Aceleração da Técnica de Cubos Marchantes para Visualização Volumétrica com Placas Gráficas

Marcos Vinicius Mussel Cirne
Instituto de Computação
Universidade Estadual de Campinas (UNICAMP)
Campinas, SP, Brasil, 13083-852
marcosvcirne@gmail.com

Helio Pedrini
Instituto de Computação
Universidade Estadual de Campinas (UNICAMP)
Campinas, SP, Brasil, 13083-852
helio@ic.unicamp.br
www.ic.unicamp.br/~helio

***Abstract*—Volume visualization has numerous applications that benefit different knowledge domains, such as Biology, Medicine, Oceanography, Geology, among others. With the continuous advances of technology, it has been possible to achieve considerable rendering rates at a high degree of realism. Visualization tools have currently assisted users with the visual analysis of complex and large data sets. Marching cubes is one of the most widely used real-time volume rendering methods. This work describes a method for speeding up the marching cubes algorithm on a graphics processing unit and discusses a number of ways to improve its performance by means of auxiliary spatial data structures. Experiments conducted with use of several volumetric data sets demonstrate the effectiveness of the developed method.**

***Resumo*—A visualização volumétrica possui uma série de aplicações que beneficiam diversos campos do conhecimento, como Biologia, Medicina, Oceanografia, Geologia, entre outros. Com os contínuos avanços tecnológicos, tem sido possível obter taxas de renderização consideráveis e um alto grau de realismo. Ferramentas de visualização têm auxiliado os usuários com a análise visual de conjuntos de dados grandes e complexos. Cubos marchantes é um dos métodos mais utilizados para renderização volumétrica em tempo real. Este trabalho descreve um método para a aceleração do algoritmo de cubos marchantes em uma unidade de processamento gráfico e discute uma série de maneiras de melhorar seu desempenho por meio de estruturas de dados espaciais auxiliares. Experimentos realizados com vários volumes de dados demonstram a eficácia do método desenvolvido.**

***Palavras-chave*-visualização volumétrica; cubos marchantes; dados volumétricos; placas gráficas.**

I. Introdução

As técnicas de visualização volumétrica têm permitido que os usuários explorem e analisem dados complexos em diversos domínios do conhecimento, tais como Biologia, Medicina, Oceanografia, Geologia, entre outros. Ferramentas de visualização oferecem funcionalidades para manipular e renderizar conjuntos de dados volumétricos, melhorando a compreensão visual de suas estruturas e padrões.

Um dos desafios da área de visualização volumétrica é o desenvolvimento de algoritmos eficientes para representar, manipular e renderizar conjuntos de dados grandes e complexos. Apesar dos significativos avanços nesse ramo, o uso de uma unidade de processamento central (CPU, do inglês *Central Processing Unit*) para desempenhar tarefas gráficas de propósito geral não tem sido suficiente para se atingir uma interatividade eficiente ou uma renderização em tempo real, principalmente quando são utilizados grandes conjuntos de dados. No entanto, os constantes progressos tecnológicos permitiram o surgimento de poderosas unidades de processamento gráfico (GPU, do inglês *Graphics Processing Unit*), as quais são capazes de renderizar modelos tridimensionais complexos e obter um alto grau de realismo.

O potencial das GPUs é impulsionado pelo seu alto grau de paralelismo e por sua habilidade de executar operações de primitivas geométricas e de ponto flutuante de uma forma mais rápida e eficiente. As GPUs têm sido utilizadas para a aceleração de várias aplicações, como dinâmica de fluidos, análise sísmica, reconstrução de imagens médicas e previsões climáticas, resultando em um ganho expressivo em relação às CPUs. Isso foi possível graças ao desenvolvimento da programação de propósito geral em GPUs (GPGPU, do inglês *General-Purpose Computing on Graphics Processing Units*), tornando-as mais flexíveis por meio de seu paralelismo e sua adaptabilidade na programação de aplicações.

Como consequência desses avanços tecnológicos, as técnicas de visualização volumétrica também evoluíram consideravelmente ao longo dos últimos anos. A renderização de volumes em tempo real acelerada por meio de GPUs tem se tornado uma ferramenta eficaz para visualizar e analisar volumes de dados.

Este trabalho apresenta uma metodologia para a aceleração em GPU de uma técnica de visualização volumétrica para extração de isossuperfícies, denominada cubos marchantes [1]. O ganho de desempenho do método é obtido por meio de estruturas de dados espaciais auxiliares. Resultados experimentais obtidos a partir de vários dados volumétricos mostram a eficácia do método proposto.

