Algorithmes sur GPU de visualisation et de calcul pour des maillages non-structurés

Luc Buatois

Nancy-Université INPL

Institut National

Polytechnique de Lorraine

Laboratoire LIAD/LORIA

École doctorale IAEM

Algorithmes sur GPU de visualisation et de calcul pour des maillages non-structurés

THÈSE

présentée et soutenue publiquement le 16 mai 2008

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine

Spécialité Informatique

par

Luc BUATOIS

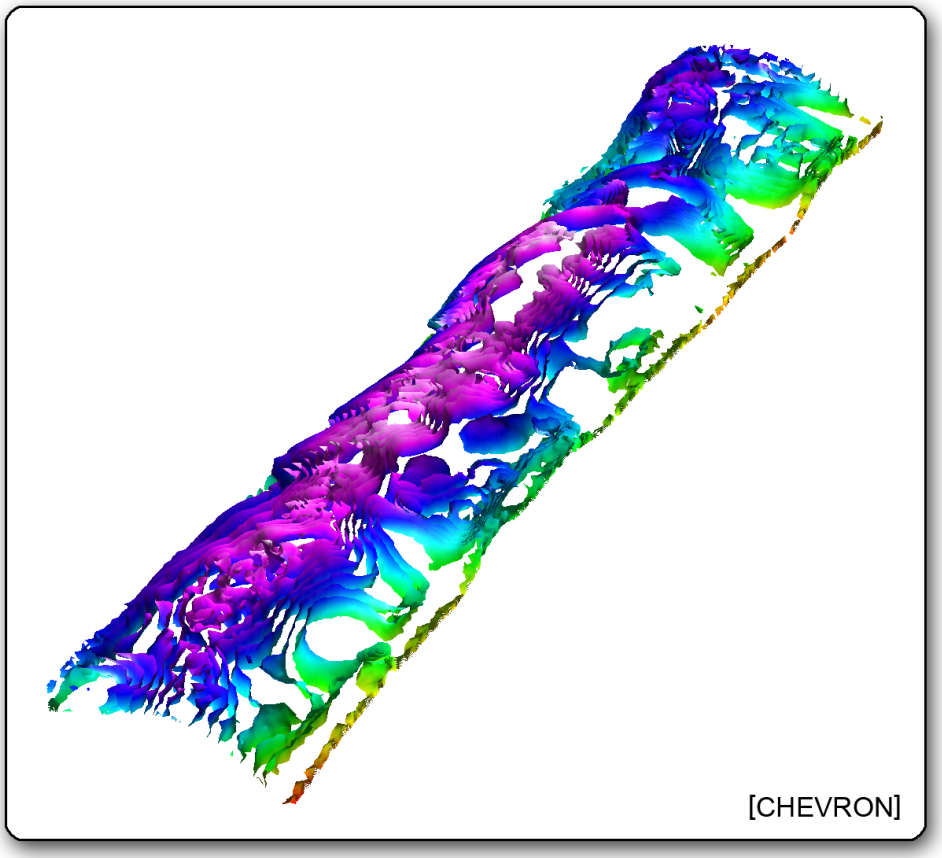
Composition du jury

Rapporteurs :	Georges-Pierre BONNEAU Jean-Michel DISCHLER
Examinateurs :	Dominique BECHMANN Sylvain LEFEBVRE Jean-Laurent MALLET Guillaume CAUMON
Directeurs :	Jean-Claude PAUL Bruno LÉVY

Centre de Recherches Pétrographiques et Géochimiques
Laboratoire d'Infographie et d'Analyse de Données
Rue du Doyen Marcel Roubault - 54500 Vandœuvre

Remerciements

Table des matières


[CHEVRON]

Introduction

1 Contexte pratique de la thèse

Les travaux présentés dans ce mémoire s'inscrivent dans le cadre du projet GOCAD[1] de l'École Nationale de Géologie de Nancy en coopération avec l'équipe ALICE[2] de l'INRIA Lorraine. L'objectif initial du projet GOCAD, lancé à la fin des années 80, est de proposer des outils informatiques de modélisation 3D de phénomènes géophysiques, également appelée *géomodélisation*. Le logiciel GOCAD est actuellement maintenu et développé commercialement par la société EarthDecision/Paradigm Geophysical[3]. L'équipe ALICE traite de problèmes fondamentaux de l'informatique graphique, plus particulièrement dans les domaines du traitement numérique de la géométrie et de la simulation des interactions lumineuses.

2 Thème d'étude

Les algorithmes les plus récents de traitement numérique de la géométrie ou bien encore de simulation numérique de type CFD (Computational Fluid Dynamics) utilisent à présent de nouveaux types de grilles composées de polyèdres arbitraires, autrement dit des grilles fortement non-structurées. Dans le cas de simulations de type CFD, ces grilles peuvent servir de support à des champs scalaires ou vectoriels qui représentent des grandeurs physiques (par exemple : densité, porosité, perméabilité).

La problématique de cette thèse concerne la définition de nouveaux outils de visualisation et de calcul sur de telles grilles. Pour la visualisation, cela pose à la fois le problème du stockage et de l'adaptativité des algorithmes à une géométrie et une topologie variables. Pour le calcul, cela pose le problème de la résolution de grands systèmes linéaires creux non-structurés. Pour aborder ces problèmes, l'augmentation incessante ces dernières années de la puissance de calcul parallèle des processeurs graphiques ou GPU[4] nous fournit de nouveaux outils.

[1] http ://www.gocad.org
[2] http ://alice.loria.fr
[3] http ://www.earthdecision.com (entité de Paradigm depuis août 2006, http ://www.paradigmgeo.com)
[4] GPU : Graphics Processing Unit.

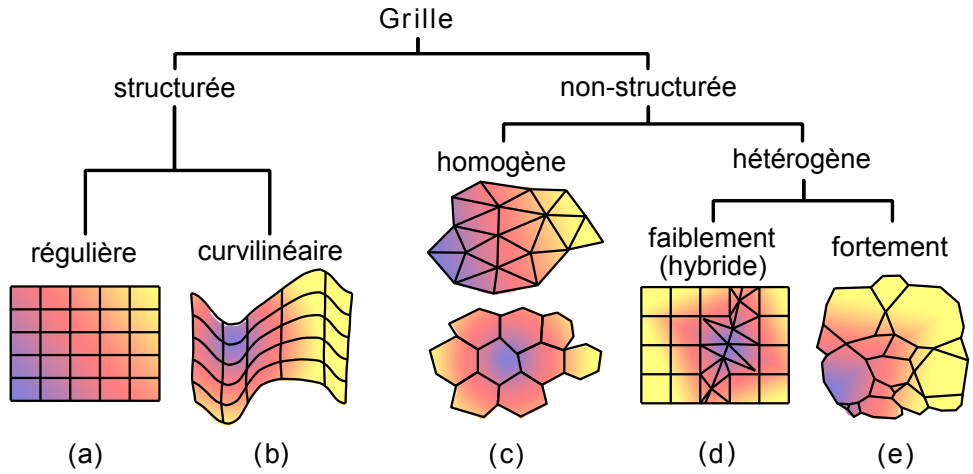


FIG. 1 – Nomenclature des différents types de grilles volumiques d'après Caumon *et al.* (2005).

3 Supports de modélisation et problématiques idoines

Qu'il soit question de simulation d'écoulement fluide CFD, de traitement numérique de la géométrie ou de visualisation de modèles bio-médicaux, chaque problématique abordée utilise un support de stockage volumique spécifiquement adapté. Ces supports de stockage volumique sont communément appelés des grilles ou des maillages. Ceux-ci sont constitués de sommets et d'arrêtes qui forment un ensemble de cellules. Des propriétés peuvent être stockées sur chacun des sommets d'un maillage. Ces propriétés peuvent être scalaires ou vectorielles, et peuvent contenir des nombres entiers, des nombres réels (on parlera de nombres à virgule flottante ou nombre flottants), etc.

Les grilles sont habituellement classées en deux familles (figure 1) : lorsque la topologie du maillage est constante (chaque cellule a un nombre de voisins constant, exception faite des cellules du bord), on parle d'une grille structurée, dans le cas contraire, d'une grille non-structurée.

La famille des grilles structurées est subdivisée en deux sous-familles : les grilles régulières dont la géométrie est implicite et les grilles curvilinéaires dont la géométrie doit être stockée explicitement. Historiquement, les grilles régulières ont été les premières à être largement utilisées dans de nombreux domaines du fait de leur simplicité d'implantation, leur consommation mémoire limitée, et de la facilité à discrétiser n'importe quelle équation/algorithme sur ce type de maillage. Les grilles curvilinéaires furent par la suite développées afin d'autoriser des modélisations plus fines tout en conservant une topologie constante et implicite.

La famille des grilles non-structurées est elle aussi subdivisée en deux sous-familles : les grilles homogènes constituées de cellules ayant toutes la même topologie, et les grilles hétérogènes constituées de tout type de cellule. Une dernière subdivision est effectuée dans cette classification : les grilles non-structurées et hétérogènes peuvent l'être faiblement ou fortement. Dans le cas faible, la grille contient un nombre restreint de types de cellules non-structurées (la grille peut contenir, par exemple, à la fois des hexaèdres et des tétraèdres). Ce type de grille peut

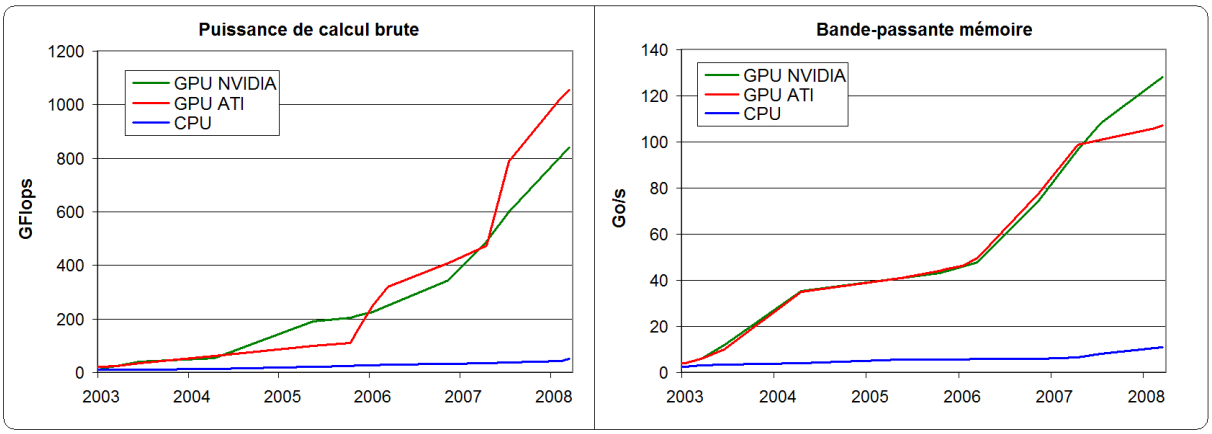


FIG. 2 – Evolutions de la puissance de calcul et de la bande-passante mémoire sur CPU et sur GPU AMD-ATI et NVIDIA. La puissance de calcul est mesurée en milliards d'opérations sur des nombres flottants par seconde ou GFlops (Giga FLoating-point Operations Per Second). La bande-passante mémoire est mesurée en giga-octets par seconde.

être utilisé dans le but de se conformer au mieux à des phénomènes physiques. Par exemple, dans l'étude d'un réservoir pétrolier, il est possible d'utiliser des hexaèdres pour modéliser les couches géologiques, et des tétraèdres pour modéliser les abords des puits et des failles (Flandrin *et al.*, 2004; Balaven *et al.*, 2000; Balaven-Clermidy, 2001; Lepage, 2004). Dans le cas d'une grille non-structurée et fortement hétérogène, aucune restriction n'est faite sur la topologie ou la géométrie des cellules. On parlera plus simplement de grilles fortement non-structurées. Ces grilles totalement génériques permettent enfin de créer des modèles qui collent au mieux à la réalité. Néanmoins, comme nous le verrons, cette généricité impose de développer de nouvelles techniques tant de visualisation que de calcul dont la complexité est amplifiée du fait de la variabilité de la topologie et de la géométrie des cellules. Ceci est d'autant plus vrai dans notre cas puisque nous avons utilisé de nouvelles techniques exploitant des processeurs graphiques pour accélérer le traitement des grilles non-structurées.

En effet, en plus de leur grande puissance de calcul et de leur large bande-passante mémoire (figure 2), les processeurs graphiques ont l'avantage d'offrir ces performances à un coût financier réduit ainsi qu'une consommation électrique faible (sur les dernières générations de processeurs graphiques, il est possible d'atteindre les 2GFlops/Watt, et environ le dixième seulement sur les derniers CPU). Toutefois, l'utilisation de ces GPU nécessite de définir de nouveaux algorithmes adaptés aux modèles de programmation parallèle qui leur sont spécifiques. L'exploitation des processeurs graphiques à des fins de calculs génériques est plus généralement désigné par le terme GPGPU[5] pour General-Purpose Computation Using Graphics Hardware.

[5] www.gpgpu.org

4 Problématiques générales, contributions et organisation du mémoire

Le premier chapitre de ce mémoire aborde le problème de la visualisation de maillages fortement non-structurés porteurs de champs scalaires. L'algorithme présenté est dit d'*extraction* car celui-ci extrait des données du volume pour ne visualiser que ce petit sous-ensemble. Le premier chapitre se concentre donc sur les techniques d'extraction de surfaces d'isovaleurs ou *isosurfaces*. Ce chapitre est subdivisé en trois sections :

- La section 1.2 présente notre algorithme, le *Marching Cells*, qui est une extension d'algorithmes de visualisation pré-existants mais adapté aux cas des grilles fortement non-structurées. Le but du Marching Cells est d'extraire efficacement des *isosurfaces* grâce à l'accélération d'un GPU sur de telles grilles.
- Dans la section 1.3 nous présentons et développons de nouvelles techniques de classification de données adaptatives. Ces méthodes permettent de pré-sélectionner rapidement des données en amont des algorithmes d'extraction. En pré-éliminant certaines parties du maillage qui ne contribuent pas au résultat final devant être visualisé, ces algorithmes accélèrent fortement ceux d'extraction et de visualisation.
- La dernière section 1.4 du premier chapitre présente notre algorithme de *Morphing 4D*. Celui-ci explique comment interpoler des isosurfaces suivant la dimension temporelle. L'objectif est de pouvoir visualiser de manière continue l'évolution d'une isosurface à travers le temps afin d'améliorer la perception de l'utilisateur.

De nombreux travaux mènent à résoudre des équations aux dérivées partielles : simulation d'écoulement fluide, lissage de surface, paramétrisation de surface etc. Afin de résoudre numériquement ces équations, elles sont discrétisées. Cette discrétisation mène à écrire de grands systèmes d'équations creux. L'organisation des coefficients non-nuls de la matrice correspondante, dans le cas des grilles non-structurées, ne présente pas de motif particulier ce qui impose d'utiliser des structures de stockage des matrices creuses totalement génériques. La résolution de tels systèmes d'équations est très consommatrice en temps de calcul. Notre idée, développée au chapitre 2, est d'utiliser la puissance des GPU pour accélérer la résolution de ces grands systèmes d'équations. Ce second chapitre détaille comment implanter efficacement sur GPU à la fois des matrices creuses génériques, des opérations d'algèbres linéaires sur ces matrices, et un solveur numérique creux générique. Toutes ces implantations sont hautement-parallélisées et optimisées afin d'exploiter au mieux les spécificités et capacités des processeurs graphiques actuels. Au final, notre solveur sur GPU ne se limite donc pas à la résolution de quelques problèmes d'optimisation, mais à tout ceux qui nécessitent de résoudre un système linéaire.

Chapitre 1

Visualisation volumique de maillages non-structurés

Sommaire

1.1 Introduction

Les progrès en modélisation numérique de processus physiques, de simulations d'écoulements dynamiques, de modèles médicaux, de tomographie... ont ouverts la voie à de nouveaux types de grilles composées, par exemple, de polyèdres complexes, autrement dit des grilles fortement non-structurées. La visualisation de champs scalaires définis sur de telles grilles, souvent d'une taille très significative, impose de pouvoir généraliser les algorithmes pré-existants sur des grilles régulières tout en garantissant des performances de premier plan. Étant donné la nature fortement non-structurée des grilles étudiées, ceci n'est pas trivial.

L'objectif de ce chapitre est de présenter de nouveaux algorithmes de visualisation haute-performance qui soient adaptés aux exigences des récentes avancées en terme de génération de maillages fortement non-structurés. Visualiser de tels maillages pose de nombreux problèmes qui souvent se traduisent par des algorithmes complexes et extrêmement coûteux en terme de puissance de calcul et de mémoire. La montée en puissance de calcul des GPU ces dernières années ainsi que la possibilité de les programmer, en font des cibles de choix pour l'implantation d'algorithmes hautement parallèles.

Ce chapitre présente toute une chaîne de méthodes qui permettent de visualiser interactivement de très grandes grilles non-structurées, grâce à l'accélération d'un GPU. Dans un premier temps, la section 1.2 présente un nouvel algorithme d'extraction d'isosurfaces générique hautement parallélisé et implanté sur GPU : le *Marching Cells*. La section 1.3 suivante présente comment combiner le Marching Cells avec de nouvelles méthodes d'accélération d'extraction d'isosurfaces à l'aide de structures de pré-classification de cellules. Quant à la dernière section 1.4 de ce chapitre, elle propose d'étendre l'ensemble de ces méthodes aux données ayant une composante temporelle. De plus, elle propose un algorithme d'interpolation temporelle permettant de visualiser de manière continue dans le temps l'évolution d'isosurfaces. On parlera de *Morphing 4D*.

Dans l'ensemble de ce chapitre, seuls des champs scalaires sont étudiés, voire des séries de champs scalaires lorsqu'une dimension temporelle est prise en compte.

1.2 Nouvelle méthode d'extraction d'isosurfaces sur GPU pour des maillages fortement non-structurés : *Le Marching Cells*

Nous décrivons et validons dans cette section notre algorithme du *Marching Cells* (Buatois *et al.*, 2006b,a) qui permet, par rapport aux approches précédentes (Pascucci, 2004; Reck *et al.*, 2004; Klein *et al.*, 2004; Kipfer et Westermann, 2005), d'accélérer l'extraction des isosurfaces sur GPU pour des maillages fortement non-structurés. De plus, notre algorithme prend en charge des grilles de très grandes tailles grâce à une stratégie de stockage basée sur des textures. Cette dernière élimine toute redondance dans les données stockées. De plus, nous proposons de traiter directement toute cellule polyédrique sans avoir recours à une pré-subdivision en tétraèdres de ces cellules (autrement appelée la *tétraédrisation*) comme on le trouve fréquemment dans la littérature (Max *et al.*, 1990; Nielson et Sung, 1997; Pascucci, 2002, 2004; Silva et Mitchell, 1997).

La sous-section 1.2.2 fait une synthèse détaillée des méthodes pré-existantes qui généralement n'accélèrent, sur GPU, que des grilles régulières à l'aide méthodes de rendu dites *volumiques*, ou des grilles non-structurées tétraédriques à l'aide de méthodes d'extraction d'isosurfaces. La sous-section 1.2.3 résume nos contributions, la 1.2.4 fournie une description détaillée de notre approche, la 1.2.5 la valide sur des grilles tétraédriques, hexaédriques et prismatiques[6], et enfin la 1.2.6 évoque les évolutions possibles de notre méthode.

1.2.1 Matériel graphique standard

Ce paragraphe traite du fonctionnement des processeurs graphiques et précise le vocabulaire que nous allons employer dans ce mémoire.

Depuis le milieu des années 90, les processeurs CPU se sont vu déchargés de plus en plus du traitement des tâches liées au rendu graphique. Ces tâches ont été prises en charge par des processeurs dédiés au graphisme : les GPU. Ceux-ci ont acquis au fur et à mesure de leur évolution une mémoire propre dédiée, la capacité à dessiner des objets 2D puis 3D. L'ensemble des étapes de traitement intervenants dans un GPU sont regroupées sous l'appellation : *pipeline graphique*. La figure 1.1 présente l'évolution de ce pipeline depuis le début des années 2000. Notez que l'ensemble du pipeline graphique a été pensé pour l'industrie du jeux-vidéo, donc essentiellement pour pouvoir visualiser des objets 3D représentés par des surfaces complexes. Le pipeline graphique prend en entrée une représentation 3D de ces objets et affiche en sortie une image plane 2D de ceux-ci dans l'espace de l'écran d'affichage. La communication entre l'application et le GPU se fait grâce à des interfaces de programmation ou API (de l'anglais Application Programming Interface) spécialisées telles que OpenGL (Segal et Akeley, 2004) ou DirectX (Microsoft Corporation, 2006).

Comme le montre la figure 1.1, le pipeline graphique prend en entrée un flux de *sommets* qui peuvent être *assemblés* en *primitives* simples (points, lignes, triangles, polygones). Cependant, l'image finale est stockée dans la mémoire écran (le *framebuffer* en anglais) sous forme

[6]Une grille prismatique est définie par une surface polyédrique qui sert de base à une extrusion multicouches suivant des vecteurs prédéfinis (figure 1.22)

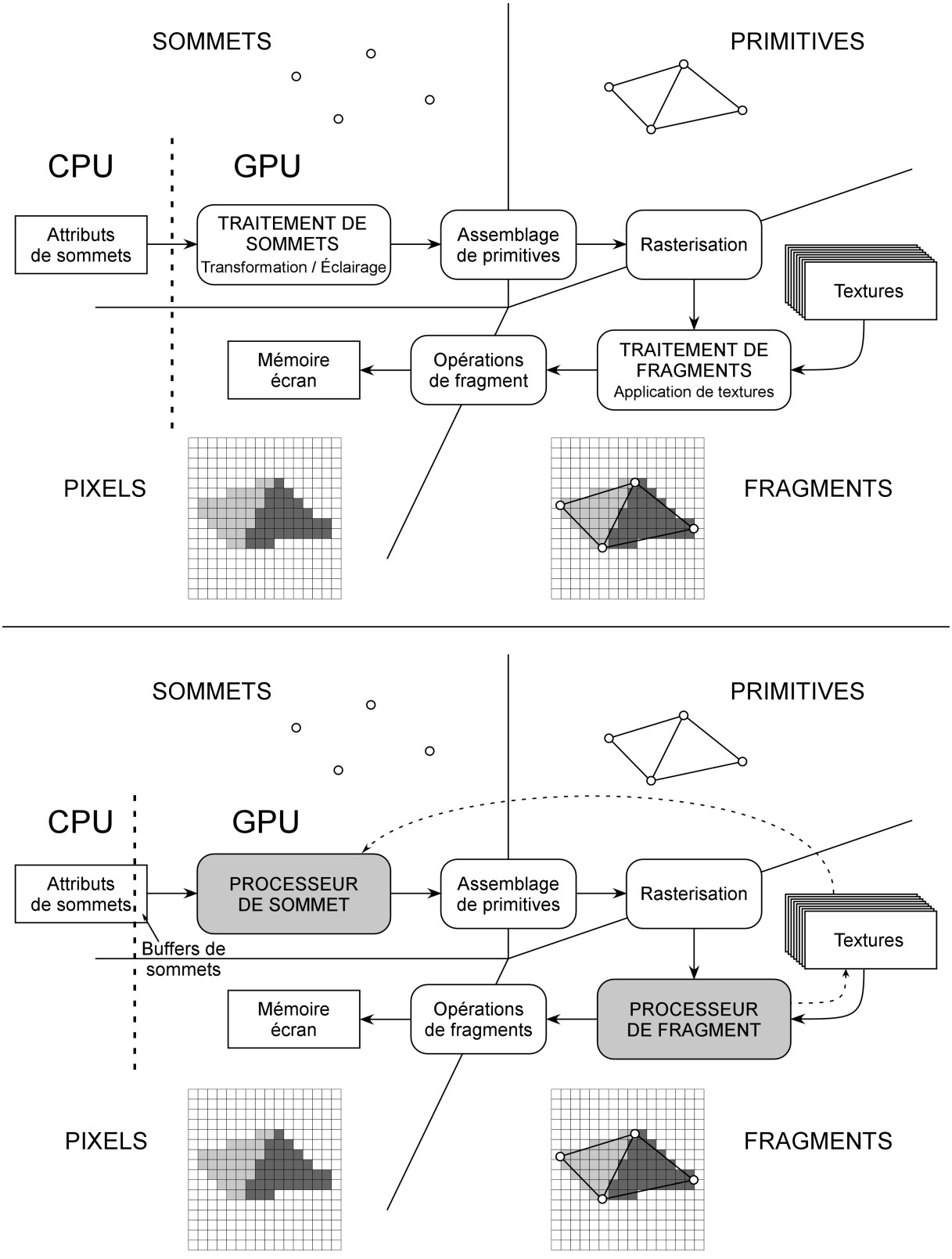


FIG. 1.1 – Principe de fonctionnement du pipeline graphique standard (haut) et son évolution vers plus de flexibilité depuis le début des années 2000 (bas). Figure d'après Castanié (2006).

d'une matrice de pixels. Entre les primitives et les pixels, les GPU manipulent des éléments intermédiaires qui peuvent être vu comme l'équivalent des pixels mais en 3D : les *fragments*. Le GPU va ainsi générer à l'étape de rasterisation un fragment dans l'espace pour chaque primitive qui affecte un pixel donné. Ainsi, si plusieurs primitives affectent un même pixel, plusieurs fragments sont générés indépendamment, et la couleur finale du pixel pourra tenir compte de chacun d'entre eux et de leur position respective dans l'espace, et ce grâce aux opérations de fragment. Par exemple, les occlusions éventuelles peuvent être résolues grâce à l'utilisation du z-buffer. En effet, ce dernier stocke la profondeur du dernier fragment ce qui permet d'ignorer ceux plus profonds.

Sur le haut de la figure 1.1, deux unités sont essentielles dans le pipeline graphique classique : l'unité de *traitement de sommets* et l'unité de *traitement de fragments*. L'unité de traitement de sommets a pour fonction de transformer les sommets de leur espace de représentation 3D vers l'espace 2D de l'écran. D'autre part, cette unité est également en charge de réaliser les calculs d'éclairage en fonction des sources lumineuses virtuelles placées dans la scène. Ces opérations sont calculées indépendamment pour chaque sommet puis interpolées linéairement à l'étape de rasterisation. Les fragments générés par cette étape sont ensuite traités par l'unité de traitement de fragments. A ce moment là, une texture peut être appliquée au fragment. Une texture est une image 1D, 2D ou même 3D stockée sous la forme de texels[7] en mémoire graphique qui vont être plaqués sur les primitives en fonction de coordonnées 2D définies sur les sommets et interpolées linéairement au niveau des fragments. On parle couramment de plaquage de texture ou de *texture mapping* en anglais. Notez que les unités de traitement de sommets et de traitement de fragments sont constituées de plusieurs sous-unités de calcul parallèles indépendantes appelées respectivement *vertex pipelines* et *pixel pipelines*.

Ce pipeline graphique a permis une évolution rapide des performances jusqu'à la fin des années 90, mais, de part sa conception figée, ne laissait pas de place pour la programmation de nouveaux effets non-*pré-cablés*. Vers le début des années 2000 (bas de la figure 1.1), ce pipeline a progressivement été ouvert à l'exécution de code utilisateur en permettant aux unités parallèles de traitement de sommets et de fragments de devenir programmables. Il n'est dès lors plus question d'unités de traitement de sommets et de fragments mais désormais de *processeurs de sommets* et de *fragments* (ou respectivement *Vertex Shader units* et*Pixel Shader units* en anglais). Ces processeurs sont programmés à l'aide de langages spécifiques de bas niveau en assembleur grâce à des extensions OpenGL (Kilgard, 2003) ou bien de haut niveau grâce à des langages comme l'*OpenGL Shading Language* ou *GLSL* (Kessenich *et al.*, 2003; Rost, 2004), le langage d'NVIDIA Cg (Fernando et Kilgard, 2003) ou celui de Microsoft (Microsoft Corporation, 2002).

Notez qu'au cours de l'évolution du pipeline graphique, l'accès aux textures n'était, au départ, disponible que pour les processeurs de fragments. L'accès aux textures depuis les processeurs de sommets ne fût implanté qu'à partir du milieu de l'année 2005, ce qui a ouvert de nouvelles possibilités que nous avons exploré par la suite dans ce chapitre.

[7]Un *texel* est le plus petit élément d'une texture (de l'anglais *Texture Element*) pouvant contenir de une à quatre composantes qui correspondent aux trois couleurs primaires Rouge, Vert et Bleu (RVB) et à une valeur d'opacité, couramment appelée Alpha (A). Un texel peut contenir des nombres entiers ou des nombres flottant sur 8, 16 ou même plus récemment sur 32 bits.

Le pipeline graphique présenté à la figure 1.1 illustre le cheminement algorithmique de tout rendu effectué sur un GPU. En revanche, il n'impose pas d'implantation hardware particulière. Jusqu'au milieu des années 2000, le pipeline graphique était pris en charge par des unités de calcul spécialisées et dédiées à chaque étape du pipeline. Ainsi, un groupe de processeurs parallèles prenait en charge le traitement des sommets et l'exécution d'un code utilisateur éventuellement associé, tandis qu'un second groupe s'occupait du traitement des fragments. Cette architecture rigide ne permettait pas de maximiser l'utilisation de toute la puissance de calcul disponible puisque dans certains cas les processeurs de sommets pouvaient être exploités à leur maximum mais pas ceux de fragments, et inversement. Pour remédier à cette situation, en 2006, l'architecture hardware des processeurs graphique a été *unifiée*. Concrètement, des unités de calcul parallèles génériques ont été développées afin de remplacer simultanément les processeurs de sommets et de fragments. Les unités de calcul génériques sont alors dynamiquement allouées au traitement des sommets ou des fragments afin de maximiser les performances globales. Cette architecture permet de bénéficier de toute la puissance de calcul parallèle des GPU en toute circonstance.

Notez que les GPU supportent depuis quelques années déjà le standard IEEE-754 sur les nombres flottants simple précision (32 bits), mais ne supportent pas encore les nombres flottants en double précision (64 bits). *In fine*, les processeurs graphiques représentent aujourd'hui une formidable ressource de puissance de calcul parallèle aussi bien pour résoudre des problématiques de visualisation que de calcul.

1.2.2 Définitions et état de l'art

Qu'est-ce qu'une isosurface ?

Soit $\mathcal{F}$ la fonction qui à tout point de l'espace associe une valeur scalaire. Alors pour tout scalaire α appelé *isovaleur* on définit une surface appelée *isosurface* comme étant l'ensemble des points de l'espace qui prennent pour valeur α selon $\mathcal{F}$. Une isosurface se définit donc comme suit :

$$\forall \alpha \in \mathbb{R}, isosurface(\alpha) = \{x \in \mathbb{R}^3 | \mathcal{F}(x) = \alpha\}$$

Différentes techniques existent pour extraire et visualiser des isosurfaces. Elles peuvent être classées en deux familles : les méthodes de *rendu volumique* et les méthodes d'*extraction* (Rezk-Salama, 2001). Les méthodes de rendu volumique visualisent les données dans leur ensemble là où les méthodes d'extraction cherchent à n'en traiter qu'un sous-ensemble restreint voisin de l'isosurface.

La figure 1.2 présente les trois méthodes de visualisation volumique les plus connues à ce jour : section, isosurface et rendu volumique. La visualisation par section est une visualisation par extraction. Celle-ci consiste à extraire un plan orthogonal à l'un des axes du système de coordonnées utilisé tout en le colorant en fonction du champ scalaire étudié (1.2-(a)). Malgré son intérêt pratique dans un but d'analyse de données, sur des grilles structurées, l'extraction de sections ne pose pas de réels problèmes algorithmiques, exception faite du cas où un très

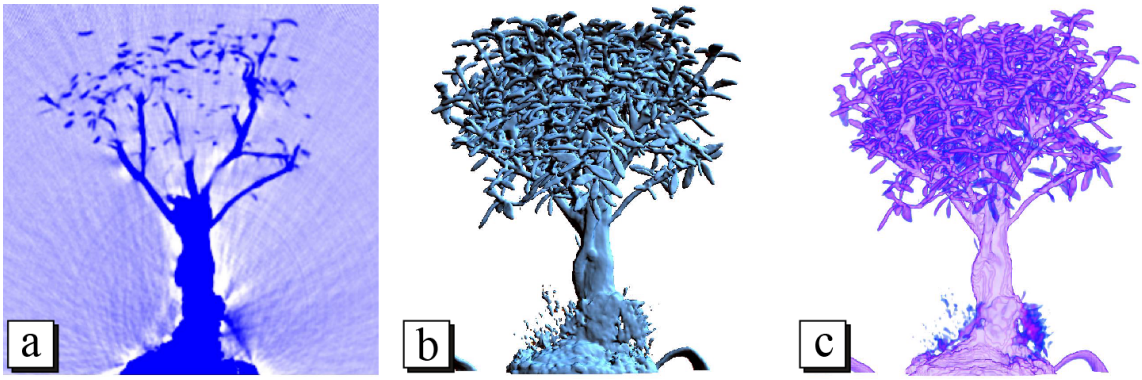


FIG. 1.2 – Ces figures illustrent différentes méthodes de visualisation volumiques : (a) par sections planaires, (b) par isosurfaces et (c) par rendu volumique. Ce jeu de données représente un bonsaï et est défini sur une grille régulière composée de 8'388'608 cellules. Figure d'après Castanié (2006).

grand jeu de données serait utilisé (plusieurs centaines de giga-octets), cas déjà abordé par L. Castanié (Castanié *et al.*, 2005; Castanié, 2006). Sur des grilles non-structurées, cette extraction peut être vue comme un cas particulier d'extraction d'isosurfaces planaires. Les paragraphes suivants abordent largement la problématique de l'extraction d'isosurfaces dans le cas général, et c'est pourquoi l'extraction de section ne sera pas davantage développée dans ce mémoire.

Les sous-figures 1.2-(c) et (b) présentent des techniques sur lesquelles nous revenons en détails dans les paragraphes suivants, dans l'ordre : le rendu volumique et l'extraction d'isosurfaces.

Méthodes de *rendu volumique*

Les méthodes de *rendu volumique* sont apparues dans les années 80 (Kajiya et Herzen, 1984; Drebin *et al.*, 1988; Sabella, 1988). Elles visualisent les données dans leur ensemble.

Intégrale de rendu volumique L'approche par rendu volumique considère le volume de la grille Ω comme un milieu semi-transparent composé de particules élémentaires (figure 1.2-(c)). Ces particules possèdent des propriétés optiques qui permettent de simuler la propagation de la lumière dans le volume en fonction de divers modèles physiques (Max, 1995). Le modèle le plus couramment utilisé est celui dit d'*émission-absorption* dans lequel chaque particule peut émettre et absorber un rayonnement lumineux. Il est ainsi possible d'accumuler la contribution de chaque particule élémentaire le long d'un rayon lumineux virtuel passant par l'oeil d'un observateur et de calculer l'intensité lumineuse qui lui parvient.

Soit $x(t)$ la paramétrisation de ce rayon où t représente la distance par rapport à l'oeil de l'observateur, D la distance à partir de laquelle le rayon quitte le volume de la grille Ω, et $e(x)$ et $a(x)$ les fonctions qui définissent l'intensité lumineuse respectivement émise et absorbée au point x. Alors, l'intensité lumineuse I reçue est définie par l'*intégrale de rendu volumique* :

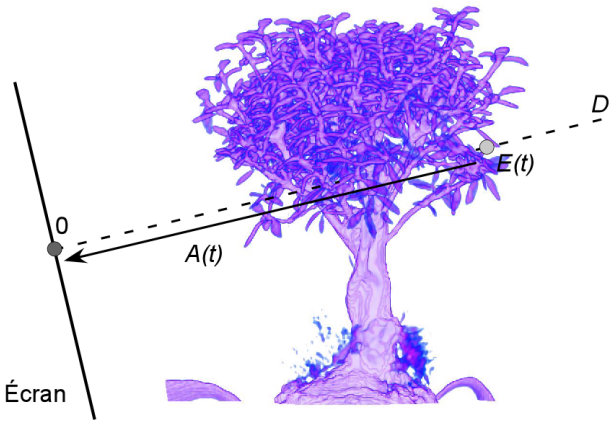


FIG. 1.3 – Principe du rendu volumique avec un modèle d'émission-absorption. L'émission lumineuse $E(t)$ à l'abscisse t est progressivement absorbée par les autres particules élémentaires rencontrées le long du rayon entre t (gris clair) et 0 (gris foncé), ce qui est simulé par la fonction $A(t)$ (d'après Castanié (2006)).

$$I = \int_0^D E(t).e^{-A(t)}dt$$

avec

$$E(t) = e(x(t)) \text{ et } A(t) = \int_0^t a(x(t'))dt'$$

$E(t)$ simule ainsi le rayonnement émis par chaque particule et $A(t)$ l'atténuation par absorption entre le point d'émission et le point d'observation. La figure 1.3 illustre le principe de fonctionnement de ce système d'émission-absorption.

Il existe des alternatives au rendu volumique par intégration le long d'un rayon, par exemple celle dite de projection de l'intensité maximale, ou MIP de l'anglais *Maximal Intensity Projection* (Wallis *et al.*, 1989; Rezk-Salama *et al.*, 2004). En fait cette dernière ne requière aucune intégration numérique. Elle ne conserve que l'intensité maximale émise le long de chaque rayon :

$$I = max_{t\in[0,D]}(E(t))$$

Le rendu MIP est fréquemment utilisé en imagerie médicale pour sa simplicité et son efficacité pour visualiser des données tomographiques issues d'angiographies. La figure 1.4 compare le rendu volumique par intégration complète (a) et par intensité maximum MIP (b). Tandis qu'en (a) une fonction de transfert doit être correctement définie (voir les paragraphes suivants) afin de faire ressortir les vaisseaux sanguins, ceux-ci apparaissent naturellement en (b). L'inconvénient principal de la méthode MIP est qu'elle perd totalement l'information de profondeur dans les images produites. Cela implique un fort risque de mauvaise interprétation de la cohérence spatiale des différentes structures comme l'a montré Hastreiter (1999) de manière saisissante.

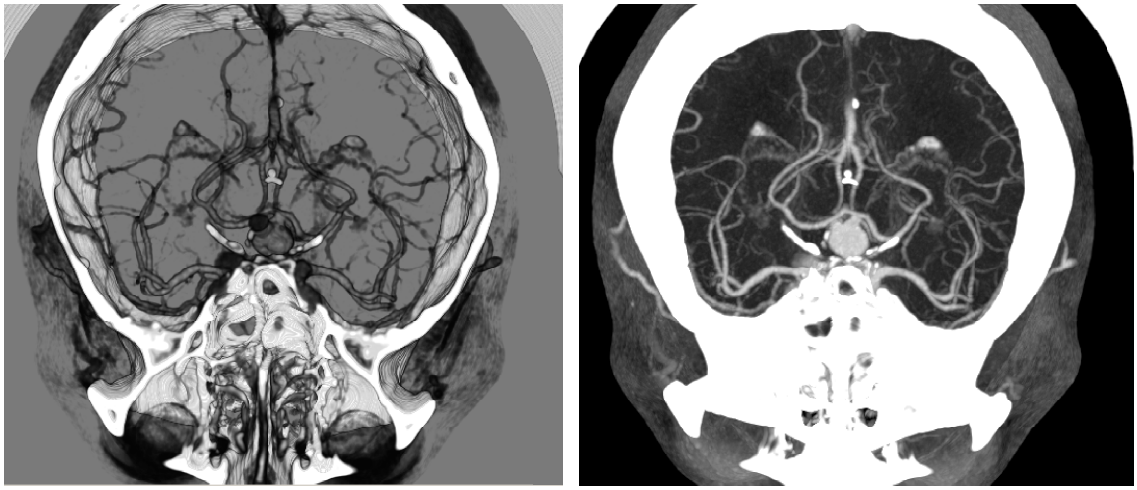


FIG. 1.4 – Comparaison de la tomographie d'une angiographie sur des vaisseaux sanguins à l'intérieur d'un crâne humain calculée par le modèle d'émission-absorption (a) et par projection de l'intensité maximale MIP (b) (d'après Rezk-Salama *et al.* (2004)).

La méthode dite des *Rayons X* est également très répandue. Celle-ci propose un modèle sans absorption où seul l'émission est prise en compte :

$$I = \int_0^D E(t)dt$$

Le résultat d'une telle intégration est très similaire à celui d'une véritable radiographie par rayons X.

L'équation générale du rendu volumique qui utilise le modèle d'émission-absorption nécessite une étape supplémentaire pour visualiser un champ scalaire continu dans l'espace. Cette étape de classification applique une fonction de transfert qui à chaque valeur du champ scalaire associe une valeur d'émission et une valeur d'absorption (Pfister *et al.*, 2001). En pratique, la valeur d'émission fournie par la fonction de transfert correspond à une couleur dans l'espace RVB (Rouge, Vert et Bleu) tandis que la valeur d'absorption correspond à une opacité. La fonction de transfert est donc une fonction de $\mathbb{R}$ dans $\mathbb{R}^4$ qui à un scalaire associe un 4-uplet RVBA, où RVB est l'émission propre de la particule et A son absorption. La figure 1.5 montre comment une fonction de transfert peut être représentée graphiquement.

Le paramétrage de la fonction de transfert n'est pas trivial et nécessite un soin particulier. De nombreux travaux ont d'ailleurs été réalisé dans le but d'automatiser cette tâche (Kindlmann et Durkin, 1998).

La visualisation d'isosurfaces à l'aide d'un rendu volumique peut être réalisée en utilisant une fonction de transfert spécifique qui à toute valeur scalaire différente de l'isovaleur associe des valeurs d'émission et d'opacité nulles.

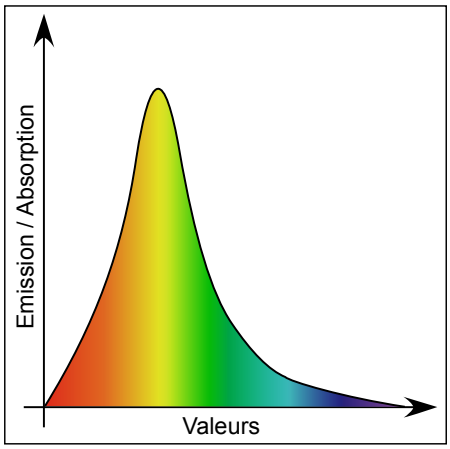


FIG. 1.5 – Fonction de transfert qui à toute valeur scalaire associe une valeur d'émission (couleur) et une valeur d'absorption (opacité).

En pratique : intégration numérique/projection Le modèle d'émission-absorption ne peut être calculé analytiquement. Il convient donc de discrétiser en sommes de Riemann les intégrales du modèle, puis d'utiliser des développements de Taylor dans un but de simplification. Le choix de la méthode d'échantillonnage est critique car pouvant mener à de nombreuses incohérences et imprécisions.

Le calcul de l'équation d'émission-absorption peut être réalisé de plusieurs manières :

- Explicite : par un lancé de rayons avec un échantillonnage le long de celui-ci (en anglais *Ray Casting* (Kajiya et Herzen, 1984; Levoy, 1988; Sabella, 1988; Garrity, 1990; Levoy, 1990; Wilhelms *et al.*, 1990)).
- Implicite : par une projection dans l'espace de l'écran de chaque élément de la grille dans l'ordre du plus lointain au plus proche de l'écran (en anglais : *Splatting* (Upson et Keeler, 1988; Westover, 1989, 1990; Shirley et Tuchman, 1990)).

Pour les méthodes explicites, chaque pixel de l'image finale est considéré indépendamment et son rayon associé est discrétisé explicitement. La contribution de l'ensemble du volume le long de chaque rayon est alors accumulée. Inversement, les méthodes implicites projettent les éléments de la grille dans l'espace de l'écran pour composer successivement les pixels. Les rayons sont ainsi discrétisés collectivement et implicitement. L'ordre de projection est fondamental du fait de la non-commutativité de la composition (Williams, 1992; Silva *et al.*, 1998; Cignoni *et al.*, 1998; Comba *et al.*, 1999; Callahan *et al.*, 2005). Celui-ci doit s'effectuer de l'élément le plus loin à l'élément le plus proche afin de respecter l'ordre de composition imposé par l'intégrale de rendu volumique du modèle d'émission-absorption. La figure 1.6 illustre un échantillonnage selon une technique explicite et une implicite.

La discrétisation de l'équation de rendu volumique est un problème fondamental qui a été longuement étudié dans la littérature (Rezk-Salama *et al.*, 2004). Le cas classique des grilles régulières ne pose plus réellement de problème depuis qu'il est possible d'accélérer à l'aide d'un GPU moderne le rendu dans le cas implicite grâce, par exemple, au placage de texture 2D/3D (Cabral

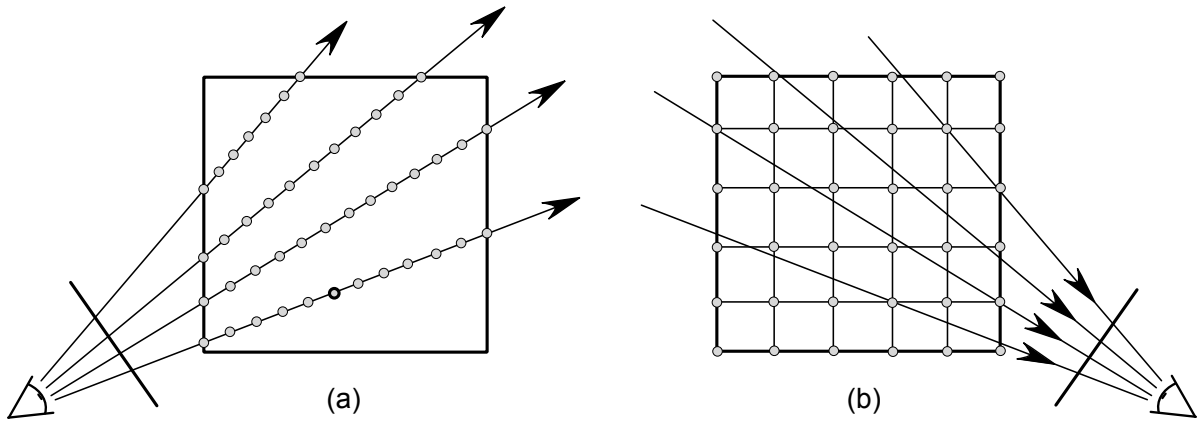


FIG. 1.6 – Discrétisation explicite par lancer de rayons (a) et implicite par splatting (b) de l'intégrale de rendu volumique.

et al., 1994; Engel *et al.*, 2001; Castanié, 2006) ou dans le cas explicite à un lancé de rayons utilisant des shaders (Röttger *et al.*, 2003; Krüger et Westermann, 2003a). L'attention se porte plus actuellement sur la visualisation des grilles non-structurées (Silva *et al.*, 2005; Krüger et Westermann, 2003a; Anderson *et al.*, 2007; Callahan *et al.*, 2006; Marchesin *et al.*, 2004). Dans ce cas, la méthode explicite par lancé de rayons est très coûteuse car elle nécessite d'échantillonner un volume dont on ne connaît pas implicitement la géométrie. Qui plus est, cette méthode est difficilement implantable sur un GPU. C'est pourquoi il est fréquent d'utiliser une discrétisation implicite par projection (splatting) qui peut être aisément accélérée matériellement par un GPU, mais qui en pratique produit un rendu de moins bonne qualité que celui par lancé de rayons.

Les performances du rendu volumique peuvent être très bonnes mais uniquement lorsqu'il est possible d'accélérer la méthode à l'aide d'un GPU. Malgré tout, le rendu volumique s'avère très coûteux en terme de calcul et de stockage tout en posant de nombreuses difficultés tant au niveau de la méthode d'échantillonnage que celle d'interpolation. De plus le rendu volumique est strictement limité à de la visualisation. En effet, il ne génère aucune structure géométrique qui aurait pu être utilisée ultérieurement, par exemple, en tant que support de calcul. En outre, travailler sur des maillages non-structurés, qui est l'objectif principal de cette thèse, rend la tâche encore plus ardue.

Méthodes d'*extraction* : algorithmes qui utilisent des liens topologiques et algorithmes à base de tables d'index comme les *Marching Tetrahedra* ou les *Marching Cubes*

Contrairement aux méthodes de rendu volumique qui traitent un volume de données dans son ensemble, les méthodes d'extraction extraient un sous-ensemble du volume qui doit être visualisé. Fréquemment ces méthodes sont dites de *tessellation* ou bien le cas échéant de *triangulation*. Le sous-ensemble sélectionné est une surface polygonale qui par la suite est visualisée à l'aide

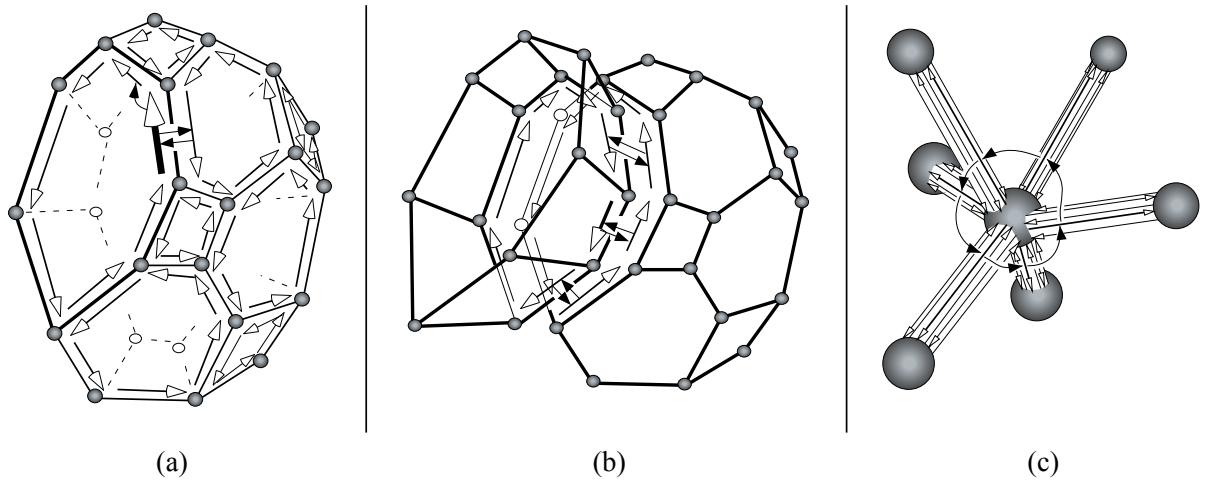


FIG. 1.7 – Exemple de structure de voisinage de type CIEL (Lévy *et al.*, 2001; Caumon *et al.*, 2005). La notion de demi-arête (flêches à têtes blanches) joue un rôle central dans la structure CIEL. Chaque demi-arête possède un pointeur vers son successeur dans le polygone courant, et un pointeur vers une demi-arête dite *jumelle* d'un polygone adjacent (flêches à têtes noires dans (a)). Une demi-arête possède également un pointeur qui permet de retrouver le polyèdre adjacent (flêches à têtes noires dans (b)). De plus, parmi les demi-arêtes sortantes d'un sommet origine, une demi-arête se trouve placée dans une liste circulaire permettant de passer d'un sommet à un autre autour de ce sommet d'origine (flêches à têtes noires dans (c)). Figure d'après Caumon *et al.* (2005).

de techniques de rendu polygonal classiques. Les méthodes d'extraction impliquent donc une étape de pré-traitement du maillage volumique ayant pour but de déterminer le sous-ensemble souhaité.

L'extraction d'une surface d'isovaleur sur des maillages non-structurés peut être réalisée de plusieurs manières différentes :

- Par déplacements suivant des liens topologiques (Bloomenthal, 1988; Silva et Mitchell, 1997; Conreaux, 2001; Lévy *et al.*, 2001; Caumon *et al.*, 2005).
- Par l'algorithme du *Marching Cubes/Tetrahedra* (Lorensen et Cline, 1987; Guéziec et Hummel, 1995).

La première méthode utilise des liens topologiques entre sommets/arêtes et faces des cellules d'un maillage pour en extraire une portion d'isosurface (figure 1.7). Le principe fondamental de la méthode est de tourner autour des faces des cellules et d'en tester les arêtes. Lorsqu'une arête intersectée est rencontrée, alors l'algorithme cesse de tourner autour de la face courante et teste la face qui partage l'arête intersectée. Finalement il continue à tester successivement les arêtes de cette nouvelle face jusqu'à retrouver l'arête par laquelle l'isosurface va ressortir de la face. La figure 1.8 illustre ce processus d'extraction d'une isosurface pour une cellule fortement non-structurée en utilisant la méthode CIEL (Lévy *et al.*, 2001; Caumon *et al.*, 2005).

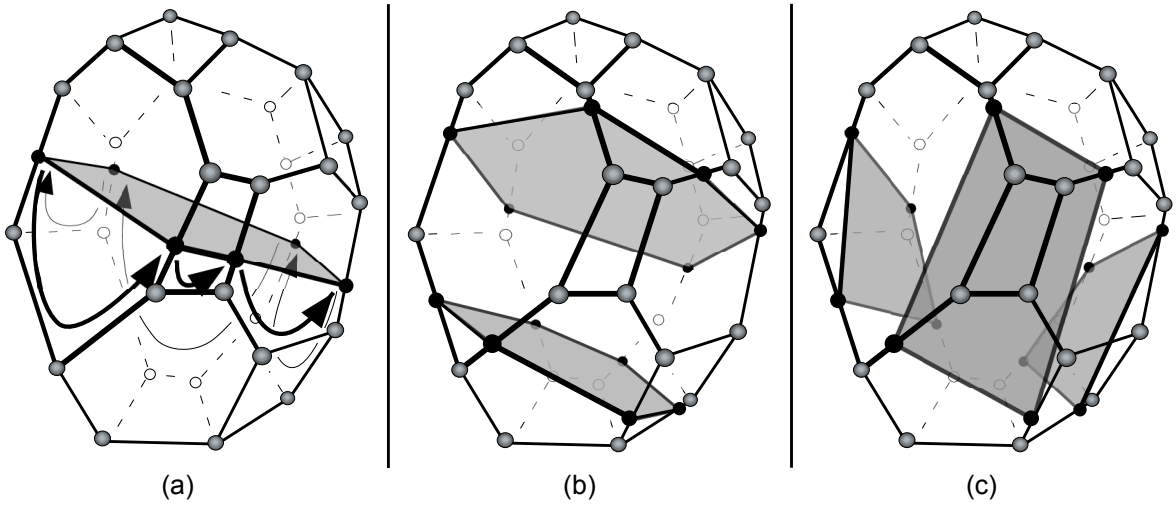


FIG. 1.8 – (a) En partant d'une arête intersectée, l'ensemble des intersections sont retrouvées en "tournant autour" des faces du polyèdre. (b) et (c) sont deux interprétations possibles d'une même configuration ambiguë. Ce type de configuration peut seulement survenir si le champ scalaire est non-monotone au sein de la cellule étudiée (voir la section 1.2.4 pour plus de détails ; figure d'après Caumon *et al.* (2005)).

Cette méthode a l'avantage d'être totalement générique et de pouvoir s'appliquer à n'importe quel type de grilles, structurées ou non. D'un autre côté, la *propagation* suivant des liens topologiques est très coûteuse en temps car elle nécessite de remonter successivement de très nombreux pointeurs mémoire, ce qui rend la méthode pas forcément très rapide. Dans le cas particulier où la topologie de chaque cellule ne change pas au sein d'une même grille, il est possible d'appliquer des algorithmes beaucoup plus spécialisés qui offrent des performances bien supérieures.

Concrètement, par exemple, lorsqu'il est question de grilles à base d'hexaèdres, la méthode des Marching Cubes est la plus efficace connue à ce jour (Lorensen et Cline, 1987). Elle décrit comment extraire la ou les portions d'isosurface(s) contenue(s) dans un hexaèdre à l'aide de deux tables d'index. La première table, appelée *table d'arêtes*, contient les index des sommets associés à chaque arête d'une cellule. Un plus (resp. un moins) est associé à chaque sommet dont la valeur est supérieure (resp. inférieure) à l'isovaleur requise. Ces signes définissent la *configuration* de la cellule, $2^8 = 256$ configurations sont donc possibles. La deuxième table, appelée *table de cas*, contient une entrée pour chaque configuration possible de l'isosurface à extraire, et liste dans un ordre précis les arêtes intersectées pour chaque cas. La méthode des Marching Cubes prend en compte les symétries des configurations afin de réduire le nombre de cas total possibles à seulement 15 cas comme le montre la figure 1.9.

La définition des tables d'arêtes et de cas permettent d'extraire aisément une isosurface d'un hexaèdre :

– La configuration de l'hexaèdre courant détermine un index k dans la table des cas suivant

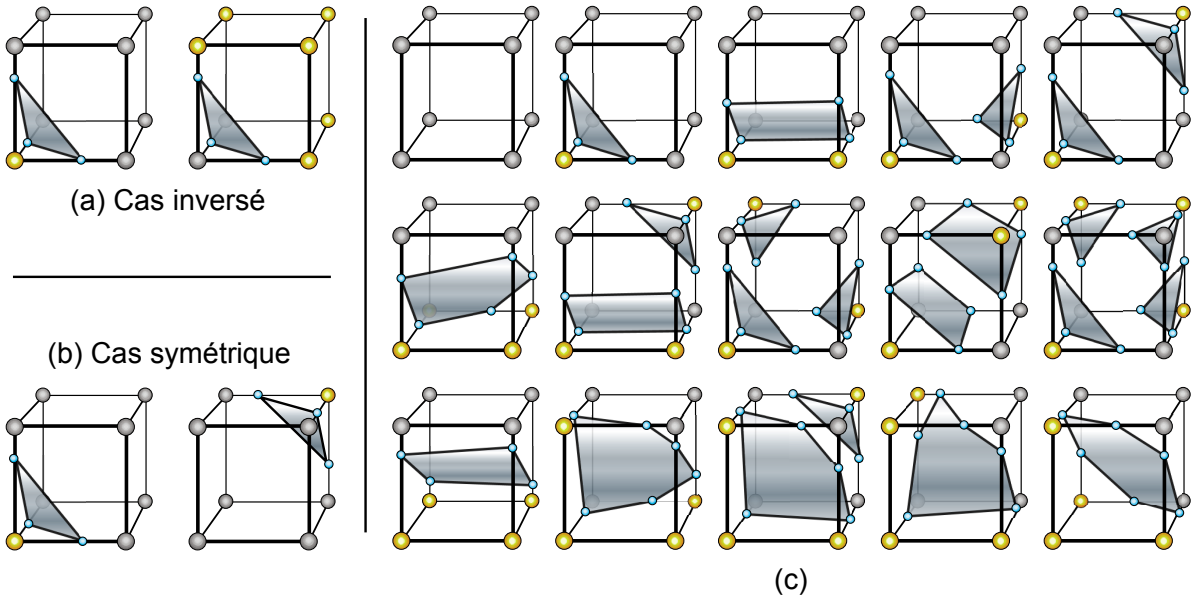


FIG. 1.9 – Le Marching Cubes. Configurations inversées (a) et symétriques (b) qui correspondent à des entrées uniques dans la table des cas. (c) Représentation réduite par symétries des configurations possibles d'un hexaèdre selon l'algorithme du Marching Cubes.

cette équation :

$$k = \sum_{i=0}^{7} (\mathcal{F}(x_i) >= \alpha) * 2^i$$

où $\mathcal{F}(x_i)$ est la valeur associée au sommet x_i et α l'isovaleur requise.

- La ligne d'index k de la table des cas est alors lue. Celle-ci contient la liste des arêtes intersectées pour la configuration k.
- Pour chaque arête intersectée, ses sommets associés sont récupérés à l'aide de la table d'arêtes.
- Pour chaque paire de ces sommets, l'intersection est explicitement calculée par interpolation linéaire de leurs positions. Cette interpolation est calculée en fonction des valeurs des sommets et de l'isovaleur choisie.

L'algorithme du Marching Cubes pose donc les bases d'une extraction indexée des hexaèdres. Cette méthode peut également être appliquée à un autre type de grilles non-structurées à base de tétraèdres, ces dernières ayant une topologie intra-cellulaire constante. Cette méthode est communément appelée *Marching Tetrahedra* par analogie avec les Marching Cubes (Guéziec et Hummel, 1995). L'algorithme de base ne change pas, et seules les tables d'arêtes et de cas sont redéfinies. Le nombre inférieur de sommet des tétraèdres par rapport à un hexaèdre permet d'avoir une table de cas très simplifiée qui ne contient plus que $2^4 = 16$ cas, voire 3 cas si les cas symétriques sont pris en compte (voir figure 1.10).

