques pour optimiser les temps de transferts entre le CPU et le GPU. De plus, le découpage devant être fait sur le CPU, celui-ci serait plus coûteux. Cependant, cela permet de ne charger que la ou les briques demandées, et non pas d'autres briques non demandées par le GPU qui serait présentes dans la grande brique aux cotés de celles qui l'étaient. La deuxième stratégie est de placer la grande brique dans le buffer accessible par le GPU pour le transfert, et de découper celle-ci sur le GPU, une fois chargée. Cette stratégie règle les problèmes de la première mais implique de devoir charger plus de briques que ce qui était réellement demandé par le GPU. Néanmoins, notre approche utilise la deuxième stratégie. En effet, les briques ajoutées en "surplus" étant spatialement à côté des briques demandées, il est fort probable que celles-ci auraient été demandées dans peu de temps par le GPU. Cette stratégie peut alors même être vue comme une méthode grossière de prédiction et d'anticipation des chargements.

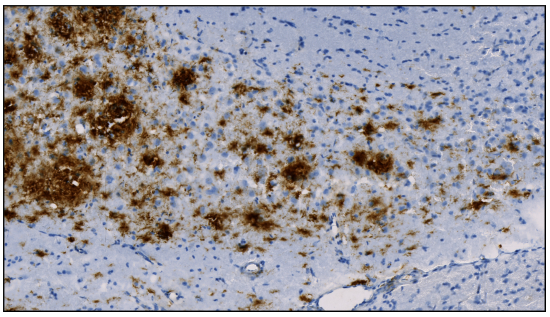
Connaissant l'identifiant d'une petite brique et le nombre de petites briques qui composent une grande, il est alors facile de retrouver l'identifiant de la grande brique corres-

pondante. Nous nous assurons de l'unicité des requêtes des grandes briques dans le cas où l'on demande plusieurs petites briques qui se trouveraient dans une seule grande. Après avoir récupéré les grandes briques, le gestionnaire de requêtes sur CPU les écrit dans le buffer de brique accessible depuis le GPU de la même manière que s'il procédait pour de petites briques. C'est ensuite le kernel CUDA se chargeant d'écrire ces briques dans le cache, qui doit faire attention au découpage des briques en calculant les coordonnées 3D correspondantes aux petites briques à l'intérieur des grandes. La liste des nouvelles petites briques qui viennent d'être ajoutées, utilisée ensuite pour la mise à jour de la structure, est calculée par rapport aux indices des grandes briques qui viennent d'être chargées et au nombre de petites briques quelles contiennent.

Chapitre 4

Microscope virtuel multi-vues





Sommaire

4.1 Introduction

L'ensemble des aspects évoqués dans ce chapitre font l'objet d'une collaboration à part égale avec les travaux de thèse de Nicolas Courilleau. Notons que les principes de construction multi-vues autostéréoscopiques appliqués ici sont approximatifs et résultent d'une étude plus phénoménologique menant à une perception de relief déformé, mais néanmoins utile à la l'appréciation 3D des échantillons biologiques. Nicolas Courilleau a mené par la suite une étude plus approfondie à partir de nos travaux communs présentés ici.

Les données volumiques utilisées dans l'imagerie médicale et biomédicale sont souvent représentées comme un empilement de coupes biologiques 2D formant un volume 3D. Il est utile de pouvoir visualiser chacune de ces coupes indépendamment et de naviguer dans celles-ci, tout en considérant le volume 3D qu'elles composent. La visualisation de telles images numériques doit pouvoir se rapprocher de la manière classique d'observer des données physiques avec un microscope, en simulant le comportement de ces appareils. La grande dimension des données est devenue courante dans l'étude des coupes biologiques (en histologie par exemple) afin de pouvoir observer des phénomènes à l'échelle microscopique. Nous nous intéressons dans ce chapitre à une méthode de visualisation interactive, permettant d'imiter le principe d'un microscope, de manière à naviguer dans une pile d'images ultra haute résolution. Nous proposons un microscope virtuel efficace, basé sur une implémentation GPU avec CUDA, combiné à notre approche out-of-core (détaillée dans le chapitre 3) pour la gestion de données volumiques de très grande dimension. Un problème de perception de la profondeur apparaît néanmoins dans un système de visualisation par microscope virtuel par rapport à une observation avec un microscope réel. La plupart des travaux précédents, dans la littérature, qui proposent des approches permettant d'observer des coupes biologiques numériques de grande dimension, se sont contentés de décrire des systèmes basés sur des microscopes virtuels ne prenant en compte qu'une seule coupe à visualiser. D'une part, ces sytèmes ne permettent pas de pouvoir naviguer dans une pile d'images considérée comme un volume 3D et surtout, n'offrent qu'une visualisation classique en 2D de données elles mêmes 2D. Notre système introduit une solution permettant de combler les lacunes d'une visualisation 2D à plat de coupe unique, en proposant un affichage sur écran 3D autostéréoscopique de coupes multiples. Ces dispositifs d'affichage, composés de plusieurs vues, permettent dans ce contexte, de recréer une information de profondeur donnant ainsi une impression de visualiser des *super-coupes* épaisses. L'utilisation de ces écrans implique la création d'images multi-vues de manière interactive. Nous proposons également un système de navigation efficace qui prend en compte l'ensemble d'une pile d'images permettant ainsi d'observer un volume 3D entier à la manière d'un microscope.

4.2 Présentation générale du système

La figure 4.1 présente une vue d'ensemble de la méthode que l'on propose pour répondre aux besoins de visualisation 3D de piles de coupes 2D de grande dimension. Le système de microscope virtuel que l'on propose se caractérise par trois aspects :

1. Un système de navigation interactive en 3D dans une pile d'images ultra haute résolution, mêlant translation et grossissement.
2. Une gestion out-of-core permettant d'accéder depuis le GPU, à n'importe quelle

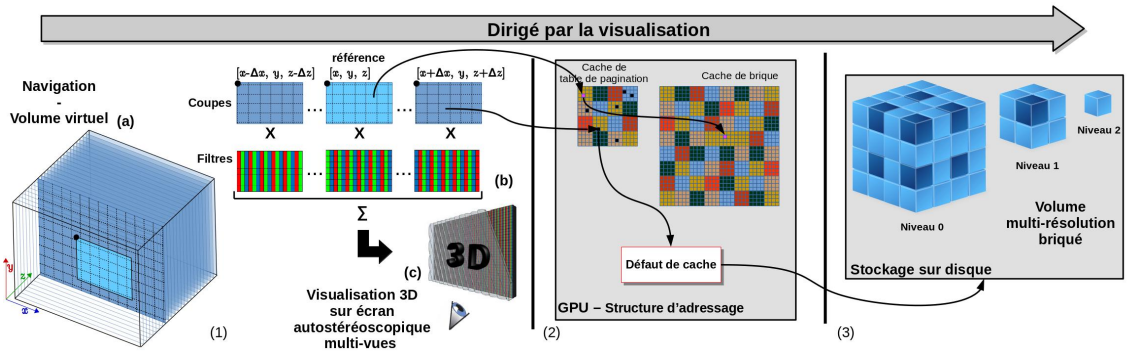


FIGURE 4.1 – **Vue d'ensemble du système de visualisation multi-vues out-of-core, interactif.** Ce système propose une navigation virtuelle dans une pile d'images ultra haute résolution (1.a), qui implique une étape de génération d'image multi-vues (1.b) pour un affichage sur écran 3D autostéréoscopique (1.c). Nous utilisons une gestion d'adressage des données depuis le GPU (2) pour un accès out-of-core au données stockées sur disque avec une représentation multi-résolution briquée (3).

partie d'un grand volume multi-résolution dépassant la capacité mémoire du GPU ou du CPU (cette approche fait l'objet d'une présentation détaillée dans le chapitre 3).

3. Une méthode de construction d'images multi-vues en temps interactif pour la visualisation sur un écran 3D autostéréoscopique.

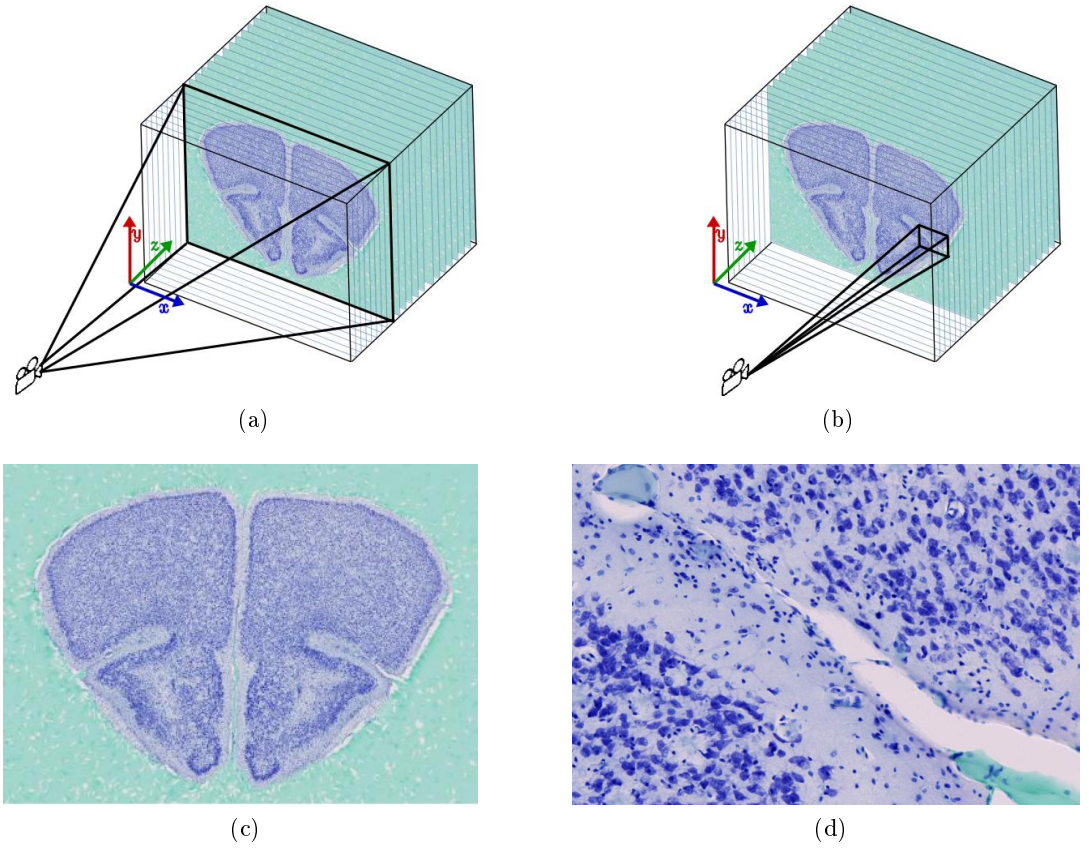
Nous utilisons, dans ce pipeline, la représentation multi-résolution et le découpage en brique d'un volume, que nous avons détaillés dans la section 3.4 (voir figure 4.1(3)). Nous proposons également d'utiliser l'ensemble de la gestion out-of-core présentée dans le chapitre 3, permettant ainsi d'accéder à n'importe quelle partie du volume multi-résolution depuis le GPU. Le principe de navigation dans un volume normalisé nous permet de parcourir l'ensemble de la pile d'images selon les besoins de l'utilisateur. Celui-ci ayant la possibilité, à la fois de se déplacer dans le volume sur l'axe z, afin de choisir une coupe à visualiser, et de se déplacer à l'intérieur d'une telle coupe avec des mouvements de navigation dans le plan, en x et y, et de zoom/dézoom. Ces actions de zoom/dézoom engendrent une navigation dans la pyramide multi-résolution selon un critère basé sur la taille de la projection sur l'écran d'un pixel de l'image. Lorsque celui-ci occupe plus d'un pixel à l'écran, nous adaptons la résolution au niveau inférieur (plus résolu) dans la pyramide multi-résolution. Nous proposons un pipeline qui est dirigé par la visualisation ("visualisation-driven" en anglais). Ce sont les interactions de l'utilisateur, impliquant des changements de région d'intérêt, qui vont déclencher les différentes étapes nécessaires à la mise à jour de la visualisation.

Lors d'une interaction de navigation par l'utilisateur, notre système se charge de constituer l'ensemble des N images qui entrent en jeu dans la composition de l'image multi-vues qui sera affichée sur l'écran 3D. Cette quantité N dépend directement du matériel et du nombre de vues attendues par l'écran multiscopique servant à l'affichage. Cette constitution implique de déterminer des zones à l'intérieur de plusieurs coupes consécutives comme nous le verrons dans la section 4.3.1. Toutes les briques qui composent ces images doivent pouvoir être accédées depuis le GPU et donc être présentes dans le cache de briques prévu à cet effet. Cela peut donc impliquer un ensemble de requêtes pour provoquer le streaming de ces briques selon leur emplacement mémoire physique. Toutes ces images sont ensuite composées de manière à construire une image multi-vues cohérente pouvant être affichée sur

l'écran 3D autostéréoscopique cible. Cela est opéré en multiplexant les N images grâce aux filtres de multiplexage spécifiquement associés à chaque vue, pour cet écran (voir section 4.3.3).

Visualisation sur GPU

L'affichage se fait au travers d'une seule grande texture 2D gérée avec openGL. Celle-ci est positionnée dans le plan orthogonal à la caméra et est approvisionnée au fur et à mesure des interactions de l'utilisateur, par un kernel CUDA. Toutes les manipulations de cette texture sont faites sur le GPU, en utilisant l'interopérabilité entre openGL et CUDA qui offre le partage d'un espace mémoire accessible par ces deux technologies. Cette stratégie nous permet d'offrir une gestion performante en évitant les transferts avec le CPU. Cela est également en partie rendu possible par le fait que la gestion des grands volumes de données que l'on manipule, est entièrement réalisée sur le GPU.



(a) (b)

(c) (d)

FIGURE 4.2 – **Visualisation d'une pile de coupes histologiques et illustrations du volume correspondant.** (a,b) Les coupes sont empilées sur l'axe des z dans le volume normalisé utilisé pour la navigation. Le modèle de caméra utilisé étant orthogonal à cet axe, notre système de microscope virtuel nous permet d'obtenir les vues (ici 2D) (c) et (d), offrant deux niveaux de zoom différents avec un niveau de résolution adapté dans les deux cas.

Dans notre représentation, une coupe est une image 2D qui s'étend sur les axes x et y du volume et la profondeur des coupes dans le volume est exprimée sur son axe z. Nous utilisons un modèle de caméra avec un champ de vision orthogonal à l'axe des z dans le volume, celui sur lequel sont empilées les coupes 2D. Ce champ de vision est donc aligné

avec le plan de coupe $[x, y]$ et la visualisation offre une vue de tout ou partie d'une de ces coupes. La figure 4.2 nous permet d'illustrer ce principe. Les vues (c) et (d) de cette figure, sont le résultat de la visualisation avec notre système de microscope virtuel avec un affichage 2D classique. Les illustrations (a) et (b) sur cette figure, nous permettent d'illustrer le volume correspondant, composé d'une pile de coupes histologiques issues de l'acquisition de lames biologiques d'un cerveau de rongeur. Nous pouvons ainsi remarquer l'orientation des coupes dans le volume normalisé et la visualisation qui peut en résulter.

4.3 Construction des images multi-vues

Afin de pouvoir afficher le rendu produit par notre microscope virtuel, sur un dispositif d'affichage 3D multiscopique et de profiter d'une perception de profondeur dans la visualisation, il est nécessaire de pouvoir générer des images multi-vues cohérentes. Nous détaillons dans cette section les différentes étapes de construction de ces images multi-vues. Nous proposons un système capable de construire les images multi-vues en temps interactif pouvant ainsi fournir une visualisation avec navigation interactive efficace. Le flux d'images multi-vues généré permet d'avoir une perception de vision en 3D, en s'appuyant sur un calcul de différence de parallaxe mettant à contribution un petit ensemble de coupes consécutives, de manière à prendre en compte une certaine profondeur.

4.3.1 Parallaxe et profondeur

Nous utilisons deux aspects afin de créer des images multi-vues qui puissent permettre de retranscrire, de manière efficace, une perception de vision tridimensionnelle et de profondeur dans la visualisation des coupes.

Premièrement, pour une image multi-vues donnée, nous proposons de mettre à contributions plusieurs coupes consécutives dans le volume de manière à cumuler l'information sur une certaine épaisseur. Cela nous permet d'introduire une notion de *super-coupe*, qui donne une impression d'épaisseur aux coupes que l'on visualise. Cette stratégie offre alors une vision augmentée par rapport à un affichage avec un microscope virtuel classique, en proposant d'observer une continuité dans les structures qui composent les images. Notre système a vocation à s'appliquer plus particulièrement à la visualisation de coupes biologiques. Les images 2D correspondantes n'offrent quasiment aucune profondeur ni aucune perspective. En effet, cela est dû au fait qu'elles sont issues de l'acquisition d'une coupe planaire, d'épaisseur négligeable. Chacune des coupes ne contient donc aucune information 3D. Une seule coupe ne permet donc pas à elle seule de faire profiter, à l'observateur, d'un effet de perspective dont il a besoin pour observer un objet en trois dimensions. Le nombre de coupes consécutives, qui sont mises à contribution pour créer une *super-coupe*, est égal au nombre de vues qui composent l'écran multiscopique utilisé pour l'affichage 3D. Un écran autostéréoscopique à N points de vue, impliquera de construire des *super-coupes* dont N coupes consécutives dans le volume 3D participent.

Deuxièmement, nous appliquons le principe de stéréopsie, permettant la vision 3D par notre système biologique de vision binoculaire basé sur la différence de parallaxe dans les images parvenant à nos deux yeux. Pour cela, nous décalons horizontalement selon l'axe x du volume chacune des coupes dans chaque vue. Cet axe est perpendiculaire au

vecteur directeur de la caméra qui intersecte le plan de la coupe à visualiser. C'est l'axe des abscisses dans le plan 2D de la coupe, qui elle, est orthogonale au champ de vision de la caméra. Ce décalage est appliqué progressivement à la zone à visualiser correspondante dans chaque coupe de la *super-coupe*. Ainsi, chaque coupe qui contribue à une *super-coupe* se voit attribuer une zone dont la position est décalée sur l'axe x par rapport à la zone de la coupe qui lui succède dans le volume. Cela permet de simuler la parallaxe propre à chacune des N vues.

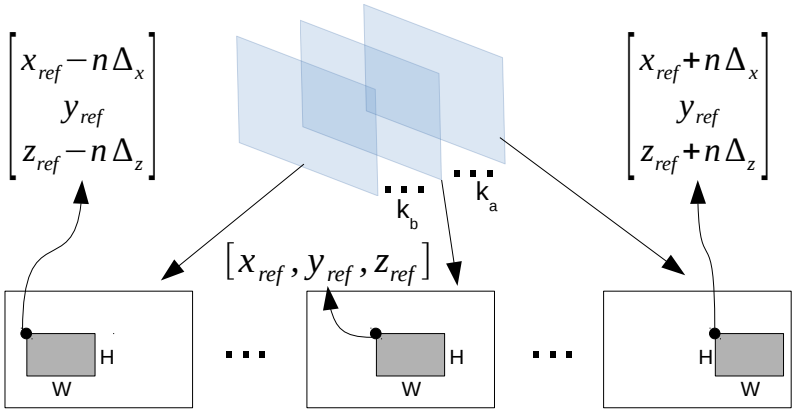
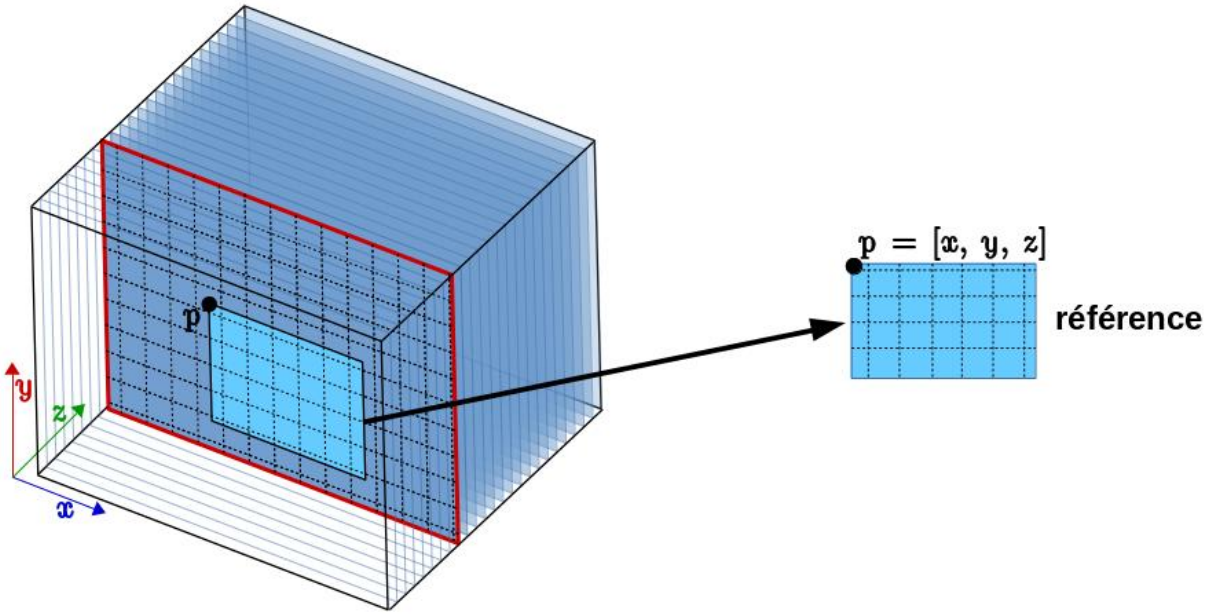
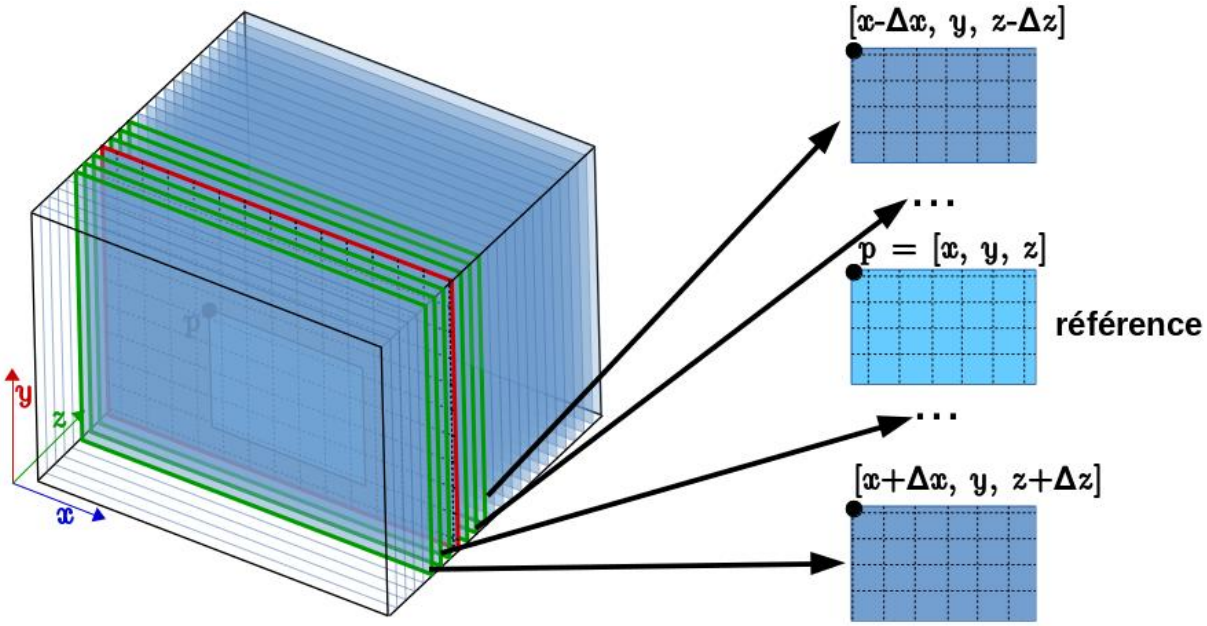


FIGURE 4.3 – **Création d'une image multi-vues à partir d'un ensemble de coupe d'une pile d'image.**

Nous utilisons deux paramètres permettant de décrire les décalages expliqués dans les paragraphes précédents. Nous les noterons Δ_x et Δ_z. Ceux-ci permettent de décrire respectivement : la quantité de pixels utilisée comme saut pour le décalage horizontal sur l'axe x attribué à la différence de parallaxe ; le nombre de coupes, dans le volume, qui séparent deux coupes consécutives de la *super-coupe*. Pour cette dernière information, avec $\Delta_z = 1$ nous choisirons des coupes qui se suivent directement dans le volume. Ces deux décalages sont exprimés par rapport à une zone de référence qui est celle utilisée pour la navigation dans le volume. Cette zone 2D est caractérisée, par une position 3D que l'on choisi de manière totalement arbitraire comme étant celle de son coin supérieur gauche, d'une part, et par sa taille, égale aux dimensions 2D de la zone d'affichage à l'écran, d'autre part. Cette zone de référence appartient à la coupe qui se trouve au centre de la *super-coupe*. Nous reviendrons sur ces aspects un peu plus en détail dans la section suivante. Nous introduisons ce concept de zone de référence ici, afin de comprendre comment sont appliqués les décalages Δ_x et Δ_z. Ceux-ci contribuent de manière négative pour les zones appartenant à chaque coupe précédant la coupe qui contient la zone de référence sur l'axe z du volume. A l'inverse, ils contribuent de manière positive pour les zones appartenant à chaque coupe succédant la coupe qui contient la zone de référence sur l'axe z du volume. La figure 4.3 permet d'illustrer l'approche proposée. On y note, la position 3D de la zone de référence par $[x_{ref}, y_{ref}, z_{ref}]$ et sa taille avec W, sa largeur et H, sa hauteur. k_a et k_b représentent la quantité de coupes se trouvant, respectivement, entre la première coupe et la coupe contenant la zone de référence, et entre la coupe contenant la zone de référence et la dernière coupe contribuant à la *super-coupe*. La quantité k_a peut être différente de k_b dans le cas où l'on se trouve avec un affichage proposant un nombre de vues paires dans la composition de l'écran multiscopique.