A principal contribuição deste trabalho está no desenvolvimento de uma aplicação capaz de manipular volumes de diferentes tamanhos e que são objetos de estudo em vários campos de conhecimento, o que permite a agregação de uma série de conhecimentos a partir desses volumes. Além disso, ela pode ser utilizada como um *framework* para a integração em diversos ambientes de visualização, obtendo-se altas taxas

Trabalho referente à Dissertação de Mestrado, financiada pela CAPES.

de renderização em tempo real.

Este resumo está organizado da seguinte forma: a Seção II mostra os conceitos mais relevantes sobre visualização volumétrica, bem como a técnica de cubos marchantes e as estruturas de dados espaciais utilizadas na sua aceleração. A Seção III descreve a metodologia proposta neste trabalho para a aceleração da técnica de cubos marchantes. A Seção IV faz uma discussão acerca dos resultados obtidos. Por fim, a Seção V mostra as conclusões e possibilidades de trabalhos futuros.

II. Conceitos Relacionados

Esta seção apresenta uma descrição geral acerca de visualização volumétrica, seguida por conceitos do algoritmo de cubos marchantes e uma breve descrição das estruturas de dados utilizadas para melhorar o desempenho desse algoritmo.

A. Visualização Volumétrica

Visualização volumétrica [2] consiste em uma série de técnicas utilizadas para se estudar objetos e fenômenos naturais provenientes de vários campos do conhecimento. A ideia básica dessas técnicas é realizar uma projeção bidimensional (geralmente em uma tela de computador) a partir de dados volumétricos tridimensionais, os quais são representados por um conjunto de elementos de volume, chamados *voxels*.

Os principais algoritmos de visualização volumétrica podem ser classificados em duas categorias: Renderização Direta de Volume (DVR, do inglês *Direct Volume Rendering*) e Renderização de Superfície (SF, do inglês *Surface Fitting*). A primeira se caracteriza pelo mapeamento direto de *voxels* no espaço da tela, sem o uso de primitivas geométricas como representações intermediárias, enquanto que a segunda consiste em estágios de extração de características e representação de isossuperfícies (superfícies que representam um conjunto de pontos com o mesmo valor escalar em um volume de dados, ou seja, $\{\{x, y, z\} \in \Re^3 : f(x, y, z) = h\}$, para uma posição 3-D (x, y, z) e para algum isovalor $h \in \Re$), as quais são renderizadas posteriormente para visualização. Essas isossuperfícies podem ser definidas por meio de primitivas de superfície (como polígonos) ou por um certo limiar.

Algumas das técnicas de DVR incluem: *Raycasting* [3], [4], *Splatting* [5], [6], *Cell-Projection* [7], [8] e *Shear-Warp* [9], [10]. Dentre as técnicas de SF, estão: Conexão de Contornos [11], [12] e Cubos Marchantes [1], [13], [14].

B. Cubos Marchantes

A técnica de cubos marchantes [1] utiliza uma abordagem de divisão e conquista na qual o volume de dados é processado a partir de suas células (*voxels*), que são equivalentes a cubos. Em cada célula, são verificadas as intersecções entre cada uma de suas 12 arestas e a isossuperfície nela contida. Os valores de cada vértice são comparados com um dado isovalor (valor comum associado às isossuperfícies contidas em um volume de dados), e então esses vértices são classificados como "dentro" ou "fora" da isossuperfície. O primeiro caso se aplica quando o valor de um vértice é maior ou igual ao isovalor dado e o segundo quando esse valor é menor que o isovalor. Uma vez definido o tipo de intersecção, uma aproximação da isossuperfície contida na célula é feita por meio de construção de triângulos. A Figura 1 mostra alguns resultados da execução do algoritmo de cubos marchantes para isovalores diferentes.