Notre approche développée à la section 1.2.4 s'inspire à la fois des Marching Cubes (Lorensen et Cline, 1987; Guéziec et Hummel, 1995) et des méthodes qui tournent autour des faces des cellules (Bloomenthal, 1988; Conreaux, 2001; Lévy *et al.*, 2001; Caumon *et al.*, 2005).

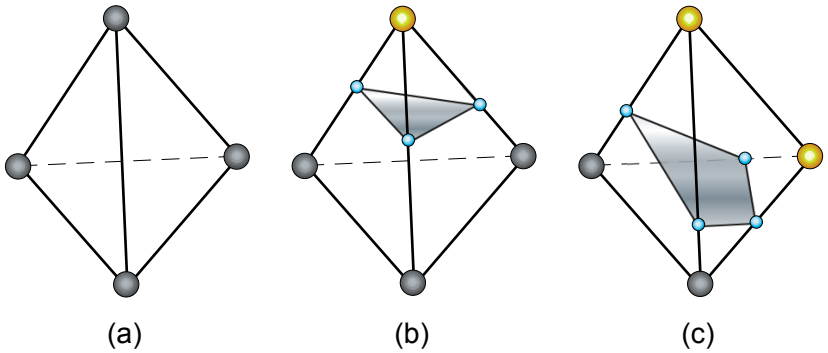


FIG. 1.10 – The Marching Tetrahedra. Configurations dans lesquelles l'isosurface : (a) n'intersecte pas le tétraèdre, (b) intersecte le tétraèdre et trois arêtes sont intersectées, (c) intersecte le tétraèdre et quatre arêtes sont intersectées.

Accélération au moyen d'un GPU des Marching Tetrahedra L'apparition des GPU programmables a ouvert de nouvelles perspectives pour l'extraction d'isosurfaces. Jusqu'à présent, à cause de limitations hardware, les seuls types de grilles qui ont été matériellement accélérées par des GPU pour de l'extraction d'isosurfaces sont des grilles non-structurées tétraédriques (Pascucci, 2004; Reck *et al.*, 2004; Klein *et al.*, 2004; Kipfer et Westermann, 2005). Sur ce type de grille, la méthode d'extraction GPU la plus rapide à ce jour a été proposée par Kipfer et Westermann (2005). Leur méthode exploite un fait singulier : une arête peut être partagée avec de nombreux tétraèdres. En conséquence, tester chaque tétraèdre indépendamment comme dans l'algorithme des Marching Tetrahedra peut introduire une forte redondance dans l'accès aux données et dans les calculs effectués. Pour éliminer cette redondance, Kipfer et Westermann ont utilisé une approche basée sur les arêtes (ou *edge-based* en anglais) dans laquelle chaque arête n'est alors testée qu'une seule et unique fois. L'algorithme nécessite de disposer d'une numérotation spéciale des sommets et des arêtes au sein de chaque tétraèdre qui implique de trier les sommets en ordre croissant de leurs valeurs scalaires associées. Ils utilisent un algorithme sur GPU constitué de trois passes qui utilisent de multiples textures afin de stocker les données nécessaires à leur méthode. Celle-ci s'avère être efficace mais souffre de nombreuses limites. Le coût du tri des sommets peut être très significatif car il impose de réordonner un grand volume de données. De même, les arêtes qui sont partagées entre plusieurs tétraèdres ne sont pas toujours connues ce qui implique de les détecter lors d'une phase de pré-caclul des plus coûteuse en temps. De surcroît, la méthode ne s'applique qu'à de simples tétraèdres. Elle est inapplicable, par exemple, sur des grilles hexaédriques ou bien des grilles fortement non-structurées. Qui plus est, l'implantation sur GPU d'un tel algorithme est beaucoup plus complexe que celui du Marching Tetrahedra.

C'est pour toutes ces raisons que nous avons décidé de baser notre travail sur une autre technique plus générale, mais également très efficace, proposée par Pascucci (2004) et modifiée/adaptée par Reck *et al.* (2004). Cette méthode propose une implantation sur GPU de l'algorithme du Marching Tetrahedra qui, pour rappel, utilise deux tables d'index : une table de cas et une table d'arêtes. L'implantation sur CPU des Marching Tetrahedra ne pose pas de problèmes insurmontables. En revanche, sur GPU, leur nature même rend la chose plus délicate.

Pascucci et Reck et al. utilisent les unités de vertex shader (section 1.2.1) afin d'extraire des isosurfaces sur GPU pour des grilles tétraédriques. Ils envoient au GPU, pour chaque tétraèdre, quatre sommets numérotés puisque la portion d'isosurface la plus complexe qui puisse être extraite d'un tétraèdre est composée au plus de 4 sommets (voir figure 1.10). Avec chacun de ces quatre sommets du futur iso-polygone, sont envoyés au GPU dans des registres variables :

- le numéro du sommet en cours d'envoi (de 0 à 3),
- la position dans l'espace des quatre sommets du tétraèdre traité,
- les quatre valeurs scalaires associées.

Les tables de cas et d'arêtes sont, quant à elles, stockées dans des registres constants pour une question d'efficacité. Pour chaque tétraèdre, quatre sommets numérotés entre 0 et 3 sont donc envoyés. Pour chacun de ces sommets envoyé aux unités de vertex shader d'un GPU, le Marching Tetrahedra est appliqué :

- calcul de l'index de la configuration du tétraèdre courant,
- lecture dans la table des cas de la liste des arêtes intersectées,
- pour l'arête intersectée portant le numéro requis (entre 0 et 3), lecture dans la table des arêtes des index des sommets associés,
- calcul par interpolation linéaire de la position du sommet requis,
- finalement déplacement du sommet envoyé au processeur graphique à cette position.

Au final, 4 sommets sont envoyés à la carte graphique, et chacun d'entre eux est déplacé à un coin de l'isosurface qui intersecte le tétraèdre. Lorsque la portion d'isosurface ne compte que trois sommets (figure 1.10-(b)), le sommet supplémentaire envoyé est empilé à la même position que le dernier calculé, et est ainsi ignoré par la carte graphique. Si l'isosurface n'intersecte pas le tétraèdre traité (figure 1.10-(a)), alors tous les sommets sont empilés au même endroit, et le polygone correspondant, dès lors homogène à un point, est ignoré par la carte graphique.

L'implantation de Pascucci est très performante, mais s'applique surtout à des grilles d'une taille moyenne car elle introduit une forte redondance dans le stockage des données. En effet, pour chaque tétraèdre à traiter, l'ensemble des données le décrivant sont envoyées au processeur graphique. La méthode ignore donc le fait que la majorité des sommets d'une grille tétraédrique sont partagés entre plusieurs tétraèdres.

Notez qu'il est possible de combiner l'algorithme des Marching Tetrahedra implanté par Pascucci avec des algorithmes de classification (*Octree*, *Interval Tree*, etc.) qui n'envoient au GPU que la liste des index des cellules dont il est certain qu'elles soient intersectées (voir section 1.3 pour plus de détails sur les algorithmes de classification).

Calcul de l'éclairage à partir des normales à la surface Les modèles d'éclairage un tant soit peu réalistes d'un point de vue visuel nécessitent systématiquement de disposer d'un vecteur normal à la surface rendue. Ce vecteur normal peut être calculé pour chaque face discrète d'un polyèdre (on parle alors de rendu *plat*), ou bien pour chaque sommet de ce polyèdre de manière à moyenner les vecteurs normaux à chacune des faces adjacentes au sommet considéré (on parle alors de rendu *doux*). La figure 1.11 illustre la différence de rendu entre les méthodes plate (a) et douce (c). La normale en chaque point d'une face d'un polyèdre est alors calculée par interpolation linéaire à partir des normales aux sommets du polyèdre. Les cartes graphiques prennent en charge nativement cette interpolation.

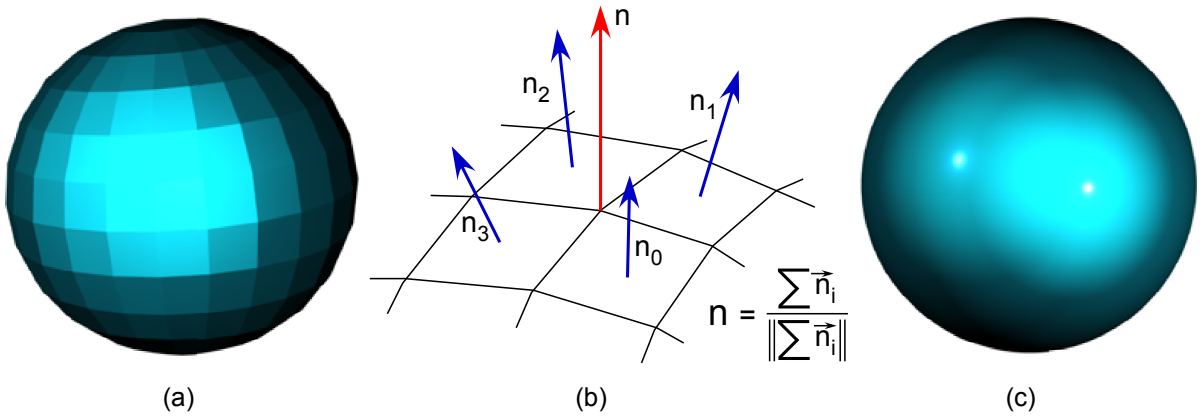


FIG. 1.11 – (a) Rendu plat, les normales sont constantes pour chaque face. (c) Rendu doux, les normales sont calculées par sommet comme la somme normalisée des normales de toutes les faces adjacentes au sommet considéré (b), puis interpolées en chaque point de la face. Schéma inspiré de Frank (2006).

Les normales peuvent être obtenues à partir du gradient de la fonction $\mathcal{F}$ qui définit le champ scalaire dans le polyèdre considéré, et ce par simple normalisation du gradient. Ces calculs peuvent être effectués à deux moments différents dans le cadre de l'extraction d'isosurfaces : sous forme d'un pré-calcul réalisé une seule et unique fois, ou en temps-réel lors de l'extraction de chaque morceau d'isosurface. Nous avons choisi la solution la plus efficace, c'est-à-dire la première solution : calcul unique des normales lors de la phase de chargement des données de la grille considérée. De cette manière il est possible d'effectuer un rendu doux tout en évitant de re-calculer des normales inutilement. Notez que tous les tests présentés dans ce mémoire ont été réalisés avec un calcul d'éclairage activé.

1.2.3 Contributions

L'extraction d'isosurfaces sur GPU à partir de grilles fortement non-structurées est très problématique. Dans la littérature, ce problème est souvent résolu en pré-*tétraédrisant* l'ensemble des cellules de la grille considérée, c'est-à-dire en subdivisant toutes ses cellules en tétraèdres. Une fois la grille transformée en tétraèdres, des techniques d'extraction plus classiques peuvent être employées. La méthode souffre néanmoins de plusieurs points faibles. Tout d'abord, elle impose une phase de pré-traitement pendant laquelle les cellules vont être tétraédrisées. Cette phase est lourde, pose de nombreux problèmes algorithmiques complexes et le choix de la manière de tétraédriser –et donc d'interpoler les champs scalaires– peut avoir un impact significatif sur le résultat obtenu. Au final, la tétraédrisation augmente considérablement, artificiellement et inutilement le nombre de cellules à traiter. Tous ces traitements ont un impact considérable sur la consommation mémoire et sur les performances d'une telle méthode. C'est pourquoi nous avons développé une méthode d'extraction qui traite directement les cellules, même fortement non-structurées, en étendant l'algorithme du Marching Cubes, l'algorithme sur GPU de Pascucci (2004), et les algorithmes qui utilisent des liens topologiques pour générer des isosurfaces.

L'algorithme sur GPU de Pascucci utilise les unités de vertex shader pour extraire une isosurface et des registres variables et constants pour envoyer la description complète de chaque tétraèdre ainsi que les tables de cas et d'arêtes. La révision 3.0 du standard Shader Model prévoit que l'unité de plaquage de textures puisse être utilisée depuis les unités de vertex shader, et ce contrairement aux Shader Model 2.0 qui n'autorisaient un accès que depuis les unités de pixel shader. Il est alors possible d'utiliser des textures en lieu et place de certains registres, dont le nombre limité interdisait certaines extensions à l'algorithme de Pascucci. Ce nombre de registres restreint interdit par exemple de stocker de larges tables, comme la table des cas correspondant à un hexaèdre. *In fine*, cela empêche d'utiliser une carte graphique pour traiter ce type de cellules.

Notre approche propose donc d'améliorer de nombreux points par rapport aux approches précédentes en permettant :

- d'éviter le stockage de données redondantes en tenant compte des sommets partagés,
- de limiter les transferts sur le bus PCI-Express,
- en théorie, d'accélérer sur GPU l'extraction d'une isosurface pour n'importe quel type de cellules (les précédentes méthodes se limitaient aux cellules tétraédriques),
- de générer automatiquement les tables de cas et d'arêtes pour n'importe quel type de cellule,
- en théorie, de réduire le nombre de tables de cas et d'arêtes à stocker grâce aux isomorphismes de cellules,
- de stocker efficacement l'ensemble des données (grille et tables d'index) dans des textures dans le but d'améliorer les performances d'extraction sur de grandes grilles,
- de traiter des grilles non-structurées constituées de millions de cellules, contrairement aux approches précédentes,
- supporter à la fois une extraction en force brute (l'ensemble des cellules de la grille sont extraites) et la combinaison avec un algorithme de classification des cellules intersectées (section 1.3).

La section suivante présente une description théorique et pratique de notre méthode générale qui autorise une extraction d'isosurfaces sur GPU pour tout polyèdre. Dans un premier temps nous décrivons comment généraliser l'algorithme du Marching Cubes pour tout type de cellules, puis comment implanter un tel algorithme sur un GPU. Par analogie nous avons nommé notre algorithme le *Marching Cells*.

Puisque notre méthode utilise des textures au niveau du vertex shader, celle-ci implique d'utiliser une carte compatible ou supérieure aux Shader Model 3.0, soit, chez NVIDIA, une carte de 6$^{\text{ème}}$ génération au minimum.

1.2.4 Notre nouvelle approche générique : *Le Marching Cells*

Les méthodes qui utilisent des liens topologiques pour extraire des isosurfaces permettent de traiter tout type de cellules (Bloomenthal, 1988; Conreaux, 2001; Lévy *et al.*, 2001; Caumon *et al.*, 2005). En revanche, leurs performances laissent à désirer. Inversement, les méthodes indexées, telles que celles du Marching Cubes ou Marching Tetrahedra (Lorensen et Cline, 1987; Guéziec et Hummel, 1995), offrent des performances de premier plan, mais se limitent à des grilles

hexaédriques ou tétraédriques. Notre approche, le Marching Cells, s'inspire à la fois des méthodes qui utilisent des liens topologiques et des Marching Cubes afin de combiner les avantages des deux approches : généricité et hautes performances. De plus, notre méthode s'inspire dans un deuxième temps de celle de Pascucci (2004) pour ce qui est de l'implantation sur GPU.

Notre algorithme est séparé en deux phases : une phase de pré-calcul et une phase temps-réel. La première phase prend principalement en charge la génération des tables d'index nécessaires au bon fonctionnement de la seconde phase d'extraction à proprement parler. La seconde phase utilise un algorithme similaire à celui du Marching Cubes mais qui s'appuie sur les tables d'index nouvellement générées.

Phase de pré-calcul

Création des tables de cas et d'arêtes La généralisation de la méthode du Marching Cubes et de celle de Pascucci (2004) pour des grilles fortement non-structurées nécessite de développer un générateur automatique de tables de cas et d'arêtes. Ce générateur automatique doit partir d'une structure de voisinage qui fournisse les notions de cellule, face, arête orientée, sommet... comme le proposent les structures de type CIEL (Lévy *et al.*, 2001; Caumon *et al.*, 2005) ou bien DCEL (Muller et Preparata, 1978; Preparata et Shamos, 1985). La figure 1.7 illustre le fonctionnement d'une structure de grille de type CIEL.

Plus précisément, dans notre approche, chaque type de cellule est défini par sa propre structure topologique, représentée par une table de demi-arêtes (issue d'une structure CIEL simplifiée) comme celle présentée en haut de la figure 1.12 pour un tétraèdre.

La génération de la table d'arête à partir d'une structure de type CIEL requiert simplement de compresser cette dernière pour ne conserver que l'information qui nous est utile : pour chaque arête, nous ne conservons que l'index de ses deux sommets associés (bas de la figure 1.12).

Afin de générer la table de cas, une première table qui contient l'ensemble des configurations possibles est tout d'abord créée (table de gauche de la figure 1.13). Puis, pour chaque entrée de cette table qui contient l'état (plus ou moins relativement à l'isovaleur) de chacun des sommets du type de cellule considéré, la liste des arêtes intersectées est établie (table de droite de la figure 1.13). Ceci est réalisé en tournant autour des faces afin de suivre les arêtes intersectées tout autour de la cellule traitée comme sur l'exemple de la figure 1.8. Ce parcours se fait grâce aux informations topologiques fournies par la structure de type CIEL (figure 1.7).

Au final, l'algorithme permet d'obtenir une *liste de composantes* pour chaque configuration. Nous parlons de *liste de composantes* puisque chaque configuration peut être composée de plusieurs portions d'isosurfaces, et donc plusieurs composantes. Nous proposons un modèle simple permettant de stocker cette liste dans la table de cas en faisant précéder chaque composante du nombre d'arêtes la constituant. La figure 1.14 illustre le remplissage d'une entrée dans la table de cas pour un hexaèdre.

L'exemple de la figure 1.13 souligne que la table des cas est en fait symétrique. Ceci est évident puisqu'échanger tous les signes d'une configuration produit une configuration identique. La table des cas peut donc être réduite de moitié en exploitant cette simple symétrie, passant de 2^n lignes à 2^{n-1} lignes, avec n le nombre de sommets composant la cellule considérée. Le calcul

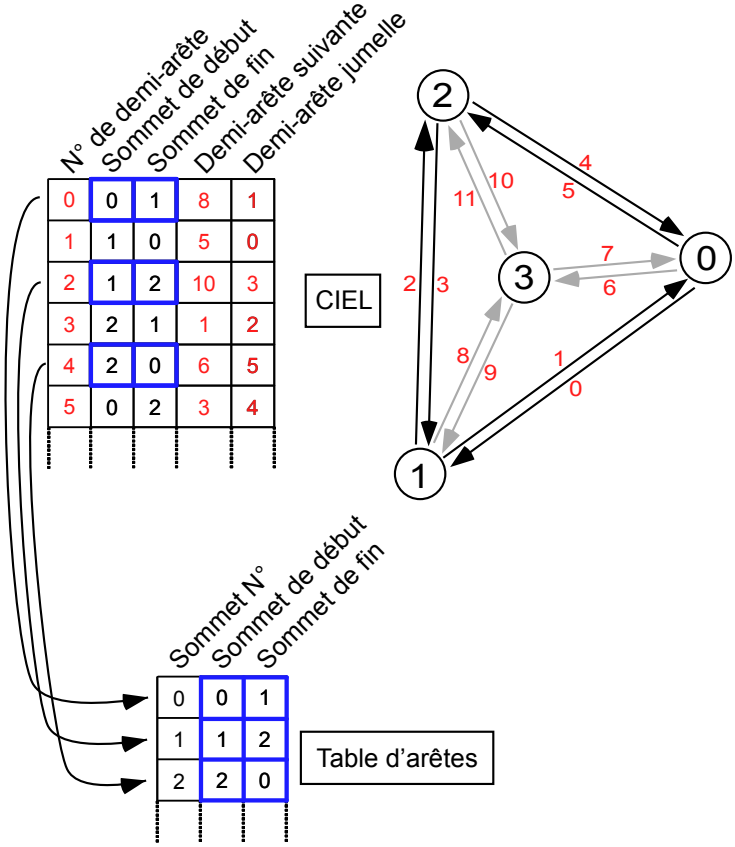


FIG. 1.12 – Structure en demi-arêtes de type CIEL pour un tétraèdre (en haut) et la table d'arêtes qui lui correspond (en bas).

de l'index dans la table des cas doit donc être légèrement modifié afin de suivre cette nouvelle formule :

$$index = \begin{cases} (2^n - 1) - index & \text{si} \quad index >= 2^{n-1} \\ index & \text{sinon} \end{cases}$$

L'exploitation d'autres symétries est un exercice très difficile qui n'a pas fait l'objet de recherches particulières durant cette thèse. Ce sujet ne sera, par conséquent, pas abordé plus en détails dans ce mémoire.

Notre algorithme générique permet de calculer automatiquement pour tout type de cellules une table de cas et une table d'arêtes à partir d'une structure CIEL. Il devient donc possible, à partir de ces tables, d'extraire une isosurface de tout type de cellules par application d'un algorithme similaire à celui du Marching Cubes.

Isomorphismes de cellules, ou comment réduire le nombre de tables d'index requises
Pour des grilles non-structurées hétérogènes (figure 1), en théorie, une table de cas et une table

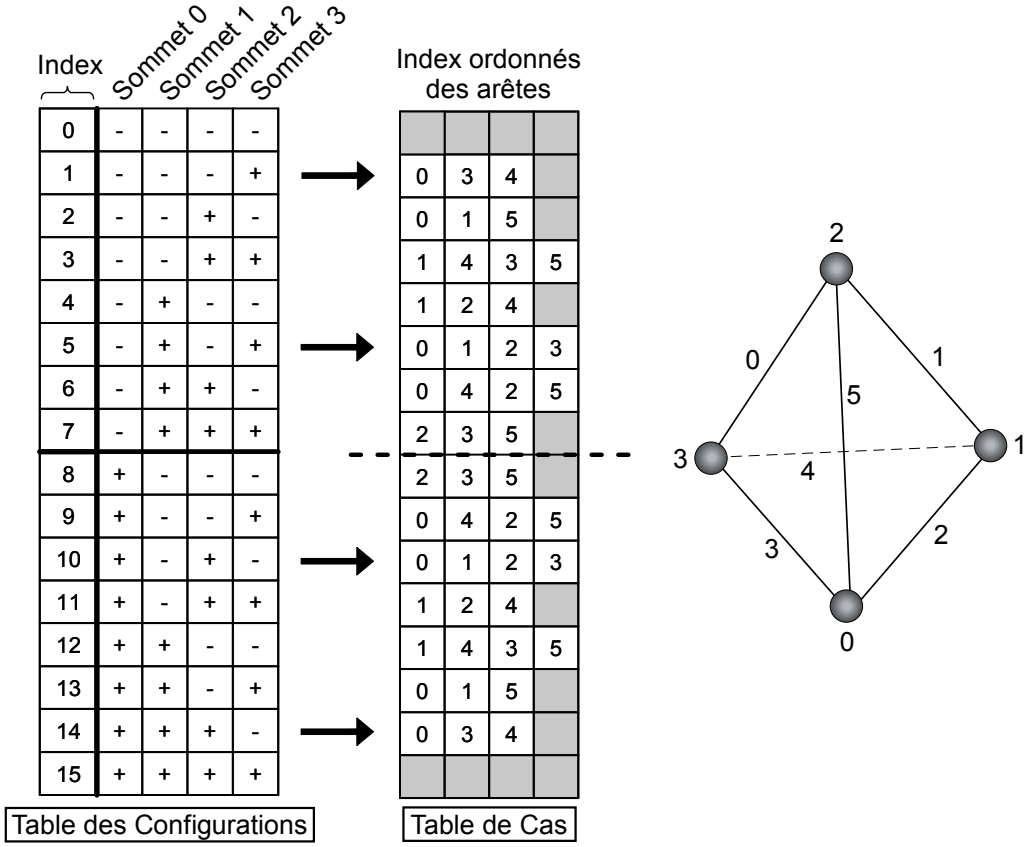


FIG. 1.13 – Table des configurations possibles et table de cas simplifiée pour un tétraèdre suivant le schéma de numérotation de droite.

d'arêtes doivent être stockées pour chaque cellule puisque potentiellement chaque cellule peut avoir une topologie intrinsèque différente des autres cellules. Cela pourrait mener à stocker une très grande quantité d'informations du fait de la taille potentiellement grande des tables à stocker. Prenons un exemple simple, celui d'une cellule hexaédrique. Si nous ne tenons pas compte de symétries plus complexes que celle présentée ci-dessus, une cellule hexaédrique génère une table d'arêtes contenant 12 entrées de deux index et une table de cas contenant $2^{8-1} = 128$ entrées correspondant à un total de 2048 index. Au minimum, chaque index doit être stocké sur 32 bits, soit 4 octets. Au total, se sont pas moins de $(2048 + 2 * 12) * 4 = 8.3Ko$ qui sont requis pour le stockage des tables de cas et d'arêtes pour une grille à maille hexaédrique. Le paragraphe suivant explique comment il est possible de réduire cette consommation mémoire, pour des grilles non-structurées hétérogènes, grâce à une détection de cellules isomorphes.

En pratique, les cellules rencontrées ont une complexité souvent limitée à un petit nombre de sommets, et par conséquent un petit nombre de types de cellules différents. De plus, si les topologies des cellules sont considérées comme des graphes, de nombreux graphes isomorphes peuvent être trouvés dans une grille. Lors de la phase temps-réel de notre algorithme du Marching Cells, pour chaque cellule, une liste ordonnée d'index pointant dans une texture est envoyé

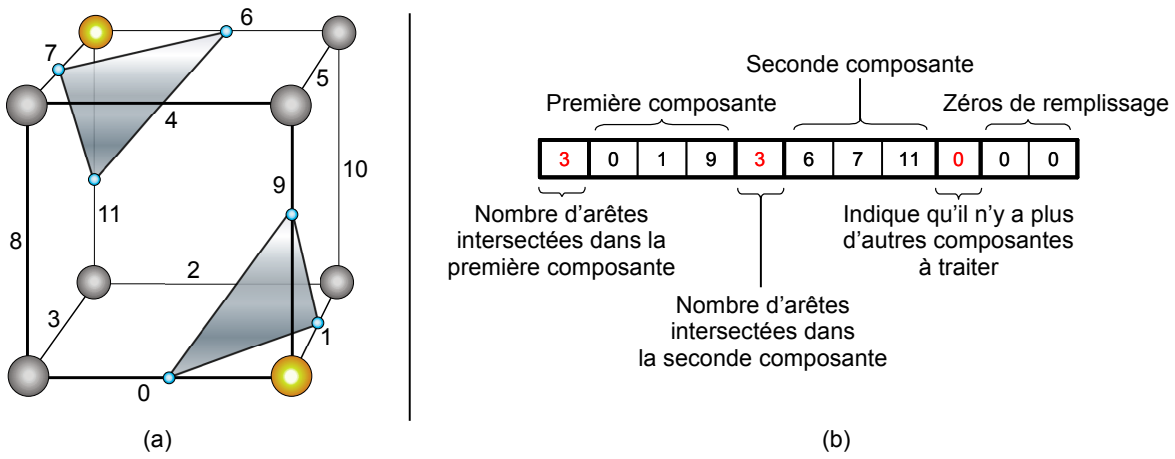


FIG. 1.14 – (a) Hexaèdre intersecté par une isosurface constituée de deux composantes (les numéros fixent un schéma d'indexation des arêtes). (b) Entrée dans la table des cas correspondant à la configuration (a).

au GPU. Cette liste permet de retrouver la géométrie et les valeurs scalaires associées à la cellule traitée. Cette liste ordonnée est au coeur de notre idée. Si deux cellules isomorphes sont considérées, il est alors possible, par définition, de réordonner la liste d'index de la première cellule pour la faire correspondre avec la liste de la deuxième cellule. La détection de ces isomorphismes peut aider à factoriser/regrouper les cellules en classes isomorphes. Au final, une seule table de cas et d'arêtes sont stockées pour chaque classe de cellules.

En apparence, le point le plus négatif de la méthode, est que détecter les isomorphismes entre cellules est un problème connu pour avoir une complexité exponentielle au regard du nombre de noeuds dans le graphe (McKay, 1981), en clair, au regard du nombre de sommets dans la cellule. Mais, puisqu'il est admis que la complexité des cellules traitées reste raisonnable, n'atteignant jamais des centaines de noeuds, détecter des isomorphismes n'est pas réellement problématique. En outre, cette détection est effectuée une seule et unique fois lors de la phase de pré-calcul. Par conséquent, cela n'impacte pas les performances de la phase temps-réel décrite plus loin dans ce mémoire. Le lecteur pourra se reporter à l'excellent article de McKay (1981) qui détaille les algorithmes classiques de détection d'isomorphismes de graphes.

Remarquez que dans le cas particulier des grilles prismatiques, une simple classification en fonction du nombre de sommets à la base de chaque prisme permet d'établir les classes d'isomorphismes.

Ambiguïtés lors de l'extraction d'une isosurface L'algorithme du Marching Cubes peut engendrer des artefacts lors de l'extraction d'une isosurface. En effet, dans certains cas, l'isosurface extraite est morcelée en plusieurs composantes dont la construction peut être ambiguë. La figure 1.15 illustre deux exemples de configurations ambiguës, en 2D et en 3D.

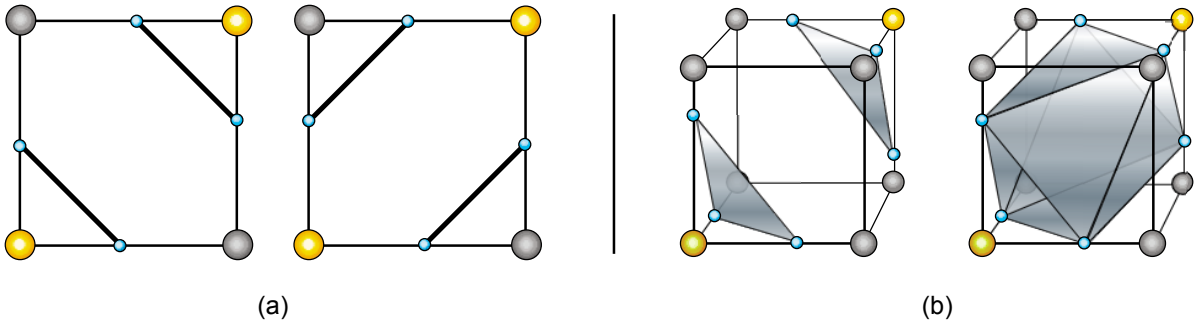


FIG. 1.15 – Exemples 2D (a) et 3D (b) de configurations ambiguës pour lesquelles plusieurs triangulations sont possibles.

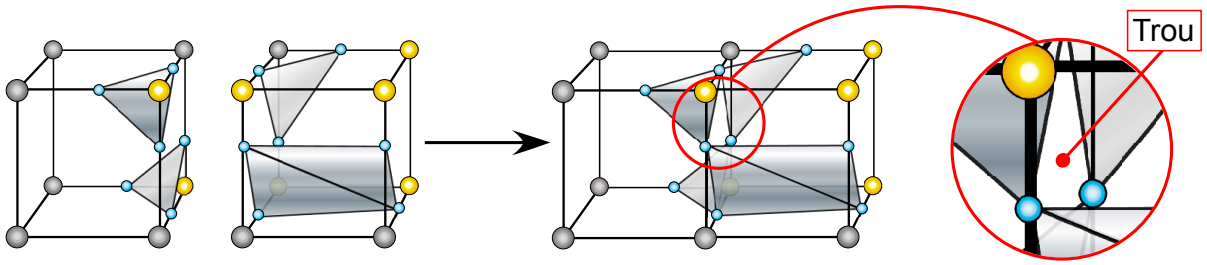


FIG. 1.16 – Exemple d'isosurface extraite à partir d'hexaèdres adjacents suivant l'algorithme du Marching Cubes classique. Les ambiguïtés non prisent en compte lors de la triangulation des deux cellules peuvent générer un trou dans l'isosurface extraite. Ce trou s'étend de part et d'autre de la face commune aux deux cellules.

Du fait de ces ambiguïtés intra-cellulaires, des ambiguïtés inter-cellulaires peuvent survenir (Fuchs *et al.*, 1977) et générer des *trous* dans l'isosurface extraite comme le montre la figure 1.16.

Les ambiguïtés rencontrées peuvent être résolues en comparant les faces adjacentes des cellules, ce qui en pratique n'est absolument pas trivial. De nombreuses méthodes ont été développées afin de pallier à ces ambiguïtés (Wilhelms et Van Gelder, 1992; Van Gelder et Wilhelms, 1994; Nielson et Hamann, 1991; Natarajan, 1994). Certaines utilisent des fonctions d'interpolation bi ou tri-linéaires qui permettent de choisir comment trianguler les cellules ambiguës. D'autres proposent de subdiviser l'ensemble des hexaèdres en tétraèdres pour lesquels les configurations ambiguës n'existent pas. Montani *et al.* (1994) et Chernyaev (1995) ont proposé d'étendre les tables d'index de l'algorithme du Marching Cubes pour que ces dernières prennent en compte implicitement les cas ambigus. Plus particulièrement, Chernyaev (1995) propose d'indexer 33 configurations au lieu des 15 d'un Marching Cubes classique. Dans tous les cas, ces méthodes ont un impact significatif sur les performances d'extraction d'isosurfaces. De surcroît, les cas

ambigus ne peuvent survenir que si le champ scalaire est non-monotone au sein d'une cellule. La prise en compte des cas ambigus sort donc de notre cadre général d'étude dans le sens où les champs scalaires que nous étudions sont majoritairement issus de simulations d'écoulement fluide –simulations qui produisent des champs scalaires majoritairement monotones au sein des cellules. Ce sont ces multiples raisons (performances et faible fréquence) qui nous ont poussé à nous concentrer sur d'autres axes de recherche plus importants à nos yeux.

Stockage de données adapté à un GPU Le stockage des données, par exemple les tables d'index de cas et d'arêtes, est un problème fondamental sur un GPU. Les approches précédentes (Pascucci, 2004; Reck *et al.*, 2004) envoient à l'aide de registres GPU les tables de cas et d'arêtes, et, pour chaque cellule, la position et les valeurs scalaires de ses sommets. Ces registres physiques sont fortement limités en nombre[8], même sur des GPU récents, et ne permettent pas de stocker les tables d'index qui correspondent à des cellules plus complexes que de simples tétraèdres. De surcroît, la plupart de ces approches envoient pour chaque cellule toute l'information qui la décrit, négligeant le partage potentiel des sommets entre de nombreuses cellules. Cela revient à consommer une grande quantité de mémoire graphique inutilement. Sur un maillage tétraédrique, Reck *et al.* (2004) constatent que leur approche permet de traiter des grilles composées seulement d'environ 200k tétraèdres avec une carte disposant de 128Mo de RAM graphique. L'utilisation de textures pour stocker l'ensemble des données nécessaires à l'extraction d'une isosurface permet de repousser les limites énoncées ci-dessus. En effet, ce type de stockage permet à la fois de stocker de grandes grilles (grâce à l'élimination de la redondance de stockage des sommets) et de grandes tables d'index (ce qui autorise l'extraction d'isosurfaces sur des maillages fortement non-structurés). Comme le montrera la section 1.2.5, l'utilisation de textures comme support de stockage permet de traiter en pratique des grilles de plusieurs millions de cellules.

Notre finalité est de stocker l'ensemble des données nécessaires dans une texture. Un *texel* est le plus petit élément d'une texture (de l'anglais *Texture Element*) pouvant contenir de une à quatre composantes qui correspondent aux trois couleurs primaires Rouge, Vert et Bleu (RVB) et à une valeur d'opacité, couramment appelée Alpha (A). En pratique, la mémoire utilisée pour le stockage d'un texel est constante quel que soit le nombre de composantes de celui-ci. En effet, un texel qui ne contient qu'une seule composante est physiquement stocké dans un texel en contenant quatre.

Quatre index peuvent être stockés dans chaque texel d'une texture, un dans chaque composante RVBA. De cette manière, l'accès à une valeur d'index particulière dans la table de cas ou la table des arêtes requiert de lire le texel correspondant lors de la phase d'extraction temps-réel. Il est possible d'utiliser une telle stratégie de stockage qui minimise l'espace mémoire nécessaire, ou bien de l'échanger, pour une plus grande facilité d'accès aux index, avec une méthode moins compacte qui ne stocke qu'un seul index par texel.

Puisque la table des arêtes est naturellement rectangulaire (figure 1.12), elle peut être dé-

[8]Sur les dernières générations de cartes graphiques actuellement sur le marché le nombre de registres est virtuellement très grand (plusieurs dizaines de milliers de registres), mais, au mieux, seuls les quelques centaines de premiers registres, qui correspondent à des registres physiques, offrent des performances acceptables.

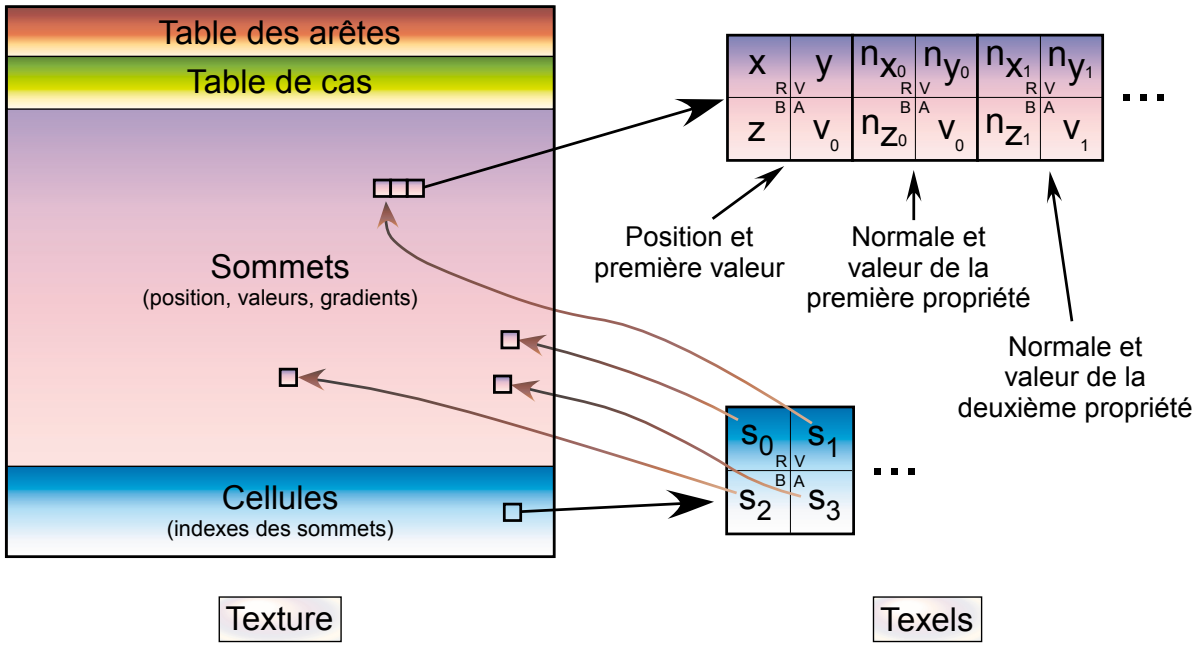


FIG. 1.17 – Stratégie de stockage générique dans une texture des tables d'arêtes et de cas, des sommets (position, valeurs et gradients) et pour chaque cellule, la liste des index (dans la texture) des sommets qui lui sont associés. Plusieurs champs scalaire différents peuvent être attachés à une même grille, c'est pourquoi il est prévu que plusieurs valeurs et gradients puissent être stockés dans la texture pour chaque sommet.

roulée en une table 1D qui est stockée séquentiellement, ligne par ligne, dans une texture 2D[9]. Chaque cellule du maillage conserve l'index dans la texture qui lui permet d'aller y lire la taille de la table d'arêtes ainsi que son contenu. La table de cas est elle aussi stockée séquentiellement dans une texture, juste après la table des arêtes. La table de cas n'est pas naturellement rectangulaire (figure 1.13), mais est transformée en une table rectangulaire en complétant les lignes les plus courtes avec des zéros (figure 1.14). Pour chaque type de cellule, la largeur de cette table doit être conservée afin d'être en mesure d'accéder directement à chacune de ces lignes.

Les isosurfaces constituées de plusieurs composantes –au sein d'une seule cellule– sont prises en charge en stockant :

- max_s, le nombre maximum d'arêtes intersectées (et donc le nombre de sommets) par le plus *long* iso-polygone possible,
- max_c, le nombre maximum de composantes que peut contenir une cellule.

Ces paramètres déterminent combien de sommets doivent être envoyés au GPU lors de la phase de rendu temps-réel. Par exemple, seulement 4 sommets sont requis pour un simple tétraèdre (Pascucci, 2004). La figure 1.17 illustre notre stratégie générique de stockage dans une texture.

[9]Comparé aux textures 1D, les textures 2D offrent de bien meilleures performances d'accès ainsi qu'une bien plus grande capacité de stockage. C'est pourquoi nous utilisons des textures 2D comme moyen de stockage.

Une fois les tables d'index stockées dans la texture, les données décrivant la grille peuvent y être, à leur tour, stockées. Tous les sommets de la grille sont successivement parcourus et les informations les décrivant sont stockées dans la texture. La position spatiale en 3D de chaque sommet est stockée dans un texel, ce qui utilise trois des quatre composantes de ce dernier. La quatrième et dernière composante est utilisée pour stocker la valeur de la première propriété afin de pouvoir éventuellement optimiser l'accès aux données. En effet, si aucun calcul d'éclairage n'est requis, un seul accès à la texture permet alors d'obtenir à la fois la position et la valeur associée à un sommet. Cela est suffisant pour extraire une isosurface. Des texels supplémentaires peuvent être utilisés pour stocker les valeurs et les gradients associés à chaque champ scalaire lié à la grille. Le stockage d'un gradient utilise trois des quatre composantes d'un texel. La composante restante est utilisée pour stocker la valeur scalaire correspondante.

Lors de la phase temps-réel, pour chaque cellule à traiter, l'index du premier texel qui stocke la liste des index des sommets de cette même cellule est envoyé au GPU à l'aide de registres, ainsi que le nombre total de sommets de la cellule. Cette stratégie permet d'extraire, en théorie, des isosurfaces de cellules comptant un nombre arbitraire de sommets, nombre potentiellement très grand, tout en limitant les transferts sur le bus graphique à un seul et unique index par cellule.

Les cartes graphiques modernes possèdent en général 32 registres vectoriels à quatre composantes, ce qui permet d'envoyer un minium de 128 index. Grâce à ces registres, il est possible d'envoyer, pour chaque cellule, directement la liste des index des sommets qui lui sont associés sans passer par la texture. Cette autre stratégie permet d'accroître en général les performances de l'algorithme d'extraction en évitant des accès consécutifs dépendants dans la texture (récupération des index des sommets dans la texture puis lecture aux index obtenus). En revanche, cette stratégie augmente la quantité de données qui doit être transférée à la carte graphique, ce qui, dans certains cas, peut contre-balancer le gain précédent. Malgré tout, cette stratégie alternative qui utilise plus de registres permet en théorie d'extraire des isosurfaces de cellules très complexes pouvant compter une centaine de sommets.

Remarques

- Dans le cas d'une grille tétraédrique, le stockage de chaque sommet requiert au minimum un texel (position 3D et une valeur) et le stockage de chaque tétraèdre un autre texel (un index pour chacun de ses sommets). En pratique, nous avons pu traiter une grille de presque deux millions de tétraèdres avec seulement 128Mo de RAM graphique. Ce chiffre est à comparer aux deux cents mille tétraèdres traités avec la méthode de Reck *et al.* (2004).
- Dans le cas particulier des grilles homogènes ou structurées (figure 1), c'est-à-dire des grilles ne contenant qu'un seul type de cellules présentant une même topologie, une seule table de cas et d'arêtes sont nécessaires et par conséquent stockées pour l'ensemble des cellules. Cette remarque permet de minimiser la consommation mémoire due au stockage des tables d'index liées à notre algorithme du Marching Cells.

Phase temps-réel

Cette section décrit comment les données générées à l'étape de pré-calcul précédente sont utilisées lors de la phase temps-réel pour extraire une isosurface d'une grille sur un GPU.

Algorithme 1 : Vue d'ensemble de la phase temps-réel (algorithme brute-force)

Initialisation (Étape 1)

- Charger la texture en mémoire graphique.
- Charger le programme de vertex shader.
- Choisir l'isovaleur.

Itérer pour chaque cellule (Étape 2)

- Envoyer l'index de la cellule dans la texture au GPU.
- Envoyer l'index du premier élément de la table des arêtes.
- Envoyer le nombre de lignes total de la table des arêtes.
- Envoyer le nombre d'index par ligne de la table de cas.
- Envoyer les sommets et les numéros des composantes souhaitées au GPU, et implicitement exécuter le vertex shader associé.

Mise-à-jour (Étape 3)

- Mettre-à-jour l'isovaleur et aller à l'Étape 2.

Gestion des données Une fois la texture contenant les données chargée en mémoire graphique, des données supplémentaires peuvent être envoyées à l'aide des registres GPU variables et constants. Un registre constant est utilisé pour envoyer l'isovaleur, tandis que des registres variables sont employés pour envoyer les index des cellules dans la texture, les index de la racine de chaque table d'arêtes etc.

En OpenGL, les registres ont la particularité d'être persistants en mémoire, c'est-à-dire que leur valeur ne change pas tant que l'utilisateur ne le demande pas explicitement. Cette propriété peut être exploitée afin de limiter la quantité de données à envoyer au GPU en regroupant les cellules qui utilisent les mêmes données. Par exemple, dans le cas d'une grille homogène ou bien structurée, les mêmes tables de cas et d'arêtes peuvent être utilisées pour l'ensemble des cellules de la grille. Les paramètres tels que l'index du premier élément de la table des arêtes, le nombre de ligne de celle-ci ou bien encore le nombre d'index par ligne de la table de cas seront envoyés une seule et unique fois au GPU dans des registres, dont la valeur persistera pour chaque cellule traitée.

Prise en charge d'un nombre variable de sommets Pour supporter le nombre variable de sommets de cellules de types différents, plusieurs tables d'arêtes et de cas doivent être calculées, stockées puis lues. Les paragraphes précédents ont montré comment générer automatiquement ces tables et les stocker dans une texture. Nous avions indiqué à ce moment là que deux chiffres

étaient également conservés : le nombre maximum de sommets contenus dans un iso-polygone max_s, et le nombre maximum de composantes max_c d'une isosurface dans une seule cellule (pour un tétraèdre, $max_s = 4$ et $max_c = 1$, pour un hexaèdre, $max_s = 6$ et $max_c = 4$, etc). Au total, $max_s * max_c$ sommets doivent être envoyés à la carte graphique pour que celle-ci puisse prendre en charge l'extraction complète d'une isosurface. Pour chaque sommet s traité d'une iso-composante c, l'algorithme parcourt la table des cas jusqu'à ce que la composante c soit trouvée. Si la composante c requise n'existe pas, le sommet est ignoré. Dans le cas contraire, le sommet s est considéré uniquement s'il existe dans la composante c (si c est constitué de 3 sommets et le sommet est numéroté 4 ($s = 4$), il est ignoré). Le cas échéant, l'index de l'arête intersectée est lu dans la table de cas pour la composante c et le sommet s. Cette méthode de parcours peut être implantée dans un vertex shader chargé en mémoire d'une carte graphique. Les paragraphes suivants montrent comment gérer, en pratique, l'étape temps-réel de notre algorithme sur GPU.

Envoi des sommets à un GPU Les numéros de la composante c et du sommet s sont envoyés simultanément au GPU grâce à l'argument de position de la fonction de création de sommets en OpenGL :

Algorithme 2 : Envoi des sommets

```
Pour ( c = 0 ; c < max_c ; c++ ) {
  glBegin(GL_POLYGON) ;
    Pour ( s = 0 ; s < max_s ; s++ ) {
      glVertex2s(s, c) ; // envoie le sommet s de la composante c
    }
  glEnd() ;
}
```

Pour une plus grande efficacité, tant sur le plan des performances que de la consommation mémoire, la liste des sommets envoyée à un GPU peut être compactée dans une *Display List*[10] ou un *Vertex Array*[11]. Nous avons d'ailleurs utilisés des Display Lists dans notre implantation présentée à la section 1.2.5.

Pas à pas détaillé du vertex shader sur GPU Afin d'implanter notre algorithme sur GPU, nous avons décidé d'utiliser un langage de programmation de haut niveau développé par NVIDIA, le CG (Fernando et Kilgard, 2003). Ce langage a la particularité d'être compatible avec les cartes graphiques commercialisées à la fois par NVIDIA et par AMD-ATI. Notez qu'il aurait été tout à fait possible d'utiliser un autre langage tel que le GLSL (Rost, 2004) qui est intégré dans les drivers OpenGL des cartes graphiques.

[10]Une display list est un bloc de commandes OpenGL précompilées et stockées afin d'être utilisées de manière répétée. Un mécanisme de nommage permet de lancer l'exécution d'une display list en une seule commande, ce qui limite les transferts mémoire.

[11]Une liste de sommets peut être stockée dans un tableau appelé Vertex Array. Un mécanisme de nommage permet de lancer la création des sommets décrit dans un Vertex Array en une seule commande.

Ce paragraphe présente, sous forme d'un pseudo-code CG, comment extraire une isosurface d'une cellule sur un GPU avec notre méthode du Marching Cells. Pour une question de clarté, ce pseudo-code se limite au traitement de 8 sommets au maximum. Il permet ainsi de trianguler à la fois des cellules tétraédriques et hexaédriques.

Pour commencer, l'algorithme lit dans la texture les index correspondants aux sommets de la cellule traitée :

Algorithme 3 : Lecture des index des sommets d'une cellule

```
int4 indexSommets_0, indexSommets_1 ;
indexSommets_0 = tex1D ( texture, indexCellule ) ;
indexSommets_1 = tex1D ( texture, indexCellule+1 ) ;
```

Dans l'algorithme 3 ci-dessus, *texture* respecte la stratégie de stockage illustrée à la figure 1.17. Le registre variable indexCellule contient l'index de la cellule dans la texture et tex1D est une fonction qui lit le contenu d'un texel d'une texture.

Ensuite, l'algorithme lit une nouvelle fois dans la texture les valeurs attachées aux sommets de la cellule, toujours en accord avec le schéma de stockage présenté à la figure 1.17 :

Algorithme 4 : Lecture des valeurs scalaires

```
float4 valeurs_0, valeurs_1 ;
valeurs_0.r = tex1D ( texture, indexSommets_0.r ).a ;
valeurs_0.g = tex1D ( texture, indexSommets_0.g ).a ;
valeurs_0.b = tex1D ( texture, indexSommets_0.b ).a ;
valeurs_0.a = tex1D ( texture, indexSommets_0.a ).a ;
valeurs_1.r = tex1D ( texture, indexSommets_1.r ).a ;
valeurs_1.g = tex1D ( texture, indexSommets_1.g ).a ;
valeurs_1.b = tex1D ( texture, indexSommets_1.b ).a ;
valeurs_1.a = tex1D ( texture, indexSommets_1.a ).a ;
```

Chacun des 8 sommet se voit alors assigné un plus ou un moins en fonction de l'isovaleur souhaitée :

Algorithme 5 : Test des valeurs des sommets comparées à l'isovaleur

```
bool4 sommets_testes_0 = ( valeurs_0 >= isovalue ) ;
bool4 sommets_testes_1 = ( valeurs_1 >= isovalue ) ;
```

Le nombre réel de sommets peut ne pas être égal à 8, ainsi certains *plus* ou *moins* préalablement assignés doivent être ignorés :

Algorithme 6 : Neutralisation des signes inutiles

```
bool4 masque_0 = ( indexSommets_0 != 0 ) ;
bool4 masque_1 = ( indexSommets_1 != 0 ) ;
sommets_testes_0 = sommets_testes_0 * masque_0 ;
sommets_testes_1 = sommets_testes_1 * masque_1 ;
```

En résumé, les variables vectorielles sommets_testes_[0|1] valent zéro lorsque la valeur du sommet correspondant est inférieure à l'isovaleur ou que celui-ci n'existe pas. Ces variables valent 1 dans tous les autres cas. Si une cellule ne contient que 5 sommets, ses 4 premiers index sont stockés dans le registre indexSommets_0, et le dernier dans la première composante du registre indexSommets_1. Les trois composantes restantes du registre indexSommets_1 sont fixées par convention à zéro. Ces zéros signifient que les sommets correspondants seront tout simplement ignorés lors du calcul de l'index dans la table de cas, ce qui en pratique est fait dans l'algorithme 6 ci-dessus. L'index de la configuration de la cellule dans la table de cas est ensuite calculé à partir des registres variables sommets_testes_[0|1] :

Algorithme 7 : Calcul de l'index de la configuration dans la table de cas

```
int index = produit_scalaire ( sommets_testes_0, int4(1,2,4,8) ) ;
index += produit_scalaire ( sommets_testes_1, int4(16,32,64,128) ) ;
```

La symétrie de la table de cas (figure 1.13) est exploitée en utilisant ce pseudo-code :

Algorithme 8 : Prise en compte de la symétrie de la table de cas

Si (index $>= 2^{nombre\_de\_sommets-1}$) alors index = $2^{nombre_de_sommets} - 1 -$index ;

La configuration courante est totalement déterminée par ce calcul d'index. A partir de ce dernier, la coordonnée de texture de la ligne de la table de cas correspondante est calculée :

Algorithme 9 : Calcul de la coordonnée de texture correspondant à l'index de la configuration précédemment déterminé

```
int index1D_Table_de_cas = index_racine_des_tables
        + nombre_de_lignes_table_d_aretes*2
        + index*nombre_index_par_ligne_dans_table_de_cas ;
```

Le registre index1D_Table_de_cas contient la coordonnée de texture correspondant à l'index de la configuration courante déterminée précédemment. L'étape suivante parcourt la table de cas dans le but de retrouver la composante requise :

Algorithme 10 : Parcours de la table de cas à la recherche de la composante c requise

```
int nombre_intersections = tex1D ( texture, index1D_Table_de_cas ) ;
int composante_actuelle = 0 ;
int index1D = index1D_Table_de_cas ;
int derniere_arete_intersectee = -1 ;
tant que ( composante_actuelle != c ) {
  index1D += nombre_intersections + 1 ;
  nombre_intersections = tex1D ( index1D ) ;
  si ( nombre_intersections == 0 && derniere_arete_intersectee == -1 ) {
    derniere_arete_intersectee = tex1D ( texture, index1D-1 ) ; }
  composante_actuelle++ ; }
```

Où nombre_intersections contient le nombre d'intersections dans la composante_actuelle. index1D contient l'index dans la table de cas de la première arête intersectée de la composante_actuelle. La variable derniere_arete_intesectee contient l'index de la dernière arête intersectée existante de la composante_actuelle. Le pseudo-code suivant permet d'ignorer les sommets excédentaires envoyés au GPU en les empilant au même endroit que le dernier sommet existant :

Algorithme 11 : Prise en compte du numero de sommet requis

```
Si ( numero_de_sommet < nombre_intersections ) alors index1D += numero_de_sommet ;
sinon index1D = derniere_arete_intersectee ;
```

Le numero_de_sommet correspond à la numérotation des sommets envoyés au GPU par la commande glVertex2s de l'algorithme 2. index1D contient désormais la coordonnée dans la texture où trouver l'index de l'arête intersectée à traiter. Ensuite, l'index de cette arête est lu dans la table de cas puis les sommets lui correspondant sont lus dans la table des arêtes :

Algorithme 12 : Détermination des index des sommets de l'arête intersectée requise

```
int arete_intersectee_courante = tex1D ( texture, index1D ) ;
int extremite_0 = tex1D ( texture, index_racine_des_tables
                  + arete_intersectee_courante*2 ) ;
int extremite_1 = tex1D ( texture, index_racine_des_tables
                  + arete_intersectee_courante*2 + 1 ) ;
```

Les étapes de traitement qui suivent :

- lisent les données relatives à chacun des deux sommets attachés à l'arête traitée en utilisant les variables indexSommets_[0|1] définies par l'algorithme 3,
- interpolent linéairement ces deux sommets en fonction de l'isovaleur afin de déterminer la position de l'intersection avec l'isosurface,
- et finalement déplacent le sommet préalablement envoyé au GPU à cette position.

Commentaires

- Notre approche repousse les limites imposées par l'utilisation exclusive des registres d'une carte graphique en permettant de stocker des tables d'index potentiellement très grandes. Cette nouvelle stratégie de stockage permet d'extraire directement des isosurfaces de cellules polyédriques arbitraires sur un GPU, sans avoir recourt à une tétraédrisation préalable des cellules du maillage.
- L'accès aux données contenues dans des registres ne posent pas de réel problème de performances, cet accès se faisant en quelques cycles d'horloge. En revanche, l'accès aux textures depuis une unité de vertex shader était relativement lente dans les premières architectures GPU qui supportaient cette fonctionnalité. À ce propos, la documentation d'NVIDIA[12] indique qu'une GeForce 6800 peut théoriquement traiter plus de 600 millions de sommets par seconde. En revanche, si un accès à une texture est ajoutée dans le traitement de chaque sommet, les performances tombent à seulement 33 millions de sommets traités par

[12] http ://developer.nvidia.com/object/using_vertex_textures.html

TAB. 1.1 – Caractéristiques des deux ordinateurs de test utilisés dans cette section.

Ordinateur	CPU	Fréq. CPU	RAM	GPU	RAM GPU	Génération du GPU	Shader Model	Nombre d'unités de vertex shader
Portable	INTEL Centrino	2 GHz	1 Go	NVIDIA Quadro FX 1400Go	256Mo	6	3.0	3
Fixe	INTEL Xeon 5140	2.33 GHz	3 Go	NVIDIA Quadro FX 5600	1.5Go	8	4.0	~ 32

seconde. Cette chute de performance est due à la nature même des mémoires embarquées sur les cartes graphiques. Ces mémoires offrent une très grande bande-passante (supérieure à 120Go/s sur les dernières générations), mais au prix d'une latence de plusieurs centaines de cycles d'horloge entre le moment où une donnée est requise, et le moment où elle est effectivement lue. Heureusement, les cartes graphiques sont conçues pour masquer autant que possible ces latences. Elles savent par exemple anticiper l'exécution d'une portion de code non dépendant de données en provenance d'une unité de texture. Il est donc nécessaire de faire très attention lors de l'écriture du code GPU afin de limiter l'accès aux textures à leur minimum. C'est la raison qui nous a poussé à proposer deux méthodes, la première qui utilise exclusivement une texture pour envoyer l'ensemble des données, et la deuxième, hybride, qui utilise des registres pour envoyer les index des sommets des cellules et une texture pour toutes les autres données. Le choix de l'une ou l'autre méthode dépend évidemment du type des cellules traitées.

1.2.5 Applications et performances comparatives

Pour nos tests, nous avons utilisés un PC portable et un PC fixe, avec des grilles de tailles variables (allant jusqu'à 10 millions de cellules) et de types variables (tétraédriques, hexaédriques, hybrides, prismatiques, etc.). Les configurations de nos deux ordinateurs de test sont reportées dans le tableau 1.1. La configuration portable utilise un processeur graphique NVIDIA de sixième génération, la première à supporter l'accès aux textures depuis les unités de vertex shader. De surcroît, ces unités de traitement étaient à l'époque totalement dédiées au traitement des sommets. Leur petit nombre, 3 au total, était par ailleurs faible comparé au nombre d'unités de pixel shader, 8 au total. Côté PC fixe, la carte graphique utilisée est de dernière génération. Elle dispose d'une architecture totalement unifiée, c'est-à-dire que tous les types de programmes GPU (vertex shader, pixel shader...) sont physiquement exécutés par les mêmes unités de calcul, appelées *Stream Processors*, au nombre de 32 unités (section 1.2.1). Ces Stream Processors sont dynamiquement alloués à chacun des types de shader dans le but de maximiser les performances globales. Notre approche utilisant de manière intensive les vertex shaders, ceux-ci bénéficieront d'un maximum d'unités de calcul, ce qui, théoriquement, doit offrir des performances très supérieures aux architectures non-unifiées.