(a)



(b)

FIGURE 4.4 – **Navigation virtuelle et sélection des zones contribuant aux images multi-vues.** Illustration de la sélection des vues composant une image multi-vues correspondant à la zone à visualiser. Les coordonnées $p \equiv [x, y, z] \in [0, 1[^3$ de la position établie pendant la navigation dans le volume normalisé sont utilisées pour déterminer une zone de référence. Les autres zones sont déterminées par rapport à celle-ci avec les paramètres Δ_x et Δ_z, ainsi que le nombre de vues N.

4.3.2 Navigation virtuelle et sélection multi-vues

Nous sommes dans un contexte de visualisation de jeu de données de grande dimension, dont la représentation ne tient pas forcement en mémoire GPU ou même CPU. Nous proposons d'intégrer l'ensemble du système de gestion out-of-core des données, qui a été présenté dans le chapitre 3. En particulier, cela inclut le principe de navigation dans un volume normalisé depuis le GPU. Nous proposons de faire le lien entre ce principe de navigation virtuelle et la construction d'images multi-vues. Cela est rendu possible par la définition d'un ensemble de zones, déterminées à partir d'une certaine zone de référence qui résulte de la navigation dans le volume et donc de la partie de celui-ci que l'on souhaite visualiser. Les coordonnées $p \equiv [x, y, z] \in [0, 1[^3$, de la position flottante dans le volume normalisé, sont utilisées pour définir la position 3D du coin supérieur gauche de cette zone de référence. Cette zone s'étend sur W pixels sur l'axe x du volume et H pixels sur l'axe y. W et H étant définis par la taille de la fênetre utilisée pour l'affichage à l'écran ("viewport" en anglais). A partir des paramètres Δ_x et Δ_z, ainsi que du nombre de vues N, nous pouvons ensuite déterminer les autres $N-1$ autres zones. La figure 4.4 permet d'illustrer la selection des coupes dans le volume et la détermination de la zone de référence et des autres zones nécessaires à la construction de l'image multi-vues correspondante, à partir de l'étape de navigation virtuelle.

4.3.3 Transparence et multiplexage colorimétrique

Une fois que les différentes zones du volume sont déterminées pour définir les différentes vues qui vont composer l'image multi-vues finale, il est possible de calculer cette image multi-vues à proprement parlé. Cette étape consiste à calculer les différentes contributions de chacune des vues pour l'image finale qui sera affichée sur le dispositif 3D. Ces contributions portent sur les aspects colorimétriques et de transparence.

Une application de microscope virtuel comme le système que l'on propose, a principalement pour vocation de permettre de visualiser des coupes biologiques. Les images utilisées, dans ce contexte, par les biologistes, sont souvent des images composées de pixels contenant une information de couleurs (RGB). Nous proposons d'utiliser une fonction de transfert afin de définir des quadruplets RGBA pour l'ensemble des valeurs des pixels. Cela nous permet en particulier d'attribuer une valeur d'opacité sur un canal *alpha*, pour toutes les vues. Les valeurs RGB sont, quand à elles, soit définies par les "vraies" valeurs issues de l'acquisition dans le cas d'images couleurs, soit par la fonction de transfert par rapport aux valeurs de luminence dans le cas inverse. L'information de transparence permet, dans notre contexte, d'accumuler l'information de couleur à travers l'épaisseur d'une *super-coupe*. Nous proposons de réaliser cette accumulation de la manière suivante : nous utilisons une méthode d'accumulation "back-to-front" sur les images provenant de la coupe 0 jusqu'à la coupe $N/2$ (la coupe de référence). Nous inversons l'ordre d'accumulation en utilisant une méthode "front-to-back" sur les images provenant de la coupe $N/2$ jusqu'à la coupe N.

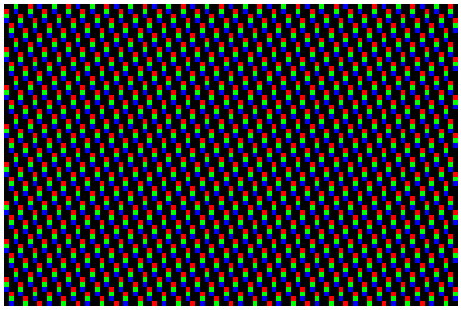


FIGURE 4.5 – **Zoom d'un filtre de multiplexage d'une des vues d'un écran autostéréoscopique HD.**

Filtres de multiplexage

Nous utilisons un ensemble de filtres, au nombre de N, qui sont des images 2D, de dimension $W \times H$ identiques à celles des N zones extraites du volume (et également identiques à celles de l'image multi-vues après construction). La composition, le nombre et la taille de ces filtres sont directement liés à l'écran autostéréoscopique utilisé pour la visualisation. La figure 4.5 montre une partie zoommée d'un filtre, d'une taille originale de 1920×1080, d'une des vues d'un écran autostéréoscopique. Chacune des vues d'un tel écran ayant son propre filtre, correspondant aux propriétés de l'écran pour la vue en question, il est important de respecter l'ordre d'application de ceux-ci, en fonction du numéro de la vue.

L'application de ces filtres sur les N images pré-calculées, permet d'attribuer la contribution de chaque composante colorimétrique aux différents point de vue de l'écran, conformément aux propriétés optiques de celui-ci. D'un point de vue mathématique, cela se résume en un produit, pixel à pixel pour chaque composante colorimétrique, de chacune des vues $\mathcal{V}_c^i(x, y)$ avec le filtre correspondant $\mathcal{F}^i : \{0, 1\}^3 \, [\mathbb{Z}^2]$. Pour terminer notre processus de construction, les N images résultantes de ces produits, sont ensuite sommées afin d'obtenir une seule image multi-vues correcte. Ces opérations sont illustrées sur la figure 4.6, et décrites par l'équation suivante :

$$\mathcal{V}_c^{final}(x, y) = \sum_i \mathcal{V}_c^i(x, y) \; \mathcal{F}_c^i(x, y) \qquad \text{with } c \in \{R, G, B\} \tag{4.1}$$

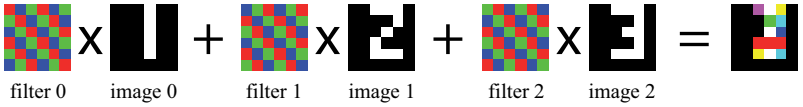
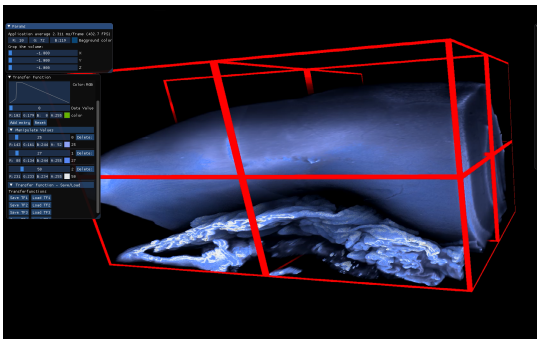
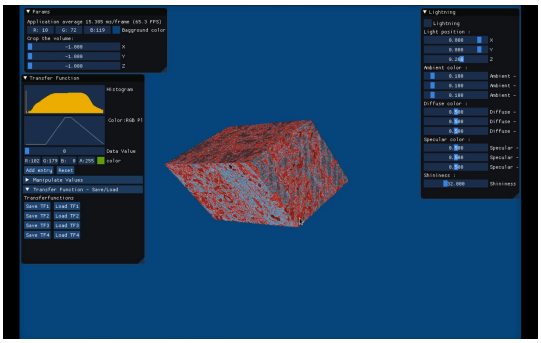


FIGURE 4.6 – **Application du multiplexage.** Une image multi-vues résulte d'une étape de multiplexage, par application de filtres sur chaque image pour calculer la contribution des trois composantes colorimétriques de toutes les vues (source : [NPB+12]).

Chapitre 5

Lancer de rayon volumique multi-GPUs avec visualisation distante





Sommaire

5.1 Introduction

Dans ce chapitre, nous allons aborder la méthode de rendu volumique direct en temps-réel sur GPU, dans un contexte out-of-core, pour des volumes dont la représentation ne tient pas en mémoire GPU. Cette technique de rendu (en dehors du contexte out-of-core) permet d'apprécier l'ensemble d'une grille scalaire 3D via une fonction de transfert permettant de déterminer la participation (couleur et opacité) de l'ensemble de la plage de données visualisée. Cette méthode est très utilisée en visualisation scientifique et plus particulièrement dans les domaines médicaux et biomédicaux. Les méthodes modernes de rendu volumique direct interactif sont basées sur un algorithme de lancer de rayon en parallèle sur GPU. Cet algorithme étant aujourd'hui le plus démocratisé pour ce type de visualisation, de par son efficacité, son caractère intuitif pour répondre aux besoins de l'approche et son adaptation à une implémentation parallèle. L'utilisation de cette solution pour visualiser de grands volumes de données requiert une solution de gestion de données out-of-core adaptée. De telles approches out-of-core sont utilisées dans ce contexte depuis environ dix ans maintenant. C'est d'ailleurs ce type de rendu qui a dirigé les recherches dans le domaine. Les méthodes proposées dans la littérature se sont, en revanche, concentrées pour la plupart, sur des solutions visant des plateformes PC standard. L'utilisation d'environnements de calcul haute performance proposant des architectures multi-GPUs semble pourtant être naturelle pour répondre à ces besoins. En effet, il est intéressant de pouvoir profiter, d'une part, de la capacité de calcul qu'offrent leurs architectures massivement parallèles afin d'améliorer l'efficacité de la méthode de rendu et d'autre part de pouvoir répartir la grande quantité de données dans les mémoires de travail des différents GPUs. Certains travaux se sont orientés sur l'utilisation de tels environnements dans ce contexte, mais à notre connaissance, uniquement avec des approches out-of-core basées sur une structure d'arbre. Comme nous l'avons vu dans le chapitre 3, le modèle de gestion out-of-core que l'on propose est plus adapté qu'un octree pour de très grands volumes de données. Des approches utilisant un modèle de gestion out-of-core, en partie similaire à celui que l'on propose, pour du rendu volumique direct par lancer de rayon sur GPU, ont déjà été utilisées, (en particulier dans les travaux de Hadwiger *et al.* [HBJP12] et de Fogal *et al.* [FSK13]) mais uniquement avec l'utilisation d'un seul GPU pour effectuer le rendu.

Nous présentons dans ce chapitre, une approche de rendu volumique direct, mettant en application notre modèle de gestion out-of-core de données volumiques dans une solution haute performance distribuée. Nous proposons une méthode parallèle de lancer de rayon volumique multi-résolution interactif, permettant de visualiser de très grands volumes de données sur des noeuds de calcul hybrides, multi-GPUs, multi-CPUs. Cette solution out-of-core distribuée permet de profiter de la puissance de calcul, des capacités de stockage, ainsi que des technologies de communication de tels environnements, souvent bien plus performants que les plateformes standard. Notre approche offre un déport d'affichage sur des clients légers distants, en garantissant un flux vidéo interactif pour une visualisation fluide. Nous proposons d'implémenter une solution de lancer de rayon volumique out-of-core interactif sur GPU par une approche "ray-guided", permettant ainsi d'offrir une méthode efficace de gestion de la visibilité, qui s'accorde avec la gestion d'acheminement des données à la demande, jusqu'aux GPUs. Nous concentrons notre étude sur l'implémentation d'une telle solution sur des environnements de calcul composés d'un seul noeud de calcul, regroupant plusieurs GPUs. Ce choix est dirigé par l'arrivée sur le marché, de serveurs de calculs dédiés à la visualisation et proposant ce type d'architecture à n GPUs regroupés dans un seul serveur. Ce type d'architecture impacte directement les choix technologiques

d'implémentation et les stratégies de parallélisation et de communication. Nous allons donc détailler dans ce chapitre l'ensemble des choix et stratégies que nous avons portés afin de proposer une solution de rendu distribué performante.

L'ensemble des travaux réalisés dans le cadre de ce chapitre ne prend pas part au travail collaboratif avec la thèse de Nicolas Courilleau. Ils sont donc uniquement et entièrement décrits dans le présent manuscrit de thèse.

5.2 Lancer de rayon out-of-core

Avant de décrire notre stratégie de distribution multi-GPUs et notre approche de visualisation distante, nous allons discuter dans cette section du module de rendu de base que l'on utilise. Celui-ci repose sur une approche de rendu volumique direct basée sur les équations présentées dans la section 2.2.3 et implémentée par un algorithme de lancer de rayon volumique parallèle, en une seule passe, sur GPU avec CUDA. L'algorithme de base repose sur le modèle des propriétés optiques d'émission et d'absorption de la lumière, décrites par Max [Max95].

5.2.1 Méthode de rendu

Notre méthode se concentre uniquement sur la génération de rayons primaires, avec autant de rayons qu'il y a de pixels dans le viewport. Cette représentation nous permet de générer un kernel CUDA avec autant de threads que de rayons, effectuant ainsi le traitement de ceux-ci en parallèle. Nous utilisons une boîte englobante pour définir les limites du volume dans l'espace. Nous définissons l'origine et la direction de chaque rayon, dans le kernel CUDA chargé de dérouler l'algorithme en parallèle, à partir des propriétés du modèle de caméra envoyé à ce kernel. Nous pouvons alors calculer l'intersection des rayons avec la boîte englobante du volume. Les rayons qui ne l'intersectent pas, ne participent pas au reste de l'algorithme et se contentent de remplir le buffer de pixels avec une couleur définie comme étant la couleur d'arrière plan. Les rayons qui intersectent la boîte englobante, vont participer à la boucle d'accumulation de la couleur C et de l'opacité α, à l'intérieur du volume, par rapport aux propriétés optiques de la lumière, décrites par une fonction de transfert interactive. Cette accumulation est faite en utilisant l'opérateur $OVER$, dans le sens "front-to-back", entre le point d'intersection d'entrée et le point d'intersection de sortie dans le volume, profitant ainsi du calcul de terminaison précoce (appelé "early ray terminaison" en anglais) permettant de quitter le volume prématurément si l'opacité α a atteint un certain seuil. En plus de l'approche classiquement utilisée dans ce contexte, nous proposons un algorithme de rendu multi-résolution utilisant notre modèle de gestion de données out-of-core par une méthode dite "ray-guided".

5.2.2 Gestion multi-résolution

Nous utilisons, dans notre méthode de rendu, la représentation multi-résolution du volume, calculée en pré-traitement et gérée par notre structure d'adressage out-of-core. Nous intégrons un système de choix du niveau de détail dans notre algorithme qui s'occupe

d'échantilloner à l'intérieur du volume le long des rayons qui le traversent. Pour chaque échantillon, nous calculons la taille projetée dans l'espace écran, du voxel auquel il apartient. Nous sélectionnons ensuite le bon niveau de résolution l pour satisfaire au mieux le fait que cette projection du voxel occupe exactement un pixel à l'écran. Nous obtenons ainsi une adaptation de la résolution utilisée, par rapport à la distance de la vue. Ce calcul peut être opéré, à partir de la position du voxel dans l'espace, des matrices qui définissent les propriétés du modèle de caméra utilisé pour la projection et, des dimensions en voxels du volume à chaque niveau de résolution.

Pas d'échantillonnage

Le nombre d'échantillons utilisés le long des rayons est directement lié, à la fois, à la qualité du rendu, mais aussi, à la performance générale de l'algorithme. En effet, plus on utilise d'échantillons, plus on obtient une bonne qualité de rendu (ce qui est vrai jusqu'à un certain point), mais plus l'on augmente la quantité de calcul. Afin de répondre à cette balance, et parce que l'on applique un rendu multi-résolution, nous utilisons un pas d'échantillonnage adaptatif pour "marcher" le long des rayons et décrire la distance entre les échantillons sur ceux-ci. Ce pas d'échantillonnage évolue au cours de la marche, par rapport au niveau de résolution l sélectionné pour chaque échantillon (voir figure 5.1). Il est possible de pré-calculer le pas d'échantillonnage propre à chaque niveau de résolution, par rapport à la dimension en voxels de chacun d'entre eux. Nous établissons ce calcul de manière à ce que la distance entre deux échantillons soit égale à la taille d'un voxel, afin de reconstituer le signal avec une qualité suffisante, tout en minimisant la quantité d'échantillons (pour garantir de bonnes performances). La taille d'un voxel dans l'espace étant plus importante pour un niveau de résolution $l = n$ que pour le niveau précédent $l = n - 1$ dans la pyramide multi-résolution, plus on utilise un niveau de détail élevé (l faible), plus le pas d'échantillonnage est grand.

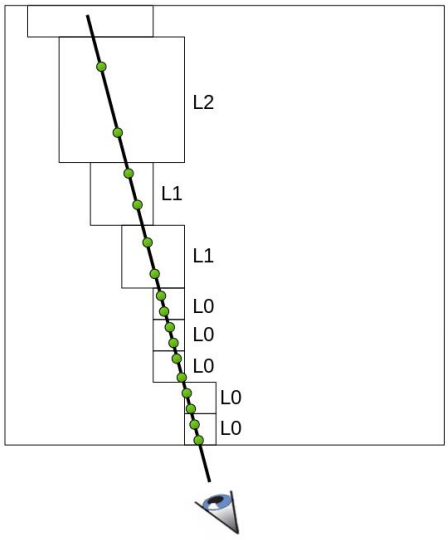


FIGURE 5.1 – **Pas d'échantillonnage adaptatif au niveau de détail.**

5.2.3 Approche "ray-guided" et gestion out-of-core

Le lancer de rayon volumique que l'on propose s'inspire des méthodes modernes dites, "ray-guided". Avec cette stratégie, le calcul de visibilité des voxels est intrinsèquement lié à l'étape d'échantillonnage du volume le long des rayons qui le traversent, et il est induit par ce principe, sans nécessiter de calculs supplémentaires. C'est la boucle principale de l'algorithme, avançant de pas en pas pour échantillonner le long des rayons, qui dirige naturellement l'accès aux voxels qu'elle rencontre. Seuls les voxels se trouvant sur le chemin d'un rayon sont accédés et jamais un voxel occulté ou en dehors du champ de vision ne sera demandé. Cette stratégie s'imbrique parfaitement avec le concept de gestion de données out-of-core et des besoins de streaming de ces données et de mise en cache sur GPU. En effet, elle permet naturellement de ne demander le chargement uniquement des briques visibles et contribuant au rendu.

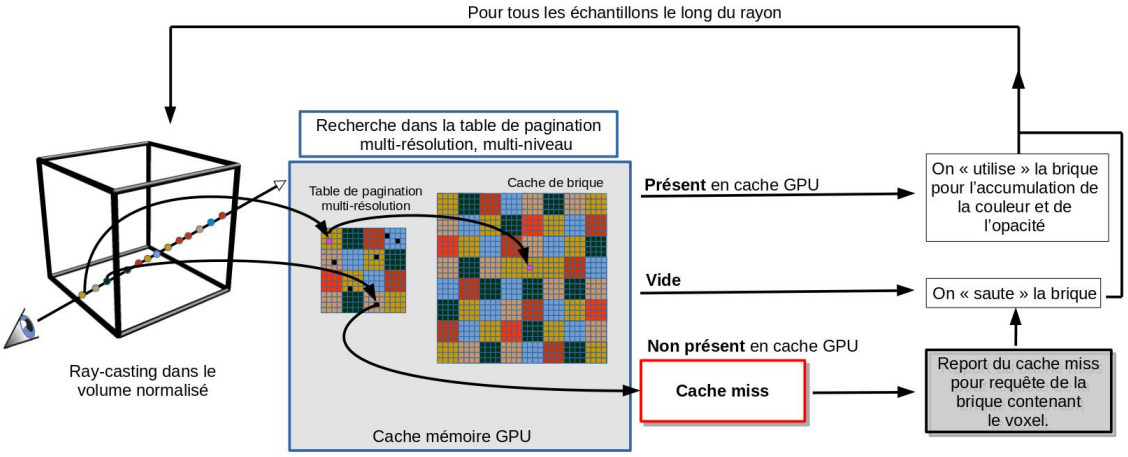


FIGURE 5.2 – **Pipeline "ray-guided" out-of-core.** La structure d'adressage virtuel est parcourue à chaque échantillon le long du volume et seules les parties visibles sont visitées et chargées sur le GPU.