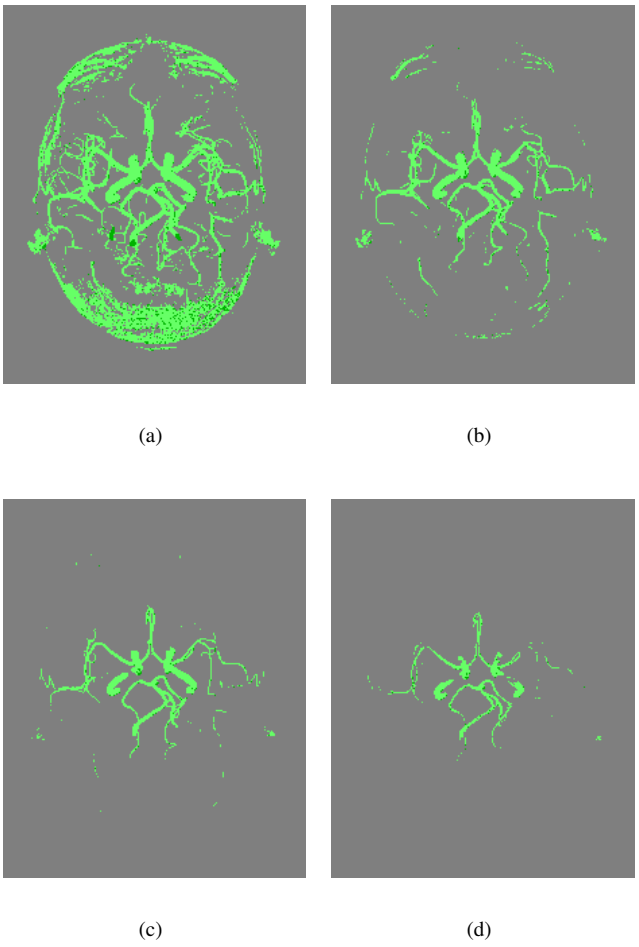


Figura 1. Renderização volumétrica de um volume de dados médico (angiografia) utilizando o algoritmo de cubos marchantes com isovalores diferentes: (a) 60; (b) 80; (c) 100; (d) 120.

Como cada um dos 8 vértices tem somente dois estados possíveis, temos então um total de $2^8 = 256$ casos de intersecção entre isossuperfície e aresta de célula, que são listados em uma tabela pré-calculada. No entanto, alguns pares desses casos são simétricos ou complementares entre si, o que restringe o problema a um total de 15 casos, ilustrados na Figura 2.

A principal vantagem dessa técnica reside no fato de que o processamento de uma célula é independente das demais, o que permite a sua paralelização. Entretanto, como desvantagem, ela pode gerar buracos nas isossuperfícies, devido a ambiguidades topológicas entre os casos. Esse problema pode ser solucionado por meio de uma extensão da tabela de casos [15], eliminando quaisquer ambiguidades. Porém, é

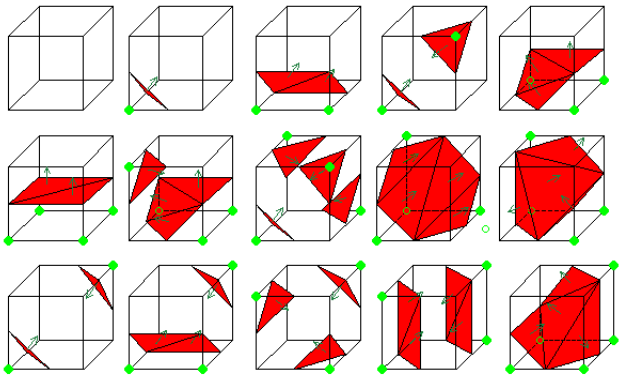


Figura 2. Ilustração dos 15 casos básicos do algoritmo de cubos marchantes. Os vértices verdes são aqueles classificados como "dentro" das isossuperfícies, e os demais como "fora" delas. Imagem extraída de [1].

um método um pouco mais custoso, pois gera uma quantidade maior de triângulos na renderização do volume de dados.

O pseudocódigo do algoritmo pode ser brevemente descrito conforme o Algoritmo 1. Ele é executado em tempo $O(n)$, em que n é o número total de células do volume de dados. Em geral, muitas das células processadas pelo algoritmo de cubos marchantes sao vazias, o que implica um gasto desnecessário de tempo.

Algoritmo 1 CubosMarchantes(V, h)

Entrada: Volume de dados V e um isovalor h.

Saída: Lista de vértices a serem renderizados, junto com suas respectivas normais.

1: **para** cada voxel (cubo) v de V **faça**
2: Calcule um índice para v, comparando os 8 valores escalares dos vértices de v com o isovalor h.
3: Utilizando o índice calculado, verifique a lista de arestas contidas em uma tabela pré-calculada.
4: Com base nos valores escalares em cada vértice da aresta, encontre as intersecções superfície-aresta por meio de interpolação linear.
5: Calcule uma normal unitária em cada vértice de v pelo método das diferenças centrais. Interpole a normal para cada vértice do triângulo.
6: Retorne os vértices dos triângulos e as normais dos vértices.
7: **fim para**

Em geral, muitas das células processadas pelo algoritmo de cubos marchantes são vazias, o que implica um gasto desnecessário de tempo. Para minimizar esse gasto, estruturas de dados espaciais são utilizadas para processar somente as células ativas, isto é, aquelas que são interceptadas por alguma isossuperfície, reduzindo a complexidade do algoritmo.

