UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Paralelização em GPU da Segmentação Vascular com Extração de *Centerlines* por *Height Ridges*

Ítalo Mendes da Silva Ribeiro

Orientador: Prof. Dr. Selan Rodrigues dos Santos

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Sistemas e Computação da UFRN (área de concentração: Computação Gráfica) como parte dos requisitos para obtenção do título de Mestre em Ciências.

Natal, RN, Agosto de 2011

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra - CCET.

Ribeiro, Ítalo Mendes da Silva.
Paralelização em GPU da segmentação vascular com extração de *centerlines* por *Height Ridges* / Ítalo Mendes da Silva Ribeiro. - Natal, RN, 2011.
85 f. ; il.

Orientador: Prof. Dr. Selan Rodrigues dos Santos.

Dissertação (Mestrado) - Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Computação gráfica - Medicina - Dissertação. 2.Diagnóstico por imagem - Segmentação vascular - Dissertação. 3. Vasos sanguíneos - *Centerlines* - Dissertação. 4. Imagens médicas - *Height ridges* - Dissertação. 5. Arquitetura CUDA - Dissertação. I. Santos, Selan Rodrigues dos. II. Título.

RN/UF/BSE-CCET CDU 004.92:61

Paralelização em GPU da Segmentação Vascular com Extração de *Centerlines* por *Height Ridges*

Ítalo Mendes da Silva Ribeiro

Dissertação de Mestrado aprovada em 2 de março de 2011 pela banca examinadora composta pelos seguintes membros:

Prof. Dr. Selan Rodrigues dos Santos (orientador) . . . . . DIMAP/UFRN

Prof. Dr. Ricardo Cordeiro de Farias . . . . . LCG/UFRJ

Prof. Dr. Bruno Motta de Carvalho . . . . . DIMAP/UFRN

Aos meus pais, Solistícios e Lúcia,
minha irmã, Alessandra e minha
namorada Simone.

Agradecimentos

Primeiramente a Deus.

Ao meu orientador, Prof. Dr. Selan Rodrigues dos Santos pela atenção e orientação.

Aos meus pais pelo apoio aos meus estudos.

A minha namorada Simone pela paciência e compreensão.

À CAPES, pelo apoio financeiro.

Ao Prof. Dr. Bruno Motta pelas sugestões e por auxiliar no contato com o Dr. Lauro Sousa.

Ao Dr. Lauro Sousa por ter cedido algumas imagens médicas.

Ao Prof. Dr. Ricardo Farias pelas informações sobre CUDA.

Resumo

A segmentação vascular é importante no diagnóstico de doenças como o acidente vascular cerebral e é dificultada por ruídos na imagem e vasos muito finos que não são vistos. Uma maneira de realizar a segmentação é extraindo a *centerline* do vaso com *height ridges*, que usa a intensidade como características para a segmentação. Este processo pode levar de segundos a minutos, dependendo da tecnologia atual empregada. O método é implementado em GPU, ou seja, é executado de maneira paralela em placa gráfica. O desempenho do método de segmentação executado em GPU é comparado com o mesmo método em CPU e o método original de Aylward em execução também na CPU. O melhoramento do novo método sobre o original é dupla. O ponto de partida para o processo de segmentação não é um único ponto no vaso sanguíneo, mas um volume, tornando assim mais fácil para o usuário a seleção de uma região de interesse, e, o ganho do método proposto foi 873 vezes mais rápido sendo executado em GPU e 150 vezes mais rápido sendo executado em CPU do que o original de Aylward em CPU.

Palavras-chave: Computação gráfica, medicina, diagnóstico por imagem, segmentação vascular, vasos sanguíneos, *centerlines*, imagens medicas, *height ridges*, GPU, arquitetura CUDA.

Abstract

The vascular segmentation is important in diagnosing vascular diseases like stroke and is hampered by noise in the image and very thin vessels that can pass unnoticed. One way to accomplish the segmentation is extracting the centerline of the vessel with height ridges, which uses the intensity as features for segmentation. This process can take from seconds to minutes, depending on the current technology employed. In order to accelerate the segmentation method proposed by Aylward [Aylward & Bullitt 2002] we have adapted it to run in parallel using CUDA architecture. The performance of the segmentation method running on GPU is compared to both the same method running on CPU and the original Aylward's method running also in CPU. The improvemente of the new method over the original one is twofold: the starting point for the segmentation process is not a single point in the blood vessel but a volume, thereby making it easier for the user to segment a region of interest, and; the overall gain method was 873 times faster running on GPU and 150 times more fast running on the CPU than the original CPU in Aylward.