Intégration du modèle de gestion out-of-core

L'utilisation de notre modèle d'adressage virtuel des données, comprenant notre structure hierarchique de table de pagination et le système de cache associé, s'intègre efficacement et naturellement à notre algorithme de lancer de rayon "ray-guided". Pendant le déroulement de ce dernier, au cours d'une passe de rendu, l'accès à un échantillon dans le volume se fait simplement, à la manière classique d'un accès direct dans une texture, juste par un appel de fonction GPU. Derrière cet appel, le parcours de notre structure s'applique (comme expliqué dans le chapitre 3) et l'on peut directement accéder au voxel, si celui-ci est actuellement en cache, dans la texture correspondante, par interpolation trilinéaire proposée directement sur le GPU. Voici comment et quand interviennent les différentes étapes de la gestion out-of-core que l'on propose, par rapport au déroulement du rendu :

— Pendant l'algorithme du lancer de rayon, à chaque échantillon, le thread correspondant à un rayon, va : effectuer le parcours de notre structure (qui est, pour rappel, moins coûteux que celui d'un octree), marquer l'utilisation d'une brique, si l'échantillon demandé est positionné dans une brique qui est présente en cache, ou, marquer la requête de la brique si elle n'est pas présente en cache.

— Entre deux passes de lancer de rayon, sont réalisées les autres étapes nécessaires aux bon fonctionnement de notre modèle d'adressage virtuel. C'est à dire, la mise à jour des LRUs, la création de la liste de requêtes de briques (et son envoi au CPU) et la mise à jour de la structure (écriture des briques en cache et mise à jour du/des niveau(x) de la table de pagination).

La figure 5.2 nous permet d'illustrer notre approche "ray-guided", utilisant notre modèle de gestion out-of-core.

Saut de briques vides ou non-présentes en cache

Lors du parcours d'échantillons dans le volume, le long d'un rayon donné, en utilisant intuitivement notre structure d'adressage virtuel et sa table de pagination multi-niveaux, multi-résolution, il est possible de sauter certaines zones vides ou, dont les données ne sont pas encore accessibles sur le GPU ; car non présentes en cache. La figure 5.3 permet d'illustrer cela avec un exemple. Si le flag d'une entrée de la table de pagination possède un

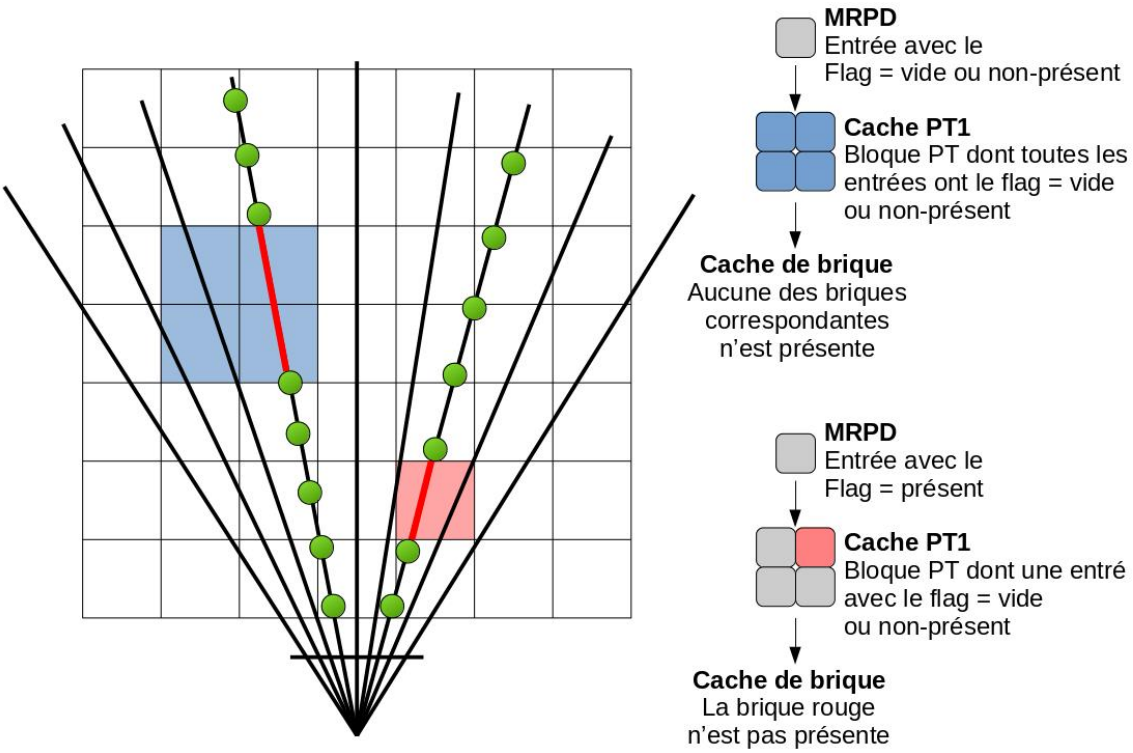


FIGURE 5.3 – **Saut d'espace vide ou de bloc non présent dans le cache GPU.** Soit une configuration de notre structure d'adressage virtuel avec 2 niveaux de virtualisation, où une entrée de la MRPD virtualise un *bloc PT* de 2×2 entrées dans le niveau de *cache PT1*. Si une entrée de la MRPD possède le flag *vide* ou *non-présent*, il est possible de sauter l'espace occupé par l'ensemble des briques qu'elle virtualise. Si c'est une entrée d'un *bloc PT* dans le niveau de *cache PT1*, qui possède un de ces deux flags, alors il est possible de sauter l'espace d'une brique.

statut *vide* ou *non-présent*, il est possible d'incrémenter le prochain pas le long du rayon, de la distance occupée par la/les briques virtualisée(s) par cette entrée, selon le niveau de résolution préalablement selectionné pour l'échantillon courant. Le fait de pouvoir se passer de ces zones pendant le parcours d'un rayon est très important d'un point de vue des performances car cela permet de réduire le nombre d'échantillons visités.

L'approche proposée et décrite dans cette section est proche, pour certaines parties, des travaux de Crassin *et al.* [CNLE09] (qui utilisent cependant une structure d'octree), de ceux d'Hadwiger *et al.* [HBJP12] ou encore de Fogal *et al.* [FSK13]. Cependant, toutes ces approches se concentrent sur l'utilisation d'un seul GPU, sur des plateformes composées d'un PC standard. Nous proposons, dans la section suivante, une stratégie de distribution de notre approche de rendu "ray-guided" out-of-core, sur des plateformes hybrides multi-GPUs, multi-CPUs.

5.3 Stratégie multi GPUs

Nous détaillons dans cette section, la stratégie proposée pour une distribution efficace de notre modèle de gestion out-of-core et de notre méthode de lancer de rayon volumique pour paralléliser le rendu sur des systèmes multi-GPUs. Nous proposons une méthode adaptée à des architectures composées d'un seul noeud de calcul comprenant plusieurs GPUs et un ou plusieurs CPUs. En prenant en compte n'importe quelle topologie, au niveau de la disposition des GPUs et CPUs ainsi que des liens de communication entre eux, il est possible de représenter une telle architecture comme montrée sur la figure 5.4.

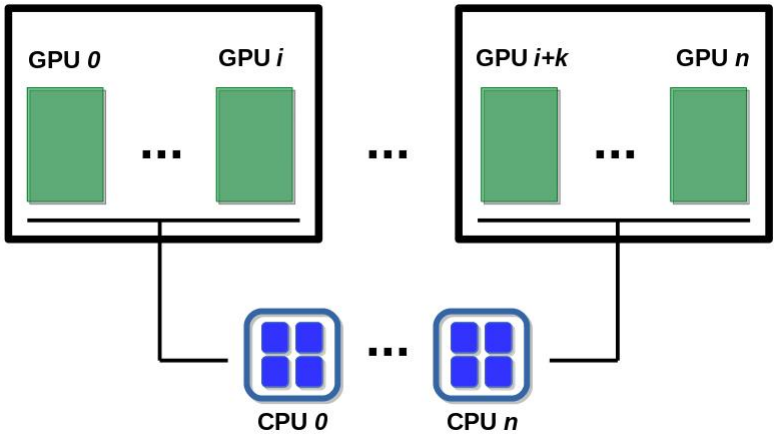
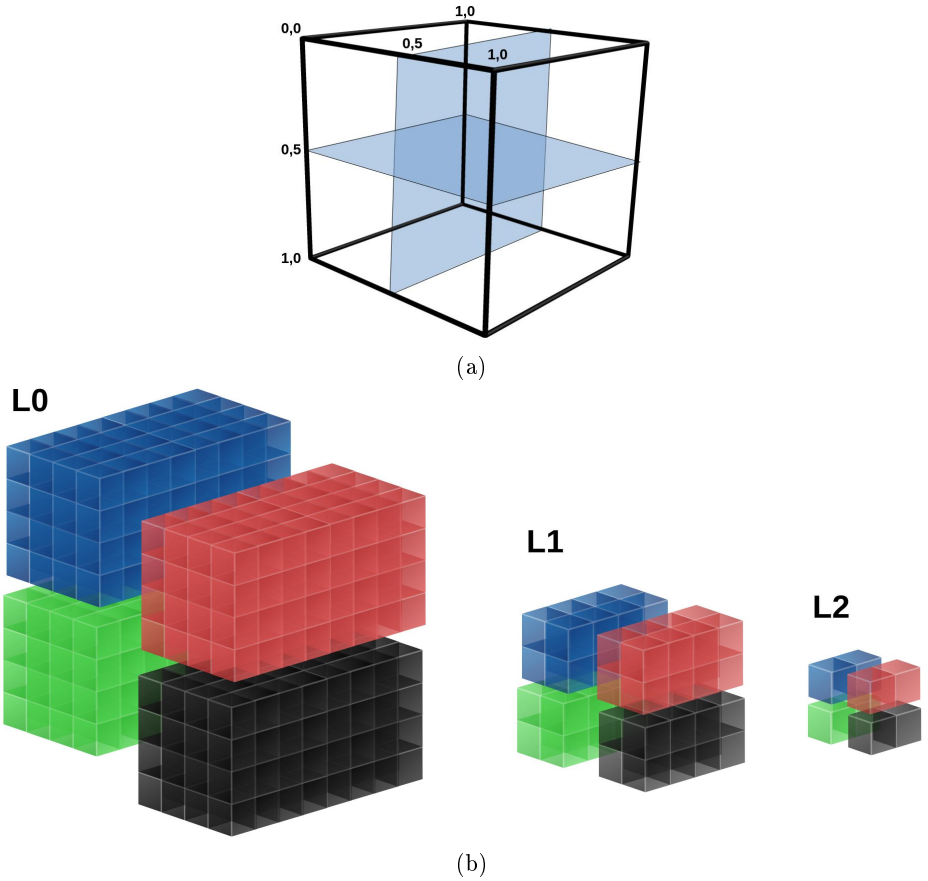


FIGURE 5.4 – **Architecture hybride multi-GPUs multi-CPUs.** Illustration d'un noeud de calcul comprenant un système multi-GPUs relié à différents CPUs multi-coeurs.

5.3.1 Approche de rendu "sort-last"

Comme nous en avons déjà discuté dans la section 2.3.3, il existe trois approches de rendu distribué, qui se différencient par l'étape du pipeline de rendu à laquelle la distribution est effectuée. L'approche de distribution "sort-last" est sûrement la plus utilisée en rendu volumique dans la littérature. Celle-ci est en effet plus adaptée à ce contexte et plus particulièrement pour de grands volumes de données. En plus de profiter d'une bonne répartition de la charge de calcul, elle offre, à l'inverse des deux autres approches, une mise à l'échelle d'un point de vue de la gestion de la mémoire. En effet, avec une telle approche, le principe de distribution est de réaliser une décomposition de domaine dans l'espace des données (du volume), afin d'attribuer une sous-partie de celles-ci à chaque processeur intervenant dans le calcul. Cette répartition et attribution des données aux différents processeurs, réalisée au préalable du rendu, permet ainsi d'éviter des transferts de données entre ceux-ci. A l'inverse par exemple, une approche "sort-first" implique de devoir basculer la gestion des données entre les différents processeurs, selon les interactions avec la caméra au cours du rendu. Une implémentation utilisant une stratégie de rendu "sort-first" nous obligerait à devoir gérer un nombre important de défauts de cache GPU, provoqués par le déplacement des données requises sur les différents GPUs pendant le rendu. De plus, cela impliquerait, soit des communications directes entre les GPUs, soit de devoir dupliquer des requêtes et donc également le streaming d'un grand nombre de briques depuis le CPU ou le disque. Dans les deux cas, on note une surcharge de traitement d'un point de vue de la gestion des caches sur les différents GPUs et surtout des



(a)

(b)

FIGURE 5.5 – **Partitionnement du volume multi-résolution.** Découpage d'un volume multi-résolution pour un rendu distribué sur 4 GPUs. (a) Découpage dans le volume normalisé et attribution d'une plage de données dans celui-ci pour l'accès virtuel aux données. (b) Découpage induit, dans la représentation du volume multi-résolution.

potentiels transferts coûteux supplémentaires depuis le disque jusqu'aux GPUs. Cet aspect étant un goulot d'étranglement dans notre contexte de gestion out-of-core et de streaming de données à la demande, nous justifions l'implémentation d'une méthode de rendu distribué "sort-last" pour garantir l'efficacité de notre solution de lancer de rayon volumique out-of-core multi-GPUs.

5.3.1.1 Répartition du volume multi-résolution

Le découpage du volume, pour la distribution de l'ensemble du domaine est effectué de manière à équilibrer la quantité de données sur chaque GPU. On peut alors parler d'une répartition de charge ("load-balancing" en anglais) d'un point de vue de la mémoire. Comme nous nous intéressons à un contexte où un volume, même découpé, ne tient pas en mémoire des GPUs, ces données ne peuvent bien évidemment pas être entièrement transférées dans leurs mémoires respectives. C'est donc une répartition virtuelle des données qui est effectuée. Le découpage se fait dans le volume normalisé et consiste donc à attribuer une plage de données adressable restreinte dans ce volume normalisé (voir figure 5.5a). Cette stratégie consiste à borner l'adressabilité des données en restreignant les valeurs possibles que peut prendre la position flottante normalisée $p \equiv [x, y, z] \in [borne_{inf}, borne_{sup}]^3$, avec

des intervalles $[borne_{inf}, borne_{sup}]^3$ propres à chaque GPU. De la même manière que la représentation virtuelle normalisée, utilisée dans notre modèle de gestion out-of-core, nous permet d'accéder à n'importe quel niveau de résolution l, le principe de découpage du volume normalisé impacte implicitement l'ensemble du volume multi-résolution (voir figure 5.5b).

Il existe différentes manières de découper un volume en n parts égales. Il est possible de réaliser cette décomposition, en "tranches" sur un des trois axes du volume, ou encore par blocs. Dans tous les cas il est possible de réaliser cette décomposition de manière contigue ou non. Un découpage non contigu, peu importe la forme choisie, n'est pas adapté dans ce contexte car nous perdons l'intérêt d'une cohérence spatiale dans l'utilisation des données sur les différents GPUs pour le rendu. Une décomposition du volume le long d'un axe unique (découpage en "tranches"), a tendance à favoriser une mauvaise répartition des calculs sur les différents GPUs pour certaines configurations de point vue d'observation et d'opacité des données (défini par la fonction de transfert). En effet, une vue parallèle à l'axe de découpe, sur un plan complètement opaque, implique qu'un seul GPU participe au rendu. Les données des autres GPUs étant occultées. Ceci peut apparaitre, quelque soit le niveau de zoom sur le volume. Cet effet peut être moindre si l'on propose un découpage en blocs contigus, de la manière dont est découpé le volume de la figure 5.5, pour une répartition sur quatre GPUs. Nous proposons alors d'adopter cette stratégie de découpage. Cette décomposition peut poser le même problème mais seulement avec un niveau de zoom élevé. Une telle répartition en blocs peut être obtenue par une décomposition suivant un kd-tree comme décrit dans les travaux de Fogal *et al.* [FCS+10]. Dans ce cas, il est possible d'utiliser des plans d'intersection choisis successivement sur les trois axes du volume.

Contrairement à notre découpage du volume en briques, qui nécessite un recouvrement de voxels à la frontière entre les briques pour garantir une interpolation correcte dans notre application de rendu volumique (voir section 3.4.2), le découpage du volume pour la répartition des données sur les différents processeurs d'un système multi-GPUs ne nécessite pas de prendre en compte de recouvrement aux frontières. En revanche, selon les dimensions du volume, le découpage peut s'effectuer au milieu de briques. Cela a pour conséquence de pouvoir demander l'acheminement de ces briques sur plusieurs GPUs en même temps pendant le rendu, et donc, de devoir gérer l'accès concurrent à ces briques du coté du CPU et du stockage de masse. Une autre solution est d'empêcher ce phénomène en décalant la plage d'adressage des différents GPUs, de manière à ce que les bornes de ces plages correspondent uniquement à des briques entières.

Les autres aspects de la méthode proposée ne diffèrent pas d'une approche classique de rendu distribué avec une stratégie "sort-last". Le découpage et l'attribution de la plage de données adressables sur les différents GPUs se fait avant de commencer la boucle principale de rendu interactif. Au cours de celle-ci, les GPUs réalisent en parallèle le rendu local sur les données de la sous-partie du volume global qui leur est attribué. A la fin de cette passe, chaque GPU possède, dans sa propre mémoire, un buffer de pixels de la taille du viewport total comprenant le résultat de leur rendu local. Tous les GPUs du noeud de calcul sont mis à contribution pour réaliser ce rendu. Il faut ensuite recomposer une image finale, composée des n images venant des n GPUs. Cette étape de composition (appellée "compositing" en anglais) est détaillée dans la section suivante.

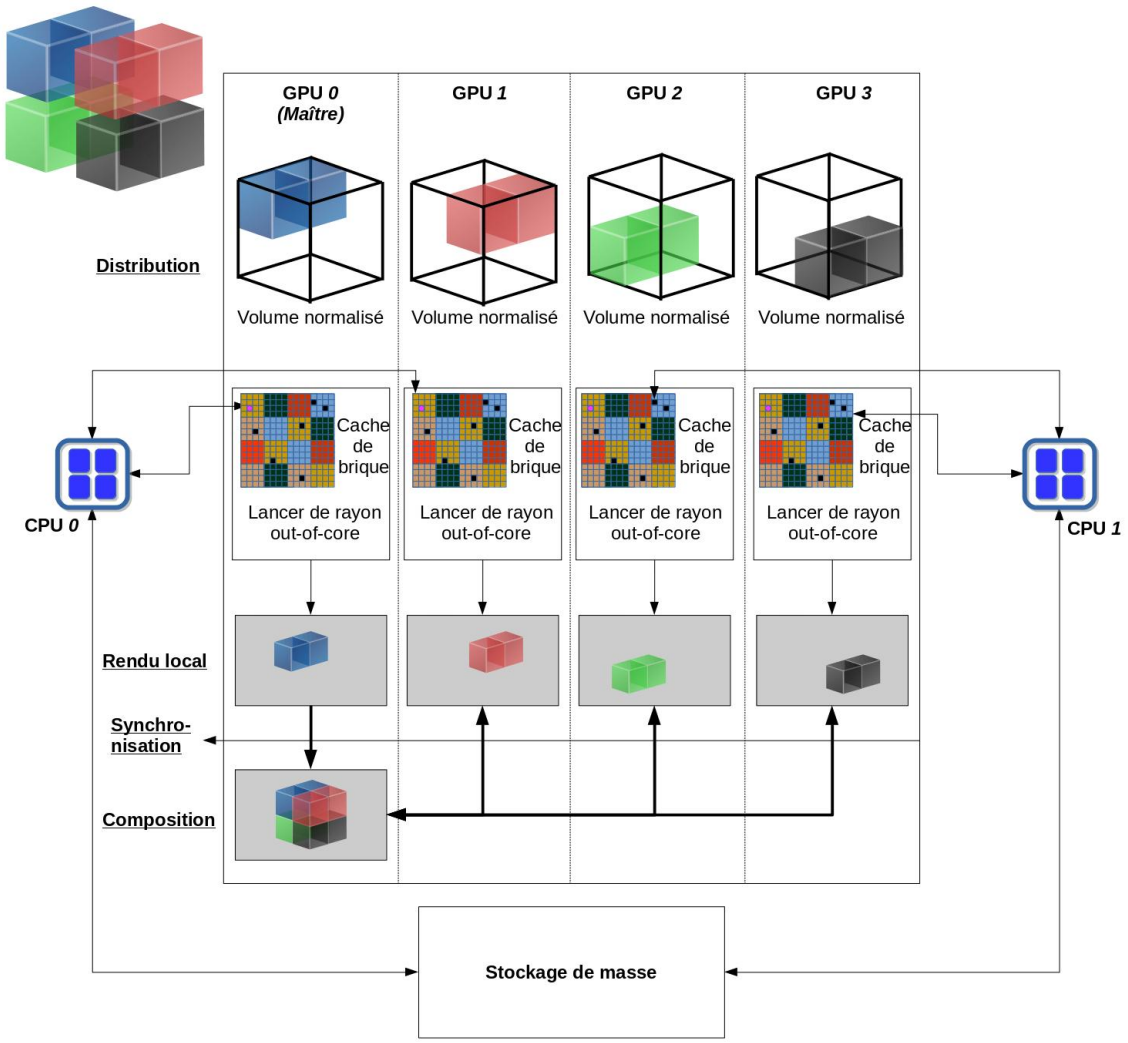


FIGURE 5.6 – **Pipeline d'une approche "sort-last" de rendu volumique distribué out-of-core.** Exemple de rendu distribué sur un système composé de quatres GPUs et de deux CPUs.

5.3.1.2 Composition du rendu

Une approche de rendu distribué "sort-last" implique une étape de recomposition de l'ensemble des images produites par le rendu de tous les GPUs, pour construire une image finale pour la visualisation. La construction de cette image permet de prendre en compte la contribution du rendu des n GPUs en accumulant la couleur et l'opacité, pour chaque pixel $[x, y]$ de l'image finale, des n pixels correspondants. Ce calcul est tout à fait adapté à une implémentation en parallèle. Nous proposons donc une méthode pour effectuer cette étape entièrement sur GPU, et ainsi éviter des transferts des buffers de pixels entre les GPUs et les CPUs du noeud de calcul. Dans notre approche, la composition est réalisée sur un seul des GPUs (arbitrairement celui de rang zéro) et implique de devoir transférer les données de tous les GPUs vers celui-ci. Nous décrivons dans la section suivante comment nous proposons d'opérer pour réaliser ces transferts.

En rendu volumique direct, l'ordre de composition est important afin de tenir compte correctement de l'accumulation de la couleur et de l'opacité à travers le volume. Une barrière de synchronisation est alors utilisée, pour que le GPU responsable de ce calcul attende que l'ensemble des GPUs aient terminé leur rendu (lui compris). Lors d'une passe de rendu, chacun des GPUs sauvegarde la plus courte distance entre l'origine d'un rayon et son intersection avec la boîte englobante de la sous-partie du volume dont il est responsable. Après que le rendu soit terminé, ces distances (représentées avec un nombre flottant par GPU) sont regroupées et triées, sur le CPU, de la plus proche à la plus plus éloignée. La liste triée ainsi obtenue est ensuite communiquée au GPU qui va procéder au calcul de composition des n images. Celui-ci réalise alors la composition en accumulant la valeur des pixels d'avant en arrière ("front-to-back") en utilisant l'opérateur *OVER*, en une seule passe avec un kernel CUDA.