C. Aceleração com Estruturas de Dados Espaciais

Há várias classes de estruturas de dados que são bastante úteis para evitar o processamento de células vazias em um volume de dados. Uma delas consiste em representações baseadas em intervalos, que utilizam intervalos de células para agrupar as células do volume de dados [16]. A vantagem desse tipo de representação está em sua flexibilidade, sendo aplicada não apenas em grades regulares, mas também em grades irregulares, uma vez que ela trabalha a partir de um espaço de intervalos, ao invés de usar o próprio espaço da malha.

Os principais métodos dessa classe são baseados em uma representação denominada *span-space* [17], no qual cada célula do volume de dados é mapeada para um ponto bidimensional, cujas coordenadas x e y correspondem, respectivamente, aos valores mínimo e máximo da célula. A partir de um dado isovalor h, os pontos do *span-space* que representam as células ativas são aqueles onde $x \leq h$ e $y \geq h$.

Esta seção descreve algumas das estruturas de dados espaciais e como elas podem ser utilizadas para melhorar o desempenho do algoritmo de cubos marchantes.

1) k-d Tree: A *k-d tree* [18] é um caso especial da árvore binária de busca, sendo utilizada para organizar pontos localizados em um espaço k-dimensional. Cada nó interno representa um hiperplano que divide o espaço em duas partes em uma direção específica, que é definida de acordo com a profundidade desse nó na árvore. A subárvore esquerda contém todos os pontos localizados à esquerda do hiperplano e a subárvore direita contém aqueles à direita do hiperplano. Os nós-folha armazenam um único ponto cada.

No algoritmo de cubos marchantes, o volume de dados é mapeado em um *span-space* antes da construção da árvore, uma vez que as buscas na *k-d tree* são mais rápidas quando se trabalha com pontos localizados em um plano bidimensional ao invés de um espaço tridimensional. Além disso, cada nó armazena um ponto no *span-space*, em vez que armazenar os pontos apenas nas folhas. A construção da árvore gasta tempo $O(n \log n)$, onde n é o número total de células no volume de dados.

Quando é feita uma busca na árvore, dado um isovalor h, serão percorridos apenas os nós que correspondem às células ativas do volume de dados. Dessa forma, a busca gasta tempo $O(\sqrt{n} + p)$, onde p é o número de células ativas.

2) Interval Tree: A *interval tree* [19] é uma árvore ordenada, utilizada para armazenar intervalos de valores em uma única dimensão. Assim como a k-d *tree*, ela é uma extensão da árvore binária de busca e permite uma busca eficiente de todos os intervalos que sobrepõem um determinado intervalo ou ponto.

A raiz da árvore armazena um valor que corresponde à mediana das extremidades de todos os intervalos, além de uma lista de intervalos que contêm esse valor. A subárvore esquerda guarda os intervalos que estão completamente abaixo da mediana e a subárvore direita guarda os que estão completamente acima da mediana. Então, esse processo é repetido recursivamente para cada subárvore. O tempo total de construção é de $O(n \log n)$.

O *span-space* é adequado para a construção de uma *interval tree*. Nesse caso, os intervalos correspondem às células do

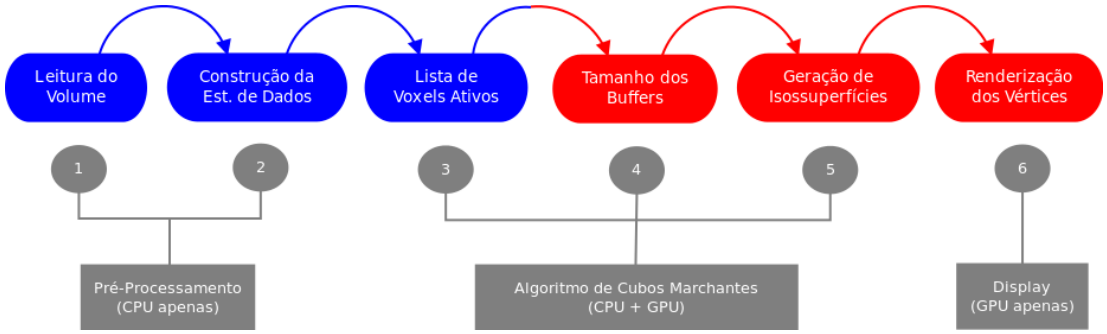


Figura 3. Abordagem utilizada para a aceleração do algoritmo de cubos marchantes. As etapas 1 e 2 são responsáveis pelo pré-processamento; as etapas 3 a 5 correspondem à execução do algoritmo de cubos marchantes e a etapa 6 faz a exibição dos vértices na tela. As caixas azuis representam as ações executadas na CPU, enquanto as caixas vermelhas representam as ações executadas na GPU.