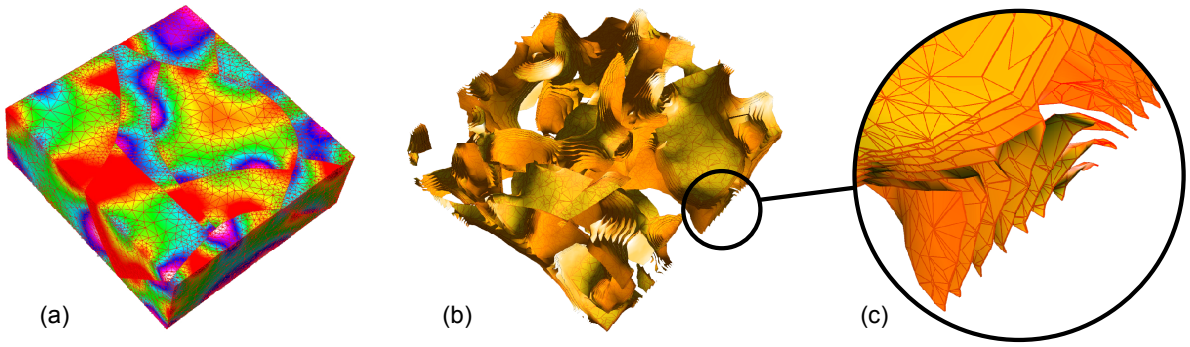


FIG. 1.18 – Le modèle tétraédrique présenté, appelé Mandaros [Earth Decision], contient un champ scalaire issu d'une distribution Gaussienne spatialement corrélée (a). Une série d'isosurfaces en est extraite en temps-réel sur GPU avec notre approche basée sur des textures en (b) et en (c).

Cette section présente les résultats de nos tests sur des grilles tétraédriques, hexaédriques, et prismatiques. Les performances sur des grilles hybrides qui mixent des tétraèdres et des hexaèdres ne sont pas directement reportées dans cette section, puisqu'une grille hexaédrique représente un pire cas d'une grille hybride. Notez que tous les tests présentés dans cette section ont été effectués avec un calcul d'éclairage activé.

Maillages non-structurés composés de tétraèdres (figure 1.18)

La figure 1.19 compare le nombre de tétraèdres traités par seconde en fonction de la taille de la grille utilisée, du PC de test utilisé (portable ou fixe), et ce pour différentes méthodes d'extraction : une purement basée sur un CPU, une autre basée sur un GPU avec un stockage dans des registres (algorithme de Pascucci (2004)), et enfin notre méthode basée sur un GPU avec un stockage dans des textures.

La courbe sur GPU qui utilise des registres peut être découpée en trois parties, et ce quel que soit le PC utilisé. Sur la configuration portable (resp. fixe), pour des grilles d'une taille inférieure à 600000 tétraèdres (resp. 2.5 millions), la mémoire graphique disponible peut stocker l'ensemble des données nécessaires à l'extraction d'isosurfaces. Les performances obtenues sont donc très bonnes, au maximum 6.7 millions (resp. 31 millions) de tétraèdres sont triangulés par seconde. Entre 600000 et 1 million de tétraèdres (resp. 2.5 et 4 millions), la grille ne tient plus totalement en mémoire graphique, et le bus PCI-Express est lourdement chargé afin d'exploiter directement la mémoire du PC pour palier cette limite. Les performances s'effondrent donc rapidement. Passé 1 million de tétraèdres (resp. 4 millions), les performances tombent à des niveaux très faibles.

Les courbes sur CPU et sur GPU basées sur notre méthode qui utilise des textures, restent quasi-linéaires quel que soit la taille de la grille traitée. Nous avons pu valider cette affirmation jusqu'à 5 millions de tétraèdres sur notre PC portable, et 10 millions sur le PC fixe, plus récent et plus performant. La forte réduction de la redondance du stockage des données de notre approche est seule responsable de cette linéarité, et ce qui fait que nous pouvons traiter des grilles bien plus grandes qu'avec les approches précédentes.

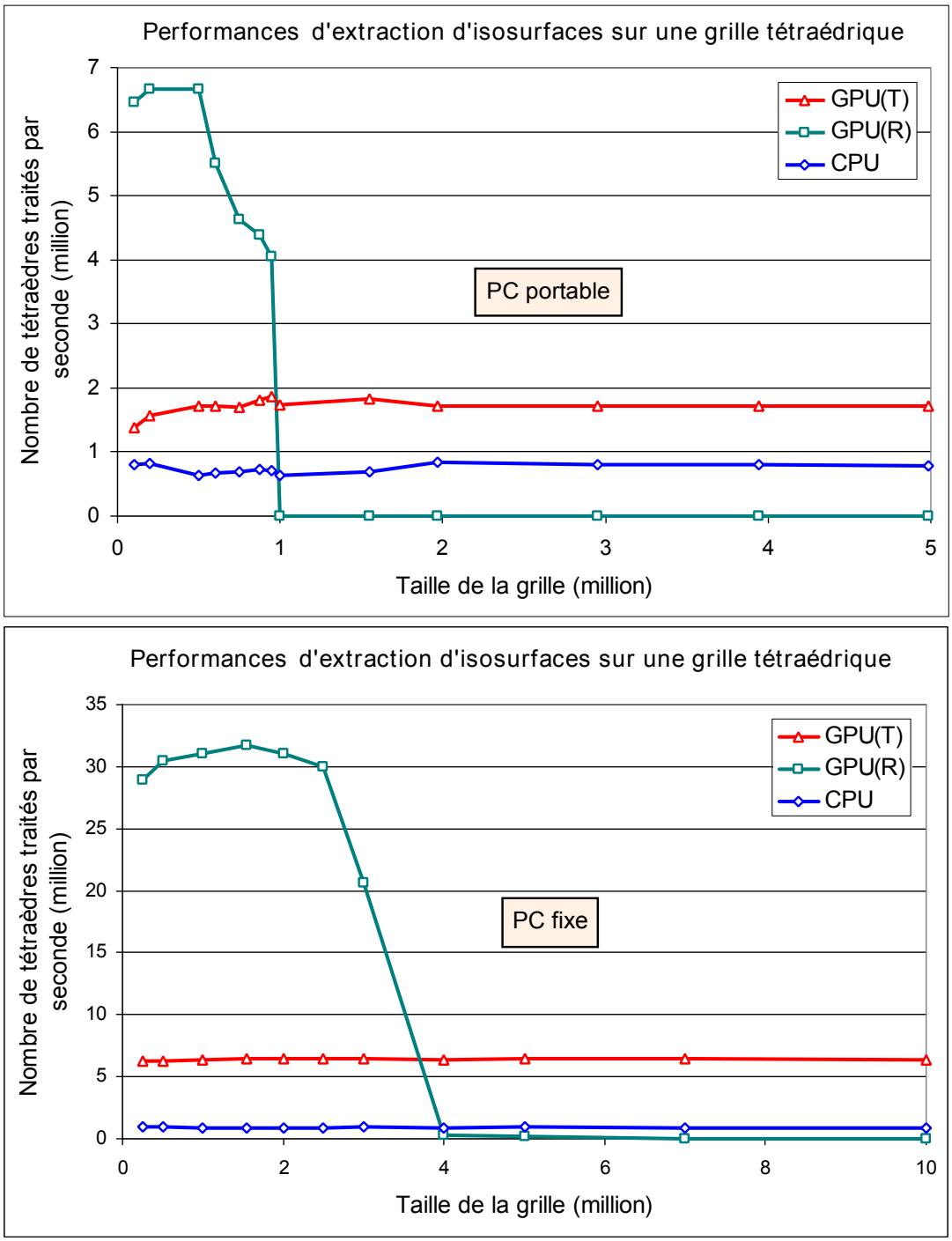


FIG. 1.19 – Performances d'extraction d'isosurfaces sur une grille tétraédrique en fonction de sa taille. GPU(T) désigne notre méthode d'extraction qui utilise un GPU et des Textures. GPU(R) désigne la méthode d'extraction développée par Pascucci (2004) qui utilise un GPU et des Registres. Les courbes du haut utilisent un PC portable, celles du bas un PC fixe. Leur configurations sont détaillées dans le tableau 1.1.

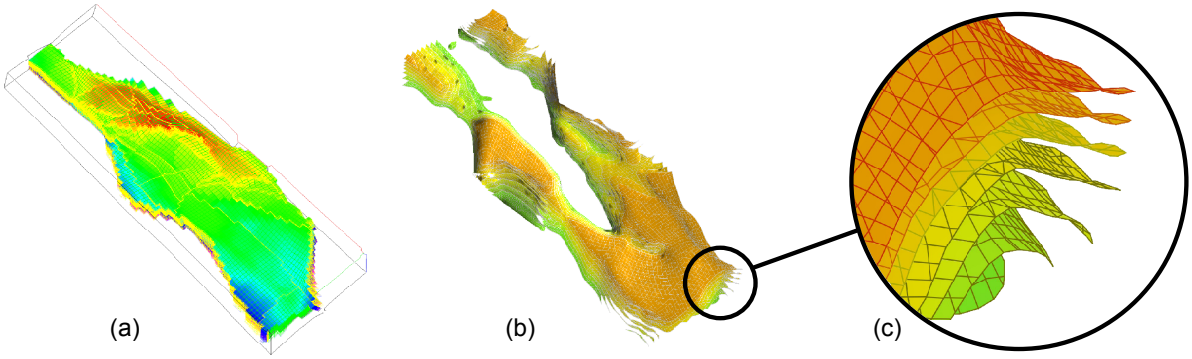


FIG. 1.20 – Le modèle hexaédrique présenté, appelé Nancy I [Total], contient un champ scalaire de pression issu du calcul d'une simulation d'écoulement fluide (a). Une série d'isosurfaces en est extraite en temps-réel sur GPU avec notre approche basée sur des textures en (b) et en (c).

Sur le PC portable, notre algorithme sur GPU est près de 3 fois plus rapide que celui sur CPU, triangulant 1.9 millions de tétraèdres par seconde. L'écart se creuse encore d'avantage sur notre machine la plus récente, le facteur d'accélération grimpant à plus de 7 fois la vitesse atteinte sur CPU, en permettant d'extraire 6.5 millions de tétraèdres par seconde. L'augmentation globale de la puissance de calcul des nouveaux GPU n'est pas la seule responsable de cette large amélioration du facteur d'accélération. Les nouvelles architectures unifiées sont fondamentalement différentes, et permettent ces gains très significatifs.

In fine, notre méthode est plus lente que celle sur GPU basée sur des registres, mais uniquement pour des grilles de moins de 1 million de tétraèdres sur notre PC portable, et de moins de 4 millions sur notre PC fixe. Au delà, notre solution est la meilleure. Notre algorithme est donc complémentaire de ceux de Pascucci (2004); Reck *et al.* (2004), et la décision d'utiliser l'un ou l'autre peut être prise de manière automatique, en fonction de la taille de la grille traitée et de la quantité de mémoire graphique disponible.

Maillages structurés curvilinéaires composés d'hexaèdres (figure 1.20)

La figure 1.21 compare le nombre d'hexaèdres triangulés par seconde en fonction de la taille de la grille utilisée, du PC de test utilisé (portable ou fixe), et ce pour différentes méthodes d'extraction : une purement basée sur un CPU, et notre méthode basée sur un GPU avec un stockage dans des textures. Comme nous l'avons montré dans la section 1.2.2, les méthodes d'extraction d'isosurfaces accélérées au moyen d'un GPU présentes dans la littérature, par exemple dans Pascucci (2004); Reck *et al.* (2004), ne peuvent pas prendre en charge directement les grilles à base d'hexaèdres. C'est pourquoi elles sont absentes de cette comparaison.

Logiquement, les comparaisons de la figure 1.21 montrent que les deux algorithmes testés ont des performances linéaires vis-à-vis de la taille de la grille, et ce quel que soit la configuration matérielle utilisée. Les performances d'extraction sur notre PC portable ont été en moyenne 40% plus rapide sur GPU que sur CPU, traitant en moyenne 850000 hexaèdres par seconde. Une nouvelle fois l'écart se creuse significativement plus sur notre deuxième configuration de test, offrant un facteur d'accélération de 5.5 en moyenne. Le PC fixe permet ainsi de traiter en

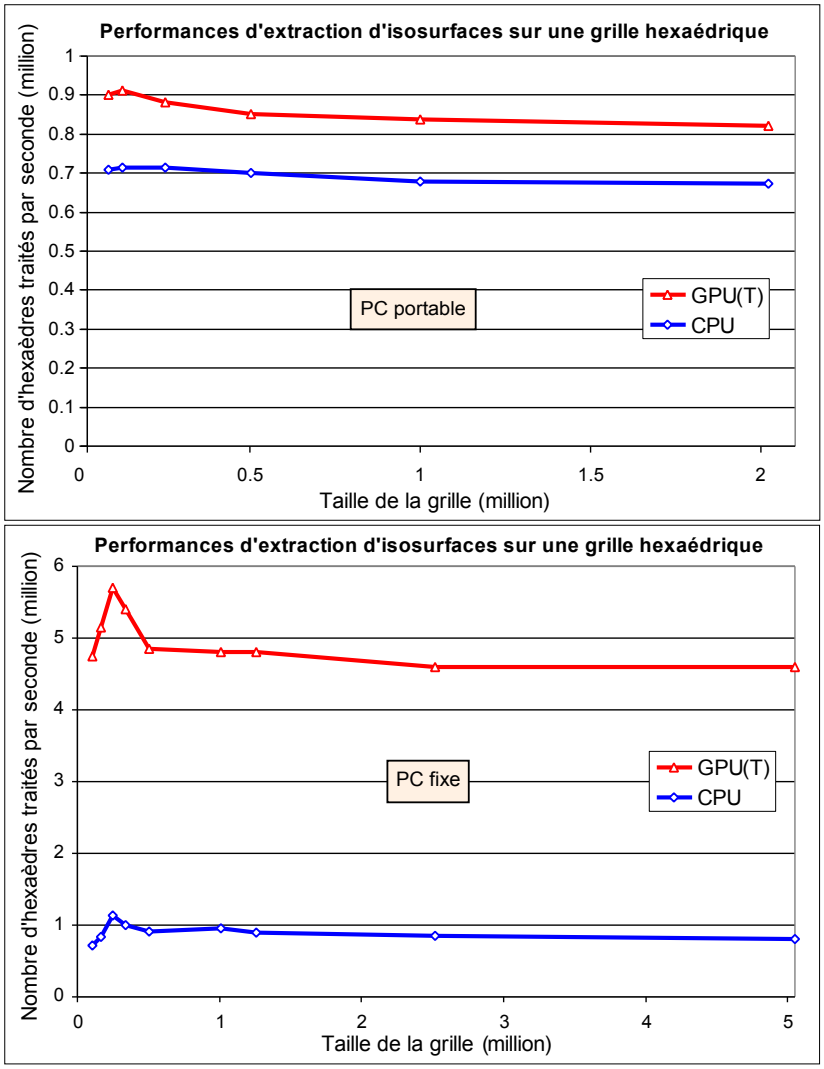


FIG. 1.21 – Performances d'extraction d'isosurfaces sur une grille hexaédrique en fonction de sa taille. GPU(T) désigne notre méthode d'extraction qui utilise un GPU et des Textures. Les courbes du haut utilisent un PC portable, celles du bas un PC fixe. Leur configurations sont détaillées dans le tableau 1.1.

moyenne 4.7 millions d'hexaèdres par seconde. L'architecture unifiée de la carte graphique de notre PC fixe prouve encore une fois sa supériorité.

Comme nous l'avons montré précédemment, au lieu de traiter directement les hexaèdres comme dans notre approche, certains auteurs suggèrent de les tétraédriser en au minimum 5 tétraèdres. Ceux-ci proposent ensuite d'utiliser des techniques d'extraction classiques fonctionnant sur ces tétraèdres. Il est ainsi possible d'utiliser un GPU pour extraire une isosurface d'un hexaèdre préalablement transformé en plusieurs tétraèdres. Un rapide calcul nous montre qu'avec l'algorithme de Pascucci (2004) il serait possible d'extraire $6.7/5 = 1.34$ millions d'hexaèdres par seconde sur notre PC portable, ou bien encore $31/5 = 6.2$ millions d'hexaèdres par seconde sur notre PC fixe. Ces performances paraissent très bonnes, puisque dans le premier cas elle sont environ 60% plus rapide que notre approche directe, et dans le deuxième cas environ 30%. Mais

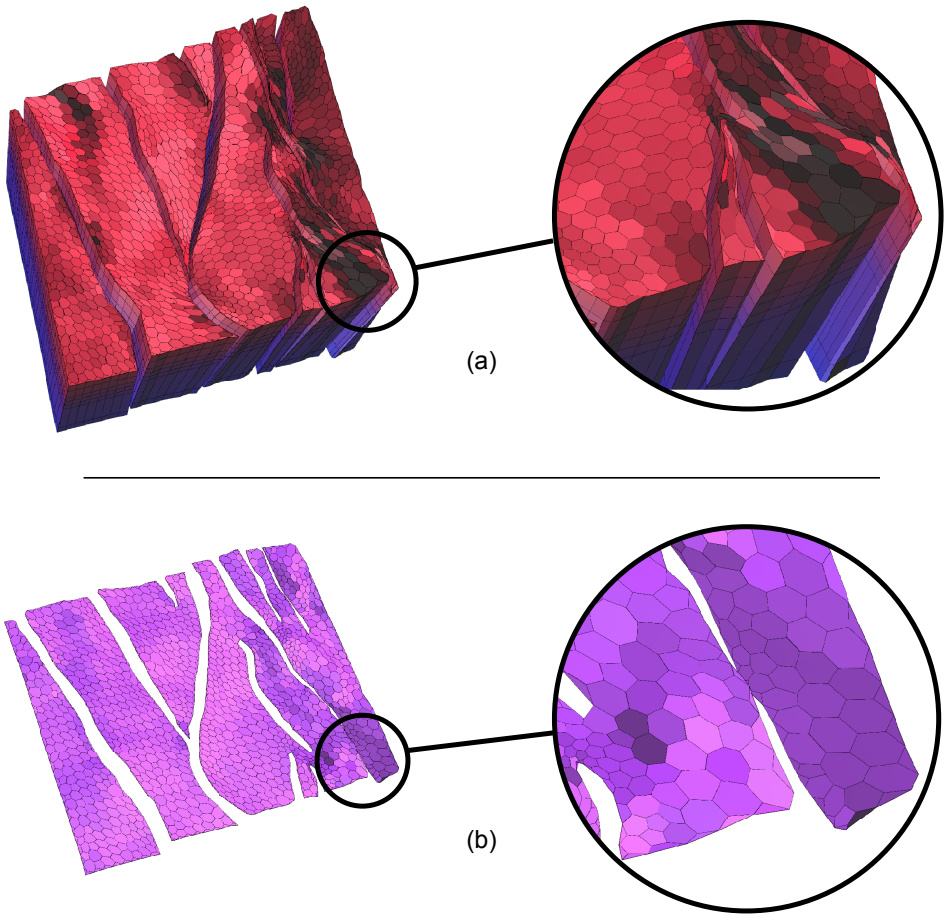


FIG. 1.22 – (a) Grille prismatique définie par une surface polyédrique qui sert de base à une extrusion multicouches suivant des vecteurs prédéfinis. (b) Isosurface extraite à partir de la grille prismatique présentée en (a) avec notre méthode du Marching Cells sur CPU.

ce que ces chiffres *bruts* ne montrent pas, c'est la consommation disproportionnée de mémoire de ces algorithmes. Un nouveau calcul trivial nous montre que, sur notre portable, le nombre maximum d'hexaèdres serait alors limité à seulement $10^6/5 = 200000$. Notre PC fixe serait un peu moins limité, mais un maximum de $4^6/5 = 800000$ hexaèdres apparaît tout de même relativement faible. En outre, ces calculs théoriques ne prennent pas en compte le temps, absolument non-négligeable, que prend la transformation des hexaèdres en tétraèdres. Au final, pour des grilles d'une taille significative, notre approche apparaît une nouvelle fois comme étant la plus appropriée.

Maillages fortement non-structurés composés de prismes (figure 1.22)

Afin de valider notre approche d'extraction directe, nous avons testé notre algorithme du Marching Cells sur des grilles fortement non-structurées de type extrudées à base prismatique (figure 1.22). Ces grilles sont définies par une surface polyédrique qui sert de base à une extrusion

multicouches suivant des vecteurs prédéfinis (Conreaux, 2001; Lévy, 1999; Lepage, 2004; Caumon *et al.*, 2005). Le modèle présenté compte 11900 cellules prismatiques réparties sur 7 couches et dont le nombre de sommets par cellule est compris entre 6 et 18, nécessitant des tables d'index différentes pour chaque type de cellule. Notez que les grilles prismatiques ont une particularité très appréciable : toutes les cellules qui possèdent le même nombre de sommets sont isomorphes, ce qui limite le nombre de tables à stocker. De plus, le nombre de sommets d'une cellule permet de déterminer directement le jeu de tables à lui appliquer.

Actuellement, nous ne disposons d'une implantation générique du Marching Cells que sur CPU, et c'est pourquoi nous ne présentons des résultats que sur CPU pour des grilles prismatiques. En outre, cette implantation en est à un stade de développement très précoce et pourrait être grandement optimisée. Quoiqu'il en soit, sur notre grille de test présentée à la figure 1.22, le PC portable de test a permis de traiter environ 400000 prismes à la seconde tandis que le PC fixe, a atteint environ les 500000 prismes par seconde. Ces performances sont donc honorables étant donné la complexité des cellules traitées. Tout ceci nous permet d'affirmer qu'une approche sans pré-tétraédrisation des données est viable, surtout lorsqu'il est question de cellules formées d'un grand nombre de sommets pour lesquels cette tétraédrisation serait des plus ardues.

Commentaires

- Le support de grilles hybrides qui contiennent à la fois des cellules tétraédriques et hexaédriques est réalisé par le même programme GPU que celui pour les grilles hexaédriques. Deux détails changent : le nombre de sommets envoyés à la carte graphique et les index des tables de cas et d'arêtes à utiliser qui doivent varier en fonction du type de la cellule à trianguler.
- Dans le but de comparer au mieux l'efficacité intrinsèque des différents algorithmes testés dans cette section, les résultats présentés utilisent un parcours exhaustif de l'ensemble des cellules disponibles dans la grille considérée. Ces méthodes, incluant la notre, peuvent être combinées avec des algorithmes dit de *classification* qui limitent le nombre de cellules traitées aux seules cellules intersectées par l'isovaleur souhaitée. L'utilisation de tels algorithmes en combinaison d'une extraction sur GPU est réalisée en n'envoyant au GPU que les index des cellules sélectionnées par l'algorithme de classification, et le nombre de sommets correspondant. La section 1.3 détaille les principaux algorithmes de classification connus à ce jour et propose deux nouvelles méthodes. Cette section 1.3 démontre également tout l'intérêt de la combinaison de ces algorithmes de classification avec notre méthode d'extraction d'isosurfaces sur GPU.
- Notre algorithme d'extraction sur GPU permet sans problème de peindre une propriété secondaire sur une isosurface extraite à l'aide d'une table de couleur. Cette table de couleur associe à chaque valeur scalaire d'une propriété donnée une couleur spécifique. Il est de même tout à fait possible de combiner notre extraction avec un plaquage de texture 2D/3D (Frank, 2006). La figure 1.23 illustre le plaquage d'une propriété secondaire sur une isosurface extraite avec nos algorithmes.

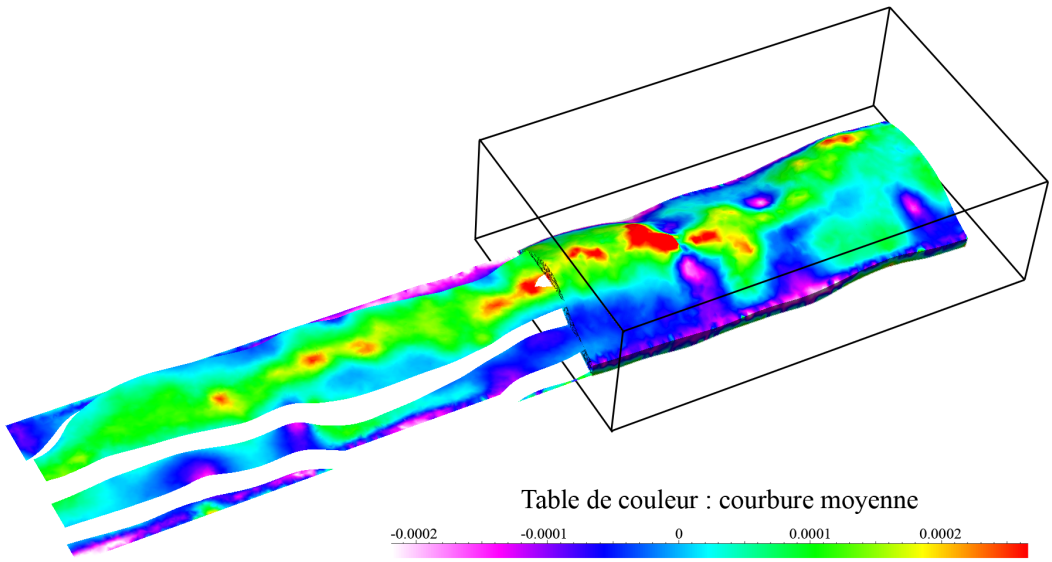


FIG. 1.23 – Isosurface extraite sur GPU peinte avec une propriété secondaire (courbure moyenne) suivant une table de couleur prédéfinie. [Chevron]

1.2.6 Évolutions actuelles et futures

- Les latences d'accès aux textures sont un point faible majeur des cartes graphiques. Heureusement, à chaque nouvelle génération de cartes, leurs concepteurs trouvent des solutions pour les réduire ou, au moins, mieux les masquer. Le fossé qui séparait les performances des registres et celui des textures se réduit d'année en année, ce qui favorise directement notre méthode d'extraction qui utilise justement des textures.
- De même, l'architecture unifiée des cartes graphiques de dernière génération a permis d'accentuer encore plus significativement l'écart de performances avec les implantations sur CPU. En effet, cette architecture permet de maximiser la charge des unités –désormais génériques– de calcul.
- Actuellement, pour chaque cellule, le nombre de sommets dans le pire des cas est systématiquement envoyé au GPU afin de pouvoir traiter toutes les configurations possibles de la cellule considérée. En outre, afin de pouvoir déterminer si un sommet est superflu ou non, la carte graphique doit obligatoirement calculer l'index de la configuration de la cellule dont fait partie le sommet et parcourir la table des cas, ce qui a un coût non négligeable. Ce calcul d'index est par ailleurs fortement redondant puisqu'effectué pour chaque sommet et non pour chaque cellule. Au final, l'envoi en sur-nombre de sommets charge inutilement le bus PCI-Express et force le GPU à effectuer des calculs redondants. L'apparition sur les dernières générations de cartes graphiques des unités de Geometry Shader permet d'envisager de nouvelles extensions à notre approche. Ces unités s'insèrent entre les unités de Vertex Shader et les unités de Pixel Shader. Elles permettraient de totalement éliminer la redondance des calculs à effectuer sur le GPU tout en limitant le nombre de sommets à lui envoyer. En effet, les Geometry Shader ont pour principale fonction d'autoriser la

génération de primitives géométriques directement au niveau du GPU. Avec leur support, un seul et unique sommet sera envoyé au GPU pour chaque cellule à traiter. Le GPU calculera alors une seule fois l'index de la configuration de la cellule et pourra, grâce à la table de cas, générer directement le nombre exact de sommets nécessaires pour représenter l'isosurface correspondante. La prise en charge de grilles fortement non-structurées serait ainsi grandement facilitée tout en offrant des performances très supérieures. Pour plus de détails sur les Geometry Shader, le lecteur est invité à se référer à Uralsky (2006); Cyril (2007)

– La combinaison de tous les facteurs cités ci-dessus permettrait à notre méthode d'encore mieux exhiber ses avantages : la vitesse des méthodes à base de tables d'index telles les Marching Cubes, les capacités de stockage des textures, et la flexibilité des méthodes qui utilisent des liens topologiques afin d'extraire des isosurfaces.

1.2.7 Bilan et perspectives

Notre approche, le Marching Cells, est un algorithme accéléré à l'aide d'un GPU, capable d'extraire efficacement des isosurfaces à partir de grilles fortement non-structurées, ce qui est totalement inédit. Cet algorithme outrepasse de nombreuses limites propres aux approches précédentes grâce à l'utilisation des textures en tant que vecteur de stockage. Celles-ci permettent de stocker efficacement l'ensemble des données qui représentent une grille sans introduire de redondance dans le stockage des sommets partagés entre plusieurs cellules. L'utilisation de textures permet également de limiter au maximum les transferts mémoire sur le bus graphique PCI-Express.

Le Marching Cells définit un générateur automatique de tables d'arêtes et de cas qui généralise l'algorithme du Marching Cubes à tout type de cellules, une stratégie de stockage de ces tables dans une texture, ainsi qu'une méthode qui propose de réduire le nombre de tables à conserver à l'aide d'isomorphismes de cellules. De surcroît, le Marching Cells définit une stratégie générique d'extraction implantable sur GPU.

Notre méthode prend en charge le traitement sur GPU de très grandes grilles structurées ou non. Sur notre PC le plus récent, elle supporte jusqu'à 10 millions de tétraèdres et 5 millions d'hexaèdres, tout en améliorant significativement les performances par rapport à la même implantation sur CPU. Les facteurs d'accélération relevés sont de 7 fois et 5.5 fois en moyenne pour respectivement des grilles tétraédriques et hexaédriques. Comparée aux précédentes approches sur GPU, définies uniquement sur des cellules tétraédriques, notre algorithme apparaît comme plus lent sur les grilles d'une taille petite à moyenne. En revanche, il reprend largement la main grâce à sa linéarité sur de grandes grilles, justement là où le besoin de performance est le plus fondamental. Sur des grilles tétraédriques, notre approche est donc complémentaire avec les précédentes.

En outre, nous avons démontré la viabilité et l'efficacité de l'extraction directe à partir de cellules fortement non-structurées –par exemple sur des grilles prismatiques– comparée aux approches qui pré-subdivisent ces mêmes cellules en tétraèdres pour appliquer ensuite des méthodes plus classiques de visualisation.

Notre méthode possède également l'avantage de pouvoir être aisément combinée, pour une plus grande efficacité, avec des algorithmes de classification comme le démontrera la section 1.3.

Côté CPU, nous disposons actuellement d'une méthode générique complète d'extraction qui supporte des grilles fortement non-structurées grâce à un générateur automatique de tables d'arêtes et de cas. Ce dernier travaille à partir d'une structure topologique de type CIEL. Côté GPU, nous disposons actuellement d'un code spécifique qui permet de trianguler des cellules tétraédriques et hexaédriques. La prochaine étape consisterait à écrire un code générique qui permettrait de trianguler n'importe quel type de cellules à partir de tables spécifiques, ce qui, vu l'étude proposée dans ce chapitre, ne devrait pas poser de problème particulier. L'utilisation des récents Geometry Shader représenterait par contre une évolution majeure qui permettrait de faire générer la géométrie des isosurfaces extraites directement au niveau d'un GPU. Les gains en performances devraient être très significatifs du fait d'une réduction de la redondance des calculs effectués ainsi que de la réduction des données à envoyer au GPU (un seul sommet devant être envoyé au GPU par cellule). De plus, l'utilisation du Geometry Shader simplifierait considérablement le code d'extraction sur GPU.

Comme nous l'avions remarqué précédemment, l'utilisation du Geometry Shader en combinaison de notre Marching Cells permettrait à ce dernier d'amplifier encore ses nombreux avantages : la vitesse des méthodes d'extraction à base de tables d'index, les grandes capacités de stockage des textures, et la flexibilité des méthodes qui utilisent des liens topologiques afin d'extraire des isosurfaces.

D'un autre côté, les API telles que CUDA (Keane, 2006) ou CTM (Peercy *et al.*, 2006) (voir la section 2.3.4 pour plus de détails à propos de ces API) ont ouvert une nouvelle voie de recherche. Ces API offrent des possibilités en terme de calcul et de gestion mémoire inégalées, et permettraient peut-être de développer des techniques innovantes d'extraction d'isosurfaces. À condition toutefois que certaines limitations dont ces API souffrent actuellement, notamment au niveau de leurs capacités d'interactions avec les API graphiques, soient résolues.

1.3 Méthodes d'accélération par classification de cellules

Lors de l'extraction d'une isosurface, il apparaît clairement que seul un sous-ensemble réduit des cellules composant la grille s'avère être réellement intersecté par cette même isosurface. Itoh et Koyamada (1995) puis Cignoni *et al.* (1997) ont par ailleurs montré que le nombre moyen de cellules intersectées par une isosurface était borné par $\mathcal{O}(n^{(d-1)/d})$, n étant le nombre total de cellules composant la grille et d la dimension de l'espace de travail. Dans le cas classique d'un volume, seules $\mathcal{O}(n^{2/3})$ cellules sont donc en moyenne intersectées par une isosurface. Ceci s'explique d'une manière assez simple. Soit une grille régulière de dimension $k \times k \times k$ et une isosurface planaire, alors cette dernière coupe en moyenne k^2 cellules. Trivialement nous pouvons écrire que $k^2 = (k^3)^{2/3}$. Or $k^3 = n$, d'où $k^2 = n^{2/3}$. Nous retrouvons donc la majoration énoncée plus haut. En conséquence, traiter l'ensemble des cellules pour en extraire une isosurface reviendrait à utiliser la majorité du temps à tester des cellules non-intersectées. Dès lors, il est possible d'appliquer une phase d'optimisation se situant en amont de la phase d'extraction présentée à la section précédente. Cette phase d'optimisation a pour but, pour une *isovaleur* donnée, de sélectionner les cellules de la grille intersectées par l'isosurface souhaitée et de les regrouper dans un sous-ensemble. Seul ce sous-ensemble est par la suite envoyé au moteur d'extraction d'isosurfaces (cf. section 1.2).

Le schéma 1.24 présente le processus général permettant d'extraire efficacement une isosurface mettant en oeuvre une méthode de classification de cellules.

La littérature fournit de nombreuses méthodes permettant de déterminer ce sous-ensemble dont les principales sont présentées dans cette section. Le principe général de fonctionnement de toute méthode de classification ainsi que le formalisme mathématique adopté ici sont présentés dans la sous-section 1.3.1. Les sous-sections suivantes présentent les différentes familles d'al-

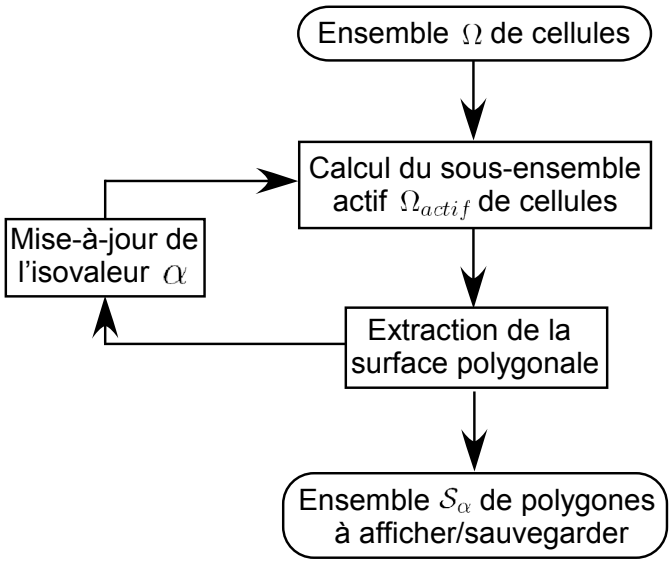


FIG. 1.24 – Processus général d'une extraction d'isosurfaces qui utilise une méthode de classification réduisant l'ensemble des cellules à traiter aux seules cellules intersectées.

gorithmes de classification, à base de partitionnement de l'espace géométrique, de l'espace des valeurs, et à base de propagation spatiale. Sont présentées ensuite deux méthodes adaptatives que nous avons développées afin d'améliorer encore l'efficacité de la classification dans le cadre de l'extraction d'isosurfaces. Enfin, la dernière sous-section 1.3.5 présente une analyse comparative théorique (complexité algorithmique) et pratique (temps de construction, de requête et espace de stockage) des principales méthodes de classification présentées dans ce mémoire.

1.3.1 Formalisme mathématique

Soit Ω l'ensemble qui contient toutes les cellules de la grille. Le sous-ensemble de cellules intersectées par une isovaleur α est classiquement appelé *ensemble actif* et dénoté $\Omega_{actif}(\alpha)$. Soit $\mathcal{F}$ la fonction de valeur qui définit le champ scalaire étudié en chaque sommet de la grille et $\mathcal{R}(c)$ la fonction qui calcul l'intervalle de valeurs pris par une cellule c au regard de la fonction $\mathcal{F}$. $\mathcal{R}(c)$ est définie telle que :

$$\forall c \in \Omega, \mathcal{R}(c) = [\min \mathcal{F}(x_0)_{x_0 \in c}, \max \mathcal{F}(x_1)_{x_1 \in c}]$$

Dès lors, Ω_{actif} est défini comme suit :

$$\forall \alpha \in \mathbb{R} : \Omega_{actif}(\alpha) = \{c \in \Omega | \alpha \in \mathcal{R}(c)\}$$

De nombreuses méthodes existent afin de calculer un sous-ensemble actif de cellules. Notez qu'elles ne sont pas toutes *exactes*, c'est-à-dire qu'elles ne sélectionnent pas exactement toutes les cellules intersectées par une isosurface, mais un ensemble potentiellement plus grand que nous appellerons Ω_{sel} et qui inclut Ω_{actif} :

$$\Omega_{actif} \subseteq \Omega_{sel}$$

Il est possible de regrouper les algorithmes de classification en deux familles : les méthodes dites de partitionnement et les méthodes dites de propagation. Ces deux grandes familles sont présentées dans les sous-sections suivantes. Le lecteur pourra se reporter à la section 1.3.5 pour une étude comparative des performances des principales méthodes présentées ici.

1.3.2 Méthodes utilisant un partitionnement

Les méthodes de partitionnement ne supposent aucun lien topologique entre les cellules et, par conséquent, les classent indépendamment les unes des autres. Dans le cadre de l'extraction d'isosurfaces, afin de calculer un ensemble actif de cellules, il est possible de classer ces méthodes en deux catégories : celles décomposant le maillage dans l'espace géométrique et celles qui le font dans l'espace des valeurs.

Décomposition dans l'espace géométrique

Les *Quad-Trees* (Finkel et Bentley, 1974) découpent récursivement un espace géométrique 2D en quatre sous-espaces nommés *quadrants*. Ils utilisent ensuite un arbre de recherche où tout noeud peut avoir quatre fils. Chaque noeud de l'arbre contient un intervalle de valeurs qui englobe

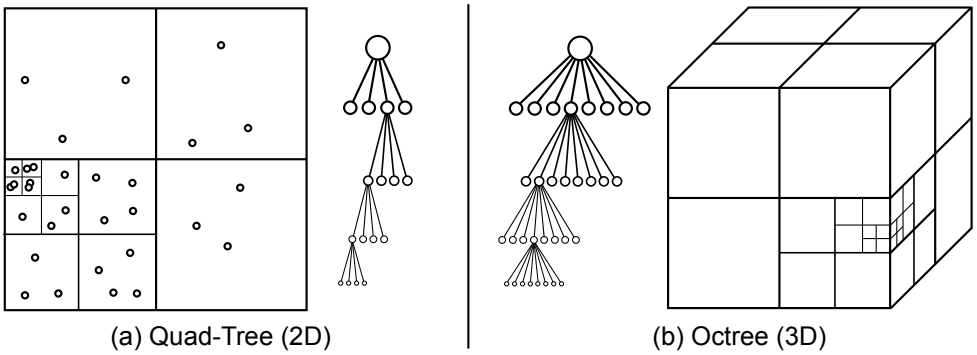


FIG. 1.25 – (a) Quad-Tree subdivisant un espace 2D en quadrants avec son arbre de recherche. (b) Octree décomposant un espace 3D en octants avec son arbre de recherche.

celui de chacun de ses fils et chaque feuille de l'arbre contient une liste des cellules du maillage qu'elle contient ou intersecte. Les *Octrees* (Samet, 1984; Wilhelms et Van Gelder, 1990, 1992) sont une extension classique à un espace 3D des Quad-Trees. Par conséquent, les Octrees subdivisent l'espace géométrique en huit sous-espaces appelés *octants*. Pour extraire l'ensemble actif Ω_{actif} des cellules d'un maillage à partir d'un Octree pour une isovaleur donnée, i.e. l'ensemble des cellules intersectées par l'isosurface, il suffit de récursivement sélectionner les feuilles de l'arbre contenant l'isovaleur recherchée puis la liste des cellules que celles-ci contiennent. La figure 1.25 représente un Quad-Tree et un Octree ainsi que leurs arbres respectifs.

Une des limitations des Quad-Trees/Octrees provient du fait que pour une isovaleur α, l'ensemble des cellules sélectionnées Ω_{sel} peut être bien plus grand que l'ensemble actif Ω_{actif}. On parlera alors de *sur-sélection* des cellules. *In fine*, cette différence peut fortement limiter les performances d'un tel algorithme. Il est a noter également que les structures d'accélération utilisant une décomposition dans l'espace géométrique sont efficaces uniquement lorsque la cohérence spatiale des données est forte.

Décomposition dans l'espace des valeurs

Lorsque les données à classer présentent une grande variabilité spatiale, les méthodes qui utilisent une décomposition dans l'espace des valeurs s'avèrent être les plus efficaces.

En accord avec la définition de la fonction d'intervalle de valeurs $\mathcal{R}$ définie au paragraphe précédent, il est possible de représenter $\mathcal{R}$ pour chaque cellule de la grille considérée sur un graphique 1D ou 2D comme sur la figure 1.26. Sur le graphique 1D, les intervalles sont représentés comme des segments recouvrant une portion du graphique, et l'ensemble des cellules actives est constitué des cellules dont l'intervalle de valeurs contient l'isovaleur α souhaitée. Sur le graphique 2D, les intervalles sont représentés par des points, et l'ensemble actif est celui recouvert par l'aire définie sur la figure 1.26.

L'algorithme des *Bucket Search* (Yagel *et al.*, 1996; Bajaj *et al.*, 1999) est classique et consiste en une subdivision de l'espace des valeurs en k intervalles successifs de taille constante (figure 1.27). Chaque case correspondant à un intervalle I_i, appelé *bucket*, contient une liste des cellules dont l'intervalle de valeurs intersecte celui du bucket. La liste des cellules contenues dans

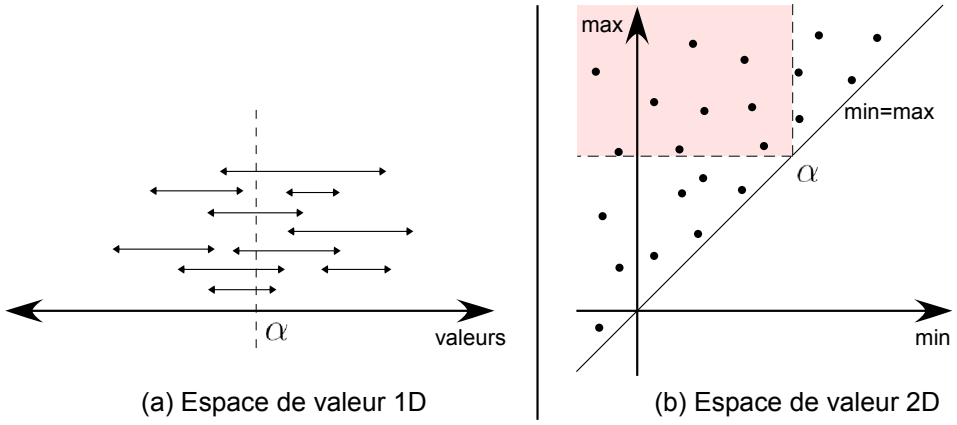


FIG. 1.26 – (a) représente une décomposition 1D de l'espace des valeurs dans laquelle l'intervalle de valeurs $\mathcal{R}$ pris par une cellule est représenté par un segment. (b) illustre une décomposition 2D de l'espace des valeurs dans laquelle un point représente une cellule. Les coordonnées 2D de ce point sont les extrema de l'intervalle de valeurs $\mathcal{R}$ de la cellule considérée.

le bucket(i), $i \in [0, k-1]$, est définie comme suit :

$$bucket(i) = \{c \in \Omega | \mathcal{R}(c) \cap I_i \neq \emptyset\}$$

Construire l'ensemble des cellules actives Ω_{actif} revient à trouver le bon bucket à l'aide d'une fonction d'indexation *index*. Cette fonction établit une correspondance entre une isovaleur α et le bucket dont l'intervalle de valeurs I contient α. Puisque la taille des buckets est constante, la fonction *index* est triviale à implanter. La complexité de recherche est très bonne puisqu'en $\mathcal{O}(k)$, où k est la taille de l'ensemble recherché. Cet algorithme est en général très efficace puisque les listes de cellules sélectionnées sont déjà en place à l'exécution et ne nécessitent pas d'être construites. Mais, à l'instar des Octrees, pour une isovaleur α, les Bucket Search ne garantissent que cette inclusion :

$$\Omega_{actif}(\alpha) \subseteq \Omega_{sel}(\alpha) = bucket(index(\alpha))$$

Par conséquent, l'ensemble Ω_{sel} des cellules sélectionnées par un Bucket Search peut être bien plus grand que l'ensemble Ω_{actif} des cellules réellement intersectées. En effet, pour une isovaleur α donnée, l'ensemble des cellules qui intersectent l'intervalle de valeurs du bucket seront sélectionnées au lieu de celles qui contiennent strictement α. Cette sur-sélection de cellules peut fortement limiter les performances d'un tel algorithme (voir la section 1.3.4 pour une implantation adaptative des Bucket Search et la section 1.3.5 pour une étude comparative des performances).

Gallagher (1991) définit le *Span Filter* afin de réduire la mémoire consommée par le Bucket Search. Initialement, l'intervalle de valeurs global du maillage est subdivisé en sous-intervalles : les buckets. Le nombre de buckets ayant une intersection non-nulle avec l'intervalle de valeurs d'une cellule est appelé *envergure* (ou bien *span length*). Les cellules sont alors distribuées dans différentes listes en fonction de leur envergure appelées *listes d'envergure* ou *span lists*. À l'intérieur de chaque span list, les cellules sont une nouvelle fois triées dans différents buckets unique-

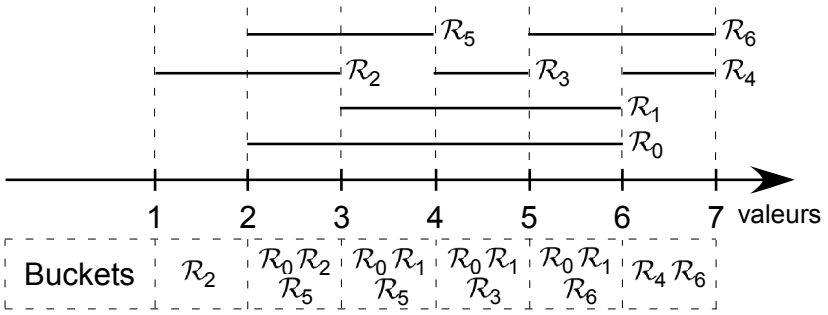


FIG. 1.27 – Décomposition en buckets de l'espace des valeurs recouvert par les intervalles de valeur $\mathcal{R}_i$ des cellules d'un maillage.

ment en fonction de la borne inférieure de leur intervalle de valeurs. Dès lors, chaque cellule n'est stockée que dans un seul bucket. Pour une isovaleur donnée, l'algorithme examine chaque span list. À l'intérieur de chaque span list, les buckets qui contiennent l'isovaleur ou qui sont inférieurs à celle-ci sont sélectionnés en fonction de l'envergure de la liste. L'ensemble Ω_{sel} est finalement reconstruit par fusion des listes contenues dans les buckets sélectionnés. L'utilisation du Span Filter à la place des Bucket Search réduit significativement la consommation mémoire mais au prix de temps de construction de la structure de données et de reconstruction de l'ensemble actif considérablement allongés. De même que pour les Octrees et les Bucket Search, l'ensemble Ω_{sel} des cellules sélectionnées par un Span Filter est potentiellement plus grand que l'ensemble Ω_{actif}.

Giles et Haimes (1990) utilisent deux listes de cellules ordonnées en fonction du minimum et du maximum de l'intervalle de valeurs de chaque cellule. Puis l'intervalle de valeurs maximum $\Delta\mathcal{R}$ sur l'ensemble des cellules est calculé. L'ensemble des cellules sélectionnées pour une isovaleur α est alors construit en deux étapes. Cet ensemble est tout d'abord initialisé avec les cellules de la liste des minima pour les valeurs comprises entre $\alpha - \Delta\mathcal{R}$ et α. La seconde étape consiste à retirer de cet ensemble les cellules dont la valeur maximum est inférieure à α. La consommation mémoire de cet algorithme reste raisonnable mais l'utilisation de multiples tris et recherches d'éléments dans des listes limite fortement ses performances. De même, il est évident que la présence d'une seule cellule ayant un large intervalle de valeurs réduit de façon spectaculaire l'efficacité d'un tel algorithme.

Shen et Johnson (1995) ont étendu l'algorithme de Giles et Haimes en proposant celui du *Sweeping Simplices*. Cet algorithme ajoute des pointeurs qui permettent d'établir une correspondance entre les entrées de la liste des minima et celles des maxima. Pour une isovaleur α donnée, les cellules de la liste des minima qui ont une valeur inférieure à α sont recherchées en ordre décroissant dans la liste des maxima. Les cellules dans la liste des maxima sont alors marquées par un booléen. Dès que la valeur d'une cellule dans la liste des maxima devient inférieure à α le processus est stoppé. Les cellules marquées dans la liste des maxima forment alors l'ensemble actif Ω_{actif} des cellules intersectées par l'isosurface. Comme la méthode de Giles et Haimes (1990), cette implémentation permet de déterminer exactement l'ensemble actif et offre des performances accrues. En revanche, sa consommation mémoire est supérieure du fait de la présence de pointeurs supplémentaires.

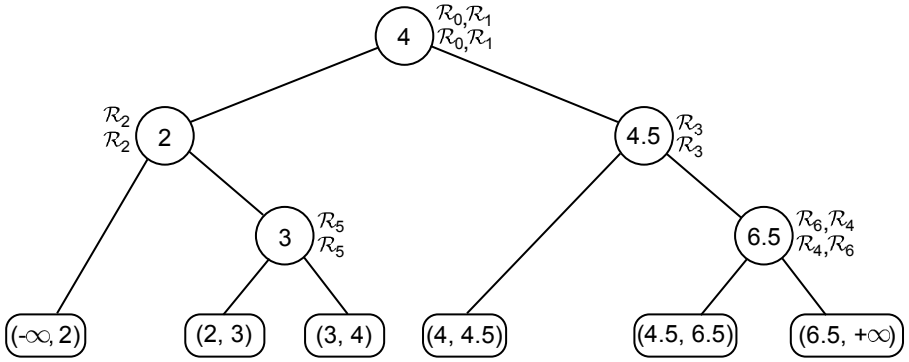


FIG. 1.28 – Interval Tree correspondant aux intervalles de valeur pris par des cellules présentés à la figure 1.27.

Les arbres de recherche sont aussi très utilisés dans le cas d'une décomposition dans l'espace des valeurs. Les *Interval Trees* utilisent un arbre binaire de recherche dont les noeuds stockent des valeurs frontières subdivisant l'espace en deux (Edelsbrunner, 1980; McCreight, 1980, 1985; Cignoni *et al.*, 1997). La construction de l'arbre s'effectue de manière séquentielle lors de l'insertion des cellules dans celui-ci. Lors d'une insertion d'une cellule dans l'arbre, deux cas sont à prendre en compte :

- L'intervalle de valeurs de la cellule intersecte la valeur frontière du noeud courant, et dès lors la cellule se verra stockée dans ce même noeud.
- L'intervalle de valeurs de la cellule est totalement en deçà (resp. au delà) de la valeur frontière du noeud courant, et dès lors le fils gauche (resp. droit) du noeud courant est amené à traiter récursivement la cellule en appliquant ces deux règles.

Si le fils d'un noeud de l'arbre devant traiter une cellule n'existe pas, celui-ci est créé et utilise comme valeur frontière la médiane de l'intervalle de valeurs de la cellule en cours de traitement. A chaque noeud de l'arbre, les cellules conservées sont classées dans deux listes ordonnées en fonction de leurs minima et de leurs maxima. Construire l'ensemble Ω_{actif} revient à parcourir récursivement en profondeur les noeuds de l'arbre en en parcourant les listes ordonnées par minima et maxima. Les cellules sélectionnées en fonction de l'isovaleur requise sont agrégées au fur et à mesure du parcours récursif de l'arbre. Les Interval Trees offrent une complexité de stockage restreinte en $\mathcal{O}(n)$. L'un des désavantages de la méthode est que l'arbre de stockage généré dépend de l'ordre d'insertion des cellules et ne garanti aucunement d'être bien équilibré. De même, la profondeur de l'arbre n'est pas contrôlable et peut freiner les performances de l'algorithme. En revanche, la méthode permet de sélectionner exactement la liste des cellules intersectées par une isovaleur donnée. Nous avons donc :

$$\Omega_{sel} = \Omega_{actif}$$

La figure 1.28 présente un exemple d'Interval Tree construit à partir de l'ensemble d'intervalles de valeur de la figure 1.27

L'un des désavantages des Interval Trees réside dans l'impossibilité de gérer les listes de cellules stockées dans les noeuds de l'arbre. Les *Segment Trees* n'ont pas cette limitation (Mehl-

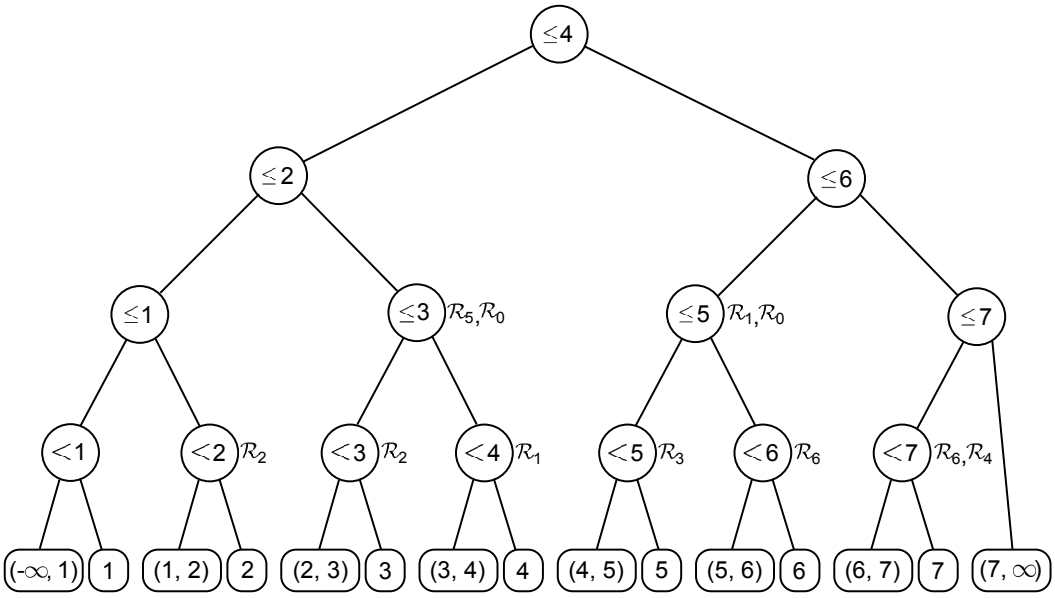


FIG. 1.29 – Segment Tree correspondant aux intervalles de valeur pris par des cellules présentés à la figure 1.27.

horn, 1984; Mulmuley, 1994; Bajaj *et al.*, 1999). Un Segment Tree réalise un découpage d'un intervalle en segments élémentaires combiné avec un arbre binaire de recherche. Les feuilles de l'arbre binaire associé stockent les segments élémentaires tandis que les autres noeuds représentent l'union des intervalles sous-jacents. Ainsi, les noeuds d'un Segment Tree forment une hiérarchie multi-résolution d'intervalles de valeur. Lorsqu'une cellule est insérée dans l'arbre, son intervalle de valeurs est récursivement découpé et propagé de la racine de l'arbre vers ses feuilles en insérant ce même intervalle dans les noeuds qui collectivement le recouvrent. Chaque cellule peut donc être stockée au maximum $\mathcal{O}(\log h)$ fois, où h est la profondeur de l'arbre. La complexité moyenne de stockage d'un Segment Tree est $\mathcal{O}(n \log h)$. Tant pour les Interval Trees que pour les Segment Trees, la complexité de recherche est $\mathcal{O}(k + \log n)$, où k est la taille de l'ensemble sélectionné. La figure 1.29 présente un exemple de Segment Tree pour l'ensemble des segments présentés à la figure 1.27.

L'algorithme *NOISE* (Livnat *et al.*, 1996; Shen *et al.*, 1996) classifie les cellules dans un *Kd-Tree* (Bentley, 1975). Les Kd-Trees sont des arbres binaires de recherche multi-dimensionnels de dimension arbitraire k, et sont en fait un cas particulier des *Binary Space Partitioning Trees* ou *BSP-Trees* (Fuchs *et al.*, 1980). Leur spécificité provient du fait que les plans de coupe utilisés sont alignés sur les axes du système de coordonnées utilisé. De plus, sous sa définition habituelle, chaque noeud d'un Kd-Tree de sa racine jusqu'à ses feuilles doit stocker au moins un élément à classer dans l'arbre. Ce dernier point diffère des BSP-Trees dans lesquels seules les feuilles de l'arbre stockent les éléments.

Étant donné qu'il existe de nombreuses manières de choisir les plans de coupe, il existe de nombreuses manières de construire un Kd-Tree. La méthode de construction canonique propose d'alterner les axes de coupe au fur et à mesure de la descente récursive dans l'arbre. Par exemple,

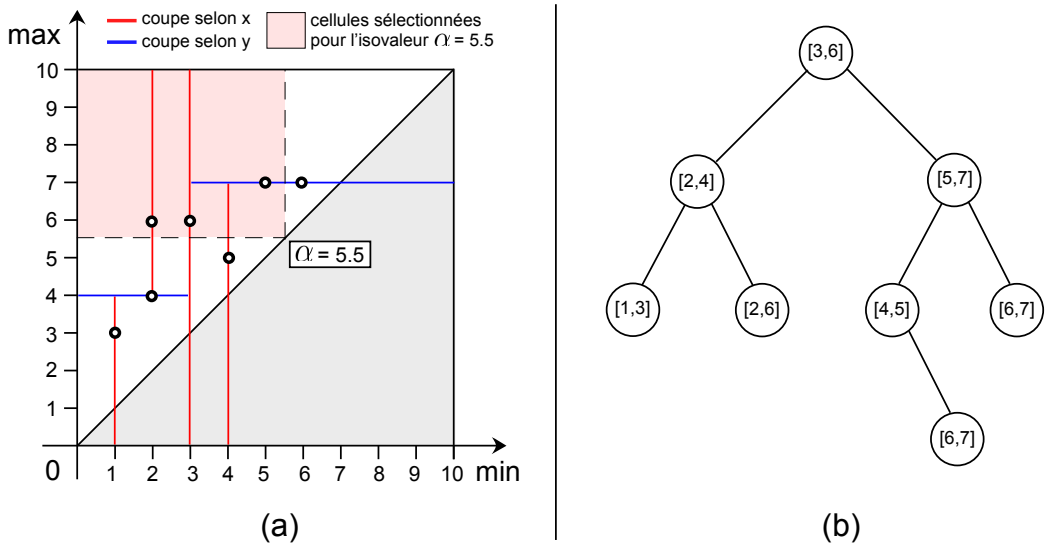


FIG. 1.30 – Classification par un Kd-Tree. (a) intervalles de valeur de cellules présentés à la figure 1.27 représentés dans un espace 2D subdivisé en utilisant la méthode des Kd-Trees. (b) Kd-Tree 2D correspondant. L'aire en rouge représente le rectangle de recherche pour une isovaleur $\alpha = 5.5$.

dans le cas d'un Kd-Tree 3D, on subdivisera alternativement selon les axes x, y puis z, et ainsi de suite. A chaque étape, l'élément sélectionné pour créer le plan de coupe est l'élément médian des éléments qui restent à stocker dans l'arbre. Cette méthode permet d'obtenir un arbre généralement équilibré. Néanmoins, trouver l'élément médian d'un ensemble –potentiellement très grand– peut s'avérer très coûteux puisque cette recherche nécessite de trier cet ensemble. Il est par contre possible d'utiliser des heuristiques qui permettent de déterminer une valeur approchée de la médiane. Par exemple, il est possible d'effectuer un tri sur un ensemble réduit d'éléments sélectionnés de manière aléatoire, ou bien d'utiliser un histogramme de répartition ce qui évite de trier les données et permet d'obtenir un résultat en temps linéaire. En pratique, ces heuristiques permettent d'obtenir des arbres relativement bien équilibrés. La construction d'un Kd-Tree peut être réalisée en $\mathcal{O}(n \log n)$ et une recherche au pire cas en $\mathcal{O}(k + n^{1-1/d})$, où k est le nombre de cellules sélectionnées et d la dimension de l'arbre.