L'utilisation d'une approche multi-GPUs avec une distribution "sort-last", peut impliquer de réaliser certaines parties inutiles du rendu, dans le sens où elles ne seront pas visibles car masquées par une autre partie du volume (rendu par un autre GPU). En effet, certaines parties qui ont contribué à l'échantillonnage le long d'un rayon pendant le rendu sur un GPU ne contribueront pas forcément à l'image finale pendant l'étape de composition. Le concept de terminaison précoce de rayon peut bien être utilisé de manière locale sur chaque GPU, mais il n'est pas possible de profiter d'une connaissance de l'opacité accumulée de manière globale pendant le rendu. Plus important encore, cela impacte directement l'aspect de streaming des briques à la demande puisque certaines d'entre elles seront chargées en cache alors qu'elles ne participeront pas au rendu final. Une approche multi-GPUs, dans ce contexte, apporte beaucoup d'avantages d'un point de vue de la gestion de la mémoire et des performances par son principe de distribution de l'un comme de l'autre. Cependant, elle apporte également certaines contraintes qui peuvent faire perdre de l'efficacité dans la combinaison d'une méthode "ray-guided" avec un système de gestion out-of-core.

5.3.2 Distribution de la structure d'adressage virtuel et communications

Nous détaillons dans cette section, la stratégie de distribution de notre solution out-of-core, basée GPU, sur des plateformes composées de n GPUs. La stratégie proposée est dirigée par le choix d'une approche de rendu distribué "sort-last".

Nous proposons d'utiliser une seule configuration de virtualisation et celle-ci est respectée par l'ensemble des n GPUs. Cette configuration est propre au volume et n'est pas dépendante du GPU. Elle comprend : la manière d'adresser les données d'un volume en terme de nombre de niveaux de virtualisation, la taille d'un *bloc PT*, les dimensions des briques et la taille des caches. Chaque GPU possède ensuite en mémoire sa propre structure d'adressage virtuel, ainsi que sa propre instance de gestionnaire de cache, afin de gérer les ressources qui lui sont associées. De cette manière, tous les GPUs du noeud de calcul sont alors capables d'accéder à n'importe quelle brique du volume multi-résolution, tout en ayant un cache de données propre. Ils sont ainsi indépendants dans leur gestion out-of-core des données, ayant chacun le contrôle sur leur propre utilisation des briques, des requêtes de données, de la gestion du streaming de celles-ci et de leur mise en cache. Le cache de briques CPU, quand à lui, n'existe qu'en un seul exemplaire.

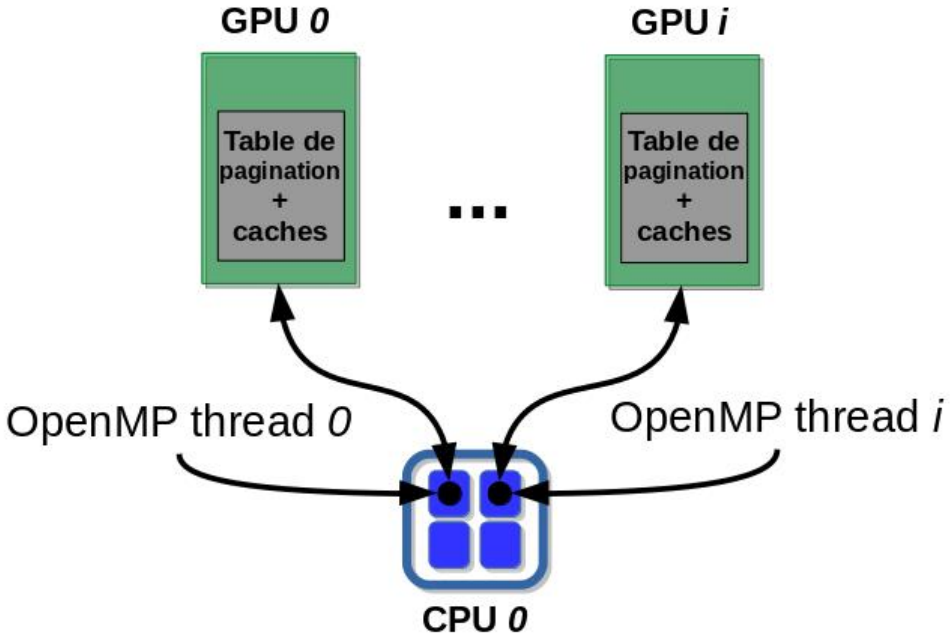


FIGURE 5.7 – **Distribution de la structure d'adressage virtuel et communications multi-threads.** Chaque GPU implémente une instance de la structure de table de pagination multi-résolution multi-niveaux et gère son propre système de caches. On utilise openMP pour lier le contexte CUDA de chacun des GPU sur les coeurs du CPU correspondant.

OpenMP et CUDA

Nous proposons d'utiliser la bibliothèque de parallélisation à mémoire partagée openMP afin de proposer une stratégie multi-thread sur CPU. Il est ainsi possible de paralléliser les différentes actions réalisées sur le CPU, en particulier la gestion des requête de briques, et d'assurer une stratégie de communication efficace avec les différents GPUs. Comme nous nous concentrons sur une représentation multi-GPUs, regroupés au sein d'un seul noeud de calcul, nous n'avons pas la nécessité d'élaborer une configuration de communication intra-noeuds et nous nous concentrons donc sur les communications inter-noeuds, entre les différents CPUs et GPUs. Notre stratégie est de créer autant de threads openMP qu'il y a de GPUs configurés sur la machine. Chaque thread openMP est lié à un contexte CUDA appartenant à un seul GPU. Cela permet de définir, pour toute la durée d'exécution de l'application, quel thread CPU est en charge de communiquer avec quel GPU, et cela pour tous les GPUs présents sur le noeud de calcul.

La répartition des différents threads au sein de la topologie d'un noeud de calcul hybride, multi-GPUs, multi-CPUs, a son importance pour garantir de bonnes performances de communication entre les différentes unités de calcul. Nous proposons d'utiliser une répartition contrôlée, en disposant les threads de manière à ce que chacun soit exécuté sur un coeur physique du CPU qui est relié au GPU (communément par un bus PCI Express) avec lequel le thread en question est associé. Cette stratégie multi-thread sur CPU permet de répartir la charge des transferts de données sur les différents GPU (à condition que le matériel le permette également). Ainsi chaque thread OpenMP est en charge de gérer uniquement les requêtes de briques du GPU avec lequel il est associé et les requêtes des

différents GPUs peuvent donc être gérées en parallèle.

L'implémentation avec openMP au sein de notre modèle, n'implique qu'une seule région parallèle englobant l'ensemble de l'application. Au début de cette région, on attribue le thread à un contexte CUDA puis, on y exécute d'abord en parallèle sur chaque GPU, la création de toutes les ressources nécessaires à la gestion out-of-core, l'adressage virtuel et la gestion du système de cache. Il est ensuite possible de lancer dans cette région parallèle, la partie applicative qui gère sur le GPU, la boucle principale de rendu et le maintien de la structure au fur et à mesure de celui-ci, ainsi que les différentes interactions de l'utilisateur. L'ensemble des ressources permettant de gérer ces interactions est néanmoins partagé par tous les threads openMP afin de faire le nécessaire au niveau de chacun d'entre eux lors d'un changement de point de vue ou de fonction de transfert par exemple.

Les communications entre les GPUs sont assurées par le système de communication "peer-to-peer" de CUDA en combinaison avec le concept d'adressage virtuel unifié (UVA pour "Unified Virtual Adressing") [Nvi]. Cela nous permet d'accéder à une zone mémoire d'un GPU A, directement depuis un kernel CUDA exécuté sur un GPU B en donnant le pointeur de cette zone mémoire du GPU A en paramètre du kernel. L'API CUDA rend cela possible uniquement si les deux GPUs sont dans le même espace d'adressage. Si ce n'est pas le cas, nous repassons sur une stratégie moins optimale, consistant à transférer explicitement les données du GPU A vers le CPU, puis du CPU vers le GPU B. La présence de la technologie NVlink [8] de NVIDIA sur le noeud de calcul utilisé permettrait de profiter de communications directes très rapides entre les différents GPUs.

5.4 Visualisation distante

L'approche proposée inclut une solution de visualisation interactive distante, qui dissocie la machine chargée de calculer le rendu, de celle chargée de l'affichage. Notre solution est basée sur un système client-serveur, connectés entre eux pour communiquer via le réseau. Dans ce système, le client est un PC standard avec un dispositif d'affichage et le serveur est un noeud de calcul multi-GPUs. Le premier est responsable de l'affichage alors que le deuxième s'occupe du calcul du rendu.

A l'initialisation d'une connexion, le client se charge de communiquer les informations d'affichages au serveur. Typiquement, la taille du "viewport". Nous sommes dans un contexte de rendu interactif et nous assurons donc la possibilité à l'utilisateur d'interagir avec le modèle qu'il visualise. Le client est alors chargé de communiquer toutes les informations nécessaires au serveur pour le rendu interactif. Celles-ci comprennent : les mouvements de caméra, les changements de la fonction de transfert ou encore toute interaction avec la scène ou le modèle. Le serveur, de son côté, assure la production du rendu interactif avec notre solution de lancer de rayon volumique out-of-core, multi-GPUs. Le résultat de chaque passe de rendu est communiqué au client sous forme d'un buffer de pixels compréssé. Du côté du client, l'affichage est réalisé avec OpenGL en alimentant le contenu d'une texture 2D mise à jour à chaque arrivée d'un nouveau buffer de pixels provenant du rendu généré par le serveur. Sur ce dernier, toute dépendance à OpenGL est écartée et l'ensemble des actions GPU est réalisé avec CUDA.

8. https://www.nvidia.com/fr-fr/data-center/nvlink/

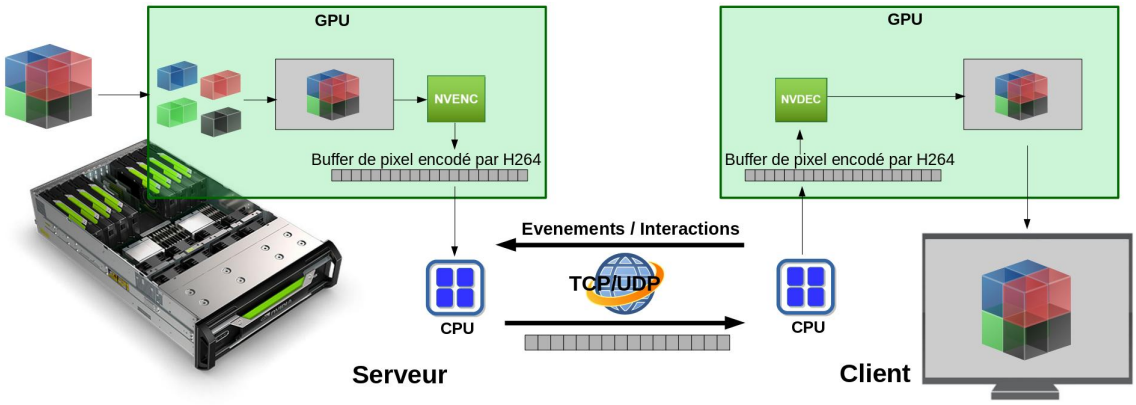


FIGURE 5.8 – **Système client-serveur de visualisation interactive distante.** Un client léger se charge d'afficher le contenu d'un flux de pixels compressé en H.264 sur GPU, envoyés via le réseau par un serveur distant. Ce dernier s'occupe d'effectuer le rendu avec notre solution multi-GPUs out-of-core. Les communications du client vers le serveur permettent de mettre à jour les informations utiles au rendu, qui proviennent des interactions de l'utilisateur.

Nous utilisons une compression H.264 sur les buffers de pixels calculés sur le serveur et envoyés au client par le réseau. Cette compression avec perte offre une grande rapidité de compression, à un taux très intéressant. De plus, elle est entièrement gérée de manière native sur GPU pour l'encodage et le décodage. Cette solution permet de garantir l'interactivité tout au long de la visualisation, en réduisant le coût des transferts de données via le réseau, pour une surcharge de temps de compression / décompression moindre. Cette stratégie est garantie en implémentant la solution NVidia NVenc [9] sur le client et sur le serveur.

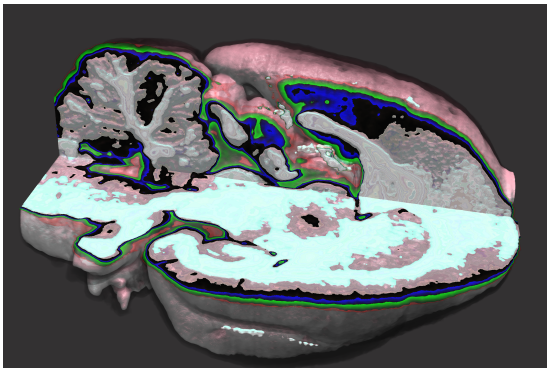
Les communications réseau sont assurées par le protocole UDP, proposant ainsi un système de streaming performant d'un flux vidéo, mais ne garantissant pas la perte ou la duplication de paquets et donc, dans notre contexte, d'images. Aussi bien du coté du client que de celui du serveur, les envois et les receptions de messages se font dans un thread dédié. Cette stratégie permet de ne pas bloquer le thread principal, chargé du rendu pour le serveur et de l'affichage pour le client.

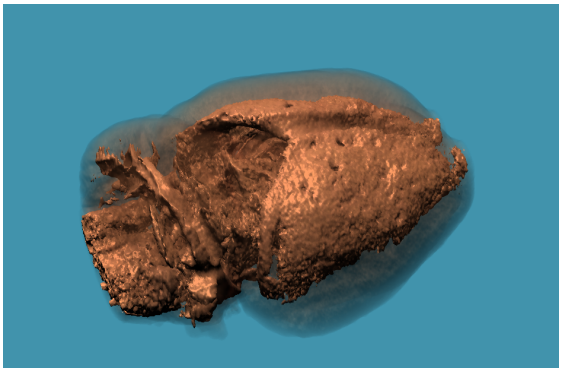
La solution client-serveur proposée permet à une machine de bureau classique de tirer parti de la puissance de calcul d'un serveur, en basculant le coût du rendu sur une machine plus adaptée, du point de vue de son architecture, de sa puissance de calcul, et de ses capacités de stockage. La surcharge de compression / décompression du flux vidéo, ainsi que le coût de son transfert à travers le réseau sont recouverts par l'efficacité du rendu sur une machine multi-GPUs hautes performance, comparé à une solution d'un calcul local.

9. https://developer.nvidia.com/nvidia-video-codec-sdk

Chapitre 6

Evaluations et discussions





Sommaire

6.1 Introduction

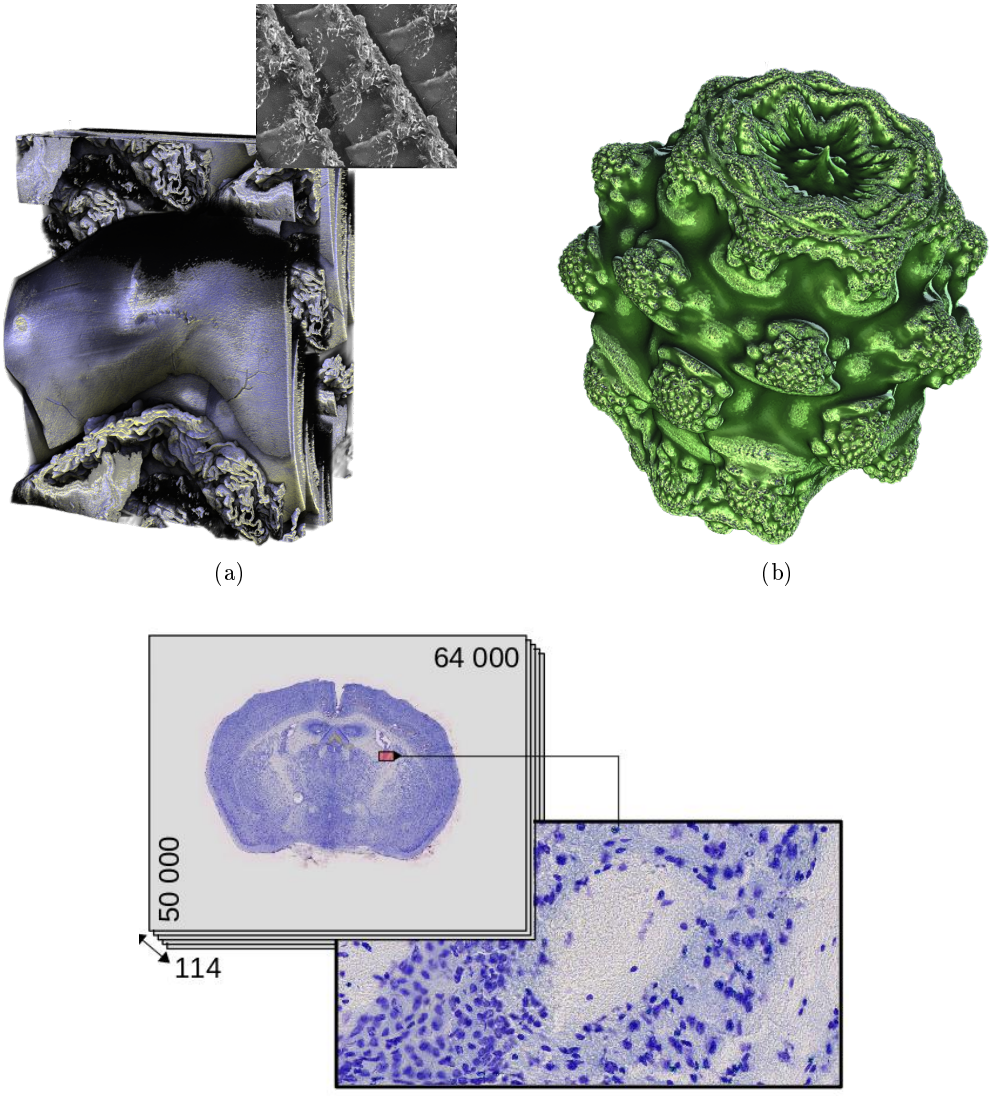
Dans ce chapitre, nous présentons une évaluation des différentes solutions mises en oeuvres dans cette thèse. Cette évaluation porte sur notre modèle de gestion out-of-core en lui même, ainsi que sur son utilisation dans différents cadres applicatifs de visualisation. Après avoir détaillé les différents jeux de données que nous utilisons pour évaluer nos solutions, nous nous intéresserons à l'empreinte mémoire GPU utilisée par notre modèle, pour l'ensemble des ressources qui permettent de le maintenir. Nous étudions ensuite nos deux cadres applicatfis de visualisation précédement décrits : notre solution de mircroscopie virtuelle et notre solution de rendu volumique direct avec une méthode de lancer de rayon multi-GPUs. A la fin de ce chapitre, nous discutons de l'utilisation de notre approche pour des traitements de données à la demande pendant la phase de visualisation interactive. Pour finir, nous étudions les forces et les faiblesses de nos solutions de visualisation haute performance, mises en oeuvre pour de très grands volumes de données.

6.2 Jeux de données

Les résultats présentés dans ce chapitre font l'objet d'une étude réalisée sur plusieurs jeux de données volumiques, représentés sous forme de grille régulière 3D et décrits ci-après. Le modèle de gestion out-of-core que l'on propose ainsi que les deux applications de visualisation mises en oeuvre, ont la capacité de gèrer des données aux caractéristiques différentes. Afin d'illustrer cela, nous utilisons trois jeux de données qui varient sur trois critères : *i)* leur volumétrie, *ii)* l'encodage des voxels et *iii)* leurs caractéristiques dûes à leur acquisition. Plusieurs jeux de données biomédicales ont été produit dans le cadre du projet 3DNeuroSecure avec différentes techniques d'acquisition. Nous utilisons deux de ces jeux de données, provenant de deux méthodes différentes de génération, amenant ainsi à des ensembles de données aux caractéristiques très différentes. Ces données nous permettent également d'avoir différentes échelles de taille et de volumétrie. Nous pouvons ainsi évaluer notre approche sur un volume de très grande taille et un autre moins conséquent. Afin de disposer d'un volume de taille intérmédiaire et qui n'est pas issu d'une acquisition provenant d'un appareil biomédical, nous avons généré un autre volume de manière empirique. Les jeux de données utilisés sont les suivants, par ordre croissant de poids mémoire :

1. **Hypocampe de primate**
   Ce volume *Hippocampus* provient de l'acquisition de l'hypocampe d'un cerveau de primate par un microscope à feuillet de lumière. La séquence d'image au format tiff obtenue après acquisition, est reconstruite en un seul volume au format raw.
   — **Dimensions :** $2160 \times 2560 \times 1072$
   — **Encodage des pixels :** luminence sur 16 bits
   — **Poids :** 11.8 GB
2. **Mandelbulb**
   Ce volume *Giga Mandelbulb* est une fractale réalisée avec le logiciel Mandelbulb3D [10].
   — **Dimensions :** $4352 \times 4352 \times 4352$
   — **Encodage des pixels :** couleur RGBA
   — **Poids :** 329.7 GB

10. http://www.mandelbulb.com/2014/mandelbulb-3d-mb3d-fractal-rendering-software/



(a) (b)

(c)

FIGURE 6.1 – **Illustrations des jeux de données utilisés.** (a) *Hippocampus*, (b) *Giga Mandelbulb*, (c) *Tera Mouse Brain*

3. **Cerveau de souris**

   Ce volume *Tera Mouse Brain* est une pile de coupes histologiques provenant de l'acquisition d'un cerveau de souris avec un scanner de lame. Le volume est obtenu après un processus de recalage de l'ensemble des lames histologiques. Ces données de très grande dimension sont générées avec une précision de 0.2 µm dans le plan (x - y) et de 200 µm pour l'épaisseur d'une coupe, ce qui implique une très forte anisotropie.

   — **Dimensions :** $64000 \times 50000 \times 114$
   — **Encodage des pixels :** couleur RGBA sur 32 bits.
   — **Poids :** 1.4 TB

Nom	Dimensions	Encodage des pixels	Poid raw
Hippocampus	$2160 \times 2560 \times 1072$	Luminence sur 16 bits : 2 octets	11.8 GB
Giga Mandelbulb	$4352 \times 4352 \times 4352$	Couleur RGBA : 4 octets	329.7 GB
Tera Mouse brain	$64000 \times 50000 \times 114$	Couleur RGBA : 4 octets	1.45 TB

TABLE 6.1 – **Synthèse des caractéristiques des jeux de données.**

6.3 Empreinte mémoire GPU du modèle de gestion out-of-core

Une des contributions de cette thèse est un modèle de gestion out-of-core, par un système d'adressage virtuel avec une structure de table de pagination multi-niveaux, multi-résolution, entièrement géré sur le GPU. Le fait de proposer une gestion complète sur le GPU, implique de devoir allouer et maintenir, sur celui-ci, plusieurs ressources dédiées à cette tâche. Nous étudions ici l'impact d'une telle gestion sur l'empreinte mémoire nécessaire sur le GPU.