volume de dados e as extremidades são os valores mínimo e máximo das células, que representam suas coordenadas no *span-space*.

A busca em uma *interval tree* requer tempo $O(\log n + p)$ (sendo p o número total de intervalos processados), o que a torna mais eficiente do que a k-*d tree*. Por outro lado, ela exige mais espaço de armazenamento.

3) Quadtree e Octree: A *quadtree* é uma estrutura de dados em que cada nó interno possui exatamente quatro filhos. Ela é usada para particionar uma determinada região localizada em um plano bidimensional em quatro regiões iguais (também chamadas de quadrantes). Essas regiões são então particionadas em outras quatro subregiões, e assim por diante, até que uma subregião fique vazia, o que caracteriza um nó folha da quadtree. A versão tridimensional análoga a essa estrutura é chamada de *octree*, que particiona um espaço tridimensional em oito regiões (ou octantes).

Como o *span-space* é um espaço bidimensional, ele pode ser representado por uma *quadtree*. Seja l o número de bits utilizados para armazenar os valores do volume de dados, o que significa que existem 2^l valores possíveis (variando entre 0 e $2^l - 1$) para um vértice de célula. Assim, o *span-space* é uma região de dimensões $2^l \times 2^l$, e os pontos correspondem às células mapeadas. Cada nó na árvore armazena a informação de um ponto no *span-space*.

No entanto, mais de uma célula pode ser mapeada para um mesmo ponto no *span-space*. Então, cada nó na *quadtree* também armazena um ponteiro para uma lista das células do volume de dados que foram mapeadas para aquele ponto. Então, as buscas podem ser efetuadas da maneira usual, percorrendo os nós associados às células ativas.

III. Metodologia

A metodologia proposta neste trabalho é composta por 6 etapas, como mostrado na Figura 3. Uma versão preliminar dessa metodologia pode ser vista em [20]. Na etapa 1, a CPU faz a leitura de um volume de dados, que é então alocado tanto na memória principal quanto na memória de vídeo. Em seguida, uma das estruturas de dados descritas na Seção II-C é construída a partir do volume e armazenada apenas na memória principal (Etapa 2).

Uma vez finalizada a etapa de pré-processamento, o algoritmo de cubos marchantes é então iniciado (Etapa 3). A partir de um isovalor h especificado pelo usuário, a CPU faz uma busca na estrutura de dados, percorrendo somente os nós que correspondem às células ativas do volume, gerando uma lista dessas células ativas, que por sua vez é transferida para a GPU por um barramento de comunicação.

Com a lista de células ativas e o isovalor h, a GPU dá continuidade ao algoritmo de cubos marchantes. Cada célula é classificada em um dos 15 casos de cubos marchantes comparando o valor de h aos valores dos 8 vértices da célula. Com isso, um índice de célula é criado e então usado para determinar o número de vértices necessários para renderizar a isossuperfície contida na célula. Uma vez que esse procedimento é feito para todas as células ativas, o número exato de vértices a serem renderizados pode ser determinado, o que definirá o tamanho exato da memória de vídeo necessária para alocar os *buffers* referentes a esses vértices (Etapa 4).

Na Etapa 5, a GPU percorre mais uma vez a lista de células ativas para gerar os triângulos que correspondem às isossuperfícies. Para cada célula, são calculadas as intersecções da isossuperfície com as 12 arestas da célula por meio de interpolações. Após obter todas as intersecções, a GPU escreve a lista de vértices (juntamente com suas respectivas normais) nos *buffers* alocados na etapa anterior. Finalmente, o volume de dados é renderizado a partir desses *buffers* (Etapa 6).

Todos os procedimentos executados pela GPU são paralelizados, uma vez que os resultados obtidos para uma célula são independentes daqueles das demais. Entretanto, a aceleração do algoritmo de cubos marchantes depende da maneira que essa paralelização ocorre. Quando uma tarefa é atribuída à GPU, ela cria uma quantidade específica de blocos[1], que contêm o mesmo número de *threads*, responsáveis por executar uma parte dessa tarefa.