Un Kd-Tree 2D sera utilisé pour trier des intervalles de valeur de cellules comme dans l'exemple de la figure 1.30. Récupérer la liste des cellules intersectées par une isovaleur α revient à effectuer une recherche rectangulaire définie entre les points de positions $(-\infty, +\infty)$ et (α, α). Ce rectangle de recherche est représenté par l'aire en rose-clair sur la figure 1.30 pour une isosvaleur $\alpha = 5.5$. La recherche s'effectue récursivement sur les noeuds de l'arbre de cette manière : (1) si le rectangle recherché est totalement inférieur (resp. supérieur) en comparaison de l'axe de recherche courant, le fils gauche (resp. droit) est exploré. (2) si le rectangle intersecte le noeud courant, alors la cellule stockée au noeud courant est testée afin d'être ajoutée le cas échéant à la liste des cellules actives, et ses deux fils sont explorés à leur tour. Cette recherche est donc effectuée en $\mathcal{O}(k + \sqrt{n})$.

1.3.3 Méthodes utilisant une propagation

Soit un champ scalaire continu, alors une isosurface qui coupe une cellule coupe nécessairement les cellules adjacentes qui partagent les même faces intersectées. Les méthodes dites de *propagation* utilisent cette propriété ainsi que des informations topologiques (la connectivité inter-cellulaire) afin d'exploiter au mieux la cohérence spatiale supposée des données traitées.

Afin de sélectionner les cellules intersectées par une isovaleur, Itoh et Koyamada (1995) calculent un graphe des valeurs extrêmes du champ scalaire $\mathcal{R}$ sur les cellules du maillage. De cette manière, chaque composante connexe d'un isocontour ou d'une isosurface intersecte obligatoirement un arc du graphe. Les isocontours sont alors générés en se propageant à partir d'une cellule dite *graine* détectée le long de ces arcs. Des données bruitées générant de très nombreux extrema peuvent alors fortement limiter les performances et la consommation mémoire d'un tel algorithme. Au pire cas, Livnat et Hansen (1998) ont montré que le nombre d'arcs intersectés est en $\mathcal{O}(n)$, ce qui revient à énumérer l'ensemble des arcs, équivalant finalement à énumérer l'ensemble des cellules.

Ensembles de cellules graines ou *Seed Sets* : algorithme général

Bajaj *et al.* (1996); Van Kreveld (1996); Van Kreveld *et al.* (1997); Bajaj *et al.* (1999) ont une approche légèrement différente de celle de Itoh et Koyamada (1995). Les prochains paragraphes vont décrire leur approche plus en détails afin de préparer la section 1.4 qui étend leurs algorithmes à un espace 4D (une composante spatiale 3D plus une composante temporelle).

L'algorithme suivant étend l'algorithme présenté à la figure 1.24 en présentant comment générer l'ensemble actif Ω_{actif} des cellules intersectées en utilisant des ensembles de graines appelés également *Seed Sets* :

Phase de pré-calcul

- Générer un ensemble de cellules graines par propagation sur le maillage. Stocker cet ensemble dans un Interval Tree pour une récupération rapide des cellules graines.

Initialisation de la phase d'extraction

- Extraire les cellules graines de l'Interval Tree qui correspondent à une isovaleur α donnée.
- Appliquer une *Propagation Spatiale* partant des cellules graines permettant de constuire Ω_{actif}.
- Trianguler les cellules contenues dans Ω_{actif} et dessiner la première isosurface.

Phase de rendu temps-réel (itérée)

- Changer l'isovaleur α.
- Extraire les cellules graines de l'Interval Tree qui correspondent à α.
- Appliquer une *Propagation Spatiale* et construire Ω_{actif}.
- Trianguler les cellules contenues dans Ω_{actif} et dessiner l'isosurface.

Une phase de pré-calcul est effectuée afin de générer un ensemble de cellules graines qui est par la suite stocké dans un Interval Tree (cf. paragraphes précédents pour plus de détails sur les Interval Trees). Lors de la deuxième phase, la liste des cellules graines pour l'isovaleur requise est extraite de l'Interval Tree. Une *propagation spatiale* est alors appliquée à partir de cet ensemble de graines afin de reconstruire l'ensemble actif Ω_{actif} des cellules intersectées. En cas de changement de l'isovaleur requise, il suffit de réitérer la deuxième phase.

Intervalle de connexion et graphe de connectivité des cellules

L'intervalle de connexion $\mathcal{C}$ entre deux cellules c_0 et c_1 appartenant à Ω est défini comme suit :

$$\mathcal{C}(c_0, c_1) = \mathcal{R}(c_0) \cap \mathcal{R}(c_1)$$

On déduit de cette formule que traiter c_0 pour une isovaleur $\alpha \in \mathcal{C}(c_0, c_1)$ implique de traiter c_1 pour le même α. Les intervalles définis par $\mathcal{C}$ peuvent être attachés à des arcs f représentant les connexions entre les cellules du maillage, et implicitement les faces partagées entre ces cellules. À partir de ces intervalles un graphe $\mathcal{G}$ est défini avec $\mathcal{R}$ attaché à ses noeuds et $\mathcal{C}(f)$ attaché à chacune de ses connexions f. Manifestement $\mathcal{G}$ reproduit la topologie de la grille en représentant ses connexions inter-cellulaire.

Deux noeuds c_0 et c_1 de $\mathcal{G}$ sont dit α-connectés s'ils sont connectés par un arc f tel que le scalaire $\alpha \in \mathcal{C}(f)$, où s'il existe un noeud c_2 qui soit α-connecté à la fois à c_0 et c_1. Cette notion est étendue à un ensemble de cellules : un noeud $c \in \mathcal{G}$ est dit connecté à un sous-ensemble $\mathcal{S}$ de $\mathcal{G}$ si, $\forall \alpha \in \mathcal{R}(c)$, il existe un noeud $c' \in \mathcal{S}$ qui est α-connecté à c.

Définition d'un ensemble de cellules graines ou Seed Set

Un sous-ensemble $\mathcal{S}$ de noeuds de $\mathcal{G}$ est un ensemble de graines de $\mathcal{G}$ si tous les noeuds de $\mathcal{G}$ sont connectés à $\mathcal{S}$. Comme $\mathcal{G}$ reproduit la topologie du maillage, un ensemble de graines $\mathcal{S}$ de $\mathcal{G}$ correspond à un ensemble de graines pour le maillage. Etant donné un algorithme de propagation, il est donc possible de construire n'importe quelle isosurface sur l'ensemble de la grille. À cette fin, il faut rechercher dans l'ensemble des cellules graines celles qui contiennent l'isovaleur α souhaitée, puis se propager à partir de ces noeuds dans $\mathcal{G}$ (et par conséquent dans la grille).

Naturellement il peut exister plusieurs ensembles de cellules graines pour une même grille, le meilleur de deux ensembles étant celui dont la cardinalité est la plus faible. Les algorithmes de génération d'ensembles de cellules graines sont basés sur une idée simple : si $\mathcal{S}$ est un ensemble de cellules graines et $c \in \mathcal{S}$ est une cellule connectée à $\mathcal{S} - \{c\}$, alors $\mathcal{S} - \{c\}$ est aussi un ensemble de cellules graines. En partant de l'ensemble de la grille comme sous-ensemble initial ($\mathcal{S} \equiv \mathcal{G}$), retirer successivement les cellules qui satisfont la propriété précédemment énoncée produit un ensemble de cellules graines de faible cardinalité.

Implicitement, retirer une cellule d'un ensemble de cellules graines requiert de mettre-à-jour le graphe $\mathcal{S}$ en lui retirant et en lui créant des arcs. Il apparaît désormais qu'une cellule c peut être retirée si elle est totalement recouverte (au sens des intervalles de valeur) par tous ses voisins directs, c'est-à-dire, si :

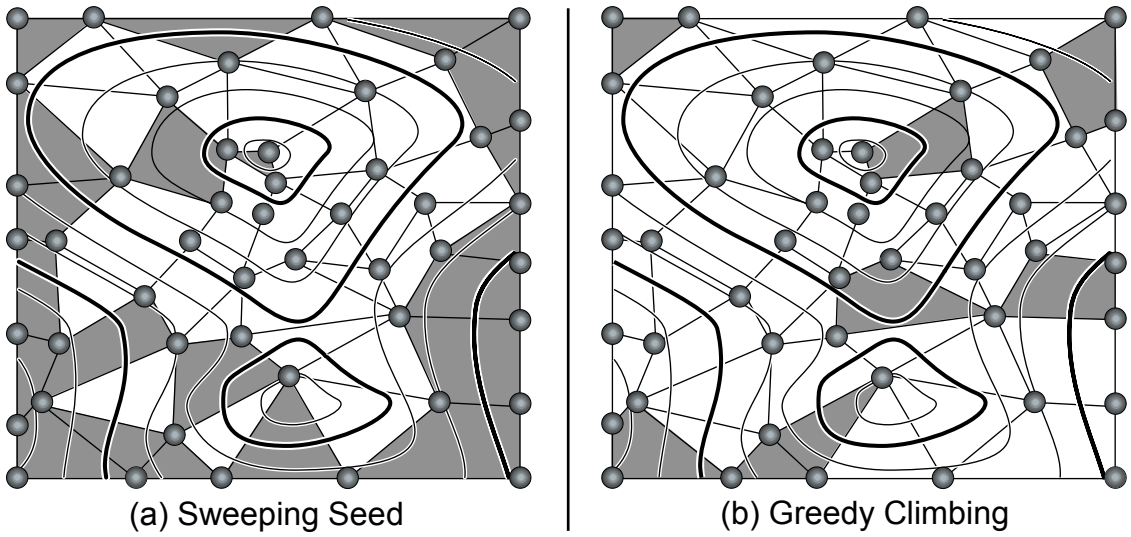


FIG. 1.31 – Comparaison de deux méthodes pour générer des ensembles de cellules graines (cellules en gris) dans une grille 2D composée de 49 cellules au total. (a) Sélection par l'algorithme du Sweeping Seed (Bajaj *et al.*, 1996, 1999) qui génère un ensemble de cellules graines composé de 21 cellules. (b) Sélection par l'algorithme du Greedy Climbing (Bajaj *et al.*, 1999) qui calcule un ensemble de cellules graines de cardinalité 8.

$$\bigcup_{i=0}^{j} \mathcal{C}(f_i) = \mathcal{R}(c)$$

où f_0,..., f_j sont les arcs incidents à c dans le graphe $\mathcal{S}$ courant. Déterminer dans quel ordre retirer des cellules n'est pas trivial et peut fortement influencer la cardinalité de l'ensemble des cellules graines en cas de choix malheureux. En effet, un ensemble minimal obtenu par le retrait de cellules dans un ordre arbitraire peut ne pas être l'ensemble optimal, i.e. l'ensemble de cellules graines le plus petit possible. Une bonne stratégie de retrait peut néanmoins produire un ensemble minimal qui s'approche raisonnablement de l'ensemble optimal. La figure 1.31 propose une comparaison 2D entre deux ensembles de cellules graines générés par deux méthodes de retrait différentes.

Génération d'ensemble de cellules graines

Une solution idéale serait de trouver un algorithme optimal de complexité linéaire. Dans les faits, un tel algorithme est très complexe à réaliser pour des grilles non-structurées. Bajaj *et al.* (1999) décrivent une méthode pratique quasi-optimale qui utilise une propagation gloutonne appelée le *Greedy Climbing*. Cette méthode est basée sur la propagation d'un front évoluant en fonction des cellules recouvertes par les cellules graines déjà sélectionnées. À chaque itération, une nouvelle cellule graine est sélectionnée dans le front d'avancement en choisissant celle présentant le plus grand intervalle de valeurs $\mathcal{R}$. L'ensemble de cellules graines initial est formé par

l'ensemble des cellules de la grille. Chaque fois qu'une nouvelle cellule graine est sélectionnée, les cellules qu'elle recouvre au sens des intervalles de valeur sont exclues. À cet effet, un intervalle $\mathcal{T}(c)$ dit *actif* d'une cellule graine est définit pour chaque cellule c. Cet intervalle représente l'intervalle de valeurs pour lequel la cellule est une cellule graine. De même, un intervalle $\mathcal{P}(c)$ est calculé pour représenter l'intervalle de valeurs propagé en allant vers c depuis ses voisins. Ces deux intervalles, $\mathcal{T}(c)$ et $\mathcal{P}(c)$, sont initialisés en utilisant l'intervalle de valeurs total de la cellule $\mathcal{R}(c)$. Un ensemble de cellules graines est alors généré en répétant itérativement les deux étapes suivantes jusqu'à ce que l'ensemble des cellules du maillage ait été traitées :

- Sélectionner la cellule du front d'avancement ayant l'intervalle $\mathcal{T}$ le plus grand.
- Propager le front d'avancement en considérant la cellule nouvellement sélectionnée comme une cellule graine.

Le front d'avancement est un ensemble dénoté Ω_{front} qui contient toutes les cellules intersectées par l'intervalle de valeurs en cours de test. Lorsqu'une cellule est totalement recouverte par le front, elle est retirée de celui-ci et ne peut plus être utilisée en tant que cellule graine. L'algorithme suivant décrit le fonctionnement de la méthode du *Greedy Climbing* :

Algorithme 13 : Greedy Climbing générant un ensemble de cellules graines

GreedyClimbing (*grille*) {
- sélectionner une cellule racine r ;
- ajouter r au front Ω_{front} ;
- **tant que** (Ω_{front} est non-vide) {
  - choisir la cellule courante c dans Ω_{front} qui a le plus large intervalle $\mathcal{T}$;
  - retirer c de Ω_{front} ;
  - ajouter c à l'Interval Tree avec l'intervalle de graine $\mathcal{T}(c)$;
  - $\mathcal{P}(c) = \mathcal{T}(c)$;
  - désélectionner toutes les cellules du front Ω_{front} ;
  - Propagation (c, *grille*, Ω_{front}) ;

}}}

Dès qu'une cellule c est sélectionnée comme nouvelle cellule graine dans le front Ω_{front}, son intervalle de valeurs $\mathcal{T}(c)$ est récursivement propagé dans l'intervalle $\mathcal{P}$ de toutes les cellules qui lui sont α-connectées et ce pour tout $\alpha \in \mathcal{T}(c)$. Au cours de ce processus de propagation/recouvrement, les intervalles $\mathcal{T}$ des voisins de la cellule graine sont réduit. L'algorithme de *Propagation* suivant montre comment maintenir à jour Ω_{front}, $\mathcal{T}$ et $\mathcal{P}$:

Algorithme 14 : Méthode de propagation pour la génération d'un ensemble de cellules graines

Propagation (c, *grille*, Ω_{front}) {
- ajouter c à une liste temporaire de voisins ltv ;
tant que(ltv est non-vide) {
- cur = premier élément de ltv ;
- retirer cur de ltv ;
- $\mathcal{T}(cur) = \mathcal{T}(cur) - \mathcal{P}(cur)$;
si($cur \in \Omega_{front}$) {
si($\mathcal{T}(cur) = \oslash$) {
- retirer cur de Ω_{front} ;
}}
sinon si($\mathcal{T}(cur) \neq \oslash$) { ajouter cur à Ω_{front} ; }
- marquer cur ;
pour chaque voisin v de cur {
si(v n'est pas marqué) {
si($\mathcal{T}(cur) \cap \mathcal{R}(v) \neq \oslash$) {
si($v \in ltv$) { $\mathcal{P}(v) = \mathcal{P}(v) + \mathcal{P}(cur) \cap \mathcal{R}(v)$; }
sinon {
- $\mathcal{P}(v) = \mathcal{P}(cur) \cap \mathcal{R}(v)$;
- ajouter v à ltv ;
}}}}}

Rendu temps-réel utilisant un ensemble de cellules graines

L'objectif de cette phase est de construire l'ensemble Ω_{actif} regroupant l'ensemble des cellules intersectées pour une isovaleur α donnée. La phase de pré-calcul définie un Interval Tree contenant une liste de cellules graines, la fonction $\mathcal{R}$ ainsi qu'un schéma de *Propagation Spatiale*. En partant des cellules graines récupérées dans l'Interval Tree pour l'isovaleur α, l'algorithme de Propagation Spatiale se propage de voisin en voisin aussi longtemps que ceux-ci sont α-connectés, construisant de la sorte Ω_{actif}. L'algorithme de Propagation Spatiale exploite ainsi la cohérence spatiale des données du fait de l'utilisation des cellules graines et de la topologie du maillage. L'algorithme suivant décrit la méthode de Propagation Spatiale utilisée par Bajaj *et al.* (2002) (recherche en largeur d'abord) :

Algorithme 15 : Propagation Spatiale à partir de cellules graines

PropagationSpatiale (Ω_{actif}, $\alpha_{nouveau}$, *grille*) {
- récuperer les cellules graines pour l'isovaleur $\alpha_{nouveau}$;
pour chaque cellule graine g {
- ajouter g à la liste temporaire de voisins ltv ;
tant que (ltv est non-vide) {
- cur = premier élément de ltv ;
- retirer cur de ltv ;
si ($\alpha_{nouveau} \in \mathcal{R}(cur)$ {
- ajouter cur à Ω_{actif} ;
pour chaque voisin v de cur {
si (v n'est pas marqué et $v \notin ltv$) {
- ajouter v à ltv ;
}}
- marquer cur ;
}}}}

Propagation Temporelle et ensemble de cellules graines

Bajaj *et al.* (2002) ont défini une méthode dite de *Propagation Temporelle* qui exploite encore plus finement la cohérence spatiale et temporelle des données lors de l'extraction d'isosurfaces successives. Le mot *temporelle* est relatif à la variation au cours du temps de l'isovaleur souhaitée, et non pas à des variations des valeurs scalaires. Cet algorithme part, pour une isovaleur α donnée, de l'ensemble des cellules comprises dans $\Omega_{actif}(\alpha)$ et se propage de voisin en voisin pour déterminer les cellules intersectées par la nouvelle isovaleur $\alpha + \varepsilon$. L'algorithme de Propagation Temporelle est défini comme suit :

Algorithme 16 : Propagation Temporelle à partir d'un ensemble actif Ω_{actif} de cellules

PropagationTemporelle (Ω_{actif}, $\alpha_{nouveau}$, *grille*) {
pour chaque cellule $c \in \Omega_{actif}$ {
si ($\alpha_{nouveau} \notin \mathcal{R}(c)$) {
- retirer c de Ω_{actif} ;
}
- marquer c ;
pour chaque voisin v de c {
si (v n'est pas marqué et $\alpha_{nouveau} \in \mathcal{R}(v)$ et $v \notin \Omega_{actif}$) {
- ajouter v à Ω_{actif} ;
}
- marquer v ;
}}}

Une fois cette Propagation Temporelle effectuée, les cellules graines pour $\alpha + \varepsilon$ sont récupérées dans l'Interval Tree construit pendant la phase de pré-calcul, et celles qui ne sont pas déjà marquées, et par conséquent déjà dans le nouvel ensemble de cellules intersectées, servent d'amorce à une phase de Propagation Spatiale. *In fine*, cette étape permet de récupérer les composantes connexes de l'isosurface qui n'ont pas été retrouvées lors de la phase de Propagation Temporelle. La Propagation Temporelle permet un gain de performance compris entre 10 et 30% selon Bajaj *et al.* (2002). Bien entendu, cette méthode est d'autant plus efficace que les isovaleurs consécutives requises sont proches. La fonction qui permet de maintenir à jour la liste des cellules intersectées lors du passage d'une isovaleur α à une isovaleur $\alpha' = \alpha + \varepsilon$ est définie comme suit :

Algorithme 17 : Fonction mettant à jour la liste des cellules intersectées lors du passage d'une isovaleur α à une isovaleur $\alpha' = \alpha + \varepsilon$

MiseAJour (Ω_{actif}, $\alpha_{nouveau}$, *grille*) {
    Exécuter PropagationTemporelle (Ω_{actif}, $\alpha_{nouveau}$, *grille*) ;
    Exécuter PropagationSpatiale (Ω_{actif}, $\alpha_{nouveau}$, *grille*) ;
}}

Remarques à propos des méthodes de classification qui utilisent une propagation

Les méthodes à base de propagation permettent de générer de très petites structures de classification très efficace dès lors qu'il s'agit de traiter de grands maillages. Inversement, elles souffrent de différentes limites et pré-requis. Elles nécessitent de disposer, par exemple, de la topologie du maillage, d'une méthode de marquage des cellules traversées qui soit efficace, et souffrent d'une phase de pré-calcul qui peut être très calculatoire. De plus ces méthodes s'avèrent très difficiles à paralléliser et extrêmement sensibles au bruit présent dans le jeu de données traité.

1.3.4 Les Bucket Search adaptatifs

Les méthodes présentées jusqu'ici ont chacune leurs avantages et leurs inconvénients, ceux-ci pouvant varier en fonction du jeu de données traité. L'algorithme idéal proposerait simultanément des complexités de construction, de stockage et de requête minimales. Les Bucket Search représentent, en moyenne, un bon candidat car ils proposent une construction simple, un espace de stockage raisonnable et un temps d'exécution des requêtes optimal puisque la liste des cellules sélectionnées est statiquement pré-déterminée. La principale limite des Bucket Search tient au fait que l'ensemble de cellules qu'ils sélectionnent Ω_{sel} ne fait qu'inclure l'ensemble Ω_{actif} des cellules réellement intersectées par l'isosurface requise. Lors de nos expérimentations nous avons constaté que, dans certains cas, la cardinalité de l'ensemble sélectionné Ω_{sel} par un Bucket Search pouvait être jusqu'à deux fois supérieure à celle de l'ensemble actif Ω_{actif} (cf. figure 1.35). Ces cas apparaissent, majoritairement, pour des isovaleurs qui intersectent une grande proportion de l'ensemble des cellules, et ce malgré l'utilisation d'un nombre de buckets (cases) relativement important. Les cas les plus problématiques sont caractérisés par une faible variance du champ scalaire et un histogramme de répartition qui exhibe un regroupement de la majorité des don-

nées dans une minorité de colonnes de ce même histogramme. Afin de remédier à ce problème de sur-sélection de cellules, plusieurs solutions existent dont la plus évidente serait d'augmenter le nombre total de buckets utilisés pour segmenter les données. Le risque serait alors de sur-segmenter une grande partie de l'intervalle de valeurs couvert par l'ensemble du champ scalaire défini sur notre maillage afin d'en segmenter correctement la partie la plus dense.

Nous avons opté pour une approche plus fine. Notre objectif est de rendre adaptative la taille des buckets utilisés pour stocker nos listes de cellules afin de segmenter de manière intelligente l'espace de valeurs. Nous proposons d'utiliser deux méthodes de construction de *Bucket Search Adaptatifs* :

- En conformant la taille des cellules à une fonction de distribution cumulée.
- Par subdivision successives de certains buckets en fonction d'un seuil empirique.

Construction conforme à une fonction de distribution cumulée

Cette méthode consiste à construire un Bucket Search Adaptatif pour lequel la taille des buckets est déterminée par la fonction de répartition du champ scalaire étudié. Pour cela on construit un histogramme de répartition des données. Puis, à partir de celui-ci, on construit une fonction de distribution cumulée. Cette dernière va conditionner la taille des buckets dans le but de ranger dans chacun d'eux le même nombre d'éléments. Une fois un nombre de buckets total n_b choisi arbitrairement, pour chaque $i \in [0, n_b - 1]$, le $(i \times n/n_b)$-quantile correspondant est recherché dans la fonction de distribution cumulée déterminant la limite maximum d'un bucket et la limite minimum du bucket suivant. La figure 1.32 montre un exemple d'histogramme, de fonction de distribution cumulée, et les 10 buckets associés pour un jeu de données synthétique.

Avec cette approche adaptative les buckets n'ont plus une taille fixe et il devient impossible de directement les indexer comme des Bucket Search classiques. La solution consiste à utiliser un tableau de pointeurs qui établissent une correspondance entre un quantile et son bucket associé. Une fonction plus générale d'indexation est définie afin d'établir une correspondance entre une valeur scalaire et un index de bucket. Cette fonction utilise la fonction de distribution cumulée pour retrouver le quantile correspondant à la valeur scalaire recherchée, puis le tableau d'indexation quantile/bucket afin de retrouver le bon bucket.

Pour ajouter une cellule dans une telle structure, il faut déterminer à l'aide de la fonction d'indexation les indices i et j des deux extrema de l'intervalle de valeurs pris par cette cellule. Cela revient à déterminer la liste des buckets recouverts par cette dernière. La cellule est ensuite référencée dans chacun des buckets dont les index sont compris dans l'intervalle $[i, j]$.

Construire l'ensemble des cellules potentiellement intersectées Ω_{sel} pour une isovaleur α revient à trouver le bon bucket à l'aide de la fonction d'indexation précédemment définie. La complexité de recherche est très bonne puisque restant en $\mathcal{O}(k)$, où k est la taille de l'ensemble recherché.

Notre approche garantit une répartition adaptative équivalente des données dans différents buckets. En moyenne elle offre de meilleurs performances à l'exécution que les Bucket Search classiques du fait d'un meilleur rapport $\frac{nombre\ de\ cellules\ intersectées}{nombre\ de\ cellules\ sélectionnées} = \frac{\#\Omega_{actif}}{\#\Omega_{sel}}$ dans les zones à forte densité (cf. 1.3.5), c'est-à-dire là où le besoin en performance est le plus essentiel. Inversement, elle sélectionne trop de cellules dans les zones à faible densité car elle garantit un remplissage

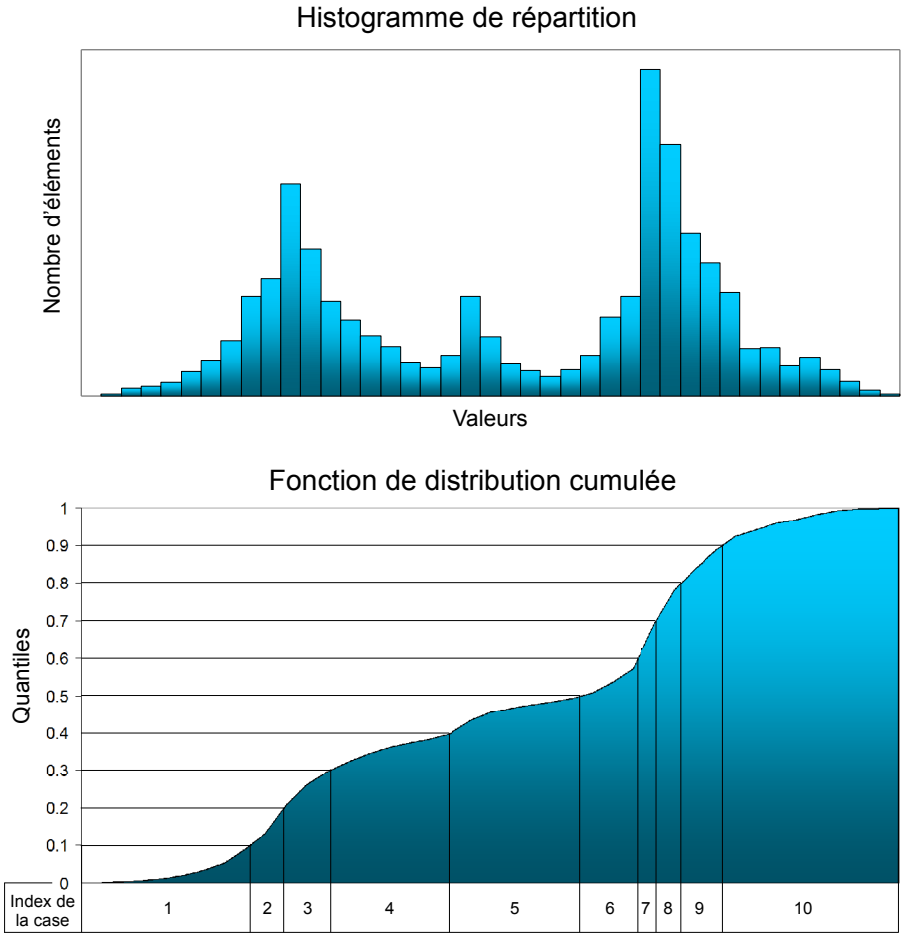


FIG. 1.32 – Méthode de construction d'un Bucket Searh Adaptatif qui utilise un histogramme de répartition (haut) et fonction de distribution cumulée (bas) pour un champ scalaire synthétique et 10 buckets (cases).

uniforme des buckets. Cette méthode impose également une étape de parcours supplémentaire lors de la création de l'histogramme de répartition. Dans certains cas limites, on ne contrôle que très mal sa consommation mémoire qui est en moyenne plus élevée que celle des Bucket Search classiques. Cela provient du fait que la méthode ne cherche pas à classifier des données ponctuelles mais des intervalles de valeur qui peuvent se retrouver à cheval sur plusieurs buckets. Ce phénomène est d'autant plus accentué –comparé aux Bucket Search classiques– lorsque cette méthode sur-subdivise une zone de forte densité. Au final, cela augmente mécaniquement le nombre de buckets successifs dans lequel chaque intervalle est référencé.

Construction par subdivisions successives en fonction d'un seuil

Lorsque la distribution des données exhibe des variations raisonnables, la méthode adaptative précédente offre de bons résultats. Inversement, cette méthode atteint ses limites lorsque la distribution montre de fortes variations. En effet elle pousse à sur-segmenter les zones de grandes

variations et sous-segmenter celles de faibles variations. C'est pourquoi nous avons développé une méthode adaptative dont l'objectif est de ne subdiviser que certains buckets, ceux qui ont la plus grande cardinalité. Cela permet d'obtenir un bon rapport $\frac{\#\Omega_{actif}}{\#\Omega_{sel}}$ dans les zones de fortes densités où ce critère conditionne les performances, tout en conservant une bonne segmentation dans les zones à faible densité, et donc encore une fois un bon rapport. De plus, cette méthode permet de mieux contrôler la consommation mémoire en comparaison de la première méthode adaptative présentée précédemment, puisque le nombre de passes de subdivision conditionne directement la consommation mémoire maximum de l'algorithme.

La première étape de l'algorithme consiste à classer les données en utilisant un Bucket Search classique. Ces buckets sont alors stockés dans un arbre binaire de recherche en attendant les futures éventuelles subdivisions. Ensuite, un seuil empirique est fixé à, par exemple, deux fois le nombre moyen d'éléments stockés dans chaque bucket. Dès lors, chaque bucket qui dépasse ce seuil se voit subdivisé en deux sous-buckets. Il est possible de subdiviser de multiples fois chaque bucket mais la méthode trouve rapidement sa limite car les données stockées sont des intervalles qui recouvrent potentiellement plusieurs buckets. Dans la pratique, seules quelques passes de subdivision suffisent donc. À chaque fois qu'un bucket se voit subdivisé, l'arbre binaire de recherche se voit mis-à-jour en conséquence. Les buckets n'ayant plus une taille constante, on ne peut plus les adresser avec une simple fonction d'indexation. L'arbre binaire de recherche permet de pallier cette difficulté en autorisant une recherche efficace du bucket intersecté par une isovaleur requise. L'exemple de la figure 1.33 représente les différentes étapes de la construction d'un Bucket Search Adaptatif : la construction du Bucket Search classique, le choix du seuil de subdivision, deux étapes de subdivision et enfin la construction de l'arbre binaire de recherche associé. Cet exemple utilise le même jeu de données que celui de la figure 1.32.

Cet algorithme de subdivision nous permet d'obtenir en moyenne un bon rapport $\frac{\#\Omega_{actif}}{\#\Omega_{sel}}$ dans les zones denses tout en conservant une subdivision de l'espace des valeurs suffisant dans les autres zones. De même la consommation de mémoire peut être maîtrisée en limitant simplement le nombre de passes de subdivision. En contrepartie, à nombre de buckets équivalent, la consommation mémoire est plus élevée qu'avec les Bucket Search classiques, mais moins élevée qu'avec notre première méthode qui utilise une fonction de distribution cumulée.

La section suivante propose une étude comparative théorique et pratique des principales méthodes de classification présentées dans ce mémoire.

1.3.5 Étude comparative théorique et pratique des principales méthodes de classification

Étude Théorique

Le tableau 1.2 regroupe les complexités au pire cas pour les principaux algorithmes de classification présentés dans ce mémoire. Dans chaque cas, trois complexités sont reportées : complexité de construction, de stockage et enfin de requête. Les versions adaptatives des Bucket Search présentées à la section précédente ne sont pas reportées dans ce tableau car elles présentent, au pire cas, les mêmes complexités que l'algorithme des Bucket Search classiques. Seules les complexités *au pire cas* sont reportées ici car elles sont les plus représentatives.

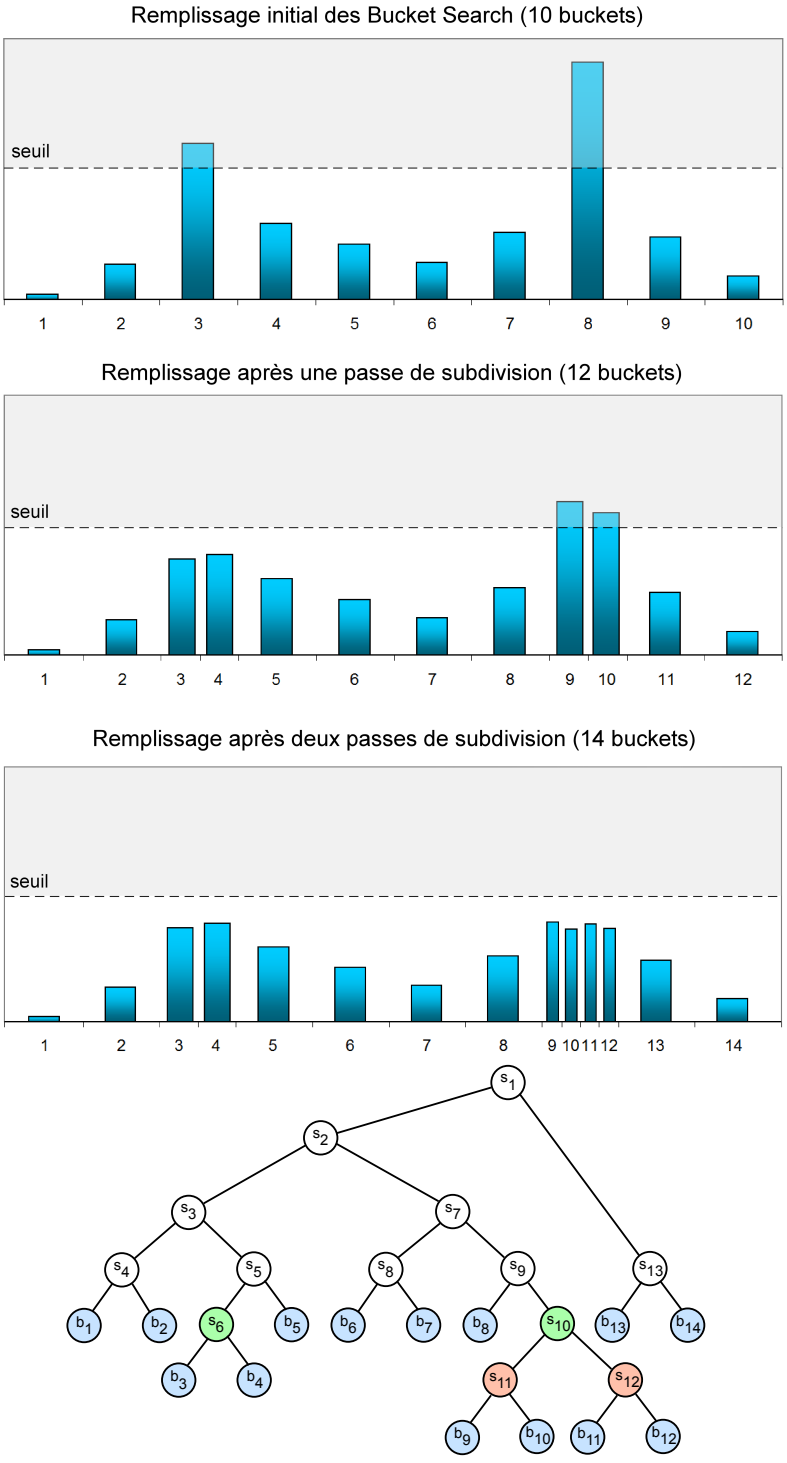


FIG. 1.33 – Méthode de construction d'un Bucket Searh Adaptatif qui utilise une méthode de subdivision. (haut) Bucket Search classique suivi de deux subdivisions successives en fonction du seuil choisi. (bas) arbre binaire de recherche final. Les noeuds bleus b_i contiennent les buckets et les s_i les scalaires qui subdivisent l'espace des valeurs. Les noeuds verts ont été subdivisés lors de la première étape de subdivision, les rouges lors de la seconde. Le jeu de données est identique à celui utilisé à la figure 1.32.

TAB. 1.2 – Complexités de construction, stockage et requête au pire cas des principaux algorithmes présentés dans ce mémoire appliqués à de l'extraction d'isosurfaces. n correspond au nombre total de cellules du maillage, n_b au nombre de buckets utilisés dans la structure de classification et k à la taille de la sortie. La complexité de construction d'un Kd-Tree reportée dans ce tableau tient compte de l'utilisation de l'heuristique de construction présentée à la section 1.3.2. La complexité de construction d'un Seed Set reportée correspond à l'utilisation de la méthode du Greedy Climbing présentée à la section 1.3.3.

	Complexité (au pire cas)		
Algorithme	Construction	Stockage	Requête
Octree	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(k)$
Bucket Search	$\mathcal{O}(n_b n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(k)$
Span Filter	$\mathcal{O}(n_b n \log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Sweeping Simplices	$\mathcal{O}(n_b n \log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Interval Tree	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$\mathcal{O}(k + \log n)$
Segment Tree	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(k + \log n)$
Kd-Tree	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$\mathcal{O}(k + \sqrt{n})$
Seed Set	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$\mathcal{O}(k + \log n)$

Au regard des complexités de construction, les Octrees et les Bucket Search doivent offrir les meilleures performances. Juste après, on trouve les Interval Trees, Segment Trees, Kd-Trees et Seed Sets puis finalement les algorithmes nommés Span Filter et Sweeping Simplices.

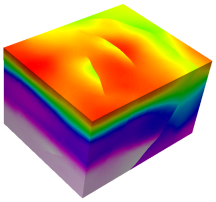
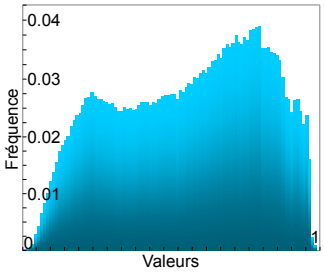
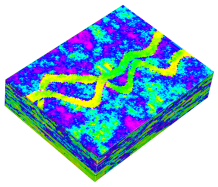
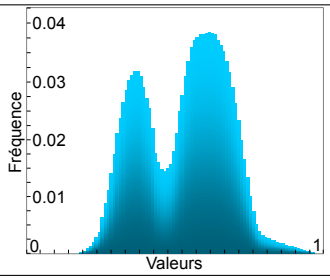
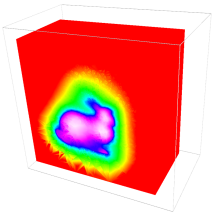
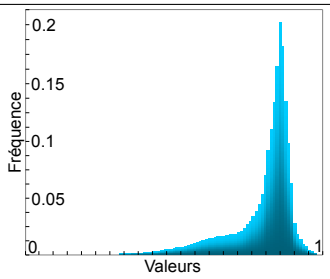
Si l'on considère la complexité de stockage au pire cas, il est difficile de faire apparaître de grandes différences, exception faite des Bucket Search et des Segment Trees qui consomment théoriquement plus de mémoire que les autres méthodes. En pratique on verra au paragraphe suivant que les différences sont souvent très significatives.

Pour finir, la complexité de requête au pire cas permet de détacher plusieurs catégories d'algorithmes. Les meilleurs offrent une complexité dépendante uniquement de la taille de la sortie, c'est le cas des Octrees et des Bucket Search. Par contre, les Bucket Search ont un avantage supplémentaire sur les autres méthodes de classification : les listes de cellules qu'ils sélectionnent n'ont pas besoin d'être construites, elles sont déjà 'en place'. Cet avantage peut également se révéler être un inconvénient puisque comme nous l'avons souligné précédemment, l'ensemble des cellules sélectionnées ne fait qu'inclure l'ensemble actif des cellules réellement intersectées. Une deuxième famille de méthodes comprend les Interval Trees, Segment Trees et Seed Sets offrant une complexité de requête en $\mathcal{O}(k + \log n)$, donc dépendante de la taille de la sortie et de la profondeur de la recherche effectuée. Viennent ensuite la méthode des Kd-Trees et enfin celles des Span Filter et Sweeping Simplices.

Étude Pratique

Ce paragraphe s'attache à exhiber les différences fondamentales entre les principaux algorithmes présentés dans cette section. À cet effet, deux modèles tétraédriques (Cloud Spin [Schlumberger, Paradigm] & Bunny [Stanford]) et un modèle hexaédrique (Stanford V [Stanford]) offrant des caractéristiques différentes et représentatives ont été sélectionnés. Leurs caractéristiques sont reportées dans le tableau 1.3.

TAB. 1.3 – Caractéristiques des modèles utilisés pour tester les principales méthodes de classification de cellules de la figure 1.34.

Modèle	Taille (#cellules)	Histogramme	Propriété étudiée
 Cloud Spin	60K Tétra.		Horizons
 Stanford V	390K Hexa.		Vélocité dans des Chenaux bruités
 Stanford Bunny	1.55M Tétra.		Carte de distance

L'histogramme de répartition du champ scalaire du premier modèle tétraédrique appelé *Cloud Spin* est relativement uniforme et offre donc une bonne répartition des données dans l'espace des valeurs. Le deuxième histogramme qui correspond au modèle hexaédrique *Stanford V* suit cette fois une distribution qui présente deux courbes gaussiennes reconnaissables à leur forme en cloche. Ce modèle a la particularité d'être fortement bruité. Le troisième et dernier modèle tétraédrique étudié, le *Stanford Bunny*, porte un champ scalaire ayant une distribution fonction d'une carte de distance à une surface en forme de lapin, et exhibe un pic très net consécutif à un fort regroupement des valeurs scalaires. Le modèle original du Stanford Bunny est uniquement surfacique, à partir duquel le modèle volumique utilisé ici a été généré à l'aide de la méthode de subdivision en tétraèdres par raffinements successifs développée par Frank (2006). Notez que la taille des modèles de test va crescendo, de 60'000 cellules à 1.55 million de cellules.

Notre machine de test est un PC fixe, le même qui a été utilisé à la section précédente (tableau 1.1). Pour mémoire, il est composé d'un Intel Xeon 5140 Quad-core accompagné de 3Go de RAM. La carte graphique utilisée est une Nvidia QuadroFX-5600 dotée de 1.5Go de RAM graphique.

La figure 1.34 rapporte, pour chaque algorithme de classification, les temps de construction en secondes, l'espace de stockage requis en octets et le nombre de millions de cellules traitées par

seconde pour la requête et l'extraction de 100 isosurfaces pour chacun des trois modèles présentés dans le tableau 1.3. La phase d'extraction d'isosurfaces utilise notre méthode basée sur le GPU et les textures présentée à la section précédente. La profondeur des Octrees construits varie entre 4 et 6 selon les cas, 100 buckets ont été utilisés pour les Bucket Search classiques ainsi que pour nos deux implémentations adaptatives. Le seuil de subdivision utilisé pour notre seconde méthode adaptative basée sur les Bucket Search est égal à deux fois le nombre moyen de cellules de chaque bucket, et deux passes de subdivision ont été utilisées. La méthode de construction utilisée pour les Seed Sets est celle du Greedy Climbing. Les paramètres choisis sont ceux qui permettent d'obtenir les meilleures performances possibles. Chaque test a été réalisé 100 fois afin de moyenner les résultats. Les tests ont été effectués avec l'éclairage activé.

Le haut de la figure 1.34 confirme que l'algorithme des Bucket Search et celui des Interval Trees sont parmi les plus rapides à initialiser. Suivent ensuite les Bucket Search adaptatifs, dans l'ordre, ceux fonction d'une distribution cumulée puis ceux générés par subdivision en fonction d'un seuil. Finalement viennent l'algorithme des Octrees et très loin derrière celui des Seed Sets. Le bruit dans le jeu de données étudié peut avoir un impact très fort selon la méthode de classification utilisée. Les méthodes qui nécessitent d'avoir une bonne cohérence spatiale des données sont très impactées par le bruit présent sur le modèle Stanford V, à savoir les Octrees et les Seed Sets. Inversement, les autres méthodes qui travaillent dans l'espace des valeurs restent efficaces.

Selon la figure 1.34, l'algorithme le plus efficace pour classifier des cellules est celui des Bucket Search adaptatifs utilisant une fonction de distribution cumulée. Comparé à un parcours exhaustif, cet algorithme permet d'accélérer l'extraction par un facteur 5 dans le meilleur des cas en traitant jusqu'à 26 millions de tétraèdres par seconde. Cet algorithme est celui qui consomme le plus de mémoire mais qui, paradoxalement, ne requiert qu'une phase relativement courte de pré-calcul et ce quel que soit le modèle considéré. La seconde méthode de Bucket Search adaptatif présentée dans ce mémoire, fonctionnant par subdivision relative à un seuil, offre des performances très légèrement inférieures tout en utilisant jusqu'à deux fois moins de mémoire à nombre de bucket équivalent. L'algorithme des Bucket Search classique est quand à lui jusqu'à 20% plus lent que nos implémentations adaptatives, mais consomme significativement moins de mémoire.

Après ces trois premières méthodes, on retrouve les Interval Trees qui offrent des performances variables selon le modèle étudié et une consommation mémoire relativement linéaire et raisonnable. Les gains sont de l'ordre de 10 à 25% comparé à un parcours exhaustif. Certes les Interval Trees ne sont que peu sensibles au bruit du champ scalaire étudié, mais la classification d'un grand jeu de données freine leur performances. Cela provient d'une profondeur d'arbre trop importante, d'un mauvais équilibre de celui-ci, et de listes d'intervalles stockés aux noeuds de l'arbre trop longues.

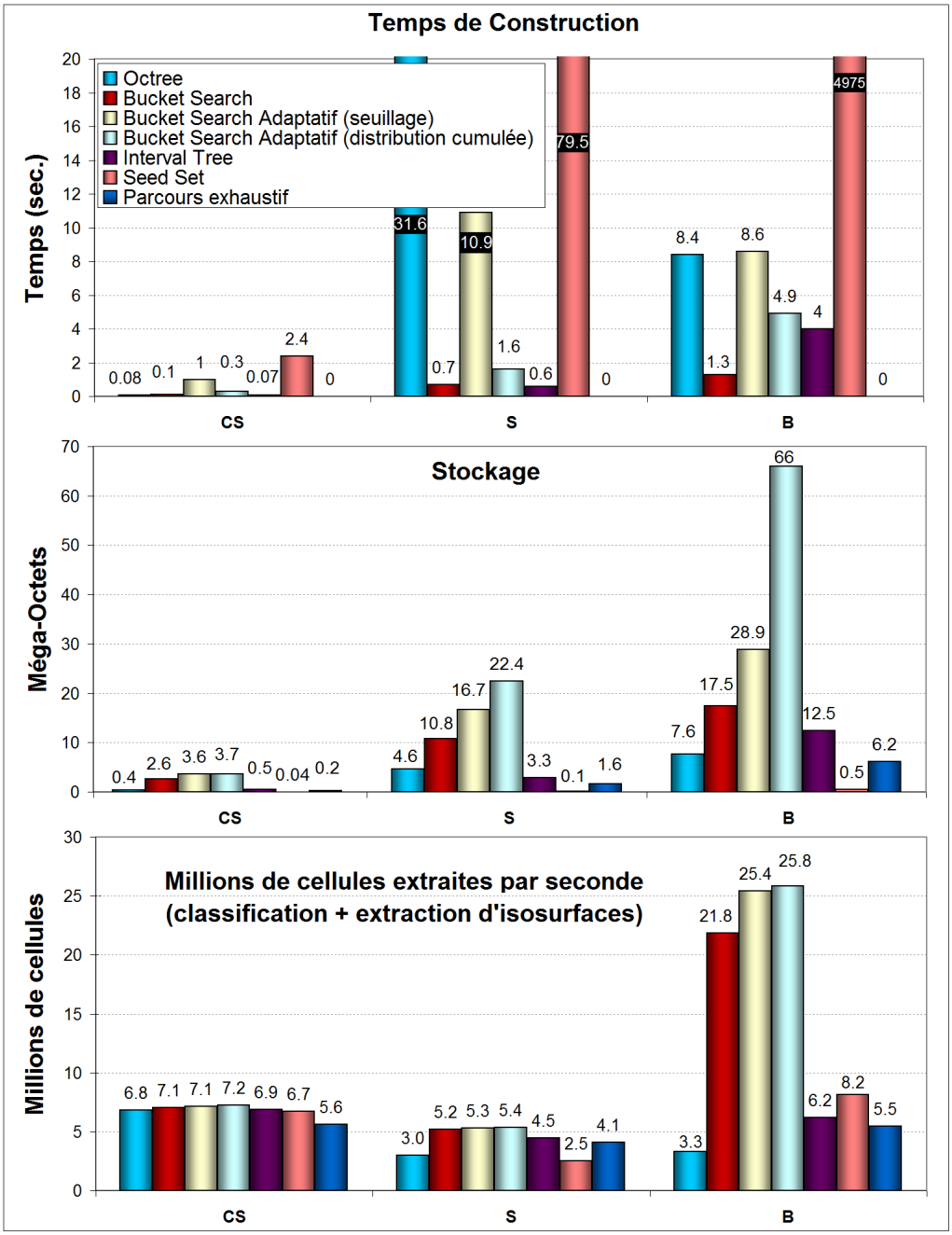


FIG. 1.34 – Performance de construction (en haut, en secondes), stockage (au milieu, en méga-octets) et requête (en bas, en millions de cellules traitées par seconde, comprenant l'utilisation de la structure de classification suivi de l'extraction de 100 isosurfaces couvrant l'ensemble de l'espace de valeurs avec notre méthode sur GPU qui utilise des textures). Les modèles utilisés sont ceux présentés dans le tableau 1.3, CS dénotant le modèle Cloud Spin, S le Stanford V et B le Bunny. La profondeur des Octrees construits varie entre 4 et 6 selon les cas, 100 buckets ont été utilisés pour les Bucket Search classiques ainsi que pour nos deux implémentations adaptatives. La méthode de construction utilisée pour les Seed Sets est celle du Greedy Climbing. Les tests ont été effectués avec l'éclairage activé.

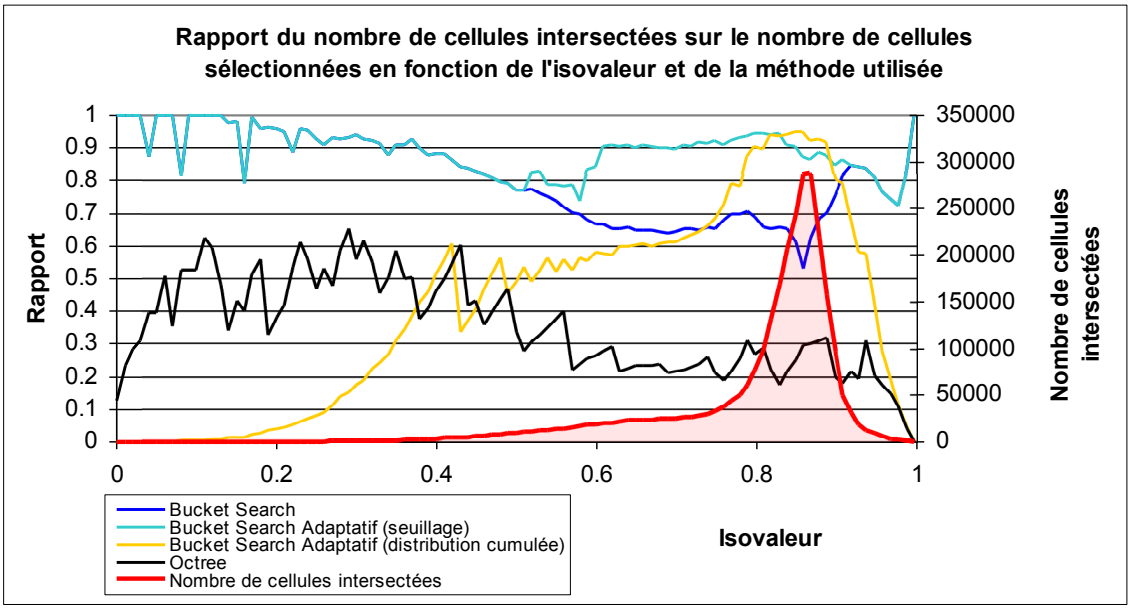


FIG. 1.35 – Rapport du nombre de cellules intersectées sur le nombre de cellules sélectionnées pour chaque algorithme en fonction de l'isovaleur requise. La courbe en rouge représente le nombre de cellules intersectées. Le modèle utilisé ici est le Stanford Bunny.

Les Seed Sets offrent des performances honorables lorsque les données sont exemptes de bruit, tout en ayant une consommation mémoire absolument négligeable (l'ensemble des cellules stockées étant un sous-ensemble de l'ensemble de toutes les cellules). Inversement, lorsque les données sont fortement bruitées, les performances d'un tel algorithme s'effondrent du fait d'une propagation spatiale particulièrement mauvaise et d'un nombre de cellules graines très élevé.

Les Octrees offrent de faibles performances dans le cadre de l'extraction d'isosurfaces sauf lorsque le jeu de données offre une bonne cohérence spatiale et une répartition plutôt uniforme (comme sur le modèle Cloud Spin). À l'instar des Seed Sets, ils réagissent également très mal aux données fortement bruitées. Sur le modèle Stanford V et le Stanford Bunny, la performance des Octrees est mauvaise du fait d'une sur-sélection de cellules et donc un temps de reconstruction de Ω_{sel} très long.

La figure 1.35 présente, sur le Bunny, le rapport $\Omega_{actif}/\Omega_{sel}$, i.e. le rapport nombre de cellules intersectées sur le nombre de cellules sélectionnées, pour les méthodes de classification non-exactes en fonction de l'isovaleur requise. Plus le rapport présenté s'approche de 1 plus la méthode de classification sélectionne bien les cellules, plus il est proche de 0 et plus il sur-sélectionne les cellules.

À cause d'une profondeur d'arbre trop uniforme, sur le modèle du Stanford Bunny, les Octrees présentent un rapport $\Omega_{actif}/\Omega_{sel}$ très inférieur à 1 sur l'ensemble de l'espace de valeurs, d'autant plus mauvais que la zone présente une forte densité de cellules. C'est la raison principale qui rend les Octrees inefficaces sur le modèle étudié. Sur cette même figure, on constate que les Bucket Search classiques offrent un bon rapport mais qui baisse très sensiblement dans les zones à forte densité, i.e. là où justement on a le plus grand besoin d'efficacité. La courbe de notre méthode adaptative basée sur une subdivision en fonction d'un seuil est superposée avec celle

des Bucket Search classiques dans les zones à faible densité, mais fait remonter le ratio dans les zones à forte densité, là où une subdivision a été effectuée. La courbe de notre autre méthode adaptative qui utilise une fonction de distribution cumulée est très différente des autres courbes, exhibant un faible ratio dans les zones à faible densité et un très bon ratio dans les zones à forte densité. Cette méthode est donc légèrement moins efficace dans les zones à faible densité que notre première méthode adaptative et celle non-adaptative, mais en contre-partie exhibe un très bon ratio là où on en a le plus besoin, dans les zones à forte densité de cellules intersectées.

Commentaires

La figure 1.35 montre que l'utilisation d'un Bucket Search adaptatif fonction d'un seuillage permet d'atteindre des rapports $\Omega_{actif}/\Omega_{sel}$ relativement proches de 1 dans la majorité des cas. Il est, par conséquent, peut-être possible de trouver un algorithme de classification qui soit moins consommateur en mémoire que le notre, mais pas un qui soit très significativement plus rapide.

Les performances d'extraction reportées par la figure 1.34 mesurent le temps total pris par l'extraction de 100 isosurfaces couvrant uniformément l'espace des valeurs prises par le champ scalaire étudié. Dans le cas général, ce temps total additionne pour les 100 isovaleurs requises le temps de requête sur la structure qui classifie les données, le temps de transfert des index des cellules sélectionnées vers le GPU et enfin le temps d'extraction sur le GPU de l'isosurface.

Lorsqu'aucune structure de classification n'est utilisée, le temps total additionne le temps d'un seul transfert vers le GPU des index de cellules à traiter –la liste étant statique– puis le temps d'extraction des 100 isosurfaces. Un seul transfert vers le GPU des index de cellules est donc suffisant, ce qui est un avantage supplémentaire lors de l'extraction d'une petite série d'isosurfaces.

En plus de notre méthode d'extraction GPU basée sur des textures, nous avons présenté à la section précédente une méthode basée sur des registres. Cette dernière se combine très mal avec les algorithmes de classification présentés ici pour deux raisons :

- L'utilisation d'une classification oblige à envoyer la liste des cellules actives à chaque changement d'isovaleur.
- La quantité de données à transférer vers le GPU avec cette méthode est très largement supérieure à notre méthode basée sur des textures du fait d'une forte redondance.

Cette dernière constatation impacte fortement les performances de l'algorithme d'extraction basé sur des registres combiné à une classification, mais pas de notre méthode basée sur des textures qui, elle, élimine toute redondance dans les données à transférer. En pratique, les performances d'extraction sur GPU basée sur des registres combinée à une classification sont très mauvaises et toujours inférieures à celles qui n'utilisent pas de structure de classification. C'est pourquoi, afin de ne pas alourdir inutilement cette section, les performances de cette combinaison ne sont pas présentées.

Notez que construire une structure de classification pour n'extraire qu'une très petite série d'isosurfaces n'aurait aucun sens. Malgré tout, sur le Stanford Bunny, en utilisant notre Bucket Search adaptatif basé sur une fonction de distribution cumulée, il suffit d'extraire, en moyenne, 23 isosurfaces pour amortir le temps de construction de la structure de classification sur notre machine de test.

1.3.6 Bilan

Cette section a montré comment il est possible d'accélérer le processus d'extraction d'isosurfaces en utilisant des méthodes de classification des cellules qui s'effectuent en amont de cette extraction, et ce quel que soit le type de cellules. Ces méthodes cherchent à déterminer efficacement un sous-ensemble de cellules Ω_{sel} de Ω qui ne contient que les cellules réellement intersectées pour une isovaleur donnée. Ces méthodes peuvent être classées en plusieurs familles : celles qui subdivisent l'espace géométrique (par exemple les Octrees), celles qui subdivisent l'espace des valeurs (par exemple les Bucket Search) et pour finir celles qui utilisent une propagation de cellule en cellule suivant la topologie du maillage (par exemple les Seed Sets).

Nous avons également présenté deux nouvelles méthodes qui étendent celle des Bucket Search en rendant la taille des buckets adaptative. Cette adaptativité permet de limiter la sur-sélection de cellules pour les isovaleurs les plus critiques. La première méthode présentée utilise une fonction de distribution cumulée afin de déterminer la taille des buckets garantissant un nombre de cellules constant par bucket ainsi qu'une adaptation sur l'ensemble de l'intervalle de valeurs pris par la propriété scalaire étudiée. La seconde méthode présentée est plus locale. Elle part d'un Bucket Search classique dont seuls les buckets qui dépassent un seuil de remplissage sont subdivisés récursivement. L'avantage de cette méthode étant de consommer moins de mémoire que la première tout en étant quasiment aussi rapide. Ces deux méthodes ont permis d'améliorer les performances d'extraction d'isosurfaces d'en moyenne 20% comparé aux Bucket Search classiques, et d'un facteur 5 comparé à un parcours exhaustif des cellules du maillage considéré. Au mieux, 26 millions de cellules par seconde ont pu être traitées sur le PC fixe de test. Dans tous les cas étudiés, nos implémentations adaptatives des Bucket Search s'avèrent être plus rapide pour accélérer une extraction d'isosurfaces que les autres méthodes évaluées.