6.3.1 Impact de la taille des briques

Les différentes ressources nécessaires à cette gestion sont, le *buffer de requête* (utilisé pour gérer les requêtes de briques dans l'ensemble du volume multi-résolution), les ressources liées au maintien de chaque niveau de cache, c'est à dire, un *buffer d'usage* et une LRU par cache, ainsi que la racine de la structure hiéarchique (la *MRPD*), qui est le seul niveau à être entièrement physiquement présent en mémoire GPU. Les autres ressources nécessaires sont les caches eux-mêmes, comprenant bien sûr, le cache de brique et éventuellement les caches des niveaux intermédiaires de virtualisation.

La table 6.2, montre les résultats de l'étude de l'empreinte mémoire GPU de l'ensemble de ces ressources, pour les trois jeux de données présentés à la section 6.2, en fonction de la taille des briques. Tous ces résultats sont exprimés pour un cache de brique d'une taille de 4 GB (ce qui correspond environ à la capacité moyenne d'un GPU qu'il est possible de trouver dans un PC standard de nos jours) et avec un seul niveau de virtualisation dans la structure de table de pagination. Les résultats obtenus dans cette table, montrent premièrement que, le choix de la taille des briques a un impact direct et assez important sur cette utilisation de la mémoire GPU. En effet, plus l'on choisit de manipuler des briques volumineuses, plus la quantité de mémoire nécessaire à la gestion out-of-core est faible. Cette indication est à prendre en considération en plus de l'étude faite par Thomas Fogal *et al.* [FSK13] en 2013, qui montre qu'il est plus intéressant pour une application de lancer de rayon volumique sur GPU, d'avoir des briques de taille de l'ordre de 32^3 voxels. Deuxièmement, nous pouvons constater qu'avec des briques de taille moyenne (64^3 voxels),

Jeu de données		Hippocampus	Giga Mandelbulb	Tera Mouse brain
Briques 32^3	MRPD (kB)	6049	81382	400228
	Buffer de requête (kB)	1512	20345	100057
	Buffer d'usage (kB)	244	122	122
	LRU (kB)	1709	854	854
	Total (MB)	9.51	102.70	501.26
	# briques dans le volume	211688	2874995	14328922
	# éléments max en cache	61036	30518	30518
Briques 64^3	MRPD (kB)	772	10283	50073
	Buffer de requête (kB)	193	2570	12518
	Buffer d'usage (kB)	30	15	15
	LRU (kB)	213	106	106
	Total (MB)	1.20	12.97	62.71
	# briques dans le volume	26728	359539	1824922
	# éléments max en cache	7630	3815	3815
Briques 128^3	MRPD (kB)	106	1313	6262
	Buffer de requête (kB)	26	328	1565
	Buffer d'usage (kB)	3.8	1.9	1.9
	LRU (kB)	26	13	13
	Total (MB)	0.16	1.65	7.84
	# briques dans le volume	3608	45107	260922
	# éléments max en cache	954	477	477

TABLE 6.2 – **Empreinte mémoire des différentes ressources sur le GPU selon la taille des briques.** La configuration utilisée comprend un seul niveau de virtualisation et un cache de brique de 4 GB.

notre système requiert seulement quelques dizaines de MB sur le GPU, pour de très grands volumes qui dépassent le TB.

6.3.2 Impact des niveaux de virtualisation

La première table, nous permet de montrer que c'est la *MRPD* qui est la plus coûteuse en mémoire. Celle-ci peut cependant être réduite en utilisant plusieurs niveaux de virtualisation. Ce phénomène est illustré dans la table 6.3. Nous notons qu'il est possible de réduire très largement l'espace mémoire occupé par la *MRPD* avec un seul niveau de virtualisation supplémentaire. Le fait d'ajouter un niveau de virtualisation implique de devoir gérer un autre cache (un cache contenant des *blocs PT*) et donc d'allouer un *buffer d'usage* et une LRU pour le maintien de celui-ci. Cependant, ces ressources sont très peu coûteuses en mémoire, et la surcharge qu'elles provoquent est très faible par rapport aux gains apportés par la réduction de la *MRPD*. Malgrè le fait que le volume en exemple soit un grand volume, qui dépasse un TB, il est inutile, d'un point de vue de l'empreinte mémoire, d'avoir trois niveaux de virtualisation. Une telle configuration deviendrait utile pour des données encore plus volumineuses ou alors pour des applications qui nécessiteraient un découpage en briques de très petites tailles. Cela vient appuyer le fait que cette structure est très peu profonde tout en offrant une très grande capacité d'adressage même sur des volumes avec un grand nombre de niveaux de résolution. Par souci d'exhaustivité de l'analyse, il serait intéressant de quantifier l'impact du temps d'adressage selon le nombre de niveau de virtualisation. Cependant, du fait que cet impact n'a pas été ressenti en pratique, il ne nous a pas semblé primordial de réaliser cette étude.

Nombre de niveaux de virtualisation	1 niveau	2 niveaux	3 niveaux
MRPD (kB)	400228	12	0.17
Buffer de requête (kB)	100057	100057	100057
Buffer(s) d'usage (kB)	122	130	138
LRU(s) (kB)	854	910	966
Total (MB)	501.26	101.10	101.16
# entrées dans la MRPD	25014252	763	11

TABLE 6.3 – **Empreinte mémoire des différentes ressources sur le GPU selon le nombre de niveaux de virtualisation.** Les résultats sont obtenus avec le volume *Tera Mouse brain* de 1.45 TB, des briques de 32^3 voxels et des *bloc PT* de 32^3 entrées.

6.3.3 Impact de la taille du cache de brique

La table 6.4, a pour but de montrer l'évolution de la quantité mémoire nécessaire en fonction de la taille du cache de brique. Cette étude est réalisée sur le volume *Hippocampus* de 11.8 GB, avec des briques de 64^3 voxels et un seul niveau de virtualisation. Nous remarquons que la taille du cache de brique agit très peu sur le total des ressources nécessaires. En effet, cela impacte uniquement les ressources qui sont liées à la gestion du cache de brique, c'est à dire, le *buffer d'usage* et la LRU correspondant. Ces deux ressources occupent peu d'espace dans notre système de gestion out-of-core sur GPU, il est alors possible d'utiliser un cache de brique conséquent (en fonction des capacités du GPU) et ainsi être plus à l'aise pour des applications qui nécessitent un espace de travail important, tout en impactant que très peu la quantité de mémoire nécessaire à la gestion du modèle.

Taille du cache de brique	4 GB	8 GB	16 GB
MRPD (kB)	772	772	772
Buffer de requête (kB)	193	193	193
Buffer d'usage (kB)	30	61	122
LRU(s) (kB)	213	427	854
Total (MB)	1.20	1.45	1.94
# briques dans le volume	26728	26728	26728
# elements max en cache	7630	15260	30520

TABLE 6.4 – **Empreinte mémoire des différentes ressources sur le GPU selon la taille du cache de brique.** Les résultats sont obtenus avec le volume *Hippocampus* de 11.8 GB pour des briques de 64^3 voxels et un seul niveau de virtualisation.

Nous avons vu que la *MRPD* n'était pas un facteur bloquant du fait qu'il est possible de réduire considérablement sa taille par l'ajout de peu de niveaux de virtualisation, si nécessaire pour de très grand volumes de données. Nous avons vu également qu'il était possible d'allouer un grand cache de brique sans impacter lourdement la quantité mémoire nécessaire pour les ressources utiles à la gestion out-of-core entièrement sur GPU. Le *buffer de requête* est quand à lui un facteur limitant de notre méthode puisque sa taille est directement liée à la taille du volume multi-résolution. Cependant, les résultats obtenus dans les tables 6.2, 6.3 et 6.4 viennent renforcer ce qui a été montré dans la section 3.8.1 : son occupation mémoire reste raisonnable, même pour de très grands volumes de données (100 MB pour le volume *Tera Mouse brain* de 1.45 TB avec de petites briques de 32^3 voxels).

6.4 Performances

Dans cette section, nous évaluons les performances des différentes solutions de visualisation de grands volumes de données proposées dans le cadre de cette thèse. Nous proposons d'analyser le comportement des deux applications de visualisation que l'on a élaborées et présentées respectivement dans les chapitres 4 et 5, qui utilisent notre modèle de gestion out-of-core présenté dans le chapitre 3. Afin d'évaluer au mieux les performances de ces applications, nous proposons d'étudier, aussi bien, la fréquence d'affichage du rendu, que les temps des chargements de briques.

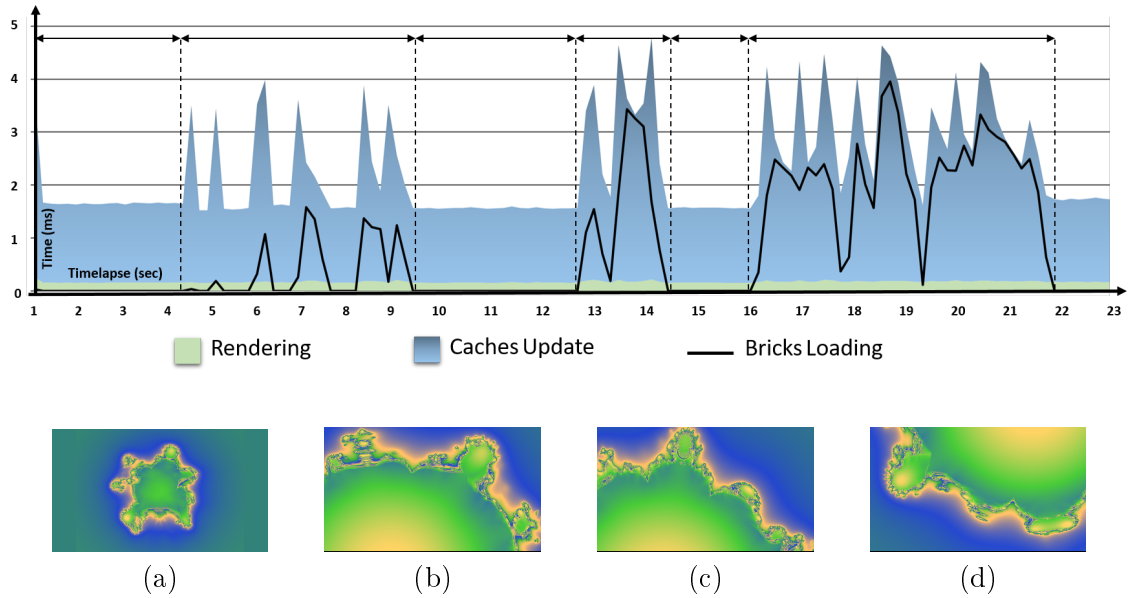
6.4.1 Microscope Virtuel

L'ensemble des résultats présentés dans cette section ont été obtenus sur un système équipé d'un GPU NVIDIA GForce Titan X avec 6 GB de VRAM, un CPU Intel i7 4790K 4 GHz et 32 GB de RAM. Nous utilisons l'interopérabilité entre CUDA 8.0 et OpenGL 4.5 pour un rendu sur un viewport HD 1920×1080.

6.4.1.1 Analyse globale

La figure 6.2 a été obtenue par benchmarking de notre application de microscope virtuel dans sa version de rendu classique, en 2D. Elle présente la comparaison du temps passé sur trois opérations : le rendu et la gestion des caches, tous deux réalisés sur le GPU, et le chargement des briques, géré de manière asynchrone sur le CPU. L'étape de gestion des caches comprend : la mise à jour des LRU(s) et la gestion des requêtes de données, fait à chaque itération, ainsi que l'écriture des briques en cache GPU et la mise à jour de la structure, réalisées quand de nouvelles briques sont envoyées au GPU. Ces temps sont calculés avec le jeu de données *Giga Mandelbulb*, avec des briques de 64^3 voxels, en réalisant le scénario suivant : un zoom depuis le plus faible niveau de résolution jusqu'au plus fort niveau de résolution, une navigation dans les différentes coupes de la pile d'image le long de l'axe z, et une navigation dans le plan d'une coupe sur les axes x et y. Un petit laps de temps, sans aucune interaction, a délibéremment été maintenu entre chacune de ces étapes, afin d'illustrer des périodes sans aucun chargement de brique.

Nous pouvons remarquer en premier, que le temps de rendu est constant et très faible (environ 0.2 ms) pendant l'ensemble du scénario. Ce temps est constant car le chargement des briques est réalisé de manière asynchrone et n'interfère pas avec le rendu. De plus, nous ajoutons que cette application fournit une fréquence d'affichage de plusieurs centaines d'images par seconde. Une si haute fréquence de rendu peut être atteinte car l'application proposée ne requiert pas de lourde charge de calcul en plus de la gestion des données. Les temps de gestion de la structure et des caches sont inférieur à 2 ms, avec des pics correspondants aux chargements de briques. Ces temps sont raisonnables et permettent de valider une navigation interactive dans la scène. Malgrè le fait que l'ensemble de la gestion des caches et de la structure d'adressage virtuel soit réalisée sur le GPU, le taux d'occupation de celui-ci est d'environ 5%. Cette faible charge d'occupation laisse donc beaucoup de place à une application de visualisation qui nécessite une grande puissance de calcul. Le goulot d'étranglement se produit au chargement des données, entre le disque et le CPU. L'empreinte du calcul sur le GPU est suffisamment faible pour permettre de lourdes charges de rendu.



(a) (b) (c) (d)

FIGURE 6.2 – **Utilisation du GPU et temps de chargement des briques.** Comparaison du temps passé (en ms) pour le rendu, la gestion des caches sur GPU et le chargement des briques, sur un scénario d'nteractions de 23 secondes. Cette séquence de navigation dans le jeu de donnée *Giga Mandelbulb* comprend : un zoom, une navigation dans la pile d'image sur l'axe z et une navigation dans une coupe sur les axes x et y. Ces mesures sont obtenues avec notre application de microscope virtuel 2D. Les quatres images (a) à (d) sont des captures écrans du scénario précédement décrit.

6.4.1.2 Chargement des briques

Afin d'analyser l'impact du temps de chargement des briques dans notre application de visualisation, nous proposons deux métriques différentes. La table 6.5 contient les mesures de temps moyen pour :

1. le chargement d'une seule brique, depuis sa requête par l'application sur le GPU, jusqu'au moment où elle est disponible pour le rendu,
2. le chargement d'une vue complète correcte, avec toutes les briques nécessaires dans le cache GPU.

Ces mesures ont été réalisées à partir d'un cache de brique complètement vide sur le GPU. Nous proposons ensuite deux scénarii différents :

1. un scenario au pire-cas, où toutes les briques sont seulement présentes sur le disque,
2. un scénario où toutes les briques sont présentes dans notre cache intermédiaire, sur le CPU

Dans le premier cas, les temps donnés, comprennent, 1) la gestion des requêtes des briques (la création de la liste des requêtes sur le GPU et son transfert au CPU), 2) le temps de lecture sur le disque, 3) la décompression LZ4, 4) le transfert du CPU vers le GPU et 5) la mise à jour de la structure sur GPU (référencement dans la table de pagination et mise à jour de la LRU). Le deuxième scénario, quand à lui, comprend uniquement les étapes 1, 4 et 5.

Jeux de données		Hippocampus	Giga Mandelbulb	Tera Mouse brain
Resolution du volume		$2160\times 2560\times 1072$	4352^3	$64000\times 50000\times 114$
Taille du volume (GB)		11.8	329.7	1459
Taille des voxels (B)		2	4	4
Taille des briques (voxels)		64^3	64^3	64^3
Taille des briques (kB)		524	1048	1048
Une brique	disque → cache GPU (ms)	5.15	13.60	11.76
	cache CPU → cache GPU (ms)	2.12	2.90	2.80
Une vue complète	disque → cache GPU (s)	1.97	7.43	6.57
	cache CPU → cache GPU (s)	0.83	1.62	1.64

TABLE 6.5 – **Analyse du temps de chargement des briques.** La première partie de cette table comprend une description des jeux de données et de leur configuration. La deuxième partie de la table montre les temps de chargement moyen d'une seule brique et d'une vue complète, pour deux scénarios différents.

Ici, le but n'est pas de réaliser une étude comparative par rapport à la taille des briques. De telles analyses peuvent être retrouvées dans la littérature, notamment dans les travaux de Fogal *et al.* [FSK13]. Nous évaluons les performances de notre solution out-of-core en application avec une méthode de visualisation. Notre scénario au pire cas, de chargement d'une vue complète, entièrement depuis le disque, prend quelques secondes. Cependant, il est important de noter que les temps donnés dans la table 6.5 sont mesurés à partir d'un cache GPU complètement vide. En pratique, pendant une phase de visualisation interactive, l'espace de travail nécessaire au rendu d'une image est en partie présente sur le disque, en partie dans le cache CPU et en partie dans le cache GPU.

6.4.1.3 Extension 3D multi-vues

Nous étudions ici la solution d'extension de notre application de microscope virtuel à la création d'images 3D multi-vues. Nous utilisons pour cela un autre jeu de données issu de l'acquisition d'un cerveau de souris, représenté par un volume d'une dimension de $5520 \times 7000 \times 800$ en couleur RGBA, de 123 GB. L'affichage du rendu est réalisé sur un écran autostéréoscopique HD (16:10) comprenant huit points de vue.

Cette approche de microscopie virtuelle en 3D se comporte de la même manière que l'application qui génère des images 2D en ce qui concerne l'aspect de la gestion out-of-core. Le changement à noter d'un point de vue des performances, se trouve au niveau du temps nécessaire pour le rendu. Le temps moyen nécessaire à la création d'une image multi-vue composée de huit slices est de 30 ms. Cela comprend la sélection des différentes vues, la projection de chaque slice sur chaque vue, la composition de l'alpha pour chaque slice et l'étape de multiplexage. Ce traitement dépend de la taille de l'écran 3D et du nombre de vues qui le compose. Il n'est pas affecté par la taille des briques où par le niveau de détail demandé pour l'affichage. Ces temps valident la navigation et la visualisation de ces images 3D multi-vues en temps interactif ($\sim$30 images par seconde) sur des volumes qui excèdent la capacité mémoire du GPU.

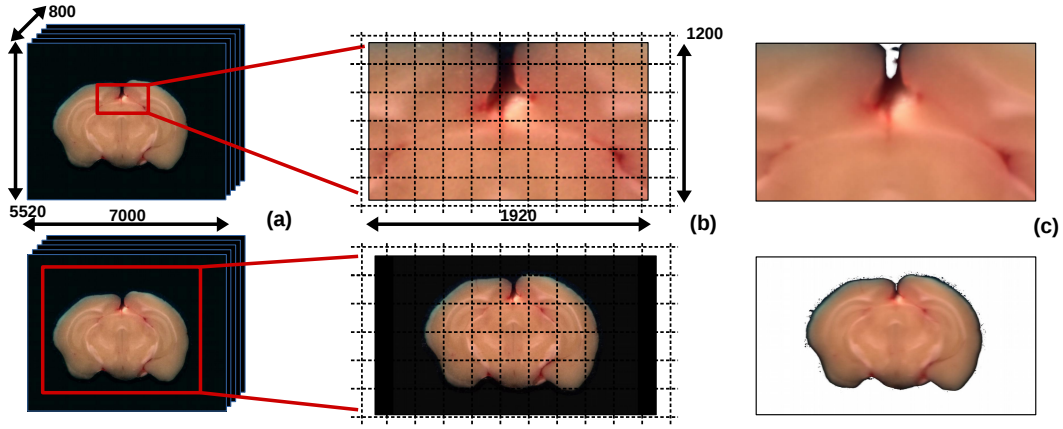
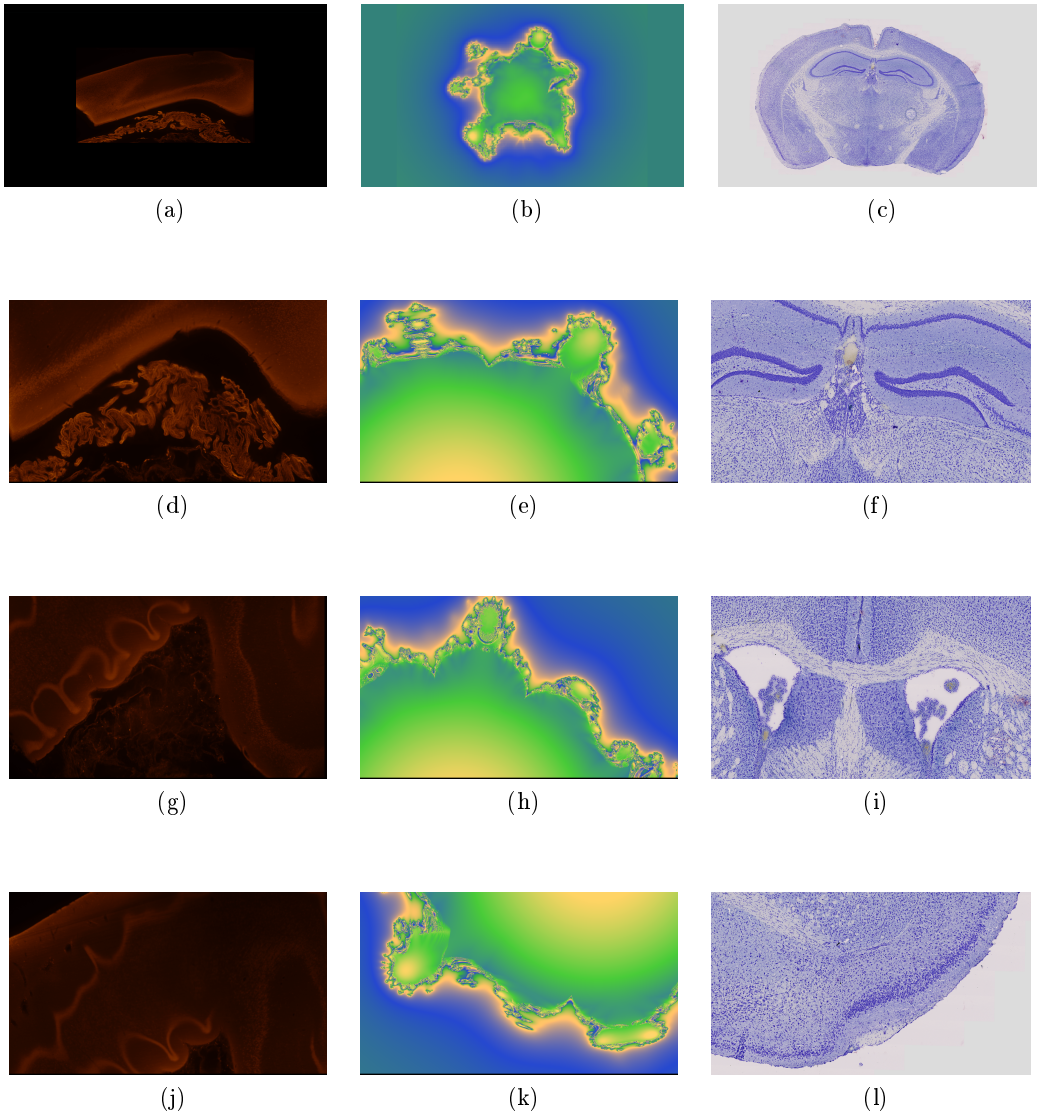


FIGURE 6.3 – **Rendu 3D autostéréoscopique.** Illustration du rendu de deux images multi-vues sur un volume de 123 GB. Le résultat du rendu est obtenu en 30 ms. (a) Vue d'une coupe haute résolution entière dans la pile d'image. (b) Les images de références recomposées avec toutes les briques nécessaires, utilisées pour construire l'image multi-vue. (c) Le rendu final pour un affichage sur un écran autostéréoscopique HD avec huit points de vue.