O número de blocos depende diretamente do número de células ativas e do número de *threads* por bloco, também chamado de *tamanho do bloco*. O tamanho do bloco é escolhido de uma maneira que ele não seja nem muito baixo, atribuindo uma alta quantidade de trabalho para todas as

[1]Conceito extraído do Modelo de Programação CUDA [21].

threads e não maximizando o desempenho no geral, e nem muito alto, causando uma sobrecarga de inicialização e encerramento de *threads*.

IV. Resultados Experimentais

Os testes foram executados em um processador AMD Phenom II X6 1090T de 3.2 GHz com 4 GB de memória RAM e uma placa gráfica NVIDIA GeForce GTS 450 com 1 GB de memória de vídeo, utilizando a linguagem de programação C e as APIs OpenGL e CUDA. A Tabela IV mostra uma lista dos volumes que foram utilizados nos testes, além de suas respectivas dimensões (em voxels), isovalores e o número de triângulos renderizados na tela (o que depende do isovalor).

Nome do Volume	Dimensões	Isovalor	Nº Triângulos
Combustível	$64 \times 64 \times 64$	10	11534
Átomo de Hidrogênio	$128 \times 128 \times 128$	20	47864
Angiografia	$256 \times 320 \times 128$	80	84974
Ventrículos	$256 \times 256 \times 124$	120	167214
Motor	$256 \times 256 \times 128$	155	207592

Tabela I
Lista de volumes de dados com seus respectivos tamanhos, isovalores e número de triângulos gerados.

A Tabela IV mostra a taxa de quadros média da execução do algoritmo de cubos marchantes em CPU e GPU para todas as estruturas de dados mencionadas na Seção II-C, comparando à versão força bruta do algoritmo (que não utiliza nenhuma estrutura de dados para aceleração). Como se pode observar, a *interval tree* forneceu os melhores resultados, não apenas entre as estruturas de dados utilizadas nos testes, mas também no fator de aceleração comparado à versão força bruta de cubos marchantes em CPU, obtendo uma aceleração de 16.4 vezes, no caso do volume de dados da angiografia. Esse resultado era esperado porque a *interval tree* possui, assintoticamente, um tempo melhor de busca em relação às outras estruturas. Na média, o fator de aceleração entre CPU e GPU para a mesma estrutura de dados ficou entre 2.0 e 6.0.

A Figura 4 mostra dois gráficos que ilustram a média de tempo de 50 execuções da implementação proposta do algoritmo de cubos marchantes em GPU para diferentes valores de tamanho de bloco, utilizando cada um dos volumes de dados e seus respectivos valores listados na Tabela IV. O gráfico (a) representa as execuções da versão força bruta do algoritmo de cubos marchantes (que é executada sem a aceleração com estruturas de dados) e (b) as execuções que utilizam uma *interval tree* para armazenar os voxels provenientes dos volumes.

A partir de ambos os gráficos, pode-se notar que, para tamanhos de bloco maiores que 64, o tempo médio de execução do algoritmo de cubos marchantes tem uma pequena alta. Isto não depende de usar ou não uma estrutura de dados para a aceleração do algoritmo, tampouco do tipo de estrutura de dados utilizado. Em outras palavras, o comportamento das curvas dos gráficos é semelhante para qualquer estrutura. Conforme especificado na Seção III, embora um número alto de *threads* seja sinônimo de uma alta paralelização, o tempo gasto para criá-las também torna-se maior, o que estabiliza o desempenho do algoritmo de uma forma geral. Assim, o tamanho de bloco utilizado para executar os testes de desempenho da técnica de cubos marchantes foi definido em 64.

O gargalo do método apresentado está no fato de que as buscas nas estruturas de dados são feitas na CPU. Uma vez que os algoritmos de busca em árvore são geralmente recursivos e a arquitetura CUDA não tem suporte à recursão, se versões não-recursivas desses algoritmos fossem implementadas em GPU, haveria um grande consumo de tempo e de espaço para criar pilhas e laços utilizados para simular as chamadas recursivas.

V. Conclusões e Trabalhos Futuros

Este trabalho descreveu um método para a aceleração da técnica de cubos marchantes em GPU. A análise de desempenho foi feita usando-se diferentes estruturas de dados espaciais, de maneira que a complexidade de tempo do algoritmo dependa das células do volume de dados que contêm uma isossuperfície, em vez do número total de células. Os resultados experimentais demonstram que é possível acelerar o algoritmo de cubos marchantes por um fator de aproximadamente 16 vezes quando comparada a uma abordagem padrão executada com CPU.