L'utilisation d'un modèle de test très bruité comme le modèle Stanford V a, comme prévu, fortement limité les performances des algorithmes qui travaillent dans l'espace géométrique (Octree) ou bien par propagation (Seed Set). Ceux-ci nécessitent d'avoir une bonne cohérence spatiale des données. Inversement, ceux qui travaillent dans l'espace des valeurs ont très bien réagi au bruit présent dans le champ scalaire étudié.

D'un point de vue pratique, les algorithmes de classification présentés dans cette section sont majoritairement limités par la bande passante mémoire disponible et non pas par la puissance de calcul des CPU utilisés, rendant une éventuelle parallélisation de ces méthodes peu efficace, tout du moins sur CPU. L'implantation actuelle de la combinaison classification plus extraction est séquentielle et totalement synchrone. Cela signifie que lorsque le CPU travaille, le GPU attend et inversement. Il serait donc tout à fait bénéfique de désynchroniser la classification et l'extraction afin de les paralléliser : pendant que le GPU extrait une isosurface, le CPU construit la future liste de cellules intersectées à l'aide de la méthode de classification choisie. De cette manière, les capacités de calcul du CPU et du GPU pourraient être exploitées de concert.

Dans une optique différente, il serait tout à fait envisageable de porter sur GPU les algorithmes de classification présentés dans ce mémoire, voire d'en développer de nouveaux plus adaptés à la nature hautement parallèle des GPU (Lefebvre *et al.*, 2005). Il serait par ailleurs intéressant d'étudier l'apport d'API telles que CUDA (Keane, 2006) ou CTM (Peercy *et al.*, 2006) dans l'implantation parallèle de tels algorithmes (cf. section 2.3.4 présentant ces API).

1.4 Visualisation avancée : extraction d'isosurfaces 4D continues dans le temps ou *Morphing 4D*

La prise en compte de la dimension temporelle est fondamentale lors de l'étude de phénomènes naturels qui, par essence, évoluent au cours du temps. Par exemple, l'étude d'une simulation de réponse dynamique d'un réservoir pétrolier (Aziz et Settari, 1979) peut permettre d'en caractériser les flux d'écoulement et de prédire la production d'hydrocarbures. C'est pourquoi la visualisation en 4D (espace+temps) s'avère être très importante pour mieux comprendre ces phénomènes. Malheureusement, ce type de visualisation est très difficile à réaliser à cause du très grand volume de données à traiter. Ainsi, en général, seul un petit nombre de pas de temps sont réellement conservés, limitant la perception de l'évolution du phénomène étudié.

L'objectif de notre méthode de *Morphing 4D* (Buatois et Caumon, 2005) est d'augmenter cette perception en générant par interpolation et propagation en temps-réel une isosurface pour n'importe quel temps. Notre approche permet d'exploiter au maximum la forte cohérence temporelle des données entre deux pas de temps successifs pour extraire des isosurfaces de manière continue dans le temps. Pour plus d'efficacité, notre approche combine les algorithmes de classification, de propagation temporelle et notre extraction sur le GPU des isosurfaces.

Nous considérons que la topologie du maillage étudié reste fixe et que seules les valeurs scalaires évoluent au cours du temps.

1.4.1 État de l'art

Le calcul et l'affichage d'un morphing continu entre deux modèles 3D ont été largement abordé dans la littérature relative à l'animation 2D/3D (Lazarus et Verroust, 1998; Alexa, 2002; Hahmann *et al.*, 2007). De même, l'interpolation entre deux images clés 2D a été exploité dans de très nombreux formats de compression vidéo numérique (Gall, 1991). Inversement, l'affichage continu de données 4D a été peu étudié au sein de la communauté scientifique, la plupart des approches ne permettant d'exploiter que les pas de temps disponibles dans le jeu de données étudié (Weigle et Banks, 1998; Shen, 1998; Sutton et Hansen, 1999, 2000; Bajaj *et al.*, 2002; Chiang, 2003). Ces techniques de visualisation génèrent des images claires mais disjointes, ce qui limite la perception de la composante temporelle.

1.4.2 Algorithme général

L'utilisation d'un pas de temps non-exact –devant donc être interpolé– invalide l'ensemble des structures de classification pré-calculées, ce qui rend obligatoire l'utilisation d'une méthode alternative d'accélération de l'extraction de l'isosurface. Bien entendu il est toujours possible de parcourir exhaustivement les cellules de la grille afin d'en extraire une isosurface tout en tenant compte du champ scalaire interpolé, mais cette méthode s'avère extrêmement inefficace. C'est la raison pour laquelle nous avons opté pour une solution intermédiaire qui exploite la cohérence temporelle des données : la propagation temporelle. Chaque cellule sélectionnée au pas de temps précédant est testée au pas de temps suivant, servant de graine pour une propagation de voisin en voisin. Il suffit que cette propagation temporelle sélectionne une seule cellule nouvellement

intersectée pour que la totalité de la composante d'isosurface dont elle fait partie soit reconstruite par propagation. Cette méthode permet d'exploiter au maximum la cohérence temporelle des données. Lorsque le pas de temps requis est un pas de temps exact, la structure de classification correspondante est utilisée pour construire l'ensemble actif.

Notre algorithme général reprend partiellement celui des méthodes de classification par propagation présenté à la section 1.3.3. La principale modification est l'ajout d'une distinction fonction du temps souhaité, les traitements divergeant selon que ce dernier corresponde à un pas de temps disponible dans le jeu de données ou à un pas de temps qui doit être calculé par interpolation. De même, on ne calcule plus une seule structure de classification, mais une pour chaque pas de temps t_i présent dans les données. L'algorithme général est défini comme suit :

Phase de pré-calcul

- Pour chaque pas de temps t_i générer une structure de classification de cellules.

Initialisation de la phase d'extraction

- Construire pour un pas de temps exact t_i et une isovaleur v la liste Ω_{actif} des cellules intersectées à partir de la structure de classification construite à l'étape précédente.
- Trianguler les cellules contenues dans Ω_{actif} et dessiner la première isosurface.

Phase de rendu temps-réel (itérée)

- Mettre-à-jour Ω_{actif} pour le temps t incrémenté de δt :
  - Appliquer une Propagation Temporelle à partir de l'ensemble Ω_{actif} déterminé au temps t pour aller au temps $t + \delta t$.
  - Si $t + \delta t$ correspond à un pas de temps exact, utiliser la structure de classification qui correspond à ce pas de temps pour compléter l'ensemble actif Ω_{actif}.
- Trianguler les cellules contenues dans Ω_{actif} et dessiner l'isosurface.

La phase d'initialisation crée pour chaque pas de temps exact une structure de classification qui permet de calculer un ensemble actif initial.

Lorsque t est incrémenté de δt, dans tous les cas on applique une propagation temporelle. Celle-ci tente de reconstruire $\Omega_{actif}(t + \delta t)$ en se propageant de voisin en voisin à partir des cellules comprises dans $\Omega_{actif}(t)$. La propagation utilisée ici doit désormais tenir compte d'un champ scalaire interpolé à la volée.

À cet effet nous avons redéfini les fonctions $\mathcal{F}$ et $\mathcal{R}$ présentées à la section 1.3.1 ainsi que l'algorithme 16 de propagation temporelle afin que ce dernier prenne en compte directement les valeurs interpolées.

Pour tout temps $t \in [t_i, t_{i+1}]$, on définit $\tilde{\mathcal{F}}(x, t)$ la fonction qui interpole linéairement les valeurs du champ scalaire à partir des deux pas de temps exacts t_i et t_{i+1} encadrant ce temps t. La nouvelle fonction d'intervalle de valeurs $\tilde{\mathcal{R}}$ est définie à partir de $\tilde{\mathcal{F}}$. Il est désormais possible de modifier le précédent algorithme 16 de propagation temporelle en remplaçant simplement $\mathcal{R}$ par $\tilde{\mathcal{R}}$.

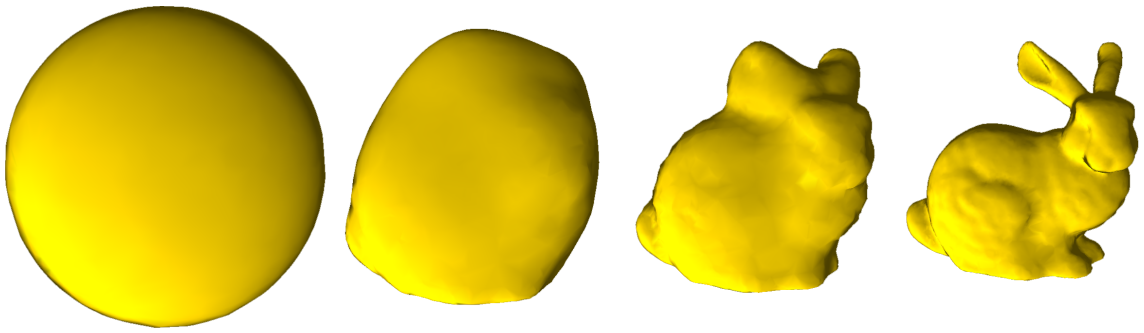


FIG. 1.36 – Exemple de morphing utilisant le modèle du Stanford Bunny.

Une fois la propagation temporelle effectuée, si $t + \delta t$ correspond à un pas de temps exact, alors la structure de classification correspondante est utilisée pour compléter la construction d'Ω_{actif}.

En pratique

En pratique, nous considérons que les pas de temps sont régulièrement espacés et que, par conséquent, $t_{i+1} - t_i, \forall i$ est une constante Δt. Alors t est défini comme ceci : $t = t_i^k = t_i + k * \Delta t / K$ où K est un entier strictement positif appelé facteur de subdivision et $k \in [0, K-1]$. Ω_{actif} est alors calculé pour $t = t_i^0 = t_i$ (pas de temps exact) en utilisant la structure de classification choisie. Pour les pas de temps $t_i^1, t_i^2 ...$ suivants, la propagation temporelle est utilisée jusqu'à ce qu'un pas de temps exact soit atteint (ce qui est garanti par notre définition des t_i^k).

1.4.3 Performances

Pour valider notre algorithme nous avons utilisé le modèle du Stanford Bunny présenté dans le tableau 1.3. L'objectif est de faire un morphing entre deux propriétés, une carte de distance à la surface du Bunny (pas de temps t_0), et une carte de distance au centre de l'objet (pas de temps t_1). Les isosurfaces extraites de cette dernière propriété sont donc des sphères concentriques.

La figure 1.36 montre un exemple de morphing temporel entre les deux propriétés attachées au Stanford Bunny.

Le calcul de l'ensemble actif Ω_{actif} pour les pas de temps exacts est effectué par notre implémentation adaptative des Bucket Search basée sur une fonction de distribution cumulée. Cet algorithme de classification est celui qui offre les meilleures performances (cf. section 1.3.5). Ω_{actif} est construit pour les pas de temps interpolés par l'algorithme de propagation temporelle. Nos méthodes d'extraction d'isosurfaces sur CPU et sur GPU (basée sur des textures) présentées à la section 1.2 ont été étendues pour pouvoir interpoler en temps-réel les champs scalaires étudiés.

Notre machine de test est identique à celle utilisée à la section précédente. Elle est donc composée d'un Intel Xeon 5140 Quad-core accompagné de 3Go de RAM. La carte graphique utilisée est une Nvidia QuadroFX-5600 dotée de 1.5Go de RAM graphique.

Le tableau 1.4 reporte les temps moyens d'extraction d'une isosurface sur notre modèle de test

en utilisant une extraction soit sur CPU soit sur GPU. Deux cas sont étudiés : pour référence, le premier cas dit exact correspond à l'extraction d'une isosurface pour un pas de temps qui existe dans le jeu de données (ici soit t_0 soit t_1) à l'aide d'une méthode de classification ou d'un parcours exhaustif. Le deuxième cas correspond à l'extraction d'isosurfaces pour des pas de temps interpolés en utilisant soit une propagation temporelle pour accélérer le processus, soit un parcours exhaustif. Les tests ont été effectués avec l'éclairage activé.

TAB. 1.4 – Temps moyen d'extraction en secondes d'une isosurface pour le modèle du Stanford Bunny présenté au tableau 1.3. Les temps d'extraction pour des pas de temps exacts sont donnés à titre de références. Nos méthodes d'extraction d'isosurfaces sur CPU et sur GPU (basée sur des textures) présentées à la section 1.2 ont été utilisés. Les tests ont été effectués avec l'éclairage activé.

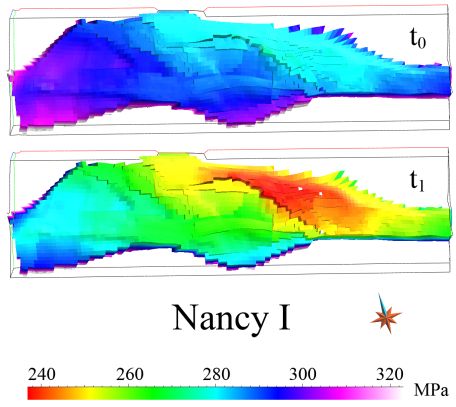
Pas de temps	Classification	Extraction	
		CPU	GPU
Exact	Parcours exhaustif	1.72	0.24
	Bucket Search adaptatif	0.18	0.06
Interpolé	Parcours exhaustif	1.92	0.28
	Propagation temporelle	0.23	0.19

L'analyse du tableau 1.4 fait ressortir plusieurs informations essentielles :

- Dans les mêmes conditions, les implantations sur GPU sont toujours plus rapides que celles sur CPU.
- Le coût de l'interpolation reste tout à fait raisonnable et, grâce à la propagation temporelle, le processus reste interactif et ce quel que soit le moteur d'extraction utilisé, sur CPU ou sur GPU.
- Par rapport à un parcours exhaustif, la propagation temporelle permet d'accélérer considérablement l'extraction d'isosurfaces interpolées : 8.5x sur CPU et 1.5x sur GPU.
- L'impact de la propagation temporelle est plus important sur CPU que sur GPU.

Ce dernier point nécessite d'être éclairci. Les performances observées sont dues à une répartition différente des temps de calcul. La phase d'extraction est beaucoup plus rapide sur GPU que sur CPU tandis que le temps pris pour classifier les cellules reste le même. Ainsi, sur CPU, le temps de classification des cellules est faible comparé au temps pris par l'extraction de l'isosurface tandis qu'on assiste au phénomène inverse sur GPU. Au final, ceci explique pourquoi la classification des cellules a un impact sur le temps total de l'algorithme sur GPU inférieur à celui sur CPU.

TAB. 1.5 – Caractéristiques du modèle de réservoir pétrolier utilisé pour tester notre algorithme de morphing 4D. [Avec l'aimable autorisation de Total]

Modèle	Taille (#cellules)	Propriété étudiée
	85K Hexaèdres	Évolution de la pression sur deux pas de temps t_0 = janvier 2001 et t_1 = janvier 2002

Remarque et évolution des performances

De même que pour notre implantation des algorithmes de classification et d'extraction, l'implantation actuelle de notre morphing 4D, et par conséquent de notre méthode de propagation temporelle, est totalement séquentielle et synchrone. Cela signifie que lorsque le CPU travaille, le GPU attend et inversement. Il serait certainement avantageux de rendre la méthode asynchrone pour que pendant que le GPU extrait une isosurface, le CPU puisse calculer par propagation la future liste de cellules intersectées en parallèle.

1.4.4 Application à un réservoir pétrolier

Cette section montre comment notre méthode de morphing 4D peut être utilisée dans un cas d'étude réel, celui d'un réservoir pétrolier. Nous avons donc utilisé le modèle appelé Nancy I [Total] dans lequel l'évolution de la pression a été simulée. Plusieurs pas de temps sont disponibles à partir desquels notre méthode peut extrapoler un nombre infini d'isosurfaces intermédiaires. Le tableau 1.5 présente les caractéristiques du modèle étudié, tandis que la figure 1.37 compare les résultats d'interpolations obtenues selon la méthode utilisée.

Dans la figure 1.37 les isosurfaces en rouge correspondent à des pas de temps exacts, celles en vert à des isosurfaces calculées par interpolation linéaire. (a) présente le résultat obtenu avec un parcours exhaustif des cellules de la grille tandis que (b) et (c) utilisent notre méthode de propagation temporelle. (b) montre les isosurfaces extraites à partir du pas de temps t_0 en direction des temps croissants jusqu'à arriver à t_1, tandis que (c) fait ce parcours en sens inverse.

(a) teste l'ensemble des cellules de la grille ce qui permet d'obtenir toutes les composantes de l'isosurface. Cette première sous-figure nous sert donc de référence. Clairement, (a) et (b) sont identiques, ce qui est naturel étant donné qu'aucune nouvelle composante connexe n'est créée au cours de l'évolution des temps de t_0 vers t_1. Dans le sens inverse, de t_1 vers t_0, qui

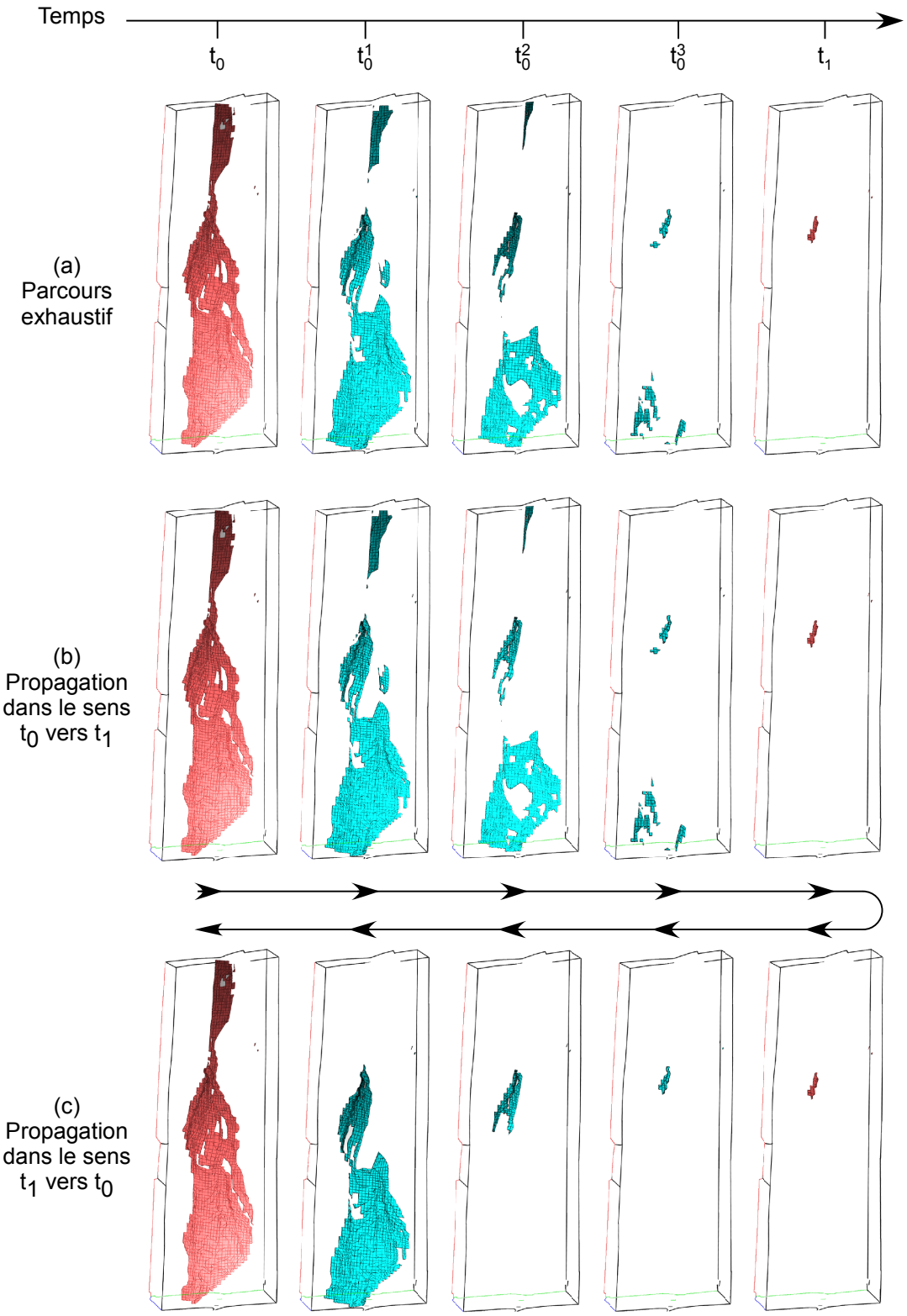


FIG. 1.37 – Isosurfaces interpolées en temps-réel et calculées à l'aide d'un parcours exhaustif des cellules (a) ou bien à l'aide de notre algorithme de propagation temporelle (b et c). t_0 correspond au premier janvier 2001 et t_1 au premier janvier 2002. L'isovaleur extraite est égale à 260 MPa. Les isosurfaces en rouge correspondent à des temps exacts, celles en vert à des temps interpolés.

correspond à la sous-figure (c), certaines composantes de l'isosurface manquent pour les pas de temps interpolés. Seule la composante subsistant en t_1 est amenée à se propager et les nouvelles composantes qui sont visibles sur (a) et (b) ne peuvent pas être reconstruite à partir de cette seule composante. Notez que le grand écart entre les deux pas de temps exact t_0 et t_1 de cet exemple a été volontairement choisi afin de montrer à la fois les possibilités et les limites de notre algorithme.

La propagation temporelle est ainsi d'autant plus efficace que la constante Δt est faible ou le facteur de subdivision K est grand, et ce du fait d'un aplanissement provenant de l'interpolation des champs scalaires. Si l'espacement entre deux pas de temps est trop grand, la propagation temporelle peut éventuellement ne pas être en mesure de reconstruire toutes les composantes connexes de l'isosurface souhaitée. Cependant, ceci est corrigé lors du passage par un pas de temps exact.

Néanmoins, les isosurfaces interpolées en vert de la figure 1.37 démontrent l'intérêt de notre approche. En augmentant virtuellement le nombre de pas de temps, la perception de l'évolution du champ de pressions dans le réservoir est accrue, ce qui permet de mieux appréhender les changements qui interviennent.

Discussion sur la validité de l'approche par propagation et interpolation linéaire

Le fait d'interpoler linéairement des propriétés entre deux pas de temps consécutifs pose le problème de la validité d'une telle interpolation. En effet, rien ne nous dit que les propriétés étudiées –provenant par exemple de simulations très complexes– évoluent de manière linéaire dans le temps. En pratique, nous avons constaté que dans la majorité des cas, les propriétés physiques que nous avons analysées sont suffisamment lisses et les pas de temps exacts suffisamment rapprochés pour pouvoir approximer les pas de temps intermédiaires par interpolation linéaire.

1.4.5 Bilan et perspectives

Afin de mieux comprendre et étudier des phénomènes physiques, il est nécessaire de disposer de méthodes de visualisation qui prennent en compte la dimension temporelle de ceux-ci. Le stockage des valeurs des propriétés étudiées pour de nombreux pas de temps pose dès lors un problème de stockage. En pratique, seul un nombre restreint de pas de temps est conservé. Du fait de ce nombre limité, l'analyse d'un tel jeu de données à l'aide d'isosurfaces est relativement délicate.

La méthode de morphing 4D présentée ici permet d'améliorer la perception de l'évolution de nos données en autorisant la génération d'une infinité d'isosurfaces pour des temps arbitraires. Dans le but d'accélérer l'extraction d'isosurfaces pour des pas de temps interpolés, l'algorithme développé utilise une propagation dite temporelle. Cette méthode se propage à partir de l'ensemble des cellules intersectées par l'isosurface au temps t pour déterminer celles intersectées au temps $t + \delta t$. La propagation se fait en fonction de valeurs interpolées linéairement à la volée, à partir des deux pas de temps disponibles qui encadrent le temps souhaité.

Dans l'optique d'avoir les meilleures performances possibles, notre algorithme d'extraction d'isosurfaces sur GPU a été étendu afin de prendre en charge matériellement l'interpolation linéaire des champs scalaires.

La méthode souffre tout de même de certains inconvénients. Elle nécessite de disposer d'une notion de voisinage entre cellules et ne garantit pas de reconstruire l'ensemble des composantes connexes d'une isosurface. De plus l'utilisation d'une interpolation linéaire n'est pas forcément toujours suffisante.

Nous avons montré que le coût de l'interpolation restait tout à fait raisonnable et que l'utilisation de la propagation temporelle permettait des facteurs d'accélération –comparés à un parcours exhaustif– compris entre 1.5x et 8.5x selon les cas.

En pratique, notre méthode permet d'augmenter considérablement la perception de l'évolution des données au cours du temps. Ses performances permettent une bonne interactivité même sur de gros jeux de données, justement là où un parcours exhaustif est insuffisant. La méthode offre donc un bon compromis entre qualité et rapidité d'extraction.

Au final, notre algorithme permet d'exploiter la cohérence spatiale, temporelle et dans l'espace des valeurs des données étudiées. L'exploitation de la cohérence dans l'espace des valeurs provient de l'utilisation de notre structure de classification qui subdivise l'espace des valeurs : les Bucket Search adaptatifs.

A l'instar de notre combinaison classification sur CPU/extraction sur GPU présentée à la section 1.3, notre Morphing 4D gagnerait grandement en performances à être implanté de manière asynchrone afin d'exploiter en parallèle les puissances de calcul CPU et GPU. Il serait également intéressant d'explorer la possibilité d'utiliser d'autres types d'interpolations que l'interpolation linéaire utilisée ici.

1.5 Conclusion et perspectives

Les techniques de visualisation interviennent dans de nombreux domaines, tels la visualisation de résultats de simulation d'écoulements fluides dans un réservoir pétrolier, les simulations météorologiques, l'imagerie médicale, ou même les jeux vidéo. Elles interviennent dans les domaines qui génèrent de grands jeux de données volumiques pour lesquels il est vital de définir des techniques à la fois performantes et qui permettent d'extraire un maximum d'information dans un but d'analyse. Notre objectif principal était d'avoir une chaîne complète de visualisation qui soit à la fois flexible et performante.

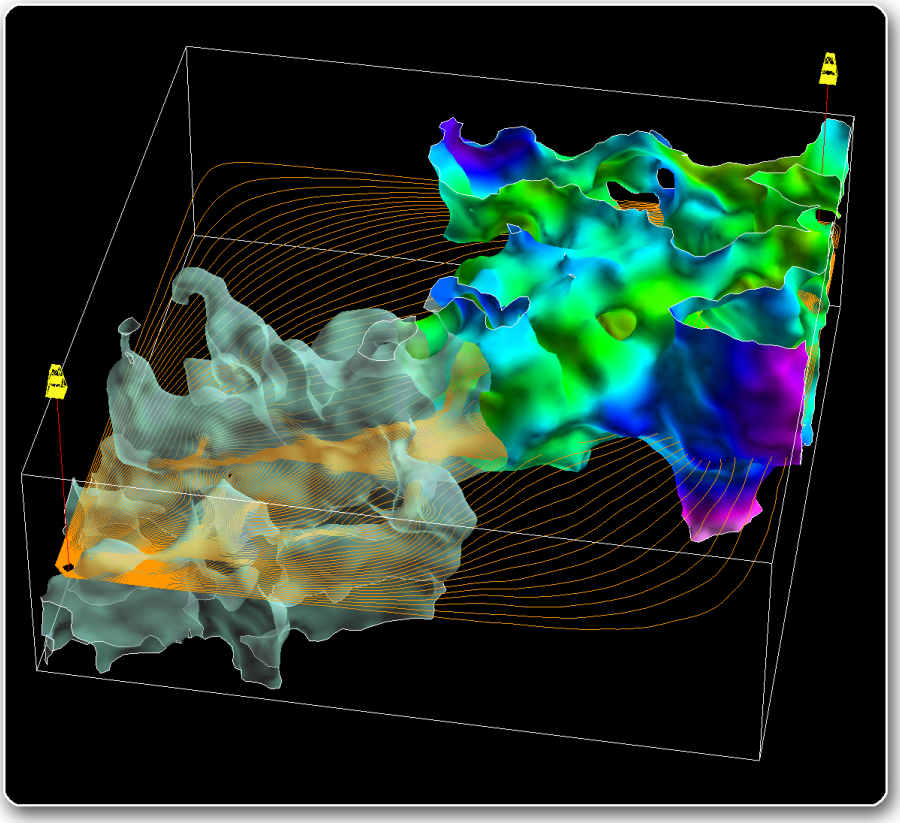
Nous avons présenté dans un premier temps les principaux algorithmes de visualisation disponibles dans la littérature, à savoir la visualisation dite volumique et la visualisation par extraction d'isosurfaces. Nous avons choisi d'étudier plus particulièrement l'extraction d'isosurfaces car elle s'avère la plus intuitive pour l'utilisateur. Les grilles qui sont générées dans de nombreux domaines, et qui servent de support à un stockage des informations, peuvent être, comme nous l'avons vu, de types très différents. La visualisation de grilles fortement non-structurées, c'est-à-dire à géométrie et topologie variables, a plus particulièrement retenu notre attention (figure 1).

Dans la section 1.2 nous avons étudié comment extraire efficacement une isosurface à partir de cellules fortement non-structurées grâce à l'utilisation d'un GPU, ce qui a résulté dans la définition de notre méthode appelée le Marching Cells. Cette méthode combine les performances d'extraction des méthodes à base de tables d'index comme l'algorithme du Marching Cubes, la flexibilité des méthodes qui utilisent des liens topologiques, et la puissance de calcul parallèle des GPU modernes. Notre algorithme sur GPU, grâce à l'utilisation de textures comme moyen de stockage, permet entre autre, contrairement aux approches précédentes, de traiter tout type de cellules, d'éviter toute redondance dans le stockage des données, de supporter de très grandes grilles composées de plusieurs millions de cellules, de limiter les transferts sur le bus PCI-Express, le tout en proposant des performances de tout premier plan. Comparé à une extraction sur CPU, notre algorithme sur GPU est 7 fois plus rapide pour des grilles tétraédriques, et 5.5 fois pour des grilles hexaédriques. Une évolution majeure du Marching Cells serait d'utiliser les Geometry Shaders. Grâce à eux, notre algorithme pourrait gagner à la fois en efficacité et en simplicité. En effet, les Geometry Shaders permettraient de générer la géométrie de l'isosurface à extraire directement au niveau du GPU, ce qui limiterait à la fois la redondance des calculs et la bande-passante mémoire nécessaire. Ils permettraient également d'implanter plus aisément un moteur d'extraction de cellules fortement non-structurées qui soit totalement générique.

Comme nous l'avons vu, le nombre de cellules intersectées par une isosurfaces est en $\mathcal{O}(n^{2/3})$. En conséquence, traiter exhaustivement les cellules d'une grille revient à passer la majorité du temps à tester des cellules non-intersectées. C'est à ce niveau qu'interviennent les algorithmes de classification. Ceux-ci cherchent à pré-sélectionner les cellules à priori intersectées pour une isovaleur donnée, afin de n'extraire que celles-ci. La section 1.3 a présenté les principaux algorithmes connus à ce jour, ainsi que deux nouvelles méthodes adaptatives de classification. L'utilisation de nos méthodes de classifications en amont de notre extraction sur GPU, le Marching Cells, nous a permis d'accélérer ce dernier d'un facteur allant jusqu'à 5. L'utilisation simultanée d'une classification sur CPU et d'une extraction sur GPU permet de combiner toutes les puissances de calcul disponibles dans un PC. Malgré tout, avec l'implantation actuelle, lorsque le CPU est

occupé, le GPU attend et vice versa (on parle d'exécution synchrone). Par conséquent, il serait possible d'améliorer encore les performances globales en rendant asynchrones les calculs CPU et GPU. De même, il serait intéressant d'étudier de nouveaux algorithmes de classifications qui seraient implantables directement sur GPU, pourquoi pas à l'aide d'API telles que CUDA ou CTM (voir la section 2.3.4 pour plus de détails à propos de ces API).

Pour finir, la section 1.4 a étendu les algorithmes présentés en 1.2 et 1.3 à la visualisation dite en 4D, avec donc une composante temporelle supplémentaire. Cette section a présenté la méthode du Morphing 4D, qui permet d'interpoler efficacement en temps-réel des isosurfaces entre deux pas de temps disponibles, en exploitant la puissance du CPU pour déterminer les cellules intersectées et celle du GPU pour interpoler les données en fonction du temps requis et extraire l'isosurface correspondante. Grâce à cette technique, il est possible d'obtenir une visualisation continue dans le temps des déformations des isosurfaces. Cette avancée permet d'augmenter la perception de l'évolution des données à travers le temps tout en limitant la quantité de données à stocker et en assurant de bonnes performances. Néanmoins, de même que pour les algorithmes de classification, il serait possible d'améliorer les performances du Morphing 4D en désynchronisant les calculs effectués sur CPU de ceux effectués sur GPU. D'un point de vue qualitatif, il serait également intéressant de tester d'autres types d'interpolations que la simple interpolation linéaire que nous avons utilisé.



Chapitre 2

Calculs massivement parallèles sur GPU pour des maillages non-structurés

Sommaire

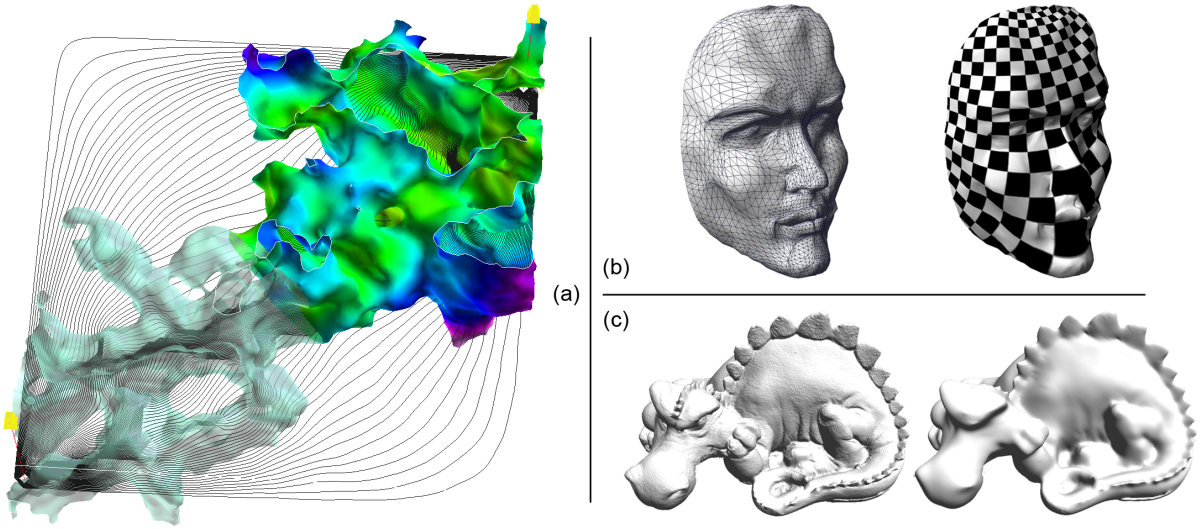


FIG. 2.1 – (a) Simulation d'écoulement fluide par lignes de courant (Fetel, 2007). (b) Paramétrisation de surface (Lévy *et al.*, 2002a). (c) Lissage de surface (Mallet, 1992).

2.1 Introduction

L'augmentation incessante de la puissance de calcul parallèle brute et de la bande-passante mémoire des GPU rend ceux-ci de plus en plus intéressants pour implanter des algorithmes hautement parallèles. Ceci est d'autant plus vrai depuis l'apparition d'API dédiées au calcul générique sur GPU (ou GPGPU) telles que celle d'AMD-ATI appelée CTM (Peercy *et al.*, 2006) pour Close-To-Metal ou bien celle d'NVIDIA appelée CUDA (Keane, 2006) pour Compute Unified Device Architecture. Ces nouvelles API ouvrent un accès direct de bas niveau aux nombreux processeurs de calcul parallèles des GPU ainsi qu'à leur mémoire (la section 2.3.4 donne plus de détails sur ces API).

Notre premier objectif était d'accélérer la résolution de problèmes d'optimisation liés à des grilles non-structurées. En général, la résolution de tels problèmes revient à résoudre un grand système d'équations linéaires qui s'avère être creux. Notre idée est donc d'implanter sur GPU un solveur numérique creux générique hautement-parallélisé afin d'accélérer la résolution de ces systèmes d'équations linéaires, l'ensemble étant basé sur une structure générique de matrice creuse. Ce solveur ne se limite donc pas à la résolution de quelques problèmes d'optimisation, mais à tout ceux qui nécessitent de résoudre un système linéaire (Mallet, 1992; Lévy *et al.*, 2002a; Sorkine et Cohen-Or, 2004; Nealen *et al.*, 2006; Floater et Hormann, 2005; Fetel, 2007). Nous allons ainsi montrer comment paralléliser efficacement un solveur numérique pour que celui-ci soit en mesure d'exploiter toute la puissance parallèle des GPU modernes. Afin de démontrer la viabilité de notre approche, nous avons testé notre solveur sur GPU pour calculer des paramétrisations de surfaces (Lévy *et al.*, 2002a), des lissages de surfaces (Mallet, 1992) ainsi que pour résoudre l'équation de pression simulant un écoulement fluide (Fetel, 2007) (figure 2.1).

Il est établit que les solveurs *directs* sont très efficaces pour résoudre certains problèmes d'optimisation tels que ceux rencontrés en traitement numérique de la géométrie (Botsch *et al.*, 2005a). En revanche, ceux-ci consomment beaucoup de mémoire et sont difficiles à implanter et à paralléliser. C'est pourquoi nous nous sommes focalisés sur des solveurs itératifs qui ont à leur avantage une consommation de mémoire limitée et une implantation et une parallélisation plus aisées. Notre algorithme, le *Concurrent Number Cruncher* (CNC), ou littéralement le *mangeur* de nombres parallèle, implante donc efficacement sur GPU à l'aide de la CTM et de CUDA un solveur itératif de type *Gradient Conjugué* préconditionné par *Jacobi* (Hestenes et Stiefel, 1952) basé sur une structure de matrice creuse par blocs de lignes compressées BCRS (Block Compressed Row Storage ; voir la section 2.3.1 et 2.4.2 pour plus de détails). Ce format de stockage est plus efficace que le format classique par lignes compressées CRS (Compressed Row Storage) car il permet d'appliquer des stratégies d'optimisations plus poussées (traitement par blocs de registres, *vectorisation*, etc.) qui réduisent à la fois la consommation en bande-passante mémoire et les temps de calcul (Barrett *et al.*, 1994).

Dans un premier temps (section 2.2), nous présentons trois problèmes d'optimisation dont nous souhaitons accélérer la résolution suivis d'un état de l'art sur les solveurs numériques et leur implantation sur GPU (section 2.3). Ensuite, à la section 2.4, nous présentons notre approche implantant un solveur numérique itératif générique creux sur GPU : le Concurrent Number Cruncher ou CNC (Buatois *et al.*, 2007b,c,a). Pour finir, nous analysons les performances de notre implantation sur les trois problèmes d'optimisation retenus (section 2.5).

2.2 Problématiques

2.2.1 Lissage de surfaces discrètes

La première application sur laquelle nous avons travaillé est celle du lissage de surfaces polygonales. Ce lissage de surface repose en fait sur une méthode d'interpolation générique appelée *interpolation lisse discrète* (ou DSI pour *Discrete Smooth Interpolation*) (Mallet, 1992). Cette technique, basée sur les moindres carrés, est conçue pour être capable d'intégrer des contraintes relativement variées dans le processus d'interpolation. Cette flexibilité permet d'appliquer cet interpolateur à de très nombreux cas pratiques, comme ici à du lissage de surface (figure 2.2). Cette section s'inspire des notes de cours *Paramétrisation de surfaces* SIGGRAPH 2007 (Hormann *et al.*, 2007) et du mémoire d'habilitation à diriger des recherches de Lévy (2008).

Cette section présente le principe de la formulation des moindres carrés, puis le principe de l'interpolateur lisse discret utilisé dans le but de lisser des surfaces. Pour plus de détails, le lecteur est invité à consulter Mallet (1992, 1997, 2002) ou encore Hormann *et al.* (2007) où sont développés les fondements théoriques de la méthode ainsi que Cognot (1996) et Muron *et al.* (2005) pour des précisions sur les différentes implantations de cette méthode.

Principe des moindres carrés

Supposons que nous souhaitions résoudre un système d'équations linéaires pour lequel le nombre d'équations m est supérieur au nombre d'inconnues n :

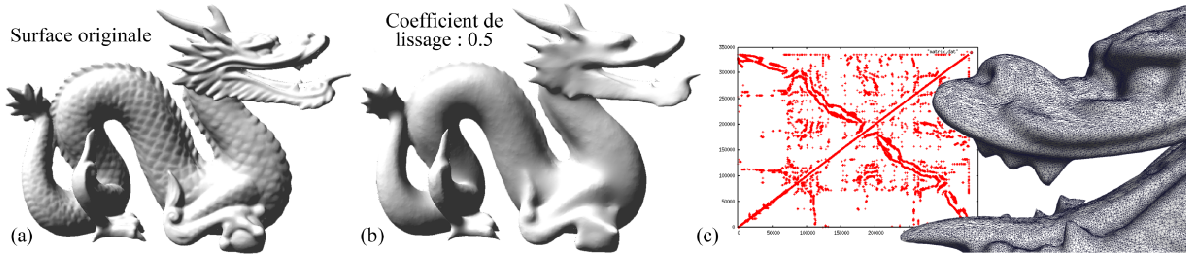


FIG. 2.2 – Surface lissée avec l'interpolateur DSI. (a) Surface originale. (b) Surface lissée avec un coefficient de 0.5. (c) Matrice creuse impliquée par le maillage non-structuré du modèle utilisé.

$$\begin{cases} a_{1,1}x_1 + \ldots + a_{1,n}x_n & = & b_1 \\ & \vdots & \\ a_{m,1}x_1 + \ldots + a_{m,n}x_n & = & b_m \end{cases}$$

soit $Ax = b$. Trois cas sont alors envisageables : il existe une infinité de solutions, il existe une seule et unique solution ou alors il n'existe pas de solution. Dans le cas général, lorsque le nombre d'équations m est, comme nous l'avons supposé, supérieur au nombre d'inconnues n, le système n'admet pas de solution. Malgré tout, il est possible de trouver une *moins mauvaise* solution en minimisant la somme des résidus au carré de chaque équation :

$$F(x) = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{i,j}x_j - b_i \right)^2 = \|Ax - b\|^2$$

Sachant que $\|x\|^2 = x^t x$, la forme quadratique F peut également s'écrire :

$$F(x) = \|Ax - b\|^2 = (Ax - b)^t(Ax - b) = x^t A^t Ax - b^t Ax - x^t A^t b + b^t b$$

Or par propriété de la transposée on a : $b^t Ax = (x^t A^t b)^t$. Comme $b^t Ax$ et $x^t A^t b$ sont des scalaires, on a $b^t Ax = x^t A^t b$ et :

$$F(x) = \|Ax - b\|^2 = x^t A^t Ax - 2x^t A^t b + b^t b$$

Sachant que $\nabla(b^t x) = \nabla(x^t b) = b$ et $\nabla(x^t Ax) = (A + A^t)x$, le gradient de F peut s'écrire sous la forme suivante :

$$\nabla F(x) = (A^t A + (A^t A)^t)x - 2A^t b$$

Or $A^t A$ est une matrice symétrique, d'où $(A^t A = (A^t A)^t)$ et :

$$\nabla F(x) = 2A^t Ax - 2A^t b$$

Le vecteur x qui minimise F est celui qui annule le gradient de la fonction F :

$$\nabla F(x) = 2A^tAx - 2A^tb = 0 \Leftrightarrow A^tAx = A^tb$$

Nous retrouvons ici la formule classique des moindres carrés, également appelé régression linéaire. Le vecteur x qui minimise la fonction F de départ est celui qui satisfait le système $A^tAx = A^tb$. Nous verrons à la section 2.3 et à la section 2.4 comment résoudre un tel système linéaire.

Moindres carrés avec suppression de degrés de libertés

Pour certains algorithmes, il peut être utile de restreindre les degrés de libertés du système en fixant certains paramètres de la fonction objectif. En termes formels, ceci revient à partitionner le vecteur de paramètres x en deux sous-vecteur $x = [x_l|x_f]$[13], dont les nl premières composantes $x_l = [x_1 \dots x_l]$ correspondent aux paramètres libres, et les $n - nl$ dernières composantes $x_f = [x_{nl+1} \dots x_n]$ correspondent aux paramètres fixés. La fonction F ne dépend à présent que du vecteur x_l, et s'exprime ainsi :

$$F(x_l) = \|Ax - b\|^2 = \left\| [A_l \mid A_f] \begin{bmatrix} x_l \\ \hline x_f \end{bmatrix} - b \right\|^2$$

Le partitionnement du vecteur x en deux sous-vecteurs x_l, x_f induit un partitionnement de la matrice A en deux sous-matrices A_l, A_f (dans le produit Ax, les coefficients de A pondérant des x_l sont dans A_l, et ceux qui pondèrent des x_f sont dans A_f). Il est alors possible de séparer les termes en x_l :

$$F(x_l) \quad = \quad \|A_lx_l + A_fx_f - b\|^2$$

En notant $b' = A_fx_f - b$ et en utilisant la formule des moindres carrés démontrée précédemment, nous trouvons l'équation satisfaite par la valeur de x_l qui minimise F :

$$A_l^tA_lx_l = A_l^tb' \quad \text{soit} \quad A_l^tA_lx_l = A_l^tb - A_l^tA_fx_f$$

Application au lissage de surfaces discrètes : la méthode DSI

Le lissage d'une surface peut être réalisé à l'aide de l'algorithme DSI fondé sur la méthode des moindres carrés détaillée précédemment, c'est ce que nous allons expliciter dans ce paragraphe.

Soit N_i la liste des sommets voisins du sommet i, $|N_i|$ le nombre de ses voisins et x_i sa position dans l'espace. Alors, pour chaque sommet i, une équation permet le lissage de la surface :

[13] La lettre l signifie libre et f fixé.

$$|N_i|xi = \sum_{j \in N_i} x_j$$

Le système formé par cette équation écrite pour l'ensemble des noeuds de la surface, peut être formulé au sens des moindres carrés. L'équation de résidu à minimiser est de la forme :

$$\forall i, G(x_i) = (\sum_{j \in N_i} x_j - |N_i|x_i)^2$$

Soit $x_i^{(0)}$ la position originelle du noeud i. Alors une seconde équation pour chaque noeud permet de contraindre arbitrairement l'ajustement des positions lissées par rapport aux positions originelles de chaque noeud i :

$$x_i = x_i^{(0)}$$

Même si le système formé à partir de cette seconde équation pris seul n'a pas besoin d'être réellement résolu, sont résidu peut être considéré, ce qui donne pour chaque sommet i :

$$\forall i, H(x_i) = (x_i - x_i^{(0)})^2$$

L'objectif final de la méthode est de minimiser grâce aux moindres carrés simultanément les deux fonctions G et H (tour à tour et indépendamment pour chaque axe du repère utilisé) afin d'assurer à la fois un bon lissage et une bonne correspondance des noeuds avec leur position originelle. Ceci est donc réalisé en minimisant la fonction F suivante :

$$F(x) = G(x) + \omega.H(x)$$

L'introduction d'un coefficient ω permet de normaliser les termes des équations de H afin d'équilibrer leurs poids par rapport aux termes des équations de G. Le coefficient ω permet ainsi de paramétrer la force du lissage relativement au respect de la position originelle des noeuds de la surface.

Bien entendu, il est tout à fait possible d'empêcher certains noeuds de la surface de bouger, par exemple les noeuds du bord, en ajoutant des contraintes fixes sur ceux-ci tout en appliquant la méthode de réduction des degrés de libertés explicitée précédemment.

Dans tous les cas, lisser une surface implique de résoudre un système linéaire de grande taille relativement creux. L'organisation des coefficients non-nuls de la matrice correspondante dépend directement du type de grille utilisé. Dans le cas général, donc pour des grilles non-structurées, cette organisation ne présente pas de motif particulier ce qui impose d'utiliser des structures de stockage des matrices creuses totalement génériques.

2.2.2 Paramétrisation de surfaces triangulées

La deuxième application abordée dans ce mémoire est celle de la *paramétrisation* de surfaces triangulées (parfois appelée dépliage de surfaces). Une paramétrisation d'une surface 3D est

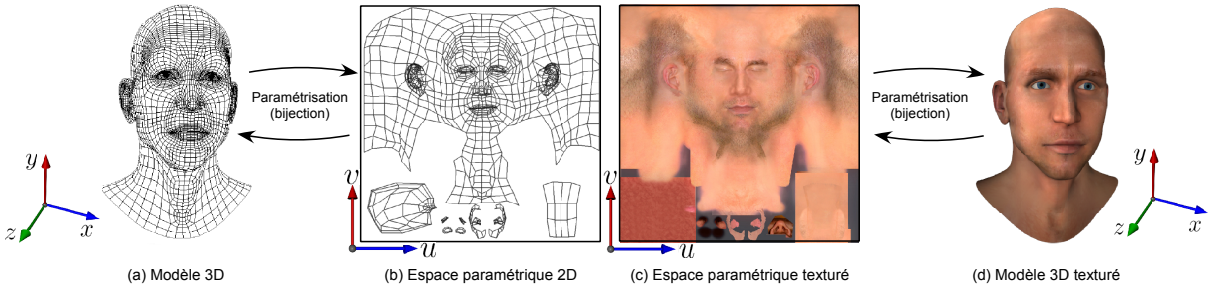


FIG. 2.3 – La paramétrisation établit une bijection entre le modèle surfacique 3D (a) et le modèle déplié dans l'espace paramétrique en 2D (b). Cette bijection permet d'associer chaque point du modèle 3D un unique point lui correspondant dans l'espace paramétrique et inversement. Il est alors possible de peindre une texture (c) de l'espace paramétrique sur le modèle surfacique 3D (d) grâce à cette paramétrisation. [Modèle disponible sur Maxon.net]

une fonction qui établit une correspondance bijective entre cette surface et un domaine 2D (figure 2.3). Cette notion joue un rôle fondamental en géométrie numérique, car elle permet de transformer des problèmes dans un espace 3D difficiles en des problèmes dans un espace paramétrique 2D beaucoup plus simples à résoudre.

La principale difficulté réside dans le choix de la manière de déplier la surface considérée, étant donné que celle-ci conditionne les déformations que va subir le maillage lors du passage de l'espace 3D à l'espace 2D. De nombreuses méthodes ont été développées (Hormann *et al.*, 2007) afin de minimiser les déformations subies par la surface lors du calcul de sa paramétrisation. Nous avons décidé d'utiliser dans nos tests la méthode de paramétrisation des *cartes conformes au sens des moindres carrés* ou LSCM (Least Squares Conformal Maps) (Lévy *et al.*, 2002b). Les prochains paragraphes présentent la méthode LSCM en s'inspirant des notes de cours SIGGRAPH 2007 (Hormann *et al.*, 2007) et du mémoire d'habilitation à diriger des recherches de Lévy (2008).

Définitions des repères 3D/2D et du gradient

Considérons un unique triangle de la surface à paramétriser. Il est alors possible d'attacher un repère orthonormé (X, Y) au plan support du triangle afin de référencer les coordonnées des sommets de ce dernier par uniquement deux coordonnées X_i et Y_i. Dans le repère paramétrique orthonormé (u, v), les coordonnées 2D des sommets du triangle sont dénotées par u_i et v_i. La figure 2.4 présente les différents repères utilisés.

Dès lors, il est possible de calculer ∇u et ∇v, les gradients des coordonnées paramétriques u et v relativement au système de coordonnées (X, Y) attaché au repère du triangle, soit pour u :

$$\nabla u = \begin{pmatrix} \partial u/\partial X \\ \partial u/\partial Y \end{pmatrix}$$

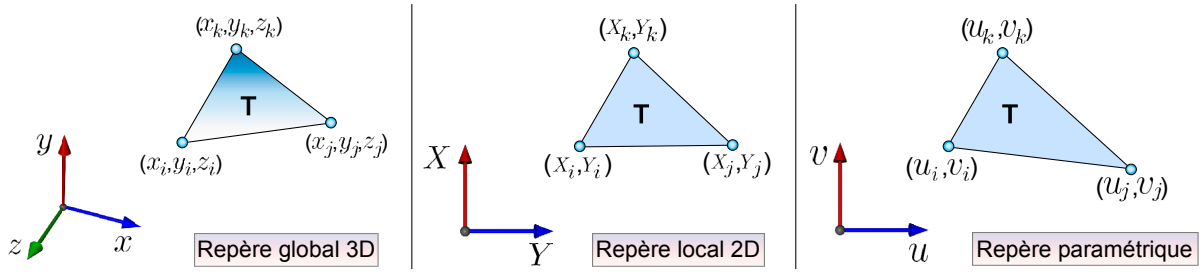


FIG. 2.4 – Le triangle T est représenté dans le repère global 3D (x, y, z), dans son repère local 2D (X, Y), puis enfin dans l'espace paramétrique (u, v).

Il est alors aisé de monter que (Hormann *et al.*, 2007) :

$$\nabla u = \frac{1}{2A_T} \begin{pmatrix} Y_j - Y_k & Y_k - Y_i & Y_i - Y_j \\ X_k - X_j & X_i - X_k & X_j - X_i \end{pmatrix} \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix}$$

où A_T est l'aire du triangle T considéré. Posons :

$$M_T = \frac{1}{2A_T} \begin{pmatrix} Y_j - Y_k & Y_k - Y_i & Y_i - Y_j \\ X_k - X_j & X_i - X_k & X_j - X_i \end{pmatrix}$$

alors M_T dénote la matrice qui permet de calculer le gradient d'une application linéaire définie sur un triangle en fonction des trois valeurs à ses sommets.

Cartes conformes au sens des moindres carrés (LSCM)

La paramétrisation par cartes conformes au sens des moindres carrés cherche à minimiser la déformation globale des triangles en essayant de conserver au maximum l'orthogonalité entre le gradient selon u et celui selon v (figure 2.5). Cette propriété peut alors s'écrire :

$$\nabla v = \mathrm{rot}_{90}(\nabla u) = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \nabla u$$

où rot_{90} dénote la rotation de 90 degrés dans le sens trigonométrique. Dans notre cas de surface triangulée munie d'une paramétrisation linéaire par morceaux, seules les surfaces développables admettent une paramétrisation conforme. Ces surfaces développables sont caractérisées par le fait que pour tout sommet i interne, la somme des angles incidents au sommet i est égale à 2π. Pour une surface générale (à savoir non développable), le principe de LSCM est de minimiser la dernière équation au sens des moindres carrés pour l'ensemble des triangles de la surface. Ceci s'écrit donc sous la forme d'une *énergie* E_{LSCM} qui mesure le caractère non-conforme de la paramétrisation et qui doit par conséquent être minimisée :

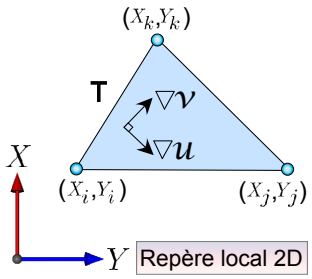


FIG. 2.5 – Dans un triangle muni d'une base locale (X, Y), la condition caractérisant les paramétrisations conformes s'exprime relativement facilement.

$$E_{LSCM} = \sum_{T=(i,j,k)} A_T \|\nabla v_T - \mathrm{rot}_{90}(\nabla u_T)\|^2$$

où l'aire A_T du triangle sert de coefficient de pondération. Cette dernière équation peut s'écrire alors :

$$E_{LSCM} = \sum_{T=(i,j,k)} A_T \left\| M_T \begin{pmatrix} v_i \\ v_j \\ v_k \end{pmatrix} - \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} M_T \begin{pmatrix} u_i \\ u_j \\ u_k \end{pmatrix} \right\|^2$$

L'énergie E_{LSCM} est une forme quadratique dont le minimum est susceptible d'être facile à trouver, puisque, par exemple, écraser tous les sommets en un même point revient à annuler la matrice M_T. En outre, il est important de noter que la conformité d'une paramétrisation est invariante par similitude dans l'espace paramétrique. Ceci permet d'affirmer que la forme quadratique E_{LSCM} n'est pas définie positive et que sa matrice est non-inversible. Plus concrètement, ceci signifie que la paramétrisation n'est pas contrainte dans l'espace puisqu'aucun des 4 degrés de liberté d'une similitude[14] n'est fixé. Néanmoins, ce problème est résolu dans la méthode LSCM en fixant simplement deux sommets dans l'espace paramétrique grâce à la méthode de suppression de degrés de liberté précédemment décrite à la section 2.2.1. En effet, ceci revient à contraindre les 4 degrés de liberté d'une similitude.

Comme nous l'avons précédemment démontré à la section 2.2.1, minimiser l'énergie E_{LSCM}, et donc calculer les coordonnées paramétriques des sommets correspondants à cette énergie, revient à résoudre un système linéaire à la fois potentiellement de grande taille et très creux. Nous verrons à la section 2.3 et à la section 2.4 comment résoudre de tels systèmes linéaires.

2.2.3 Simulation d'écoulement fluide biphasique sur lignes de courant

La troisième et dernière application abordée dans ce mémoire porte sur la simulation d'écoulement biphasique (mélange huile-eau) en milieu poreux sur lignes de courant à l'échelle d'un réservoir pétrolier. L'objectif est de simuler les écoulements des fluides (huile et eau) dans un

[14] deux degrés pour la translation, un pour la rotation et un pour l'homothétie.

modèle de réservoir pétrolier pour lequel nous disposons de *puits* qui injectent de l'eau avec un débit ou une pression imposée tandis que d'autres servent à récupérer l'huile. Cette partie est inspirée par le travail de Fetel (2007).

Équations fondamentales

Comme la plupart des processus physiques, le comportement d'un fluide peut être décrit par une ou plusieurs équations de conservation. Dans le cas d'écoulements en milieu poreux, il s'agit de l'équation de conservation de la masse. Celle-ci spécifie que, pour un volume de contrôle donné, la masse de matière échangée avec l'extérieur doit être égale à celle s'accumulant à l'intérieur. Considérons un système biphasique (huile-eau), des fluides non-miscibles et incompressibles ainsi que l'absence de pression capillaire, alors la conservation de la masse se traduit par les trois équations aux dérivées partielles suivantes (Aziz et Settari, 1979; Ertekin *et al.*, 2001; Dake, 2001) :

1. *L'équation de pression* :

$$\nabla \cdot \mathbf{u}_t = -\nabla \cdot \mathbf{K} \cdot (\lambda_t \ \nabla P + \lambda_g \ \nabla D) = q_s \tag{2.1}$$

où $\mathbf{u}_t$ (en $m.s^{-1}$) est la vitesse totale dite de Darcy (c'est-à-dire, la vitesse moyenne du mélange huile-eau dans le milieu poreux), $\mathbf{K}$ (en m^2) le tenseur de perméabilité intrinsèque de la roche, P (en Pa), la pression totale des fluides, D (en m) l'altitude, q_s (en $m^3.s^{-1}$) une éventuelle injection ou production de fluide, et enfin, λ_t (en $Pa^{-1}.s^{-1}$) et λ_g (en $m^{-1}.s^{-1}$) sont, respectivement, la mobilité totale et la mobilité gravitaire calculées à partir des propriétés physiques des fluides, telles que :

$$\lambda_t = \frac{k_{rw}}{\mu_w} + \frac{k_{ro}}{\mu_o} \quad \text{et} \quad \lambda_g = \frac{k_{rw}\rho_w g}{\mu_w} + \frac{k_{ro}\rho_o g}{\mu_o} \tag{2.2}$$

où, g (en $m.s^{-2}$) est l'accélération due à la gravité, et pour une phase donnée, $i \in [o, w]$ [15], k_{ri} représente la perméabilité relative[16], μ_i (en $Pa.s$) la viscosité dynamique et ρ_i ($kg.m^{-3}$) la masse volumique.

2. *L'équation de saturation en eau* (aussi appelée équation de Buckley-Leverett) :

$$\phi \frac{\partial S_w}{\partial t} + \mathbf{u}_t \cdot \nabla f_w + \nabla \cdot \mathbf{g}_w = f_{w,s} \ q_s \tag{2.3}$$

où S_w est la saturation en eau, c'est-à-dire la proportion du volume poreux remplie d'eau, ϕ la porosité[17] effective du milieu, f_w est la fraction du flux liée à l'eau, égale à :

$$f_w \ = \ \frac{kr_w/\mu_w}{kr_w/\mu_w + kr_o/\mu_o} \tag{2.4}$$

[15] Les indices o et w correspondent respectivement à l'huile (oil) et à l'eau (water).