Perception 3D

Les valeurs de Δ_x et de Δ_z sont choisies de manière à optimiser la perception de profondeur. Les tests réalisés mettent en évidence le fait que la distance physique entre deux coupes dans le volume, doit être considérée dans le choix de la valeur de Δ_z. Si le volume à visualiser contient des coupes dont la distance physique inter-coupes est très supérieure à la distance entre les pixels d'une même coupe, la visualisation ne sera pas plaisante, peu importe la valeur du paramètre Δ_z choisie. C'est le cas en particulier du volume histologique *Tera Mouse brain* décrit dans la section 6.2. Celui-ci contient des coupes avec une précision très fine dans le plan, mais une distance beaucoup plus importante entre deux coupes. En conséquence, les structures biologiques que l'on retrouve sur une coupe ne sont pas présentes sur la coupe suivante. Si, à l'inverse, le ratio entre la précision dans le plan et la distance inter-coupe est plus proche de 1, la visualisation offre une perception qualitative de la profondeur qui est intéressante. Cependant, le choix du paramètre Δ_x est important pour régler cet effet de profondeur. En effet, le fait de diminuer la valeur de Δ_x diminue en conséquence la perception de profondeur. En revanche, un Δ_x trop grand provoque un inconfort visuel en plus d'un effet de "flou" non désiré. La figure 6.3 illustre le résultat du rendu de deux images, avec des valeurs des paramètres $[\Delta_x, \Delta_z] = [4, 1]$ pour notre écran HD comprenant huit point de vues.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

FIGURE 6.4 – **Illustration du rendu avec le microscope virtuel 2D.**

6.4.2 Lancer de rayon volumique multi-GPUs

Nous allons maintenant présenter une évaluation de notre application de lancer de rayon volumique out-of-core, multi-GPUs. Pour cela, nous utilisons un serveur de rendu Nvidia Quadro VCA. Celui-ci est composé d'un seul noeud de calcul dont l'architecture est représentée sur la figure 6.5. Il contient huit GPUs NVIDIA Quadro P6000 avec 24 GB de VRAM, deux CPU Intel Xeon E5-2698 2.2 GHz et 256 GB de RAM. L'affichage est fait sur un viewport HD 1920×1080. L'évaluation proposée se base sur des mesures de la fréquence d'affichage afin de montrer la réponse du rendu, ainsi que sur l'étude de la réactivité du chargement des données dans notre contexte de streaming à la demande. Elle est réalisée à partir des trois jeux de données présentés à la section 6.2 avec des briques de 64^3 voxels. De plus, les briques du volume *Tera Mouse brain* sont, stockées sur disque, mises en cache CPU et transférées vers le GPU avec des briques de 256^3 voxels, qui regroupe 4^3 briques de 64^3.

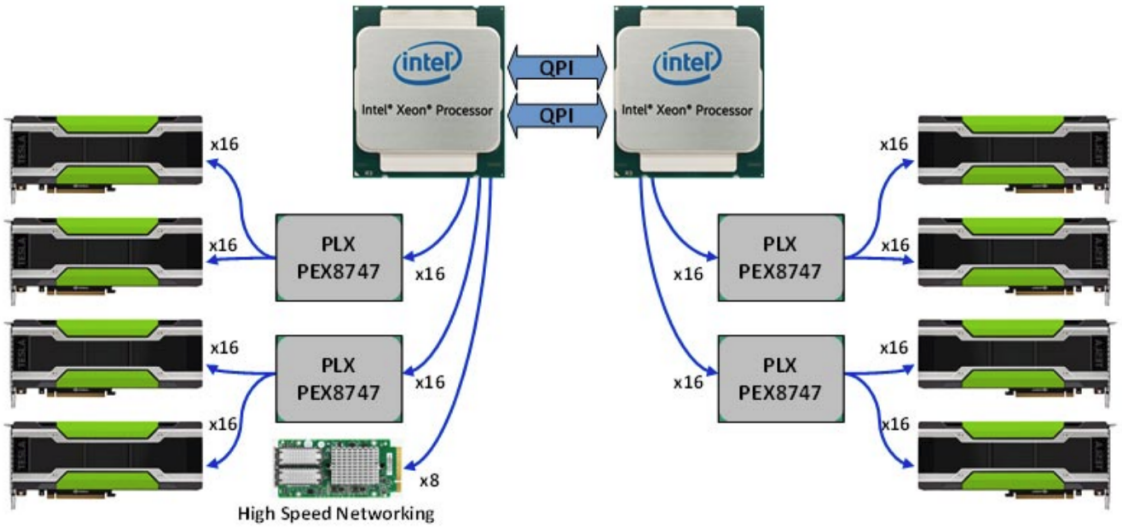
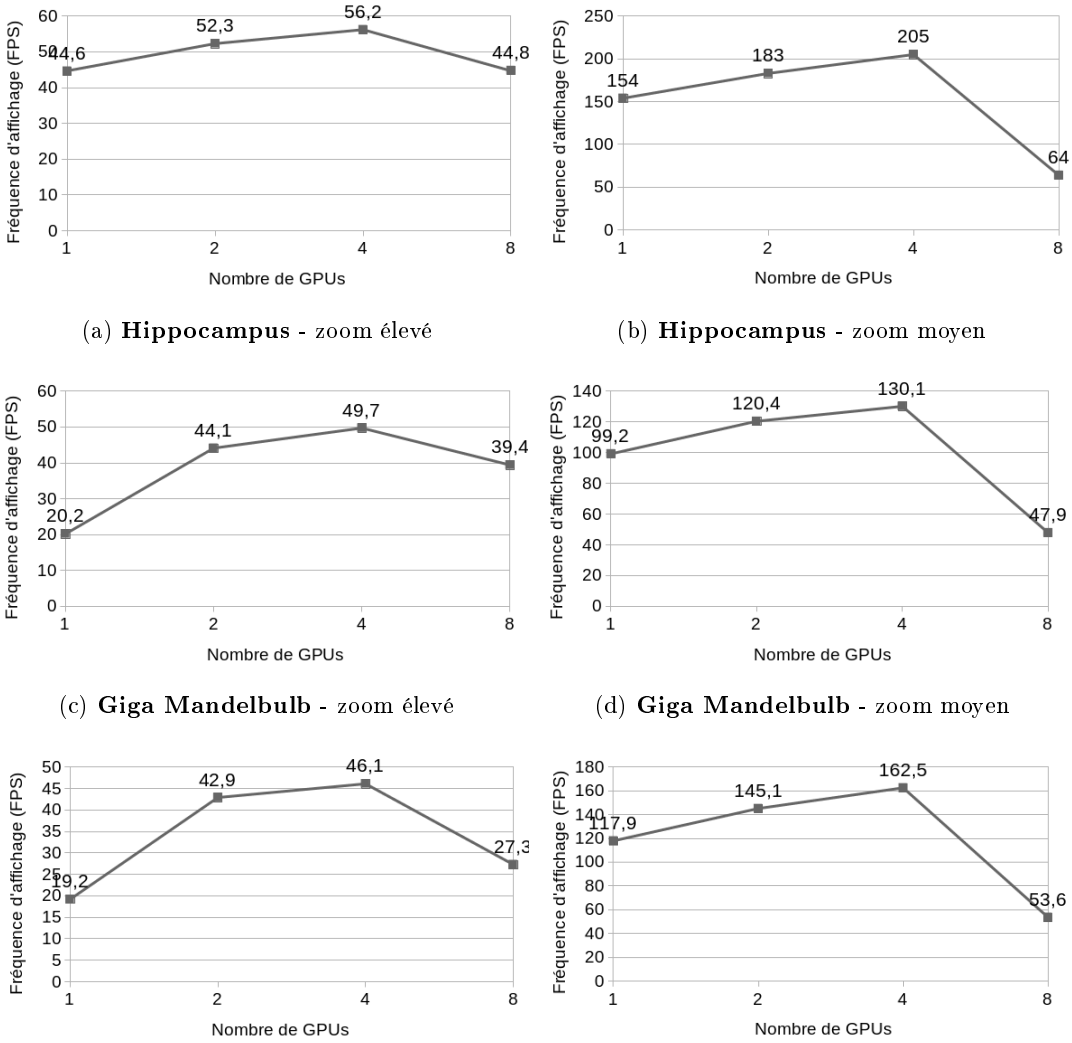


FIGURE 6.5 – **Architecture du serveur de rendu Nvidia Quadro VCA.**

6.4.2.1 Temps de rendu

La figure 6.6 montre les résultats obtenus par l'évaluation de notre solution de lancer de rayon volumique out-of-core multi-GPUs. Ceux-ci représentent les FPS moyens de deux scénarios de rotations complètes du volume selon le nombre de GPUs utilisés pour effectuer le rendu. Les deux scénarios permettent d'évaluer la performance du rendu selon le niveau de zoom appliqué sur le volume et donc, selon le niveau de résolution (LOD) sélectionné. Ces résultats sont, entre autre, dépendants de la fonction de transfert définie. Nous utilisons ici une fonction de transfert qui permet d'obtenir une visualisation intéressante selon le jeu de données.

La première chose que l'on peut voir avec ces résultats, c'est que la tendance des courbes est la même pour les trois jeux de données, et que les différentes valeurs des FPS sont du même ordre de grandeur pour les trois jeux de données. Cela nous permet de montrer que les performances du rendu ne sont pas impactées par les dimensions originales du volume visualisé. Les performances sont directement dépendantes de la taille de la sortie (de la zone d'affichage) et non de la taille de l'entrée (les dimensions du volume). Pour rappel, les données brutes sont de 11.8 GB pour le volume *Hippocampus*, 329.7 GB pour le volume *Giga Mandelbulb* et 1.45 TB pour le volume *Tera Mouse brain*. Deuxièmement, nous pouvons voir que les FPS donnés pour un rendu avec un seul GPU permettent déjà d'être dans un contexte de rendu interactif, même pour une visualisation des données en plein écran, à un niveau de résolution élevé. L'évolution de la fréquence d'affichage selon le nombre de GPUs utilisés permet de renforcer cet aspect de visualisation interactive. Nous remarquons cependant une coupure de cette hausse dans le cas de l'utilisation de 8 GPUs. Ce phénomène peut être expliqué par le fait que, dans ce contexte, 4 des GPUs utilisés sont reliés à un CPU différent de celui auquel est relié le GPU qui est en charge de recomposer les images à partir des sous-images locales de chaque GPU. Dans ce scénario, il n'est pas possible de profiter des communications directes de GPU à GPU avec le système "peer-to-peer" et UVA de CUDA pour les 4 GPUs en question. Cela est dû au fait que les deux CPUs soit reliés par un port QPI. Une copie explicite de chacun de ces 4 GPUs est alors initiée jusqu'au CPU auquel il sont connectés. Une autre copie de ces 4 buffers



(a) **Hippocampus** - zoom élevé

(b) **Hippocampus** - zoom moyen

(c) **Giga Mandelbulb** - zoom élevé

(d) **Giga Mandelbulb** - zoom moyen

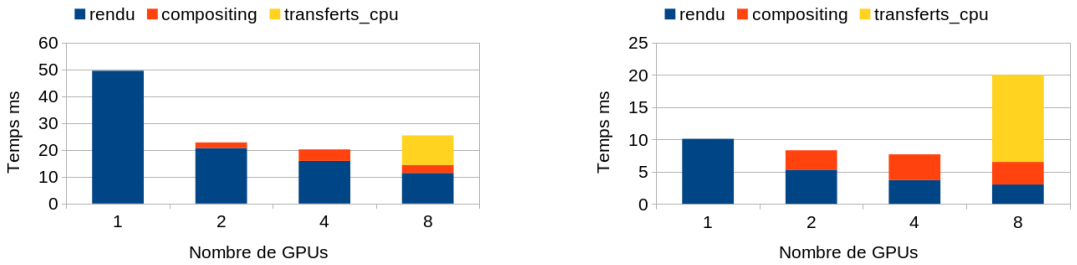
(e) **Tera Mouse brain** - zoom élevé

(f) **Tera Mouse brain** - zoom moyen

FIGURE 6.6 – **Mesures de la fréquence d'affichage du lancer de rayon volumique out-of-core multi-GPUs.** Ces mesures représentent les FPS moyens, obtenus à partir d'un scénario de plusieurs rotations complètes du volume, à deux niveaux différents de zoom, selon le nombre de GPUs et pour chacun des trois jeux de données présentés à la section 6.2. Les résultats de la colonne gauche (a, c, e) ont été calculés pour un zoom élevé avec un affichage du volume en plein écran à un niveau de détail élevé (voir illustrations (j, k, l) de la figure 6.9). Ceux de la colonne droite (b, d, f) ont été calculés pour un zoom moyen à un niveau de détail intermédiaire (voir illustrations (a, b, c) de la figure 6.9).

est à nouveau réalisée, vers le GPU "maître". Ces copies viennent ralentir le processus global de rendu. Afin de valider cette analyse, la figure 6.7, donne le détail des différentes étapes nécessaires au rendu. Les résultats obtenus pour cette figure sont issus du scénario présenté sur la figure 6.6(c,d). On peut alors voir l'impact de ces transferts sur le temps global du rendu. Ce comportement pourrait être corrigé avec l'utilisation de noeuds de calcul proposant des communications directes entre tous les GPUs, avec la technologie NVLink [11] de NVIDIA par exemple.

11. https://www.nvidia.com/fr-fr/data-center/nvlink/



(a) **Giga Mandelbulb** - zoom élevé

(b) **Giga Mandelbulb** - zoom moyen

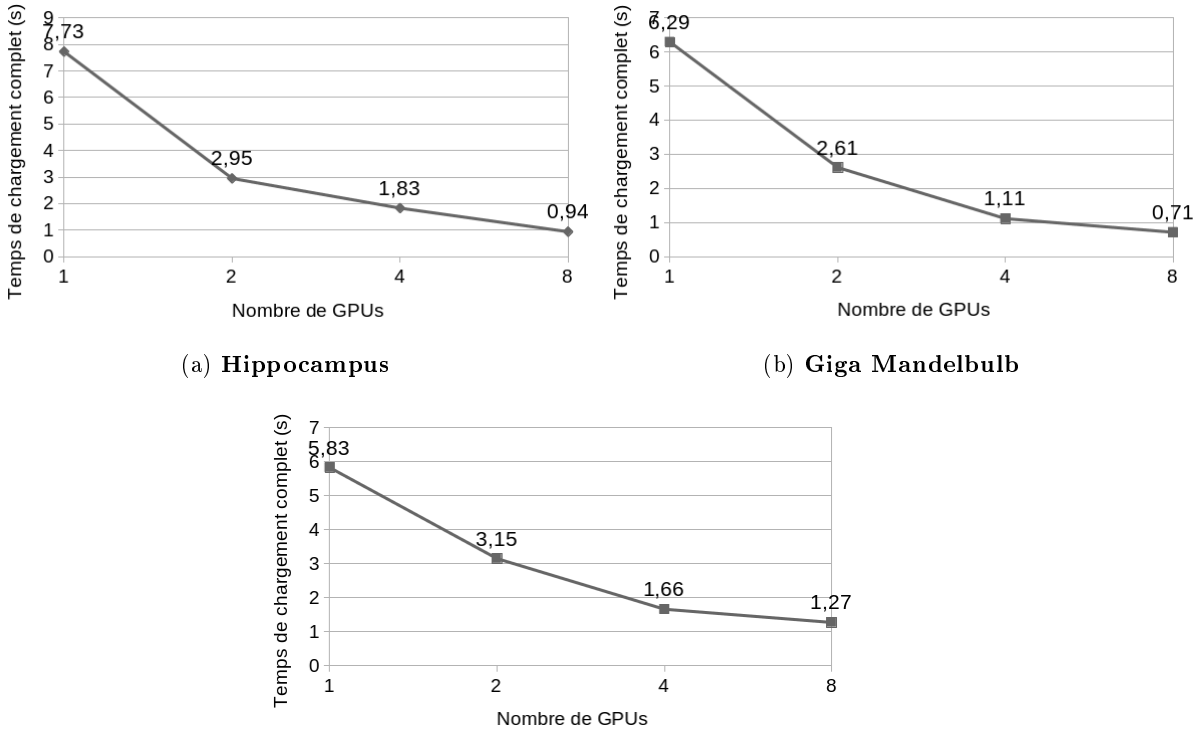
FIGURE 6.7 – **Somme des étapes de rendu du lancer de rayon volumique out-of-core multi-GPUs.** Ces mesures représentent le temps nécessaire aux différentes étapes du rendu, obtenus à partir d'un scénario de plusieurs rotations complètes du volume, à deux niveaux différents de zoom, selon le nombre de GPUs et pour le jeu de données *Giga Mandelbulb*. (a) Pour un zoom élevé avec un affichage du volume en plein écran à un niveau de détail élevé (voir illustration (k) de la figure 6.9).(b) Pour un zoom moyen à un niveau de détail intermédiaire (voir illustration (b) de la figure 6.9).

6.4.2.2 Chargement des briques

La figure 6.8 montre les résultats obtenus par l'évaluation du temps de chargement des données dans notre contexte de gestion out-of-core et de streaming à la demande. Cette évaluation est faite pour le chargement complet d'une vue, pour un scénario au pire cas, à partir de caches GPUs et CPUs vides. Les données sont donc uniquement initialement présentes sur le disque du serveur et stockées avec compression.

La tendance de ces courbes montre que le temps de chargement des données est réduit par le nombre de GPUs utilisés pour le rendu. En effet, notre solution de rendu distribué "sort-last", permet de répartir la charge de données sur les différents GPUs. Les connexions entre les CPUs et les différents GPUs de l'architecture du serveur utilisé (voir figure 6.5), permettent de paralléliser le temps de chargement des données. Les valeurs de ces temps de chargement sont de l'ordre de quelques secondes avec un seul GPU et de l'ordre d'une seconde ou moins avec l'ensemble des 8 GPUs du noeud. De plus ces valeurs sont sensiblement les mêmes pour les trois jeux de données.

Ces résultats nous permettent de justifier l'utilisation de méthodes multi-GPUs dans un contexte de calcul haute peformance. Cela permet de réduire le goulot d'étranglement du chargement des données à la demande, depuis un espace de stockage massif, jusqu'au GPU. Cette approche est donc tout à fait adaptée à de telles solutions de rendu out-of-core de très grands volumes de données.

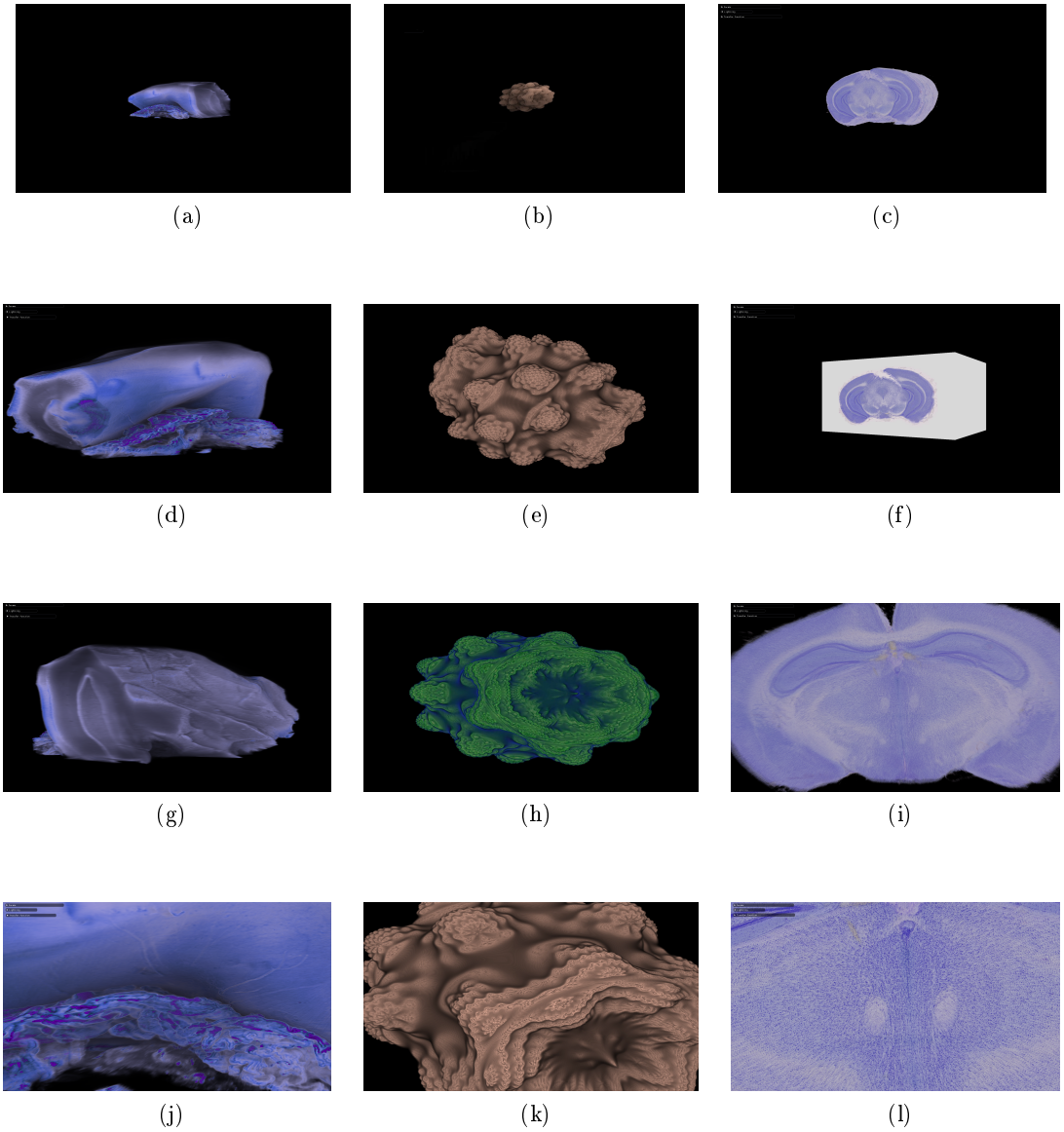


(a) **Hippocampus**

(b) **Giga Mandelbulb**

(c) **Tera Mouse brain**

FIGURE 6.8 – **Temps de chargement d'une vue complète au pire cas.** Ces mesures représentent les temps de chargement nécessaires à la génération d'une vue complète selon le nombre de GPUs utilisés. Les trois graphiques représentent respectivement les trois jeux de données présentés à la section 6.2. Ils sont obtenus pour le chargement d'une vue complète à partir de caches GPUs et CPUs vides.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

FIGURE 6.9 – **Illustration du rendu avec le lancer de rayon volumique.** Ces illustrations sont obtenues avec notre solution de rendu volumique par lancer de rayon multi-GPUs et représentent, à différents niveau de résolution, (a, d, j) le jeu de données *Hippocampus*, (b, e, k) le jeu de données *Giga Mandelbulb* et (c, f, l) le jeu de données *Tera Mouse brain*.