Uma das possibilidades de trabalhos futuros é a extensão do método proposto para plataformas abertas, como o OpenCL [22], uma vez que CUDA é restrita a placas gráficas da NVIDIA [21]. Além disso, espera-se que, futuramente, as técnicas de visualização volumétrica em geral sejam aceleradas com o auxílio da nova geração de processadores, que integram CPU e GPU em um mesmo circuito, permitindo que ambos tenham acesso aos mesmos recursos de *hardware*. Assim, o gargalo existente na comunicação entre as duas unidades é eliminado, o que pode melhorar ainda mais o desempenho dessas técnicas.

Referências

[1] W. Lorensen and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163–169, Jul. 1987.

[2] R. Fuchs and H. Hauser, "Visualization of Multi-Variate Scientific Data," *Computer Graphics Forum*, vol. 28, no. 6, pp. 1670–1690, 2009.

[3] M. Levoy, "Efficient Ray Tracing of Volume Data," *ACM Transactions on Graphics*, vol. 9, no. 3, pp. 245–261, Jul. 1990.

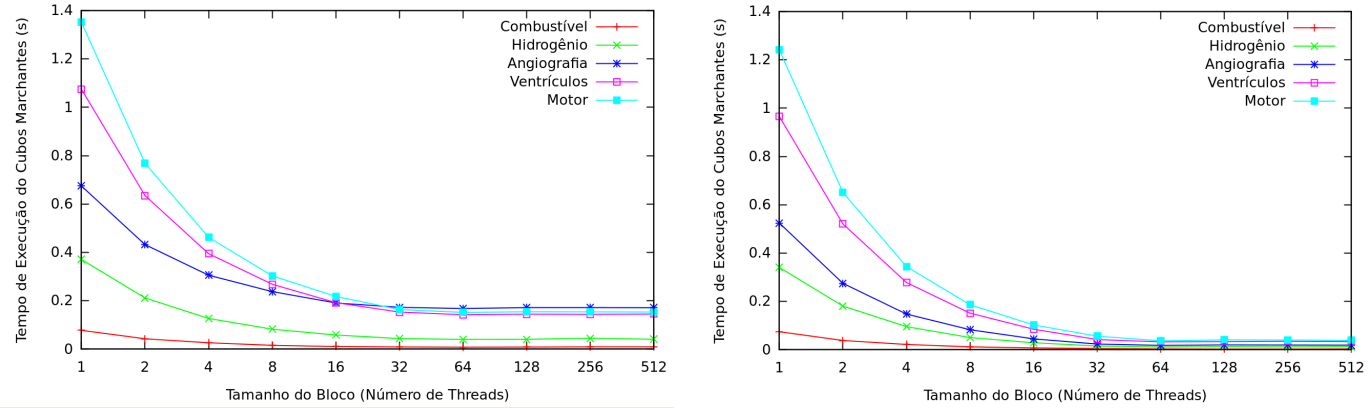
[4] B. Liu, G. Clapworthy, and F. Dong, "Accelerating Volume Raycasting using Proxy Spheres," *Computer Graphics Forum*, vol. 28, no. 3, pp. 839–846, Jun. 2009.

[5] P. Schlegel and R. Pajarola, "Layered Volume Splatting," in *Advances in Visual Computing*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5876, pp. 1–12.

[6] L. Westover, "Footprint Evaluation for Volume Rendering," in *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, Dallas, TX, USA, 1990, pp. 367–376.

[7] J. Wilhelms, "A Coherent Projection Approach for Direct Volume Rendering," University of California at Santa Cruz, Santa Cruz, CA, USA, Tech. Rep., 1990.

[8] S. Zhu and Y.-L. Gu, "Volume Rendering Algorithm of Irregular Volume based on Cell Projection," *Computer Engineering and Applications*, vol. 44, no. 15, pp. 68–70, May 2008.



(a) (b)

Figura 4. Tempos médios de execução do algoritmo de cubos marchantes em GPU para diferentes tamanhos de bloco. O gráfico (a) mostra os resultados para a versão força bruta e (b) para a *interval tree*.