[16] La perméabilité relative représente l'évolution de la perméabilité du milieu poreux à un fluide donné en fonction de sa saturation.

[17] Rapport du volume des vides du milieu au volume total.

et $\mathbf{g}_w$ est le flux d'eau dû à la gravité :

$$\mathbf{g}_w = \mathbf{K} \cdot \left(g f_w \frac{kr_o \ (\rho_o - \rho_w)}{\mu_o} \nabla D \right) \tag{2.5}$$

3. Et, enfin, *la définition de la saturation* :

$$S_o + S_w = 1 \tag{2.6}$$

Les équations (2.1) à (2.6) forment un système où les inconnues sont la pression P et les saturations en fluide S_w et S_o. Les autres variables sont considérées connues et doivent être définies dans la description du problème. Bien que ce système soit formellement composé d'équations fortement couplées, il est important de noter qu'elles ont des comportements mathématiques et numériques différents ce qui permet de les résoudre séparément. Cette considération est la base de la formulation IMPES (pour IMplicite en Pression Explicite en Saturation) et donc de la méthode de résolution sur lignes de courant. Le champ de pression est d'abord estimé en premier implicitement afin de pouvoir calculer les lignes de courant, puis dans un second temps, l'évolution de la distribution de saturation est estimée explicitement le long des lignes de courant. Pour plus de détails sur la formulation de ces équations, le lecteur est invité à se reporter, par exemple, à Aziz et Settari (1979) pour des simulations conventionnelles ou Thiele *et al.* (1994) et Batycky *et al.* (1997) pour des simulations sur lignes de courant.

Méthodologie

Le principe de la simulation d'écoulement sur lignes de courant est de découpler un problème complexe tridimensionnel sous forme d'un ensemble de sous-problèmes unidimensionnels. Pratiquement, cela consiste à simuler le déplacement de particules fluides le long de lignes de courant définies comme étant, en tout point du domaine d'étude, tangentes au champ de vitesses. Dans le détail, la chaîne d'opérations nécessaires pour réaliser une simulation d'écoulement sur lignes de courant peut être résumée de la façon suivante :

1. Résolution de l'équation de pression et calcul du champ de vitesse dans la grille tridimensionnelle.

2. Tracé des lignes de courant à partir du champ de vitesse.

3. Transfert de l'état du front de saturation et des vitesses de la grille vers les lignes de courant.

4. Résolution de l'équation de saturation le long des lignes de courant.

5. Transfert de l'état du nouveau front de saturation des lignes de courant vers la grille.

6. Post processus, le cas échéant, pour simuler des phénomènes n'agissant pas le long des lignes de courant, comme par exemple la gravité (Blunt *et al.*, 1996) ou la pression capillaire (Berenblyum *et al.*, 2004), etc.

7. Enfin retour à l'étape 1 si les conditions d'arrêt de la simulation ne sont pas atteintes. Ces conditions peuvent être soit une durée globale de simulation soit le non respect de contraintes numériques (par exemple, non respect de la loi de conservation de la masse, etc.).

Résolution numérique

Les étapes présentées ci-dessus sont résolues à l'aide de différentes méthodes spécifiques (Fetel, 2007). Comme nous allons le montrer dans le paragraphe suivant, résoudre l'équation de pression revient à résoudre un grand système linéaire. Le calcul du champ de vitesse à partir de la pression est à la fois direct et très rapide. Le tracé des lignes de courant à partir du champ de vitesse est réalisé efficacement à l'aide de la méthode de Runge-Kutta (Abramowitz et Stegun, 1972) ou de Pollock (Chiang et Kinzelbach, 2001; Batycky *et al.*, 1996) suivant le type de maillage utilisé. Pour finir, la résolution de l'équation de saturation le long des lignes de courant peut être réalisée de manière analytique, ce qui garanti de très bonnes performances.

Au final, dans une simulation d'écoulement en milieu biphasique, l'étape la plus consommatrice en temps de calcul est celle de la résolution de l'équation de pression. C'est pourquoi nous nous sommes focalisés sur la résolution de celle-ci.

Résolution de l'équation de pression La résolution de l'équation 2.1 aux dérivées partielles correspondant à la pression est réalisée à l'aide de la méthode des volumes finis. Cette approche repose sur le strict respect de l'équation de conservation de la masse sur un volume de contrôle. Ce respect est l'un des éléments clefs lors d'une simulation d'écoulement.

La discrétisation de l'équation de pression sur un maillage arbitraire transforme, en chaque noeud du maillage, l'équation aux dérivées partielles en une équation algébrique faisant intervenir la valeur, inconnue, de la pression au noeud et celle de ses voisins. Exprimé sur l'ensemble du maillage, ce processus conduit à un système linéaire qu'il faut résoudre en fonction de l'état initial et de conditions aux limites (pression ou débits aux puits d'injection etc.). Ce système s'écrit sous la forme matricielle suivante :

$$\mathbf{T}\ \mathbf{p} = \mathbf{b} \tag{2.7}$$

où

1. $\mathbf{T}$ est une matrice contenant les *transmissibilités*[18], que ce soit entre cellules voisines ou entre les cellules traversées par un puits et celui-ci. Dans le cadre de la formulation IMPES, cette matrice est carrée, symétrique, définie positive et a pour taille le nombre de cellules de la grille plus le nombre de puits ouverts et contrôlés en débit ;

2. $\mathbf{p}$ est un vecteur contenant les pressions inconnues, c'est-à-dire celles dans toutes les cellules de la grille ainsi que celles des puits contrôlés en débit ;

3. enfin, $\mathbf{b}$ est un vecteur contenant les termes dus à la gravité et les conditions imposées aux puits (qu'ils soient contrôlés en débit ou en pression) et aux frontières du réservoir.

[18]Capacité à permettre un écoulement fluide au travers d'une face d'une cellule du maillage.

L'équation (2.7) représente donc un système linéaire de très grande taille relativement creux. L'organisation des coefficients non-nuls de la matrice correspondante dépend directement du type de grille utilisé. Dans le cas général, donc pour des grilles non-structurées, cette organisation ne présente pas de motif particulier ce qui impose d'utiliser des structures de stockage des matrices creuses totalement génériques. Les sections 2.3 et 2.4 suivantes montrent comment résoudre de tels systèmes linéaires.

2.3 Les solveurs numériques et leur implantation sur GPU : État de l'art

Comme nous venons de le voir, un grand nombre de problèmes reposent sur la résolution d'un grand système linéaire du type $Ax = b$, A et b étant connu et x étant le vecteur solution recherché. En fonction du problème et de sa discrétisation (grilles structurées ou non), résoudre des problèmes d'optimisation –comme ceux que nous avons présentés à la section précédente– met en jeux différents types de matrices : denses, creuses par bandes ou creuses génériques. Bien entendu, chaque type de matrice implique d'utiliser des structures mémoire spécifiquement optimisées.

Dans le cas des grilles non-structurées, qui représentent le cas le plus générique, les systèmes linéaires construits sont de très grandes tailles, très creux, et ne présentent aucun motif aisément identifiable. Il est donc nécessaire d'utiliser des structures de matrices creuses à la fois génériques (pouvant stocker un motif arbitraire) et hautement-optimisées pour les stocker.

La résolution de systèmes linéaires peut être effectuée à l'aide de différents solveurs numériques qui sont habituellement regroupés en deux familles : les solveurs itératifs et les solveurs directs. Ces solveurs sont systématiquement basés sur des librairies de fonctions qui implantent des opérations basiques d'algèbre linéaire sur des matrices et des vecteurs (ou opérations *BLAS* en anglais pour Basic Linear Algebra Subprograms). Bien entendu, chaque type de matrice et de solveur implique une implantation spécifique sur un GPU. Une partie de cette section s'inspire des notes de cours SIGGRAPH 2007 de Hormann *et al.* (2007).

2.3.1 Solveurs itératifs

Algorithmes classiques

Relaxation La méthode de relaxation est la plus simple, à la fois du point de vue conceptuel, et du point de vue de l'implantation. Cette méthode a beaucoup été utilisée dans les années 90 pour implanter des algorithmes de traitement numérique de la géométrie. Elle a plus tard été remplacée par des méthodes plus sophistiquées, évoquées plus loin.

La méthode de relaxation peut se comprendre facilement en considérant le problème à résoudre $Ax = b$ comme un système linéaire :

$$\begin{cases} a_{1,1}x_1 & +a_{1,2}x_2 & +\dots & +a_{1,n}x_n & = & b_1 \\ & & \vdots & & & \\ a_{i,1}x_1 & +a_{i,2}x_2 & +\dots & +a_{i,n}x_n & = & b_i \\ & & \vdots & & & \\ a_{n,1}x_1 & +a_{n,2}x_2 & +\dots & +a_{n,n}x_n & = & b_n \end{cases}$$

La méthode consiste alors à parcourir les équations une par une, et à calculer pour l'équation i la valeur de x_i obtenue en faisant "comme si" tous les autres x_j pour $j \neq i$ étaient connus, ce qui donne le schéma de mise à jour suivant pour la valeur x_i :

$$x_i \leftarrow \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j \right)$$

L'algorithme de résolution de système linéaire par la méthode de relaxation s'écrit alors :

Algorithme 18 : Résolution par relaxation

tant que ($\|Ax - b\| < \epsilon$) {
  pour i de 1 à n {
    $x_i \leftarrow \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j \right)$
  }
}

avec ϵ la précision souhaitée par l'utilisateur. Il est également possible de spécifier un nombre d'itération maximum, afin d'arrêter l'algorithme lorsqu'il ne parvient pas à converger.

Comme nous pouvons l'observer, cet algorithme ne peut pas s'appliquer à des matrices ayant des valeurs nulles sur la diagonale. D'une manière plus générale, il est possible de démontrer une condition suffisante de convergence de l'algorithme : si la matrice est diagonale dominante, à savoir :

$$\forall i, |a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$$

alors l'algorithme converge.

Il est possible d'accélérer la méthode de relaxation, en utilisant le fait que la mise à jour effectuée pour chaque variable x_i "allait dans la bonne direction". Intuitivement, en allant "un peu plus loin" dans cette direction, à savoir en multipliant le déplacement par un facteur ω, il sera possible d'accélérer la convergence. Le schéma de mise à jour modifié s'écrit alors :

$$\begin{array}{lcl} x_{prev} & \leftarrow & x_i \\ x_i & \leftarrow & \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j \right) \\ x_i & \leftarrow & x_{prev} + \omega (x_i - x_{prev}) \end{array}$$

Ce schéma calcule tout d'abord la mise à jour de x_i comme précédement, puis augmente le déplacement effectué par rapport à l'ancienne valeur x_{prev} d'un facteur ω. Il est possible de démontrer que l'algorithme converge sous les mêmes conditions que pour la relaxation (matrice A diagonale dominante), et pour $\omega \in [1, 2[$. L'algorithme ainsi modifié est connu sous le nom de *sur-relaxations successives* (ou SOR pour *successive over-relaxation* dans la littérature anglophone). En ré-écrivant sous une forme plus compacte le schéma de mise à jour, l'algorithme SOR s'écrit alors de la manière suivante :

Algorithme 19 : Résolution par sur-relaxations successives (SOR)

tant que ($\|Ax - b\| < \epsilon$) {
  pour i de 1 à n {
    $x_i \leftarrow (1 - \omega)x_i + \frac{\omega}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j \right)$
  }
}

Le choix optimum du paramètre ω est déterminé par les valeurs propres de A. Comme calculer ces valeurs propres est en général plus difficile que de résoudre le système linéaire, le paramètre ω est en général déterminé soit par une étude théorique des valeurs propres (pour des problèmes particuliers), soit de manière empirique.

Le principal avantage de la méthode SOR est sa grande simplicité d'implantation. Cette simplicité a favorisé son utilisation dans la communauté "traitement numérique de la géométrie" (Taubin, 1995). Toutefois, comme nous le verrons par la suite, des méthodes plus sophistiquées, telles que la méthode du Gradient Conjugué, permettent de résoudre plus efficacement ce type de problèmes.

Le Gradient Conjugué L'algorithme du Gradient Conjugué (Hestenes et Stiefel, 1952) est bien plus efficace, et à peine plus compliqué à implanter que celui de la relaxation. Cette méthode de résolution de systèmes symétriques se fonde sur l'équivalence entre la résolution du système $Ax = b$ et la minimisation de la forme quadratique $F(x) = 1/2x^tAx - b^tx$ (cf. paragraphe sur les moindres carrés de la section 2.2.1). De manière plus précise, l'algorithme calcule une base de vecteurs orthogonaux dans l'espace de la matrice (à savoir une base de vecteurs conjugués), en appliquant l'algorithme d'orthogonalisation de Schmidt. Les calculs se simplifient de manière remarquable, et permettent de calculer les vecteurs un par un, en ne gardant en mémoire que le dernier vecteur. Le suivant est alors obtenu sous forme d'une combinaison linéaire entre le précédent et le gradient de F au point courant. L'algorithme complet s'écrit de la manière suivante :

Algorithme 20 : Gradient Conjugué pré-conditionné

$i \leftarrow 0$; $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{Ax}$; $\mathbf{d} \leftarrow \mathbf{M}^{-1}\mathbf{r}$;
$\delta_{new} \leftarrow \mathbf{r}^T\mathbf{d}$; $\delta_0 \leftarrow \delta_{new}$;
tant que $i < i_{max}$ **et** $\delta_{new} > \epsilon^2\delta_0$
    $\mathbf{q} \leftarrow \mathbf{Ad}$; $\alpha \leftarrow \frac{\delta_{new}}{\mathbf{d}^T\mathbf{q}}$;
    $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{d}$; $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{q}$;
    $\mathbf{s} \leftarrow \mathbf{M}^{-1}\mathbf{r}$; $\delta_{old} \leftarrow \delta_{new}$;
    $\delta_{new} \leftarrow \mathbf{r}^T\mathbf{s}$; $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$;
    $\mathbf{d} \leftarrow \mathbf{r} + \beta\mathbf{d}$; $i \leftarrow i + 1$;
fin

Dans cet algorithme, A et b sont des données connues, x est le vecteur solution du système $Ax = b$ recherché, i_{max} est le nombre maximum d'itérations, ϵ la précision souhaitée et la matrice M est appelée un préconditionneur. Ce dernier permet d'améliorer la vitesse de convergence de l'algorithme. Le préconditioneur de Jacobi définit M comme la matrice composée des éléments diagonaux de A. C'est le préconditionneur que nous utilisons dans notre CNC. Comme nous pouvons le remarquer, les seules opérations effectuées par cet algorithme, mis à part de simples opérations sur des vecteurs, sont des produits matrice-vecteur. Ainsi, il est possible d'implanter l'algorithme uniquement à l'aide d'opérations d'algèbre linéaire sur des vecteurs et des matrices –soit des opérations BLAS.

La méthode du Gradient Conjugué ne peut s'appliquer qu'à des matrices symétriques. Dans le cas où la matrice A n'est pas symétrique, il est possible de dériver du système $Ax = b$ un système symétrique équivalent, de la manière suivante :

$$\begin{pmatrix} Id & A \\ A^t & 0 \end{pmatrix} \begin{pmatrix} 0 \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}$$

Il est alors possible d'appliquer la méthode du Gradient Conjugué à ce système. Ceci définit l'algorithme du *Gradient bi-Conjugué*, ou encore biCG. Une variante nommée biCGSTAB (Gradient bi-Conjugué stabilisé) permet d'améliorer la vitesse de convergence en stabilisant les calculs.

Le lecteur souhaitant en apprendre plus sur la théorie du Gradient Conjugué pourra se référer à l'excellent texte de Shewchuk (1994) sur le sujet, ainsi qu'à celui de Barrett *et al.* (1994) pour ce qui est des extensions du Gradient Conjugué aux matrices non-symétriques.

Notre CNC implante, quant à lui, un Gradient Conjugué préconditionné par la méthode de Jacobi pour de grands systèmes linéaires accéléré sur GPU à l'aide des API CUDA de NVIDIA et CTM d'AMD-ATI.

Matrices denses et par bandes sur GPU

Les premiers solveurs numériques sur GPU furent des solveurs itératifs fonctionnant uniquement sur des matrices denses ou au mieux par bandes (Krüger et Westermann, 2003b). Ceci est bien entendu dû au fait que les matrices denses ou par bandes sont facilement et efficacement

représentables en mémoire à l'aide de textures 2D. De fait, ces solveurs limitaient leur champ d'application à des classes de problèmes très spécifiques, utilisant par exemple des grilles régulières. La plupart des travaux dans le domaine ont cherché à résoudre la pression selon l'équation de Poisson lors d'une simulation d'écoulement fluide dans un cas incompressible, le tout à base de textures. Krüger et Westermann (2003b) ont résolu cette équation à l'aide d'un Gradient Conjugué. Ce dernier est basé sur des opérations d'algèbre linéaire BLAS sur des vecteurs et des matrices par bandes implantées sur GPU.

Matrices creuses génériques sur GPU

La discrétisation d'équations aux dérivées partielles sur des grilles irrégulières mène à résoudre des problèmes irréguliers. Dans ce cas précis, les matrices misent en jeu sont particulièrement creuses tout en présentant un remplissage très irrégulier. Deux articles ont montré la faisabilité de l'implantation de structures de matrices creuses simples sur GPU (Bolz *et al.*, 2003; Krüger et Westermann, 2003b). Ils ont à chaque fois implanté une structure de matrice creuse de type par lignes compressées ou Compressed Row Storage (CRS) en anglais (voir la section 2.4.1 pour plus de détails sur cette structure de stockage). Bolz *et al.* (2003) ont ainsi utilisé des textures pour stocker les coefficients non-nuls de la matrice creuse et la table d'index à deux niveaux associée. La table d'index est utilisée pour accéder aux coefficients et trier les lignes de la matrice en fonction du nombre de coefficients non-nuls dans chacune d'entre elles. Grâce à cette table, une itération est exécutée sur le GPU simultanément pour chaque groupe de lignes de la matrice de taille identique, dans le but de réaliser par exemple un produit matrice/vecteur. Bolz et al. ont réussi avec succès à implanter un Gradient Conjugué ainsi qu'un solveur dit multi-grille. Les performances étaient à l'époque honorables, mais souffraient de la complexité ajouté par l'implantation sur GPU. La seconde approche proposée cette fois-ci par Krüger et Westermann (2003b), implante les matrices creuses de type CRS à l'aide de vertex buffers, chaque coefficient non-nul étant représenté par un *sommet* envoyé à la carte graphique. Krüger et Westermann ont également implanté un Gradient Conjugué sur GPU à l'aide d'opérations d'algèbre linéaire de type BLAS. Malheureusement, l'utilisation de *sommets* pour représenter les coefficients non-nuls de la matrice creuse charge fortement la carte graphique ce qui, au final, ne permet pas d'obtenir des performances significativement supérieures à une implantation sur CPU optimisée.

Notre CNC implante également sur GPU la structure de matrice creuse de type CRS, mais utilise des structures mémoire beaucoup plus compactes que dans les approches précédentes. En outre, le CNC implante un format amélioré de stockage qui est dérivé du format CRS et qui est présenté à la section 2.4.2 : le format *BCRS* (Barrett *et al.*, 1994), pour Block Compressed Row Storage (stockage par blocs de lignes compressées). A titre de comparaison, le CNC implante également le format de matrice creuse dite par bandes.

2.3.2 Solveurs directs

Algorithmes classiques

Une méthode possible pour résoudre un système linéaire consiste à factoriser la matrice, sous forme d'un produit de matrices simples à inverser. Par exemple, la factorisation LU (Press *et al.*, 1992) consiste à trouver la matrice triangulaire inférieure L et la matrice triangulaire supérieure U dont le produit est égal à la matrice M du système (si la matrice est symétrique, U correspond à la transposée de L). Une fois cette factorisation obtenue, il devient très facile de résoudre des systèmes linéaires impliquant $M = LU$, de la manière suivante :

$$LUx = b \quad \rightarrow \begin{cases} Lx_1 &= b \\ Ux &= x_1 \end{cases}$$

Ainsi, l'algorithme résout le système linéaire en résolvant deux systèmes triangulaires (par substitutions successives). D'une manière conceptuelle, la méthode du pivot de Gauss, enseignée à l'école pour résoudre des systèmes linéaires à la main, correspond à cet algorithme (exprimé d'une manière légèrement différente).

Dans le cas de matrices creuses, différentes méthodes permettent de calculer les facteurs L et U représentés également par des matrices creuses. Différentes librairies, disponibles sur Internet en OpenSource, implantent ces algorithmes complexes, à savoir SuperLU, TAUCS, MUMPS, UMFPACK. Dans le cas particulier où la matrice M est symétrique définie positive, la méthode directe de décomposition de Cholesky peut s'appliquer. Elle consiste en une décomposition similaire à une décomposition LU mais du type $M = LL^t$ où L est une matrice triangulaire inférieure. Comme l'ont remarqué Botsch *et al.* (2005b), ces solveurs directs sont particulièrement efficaces pour des problèmes de géométrie numérique.

Matrices denses sur GPU

Des solveurs directs qui utilisent la décomposition de Cholesky (Jung et O'Leary, 2006) ou bien le pivot de Gauss et la factorisation LU (Galoppo *et al.*, 2005) ont déjà été implanté sur GPU pour des matrices denses. Leur efficacité relativement aux implantations sur CPU a déjà été prouvé, même si les facteurs d'accélération obtenus restent souvent limités.

Matrices creuses génériques sur GPU

L'implantation d'un solveur direct sur GPU qui nécessite une structure de matrice creuses est un problème très difficile. En effet, les solveurs directs impliquent d'avoir une structure de matrice creuse qui soit dynamique, ce qui les rend intrinsèquement inadaptés aux GPU. De plus, encore une fois de part leur nature même, la parallélisation de ces algorithmes est une tâche très ardue. A notre connaissance, aucune implantation efficace sur GPU n'est disponible à ce jour.

2.3.3 Bilan sur les méthodes de résolution de systèmes linéaires

TAB. 2.1 – Bilan sur les méthodes de résolution de systèmes linéaires

Méthode	Avantages	Inconvénients	Implantation sur GPU
relaxation - SOR	-très facile à implanter -faible coût mémoire	-pas très efficace	-non-disponible
Gradient Conjugué	-facile à implanter -faible coût mémoire	-efficacité moyenne	-disponible pour des matrices denses ou par bandes
méthodes directes	-les plus efficaces -codes publics disponibles	-grand coût mémoire -algorithmes très complexes	-non-disponibles

Nous concluons cette sous-section par un bilan rapide de ces méthodes de résolution de systèmes linéaires (table 2.1). En résumé, les méthodes de type SOR ont l'avantage d'être extrêmement faciles à implanter, mais ont du mal à converger pour des maillages comportant plus de dix mille sommets, elles n'ont pas été implanté sur GPU à notre connaissance. Les méthodes de type Gradient Conjugué réalisent un bon compromis entre efficacité et difficulté d'implantation. Des implantations sur GPU sont disponibles, mais ne sont efficaces que sur des matrices denses ou par bandes. Les méthodes directes creuses sont les plus rapides, mais ont parfois l'inconvénient de consommer une très grande quantité de mémoire. Celles-ci ont été portées sur GPU uniquement pour des matrices denses, jamais pour des matrices creuses. Enfin, il est intéressant de noter que des méthodes directes creuses en mémoire externe sont apparues récemment (Meshar *et al.*, 2006). Elles permettent de bénéficier de l'efficacité des méthodes directes, sans présenter le risque de dépasser les ressources RAM disponible. En revanche, elles présentent les mêmes difficultés d'implantation sur GPU que les méthodes directes classiques.

2.3.4 Nouvelles API, nouvelles possibilités : *CUDA* et *CTM*

Précédemment, faire du calcul générique sur GPU (on parle souvent de General-Purpose Computation Using Graphics Hardware, également appelé GPGPU[19]) nécessitait de détourner l'utilisation normale des API graphiques standard comme DirectX (Microsoft Corporation, 2006) ou bien encore OpenGL (Segal et Akeley, 2004). Les programmes exécutables sur GPU étaient alors écrits dans différents langages tels que Brook (Buck *et al.*, 2004), Sh (McCool et DuToit, 2004), Cg (Fernando et Kilgard, 2003), GLSL (Rost, 2004) ou bien encore HLSL (Microsoft Corporation, 2002). Malheureusement, l'utilisation de ces modèles de programmation focalisés sur le rendu graphique temps-réel limite la flexibilité, les performances, et les possibilités des GPU modernes en terme de GPGPU car ils imposent de passer par l'ensemble des étapes de rendu du pipeline graphique (section 1.2.1, figure 1.1) pour réaliser le moindre calcul, ce qui la plupart du temps est inutile. Par exemple, les performances peuvent être fortement limitées par des fonctions graphiques inefficaces comme l'était la fonction d'échange de *pBuffer* sur les premières NVIDIA GeForce FX. Cette fonction d'échange était à l'époque limité à seulement 200 échanges par seconde (Bolz *et al.*, 2003).

[19] www.gpgpu.org

Depuis peu, les deux grands concepteurs mondiaux de cartes graphiques, AMD-ATI et NVIDIA, ont publié des API totalement dédiées au GPGPU, à savoir respectivement les API *Close-To-Metal* (CTM) et *Compute Unified Device Architecture* (CUDA). Ces deux API permettent d'accéder directement à la mémoire graphique très rapide ainsi qu'aux processeurs de calculs parallèles très puissants des cartes graphiques. Ces nouvelles API se substituent au pipeline graphique qui est finalement bien inutile dans le cadre d'une utilisation en GPGPU.

La mémoire graphique est exposée directement grâce à des pointeurs mémoire fonctionnant de manière similaire à ceux sur CPU. CTM et CUDA fournissent les fonctions permettant de copier des données depuis la mémoire CPU sur la mémoire GPU et inversement. Les processeurs de calcul parallèle sont programmables en langage assembleur pour ce qui est de la CTM et dans un langage proche du C grâce à un compilateur spécifique pour ce qui est de CUDA. L'exécution de code utilisateur sur le GPU est relativement aisée avec ces API. Pour simplifier, cela revient à appeler les bonnes fonctions des API en utilisant les bons pointeurs mémoire comme paramètres. Que cela soit avec CTM ou CUDA, les GPU sont exposés comme des *multi-processeurs* de calcul travaillant en parallèle (on parle parfois de multiple-pipelines). Ces multi-processeurs sont ainsi capables d'exécuter un très grand nombre de *threads*[20] simultanément et indépendamment. La difficulté d'implantation des algorithmes sur GPU réside donc majoritairement dans l'efficacité de leur parallélisation, et donc dans la maximisation de l'utilisation des multi-processeurs parallèles de calcul. Dans tous les cas, les API telles que CTM ou CUDA permettent de simplifier le modèle de programmation tout en optimisant les performances globales, notamment grâce à des surcoûts d'exécution sur le GPU grandement réduits. En plus de l'utilisation de structures de données optimisées, nous utilisons à la fois CTM et CUDA afin d'obtenir les meilleures performances possibles.

2.3.5 Contributions

Le *Concurrent Number Cruncher* (CNC), est un Gradient Conjugué préconditionné haute-performances sur GPU qui tire parti des dernières API disponibles, CUDA de NVIDIA et CTM d'AMD-ATI. Ces API sont dédiées au calcul générique sur processeur graphique. Elles permettent de maximiser les performances de calcul grâce à des mécanismes d'optimisation spécifiques. Notre CNC se base sur une implantation générique optimisée de structures de matrices creuses sur GPU utilisant le format par bloc de lignes compressées ou BCRS pour Block Compressed Row Storage. Cette implantation prend en charge différentes tailles de blocs et permet d'effectuer efficacement des opérations d'algèbre linéaire grâce à une parallélisation massive, une stratégie dite de vectorisation et l'utilisation de techniques dites de *register blocking*. Les sections qui suivent décrivent toutes ces notions plus en détails.

A notre connaissance, le CNC est le premier solveur numérique sur GPU qui soit capable d'efficacement résoudre des problèmes d'optimisation non-structurés, c'est-à-dire qui soit capable d'utiliser des matrices creuses génériques sur GPU.

[20]Sur CPU, un thread, parfois appelé *processus léger*, est une sorte de processus à l'intérieur d'un processus qui peut exécuter ses propres instructions. Chaque thread possède son propre environnement d'exécution (valeurs des registres du processeur) ainsi que sa propre pile. En revanche, les threads appartenant à un même processus partagent la même mémoire virtuelle. Sur GPU, cette notion de thread se transpose de manière relativement similaire, à la différence près que la communication entre threads est, lorsqu'elle est possible, souvent limitée.

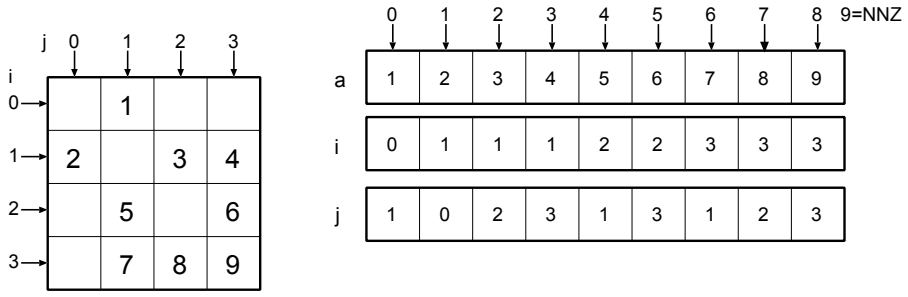


FIG. 2.6 – Exemple de matrice creuse stockée en format TRIAD

2.4 Algèbre linéaire sur GPU et matrices creuses génériques : Le Concurrent Number Cruncher (CNC)

Notre méthode, appelée le Concurrent Number Cruncher (CNC), se base sur deux composants : une API qui sert à construire le système linéaire (par exemple OpenNL (Lévy, 2005) pour les applications de lissage de surface ou de calcul de paramétrisation et StreamLab (Fetel, 2007) pour les simulations d'écoulement fluide), et une implantation hautement-performante d'opérations d'algèbre linéaire (BLAS) sur GPU. Les sections suivantes présentent quelques structures de données bien connues dans le domaine du calcul haute-performance et qui sont utilisées dans le CNC. Ensuite une section présentera comment nous avons optimisé au mieux ces opérations pour des GPU récents, et enfin une dernière section présentera quelques points techniques fondamentaux.

2.4.1 Structures de matrices creuses usuelles en calcul haute-performance et produit matrice creuse/vecteur (SpMV)

Cette section présente des structures usuelles de stockage de matrices creuses. Notez que cette section est inspirée par les notes de cours SIGGRAPH 2007 de Hormann *et al.* (2007).

Une première approche : le format de stockage TRIAD

Algorithme 21 : Structure de donnée TRIAD

```
struct TRIADMatrix {
        int N     ; // dimension de la matrice
        int NNZ   ; // nombre de coefficients non nuls
        float * a ; // coefficients non nuls (tableau de taille NNZ)
        int * I   ; // indices de lignes (tableau de taille NNZ)
        int * J   ; // indices de colonnes (tableau de taille NNZ)
} ;
```

Algorithme 22 : Produit matrice TRIAD × vecteur (SpMV)

```
void mult ( float * y, TRIADMatrix * M, float * x ) {
   for ( int i=0; i<M->N; i++ ) {
      y[i] = 0.0 ;
   }
   for ( int k=0; k<M->NNZ; k++ ) {
      y[M->I[k]] += M->a[k] * x[M->J[k]] ;
   }
}
```

Pour optimiser le stockage des matrices creuses, la première idée qui peut venir à l'esprit consiste à ne stocker que les entrées non-nulles de la matrice ainsi que les indices (i, j) associés. Ainsi, on ne stocke que NNZ coefficients (NNZ : Number of Non-Zero à savoir nombre d'entrées non-nulles) ainsi que les indices associés. Ce mode de stockage de matrices creuses, intitulé *format TRIAD*, est illustré sur la figure 2.6, et une implantation en langage C est donnée à titre d'exemple (algorithme 21). Nous donnons également la fonction *mult* (fréquemment appelée SpMV pour produit Sparse Matrix-Vector en anglais) permettant de calculer le produit entre une matrice stockée au format TRIAD et un vecteur (algorithme 22). Comme nous l'avons vu à la section 2.3.1, le calcul de produits matrice-vecteur est par exemple particulièrement utile pour implanter un solveur fondé sur la méthode du Gradient Conjugué.

Bien que le mode de stockage TRIAD permette de ne pas utiliser inutilement de l'espace de stockage pour les entrées non-nulles, il occasionne une forte redondance des données dans le stockage des coefficients i qui encodent le numéro de la ligne associé à chaque entrée. Pour cette raison, ce format reste peu utilisé en pratique dans les grandes libraries de résolution numérique (il se limite à la définition de formats de données, car sa grande simplicité en favorise la compréhension et l'utilisation). En pratique, les développeurs de grandes libraries de résolution numérique préfèrent utiliser le format CRS (Compressed Row Storage, pour stockage de lignes compressées), bien plus compact en mémoire. Nous détaillons ce dernier format dans la sous-section suivante.

Une approche plus compacte : le format de stockage par lignes compressées

Algorithme 23 : Structure de données CRS (Compressed Row Storage)

```
struct CRSMatrix {
        int N    ; // dimension de la matrice
        int NNZ   ; // nombre de coefficients non nuls
        float * a ; // coefficients non nuls (tableau de taille NNZ)
        int * index_colonne  ; // index de colonnes (tableau de taille NNZ)
        int * pointeur_ligne ; // pointeurs de lignes (tableau de taille N+1)
        float * diag   ; // éléments diagonaux (tableau de taille N)
} ;
```

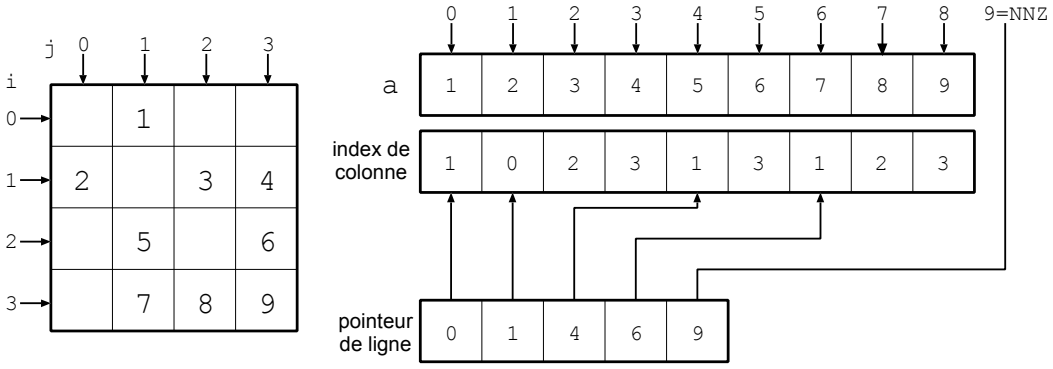


FIG. 2.7 – Exemple de matrice creuse stockée en format CRS (Compressed Row Storage)

Le format de stockage par lignes compressées, ou CRS pour Compressed Row Storage en anglais, est la représentation la plus commune pour les matrices creuses (Barrett *et al.*, 1994). Elle est largement utilisée par les grandes librairies de calcul scientifique. La variante transposée CCS (ou Compressed Column Storage pour stockage par colonnes compressées) se rencontre également (suivant les types d'algorithmes et choix d'implantation des librairies). Nous présentons et utilisons ici le format CRS. L'algorithme 23 décrit l'implantation (en langage C), et les figures 2.7 et 2.8 en montre deux exemples. La structure de données CRS utilise trois tableaux pour représenter les coefficients non-nuls et leurs indices. Le tableau `a` stocke l'ensemble des entrées non-nulles de la matrice, le tableau `index_colonne` indique pour chaque entrée l'index de colonne qui lui correspond. Les lignes sont encodées d'une manière différente, par le tableau `pointeur_ligne`, qui indique pour chaque ligne son début et sa fin dans les tableaux `a` et `index_colonne`. Afin de faciliter l'écriture des algorithmes (en évitant un test), il est d'usage de compléter le tableau `index_colonne` par une entrée supplémentaire, pointant une case plus loin que la dernière entrée de la matrice. Cette entrée est appelée communément une *sentinelle*. Nous verrons plus loin comment celle-ci facilite l'écriture de l'algorithme calculant le produit entre la matrice creuse et un vecteur donné (opération SpMV).

Algorithme 24 : Produit matrice CRS × vecteur (SpMV)

```
void mult ( float * y, CRSMatrix * M, float * x ) {
   for ( int i=0; i<M->N; i++ ) {
      y[i] = 0.0 ;
      for ( int j=M->pointeur_ligne[i]; j<M->pointeur_ligne[i+1]; j++ ) {
         y[i] += M->a[j] * x[M->index_colonne[j]] ;
      }
   }
}
```

L'algorithme 24 décrit comment implanter le calcul du produit entre une matrice creuse `M` stockée en format CRS et un vecteur donné `x`. Comme nous pouvons le voir, l'utilisation de la sentinelle `pointeur_ligne[N] = NNZ` permet de ne pas avoir de cas particulier pour traiter

la dernière ligne de la matrice. Le nombre d'opérations réalisées par un produit matrice creuse CRS/vecteur est de deux fois le nombre de coefficients non-nuls de la matrice creuse (Shewchuk, 1994). Avec une matrice dense de taille équivalente, il faudrait pas moins de $2n^2$ opérations. Le format CRS permet ainsi de fortement réduire le temps de calcul du produit matrice/vecteur.

Différentes variantes de la structure CRS existent. Ainsi, si la matrice est symétrique, il est possible d'économiser de l'espace de stockage en ne représentant que la moitié triangulaire inférieure de la matrice. Suivant les besoins, d'autres variantes stockent la diagonale dans un vecteur séparé, par exemple pour faciliter le calcul d'un préconditioneur de Jacobi (voir plus haut, section 2.3.1). Des variantes stockent les colonnes (CCS pour Compressed Column Storage), ce qui peut faciliter certains types de calculs. Finalement, d'autres variantes stockent des blocs de coefficients (format BCRS pour Bloc Compressed Row Storage), ce qui permet à la fois de diminuer la quantité mémoire utilisée, d'accélérer les accès à la mémoire et de favoriser l'utilisation des instructions vectorielles (jeux d'instructions SSE sur les processeurs Intel, ou toutes les instructions définies sur des vecteurs pour les cartes graphiques). Nous avons expérimenté ce dernier format pour implanter notre solveur sur GPU au sein du CNC.

2.4.2 Optimisations pour les GPU

Les GPU modernes sont des super-calculateurs massivement parallèles qui ont une architecture de type Simple Instruction sur de Multiples Données (SIMD). Cette architecture massivement parallèle rend spécifique toute implantation et toute optimisation d'algorithmes. Cette section présente comment exploiter au mieux les nombreux niveaux de parallélismes offerts par les GPU, au travers de leurs multiples pipelines de calcul, d'accès à la mémoire, de leur traitement parfois vectoriel etc.

De multiples pipelines de calcul : parallélisation

Lorsqu'un algorithme est parallélisable, les multiples pipelines de calcul que possèdent les GPU peuvent être exploités au mieux. Dans notre implantation, le calcul de $\mathbf{y} \leftarrow \mathbf{Ax}$ (SpMV) est réalisé en parallèle. Chaque élément du vecteur $\mathbf{y}$ est calculé indépendamment par un thread s'exécutant sur le processeur graphique. De cette manière, chaque thread parcourt une ligne de la matrice creuse $\mathbf{A}$ pour calculer le produit matriciel.

Afin de maximiser les performances et de masquer au mieux les latences d'accès à la mémoire, le nombre de thread doit être significativement supérieur au nombre de pipelines disponibles sur la carte graphique utilisée. Par exemple, sur une NVIDIA G80 possédant 128 pipelines, la taille de $\mathbf{y}$ doit être au moins un ordre de magnitude au dessus de 128, ce qui est souvent le cas en traitement numérique de la géométrie.

Les opérations sur des vecteurs peuvent en général être parallélisées de la même manière que les opérations sur des matrices. Par exemple, le calcul de $\mathbf{y} \leftarrow \alpha \times \mathbf{x} + \mathbf{y}$ (cette opération est nommée SAXPY suivant la bibliothèque BLAS standard) peut se faire composante par composante, chaque thread calculant un élément de $\mathbf{y}$.

La parallélisation d'un produit scalaire de deux vecteurs est un peu plus compliquée à réaliser. En effet, celle-ci s'apparente à une somme-réduction de l'ensemble des éléments des deux vecteurs

considérés aboutissant à un seul et unique élément. Cette opération est ainsi difficile à paralléliser efficacement. Dans le CNC, cette opération est implantée au travers d'une somme-réduction itérative des données comme le montre la figure 2.12. Sur les cartes d'AMD-ATI, la somme-réduction d'un vecteur est efficacement implantée comme dans Krüger et Westermann (2003b). À chaque itération, chaque thread lit et traite quatre valeurs scalaires puis écrit le seul scalaire réduit résultant. Le vecteur de taille originale n est ainsi réduit d'un facteur quatre à chaque itération de l'algorithme, et ce jusqu'à ce qu'un seul scalaire subsiste, après $\log_4(n)$ itérations. Nous utilisons une approche légèrement différente sur les cartes d'NVIDIA car celles-ci disposent de certaines fonctionnalités originales qui permettent des optimisations plus poussées. Nous utilisons une mémoire dite *partagée* (Keane, 2006) qui permet de réduire à chaque itération la taille du vecteur d'un facteur 512, donc bien supérieur à 4, et ce de manière très efficace. La section 2.4.3 donne plus de détails sur l'implantation de notre algorithme de somme-réduction sur les GPU d'AMD-ATI ainsi que ceux d'NVIDIA.

Traitement vectoriel

Certaines architectures (par exemple les AMD-ATI série X1k) possèdent des processeurs non pas scalaires mais vectoriels. Les opérations de bases autorisées sur ces GPU s'appliquent donc directement à des vecteurs de scalaires, plus exactement des quadruplets. Concrètement cela signifie que lire/écrire/traiter un scalaire ou quatre à la fois prend exactement le même temps. Il est donc fondamental, sur de telles architectures, de *vectoriser* au maximum les données dans des quadruplets afin de bénéficier d'un maximum de bande passante mémoire et d'un maximum de puissance de calcul parallèle.

Les derniers GPU d'NVIDIA (série 8) sont des processeurs scalaires, tout du moins pour ce qui est des calculs. Il n'est donc pas nécessaire de vectoriser les données lors de leur traitement. En revanche, pour ce qui est des accès aléatoires en lecture/écriture, il est beaucoup plus intéressant de lire un vecteur de 4 nombres plutôt que 4 nombres isolés puisque les séries 8 d'NVIDIA sont capables de lire et d'écrire 128bits de données en un seul cycle d'horloge et une seule instruction. Pour des accès contigus en mémoire graphique, ces GPU sont capables d'automatiquement vectoriser virtuellement la récupération des données afin de maximiser les performances.

Côté CPU, il est également possible d'utiliser des instructions SSE-1/2/3/4 pour bénéficier d'une vectorisation des accès mémoire et des opérations de calcul. Ces instructions introduisent une touche de parallélisme même sur des CPU mono-cores.

Dans l'optique d'obtenir les meilleures performances possibles quel que soit le GPU utilisé, notre implantation s'adapte à chaque architecture en vectorisant les données lorsque cela est utile.

Traitement par blocs : stockage par blocs de lignes compressées et blocs de registres

Nous avons vu précédemment que le format de stockage par lignes compressées (CRS) était un format à la fois compact et efficace. Mais il est possible de faire encore mieux en stockant non pas de simples coefficients indépendants, mais des blocs de coefficients. Ce format porte par

analogie le nom de stockage par blocs de lignes compressées, ou Block Compressed Row Storage (BCRS) en anglais.

Barrett *et al.* (1994) a prouvé l'efficacité sur CPU du format BCRS comparé au format CRS. C'est pourquoi nous l'avons utilisé dans notre CNC sur GPU, et ce contrairement aux approches précédentes qui se limitent systématiquement au format CRS (Bolz *et al.*, 2003; Krüger et Westermann, 2003b).

Le format BCRS regroupe des coefficients dans des blocs de taille $BN \times BM$. Le stockage dans des blocs permet d'exploiter au mieux la bande passante mémoire des GPU du fait d'une vectorisation plus naturelle des données. Elle permet également de tirer parti des registres disponibles sur les cartes graphiques afin d'éviter des accès redondants à la mémoire. Finalement, le format BCRS permet de minimiser le nombre d'indirections à résoudre du fait de la taille réduite des tables d'index. Nous avons essayé différentes tailles de bloc, et nous avons conservé les deux plus efficaces : 2 par 2 et 4 par 4 éléments par bloc.

Le calcul de la multiplication avec le vecteur x d'un bloc couvrant plusieurs lignes peut être optimisé en ne lisant qu'une seule fois les valeurs contenues dans x. Ces valeurs sont stockées dans des registres afin d'être réutilisées pour chaque ligne du bloc concerné. De cette manière, de nombreux accès mémoire redondants sont évités. Cette technique s'appelle le *Register Blocking* (Im et Yelick, 2001). La figure 2.8 illustre l'influence de la taille des blocs sur le nombre d'accès à la mémoire dans le cas particulier d'un produit matrice/vecteur. Puisque le pattern d'accès à la mémoire –dans notre cas d'étude– est aléatoire, lire et écrire des données 4 par 4 en mémoire est fondamental pour exploiter efficacement la bande passante mémoire disponible, et ce quel que soit l'architecture visée, scalaire ou vectorielle. Dans le cas d'un produit matrice/vecteur ($\mathbf{y} \leftarrow \mathbf{Ax}$), il est par conséquent très souhaitable de traiter les éléments de y consécutifs 4 par 4. Cela revient à utiliser des blocs de taille 4x4. Une fois la double indirection permettant de trouver un bloc résolue à l'aide des tables d'index définies par le format BCRS, quatre accès en lecture permettent de lire un bloc 4x4 de coefficients, un accès en lecture supplémentaire permet de lire les quatre scalaires du vecteur x correspondant au bloc traité, et enfin un accès en écriture renvoie les quatre éléments du résultat de la multiplication dans y. Les valeurs de x sont stockées dans des registres et réutilisées pour chaque ligne d'un bloc. C'est ce qui limite le nombre d'accès à la mémoire, les registres jouant le rôle d'un cache, et ce qui fait toute la différence de performance avec le format CRS.

Le stockage par bloc permet de réduire la taille des tables d'index comparé à un stockage coefficient par coefficient. Cette réduction de la taille des tables limite la consommation mémoire ainsi que le nombre d'indirections à résoudre lors d'une opération sur la matrice. Cela a d'autant plus d'impact sur les performances de l'algorithme que chaque indirection introduit un accès dépendant à la mémoire. Au final, cet accès dépendant introduit de grandes latences qui doivent être masquées au mieux par le GPU afin d'obtenir une bonne efficacité.

Même si de grandes tailles de blocs peuvent paraître optimales en terme d'utilisation des registres et de bande passante mémoire, ces tailles ne sont pas forcément adaptées à tous les types de matrices creuses. En effet, l'utilisation de grandes tailles de blocs nécessite d'avoir une matrice très compacte afin d'éviter d'avoir des blocs eux-même très creux (figure 2.8). Il est donc nécessaire de trouver un compromis entre l'efficacité des blocs de grandes tailles et un taux de

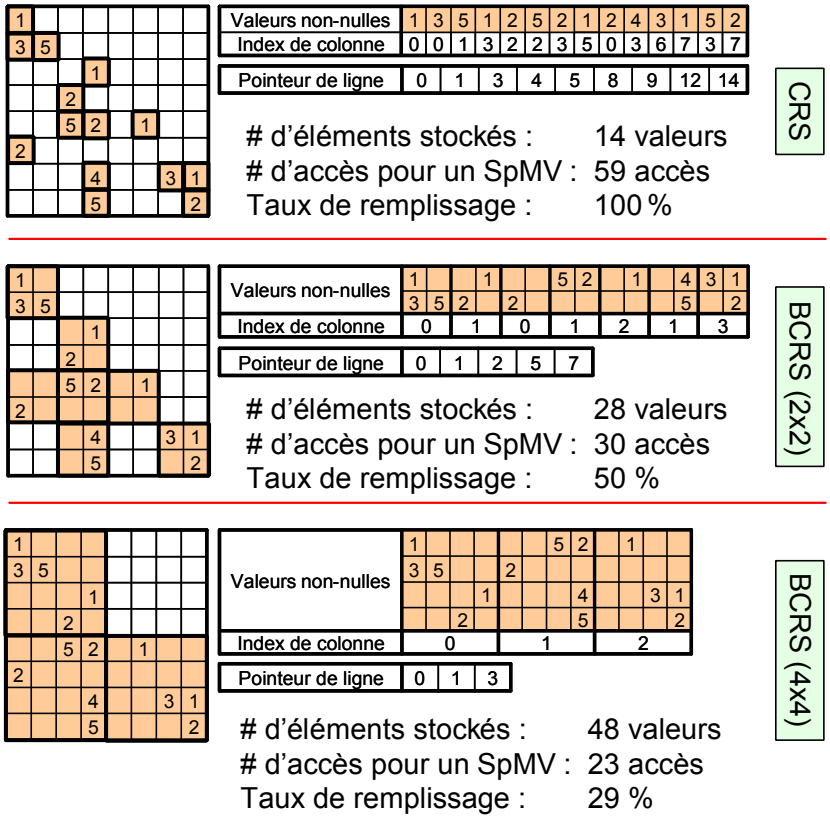


FIG. 2.8 – Exemples de stockages d'une matrice creuse selon le format par lignes compressées (CRS) et par blocs de lignes compressées (BCRS). Le nombre d'éléments stockés est égal au nombre de coefficients conservés dans la structure de matrice creuse, qu'ils soient utiles ou non. Le nombre d'accès à la mémoire correspond au nombre d'accès requis pour calculer un produit matrice/vecteur (SpMV) pour une architecture vectorielle en lecture/écriture. Plus le nombre d'accès est faible, plus la consommation en bande-passante mémoire est faible, et plus l'algorithme est efficace. Le taux de remplissage correspond au pourcentage moyen de coefficients non-nuls dans chaque bloc. Plus le taux de remplissage est élevé, moins on perd de temps à traiter des coefficients nuls.

remplissage raisonnable de ces blocs. Le CNC implante, en sus du format CRS, des blocs de taille 2x2 et 4x4 pour le format BCRS. Les petits blocs 2x2 sont utilisés dans les cas où les blocs 4x4 n'offrent pas un taux de remplissage suffisant. Dans le cas d'un produit matrice/vecteur (SpMV), les blocs de taille 2x2 sont optimaux au regard des accès mémoire en lecture des coefficients d'un bloc. En revanche, la lecture des coefficients de x et l'écriture du résultat dans y n'exploitent que la moitié de la bande passante disponible car ils n'utilisent que deux composantes sur les quatre disponibles.

La section 2.5 compare les performances de notre implantation sur GPU du format CRS, BCRS-2x2 et BCRS-4x4 avec des implantations sur CPU.

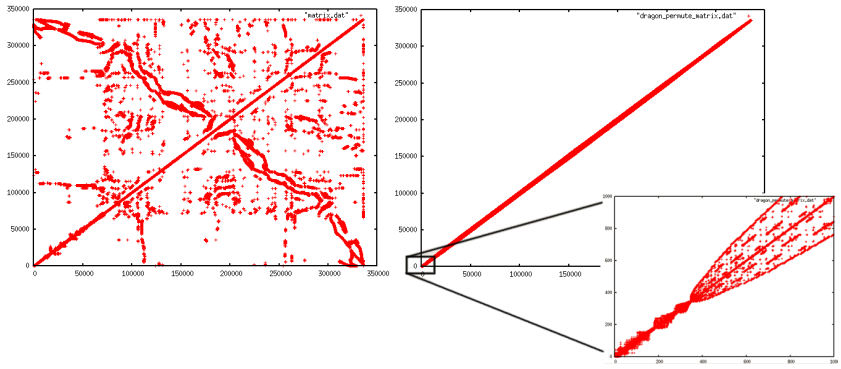


FIG. 2.9 – Matrice dans le cas d'un lissage de surface avant (gauche) et après (droite) réordonnancement selon l'heuristique Cuthill McKee inversée ou RCMK. Étonnamment, ce réordonnancement n'améliore pas les performances sur GPU, probablement du fait du schéma aléatoire d'adressage qui est fortement optimisé pour du plaquage de texture 2D.

Réordonnancement pour une meilleure efficacité du cache mémoire

Les techniques de réordonnancement par permutation de lignes et de colonnes sont très souvent utilisées lorsque qu'une matrice dense a ses lignes et colonnes numérotées en accord avec la numérotation des noeuds d'un maillage. L'utilisation du cache de données peut être optimisée en utilisant, par exemple, l'heuristique Cuthill McKee inversée (RCMK) (Cuthill et McKee, 1969; Gibbs *et al.*, 1974). Comme le montre la figure 2.9 le RCMK essaie de compresser les coefficients non-nuls d'une matrice autour de sa diagonale, ce qui en théorie augmente la probabilité d'utiliser la mémoire cache plutôt que la mémoire classique lors d'une lecture séquentielle des coefficients de la matrice. Nous avons testé un tel réordonnancement dans le CNC, mais tous nos tests ont montré que celui-ci n'avait qu'une influence très faible sur les performances globales que cela soit sur CPU ou sur GPU. Nous n'avons ainsi pas constaté d'améliorations nettes des performances. Sur GPU, ce fait est probablement dû à la stratégie de cache 2D spécifiquement adaptée au plaquage de texture et non pas au calcul numérique avec des accès séquentiels 1D. *In fine*, aucun réordonnancement n'est utilisé dans nos tests de performances de la section 2.5.

2.4.3 Points techniques

L'API d'AMD-ATI : Close-To-Metal (CTM)

L'API Close-To-Metal (CTM) d'AMD-ATI (Peercy *et al.*, 2006) est en fait un driver dédié aux cartes graphiques AMD-ATI pour les séries X1k et suivantes. Celui-ci offre un accès de bas niveau à la fois aux processeurs graphiques parallèles et à la mémoire graphique. La mémoire est exposée au travers de deux pointeurs, l'un sur la base de la zone adressable de la mémoire du GPU (accessible uniquement côté GPU), et l'autre sur la base de la mémoire dite PCI-Express (accessible à la fois côté GPU et côté CPU). La CTM ne propose pas de mécanisme d'allocation mémoire à proprement parler, seulement des pointeurs à la base des zone adressables. C'est donc à l'utilisateur de gérer la mémoire manuellement. La CTM fournit des fonctions de compilation

et de chargement de code écrit en langage assembleur, donc de très bas-niveau, exécutable sur le GPU, ainsi que des fonctions qui permettent de lier différents pointeurs mémoire en entrée comme en sortie des multi-processeurs de la carte graphique.

Notre CNC implante une surcouche de haut-niveau à la CTM qui masque totalement la complexité de la gestion de la mémoire. Ceci est réalisé grâce à l'implantation de mécanismes d'allocation dynamique de mémoire (fonctions malloc/free) pour les deux mémoires disponibles : GPU et PCI-Express. Dans le but d'optimiser l'utilisation du cache GPU, les pages mémoires allouées sont automatiquement alignées en mémoire graphique par nos fonctions d'allocation selon des règles spécifiques.

Le CNC fournit également une interface pour créer et manipuler des vecteurs et des matrices directement sur le GPU. Les vecteurs et matrices sont automatiquement téléchargés en mémoire graphique lors de leur instanciation. Ils sont alors directement prêt à être utilisés dans des opérations d'algèbre linéaire BLAS. Le CNC pré-compile et pré-alloue les codes en assembleur dans le but d'optimiser leur exécution. Pour exécuter une opération BLAS, le CNC lie les pointeurs mémoire GPU d'entrée/sortie nécessaires, et demande l'exécution du code assembleur souhaité.

L'API d'NVIDIA : Compute Unified Device Architecture (CUDA)

L'API CUDA (Keane, 2006) est l'équivalent chez NVIDIA de la CTM d'AMD-ATI. Elle tend à développer les mêmes idées mais d'une manière significativement différente. CUDA repose sur une librairie spécifique désormais intégrée aux drivers graphiques des cartes de génération 8 et supérieures. CUDA offre un accès bas niveau à la mémoire graphique (appelée mémoire globale) ainsi qu'aux unités de calcul parallèles génériques (appelées processeurs de flux par NVIDIA). L'API expose la mémoire graphique à l'aide de pointeurs alloués par des fonctions spécifiques de l'API et qui fonctionnent de manière analogue aux malloc et free du langage C. L'API permet également de compiler/charger/exécuter du code utilisateur (appelés *kernels*) sur les processeurs de flux. Ce code est écrit dans un langage de haut niveau, sorte de langage C étendu.

Sur la série 8 d'NVIDIA, une structure appelée multi-processeur regroupe huit processeurs de flux. Chaque multi-processeur peut exécuter un seul bloc de threads en même temps, bloc qui regroupe un nombre prédéfini par l'utilisateur de threads. Chaque thread d'un bloc peut alors accéder à une mémoire très rapide spéciale appelée *mémoire partagée* qui comme son nom l'indique est partagée entre tous les threads d'un même bloc. Il est par conséquent possible de faire communiquer des threads entre-eux grâce à cette zone mémoire partagée et l'aide du mécanisme de synchronisation des kernels fourni par CUDA. Ce mécanisme permet de synchroniser l'exécution de tous les threads d'un bloc aux lignes de codes souhaitées, et ce afin d'éviter par exemple des lectures impropres, des lectures non-reproductibles ou des pertes de mise-à-jour (Delmal, 2001). Cette mémoire partagée ne permet pas de communiquer entre threads de blocs différents, seulement entre threads d'un même bloc.

Notre CNC utilise CUDA de la même manière que CTM, afin de définir une couche de haut-niveau pour les GPU NVIDIA. Au final, le CNC propose une interface simple et flexible, grâce à l'utilisation de CTM et CUDA, pour effectuer des opérations d'algèbre linéaire BLAS et résoudre des systèmes linéaire à l'aide d'un Gradient Conjugué sur tout GPU moderne.

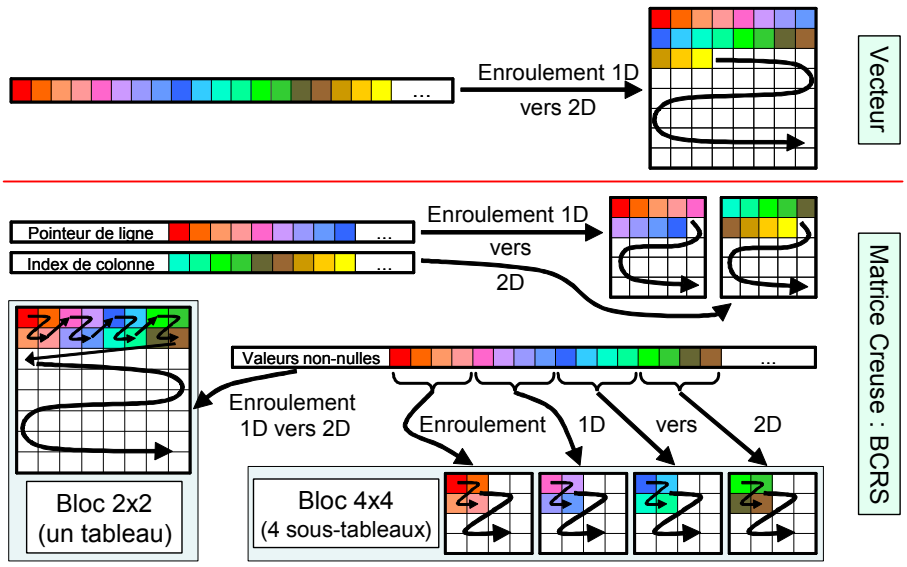


FIG. 2.10 – Stratégies de stockage par enroulement/découpage des vecteurs et des matrices creuses de type BCRS (2x2 et 4x4) sur des GPU à architecture vectorielle (par exemple la série X1k d'AMD-ATI).

Structure mémoire sur une architecture GPU vectorielle (AMD-ATI)

Sur les architectures vectorielles comme celle de la série X1k d'AMD-ATI, le CNC "enroule" les vecteurs dans des tableaux 2D. Cela permet d'optimiser au mieux l'accès au cache mémoire qui lui aussi est 2D. De surcroît, cette stratégie permet de stocker de très grands vecteurs dans un seul tableau (sur les AMD-ATI X1k, la taille maximum d'un tableau 1D est de 4096 quadruplets flottants 32bits, là où celle d'un tableau 2D est de 4096^2 quadruplets).