6.5 Discussions

6.5.1 Extension aux traitements

Les travaux de cette thèse se sont concentrés sur des solutions de visualisation tout en prenant en compte l'objectif de pouvoir utiliser le modèle de gestion-out-of-core proposé, dans un cadre plus général. Nous discutons dans cette section, des possibilités d'utilisation de notre approche pour une application qui offre la possibilité d'effectuer des traitements à la demande sur un volume de grande taille, pendant une phase de visualisation interactive de celui-ci. Nous discutons également des différents aspects qu'impliquerait une telle configuration dans un contexte de gestion out-of-core. La plupart des approches modernes dans la littérature, se sont concentrées sur l'élaboration de méthodes dirigées par la visualisation (dites "visualisation-driven"). Cependant cet aspect s'accompagne souvent d'une gestion des requêtes de données fortement orientée dans ce sens et empechant ainsi l'application sur GPU, qui utilise ce modèle, de pouvoir réclamer des briques qui ne sont pas visibles.

L'approche proposée dans cette thèse, à l'inverse, introduit la possibilité de réaliser des traitements à la demande pendant la phase de visualisation interactive, sur des données qui sont visibles ou non à l'écran. Un algorithme de traitement, implémenté sur GPU, peut être initié par l'action de l'utilisateur pendant la visualisation interactive. Cet algorithme peut alors être exécuté dans un stream CUDA différent de celui sur lequel s'exécute l'algorithme de rendu pour la visualisation interactive. De cette manière, il est possible de faire cohabiter ces deux algorithmes en parallèle indépendamment l'un de l'autre. Les deux algorithmes (de visualisation et de traitements), peuvent effectuer des requêtes de briques en même temps, en utilisant le *buffer de requêtes* partagé, sans problème de concurrence d'accès. Notre gestionnaire de cache GPU est ensuite capable de gérer la requête de la même manière, que celle-ci provienne de l'algorithme de visualisation ou de traitement. Ceci est rendu possible par notre approche car elle ne lie en rien la visibilité d'une donnée à l'écran, à sa possibilité d'être reclamée par l'application sur le GPU. L'application de visualisation interactive peut très bien respecter une approche dirigée par la visibilité des données (comme une méthode "ray-guided" pour un lancer de rayon volumique par exemple), tout en utilisant notre méthode de gestion out-of-core, capable d'initier des requêtes pour n'importe quelle partie du volume multi-résolution, que celle-ci soit visible ou non du point de vue de l'application de visualisation.

Afin de réaliser de simples tests, nous avons implémenté un algorithme de coloriage par calcul de composantes connexes 3D. Celui-ci peut être exécuté pendant le rendu, à la demande de l'utilisateur, à partir de la valeur d'un voxel qu'il a sélectionné. La figure 6.10 montre le résultat du rendu de notre application de microscope virtuel en 2D, comprenant l'affichage de certaines zones classifiées par l'algorithme de traitement. Cette application valide, avec un cas simple, le fait de pouvoir utiliser notre modèle out-of-core dans un contexte de traitement de données pendant la visualisation. Les chargements de briques initiées par des requêtes sur des briques qui ne font pas partie de l'espace de travail de l'algorithme de visualisation sont possibles et ne gène pas ce dernier.

L'étape de traitement des données à la demande peut avoir un impact sur ces données, à différents niveaux. Ces traitements peuvent fournir une information sur tout ou partie du volume (la taille ou le nombre d'occurence d'un élement par exemple), sans modifier les données elles-mêmes. Ils peuvent modifier indirectement un voxel, en fournissant une

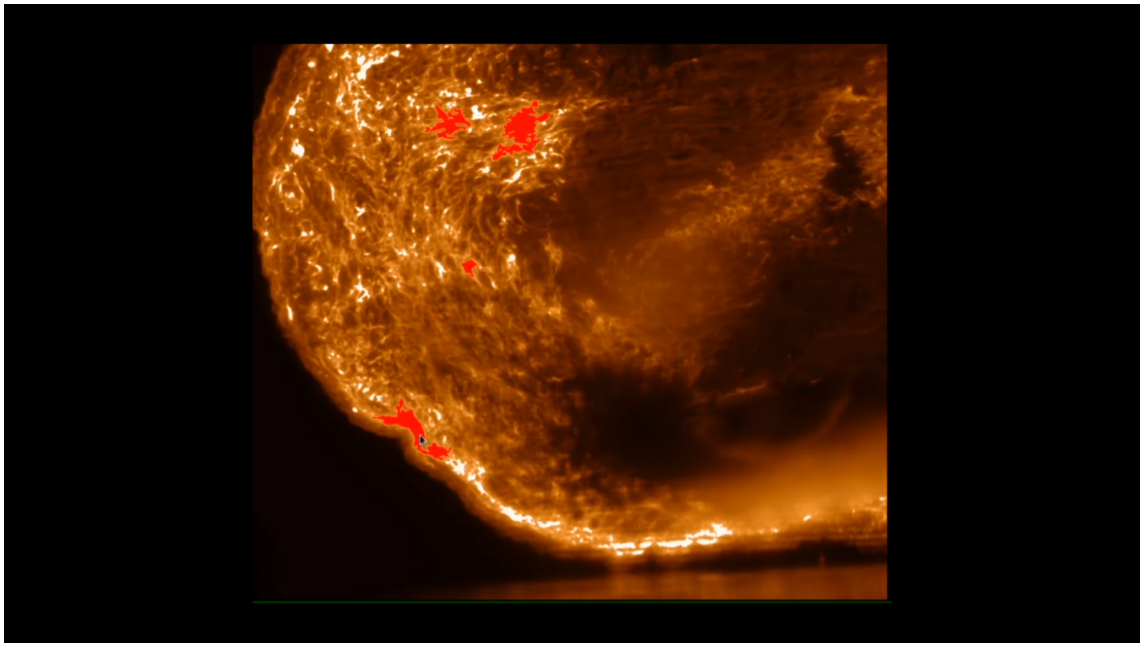
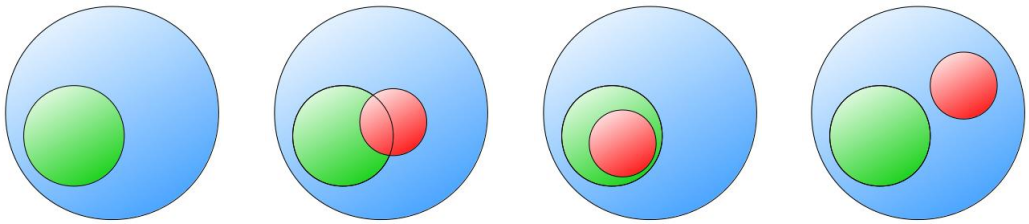


FIGURE 6.10 – **Traitements à la demande pendant l'affichage interactif du rendu du microscope virtuel 2D.** Des zones 3D fermées sont calculées dans un kernel CUDA dédié, à la demande de l'utilisateur, puis affiché en rouge par l'algorithme de visualisation.

information sur celui-ci (si un voxel est présent ou non dans un ensemble, pour de la classification par exemple), qui sera ensuite interprétée par l'algorithme de visualisation pour traiter ce voxel différemment (l'afficher d'une certaine couleur par exemple). Et enfin, l'algorithme de traitement peut modifier la valeur d'un voxel (application d'une convolution par exemple). Cette modification peut être permanente, auquel cas, la nouvelle valeur du voxel est écrite dans le cache et est communiquée au CPU pour propager ce changement jusqu'à son écriture sur le disque, venant ainsi remplacer l'ancien voxel. Il est également possible de ne pas gérer cette contrainte, auquel cas, le traitement doit être appliqué à nouveau, à chaque fois que la brique est retirée puis remise en cache par la suite.

Espace de travail

La figure 6.11 montre les différentes configurations possible de l'espace de travail requis par la ou les applications de traitement ou visualisation. Une approche qui ne permet qu'une seule application de visualisation ou de traitement, ne cohabitant pas avec une autre application, gérera des espaces de travail ayant une configuration comme celle représentée sur la figure 6.11.(a). Dans le contexte de traitement à la demande pendant la visualisation interactive, l'espace de travail requis peut être défini par l'une des autres configurations illustrées sur la figure 6.11.(b, c, d). Un algorithme de traitement peut opérer uniquement sur les données qui font partie de l'espace de travail de l'application de visualisation (figure 6.11.(c)). Dans ce cas, les données impliquées dans l'algorithme de traitement sont déjà présentes dans le cache GPU (ou déjà requêtées par l'étape de visualisation). A l'inverse, si l'algorithme de traitement traite des données qui ne font pas partie de l'espace de travail spécifique à la visualisation (6.11.(b, d)), cela peut entraîner des nouvelles requêtes de briques. Cela peut augmenter le risque d'avoir un espace de travail trop grand pour être stocké en cache. En pratique, la dernière configuration serait sûrement la moins fréquente. Cependant, il serait possible de trouver ce type de configuration dans un environnement



(a) (b) (c) (d)

FIGURE 6.11 – **Configurations de l'espace de travail.** L'ensemble bleu représente le volume multi-résolution entier. Les ensembles vert et rouge représentent l'espace de travail requis par l'algorithme de visualisation et/ou de traitement. Ces différentes configurations sont organisées selon (a) une seule application ou, (b, c, d) la cohabitation d'une application de traitement à la demande et de visualisation interactive.

distribué avec un sous-ensemble des noeuds de calcul dédié à la visualisation et un autre sous-ensemble en charge du traitement d'une partie du volume qui n'est pas affichée à ce moment.

6.5.2 Communications GPU-CPU

La méthode de gestion out-of-core proposée dans cette thèse est conçue sur un système permettant de réduire les communications entre le CPU et le GPU à leur strict minimum. En comparaison avec l'utilisation d'un octree, comme déjà évoqué, notre système évite de devoir transférer les noeuds d'un arbre (utilisés comme pages, pour l'adressage des données) sur le GPU à la demande. Hadwiger *et al.* [HBJP12] utilisent le même type de structure d'adressage que celle présentée dans cette thèse, mais ils proposent une gestion de leur structure sur le CPU. Cette approche engendre une quantité plus importante de données à transférer entre le GPU et le système central. En effet, contrairement à notre approche, il est nécessaire dans leur contexte, de communiquer au CPU les informations de l'utilisation des briques dans le cache GPU. De plus, leur système de gestion des requêtes de données depuis le GPU peut être à l'origine d'une grande quantité de données à transférer au CPU. Ils proposent de lister les défauts de cache sur GPU en les regroupant dans l'espace d'affichage de la visualisation. Pour cela, ils stockent sur GPU, une table de hachage par tuiles 2D dans l'espace écran. Pour une taille de tuile de 64×64 pixels, avec un affichage HD, cela correspond à 510 tables de hachage avec des opérations atomiques pour rapporter les défauts de cache au CPU. Il est possible de quantifier la quantité de données à transférer avec cette approche. Pour cela, il est important de noter qu'ils utilisent un identifiant sur 32 bits ou sur 64 bits pour chaque brique selon le nombre de briques présentes dans le volume multi-résolution. De plus, dans leur contexte de lancer de rayon volumique, ils limitent la quantité de requêtes à un petit nombre M par rayon, de manière à distribuer les requêtes de données sur plusieurs images. Cela représente des messages du GPU vers le CPU de $1920 \times 1080 \times 4 \times 4 \approx 33\,\text{MB}$ (au pire cas) pour un viewport HD à chaque passe de rendu. Cette quantité peut atteindre $3840 \times 2160 \times 4 \times 4 \approx 133\,\text{MB}$ pour un viewport 4K. Dans un contexte de rendu multi-vue, ces valeurs doivent être multipliées par N à chaque passe de rendu, pour un affichage sur un écran multiscopique avec N vues. Nous pouvons noter cependant que les travaux de Fogal *et al.* [FSK13] utilisent le même type de gestion des requêtes de briques, mais en proposant une version compressée des tables de hachage.

6.5.3 Quelques limitations

Mise à jour de la table de pagination multi-niveaux

Il y a deux limitations principales liées à la mise à jour de la table de pagination hiérarchique. La première peut se produire lorsqu'une entrée est retirée d'un niveau de *cache PT*. Lorsqu'un nouveau *bloc PT* vient en remplacer un déjà présent en cache, le lien des briques que ce dernier référençait dans le cache de brique est perdu. En conséquence, certaines données peuvent rester présentes en cache mais ne plus être référencées. Cependant, ces blocs/briques vont être rapidement relégués à la fin de la LRU associée à ce cache, et ainsi être rapidement remplacés dans les itérations futures. La seconde limitation est liée à la manière de référencer, dans un *cache PT*, les briques qui viennent d'êtres ajoutées au cache de brique. Bien que l'on propose une implémentation de cette étape sur le GPU, celle-ci est réalisée sequentiellement pour chaque brique. Par exemple, supposons que nous voulons ajouter en même temps, deux briques spatialement proches, toutes les deux référencées par le même *bloc PT*. Un système parallèle aurait ajouté deux *blocs PT* en cache, qui auraient tous deux référencé une seule brique alors qu'un seul bloc était nécessaire pour référencer ces deux nouvelles briques. Pour éviter un tel scénario, les briques doivent être référencées l'une après l'autre.

Perte de pages virtuelles

Le système de mise en cache de certaines parties de la table de pagination, dans l'approche hiérarchique de virtualisation de la mémoire proposée, ne garantit pas la pérénité des méta-informations des pages concernées. Si l'on utilise plus d'un niveau de virtualisation (c'est à dire, des niveaux intérmédiaires, constitués de *caches PT*), il y a un problème de préservation de l'information dans les entrées de ces niveaux. Cela vient du fait que les pages sont créées à la volée sur le GPU, quand nécessaire, puis mises en cache. Quand une page est retirée d'un cache, toute les informations qu'elle contenait, sont perdues. Typiquement, le statut d'une page qui indique l'état vide de la/les brique(s) qu'elle addresse, ne sera pas préservé. A l'inverse, ce problème n'apparaît pas avec l'utilisation d'une structure d'arbre (comme un octree par exemple) car ce sont les noeuds de l'arbre eux même, qui sont utilisés comme page. Généralement, ceux-ci sont stockés de manière permanente sur le CPU, puis transférés au GPU à la demande. Lorsqu'il sont supprimés du GPU, les méta-informations qu'ils contenaient sont toujours présentes sur le CPU.

Applications de traitement de données

Nous avons discuté, dans la section précédente (voir section 6.5.1), des capacités de notre approche à être utilisée dans un système applicatif général, incluant des applications de visualisation et/ou de traitement de données. Si l'on utilise notre approche de gestion de données out-of-core, avec une application de traitement de données, sans cohabitation avec un système de visualisation interactive de ces même données, il est possible d'implémenter n'importe quel type de traitement. En revanche, un algorithme de traitement à la demande, initié pendant une phase de visualisation interactive, doit être restreint à certaines contraintes. En effet, un algorithme qui manipulerait l'ensemble d'un grand volume géré de manière out-of-core, génèrerait beaucoup d'activité de cache, qui pourrait très fortement impacter la possibilité de maintenir l'espace de travail nécessaire à la visualisation interactive. Ansi, il serait plus adapté d'utiliser des algorithme de traitement de données limités à un espace de travail restreint, correspondant à un ensemble de traitements locaux.

6.6 Conclusion

L'évaluation de notre modèle de gestion out-of-core, en plus de l'analyse de performance des différentes applications implémentées, nous permettent de montrer que :

1. la méthode proposée permet d'obtenir de bonnes performances en terme de fréquence d'affichage du rendu pour la visualisation, ainsi que de bonnes réponses de chargement des données.
2. la méthode de gestion out-of-core proposée offre une empreinte mémoire GPU faible, un taux d'occupation du GPU faible ainsi que des communications très limitées entre le GPU et la mémoire centrale.
3. la méthode proposée est générique, permet d'avoir un cadre applicatif large et est adaptée aux environnements de calcul haute performance multi-GPUs.

Ces différents aspects montrent que les solutions étudiées dans les travaux de cette thèse sont efficaces pour visualiser en temps interactif, des données dépassant très largement la quantité de mémoire disponible sur GPU ou sur CPU. De plus, contrairement aux différentes approches que l'on peut trouver dans la littérature, nous proposons un cadre applicatif large, permettant de visualiser les données de différentes manières et de proposer des traitements à la demande sur ces données pendant la visualisation.

Chapitre 7

Conclusion

Synthèse des contributions

Dans le cadre de cette thèse, nous nous sommes intéressé à une problématique de gestion de données volumique massives, en temps interactif, pour différents types d'application de visualisation et possiblement de traitement. Ces objectifs sont en particulier portés par l'intéret d'analyser des données biomédicales de grande dimension dans un cadre applicatif lié à l'étude de maladies neurodégénératives.

Nous avons conçu un pipeline complet intègrant toutes les étapes nécessaires aux applications de visualisation sur GPU, pour manipuler n'importe quelle partie d'un très grand volume de données dépassant la capacité mémoire GPU et CPU. Celui-ci comprend un modèle de gestion out-of-core composé d'un système de caches avec une gestion d'adressage virtuel des données depuis le GPU. La structure utilisée est une hiérarchie de tables de pagination multi-résolution. Cette solution d'adressage virtuel introduite en 2012 par Markus Hadwiger et al. [HBJP12] fait l'objet d'une extension dans les travaux de cette thèse. En particulier, nous avons proposé un formalisme mathématique complet de cette méthode, ainsi qu'une étude en profondeur de ces différents aspects de virtualisation de la mémoire. De plus, nous avons proposé une version étendue de la gestion de cette structure, afin d'obtenir une approche out-of-core avec les caratéristiques suivantes :

1. Une structure d'adressage virtuel entièrement gérée sur GPU : notre structure est uniquement et entièrement présente sur GPU, en plus de l'ensemble des ressources nécessaires au maintien des caches qui la composent (mise à jour des données présentes en cache et requêtes de données à la demande de l'application). Nous proposons aussi une gestion de la mise à jour de la hiérarchie de la table de pagination multi-résolution, également opérée sur GPU, tout en laissant un maximum des ressources GPU pour l'application hôte.
2. Des communications entre le GPU et la mémoire centrale réduites à leur strict minimum : la communication de méta-informations est revue à la baisse de manière à limiter la quantité de données qui transitent entre le GPU et le CPU et laisser ainsi le maximum de bande passante aux données elles mêmes, lorsqu'elles doivent être transférées au GPU.
3. Une approche adaptée à un contexte applicatif large en proposant une conception générique capable de fontionner pour différents types d'applications de visualisation et/ou de traitement de données volumiques de grande dimension.
4. Une approche adaptée à l'utilisation d'environnement de calcul haute performance avec un système de distribution de notre modèle sur des noeuds de calcul hybrides, multi-GPUs, multi-CPUs.

Dans ce contexte de manipulation interactive de données volumiques massives, nous avons proposé deux applications de visualisation utilisant notre modèle de gestion out-of-core. Dans un premier temps, nous avons présenté une solution proposant de simuler le principe d'un microscope, permettant de visualiser des piles d'images ultra haute résolution sur des dispositifs 3D autostéréoscopiques. Celui-ci est capable de naviguer en temps interactif dans un volume de grande dimension afin de visualiser des coupes biologiques épaisses en offrant une perception de profondeur. Dans un second temps, nous avons implémenté une approche de rendu volumique direct par lancer de rayon multi-GPUs. Cette méthode est basée sur une implémentation de rendu multi-résolution out-of-core, dite "ray-guided". Elle intègre une distribution de notre modèle d'adressage virtuel dans un environnement

de calcul haute performance, sur des noeuds de calcul multi-CPUs, multi-GPUs. Ajouté à notre système de client-serveur capable de diffuser un flux vidéo jusqu'à un client léger, cette approche permet de déporter la charge de calcul sur des environnements plus adaptés et ainsi profiter de meilleures performances de rendu.

L'évaluation des différents aspects implémentés dans ces travaux de thèse, nous ont permis de montrer que *i)* le modèle de gestion out-of-core proposé est très bien adapté aux très grands volumes de données et *i)* il possède une empreinte mémoire GPU faible, malgrè la présence de l'ensemble des ressources nécessaires à son maintien sur GPU. Cela lui permet ainsi de garantir l'instanciation d'un cache de données large, assurant l'interactivité de la visualisation des données en limitant le goulot d'étranglement des transferts à la demande depuis le disque ou le CPU. Dans le cas des deux applications proposées, notre système garantit une bonne fréquence d'affichage du rendu et de bonnes réponses de chargement des données. Cela nous permet de justifier d'une visualisation interactive des données de grande dimension dans plusieurs cadres applicatifs différents. De plus, la tendance de ces deux métriques (FPS et temps de chargement) est renforcée dans un contexte de calcul haute performance, sur des plateformes multi-GPUs. La gestion complète de l'ensemble de la structure d'adressage virtuel multi-résolution sur GPU ne sollicite qu'un faible taux d'occupation de celui-ci, laissant ainsi une majeure partie de la puissance de calcul disponible pour de lourdes applications.

Perspectives

Les travaux que nous avons présentés dans cette thèse ouvrent plusieurs directions d'études futures que nous décrivons ci-après.

En premier lieu, et bien que le modèle de gestion out-of-core proposé dans nos travaux se positionne particulièrement sur des données volumiques, il serait intéressant d'envisager la possibilité d'étendre l'utilisation de cette approche pour des données structurées différemment. Il existe en effet des besoins de manipulation de données massives dans des domaines qui nécessitent des représentations graphiques autres que des grilles régulières de voxels. En particulier, certaines applications industrielles ont la nécessité de fournir des prototypages virtuels (pour l'industrie automobile par exemple) constitués d'objets représentés sous forme de maillages de très grande dimension afin de fournir une précision suffisante pour la validation de la qualité de produits. Ces objets représentés sous forme de maillage, peuvent atteindre des tailles dépassant la capacité mémoire disponible. L'utilisation de notre solution dans ce contexte nécessiterait de concevoir une méthode permettant de faire correspondre les caractéristiques de la représentation du modèle avec le caractère régulier de la grille manipulée par la structure de table de pagination multi-résolution. Il pourrait s'agir de définir un ensemble de brique à partir du maillage, en se basant sur un kd-tree ou sur le voisinage topologique de celui-ci par exemple. Ces briques pourraient contenir d'une part, les informations de géométrie (positions des sommets...) et d'autre part, les informations de topologie pour la connexion des sommets entre eux. La structure d'adressage et le cache de brique devraient cependant être capables de manipuler ces briques qui sont différentes des briques qui contiennent un nombre régulier et constant de voxels.

Dans un second temps, l'extension aux applications de traitement à la demande, pendant la visualisation interactive, sur des données volumique massives, a été abordée très succinctement dans cette thèse. Une première étude des problèmes pouvant être posés par cette combinaison algorithmique, a été abordée dans les sections 6.5.1 et 6.5.3. Des re-

cherches sur la cohabitation de telles applications peuvent êtres envisagées, de manière à analyser les possibilités d'interactivité (aussi bien en visualisation que en traitement), ainsi que le comportement du cache de données et du/des caches de table de pagination sur GPU, au cours de l'exécution.