CPU					
Volume	**Força Bruta**	**k-d Tree**	**Interval Tree**	**Quadtree**	**Octree**
Combustível	82.4	106.0 (1.3)	**111.5 (1.4)**	107.5 (1.3)	104.7 (1.3)
Hidrogênio	14.8	25.5 (1.7)	**32.1 (2.2)**	30.2 (2.0)	28.8 (1.9)
Angiografia	3.6	8.2 (2.3)	**11.5 (3.2)**	10.1 (2.8)	9.2 (2.6)
Ventrículos	4.0	7.7 (1.9)	**10.8 (2.7)**	10.1 (2.5)	9.4 (2.3)
Motor	3.7	6.4 (1.7)	**8.9 (2.4)**	8.4 (2.3)	7.7 (2.1)
GPU					
Volume	**Força Bruta**	**k-d Tree**	**Interval Tree**	**Quadtree**	**Octree**
Combustível	94.6 (1.1)	149.5 (1.8)	**178.9 (2.2)**	170.4 (2.1)	146.9 (1.8)
Hidrogênio	25.3 (1.7)	67.6 (4.6)	**88.5 (6.0)**	87.3 (5.9)	49.7 (3.4)
Angiografia	6.1 (1.7)	39.2 (10.9)	**59.0 (16.4)**	51.0 (14.2)	28.2 (7.8)
Ventrículos	6.8 (1.7)	20.7 (5.2)	**32.4 (8.1)**	27.5 (6.9)	19.7 (4.9)
Motor	6.4 (1.7)	17.5 (4.7)	**28.5 (7.7)**	22.1 (6.0)	16.9 (4.6)

Tabela II
TAXA MÉDIA DE QUADROS POR SEGUNDO DA EXECUÇÃO DO ALGORITMO DE CUBOS MARCHANTES EM CPU E EM GPU, COM DIFERENTES ESTRUTURAS DE DADOS.

[9] P. Lacroute, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1996.

[10] T. Li, M. Xie, W. Zhao, and Y. Wei, "Shear-Warp Rendering Algorithm Based on Radial Basis Functions Interpolation," in *Second International Conference on Computer Modeling and Simulation*, Jan. 2010, pp. 425–429.

[11] E. Keppel, "Approximating Complex Surfaces by Triangulation of Contour Lines," *IBM Journal of Research and Development*, vol. 19, no. 1, pp. 2–11, 1975.

[12] K. Nurzyńska, "3D Object Reconstruction from Parallel Cross-Sections," in *Computer Vision and Graphics*, ser. Lecture Notes in Computer Science, L. Bolc, J. Kulikowski, and K. Wojciechowski, Eds. Springer, 2009, vol. 5337, pp. 111–122.

[13] F. Gong and X. Zhao, "Three-Dimensional Reconstruction of Medical Image Based on Improved Marching Cubes Algorithm," in *International Conference on Machine Vision and Human-Machine Interface*, Kaifeng, China, Apr. 2010, pp. 608–611.

[14] Z. Wang, B. Fan, N. Li, and H. Zhang, "Iso-surface Extraction and Optimization Method Based on Marching Cubes," in *International Conference on Semantics, Knowledge and Grid*, Zhuhai, China, Oct. 2009, pp. 458–460.

[15] T. Lewiner, H. Lopes, A. W. Vieira, and G. Tavares, "Efficient Implementation of Marching Cubes' Cases with Topological Guarantees," *Journal of Graphics Tools*, vol. 8, p. 2003, 2003.

[16] T. Newman and H. Yi, "A Survey of the Marching Cubes Algorithm," *Computers & Graphics*, vol. 30, no. 5, pp. 854–879, Oct. 2006.

[17] Y. Livnat, H.-W. Shen, and R. Johnson, "A Near Optimal Isosurface Extraction Algorithm Using the Span Space," *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73–84, Mar. 1996.

[18] J. Bentley, "Multidimensional Binary Search Trees used for Associative Searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[19] P. Cignoni, P. Marino, C. Montani, and R. Scopigno, "Speeding up Isosurface Extraction using Interval Trees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, pp. 158–170, 1997.

[20] M. Cirne and H. Pedrini, "Acceleration of the Marching Cubes Technique for Volumetric Visualization on Graphics Processing Unit Using Spatial Data Structures," in *Proceedings of the XVII Simpósio Brasileiro de Sistemas Multimídia e Web*, Florianópolis, SC, Brazil, 2011, (Prêmio de Menção Honrosa como Melhor Artigo Completo).

[21] "NVIDIA CUDA - Parallel Programming and Computing Platform," 2012, http://developer.nvidia.com/what-cuda/.

[22] "OpenCL," 2012, the Khronos Group – http://www.khronos.org/opencl/.