Le format BCRS utilise trois tables pour stocker une matrice creuse. La première stocke les indices de colonne, la seconde les pointeurs de ligne et la troisième les coefficients non-nuls de la matrice. Les indices de colonne et les pointeurs de ligne sont enroulés comme de simples vecteurs, et les coefficients non-nuls sont enroulés en fonction de la taille des blocs utilisés dans le format BCRS. Plus particulièrement, le CNC utilise des techniques spécifiques d'optimisation (appelées *strip mining* en anglais (Callahan *et al.*, 1990, 2004)) qui enroulent/découpent des données 1D dans des tableaux 2D dans le but, par exemple, de maximiser l'efficacité du cache. Ainsi, les données des blocs 2x2 sont enroulées bloc par bloc dans un seul tableau 2D de quadruplets (float4). Les données des blocs 4x4 sont quant à elles distribuées dans quatre sous-tableaux 2D remplis alternativement à partir de sous-blocs de taille 2x2 comme proposé par Fatahalian *et al.* (2004). Chaque accès en lecture permettant de lire quatre nombres flottants simultanément (soit un float4) sur un GPU, la lecture des 16 valeurs d'un bloc 4x4 se fait à l'aide de quatre accès en lecture à une même adresse mémoire dans les quatre sous-tableaux qui contiennent les données. Cette stratégie maximise l'utilisation de la bande-passante mémoire tout en minimisant les calculs de translation d'adresses. La figure 2.10 illustre la stratégie de stockage utilisée au sein du CNC pour des architectures vectorielles.

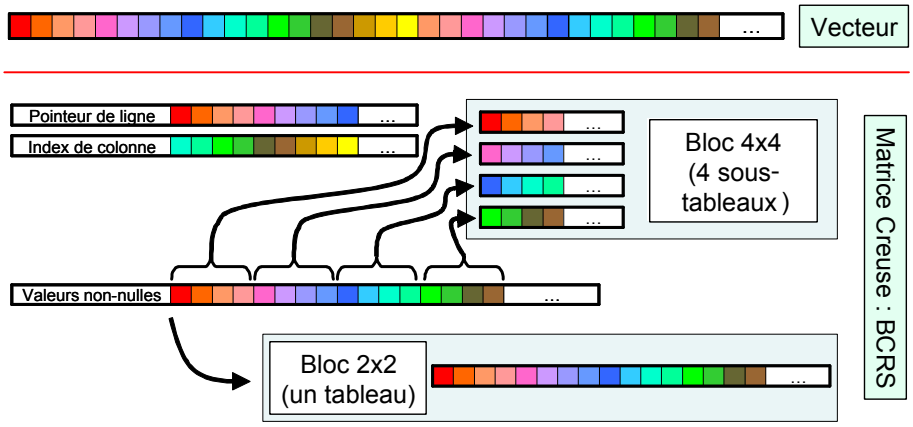


FIG. 2.11 – Stratégies de stockage des vecteurs et des matrices creuses de type BCRS (2x2 et 4x4) sur des GPU à architecture scalaire supportant de grandes tables 1D (par exemple la série 8 d'NVIDIA).

Structure mémoire sur une architecture GPU scalaire (NVIDIA)

Sur une architecture GPU scalaire comme celle de la série 8 d'NVIDIA, il est possible de stocker des vecteurs de grande taille sans les enrouler de manière complexe tout en conservant de très bonnes performances (sur la série 8 d'NVIDIA, la taille maximum d'un tableau 1D est de 4096^2 quadruplets flottants 32bits). Les trois tables utilisées dans le format BCRS sont stockées dans des tableaux 1D de différentes manières. Les tableaux de pointeurs de ligne et d'index de colonne sont stockés comme de simples vecteurs. Dans un soucis d'efficacité, le tableau qui contient les coefficients non-nuls de la matrice est stocké différemment suivant la taille des blocs qui sont utilisés. Plus particulièrement, comme sur les architectures vectorielles, des techniques d'optimisation dites de *strip mining* sont utilisées dans le CNC. Les blocs 2x2 sont stockés bloc par bloc dans un seul tableau 1D de float4 là où les blocs 4x4 sont distribués dans quatre sous-tableaux de float4. Chaque sous-tableau est rempli avec de sous-blocs 2x2 des blocs 4x4 comme proposé par Fatahalian *et al.* (2004). La lecture des 16 valeurs qui constituent un bloc 4x4 est réalisée par quatre accès en lecture à la même adresse mémoire dans chaque sous-tableau. Cette méthode permet de maximiser la bande-passante mémoire et minimiser les calculs de translation d'adresses. La figure 2.11 illustre la stratégie de stockage employée au sein du CNC pour des architectures GPU scalaires.

Implantation efficace d'opérations d'algèbre linéaire sur GPU

Le code assembleur utilisé par l'API Close-To-Metal (CTM) d'AMD-ATI est fait, comme son nom le laisse pressentir, d'instructions réellement très *proches du matériel* (proche du métal). On retrouve par exemple des instructions comme la multiplication-addition combinées ou multiply-and-add (MAD) en anglais, ou bien encore l'accès à des textures 2D au travers de l'instruction TEX. Le code assembleur peut être finement optimisé à l'aide de sémaphores précis qui permettent de lire des données en mémoire graphique de manière asynchrone. Ces sémaphores

aident donc à mieux paralléliser/masquer les latences d'accès à la mémoire en autorisant l'exécution d'instructions tout en attendant la donnée requise. L'implantation d'un produit matrice creuse/vecteur (SpMV) nécessite un peu plus d'une centaine de lignes de code et une trentaine de registres avec la CTM. Le nombre de registres utilisés conditionne fortement les performances, c'est pourquoi nous en avons minimisé l'usage.

Chez NVIDIA avec l'API CUDA, le code GPU est écrit dans un langage proche du langage C. Le kernel qui calcul un SpMV est constitué d'environ 40 lignes de code, et comme avec la CTM, le CNC utilise le minimum de registres possibles afin d'obtenir de bonnes performances.

Le SpMV effectue une boucle sur chaque ligne de blocs d'une matrice creuse. Cette boucle peut être partiellement déroulée afin de traiter les blocs par paire, on parle alors de *loop unrolling*. Sur les cartes d'AMD-ATI avec la CTM, cette tactique permet de pré-charger de manière asynchrone les données du second bloc ce qui améliore sensiblement le masquage des latences d'accès à la mémoire graphique. Ceci est particulièrement vrai lors d'accès dépendants aux données. Les performances du CNC ont ainsi été améliorées d'en moyenne 30% en traitant les blocs deux par deux, et ce que ce soit pour des blocs 2x2 ou 4x4. En revanche, sur les cartes NVIDIA avec CUDA, il n'est pas possible d'accéder à un langage de programmation de bas-niveau qui permettrait de gérer les accès asynchrones à la mémoire GPU. De ce fait, dérouler des boucles tout en cherchant à pré-charger des données n'a aucunement amélioré les performances sur les GPU NVIDIA. Selon la documentation de NVIDIA, le compilateur CUDA est capable de dérouler automatiquement des boucles et d'auto-optimiser la synchronisation des accès à la mémoire graphique. Ceci est certainement une des raisons pour laquelle dérouler à la main des boucles ne procure aucun bénéfice en terme de peformances avec CUDA. La section 2.5.4 tendrait à accréditer cette thèse en montrant que les pourcentages d'efficacité obtenus avec CUDA sont certes moindre qu'avec la CTM, mais tout de même très proches.

La figure 2.12 présente deux méthodes différentes, une pour chaque architecture GPU visée, qui peuvent être utilisées pour implanter des opérations de réduction de vecteurs (par exemple un produit scalaire). Comme nous l'avons vu précédemment, sur les GPU d'AMD-ATI, nous avons utilisé la méthode de réduction itérative présentée par Krüger et Westermann (2003b). Cette méthode réduit à chaque itération la taille du vecteur par quatre. Sur les GPU NVIDIA, il est possible de tirer parti de la mémoire partagée disponible pour effectuer des réductions plus rapides et plus efficaces. La méthode est itérative et basée sur l'utilisation de deux kernels dans le cas d'un calcul de produit scalaire. Le premier kernel effectue les opérations de multiplication du produit scalaire et effectue une première passe de réduction par addition des données. Le deuxième kernel effectue ensuite une nouvelle réduction par addition. Ce dernier kernel est bien entendu itéré autant de fois que nécessaire. Dans chacun des deux kernels utilisés, chaque thread d'un bloc de threads lit quatre valeurs, les réduits par addition, et écrit le résultat en mémoire partagée. Dans chaque bloc de threads, un thread est alors chargé d'additionner l'ensemble des résultats partiels précédemment calculés par les autres threads de son bloc, et d'écrire le résultat en mémoire globale. Au final, le vecteur original est réduit d'un facteur égal à la taille des blocs de threads. Toute l'efficacité d'un tel algorithme est donc conditionnée par le choix de la taille des blocs de threads. Une petite taille de blocs impose un facteur de réduction faible et un grand nombre d'itérations, tandis qu'une grande taille implique un facteur de réduction grand et un petit nombre d'itérations. Dans ce second cas cela signifie également

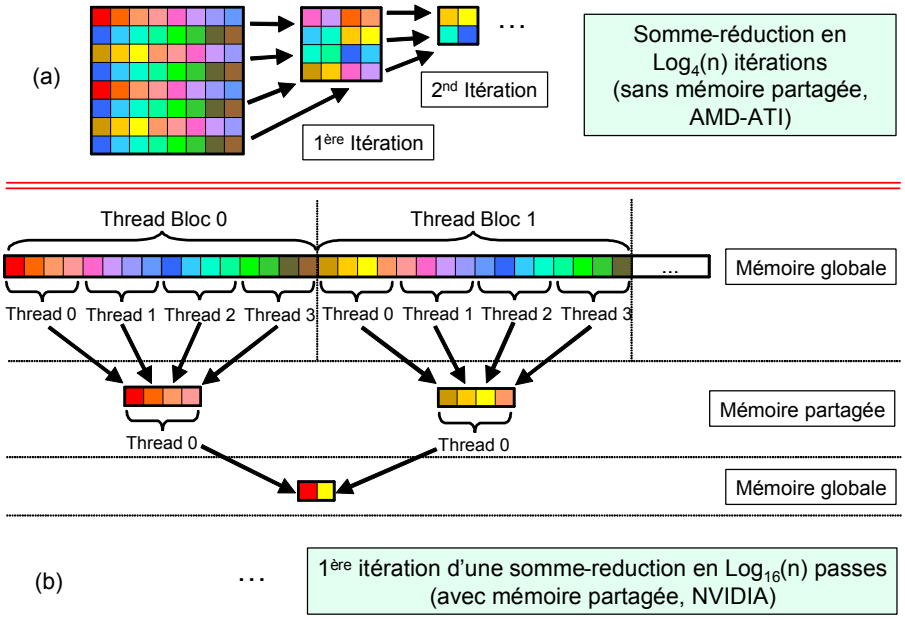


FIG. 2.12 – (a) Parallélisation d'une somme-réduction de vecteur sur une architecture GPU qui ne possède pas de mémoire partagée comme la série X1k d'AMD-ATI. Chaque itération réduit la taille du vecteur d'un facteur 4. (b) La même opération parallélisée sur une architecture GPU qui dispose d'une mémoire partagée comme la série 8 d'NVIDIA. Comme les blocs de threads sont constitués dans cet exemple de quatre threads, chaque itération réduit la taille du vecteur d'un facteur $4 \times 4 = 16$.

qu'un seul thread par bloc sera responsable de l'addition d'un grand nombre de résultats partiels stockés en mémoire partagée, tandis que les autres threads de son bloc seront inactifs. Nos tests ont montré que le meilleur compromis était atteint pour des blocs de 128 threads qui réduisent chacun quatre valeurs scalaires. Par conséquent, le vecteur d'entrée est réduit d'un facteur $128 \times 4 = 512$ à chaque itérations. Les accès consécutifs en lecture puis en écriture dans une même zone mémoire sont mal gérés par les cartes graphiques actuelles. Il est donc préférable d'utiliser deux zones mémoire temporaires pour réaliser les itérations intermédiaires. Ces itérations effectuent une sorte de ping-pong entre les deux zones temporaires, le résultat d'une itération servant d'entrée à la suivante et ainsi de suite. L'exécution sur un GPU de tout kernel se fait au prix d'un certain surcoût de temps qui en général est une constante, et ce quel que soit le kernel exécuté. Ce surcoût est généralement appelé *overhead*. Ce temps sert au chargement du kernel, à la fixation des paramètres de son exécution (pointeurs d'entrée/sortie...) etc. La réduction du nombre d'itérations que permet l'utilisation de la mémoire partagée permet ainsi de limiter le cumul de ces overheads à l'exécution. Notez qu'il est possible d'utiliser les méthodes de réductions présentées pour réaliser d'autres opérations qui nécessitent de parcourir l'ensemble des éléments d'un vecteur (par exemple le calcul d'un minimum ou d'un maximum). Pour plus de détails sur l'utilisation de la mémoire partagée pour l'implantation d'algorithmes de parcours, le lecteur pourra se reporter à la documentation des exemples du SDK de CUDA qui propose une méthode générique optimale de parallélisation pour de tels algorithmes (Harris, 2007).

La mémoire graphique est gérée grâce à des pointeurs que ce soit avec CTM ou CUDA. De cette manière, les données résultants de l'exécution d'un code assembleur ou d'un kernel peuvent être réutilisées en entrée d'un autre code. Ce mécanisme par pointeurs permet d'éviter de nombreuses copies inutiles. Dans notre implantation de l'algorithme du Gradient Conjugué, le code CPU ne fait que demander l'exécution de codes sur GPU dans un ordre prédéfini tout en gérant les pointeurs mémoire en entrée et sortie des codes exécutés. À chaque itération, le code CPU ne récupère qu'une seule valeur scalaire, celle de l'erreur δ de l'algorithme du Gradient Conjugué calculé sur le GPU. A partir de cette unique valeur δ le code CPU peut décider s'il doit ou non stopper les itérations calculées sur le GPU. Au final, le vecteur solution calculé sur le GPU est copié de la mémoire graphique en mémoire PCI-Express si CTM est utilisé, ou en mémoire RAM standard si CUDA est utilisé. Ce qui est fondamental pour les performances, c'est que l'ensemble de la boucle principale de calcul du Gradient Conjugué est exécutée sur le GPU sans jamais nécessiter l'intervention du CPU pour aucuns calculs.

2.5 Applications et performances

L'algorithme du Gradient Conjugué préconditionné nécessite de disposer d'un petit nombre d'opérations d'algèbre linéaire :

- Le produit matrice creuse/vecteur (SpMV), dans notre cas basé sur les formats CRS, BCRS ainsi que par bandes (pour référence).
- Le produit scalaire de vecteurs (SDOT).
- La norme2 d'un vecteur (SNRM2).
- L'addition de deux vecteurs dont le premier est multiplié par un scalaire ($\mathbf{y} \leftarrow \alpha \times \mathbf{x} + \mathbf{y}$, dénoté SAXPY).
- etc.

Cette section présente pour ces opérations des comparaisons de performances en milliards d'opérations à virgule flottante par seconde, ou GFlops pour Giga FLoating-point Operations Per Second. Nous présenterons également des pourcentages d'efficacité en terme de calcul et de bande-passante mémoire. Ces comparaisons portent sur les multiples implantations que nous avons à notre disposition sur CPU et GPU avec la CTM et CUDA. Dans un souci d'équité, les GFlops que nous reportons tiennent comptent systématiquement du temps total d'exécution que cela soit sur CPU ou sur GPU. Nous incluons donc dans nos tests tous les surcoûts (overheads) éventuels introduits par les calculs sur chaque matériel comme les temps de transfert mémoire nécessaires, etc. De même, les GFlops reportés tiennent compte uniquement des opérations sur les nombres flottants utiles à l'opération implantée et non toutes les opérations sur des nombres flottants réalisées. Par exemple, pour un vecteur de taille n, nous comptons $2 \times n - 1$ opérations utiles pour un produit scalaire (SDOT) et ce quel que soit l'implantation utilisée. La table 2.2 précise pour chaque opération et chaque matériel quelle implantation a été utilisée pour effectuer nos tests.

TAB. 2.2 – Implantations utilisées en fonction de l'opération réalisée et du matériel de calcul : CNC correspond à notre implantation, MKL à celle d'Intel, CTM-SDK à celle d'AMD-ATI et CUDA-CUBLAS celle de NVIDIA.

	Matériel		
Opération	**CPU**	**GPU - ATI**	**GPU - NVIDIA**
SAXPY	MKL	CNC	CNC
SDOT/SNRM2	MKL	CNC	CNC
SpMV	CNC	CNC	CNC
Pre-CG	MKL+CNC	CNC	CNC
SGEMM (pour référence)	MKL	CTM-SDK	CUDA-CUBLAS

Pour le test sur CPU des opérations SAXPY, SDOT/SNRM2 et SGEMM (produit de deux matrices denses, donné à titre de référence), nous avons utilisé l'Intel Math Kernel Library (MKL)[21] et l'AMD Core Math Library (ACML)[22] qui sont toutes deux hautement optimisées et parallélisées à l'aide d'instructions SSE3 par exemple. Lors de nos tests, les performances sur CPU entre la MKL et l'ACML étaient très proches, c'est pourquoi nous avons décidé de ne présenter que les résultats de la MKL.

Quant au produit matrice creuse BCRS/vecteur, puisque ces opérations ne sont ni disponibles dans la MKL ni dans l'ACML, les implantations testées sont celles qui ont été réalisées au sein de l'équipe ALICE du LORIA. Ces opérations sont hautement optimisées et parallélisées à l'aide d'instructions SSE3 et d'OpenMP[23].

Les implantations sur GPU de l'opération SGEMM proviennent, pour les cartes AMD-ATI, du SDK de la CTM, et pour les cartes NVIDIA, de la librairie CUBLAS inclue dans l'API CUDA. L'opération SGEMM est ici présentée à titre de référence. Malheureusement, la librairie CUBLAS ne propose pas de structures de matrices creuses.

Afin de valider les supposées améliorations apportées par les API comme CTM et CUDA, nous présentons également des tests sur des implantations, *à l'ancienne*, en OpenGL des opérations SAXPY et SDOT/SNRM2. Les implantations utilisées sont celles de Krüger et Westermann (2003b) qui sont disponibles publiquement.

Tous les tests ont été réalisés sur un PC équipé d'un processeur Intel Xeon 5140 Quad-core accompagné de 3Go de RAM. Nous avons utilisé deux cartes graphiques : une NVIDIA QuadroFX-5600 dotée de 1.5Go de RAM graphique ainsi qu'une AMD-ATI X1900XTX dotée de 512Mo de RAM graphique. La carte NVIDIA est plus récente que celle d'AMD-ATI, ce qui naturellement fait qu'elle offre de meilleures performances dans nos tests.

Les tests réalisés ont été itérés au moins 100 fois afin de moyenner les résultats obtenus. Les tests des opérations SAXPY et SDOT/SNRM2 utilisent des vecteurs synthétiques. Ceux de l'opération SGEMM utilisent des matrices denses synthétiques. Ceux des autres opérations utilisent des matrices issues de la résolution des trois problèmes étudiés à la section 2.2 : la paramétrisation de surface, le lissage de surface et le calcul de simulation d'écoulement fluide.

[21] www.intel.com/software/products/mkl
[22] http ://developer.amd.com/acml.jsp
[23] www.openmp.org

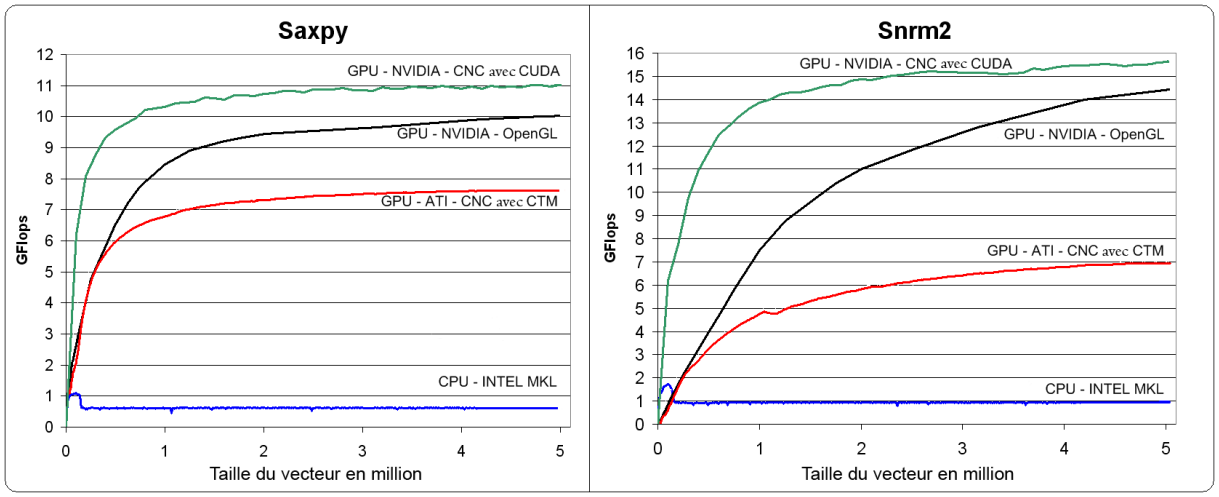


FIG. 2.13 – Comparaison des performances des opérations SAXPY ($\mathbf{y} \leftarrow \alpha \times \mathbf{x} + \mathbf{y}$) et SDOT/SNRM2 (somme-réduction) en fonction de la taille du vecteur traité et du matériel utilisé pour réaliser les calculs. L'implantation CPU est celle de l'Intel MKL. Côté GPU, le CNC propose les deux implantations sur NVIDIA avec CUDA et sur AMD-ATI avec la CTM. L'implantation OpenGL est celle de Krüger et Westermann (2003b).

2.5.1 Opérations sur des vecteurs

La figure 2.13 présente les tests comparatifs des opérations SAXPY ($\mathbf{y} \leftarrow \alpha \times \mathbf{x} + \mathbf{y}$) et SDOT/SNRM2 (somme-réduction) à la fois sur CPU et sur GPU en fonction de la taille des vecteurs utilisés.

Tandis que les performances sur CPU restent stables quel que soit la taille du vecteur utilisé (aux alentours de 0.62 GFlops pour le SAXPY et 0.94 GFlops pour le SNRM2), les performances sur GPU augmentent de pair avec la taille du vecteur utilisé. L'augmentation de la taille des vecteurs, et par conséquent le nombre de threads exécutés, aide le GPU à significativement mieux masquer les latences d'accès à la mémoire graphique.

Pour le SDOT/SNRM2, les performances n'augmentent pas aussi rapidement que pour le SAXPY du fait d'une méthode de calcul itérative qui introduit plus d'overheads et potentiellement plus de latences à masquer. En outre, la pente de la courbe sur les cartes NVIDIA avec CUDA est plus forte pour des petits vecteurs que sur AMD-ATI avec CTM du fait d'une implantation différente qui utilise des facteurs de réduction supérieurs (512 sur NVIDIA au lieu de 4 sur AMD-ATI ou avec OpenGL). Ainsi, avec une carte AMD-ATI ou avec OpenGL, plus d'itérations sont nécessaires pour réaliser une opération de somme-réduction ce qui implique une consommation en bande-passante supérieure ainsi qu'une accumulation plus importante d'overheads (section 2.5.4).

Logiquement, que ce soit pour les opérations SAXPY ou bien SDOT/SNORM2, sur une carte NVIDIA, notre CNC avec CUDA est toujours plus rapide que l'implantation équivalente en OpenGL de Krüger et Westermann (2003b). Ceci est dû à des overheads considérablement

réduits grâce à l'utilisation de l'API entièrement dédiée au GPGPU qu'est CUDA, ainsi qu'à la mémoire partagée disponible.

Au mieux, sur notre carte AMD-ATI avec l'API CTM, le SAXPY est 12.2 fois plus rapide que sur CPU, atteignant plus de 7.6 GFlops, alors que le SNRM2 est 7.4 fois plus rapide avec environ 7 GFlops. Sur notre carte NVIDIA avec CUDA (resp. avec OpenGL), le SAXPY est 18.0 fois plus rapide (resp. 16.4 fois) que sur CPU atteignant 11.0 GFlops (resp. 10.0 GFlops), tandis que le SNRM2 est 16.5 fois plus rapide (resp. 15.2 fois) avec 15.5 GFlops (resp. 14.3 GFlops). Comme prévu, les performances de la carte d'NVIDIA sont systématiquement supérieures à celles de la carte d'AMD-ATI du fait de la différence de génération qui les sépare.

2.5.2 Application à la paramétrisation et au lissage de surface

Les performances des opérations sur des matrices creuses sont grandement conditionnées par le nombre et agencement des coefficients non-nuls de ces même matrices. En conséquence, le choix des modèles ainsi que des tâches à réaliser pour tester notre solveur sur GPU est primordial. Afin de tester nos implantations du produit matrice creuse/vecteur et l'algorithme du Gradient Conjugué dans sa globalité, nous avons choisi cinq modèles et deux tâches qui nous semblent caractéristiques du domaine du traitement numérique de la géométrie. Les cinq modèles utilisés sont présentés dans la table 2.3. La figure 2.14 illustre les résultats obtenus à l'aide du CNC sur GPU pour la paramétrisation et le lissage de deux surfaces.

Tab. 2.3 – Caractéristiques des surfaces utilisées pour tester les opérations sur des matrices creuses. Cette table fournit pour chaque surface : le nombre de variables inconnues à calculer dans le cas d'une paramétrisation ou d'un lissage (#var) ainsi que le nombre de coefficients non-nuls dans la matrice creuse associée (#non-nuls). Le Phlegmatic Dragon est un modèle Eurographics.

	Paramétrisation		**Lissage**	
Surface	**#var**	**#non-nuls**	**#var**	**#non-nuls**
Girl Face 1	1.5K	50.8K	2.3K	52.9K
Girl Face 2	6.5K	246.7K	9.8K	290.5K
Girl Face 3	25.9K	1.0M	38.8K	1.6M
Girl Face 4	103.1K	4.2M	154.7K	6.3M
Phlegmatic Dragon	671.4K	19.5M	1.0M	19.6M

Produit matrice creuse/vecteur (SpMV)

La figure 2.15 compare les performances du produit matrice creuse/vecteur pour différentes implantations et applications. Il est possible de tirer cinq conclusions de ces graphiques :

1. Les performances sur CPU restent quasi stables pour les deux applications tandis que celles sur GPU augmentent de concert avec la taille des matrices.

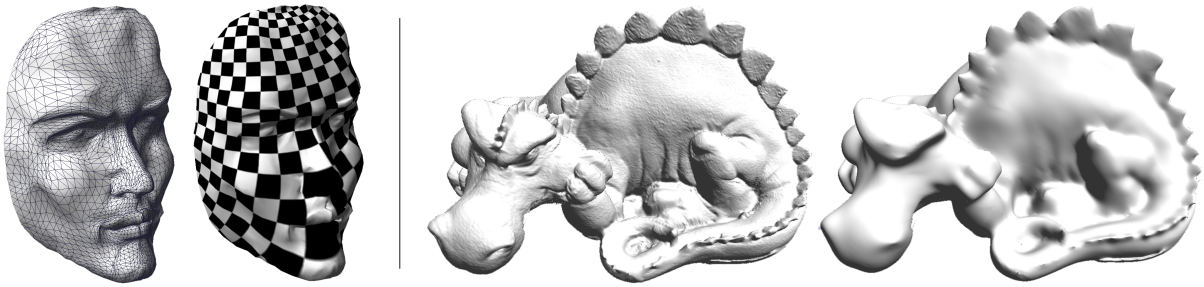


FIG. 2.14 – Figures de gauche : paramétrisation de la surface Girl Face 1 calculée sur GPU. Figures de droite : lissage de la surface du Phlegmatic Dragon calculé sur GPU.

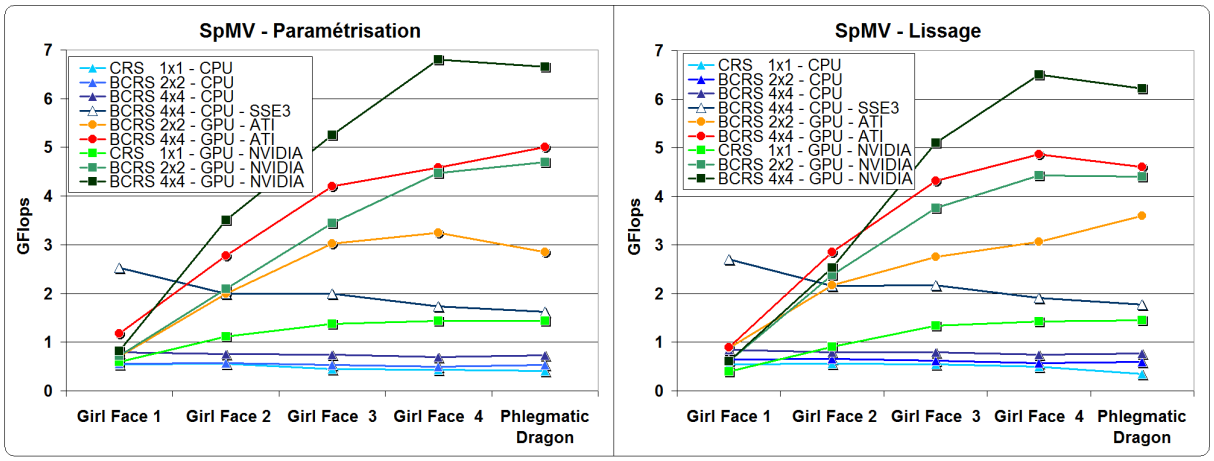


FIG. 2.15 – Comparaison des performances en GFlops de différentes implantations du produit matrice creuse/vecteur (SpMV) dans le cas du calcul d'une paramétrisation de surface et d'un lissage de surface.

2. Grâce aux stratégies de Register Blocking et de vectorisation, le format BCRS 4x4 est plus rapide que le 2x2 (en moyenne 33% sur CPU –sans SSE3– et 50% sur les GPU) et le BCRS 2x2 est lui-même plus rapide que le format CRS (18% sur CPU et 300% sur les GPU d'NVIDIA).
3. En BCRS 4x4, sur des grilles d'une taille significative, l'implantation sur les cartes d'AMD-ATI avec la CTM (resp. sur NVIDIA avec CUDA) est 3.1 fois plus rapide (resp. 4.0 fois) que celle multi-threads sur CPU accélérée à l'aide d'instructions SSE3, atteignant les 5 GFlops (resp. 6.8 GFlops).
4. L'implantation SSE3 multi-threads est entre 2 et 2.5 fois plus rapide que l'implantation standard qui n'utilise pas les instructions SSE3.
5. Pour de petites matrices, le CNC offre des performances tout juste comparables avec les implantations sur CPU. Dans ce cas, il est de toute manière souvent préférable d'utiliser un solveur direct.

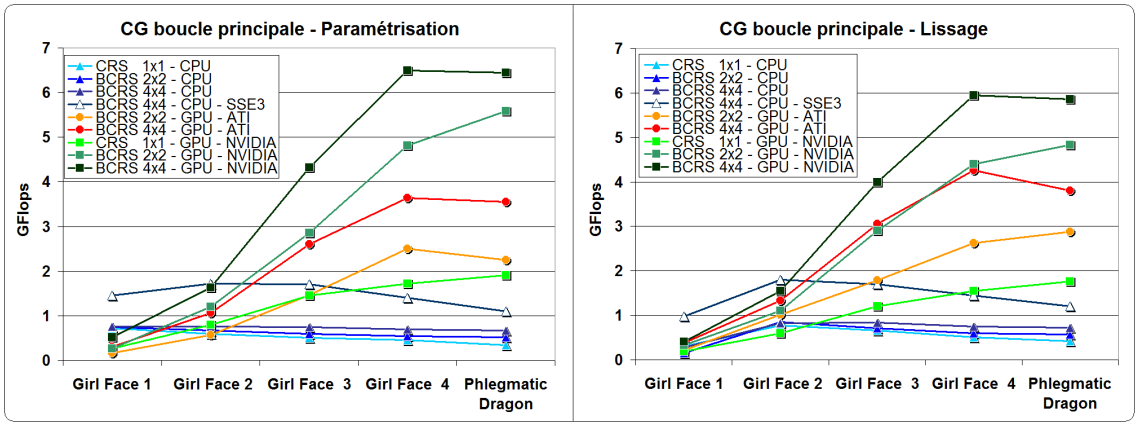


FIG. 2.16 – Comparaison des performances de différentes implantations du Gradient Conjugué (CG) pour le cas du calcul d'une paramétrisation de surface et d'un lissage de surface.

Gradient Conjugué (CG) préconditionné

Performances brutes Les performances mesurées en GFlops du Gradient Conjugué préconditionné par Jacobi sont reportées par la figure 2.16 pour les mêmes implantations qu'à la section précédente. En pratique, la majorité du temps d'exécution du Gradient Conjugué, environ 80%, est occupée par le calcul du produit matrice creuse/vecteur et ce quel que soit l'implantation utilisée et donc quel que soit le matériel de calcul utilisé. Par conséquent, les performances du SpMV conditionnent directement celles du Gradient Conjugué. Les commentaires faits précédemment sont donc parfaitement applicables au cas du Gradient Conjugué dans son ensemble. Le solveur GPU sur AMD-ATI avec la CTM est 3.2 fois plus rapide que celui SEE3 multi-threads sur CPU, tandis que celui sur NVIDIA avec CUDA est 6.0 fois plus rapide. La version SSE3 multi-threads sur CPU est 1.8 fois plus rapide que celle non-SSE3. Comme pour le SpMV, le CNC est inefficace pour les systèmes de petites tailles, auxquels les solveurs directs sont plus adaptés.

Temps de résolution du système linéaire Les figures 2.15 et 2.16 nous ont permis de comparer les diverses implantations dont nous disposions en terme de puissance de calcul brute en GFlops. L'objectif était de démontrer l'intérêt à utiliser des techniques de traitement par bloc, de vectorisation, de strip mining, des instructions SSE, etc. Ce que ces calculs ne prenaient pas en compte, c'est que les blocs des matrices creuses ne sont pas nécessairement totalement remplis de coefficients non-nuls (surtout lorsque des blocs de grandes tailles sont utilisés) ce qui concrètement revient à pénaliser le temps de résolution du système linéaire considéré. Le tableau suivant reporte les taux de remplissage, c'est-à-dire le pourcentage moyen de coefficients non-nuls par bloc, en fonction de la taille du bloc et de l'application considérée pour le modèle qui nous semble être le plus significatif, le Girl Face 4 :

TAB. 2.4 – Taux de remplissage des blocs en fonction de leur taille et de l'application envisagée sur le modèle du Girl Face 4.

Taille des blocs	Paramétrisation	Lissage
CRS : 1x1	100%	100%
BCRS : 2x2	92.8%	35.6%
BCRS : 4x4	29.3%	14.4%

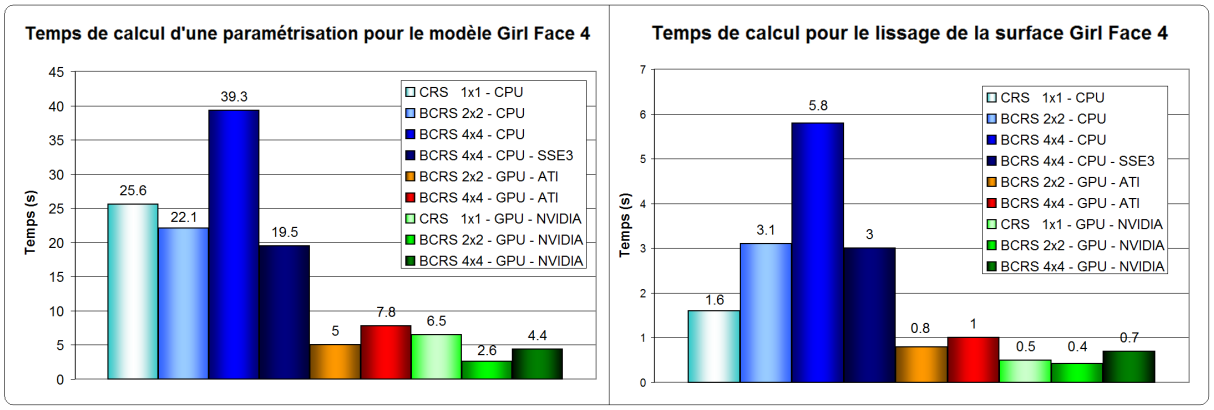


FIG. 2.17 – Temps de résolution des systèmes linéaires correspondants à la paramétrisation et au lissage de la surface Girl Face 4, et ce pour différentes implantations sur CPU et sur GPU.

Ces taux de remplissage diminuent donc rapidement avec l'augmentation de la taille des blocs, ce qui a pour effet de contre-balancer l'amélioration de performances apportée par les grandes tailles de blocs. De plus, ce taux diminue plus vite pour la matrice correspondant à un lissage de surface que pour celle correspondant à une paramétrisation.

Ce paragraphe compare ce qui finalement est le plus important, à savoir les temps de résolution totaux des systèmes linéaires générés pour le calcul d'une paramétrisation et d'un lissage (figure 2.17) sur le modèle du Girl Face 4.

La figure 2.17 nous permet d'affirmer que globalement les remarques formulées aux paragraphes précédents sont toujours valables ici, avec tout de même quelques différences notables :

- Le format BCRS 4x4 est systématiquement plus lent que le 2x2 du fait d'un taux de remplissage des blocs plus faible qui implique de nombreuses multiplications inutiles par des coefficients nuls. Cette perte d'efficacité n'est pas suffisamment compensée par l'augmentation brute des performances.
- L'utilisation de blocs de taille 2x2 est un bon compromis puisque ce format offre un bon taux de remplissage combiné avec de bonnes performances brutes. Il est systématiquement plus rapide que les autres format, exception faite du cas du lissage de surface uniquement sur CPU pour lequel le format 1x1 est le plus rapide.
- Les implantations GPU sont entre 3 et 8.5 fois plus rapides que celles sur CPU.
- Toutes implantations confondues, pour le calcul d'une paramétrisation de surface, la meilleure implantation GPU est 7.5 fois plus rapide que la meilleure implantation sur CPU. Pour le calcul d'un lissage, on descend à 3.5 fois plus rapide, du fait d'un taux de remplissage des blocs moindre qui favorise le format CRS implanté sur CPU.

2.5.3 Application à la simulation d'écoulement fluide : résolution de l'équation de pression

Comme nous l'avons vu à la section 2.2.3, le calcul d'une simulation d'écoulement fluide se décompose en plusieurs étapes, dont la plus consommatrice en puissance de calcul est celle de la résolution de l'équation de pression. Cette étape nécessite de résoudre un système linéaire qui peut être très grand puisqu'il compte autant d'équations que de cellules dans le maillage servant de support à la simulation. Nous avons donc utilisé notre CNC afin d'accélérer la résolution de l'équation de pression sous les conditions présentées à la section 2.2.3. En pratique, nous avons intégré le CNC au plugin pour Gocad nommé StreamLab (Fetel, 2007) afin de bénéficier de toutes ses commodités. Ce plugin implante un simulateur d'écoulement complet à base de lignes de courant.

Le modèle *Karst 1* utilisé pour le calcul de simulations d'écoulement fluide est décrit dans la table 2.5. Ce modèle a la particularité de prendre en compte des *karsts*[24]. La figure 2.18 illustre les résultats obtenus à l'aide du CNC sur GPU lors d'une simulation d'écoulement fluide dans une tranche du modèle Karst 1.

Tab. 2.5 – Caractéristiques du maillage synthétique utilisé pour les simulations d'écoulement fluide. Le tableau reporte le nombre de variables inconnues à calculer (#var) ainsi que le nombre de coefficients non-nuls dans la matrice creuse associée (#non-nuls).

Maillage	**#var**	**#non-nuls**
Karst 1	156.8K	784.0K

Gradient Conjugué (CG) préconditionné : temps de résolution du système linéaire

La figure 2.19 présente les temps de résolution de l'équation de pression pour diverses implantations sur CPU et sur GPU. Dans les conditions particulières que nous avons adoptées ici (contraintes aux puits injecteurs en pression, etc.), la matrice de notre système linéaire a la spécificité d'être une matrice creuse par bandes. C'est pourquoi nous avons implanté une structure de matrice creuse par bandes au sein du CNC à la fois sur CPU et sur GPU afin de permettre des comparaisons avec les formats CRS et BCRS. Afin d'éviter de charger inutilement la figure 2.19, nous ne reportons plus les performances de la carte graphique d'AMD-ATI.

Cinq points sont à noter :

1. Les structures de matrices par bandes, lorsqu'elles sont utilisables, sont intrinsèquement plus efficaces que celles de type CRS ou BCRS. Ceci est dû au fait qu'elles utilisent un adressage direct des coefficients non-nuls de la matrice là où, à contrario, les structures CRS et BCRS utilisent des tables d'indexation à deux niveaux. Cette différence de performances se vérifie à la fois sur CPU et sur GPU.

[24] Les karsts sont des reliefs particuliers aux régions dans lesquelles les roches calcaires forment d'épaisses assises, et résultent de l'action, en grande partie souterraine, d'eaux qui dissolvent le carbonate de calcium.

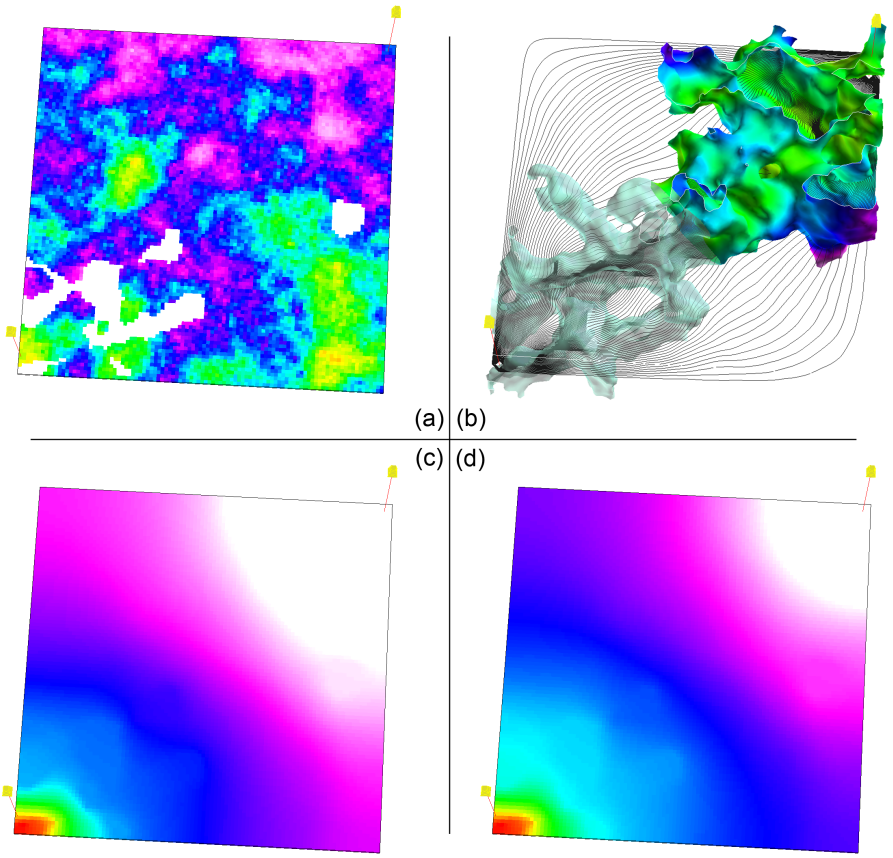


FIG. 2.18 – (a) est le maillage Karst 1 utilisé comme support de la simulation d'écoulement fluide. La propriété peinte sur le maillage est la perméabilité. (c) et (d) présentent la pression simulée sur GPU pour deux pas de temps consécutifs dans une tranche du modèle Karst 1. (b) illustre les lignes de courant calculées ultérieurement sur une tranche du modèle Karst 1 ainsi que les surfaces délimitant les volumes karstiques.

2. Les blocs de trop grande taille ne sont, encore une fois, pas toujours les plus efficaces du fait de taux de remplissages relativement bas (50% pour des blocs 2x2 et seulement 25% pour des blocs 4x4).

3. Encore une fois, les instructions SSE3 permettent sur CPU d'obtenir des performances honorables, avec un facteur d'accélération d'environ 1.9 fois par rapport à l'implantation standard.

4. Systématiquement, les temps de résolution sur GPU sont plus courts et donc meilleurs que ceux sur CPU. Les facteurs d'accélération constatés sont compris entre 7.5 et 13.5 fois à structure de matrice équivalente.

5. Toutes implantations confondues, le meilleur temps de résolution sur GPU est 11.5 fois plus court que le meilleur temps sur CPU.

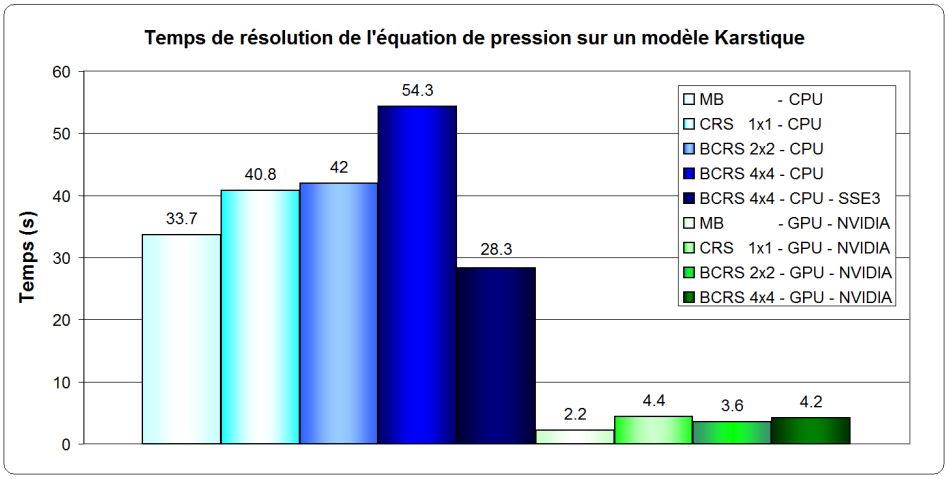


FIG. 2.19 – Temps de résolution de l'équation de pression pour diverses implantations CPU et GPU disponibles dans le CNC pour le modèle Karst 1. MB signifie Matrice par Bandes.

Comparaison avec des librairies de solveurs directs et itératifs

Jusqu'ici nous avons comparé des implantations du Gradient Conjugué préconditionné sur GPU avec des implantations sur CPU. Nos conclusions ont été très claires, les tests sont systématiquement à l'avantage des implantations GPU pour des matrices de grande taille. Il nous reste donc à voir comment se comporte notre CNC face à des librairies de solveurs numériques hautement-optimisées classiques sur CPU. Pour ce faire, nous avons sélectionnés à la fois des librairies de solveurs directs (UMFPACK, TAUCS) et itératif (LASPACK). UMFPACK[25] signifie Unsymmetric Multifrontal sparse LU Factorization Package. C'est donc un solveur direct creux non-symétrique basé sur une factorisation LU extrêmement optimisée. UMFPACK a d'ailleurs été partiellement intégré dans MATLAB. TAUCS[26] intègre un solveur creux direct basé sur la méthode de Cholesky, méthode qui est dérivée de la décomposition LU mais spécifiquement adaptée au cas des matrices symétriques définies positives. LASPACK[27] implante plusieurs solveurs creux itératifs, principalement basés sur l'algorithme du Gradient Conjugué préconditionné.

Systématiquement, pour chaque librairie, nous avons choisi les paramètres qui offrent les meilleures performances dans le but de comparer ce que chacun propose de mieux. Dans notre cas particulier, la résolution de l'équation de pression dans une simulation d'écoulement suivant un schéma IMPES, la matrice du système linéaire à résoudre est symétrique. De base, le CNC n'exploite pas la symétrie potentielle de la matrice car il a été conçu pour être le plus générique possible, tout du moins au niveau des structures mémoire utilisées. En revanche, UMFPACK, TAUCS et LASPACK peuvent tous tirer parti de cette symétrie, et bien entendu nous avons sélectionné les options correspondantes. Nous avons également choisi le préconditionneur SSOR disponible dans LASPACK car celui-ci offre de meilleures performances en pratique que celui de Jacobi.

[25] http ://www.cise.ufl.edu/research/sparse/umfpack/
[26] http ://www.tau.ac.il/ stoledo/taucs/
[27] http ://www.mgnet.org/mgnet/Codes/laspack/html/laspack.html

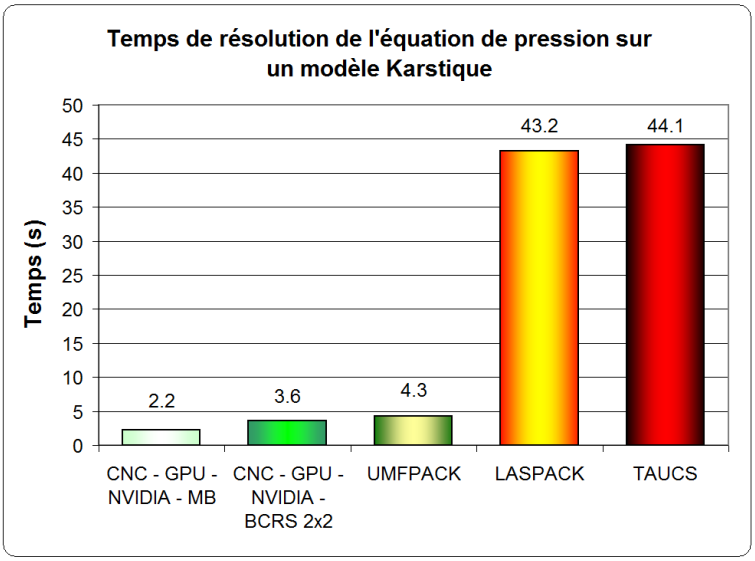


FIG. 2.20 – Temps de résolution de l'équation de pression sur le modèle karstique Karst 1 : comparaison entre le CNC et des librairies de solveurs directs et itératifs classiques. MB signifie Matrice par Bandes.

Notez qu'à la fois UMFPACK, TAUCS et LASPACK utilisent des nombres flottants double précision. Inversement, comme nous l'avons vu, notre CNC se limite à des flottants simple précision du fait de l'absence de support des nombres flottants double précision sur les GPU actuels.

La figure 2.20 compare les temps de résolution de l'équation de pression entre notre CNC sur GPU et les librairies de solveurs directs UMFPACK et TAUCS ainsi que le solveur itératif de LASPACK.

Plusieurs conclusions peuvent être tirées de cette figure :

- Parmi les trois solveurs sur CPU testés, UMFPACK est, et de loin, le plus rapide. Il est environ 10 fois plus rapide que LASPACK ou TAUCS sur le modèle testé.
- Comparé à UMFPACK, les temps obtenus avec le CNC sur GPU sont 16% plus rapides si le format BCRS 2x2 est utilisé, et 50% plus rapides si le format de matrice par bandes est utilisé.
- Le temps de calcul avec l'implantation de LASPACK du Gradient Conjugué est relativement proche du temps que nous avons obtenu à la section précédente (figure 2.19) avec notre implantation, au sein du CNC, sur CPU de ce même algorithme. Ceci valide donc notre implantation CPU.

Remarquez que l'exploitation de la symétrie de la matrice permet d'améliorer mécaniquement les performances. Inversement, généralement les performances de calculs en double précision sont inférieures aux performances des mêmes calculs réalisés en simple précision. Sachant que d'un côté on exploite la symétrie de la matrice et des nombres flottants double précision (UMFPACK, TAUCS et LASPACK), et que de l'autre on exploite pas cette symétrie et on utilise des flottants

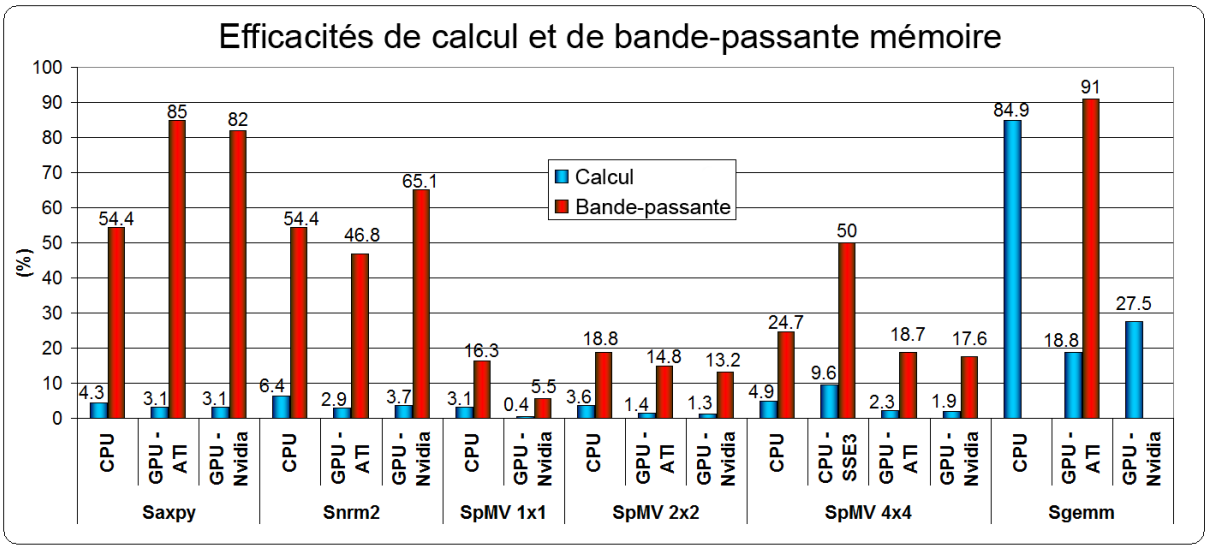


FIG. 2.21 – Pourcentages d'efficacité de calcul et de bande-passante mémoire en fonction de l'opération implantée et du matériel utilisé (CPU, GPU AMD-ATI ou NVIDIA). Les vecteurs utilisés ont une taille de 1024^2, les matrices denses de 1024×1024, et les matrices creuses sont issues du modèle Girl Face 4 dans le cas d'une paramétrisation de surface. La table 2.2 indique quelles implantations ont été utilisées dans chaque cas.

simple précision (CNC), les résultats présentés dans cette section doivent être relativisés. Ce que nous pouvons affirmer, c'est que le CNC sur GPU rivalise avec les meilleurs solveurs directs sur CPU lorsque les systèmes linéaires à résoudre sont d'une taille raisonnable. Comme nous l'avons souligné précédemment, lorsque la taille des systèmes à résoudre est trop grande, les solveurs directs ne fonctionnent plus du fait de leur consommation mémoire très importante. Dans ces cas précis, notre solveur itératif sur GPU peut efficacement prendre le relais.

2.5.4 Surcoût, efficacité et précision

Les approches précédentes en GPGPU étaient toutes limitées par l'utilisation de librairies dédiées au graphisme telles qu'OpenGL ou Direct3D. Ces librairies introduisaient des surcoûts (overheads) à l'exécution très significatifs du fait de l'obligation de passer au travers de l'ensemble du pipeline graphique pour effectuer le moindre calcul. Avec les nouvelles API telles que CTM ou CUDA, la donne est complètement différente. En offrant un accès direct aux ressources de calcul et de mémoire, celles-ci réduisent ces overheads très significativement. Au cours de l'ensemble de nos tests, le temps total de calcul passé sur le GPU a toujours été supérieur à 93%, ce qui signifie en clair que moins de 7% du temps a été perdu à cause des overheads. Ceci est une avancée majeure par rapport aux approches précédentes. A titre d'exemple, le coût d'exécution d'un unique kernel sur une carte NVIDIA avec CUDA est seulement d'environ 15 μs.

Toute architecture, qu'elle soit axée CPU ou GPU, est limitée à la fois par sa puissance de calcul maximum et sa bande-passante mémoire maximum. Comme le montre la figure 2.21, l'efficacité de calcul (définie comme le nombre d'opérations réalisées rapporté au nombre d'opérations

maximum possibles sur un intervalle de temps) et l'efficacité de bande-passante sont conditionnées par l'opération implantée. Par exemple, des opérations peu consommatrices en puissance de calcul comme le SAXPY ou le SNRM2/SDOT sont finalement limitées par la bande-passante mémoire et pas par la puissance de calcul disponible. C'est pourquoi le SAXPY atteint une très bonne efficacité en terme de bande-passante, entre 55 et 85%, et une très faible efficacité en terme de puissance de calcul, entre 3 et 4%, tous matériels confondus (CPU et GPU). Inversement, pour référence, le produit de deux matrices denses (SGEMM) sur GPU atteint une bonne efficacité de calcul, entre 19% et 28%, et une très bonne efficacité de bande-passante aux environs de 91% sur une carte AMD-ATI (la bande-passante sur les cartes NVIDIA n'est pas mesurable puisqu'elle dépend de la façon d'implanter le produit de matrices, chose qui nous est inconnue). Dans le cas du produit d'une matrice creuse par un vecteur (SpMV), les efficacités de calcul et de bande-passante sont assez faibles que cela soit sur GPU ou sur CPU (même si l'utilisation du SSE3 aide considérablement). Le responsable de ces taux d'efficacités faibles est à chercher du côté du type de matrice creuse utilisé. En effet, le format BCRS implique de résoudre une indexation à double niveau qui oblige donc à réaliser des accès mémoire successivement dépendants les uns des autres. En outre, les caches de données présents sur CPU et sur GPU sont rendus inefficaces de part le positionnement *aléatoire* des coefficients non-nuls dans la matrice creuse. Néanmoins, comme nous l'avons précédemment montré, le CNC calcule un SpMV, relativement à notre meilleure implantation CPU, 3.1 fois plus vite sur une carte AMD-ATI et 4.0 fois sur une NVIDIA.

La figure 2.21 permet de comparer les efficacités obtenues à l'aide des deux approches à la fois siamoises et antagonistes que sont CTM et CUDA. En effet, CTM et CUDA offrent toutes les deux un accès direct aux puissantes ressources de calcul parallèles et à la très large bande-passante des GPU. Là où les deux approches divergent, c'est dans la manière de programmer : CTM ne peut interpréter que du code assembleur de très bas niveau qui autorise de très fines et nombreuses optimisations, là où CUDA propose un langage de haut-niveau proche du C et qui nécessite d'être compilé préalablement à toute exécution sur un GPU. Il est donc intéressant de constater que les pourcentages d'efficacité du code compilé par CUDA sont certes toujours plus faibles que ceux obtenus avec la CTM, mais finalement pas si éloignés que ça. Ceci permet d'affirmer que le compilateur CUDA est, malgré sa jeunesse manifeste, déjà relativement mûr.

Depuis quelques années déjà les cartes graphiques supportent le standard IEEE-754 sur les nombres flottants simple précision. En pratique, certaines petites déviations par rapport à ce standard existent, mais théoriquement celles-ci ne devraient pas avoir d'impact, tout du moins dans notre cas particulier, sur la précision des calculs effectués sur GPU. Lors de nos nombreux tests, les résultats obtenus sur GPU en simple précision ont toujours été suffisamment proches de ceux obtenus sur CPU pour pouvoir être considérés comme étant équivalents.

Lorsqu'une application nécessite de résoudre un système linéaire en double précision, il est possible d'utiliser des algorithmes de résolution dits en précision-mixte. Ceux-ci utilisent un solveur itératif sur GPU afin de déterminer une première solution approchée du système linéaire, puis un solveur itératif sur CPU en double précision qui réutilise le résultat préalablement obtenu sur GPU. De cette manière, un grand nombre d'itérations sont réalisées très rapidement sur le GPU, suivi d'un très petit nombre d'itérations réalisées lentement sur CPU. Ces approches permettent de bénéficier à la fois de la puissance de calcul des GPU et de la double précision des

CPU. Pour plus de détails sur ce type d'approches, le lecteur est invité à se référer aux articles de Göddeke *et al.* (2005), Göddeke *et al.* (2007) et Strzodka et Göddeke (2006).