Enfin, les résultats obtenus pour l'évaluation de notre solution de lancer de rayon volumique multi-GPUs présentent une limitation dans les performances de la distribution du rendu out-of-core, dans le cas d'un système de communications non directes entre les différents GPUs. Il serait intéressant d'étendre cette évaluation sur des architectures équipées de la technologie NVLink permettant ainsi de profiter de performances bien plus intéressantes en matière de communications. Il serait également intéressant de comparer les performances obtenues avec une extension de notre solution distribuée sur des architectures multi-noeuds.

Bibliographie

[AD10] S. Arens and G. Domik. A Survey of Transfer Functions Suitable for Volume Rendering. In *Proceedings of the 8th IEEE/EG International Conference on Volume Graphics*, VG'10, pages 77–83, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[ALB+13] K. Amunts, C. Lepage, L. Borgeat, H. Mohlberg, T. Dickscheid, M.-E. Rousseau, S. Bludau, P.-L. Bazin, L. B. Lewis, A.-M. Oros-Peusquens, N. J. Shah, T. Lippert, K. Zilles, and A. C. Evans. BigBrain: An Ultrahigh-Resolution 3d Human Brain Model. *Science*, 340(6139):1472–1475, June 2013.

[AM00] Ulf Assarsson and Tomas Moller. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*, 5(1):9–22, January 2000.

[BBJL05] A. Benassarou, E. Bittar, N. W. John, and L. Lucas. Mc slicing for volume rendering applications. In Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science – ICCS 2005*, pages 314–321, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BCH12] E. Wes Bethel, Hank Childs, and Charles Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press, October 2012.

[BD02] P. Bhanirantka and Y. Demange. OpenGL volumizer: a toolkit for high quality volume rendering of large data sets. In *Symposium on Volume Visualization and Graphics, 2002. Proceedings. IEEE / ACM SIGGRAPH*, pages 45–53, October 2002.

[BHAA+13] 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.

[BHJ+11] Johanna Beyer, Markus Hadwiger, Won-Ki Jeong, Hanspeter Pfister, and Jeff Lichtman. Demand-driven volume rendering of terascale EM data. page 1. ACM Press, 2011.

[BHMF08] Johanna Beyer, Markus Hadwiger, Torsten Möller, and Laura Fritz. Smooth Mixed-Resolution GPU Volume Rendering. page 8, 2008.

[BHP14] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. A Survey of GPU-Based Large-Scale Volume Visualization. IEEE Visualization and Graphics Technical Committee (IEEE VGTC), 2014.

[BHP15] 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.

[BHWB07] J. Beyer, M. Hadwiger, S. Wolfsberger, and K. Bühler. High-Quality Multimodal Volume Rendering for Preoperative Planning of Neurosurgical Interventions. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1696–1703, November 2007.

[BLD14] J. Baert, A. Lagae, and Ph. Dutré. Out-of-Core Construction of Sparse Voxel Octrees. *Computer Graphics Forum*, 33(6):220–227, 2014.

[Bli82] James F. Blinn. Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '82, pages 21–29, New York, NY, USA, 1982. ACM.

[BNS01] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, May 2001.

[BSS00] Dirk Bartz, Bengt-Olaf Schneider, and Claudio Silva. Rendering and Visualization in Parallel Environments. page 72, 2000.

[BWPP04] Jiri Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum*, 23(3):615–624, September 2004.

[CBC+03] U. Catalyurek, M. D. Beynon, Chialin Chang, T. Kurc, A. Sussman, and J. Saltz. The virtual microscope. *IEEE Transactions on Information Technology in Biomedicine*, 7(4):230–248, December 2003.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of the 1994 Symposium on Volume Visualization*, VVS '94, pages 91–98, New York, NY, USA, 1994. ACM.

[CIAR13] Germán Corredor, Marcela Iregui, Viviana Arias, and Eduardo Romero. Flexible Architecture for Streaming and Visualization of Large Virtual Microscopy Images. In *Medical Computer Vision. Large Data in Medical Imaging*, Lecture Notes in Computer Science, pages 34–43. Springer, Cham, September 2013.

[CM09] C. D. Correa and K. L. Ma. Visibility-driven transfer functions. In *2009 IEEE Pacific Visualization Symposium*, pages 177–184, April 2009.

[CM11] Carlos D. Correa and Kwan-Liu Ma. Visibility histograms and visibility-driven transfer functions. *Ieee Transactions on Visualization and Computer Graphics*, page 2011, 2011.

[CN93] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3d texture hardware. 1993.

[CNLE09] 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.

[Cra11] Cyril Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes.* phdthesis, Université de Grenoble, 2011.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume Rendering. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 65–74, New York, NY, USA, 1988. ACM.

[Dee] Deep zoom. `https://www.microsoft.com/silverlight/deep-zoom`. [Online ; accessed 20-September-2018].

[DF93] Daniel B. Diner and Derek H. Fender. *Human Engineering in Stereoscopic Viewing Devices*. Plenum Press, New York, NY, USA, 1993.

[DH92] John Danskin and Pat Hanrahan. Fast Algorithms for Volume Ray Tracing. In *Proceedings of the 1992 Workshop on Volume Visualization*, VVS '92, pages 91–98, New York, NY, USA, 1992. ACM.

[Dod02] Neil A. Dodgson. Analysis of the viewing zone of multiview autostereoscopic displays. In *Stereoscopic Displays and Virtual Reality Systems IX*, volume 4660, pages 254–266. International Society for Optics and Photonics, May 2002.

[Dod05] N. A. Dodgson. Autostereoscopic 3d Displays. *Computer*, 38(8):31–36, August 2005.

[EHMK+06] Klaus Engel, M. Hadwiger, Joe M. Kniss, Christof Rezk-salama, and Daniel Weiskopf. *Real-time Volume Graphics*. 2006.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, HWWS '01, ACM*, pages 9–16, 2001.

[Eng11] 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, October 2011.

[FCS+10] Thomas Fogal, Hank Childs, Siddharth Shankar, Jens Krüger, R. Daniel Bergeron, and Philip Hatcher. Large Data Visualization on Distributed Memory multi-GPU Clusters. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 57–66, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[FS05] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. page 15. ACM Press, 2005.

[FSK13] 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, 2013.

[GDHRR+16] Miguel Gonzalo-Domínguez, Cristina Hernández-Rodríguez, Pablo Ruisoto, Juan Antonio Juanes, José Martín Marín Balbin, and Alberto Prats-Galino. 3d Reconstructions of Brain Ventricles Using Anaglyph Images. In *Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality*, TEEM '16, pages 491–494, New York, NY, USA, 2016. ACM.

[GH06] Attila Gyulassy and Bernd Hamann. *Time- and Space-efficient Error Calculation for Multiresolution Direct Volume Rendering*. 2006.

[GMG08] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias 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.

[GPU05] GPU Gems, 2005.

[GS04] S. Guthe and W. Strasser. Advanced techniques for high-quality multi-resolution volume rendering. *Computers & Graphics*, 28(1):51–58, February 2004.

[GWGSs02] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Stra\s ser. Interactive rendering of large volume data sets. In *Visualization, 2002. VIS 2002. IEEE*, pages 53–60. IEEE, 2002.

[HBC10] M. Howison, E. W. Bethel, and H. Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'10, pages 1–10, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[HBC12] M. Howison, E.W. Bethel, and H. Childs. Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems. *IEEE Transactions on Visualization and Computer Graphics*, 18(1):17–29, January 2012.

[HBJP12] 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.

[HDFP11] N. S. Holliman, N. A. Dodgson, G. E. Favalora, and L. Pockett. Three-Dimensional Displays: A Review and Applications Analysis. *IEEE Transactions on Broadcasting*, 57(2):362–371, June 2011.

[HKSB06] Markus Hadwiger, Andrea Kratz, Christian Sigg, and Katja Bühler. *GPU-Accelerated Deep Shadow Maps for Direct Volume Rendering.* 2006.

[HLSR08] Markus Hadwiger, Patric Ljung, Christof Rezk Salama, and Timo Ropinski. Advanced illumination techniques for GPU volume raycasting. In *ACM Siggraph Asia 2008 Courses*, page 1. ACM, 2008.

[HLY07] Frida Hernell, Patric Ljung, and Anders Ynnerman. Efficient Ambient and Emissive Tissue Illumination using Local Occlusion in Multiresolution Volume Rendering. In *DIVA*, pages 1–8. IEEE, 2007.

[Hoe16] Rama Karl Hoetzlein. GVDB: Raytracing Sparse Voxel Database Structures on the GPU. In *Proceedings of High Performance Graphics*, HPG '16, pages 109–117. Eurographics Association, 2016.

[Hor13] Michael Hortsch. From Microscopes to Virtual Reality – How Our Teaching of Histology is Changing. *Journal of Cytology & Histology*, 04(03), 2013.

[HP07] M. Gross H. Pfister. Point-based graphics. *Morgan Kaufmann Publishers Inc.*, 2007.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. page 167. ACM Press, 2007.

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Khatja Bühler, and Markus Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24(3):303–312, September 2005.

[ILC10] Martin Isenburg, Peter Lindstrom, and Hank Childs. Parallel and Streaming Generation of Ghost Data for Structured Grids. *IEEE Computer Graphics and Applications*, 30(3):32–44, May 2010.

[JBH+09] 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, November 2009.

[JLHE01] Graham R. Jones, Delman Lee, Nicolas S. Holliman, and David Ezra. Controlling perceived depth in stereoscopic images. volume 4297, pages 42–53, 2001.

[Joh04] C. Johnson. Top scientific visualization research problems. *IEEE Computer Graphics and Applications*, 24(4):13–17, July 2004.

[JST+10] Won-Ki Jeong, Jens Schneider, Stephen G. Turney, Beverly E. Faulkner-Jones, Dominik Meyer, Rüdiger Westermann, R. Clay Reid, Jeff Lichtman, and Hanspeter Pfister. Interactive histology of large-scale biomedical image stacks. *IEEE Transactions on Visualization and Computer Graphics*, 16:1386–1395, 2010.

[KD98] Gordon Kindlmann and James W. Durkin. Semi-automatic Generation of Transfer Functions for Direct Volume Rendering. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, pages 79–86, New York, NY, USA, 1998. ACM.

[Khra] Group khronos. opengl official website. `http://www.opengl.org`. [Online ; accessed 2018-June-29].

[Khrb] Opencl reference documentation. `https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/`. [Online ; accessed 2018-June-29].

[KKF+17] B. Kozlíková, M. Krone, M. Falk, N. Lindow, M. Baaden, D. Baum, I. Viola, J. Parulek, and H.-C. Hege. Visualization of Biomolecular Structures: State of the Art Revisited: Visualization of Biomolecular Structures. *Computer Graphics Forum*, 36(8):178–204, December 2017.

[KKH02] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, July 2002.

[Kno06] Aaron Knoll. *A Survey of Octree Volume Rendering Methods*. 2006.

[KSA13] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High Resolution Sparse Voxel DAGs. *ACM Trans. Graph.*, 32(4):101:1–101:13, 2013.

[KSSE05] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *VIS 05. IEEE Visualization, 2005.*, pages 223–230, October 2005.

[KUDC07] Johannes Kopf, Matt Uyttendaele, Oliver Deussen, and Michael F. Cohen. Capturing and viewing gigapixel images. *ACM Transactions on Graphics (TOG)*, 26(3):93, 2007.

[KVH84] James T. Kajiya and Brian P Von Herzen. Ray Tracing Volume Densities. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 165–174, New York, NY, USA, 1984. ACM.

[KW03] 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.

[LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3d Surface Construction Algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[Lev88] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[Lev90] Marc Levoy. Efficient Ray Tracing of Volume Data. *ACM Trans. Graph.*, 9(3):245–261, July 1990.

[LHJ00] Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. volume 3960, pages 365–374, 2000.

[LHJ03] Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Efficient Error Calculation for Multiresolution Texture-based Volume Visualization. In Gerald Farin, Hans-Christian Hege, David Hoffman, Christopher R. Johnson, Konrad Polthier, Gerald Farin, Bernd Hamann, and Hans Hagen, editors, *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 51–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[LK11] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1048–1059, 2011.

[LKG+16] Patric Ljung, Jens Krüger, Eduard Groller, Markus Hadwiger, Charles D. Hansen, and Anders Ynnerman. State of the Art in Transfer Functions for Direct Volume Rendering. *Computer Graphics Forum*, 35(3):669–691, June 2016.

[LLR13] Laurent Lucas, Céline Loscos, and Yannick Rémion. *3D Video: From Capture to Diffusion*. John Wiley & Sons, December 2013. Google-Books-ID: maJMAgAAQBAJ.

[LLY06] Claes Lundström, Patric Ljung, and Anders Ynnerman. Multi-Dimensional Transfer Function Design Using Sorted Histograms. *DIVA*, pages 1–8, 2006.

[LMK03] Wei Li, Klaus Mueller, and Arie Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *Visualization, 2003. VIS 2003. IEEE*, pages 317–324. IEEE, 2003.

[MAWM11] B. Moloney, M. Ament, D. Weiskopf, and T. Moller. Sort-First Parallel Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1164–1177, August 2011.

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

[MBT+16] Jesper Molin, Anna Bodén, Darren Treanor, Morten Fjeld, and Claes Lundström. Scale Stain: Multi-Resolution Feature Enhancement in Pathology Visualization. *arXiv preprint arXiv:1610.04141*, 2016.

[MCEF94] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *Ieee Computer Graphics and Applications*, pages 23–32, 1994.

[MM10] Stéphane Marchesin and Kwan-Liu Ma. Cross-Node Occlusion in Sort-Last Volume Rendering. page 8, 2010.

[MMD06] Stéphane Marchesin, Catherine Mongenet, and Jean-Michel Dischler. Dynamic load balancing for parallel volume rendering. In *EGPGV*, pages 43–50, 2006.

[MMD08] Stéphane Marchesin, Catherine Mongenet, and Jean-Michel Dischler. Multi-GPU Sort-last Volume Visualization. In *Proceedings of the 8th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '08, pages 1–8, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[MPHK94] Kwan-Liu Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, July 1994.

[NPB+12] O. Nocent, S. Piotin, A. Benassarou, M. Jaisson, and L. Lucas. 3d displays and tracking devices for your browser: A plugin-free approach relying on web standards. In *International Conference on 3D Imaging (IC3D)*, pages 1–8, 2012.

[Nvi] Nvidia CUDA programming guide 9.2. `https://docs.nvidia.com/cuda/index.html`. [Online ; accessed 2018-June-29].

[Ope] Openseadragon. `http://openseadragon.github.io`. [Online ; accessed 20-September-2018].

[PB13] Bernhard Preim and Charl P. Botha. *Visual Computing for Medicine: Theory, Algorithms, and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[PD84] Thomas Porter and Tom Duff. Compositing Digital Images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 253–259, New York, NY, USA, 1984. ACM.

[PGR+09] T. Peterka, D. Goodell, R. Ross, H. W. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10, November 2009.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007.

[QCJJ18] T. M. Quan, J. Choi, H. Jeong, and W. K. Jeong. An Intelligent System Approach for Probabilistic Volume Rendering Using Hierarchical 3d Convolutional Sparse Coding. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):964–973, January 2018.

[RGI07] E Romero, F Gómez, and M Iregui. Virtual microscopy in medical images: A survey. *Microscopy Book Series. Modern research and educational topics in microscopy. Badajoz: Formatex*, pages 996–11, 2007.

[RGW+03] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart Hardware-Accelerated Volume Rendering. page 9, 2003.

[RKH] Timo Ropinski, Jens Kasten, and Klaus Hinrichs. Efcient Shadows for GPU-based Volume Raycasting. page 8.

[RLKG+09] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*. Pearson Education, July 2009.

[RV06] Daniel Ruijters and Anna Vilanova. Optimizing GPU Volume Rendering. *Journal of WSCG*, page 8, 2006.

[Sch05] Henning Scharsach. Advanced GPU Raycasting. pages 69–76, 2005.

[SHN+06] Henning Scharsach, Markus Hadwiger, André Neubauer, Stefan Wolfsberger, and Katja Bühler. Perspective Isosurface and Direct Volume Rendering for Virtual Endoscopy Applications, 2006.

[SS10] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Transactions on Graphics (TOG)*, 29(6):179, 2010.

[SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. pages 187–241, June 2005.

[STKS17] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. IBM Power9 Processor Architecture. *IEEE Micro*, 37(2):40–51, March 2017.

[SZM+14] Min Shih, Yubo Zhang, Kwan-Liu Ma, Jayanarayanan Sitaraman, and Dimitri Mavriplis. Out-of-core visualization of time-varying hybrid-grid volume data. In *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, pages 93–100. IEEE, 2014.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 169–177, New York, NY, USA, 1998. ACM.

[WHH15] Ching-Wei Wang, Cheng-Ta Huang, and Chu-Mei Hung. VirtualMicroscopy: ultra-fast interactive microscopy of gigapixel/terapixel images over internet. *Scientific Reports*, 5:14069, September 2015.

[wik18] Octree, February 2018. Page Version ID: 828163396.

[Wil83] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph*, 1983.

[WJA+17] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, January 2017.

[WM92] Peter L. Williams and Nelson Max. A Volume Density Optical Model. In *Proceedings of the 1992 Workshop on Volume Visualization*, VVS '92, pages 61–68, New York, NY, USA, 1992. ACM.

[WMG98] Craig M. Wittenbrink, Thomas Malzbender, and Michael E. Goss. Opacity-weighted Color Interpolation, for Volume Sampling. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, pages 135–142, New York, NY, USA, 1998. ACM.

[WWH+00] Manfred Weiler, Rüdiger Westermann, Chuck Hansen, Kurt Zimmerman, and Thomas Ertl. Level-Of-Detail Volume Rendering via 3d Textures. pages 7–13, 2000.

[XYL+16] Jian Xue, Jun Yao, Ke Lu, Ling Shao, and Mohammad Muntasir Rahman. Efficient volume rendering methods for out-of-Core datasets by semi-adaptive partitioning. *Information Sciences*, 370-371:463–475, November 2016.

[YWM08] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 48:1–48:11, Piscataway, NJ, USA, 2008. IEEE Press.

[ZSJ+05] Jiawan Zhang, Jizhou Sun, Zhou Jin, Yi Zhang, and Qi Zhai. Survey of Parallel and Distributed Volume Rendering: Revisited. In *Computational Science and Its Applications – ICCSA 2005*, Lecture Notes in Computer Science, pages 435–444. Springer, Berlin, Heidelberg, May 2005.

[ZSL18] S Zellmann, J P Schulze, and U Lang. Rapid k-d Tree Construction for Sparse Volume Data. *Eurographics Symposium on Parallel Graphics and Visualization*, page 10, 2018.

Titre
Visualisations interactives haute-performance de données volumiques massives : une approche out-of-core multi-résolution basée GPUs

Résumé
Les besoins en visualisation de données volumiques sont courants dans plusieurs domaines scientifiques et en particulier en imagerie médicale et bio-médicale. En effet, plusieurs types d'appareils d'acquisition fréquemment utilisés, génèrent des champs scalaires ou vectoriels, représentés sous forme de grille régulière 3D, qu'il est important de pouvoir visualiser de manière interactive pour en extraire des informations, ou pour valider des résultats expérimentaux. L'accroissement de la précision d'acquisition de ces appareils modernes engendre cependant une hausse exponentielle de la quantité des données générées. Les algorithmes de visualisation doivent, non seulement, faire face à cette problématique en s'adaptant à la volumétrie des données qu'ils manipulent, mais aussi à cette évolution rapide. L'utilisation de cartes accélératrices de type GPU est particulièrement bien adaptée à la nature des données volumiques et aux algorithmes de visualisation généralement associés. Les environnements de calcul haute performance se tournent aujourd'hui vers des solutions qui utilisent un grand nombre de ces cartes. Ceux-ci sont, par leur nature massivement paralléle, de bons candidats pour proposer des solutions de visualisation haute performance. La quantité de mémoire des GPUs est cependant très limitée, et bien moins importante que les données brutes des volumes à manipuler. Une solution est alors de concevoir des algorithmes "out-of-core", dont l'unité de calcul est dissociée de l'unité de stockage des données.
Dans ces travaux de thèse, nous proposons un pipeline complet permettant de visualiser de manière interactive sur GPU, de très grands volumes de données dépassant les capacités physiques de la mémoire du GPU et du CPU de la machine sur laquelle est réalisé le rendu. Nous étudions pour celà, un modèle de gestion "out-of-core", basé sur un principe de virtualisation de la mémoire, particulièrement bien adapté à de très grands volumes. Nous proposons une approche qui comprend une structure d'adressage virtuel, entièrement gérée sur GPU. Nous nous intéressons également à la compatibilité de ce modèle pour différents types d'applications de visualisation de données volumiques. Une première qui s'appuie sur le principe de microscope virtuel pour proposer une visualisation 3D autostéréoscopique de piles d'images ultra haute résolution. Une deuxième qui propose un rendu volumique direct interactif par une approche "ray-guided", en montrant les capacités d'utilisation de notre modèle de gestion "out-of-core" dans des environnements de calcul haute performance hybrides, multi-GPUs, multi-CPUs.

Mots-clés
Visualisation scientifique, données volumiques, ultra haute définition, données biomédicales, GPU, out-of-core, multi-résolution, microscope virtuel, lancer de rayon volumique, multi-GPUs, rendu distribué, rendu distant, visualisation interactive

Title
High performance interactive visualization of large volume data: a GPU-based multiresolution out-of-core approach

Abstract
The needs for volume data visualization are common in several scientific fields, in particular in bio-medical imaging. Indeed, several types of frequently used acquisition devices generate scalar and vectorial fields represented as a 3D regular grid. It is important to be able to visualize interactively these volumes, in order to extract information or to validate experimental results. However, the increase in acquisition accuracy of these modern devices induces an exponential growth of the amount of data. To deal with this problem, visualization algorithms must be adapted, both, to the volume of data they handle and its steady growth. The use of GPU accelerator cards is particularly well suited to the nature of volume data and the associated visualization algorithms. High-performance computing environments are now turning to solutions that uses a large number of such cards. These are, by their massively parallel nature, good candidates to offer high-performance visualization solutions. However, the amount of memory in GPUs is very limited, and is much less important than the size of the raw data of the volumes to be handled. One solution is to design out-of-core algorithms, where the computing unit is dissociated from the data storage unit.
In this thesis work, we propose a complete pipeline for interactive visualization on the GPU of very large volumes of data exceeding the physical capacities of the GPU and the CPU memory independently of the machine used for the rendering. For this purpose, we study an out-of-core management model, based on a memory virtualization principle, particularly well adapted to very large volumes. We propose an approach that includes a virtual addressing structure, fully managed on the GPU. We are also interested in the compatibility of this model for different types of volume data visualization applications. We propose a first application that uses a virtual microscope principle to provide autostereoscopic 3D visualization of ultra-high resolution image stacks ; a second one that offers interactive direct volume rendering with a ray-guided approach, showing the usability of our out-of-core management model in hybrid, multi-GPUs, multi-CPUs high-performance computing environments.

Keywords
Scientific visualization, large volume data, biomedical dataset, GPU, out-of-core, multi-resolution, virtual microscope, volume ray-casting, multi-GPUs, distributed rendering, remote rendering, interactive visualization

Discipline
Informatique

Unité de Recherche
Unité de recherche EA 3804 CENTRE DE RECHERCHE EN STIC (CRESTIC)
UFR SCIENCES EXACTES ET NATURELLES
Moulin de la Housse
51687 REIMS CEDEX 2