2.6 Bilan

2.6.1 Conclusion

Le Concurrent Number Cuncher (CNC) vise à proposer le solveur creux le plus performant possible grâce à l'accélération matérielle des GPU modernes et à l'utilisation des nouvelles API dédiées au GPGPU qui vont de pair avec ces GPU. Le CNC utilise un format de stockage des matrices creuses par blocs de lignes compressées (BCRS). Celui-ci est à la fois plus rapide et plus compact que le format plus classique sans blocs appelé stockage par lignes compressées (CRS). L'utilisation du format BCRS à la place du CRS permet d'appliquer des techniques de cache basées sur des blocs de registres ou bien encore de réaliser une vectorisation des données. Ces optimisations permettent d'exploiter au maximum les capacités des GPU et même des CPU (grâce aux instructions SSE3). À notre connaissance, notre CNC est la première implantation d'un solveur symétrique creux générique sur GPU qui puisse résoudre efficacement des problèmes d'optimisation non-structurés.

Comparées à l'implantation sur CPU hautement optimisée de l'Intel Math Kernel Library (MKL), les opérations sur des vecteurs sont jusqu'à 18 et 16.5 fois plus rapide avec le CNC sur GPU, respectivement pour les opérations SAXPY et SDOT/SNRM2. Comparé à notre implantation CPU SSE3 multi-threads, le produit matrice creuse/vecteur (SpMV) est jusqu'à 4 fois plus rapide sur GPU. Les temps de résolution des systèmes linéaires sont jusqu'à 3.5 fois plus courts pour calculer un lissage de surface, 7.5 fois plus courts pour calculer une paramétrisation de surface et enfin 11.5 fois plus courts pour la résolution de l'équation de pression d'une simulation d'écoulement fluide. Nous avons également montré que l'utilisation du format BCRS a permis dans la majorité des cas d'améliorer sensiblement les performances, tout du moins lorsque la taille des blocs restait raisonnable. Pour des systèmes linéaires d'une taille raisonnable, dans le cas de la résolution de l'équation de pression, comparé à des librairies de solveurs classiques, le CNC sur GPU s'est avéré être significativement plus rapide que TAUCS (solveur direct) ou LASPACK (solveur itératif) et a rivalisé avec UMFPACK (solveur direct). Remarquez que pour des systèmes linéaires de grande taille, les solveurs directs ne fonctionnent pas du fait de leur consommation mémoire trop importante. Cette limite ne s'applique pas à notre solveur puisque celui-ci est de type itératif.

2.6.2 Perspectives

Actuellement les cartes graphiques ne supportent que les nombres flottants en simple précision. C'est pourquoi notre CNC vise dans un premier temps les applications qui ne se focalisent pas sur la précision des résultats mais plutôt sur la haute-performance des calculs réalisés. Malgré tout, il serait tout à fait envisageable d'utiliser notre solveur sur GPU combiné à un solveur sur CPU afin de réaliser un schéma de résolution en précision-mixte (Göddeke *et al.*, 2005, 2007; Strzodka et Göddeke, 2006). Cette approche permettrait de bénéficier à la fois de la puissance

de calcul des GPU et de la double précision des CPU. Notez qu'NVIDIA a récemment indiqué que leur prochaine génération de cartes graphiques attendue au second semestre 2008, le G100, supportera les nombres flottants en double précision.

Comme nous l'avons montré, toute opération est limitée à la fois par la bande-passante maximum de la mémoire (et de ses latences) et par la puissance de calcul maximum du matériel d'exécution (CPU ou GPU). Pour la plupart des opérations d'algèbre linéaire BLAS, le facteur limitant est la bande-passante mémoire et pas la puissance de calcul. La bande-passante maximum s'établit à 49.6 Go/s pour notre carte de test AMD-ATI et à 76.8 Go/s pour notre carte NVIDIA. Il est alors intéressant de remarquer que, par exemple, la dernière carte graphique d'AMD-ATI, la HD 3870 X2, offre une bande-passante nettement supérieure à celles de nos cartes de test, de l'ordre de 120 Go/s. Par conséquent, l'utilisation d'une carte de ce type permettrait sûrement de démultiplier les performances du CNC et de creuser encore plus l'écart qui sépare notre solveur sur GPU de ceux sur CPU.

Le solveur que nous avons implanté est un Gradient Conjugué préconditionné par Jacobi, ce qui limite son champ d'application à la résolution de systèmes symétriques. Comme nous l'avons vu, lorsque le système à résoudre n'est pas symétrique, il est possible de le transformer pour le rendre symétrique afin de pouvoir lui appliquer l'algorithme du Gradient Conjugué. Cette méthode est connue sous le nom de Gradient Bi-Conjugué (biCG). Cette dernière, ou bien sa version à convergence améliorée nommée biCGSTAB, peut facilement être implantée au niveau du CNC puisque le format de matrice creuse BCRS que nous utilisons est totalement générique et supporte aisément des matrices non-symétriques. L'implantation d'un biCGSTAB au sein du CNC permettrait de bénéficier de toute la puissance d'un GPU pour résoudre des systèmes non-symétriques. L'implantation d'un solveur direct sur GPU est également une direction de recherche très importante à explorer. Celle-ci pourrait déboucher sur des solveurs d'une rapidité inégalée.

Notre CNC implante un Gradient Conjugué parallélisé soit sur un CPU multi-cores (grâce au SSE3 et au multi-threading) soit sur un seul GPU. Depuis quelques années, la tendance est à la mise en parallèle de plusieurs PC dans des clusters, à l'installation de plusieurs cartes graphiques dans un seul et même PC (on parle de SLI ou CrossFire), à la création de clusters de PC contenant chacun plusieurs cartes graphiques, voire à l'utilisation de nouveaux systèmes de calcul (NVIDIA QuadroPlex) contenants plusieurs cartes graphiques disposant chacune de plusieurs GPU disposés dans un seul et même boîtier dédié (actuellement il est possible d'avoir jusqu'à 8 GPU simultanément). Ces nouvelles approches impliquent de paralléliser à de multiples niveaux les algorithmes implantés, ce qui représente de nouveaux challenges. Par conséquent, il serait intéressant d'étudier l'extension de la parallélisation de notre solveur non plus à un seul processeur massivement parallèle, mais à un ensemble de processeurs parallèles (GPU ou CPU) localisés ou non dans un même boîtier.

Pour conclure, notre solveur creux générique sur GPU sera très prochainement disponible publiquement sous la forme d'une librairie. Celle-ci permettra à tout un chacun de la réutiliser, et d'exploiter la puissance des GPU qu'ils soient d'AMD-ATI ou d'NVIDIA afin de résoudre tout type de problème d'optimisation non-structuré.

Conclusion

Conclusions

La démarche classique, lors de l'étude de phénomènes ou de propriétés physiques, consiste à modéliser l'objet de l'étude sous la forme d'une grille discrète, de simuler son comportement lorsque celui-ci est soumis à différentes contraintes, puis d'en visualiser le résultat. L'extension de l'utilisation des grilles fortement non-structurées dans ce processus de modélisation, de calcul, et de visualisation, pose des problèmes majeurs. Pour la simulation, cela pose le problème de la résolution d'équations aux dérivées partielles qui, une fois discrétisées sur ce type de grille, revient à résoudre de grands systèmes d'équations linéaires creux dont la répartition des coefficients non-nuls ne présente aucun motif identifiable. La résolution de tels systèmes impose donc de disposer de structures de matrices creuses efficaces et de solveurs numériques performants. Pour la visualisation des résultats de simulation, cela pose à la fois le problème du stockage et de l'adaptativité des algorithmes à une géométrie et une topologie variables.

Le point commun à toutes les méthodes relatives aux grilles non-structurées est leur complexité algorithmique élevée, et par conséquent leur grande consommation en temps de calcul. C'est sur ce dernier point que les processeurs graphiques récents peuvent devenir indispensables de par leur colossale puissance de calcul parallèle et bande-passante mémoire. Les GPU permettent d'accélérer fortement des algorithmes parallélisables une fois ceux-ci adaptés aux spécificités des modèles de programmation propres aux GPU.

Dans cette thèse, nous avons défini de nouveaux algorithmes de calcul et de visualisation sur des grilles fortement non-structurées qui sont accélérés à l'aide de GPU modernes. Ces méthodes forment une chaîne complète que les paragraphes suivants résument.

Comme nous l'avons montré à la section 2.2, de nombreuses applications utilisent des solveurs numériques creux dont les performances conditionnent celles de l'application considérée (figure 1). Notre algorithme, le Concurrent Number Cuncher (CNC), vise à proposer un solveur creux le plus performant possible afin d'accélérer la résolution de ces problèmes (chapitre 2). A cet effet, le CNC tire parti de l'accélération matérielle des GPU modernes et de nouvelles API dédiées au GPGPU qui vont de pair avec ces GPU : CTM et CUDA. Ainsi, le CNC implante un solveur itératif de type Gradient Conjugué préconditionné hautement parallélisé sur GPU. Il utilise un format de stockage des matrices creuses, inédit sur GPU, par blocs de lignes compressées (BCRS) qui est à la fois très compact et très rapide. L'utilisation du format BCRS permet d'appliquer des techniques de cache basées sur des blocs de registres ou bien encore de réaliser une vectorisation des données. Ces optimisations permettent d'exploiter au maximum les capacités des GPU et même des CPU (grâce aux instructions SSE3). À notre connaissance,

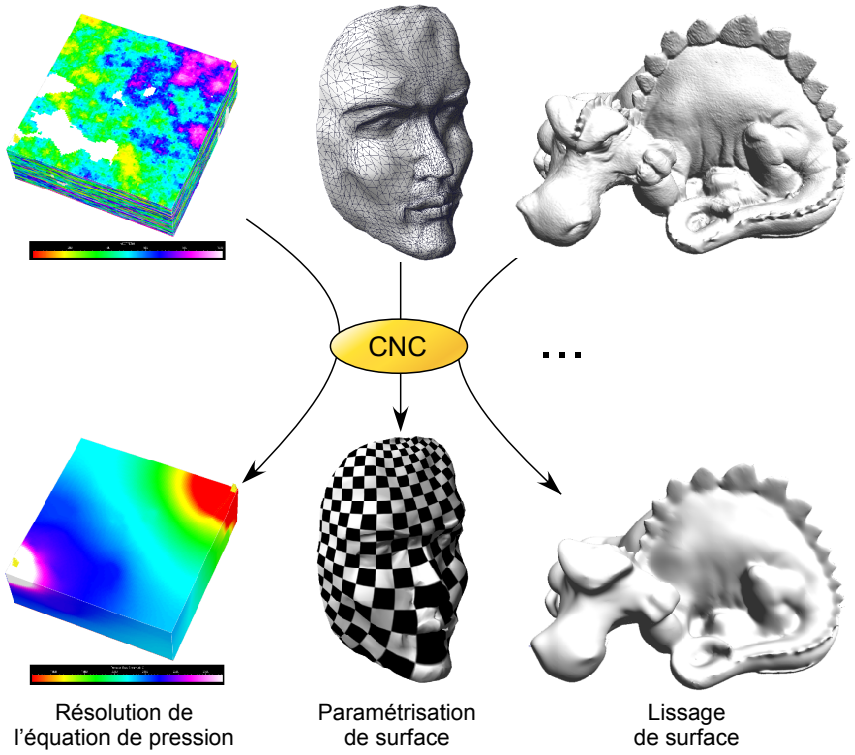


FIG. 1 – Le CNC s'applique à tout problème nécessitant de résoudre un système linéaire creux : résolution de l'équation de pression dans un réservoir pétrolier (colonne de gauche), calcul d'une paramétrisation de surface triangulée (centre), calcul du lissage d'une surface (droite), etc.

notre CNC est la première implantation d'un solveur symétrique creux générique sur GPU qui puisse résoudre efficacement des problèmes d'optimisation non-structurés.

Afin de démontrer l'efficacité du solveur CNC sur GPU (et des opérations d'algèbre linéaire sous-jacentes), nous l'avons comparé sur différentes applications de traitement numérique de la géométrie (lissage de surface et paramétrisation de surface) et de simulation d'écoulement fluide, avec des implantations hautement optimisées sur CPU de différents types de solveurs. Selon nos tests, les opérations sur des vecteurs sont jusqu'à 18 fois plus rapides sur GPU que sur CPU tandis que celles sur des matrices creuses sont jusqu'à 4 fois plus rapides. Les temps de résolution des systèmes linéaires sur GPU, comparés aux temps de résolution sur CPU utilisant le même type de solveur, sont jusqu'à 3.5 fois plus courts pour calculer un lissage de surface, 7.5 fois plus courts pour calculer une paramétrisation de surface et enfin 11.5 fois plus courts pour la résolution de l'équation de pression d'une simulation d'écoulement fluide. Pour des systèmes linéaires d'une taille raisonnable, dans le cas de la résolution de l'équation de pression, le CNC sur GPU s'est avéré être largement plus rapide que des librairies de solveurs classiques comme TAUCS (solveur direct) ou LASPACK (solveur itératif) et a rivalisé avec UMFPACK (solveur direct). En outre, pour des systèmes linéaires de grande taille, les solveurs directs ne fonctionnent pas du fait de leur consommation mémoire trop importante. Cette limite ne s'applique pas à notre solveur puisque celui-ci est de type itératif.

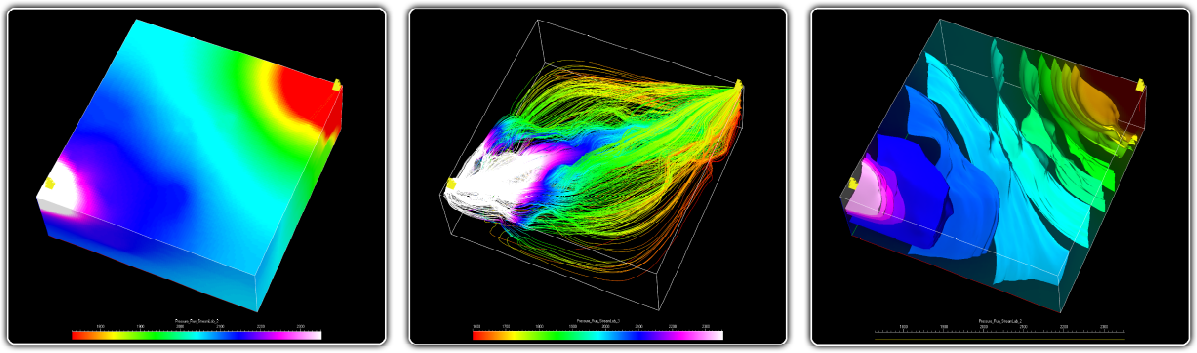


FIG. 2 – Gauche : pression calculée à l'aide du CNC sur GPU dans un modèle synthétique de réservoir pétrolier. Centre : saturation en eau calculée analytiquement sur des lignes de courant (ou Streamlines) à partir notamment de la pression. Droite : série de surfaces d'iso-pressions extraites de la grille volumique de gauche.

Le CNC permet de résoudre un grand nombre de problèmes d'optimisation non-structurés efficacement sur GPU. Il permet ainsi, par exemple, de simuler l'écoulement de fluides dans un réservoir pétrolier, ce qui crée au final un grand volume de données nécessitant d'être visualisées à des fins d'analyses (figure 2). Ces données représentent en général des propriétés physiques (pression, perméabilité, saturation...) qui sont définies sur la grille de simulation, et par conséquent, dans notre cadre d'étude, sur des grilles fortement non-structurées. Comme nous l'avons montré, l'utilisation de telles grilles impose de définir de nouveaux algorithmes de visualisation qui soient en mesure de s'adapter à la variabilité de la topologie et de la géométrie de celles-ci. C'est pourquoi nous avons développé l'algorithme du Marching Cells (section 1.2). Celui-ci définit un algorithme accéléré à l'aide d'un GPU capable d'extraire efficacement des isosurfaces à partir de grilles fortement non-structurées, ce qui est totalement inédit (figure 3). Cet algorithme outrepasse de nombreuses limites propres aux approches précédentes grâce à l'utilisation des textures en tant que vecteurs de stockage. Celles-ci permettent de stocker efficacement l'ensemble des données qui représentent une grille sans introduire de redondance dans le stockage des sommets partagés entre plusieurs cellules. L'utilisation de textures permet également de limiter au maximum les transferts mémoire sur le bus graphique PCI-Express. Le Marching Cells définit un générateur automatique de tables d'index qui généralise l'algorithme du Marching Cubes à tout type de cellules ainsi qu'une méthode qui propose de réduire le nombre de tables à conserver à l'aide d'isomorphismes de cellules.

En pratique, côté CPU, nous disposons actuellement d'une méthode générique complète d'extraction qui supporte des grilles fortement non-structurées. Côté GPU, nous disposons actuellement d'un code spécifique qui permet de trianguler des cellules tétraédriques et hexaédriques.

Le Marching Cells prend en charge le traitement sur GPU de très grandes grilles structurées ou non. Nous l'avons validé jusqu'à 10 millions de tétraèdres et 5 millions d'hexaèdres, tout en améliorant significativement les performances par rapport à la même implantation sur CPU. Les facteurs d'accélération relevés sont en moyenne respectivement de 7 fois et 5.5 fois, pour des grilles tétraédriques et hexaédriques. En outre, nous avons démontré la viabilité et l'efficacité de

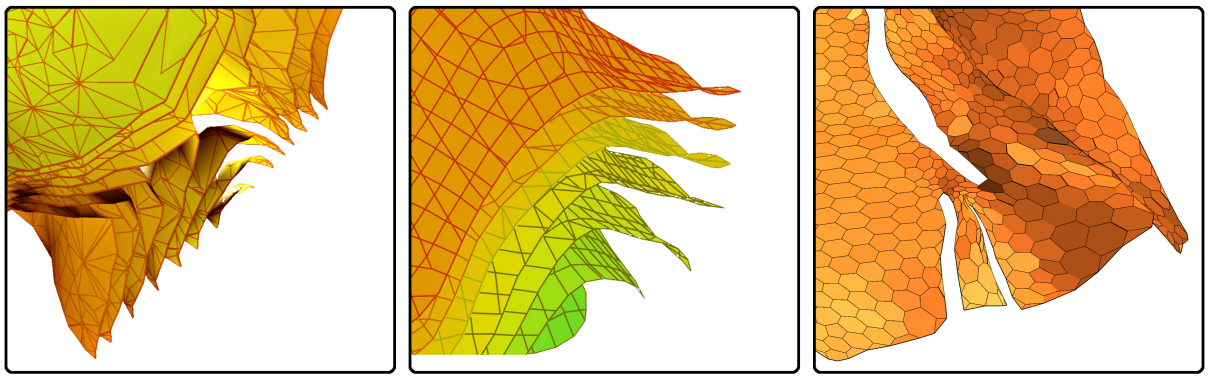


FIG. 3 – Notre algorithme du Marching Cells permet d'extraire des isosurfaces sur n'importe quel type de grille, même fortement non-structurées. Cette figure montre des isosurfaces extraites à partir d'une grille tétraédrique (gauche), hexaédrique (centre) et prismatique (droite).

l'extraction directe à partir de cellules fortement non-structurées –par exemple sur des grilles prismatiques– comparée aux approches qui pré-subdivisent ces mêmes cellules en tétraèdres pour appliquer ensuite des méthodes plus classiques de visualisation.

Le Marching Cells possède également l'avantage de pouvoir être aisément combiné, pour une plus grande efficacité, avec des algorithmes de classification (section 1.3). Ces algorithmes utilisent une structure de données pré-calculée afin de pré-sélectionner, en amont de la phase d'extraction, les seules cellules intersectées pour une isovaleur donnée. Nous avons proposé deux structures adaptatives de classification dont la construction s'est avérée rapide, la consommation mémoire certes un peu élevée, mais qui ont surtout permis d'accélérer environ 5 fois le processus d'extraction d'isosurfaces comparé à un parcours exhaustif des cellules. Comparé à la meilleure méthode de classification que nous avons testé, notre méthode adaptative est environ 20% plus rapide.

L'étude de phénomènes physiques via, par exemple, la réalisation de simulations d'écoulement fluide, comporte naturellement une dimension temporelle. Le stockage des valeurs des propriétés étudiées pour de nombreux pas de temps pose dès lors un problème de stockage. En pratique, seul un nombre restreint de pas de temps est conservé. Du fait de ce nombre limité, l'analyse d'un tel jeu de données à l'aide d'isosurfaces est relativement délicate.

La méthode de morphing 4D présentée à la section 1.4 propose ainsi d'améliorer la perception de l'évolution de nos données en autorisant la génération d'une infinité d'isosurfaces calculées par propagation et interpolation pour des temps arbitraires, cette interpolation bénéficiant d'une accélération sur GPU (figure 4). Nous avons montré que le coût de l'interpolation restait tout à fait raisonnable et que l'utilisation de notre méthode permettait des facteurs d'accélération –comparés à un parcours exhaustif– compris entre 1.5x et 8.5x selon les cas. En pratique, ces performances permettent une bonne interactivité même sur de gros jeux de données, justement là où un parcours exhaustif est insuffisant. La méthode offre donc un bon compromis entre qualité et rapidité d'extraction, tout en augmentant considérablement la perception de l'évolution des données au cours du temps.

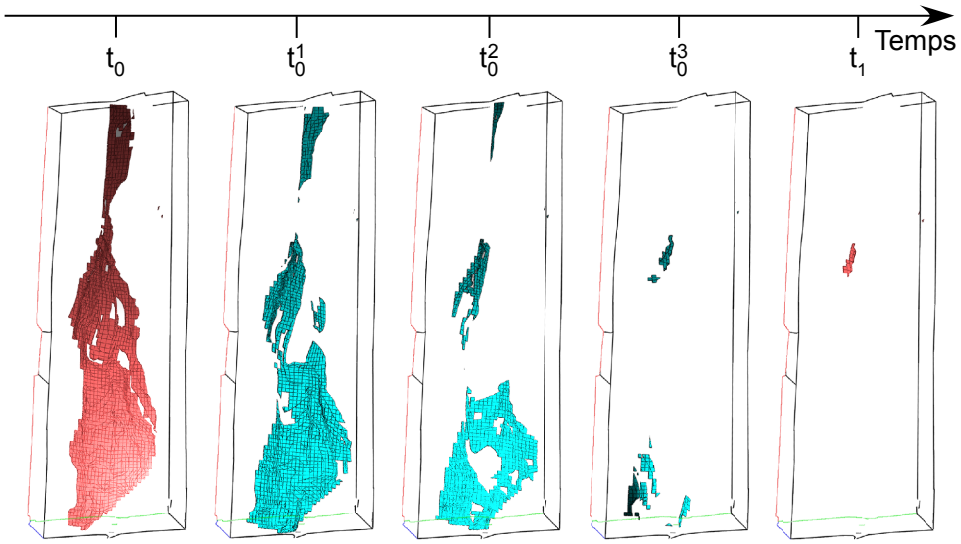


FIG. 4 – Isosurfaces extraites sur GPU pour différents pas de temps. Les isosurfaces en rouges correspondent à des pas de temps exacts disponibles dans le jeu de données initial, tandis que les isosurfaces en bleues sont calculées par interpolation.

Perspectives

Actuellement les cartes graphiques ne supportent que les nombres flottants en simple précision. C'est pourquoi notre solveur numérique CNC vise dans un premier temps les applications qui ne se focalisent pas sur la précision des résultats mais plutôt sur la haute-performance des calculs réalisés. Malgré tout, il serait tout à fait envisageable d'utiliser notre solveur sur GPU combiné à un solveur sur CPU afin de réaliser un schéma de résolution en précision-mixte (Göddeke *et al.*, 2005, 2007; Strzodka et Göddeke, 2006). Cette approche permettrait de bénéficier à la fois de la puissance de calcul des GPU et de la double précision des CPU. En outre, NVIDIA a récemment indiqué que leur prochaine génération de cartes graphiques attendue au second semestre 2008, le G100, supportera nativement les nombres flottants en double précision.

Le solveur que nous avons implanté est un Gradient Conjugué préconditionné par Jacobi, ce qui limite son champ d'application à la résolution de systèmes symétriques. Néanmoins, il serait tout à fait envisageable et aisé d'implanter un Gradient Bi-Conjugué (biCG) ou bien sa version à convergence améliorée nommée biCGSTAB au sein du CNC afin de résoudre des systèmes linéaires non-symétriques. En effet, le format de matrice creuse BCRS que nous utilisons au sein du CNC est totalement générique et supporte aisément des matrices non-symétriques. L'implantation d'un biCGSTAB au sein du CNC permettrait de bénéficier de toute la puissance d'un GPU pour résoudre des systèmes non-symétriques. L'implantation d'un solveur direct sur GPU est également une direction de recherche très importante à explorer. Celle-ci pourrait déboucher sur des solveurs d'une rapidité inégalée.

Notre CNC implante un Gradient Conjugué parallélisé soit sur un CPU multi-cores (grâce au SSE3 et au multi-threading) soit sur un seul GPU. Par conséquent, il serait intéressant d'étudier

l'extension de la parallélisation de notre solveur non plus à un seul processeur massivement parallèle, mais à un ensemble de processeurs parallèles (GPU ou CPU) localisés ou non dans un même boîtier.

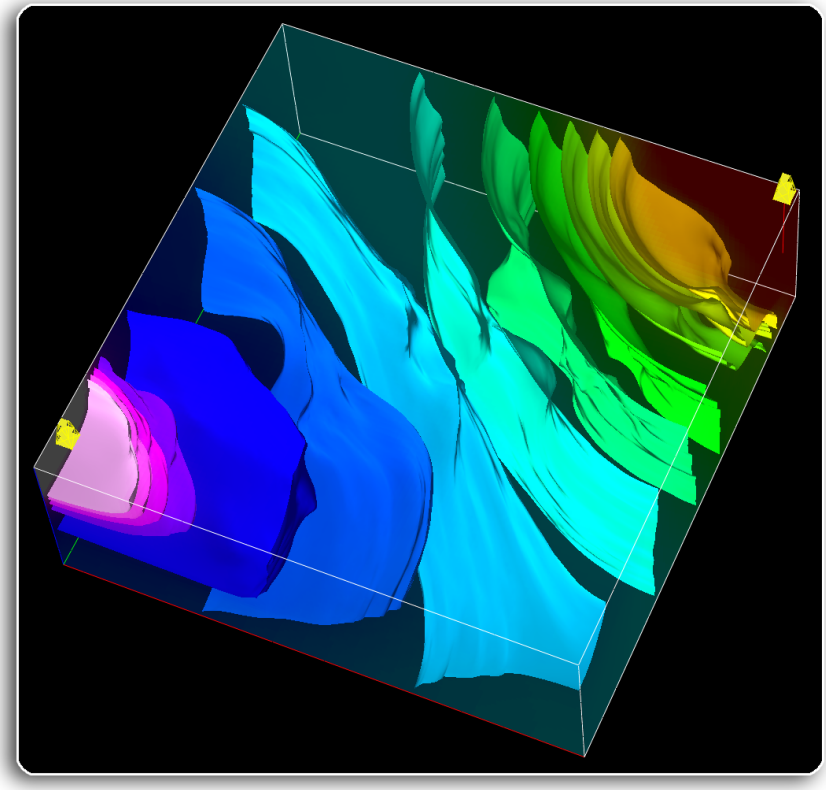
Par ailleurs, notre solveur creux générique sur GPU sera très prochainement disponible publiquement sous la forme d'une librairie. Celle-ci permettra à tout un chacun de la réutiliser, et d'exploiter la puissance des GPU qu'ils soient d'AMD-ATI ou d'NVIDIA afin de résoudre tout type de problème d'optimisation non-structuré.

Côté visualisation, utiliser les Geometry Shaders serait une évolution majeure de notre méthode d'extraction d'isosurfaces sur GPU, le Marching Cells. Grâce à eux, notre algorithme pourrait gagner à la fois en efficacité et en simplicité. En effet, les Geometry Shaders permettraient de générer la géométrie de l'isosurface à extraire directement au niveau du GPU, ce qui limiterait à la fois la redondance des calculs et la bande-passante mémoire nécessaire. Ils permettraient également d'implanter plus aisément un moteur d'extraction de cellules fortement non-structurées qui soit totalement générique.

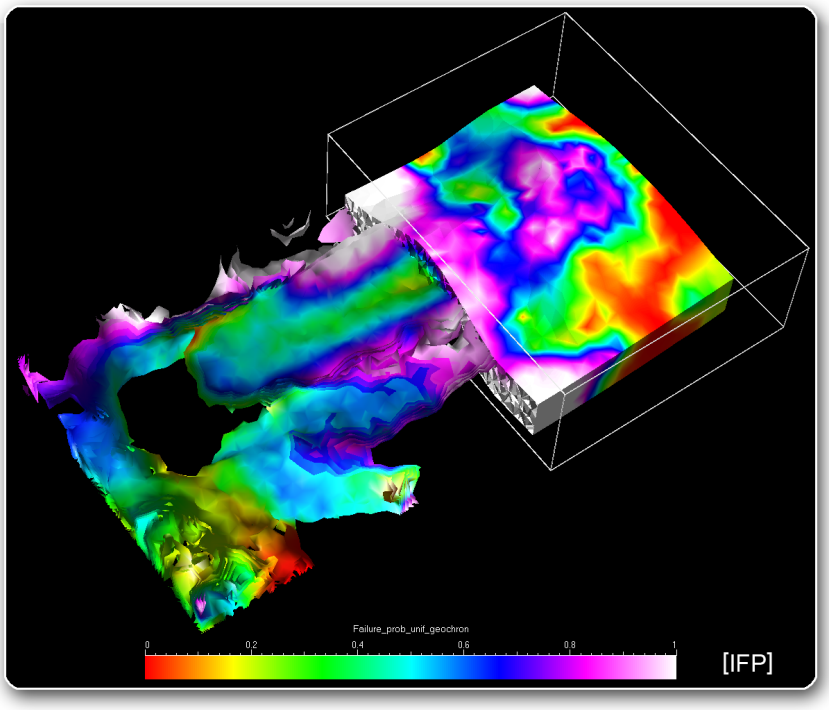
Le fonctionnement des méthodes de pré-classification de cellules opérant en amont de l'extraction d'isosurfaces pourrait être optimisé. En effet, avec l'implantation actuelle, lorsque le CPU est occupé, le GPU attend et vice versa (on parle d'exécution synchrone). Par conséquent, il serait bénéfique de rendre asynchrones les calculs CPU et GPU. De même, il serait intéressant d'étudier de nouveaux algorithmes de classification qui seraient implantables directement sur GPU, pourquoi pas à l'aide d'API telles que CUDA ou CTM.

De même que pour les algorithmes de classification, il serait possible d'améliorer les performances de notre méthode de Morphing 4D en désynchronisant les calculs effectués sur CPU de ceux effectués sur GPU. D'un point de vue qualitatif, il serait également intéressant de tester d'autres types d'interpolation que la simple interpolation linéaire que nous avons utilisé.

Table des figures



Liste des tableaux



Bibliographie

Abramowitz, M. et Stegun, I. A. (1972). *Handbook of Mathematical Functions*. National Bureau of Standards, Washington, D.C.

Alexa, M. (2002). Recent advances in mesh morphing. *Computer Graphics forum*, 21(2):173–197.

Anderson, E. W., Callahan, S. P., Scheidegger, C. E., Schreiner, J. et Silva, C. T. (2007). Hardware-assisted point-based volume rendering of tetrahedral meshes. *In Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, pages 163–170.

Aziz, K. et Settari, A. (1979). *Petroleum Reservoir Simulation*. Applied Science Publishers, Essex, England.

Bajaj, C. L., Pascucci, V. et Schikore, D. R. (1996). Fast isocontouring for improved interactivity. *Proceedings of the 1996 Symposium for Volume Visualization*.

Bajaj, C. L., Pascucci, V. et Schikore, D. R. (1999). Seed sets and search structures for optimal isocontour extraction. *Technical Report, University of Austin, Texas*.

Bajaj, C. L., Shamir, A. et Sohn, B.-S. (2002). Progressive tracking of isosurfaces in time-varying scalar fields. *Technical Report, University of Austin, Texas*.

Balaven, S., Bennis, C., Boissonnat, J. et Sarda, S. (2000). Generation of hybrid grids using power diagrams. *International Conference on Numerical Grid Generation in Computational Field Simulations*.

Balaven-Clermidy, S. (2001). *Génération de maillages hybrides pour la simulation des réservoirs pétroliers*. Thèse de doctorat, École des mines de Paris.

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. et der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems : Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA.

Batycky, R., Blunt, M. et Thiele, M. (1996). A 3d field scale streamline based reservoir simulator. *Proceedings of the SPE Annual Technical Conference and Exhibition, Denver, Colorado*, SPE 36726.

Batycky, R., Thiele, M. R. et Blunt, M. J. (1997). A streamline-based reservoir simulation of the house mountain waterflood. *In Proceedings of the SCRF Meeting*.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

Berenblyum, R., Shapiro, A. et Stenby, E. (2004). Reservoir streamline simulation accounting for effects of capillirity and wettability. *In 9th European Conference on the Mathematics of Oil Recovery, Cannes, France*.

Bloomenthal, J. (1988). Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4):53–60.

Blunt, M., Liu, K. et Thiele, M. (1996). A generalized streamline method to predict reservoir flow. *Petroleum Geoscience*, 2:256–269.

Bolz, J., Farmer, I., Grinspun, E. et Schröder, P. (2003). Sparse matrix solvers on the GPU : conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924.

Botsch, M., Bommes, D. et Kobbelt, L. (2005a). Efficient linear system solvers for mesh processing. *In IMA conference on Mathematics of Surfaces XI*, volume 3604 de *Lecture Notes in Computer Science*, pages 62–83.

Botsch, M., Bommes, D. et Kobbelt, L. (2005b). Efficient linear system solvers for mesh processing. *In IMA Mathematics of Surfaces XI, Lecture Notes in Computer Science*.

Buatois, L. et Caumon, G. (2005). 4d morph : Dynamic visualization of unstructured grids with continuous transitions between time steps. *Proceedings of the 25th Gocad Meeting, Nancy, France*.

Buatois, L., Caumon, G. et Lévy, B. (2006a). Gpu accelerated isosurface extraction on complex polyhedral grids. *Proceedings of the 26th Gocad Meeting, Nancy, France*.

Buatois, L., Caumon, G. et Lévy, B. (2006b). Gpu accelerated isosurface extraction on tetrahedral grids. *In Proceedings of the International Symposium on Visual Computing*. Lecture Notes in Computer Science (LNCS).

Buatois, L., Caumon, G. et Lévy, B. (2007a). Concurrent Number Cruncher : A GPU Implementation of a General Sparse Linear Solver. *Submitted to the International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*.

Buatois, L., Caumon, G. et Lévy, B. (2007b). Concurrent Number Cruncher : An Efficient Sparse Linear Solver on the GPU. *In High Performance Computation Conference 2007 (HPCC'07)*. Lecture Notes in Computer Science.

Buatois, L., Caumon, G. et Lévy, B. (2007c). Concurrent Number Cruncher : An Efficient Sparse Linear Solver on the GPU. *Proceedings of the 27th Gocad Meeting, Nancy, France*.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. et Hanrahan, P. (2004). Brook for gpus : stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786.

Cabral, B., Cam, N. et Foran, J. (1994). Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *In IEEE Symposium on Volume Visualization*, pages 91—98.

Callahan, D., Carr, S. et Kennedy, K. (1990). Improving register allocation for subscripted variables. *In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 53–65, White Plains, NY.

Callahan, D., Carr, S. et Kennedy, K. (2004). Improving register allocation for subscripted variables. *SIGPLAN Not.*, 39(4):328–342.

Callahan, S. P., Bavoil, L., Pascucci, V. et Silva, C. T. (2006). Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1307–1314.

Callahan, S. P., Ikits, M., Comba, J. L. D. et Silva, C. T. (2005). Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295.

Castanié, L. (2006). *Visualisation de Données Volumiques Massives, Applications aux Données Sismiques*. Thèse de doctorat, Institut National Polytechnique de Lorraine, Nancy, France.

Castanié, L., Lévy, B. et Bosquet, F. (2005). VolumeExplorer : Roaming large volumes to couple visualization and data processing for oil and gas exploration. *In IEEE Visualization*, page 32.

Caumon, G., Lévy, B., Castanié, L. et Paul, J.-C. (2005). Visualization of grids conforming to geological structures : a topological approach. *Computers and Geosciences, vol 32*.

Chernyaev, E. (1995). Marching cubes 33 : Construction of topologically correct isosurfaces. Rapport technique, CERN CN 95-17.

Chiang, W. et Kinzelbach, W. (2001). *3D Groundwater Modeling with PMWIN*. Springer.

Chiang, Y.-J. (2003). Out-of-core isosurface extraction of time-varying fields over irregular grids. *Proceedings IEEE Visualization '03 (Vis '03)*, pages 217–224.

Cignoni, P., Marino, P., Montani, C., Puppo, E. et Scopigno, R. (1997). Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170.

Cignoni, P., Montani, C. et Scopigno, R. (1998). Tetrahedra based volume visualization. *In* Hege, H.-C. et Polthier, K., éditeurs : *Mathematical Visualization*, pages 3–18. Springer-Verlag, Heidelberg.

Cognot, R. (1996). *La méthode D.S.I. : optimisation, implémentation et applications*. Thèse de doctorat, Institut National Polytechnique de Lorraine, Nancy, France.

Comba, J., Klosowski, J. T., Max, N., Mitchell, J. S. B., Silva, C. T. et Williams, P. L. (1999). Fast polyhedral cell sorting for interactive rendering of unstructured grids. *In* Brunet, P. et Scopigno, R., éditeurs : *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 369–376. The Eurographics Association and Blackwell Publishers.

Conreaux, S. (2001). *Modélisation de 3-variétés à base topologique : application à la géologie.* Thèse d'université, Institut National Polytechnique de Lorraine, Nancy, France.

Cuthill, E. et McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. *In Proceedings of the 1969 24th national conference*, pages 157–172, New York, NY, USA. ACM Press.

Cyril (2007). Opengl geometry shader marching cubes. *Icare3D.org.*

Dake, L. (2001). *Fundamentals of Reservoir Engineering.* Elsevier Science.

Delmal, P. (2001). *SQL2 - SQL3 - Applications à Oracle.* Bibliothèque des Universités - 3ème édition.

Drebin, R. A., Carpenter, L. et Hanrahan, P. (1988). Volume rendering. *In SIGGRAPH '88 : Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA. ACM.

Edelsbrunner, H. (1980). Dynamic data structures for orthogonal intersection queries. Rapport technique, Technical Report Rep. F59, Tech. Univ. Graz, Institute für Informationsverarbeitung.

Engel, K., Kraus, M. et Ertl, T. (2001). High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. *In Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, pages 9–16. Addison-Wesley Publishing Company, Inc.

Ertekin, T., Abou-Kassem, J. et King, G. (2001). *Basic Applied Reservoir Simulation.* Society of Petroleum Engineers.

Fatahalian, K., Sugerman, J. et Hanrahan, P. (2004). Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. *In HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA. ACM Press.

Fernando, R. et Kilgard, M. J. (2003). *The Cg Tutorial : The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fetel, E. (2007). *Quantification des incertitudes liées aux simulations d'écoulement dans un réservoir pétrolier à l'aide de surfaces de réponse non linéaires.* Thèse de doctorat, Institut National Polytechnique de Lorraine, Nancy, France.

Finkel, R. A. et Bentley, J. L. (1974). Quad trees : A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9.

Flandrin, N., Borouchaki, H. et Bennis, C. (2004). 3d hybrid mesh generation for reservoir flow simulation. *In roceedings, 13th International Meshing Roundtable, Williamsburg, VA, Sandia National Laboratories, SAND 2004-3765C*, pages 133–144.

Floater, M. S. et Hormann, K. (2005). *Surface Parameterization : a Tutorial and Survey*, pages 157–186. N. A. Dodgson and M. S. Floater and M. A. Sabin.

Frank, T. (2006). *Advanced Visualization and Modeling of Tetrahedral Meshes*. Thèse de doctorat, Institut National Polytechnique de Lorraine, Nancy, France - Freiberg University, Allemagne.

Fuchs, H., Kedem, Z. M. et Naylor, B. F. (1980). On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph.*, 14(3):124–133.

Fuchs, H., Kedem, Z. M. et Uselton, S. P. (1977). Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702.

Gall, D. L. (1991). Mpeg : a video compression standard for multimedia applications. *Commun. ACM*, 34(4):46–58.

Gallagher, R. S. (1991). Span filtering : an optimization scheme for volume visualization of large finite element models. *In VIS '91 : Proceedings of the 2nd conference on Visualization '91*, pages 68–75, Los Alamitos, CA, USA. IEEE Computer Society Press.

Galoppo, N., Govindaraju, N. K., Henson, M. et Manocha, D. (2005). LU-GPU : Efficient algorithms for solving dense linear systems on graphics hardware. *In SC '05 : Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA. IEEE Computer Society.

Garrity, M. P. (1990). Raytracing irregular volume data. *In VVS '90 : Proceedings of the 1990 workshop on Volume visualization*, pages 35–40, New York, NY, USA. ACM.

Gibbs, N., Poole, W. et Stockmeyer, P. (1974). An algorithm for reducing the bandwidth and profile of a sparse matrix. Rapport technique, College of William and Mary Williamsbourg, VA, Dept of Mathematics.

Giles, M. et Haimes, R. (1990). Advanced interactive visualization for CFD. *Computing Systems in Education*, 1(1):51–62.

Göddeke, D., Strzodka, R. et Turek, S. (2005). Accelerating double precision FEM simulations with GPUs. *In Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*.

Göddeke, D., Strzodka, R. et Turek, S. (2007). Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*. accepted for publication November 2006, to appear.

Guéziec, A. et Hummel, R. (1995). Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):328–342.

Hahmann, S., Bonneau, G.-P., Caramiaux, B. et Cornillac, M. (2007). Multiresolution morphing for planar curves. *Computing*, 79(2):197–209.

Harris, M. (2007). Parallel prefix sum (scan) with CUDA. Rapport technique, NVIDIA.

Hastreiter, P. (1999). *Registrierung und Visualisierung medizinischer Bilddaten unterschiedlicher Modalitäten*. Thèse de doctorat, Erlangen, Nürnberg University.

Hestenes, M. R. et Stiefel, E. (1952). Methods of Conjugate Gradients for Solving Linear Systems. *J. Research Nat. Bur. Standards*, 49:409–436.

Hormann, K., Lévy, B. et Sheffer, A. (2007). Mesh parameterization : theory and practice. *In SIGGRAPH '07 : ACM SIGGRAPH 2007 courses*, New York, NY, USA. ACM Press.

Im, E.-J. et Yelick, K. A. (2001). Optimizing sparse matrix computations for register reuse in sparsity. *In International Conference on Computational Science*, pages 127–136.

Itoh, T. et Koyamada, K. (1995). Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327.

Jung, J. H. et O'Leary, D. P. (2006). Cholesky decomposition and linear programming on a gpu. Scholarly Paper, University of Maryland.

Kajiya, J. T. et Herzen, B. P. V. (1984). Ray tracing volume densities. *In SIGGRAPH '84 : Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA. ACM.

Keane, A. (2006). CUDA (compute unified device architecture). *In http ://developer.nvidia.com/object/cuda.html*.

Kessenich, J., Baldwin, D. et Rost, R. (2003). The opengl shading language. Rapport technique, 3Dlabs.

Kilgard, M. (2003). Nvidia opengl extension specifications. Rapport technique, NVIDIA Corporation.

Kindlmann, G. et Durkin, J. W. (1998). Semi-automatic generation of transfer functions for direct volume rendering. *In VVS '98 : Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 79–86, New York, NY, USA. ACM.

Kipfer, P. et Westermann, R. (2005). Gpu construction and transparent rendering of isosurfaces. *In* Greiner, G., Hornegger, J., Niemann, H. et Stamminger, M., éditeurs : *Proceedings of Vision, Modeling and Visualization 2005*, pages 241–248. IOS Press, infix.

Klein, T., Stegmaier, S. et Ertl, T. (2004). Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. *In PG '04 : Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 186–195, Washington, DC, USA. IEEE Computer Society.

Krüger, J. et Westermann, R. (2003a). Acceleration techniques for gpu-based volume rendering. *In IEEE Visualization Conference*, pages 287—292.

Krüger, J. et Westermann, R. (2003b). Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916.

Lazarus, F. et Verroust, A. (1998). Three-dimensional metamorphosis : a survey. *The Visual Computer*, 14(8/9):373–389.

Lefebvre, S., Hornus, S. et Neyret, F. (2005). *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapitre Octree Textures on the GPU, pages 595–613. Addison Wesley.

Lepage, F. (2004). *Génération de Maillages Tridimensionnels pour la Simulation des Phénomènes Physiques en Géosciences.* Thèse de doctorat, Institut National Polytechnique de Lorraine, Nancy, France.

Levoy, M. (1988). Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8, n°5:29–37.

Levoy, M. (1990). Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261.

Lévy, B. (1999). *Topologie Algorithmique : Combinatoire et Plongement.* Thèse de doctorat, Institut National Polytechnique de Lorraine, Nancy, France.

Lévy, B. (2005). OpenNL (Open Numerical Library). *http ://www.loria.fr/∼levy/software/.*

Lévy, B. (2008). *Géométrie Numérique (HDR).* Thèse de doctorat, Institut National Polytechnique de Lorraine, Nancy, France.

Lévy, B., Caumon, G., Conreaux, S. et Cavin, X. (2001). Circular incident edge lists : a data structure for rendering complex unstructured grids. *In* Ertl, T., Joy, K. et Varshney, A., éditeurs : *IEEE Visualization.*

Lévy, B., Petitjean, S., Ray, N. et Maillot, J. (2002a). Least squares conformal maps for automatic texture atlas generation. *In* ACM, éditeur : *SIGGRAPH 02, San-Antonio, Texas, USA.*

Lévy, B., Petitjean, S., Ray, N. et Maillot, J. (2002b). Least squares conformal maps for automatic texture atlas generation. *In SIGGRAPH 02, San-Antonio, Texas, USA.*

Livnat, Y. et Hansen, C. (1998). View dependent isosurface extraction. *In* Ebert, D., Hagen, H. et Rushmeier, H., éditeurs : *IEEE Visualization '98*, pages 175–180.

Livnat, Y., Shen, H.-W. et Johnson, C. R. (1996). A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics, 2 :73-84*.

Lorensen, W. E. et Cline, H. E. (1987). Marching cubes : A high resolution 3d surface construction algorithm. *ACM SIGGRAPH'87*.

Mallet, J. (1992). Discrete Smooth Interpolation in geometric modeling. *Computer Aided Design*, 24(4):177–191.

Mallet, J. (1997). Discrete modeling for natural objects. *Mathematical Geology*, 29:199–219.

Mallet, J. (2002). *Geomodeling*. Oxford University Press, New York, NY, U.S.A.

Marchesin, S., Dischler, J.-M. et Mongenet, C. (2004). 3d roam for scalable volume visualization. *In VV '04 : Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, pages 79–86, Washington, DC, USA. IEEE Computer Society.

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

Max, N., Hanrahan, P. et Crawfis, R. (1990). Area and volume coherence for efficient visualization of 3D scalar functions. volume 24, pages 27–33.

McCool, M. et DuToit, S. (2004). *Metaprogramming GPUs with Sh*. AK Peters.

McCreight, E. M. (1980). Efficient algorithms for enumerating intersecting intervals and rectangles. Rapport technique, Xerox Palo Alto Research Center, Palo Alto, CA, USA.

McCreight, E. M. (1985). Priority search trees. *SIAM J. Comput. 14, 257-276*.

McKay, B. (1981). Practical graph isomorphism. *In Numerical mathematics and computing, Proc. 10th Manitoba Conf., Winnipeg/Manitoba 1980, Congr. Numerantium 30*, pages 45–87.

Mehlhorn, K. (1984). *Data structures and algorithms 3 : multi-dimensional searching and computational geometry*. Springer-Verlag New York, Inc., New York, NY, USA.

Meshar, O., Irony, D. et Toledo, S. (2006). An out-of-core sparse symmetric indefinite factorization method. *ACM Transactions on Mathematical Software*, 32:445–471.

Microsoft Corporation (2002). High-level shader language. directx 9.0 graphics. Rapport technique, http ://msdn.microsoft.com/directx.

Microsoft Corporation (2006). Direct3d reference. *msdn.microsoft.com*.

Montani, C., Scateni, R. et Scopigno, R. (1994). A modified look-up table for implicit disambiguation of marching cubes. *The Visual Computer*, 10(6):353–355.

Muller, D. E. et Preparata, F. P. (1978). Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236.

Mulmuley, K. (1994). *Computational geometry : An introduction through randomized algorithms*. Prentice Hall, Englewood Cliffs, NJ.

Muron, P., Tertois, A.-L., Mallet, J. et Hovadik, J. (2005). An Efficient and Extensible Interpolation Framework Based on the Matrix Formulation of the Discrete Smooth Interpolation. *In Proceedings of the 25th Gocad Meeting, Nancy, France*.

Natarajan, B. K. (1994). On generating topologically consistent isosurfaces from uniform samples. *Vis. Comput.*, 11(1):52–62.

Nealen, A., Igarashi, T., Sorkine, O. et Alexa, M. (2006). Laplacian mesh optimization. *In Proceedings of ACM GRAPHITE*, pages 381–389.

Nielson, G. M. et Hamann, B. (1991). The asymptotic decider : Rosolving the ambiguity in marching cubes. *In* Nielson, G. M. et Rosenblum, L. J., éditeurs : *IEEE Visualization*, pages 83–91. IEEE, IEEE Computer Society Press.

Nielson, G. M. et Sung, J. (1997). Interval volume tetrahedrization. *In VIS '97 : Proceedings of the 8th conference on Visualization '97*, page 221, Los Alamitos, CA, USA. IEEE Computer Society Press.

Pascucci, V. (2002). Slow growing subdivision (sgs) in any dimension : Towards removing the curse of dimensionality. *Comput. Graph. Forum*, 21(3).

Pascucci, V. (2004). Isosurface Computation Made Simple : Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. *In IEEE TVCG Symposium on Visualization '04*, pages 293–300.

Peercy, M., Segal, M. et Gerstmann, D. (2006). A performance-oriented data-parallel virtual machine for gpus. *In ACM SIGGRAPH'06*.

Pfister, H., Lorensen, B., Bajaj, C., Kindlmann, G., Schroeder, W., Avila, L. S., Martin, K., Machiraju, R. et Lee, J. (2001). The transfer function bake-off. *IEEE Comput. Graph. Appl.*, 21(3):16–22.

Preparata, F. P. et Shamos, M. I. (1985). *Computational Geometry : An Introduction*. Springer-Verlag, New York, NY.

Press, W., Flannery, B., Teukolsky, S. et Vetterling, W. (1992). *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press.

Reck, F., Dachsbacher, C., Grosso, R., Greiner, G. et Stamminger, M. (2004). Realtime isosurface extraction with graphics hardware. *In Proceedings of Eurographics 2004*.

Rezk-Salama, C. (2001). *Volume Rendering Techniques for General Purpose Graphics Hardware*. Thèse de doctorat, Universität Erlangen-Nürnberg.

Rezk-Salama, C., Engel, K., Hadwiger, M., Kniss, J. M., Lefohn, A. E. et Weiskopf, D. (2004). Real-time volume graphics. *In SIGGRAPH '04 : ACM SIGGRAPH 2004 Course Notes*, page 29, New York, NY, USA. ACM.

Rost, R. J. (2004). *OpenGL Shading Language*. Addison-Wesley Professional.

Röttger, S., Guthe, S. et Weiskopf, D. (2003). Smart hardware-accelerated volume rendering. *In Eurographics/IEEE Symposium on Visualization*, pages 231—238.

Sabella, P. (1988). A rendering algorithm for visualizing 3d scalar fields. *In SIGGRAPH '88 : Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 51–58, New York, NY, USA. ACM.

Samet, H. (1984). The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260.

Segal, M. et Akeley, K. (2004). The OpenGL graphics system : A specification, version 2.0. *www.opengl.org*.

Shen, H.-W. (1998). Isosurface extraction in time-varying fields using a temporal hierarchical index tree. *In VIS '98 : Proceedings of the conference on Visualization '98*, pages 159–166, Los Alamitos, CA, USA. IEEE Computer Society Press.

Shen, H.-W., Hansen, C. D., Livnat, Y. et Johnson., C. R. (1996). Isosurfacing in span space with utmost efficiency (issue). *In IEEE Proceedings of Visualization'96*.

Shen, H.-W. et Johnson, C. R. (1995). Sweeping simplices : A fast iso-surface extraction algorithm for unstructured grids. *In VIS '95 : Proceedings of the 6th conference on Visualization '95*, page 143, Washington, DC, USA. IEEE Computer Society.

Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. Rapport technique, CMU School of Computer Science. ftp ://warp.cs.cmu.edu/quake-papers/painless-conjugate-gradient.ps.

Shirley, P. et Tuchman, A. A. (1990). A polygonal approximation to direct scalar volume rendering. *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, 24(5):63–70.

Silva, C., Comba, J., Callahan, S. et Bernardon, F. (2005). A survey of gpu-based volume rendering of unstructured grids. *Revista de Informatica Teorica e Aplicada*, 12(2):pp. 9–29.

Silva, C. T. et Mitchell, J. S. B. (1997). The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):142–157.

Silva, C. T., Mitchell, J. S. B. et Williams, P. L. (1998). An exact interactive time visibility ordering algorithm for polyhedral cell complexes. *In VVS '98 : Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 87–94, New York, NY, USA. ACM.

Sorkine, O. et Cohen-Or, D. (2004). Least-squares meshes. *In Proceedings of Shape Modeling International*, pages 191–199. IEEE Computer Society Press.

Strzodka, R. et Göddeke, D. (2006). Pipelined mixed precision algorithms on fpgas for fast and accurate pde solvers from low precision components. *In FCCM '06 : Proceedings*

of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06), pages 259–270, Washington, DC, USA. IEEE Computer Society.

Sutton, P. et Hansen, C. D. (1999). Isosurface extraction in time-varying fields using a temporal branch-on-need tree (t-bon). *In VIS '99 : Proceedings of the conference on Visualization '99*, pages 147–153, Los Alamitos, CA, USA. IEEE Computer Society Press.

Sutton, P. et Hansen, C. D. (2000). Accelerated isosurface extraction in time-varying fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107.

Taubin, G. (1995). A signal processing approach to fair surface design. *In SIGGRAPH Conference Proceedings*, pages 351–358. ACM.

Thiele, M., Batycky, R., Blunt, M. et Orr, F. J. (1994). Simulating flow in heterogeneous systems using streamtubes and streamlines. *Proceedings of the SPE/DOE Improved Oil Recovery Symposium, Tulsa, Oklahoma*, SPE 27834:5–12.

Upson, C. et Keeler, M. (1988). V-buffer : visible volume rendering. *In SIGGRAPH '88 : Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 59–64, New York, NY, USA. ACM.

Uralsky, Y. (2006). Practical metaballs and implicit surfaces. *In Game Developper Conference.*

Van Gelder, A. et Wilhelms, J. (1994). Topological considerations in isosurface generation. *ACM Transactions on Graphics*, 13(4):337–375.

Van Kreveld, M. (1996). Efficient methods for isoline extraction from a tin. *GIS, 10 :523-540.*

Van Kreveld, M., van Oostrum, R., Bajaj, C. L., Schikore, D. R. et Pascucci, V. (1997). Contour trees and small seed sets for isosurface traversal. *Proceedings Thirteenth ACM Symposium on Computational Geometry.*

Wallis, J., Miller, T., Lerner, C. et Kleerup, E. (1989). Three-dimensional display in nuclear medicine. *IEEE Transactions on Medical Imaging*, Volume 8, Issue 4:297–230.

Weigle, C. et Banks, D. C. (1998). Extracting iso-valued features in 4-dimensional scalar fields. *In VVS '98 : Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 103–110, New York, NY, USA. ACM Press.

Westover, L. (1989). Interactive volume rendering. *In Chapell Hill Volume Visualization Workshop*, pages 9–16.

Westover, L. (1990). Footprint evaluation for volume rendering. *In SIGGRAPH '90 : Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, New York, NY, USA. ACM.

Wilhelms, J., Challinger, J., Alper, N., Ramamoorthy, S. et Vaziri, A. (1990). Direct volume rendering of curvilinear volumes. *In Workshop on Volume visualization*, pages 41–47, New York, NY, USA. ACM.

Wilhelms, J. et Van Gelder, A. (1990). Octrees for faster isosurface generation. *In Workshop on Volume visualization*, pages 57–62, New York, NY, USA. ACM.

Wilhelms, J. et Van Gelder, A. (1992). Octrees for faster isosurface generation. *ACM Transaction on Graphics*, 11(3):201–227.

Williams, P. L. (1992). Visibility-ordering meshed polyhedra. *ACM Trans. Graph.*, 11(2):103–126.

Yagel, R., Reed, D. M., Law, A., Shih, P. et Shareef, N. (1996). Hardware assisted volume rendering of unstructured grids by incremental slicing. *In VVS '96 : Proceedings of the 1996 symposium on Volume visualization*, pages 55–62, Piscataway, NJ, USA. IEEE Press.

Résumé

Les algorithmes les plus récents de traitement numérique de la géométrie ou bien encore de simulation numérique de type CFD (Computational Fluid Dynamics) utilisent à présent de nouveaux types de grilles composées de polyèdres arbitraires, autrement dit des grilles fortement non-structurées. Dans le cas de simulations de type CFD, ces grilles peuvent servir de support à des champs scalaires ou vectoriels qui représentent des grandeurs physiques (par exemple : densité, porosité, perméabilité).

La problématique de cette thèse concerne la définition de nouveaux outils de visualisation et de calcul sur de telles grilles. Pour la visualisation, cela pose à la fois le problème du stockage et de l'adaptativité des algorithmes à une géométrie et une topologie variables. Pour le calcul, cela pose le problème de la résolution de grands systèmes linéaires creux non-structurés. Pour aborder ces problèmes, l'augmentation incessante ces dernières années de la puissance de calcul parallèle des processeurs graphiques nous fournit de nouveaux outils. Toutefois, l'utilisation de ces GPU nécessite de définir de nouveaux algorithmes adaptés aux modèles de programmation parallèle qui leur sont spécifiques.

Nos contributions sont les suivantes : (1) Une méthode générique de visualisation tirant partie de la puissance de calcul des GPU pour extraire des isosurfaces à partir de grandes grilles fortement non-structurées. (2) Une méthode de classification de cellules qui permet d'accélérer l'extraction d'isosurfaces grâce à une pré-sélection des seules cellules intersectées. (3) Un algorithme d'interpolation temporelle d'isosurfaces. Celui-ci permet de visualiser de manière continue dans le temps l'évolution d'isosurfaces. (4) Un algorithme massivement parallèle de résolution de grands systèmes linéaires non-structurés creux sur le GPU. L'originalité de celui-ci concerne son adaptation à des matrices de motif arbitraire, ce qui le rend applicable à n'importe quel système creux, dont ceux issus de maillages fortement non-structurés.

Mots-clés: Maillages non-structurés, extraction d'isosurfaces, solveurs numériques creux, GPU

Abstract

Most recent algorithms for Geometry Processing or Computational Fluid Dynamics (CFD) are using new types of grids made of arbitrary polyhedra, in other words strongly unstructured grids. In case of CFD simulations, these grids can be mapped with scalar or vector fields representing physical properties (for example : density, porosity, permeability).

This thesis proposes new tools for visualization and computation on strongly unstructured grids. Visualization of such grids that have variable geometry and topology, poses the problem of how to store data and how algorithms could handle such variability. Doing computations on such grids poses the problem of solving large sparse unstructured linear systems. The ever-growing parallel power of GPUs makes them more and more valuable for handling theses tasks. However, using GPUs calls for defining new algorithms highly adapted to their specific programming model.

Our contributions are : (1) An efficient generic visualization method that uses GPU's power to accelerate isosurface extraction for large unstructured grids. (2) An adaptative cell classification method that accelerates isosurface extraction by pre-selecting only intersected cells. (3) An efficient algorithm for temporal interpolation of isosurfaces. This algrithm helps to visualize in a continuous maner the evolution of isosurfaces through time. (4) A massively parallel algorithm for solving large sparse unstructured linear systems on the GPU. Its originality comes from its adaptation to sparse matrices with random pattern, which enables to solve any sparse linear system, thus the ones that come from strongly unstructured grids.

Keywords: Unstructured grids, isosurface extraction, sparse numerical solvers, GPU

Centre de Recherches Pétrographiques et Géochimiques

Laboratoire d'Infographie et d'Analyse de Données

Rue du Doyen Marcel Roubault - 54500 Vandœuvre