Keywords: Computer graphics, medicine, diagnostic imaging, vascular segmentation, blood vessels, centerlines, medical images, height ridges, GPU, CUDA.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A Visualização Científica é uma área da Computação Gráfica que transforma símbolos em geometria, possibilitando que pesquisadores analisem suas simulações e processamentos, assim enriquecendo a descoberta científica [McCormick et al. 1987].

São várias as áreas de aplicação da Visualização Científica, como por exemplo: Cartografia, Geologia, Meteorologia, Bioquímica, Engenharia Mecânica, Nano Aplicações e Medicina [Taylor 2000]. A Visualização Médica (VM) consiste em apresentar exames médicos de maneira a extrair mais dados, ou seja, permitir que o usuário consiga perceber informações, ou estabelecer conexões e tendências, que de outra forma não seria possível perceber através da imagem original.

O conhecimento acerca dos dados auxilia a avaliação médica de tecidos e funções orgânicas do corpo humano, normais e anormais, para obtenção máxima de informações de imagens médicas e diagnosticar doenças o mais cedo possível, com maior precisão. As imagens médicas para a visualização são geradas a partir de exames de Ressonância Magnética (RM), Tomografia Computadorizada (TC), Raio-X, Ultra-sonografia e etc.

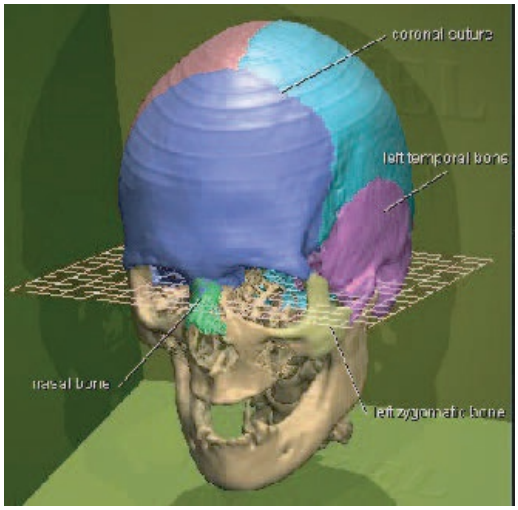
As principais aplicações da Visualização Médica são:

- **Educacional:** Comumente utilizado em aulas de anatomia para substituir os atlas do corpo humano por modelos tridimensionais (3D). Outras formas de se utilizar a VM são em animações para apresentar o funcionamento dos órgãos e visualização simultânea de vários sistemas do corpo. Estas aplicações são úteis para o entendimento dos mesmos. Na Figura 1.1a é visto um crânio e a indicação de algumas de suas partes.

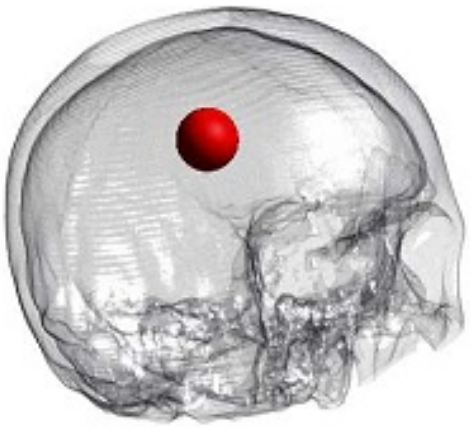
- **Diagnóstico:** Existem várias doenças e problemas que têm diagnósticos facilitados através da análise de imagens médicas, como hipertensão arterial [Linguraru et al. 2008], arteriosclerose [Gao & Zhang 2009] e fraturas [Rahman et al. 2009]. Por não ser invasivo, é uma ótima maneira de atingir o diagnóstico correto sem riscos de causar problemas à saúde do paciente, como localizar um tumor no cérebro (Figura 1.1b), ou mesmo intervenções incômodas ao paciente como colonoscopia ou endoscopia, que podem ser substituídas pela colonoscopia e endoscopia virtuais. O médico também pode obter informações como a espessura de uma artéria, o tamanho de um tumor, o fluxo sanguíneo, a gravidade de uma fratura, através de funcionalidades implementadas nos sistemas de visualização, que utilizam métodos de análise e extração dos dados.

- **Planejamento de Tratamento:** Através de um ambiente 3D criado com os dados obtidos, é possível realizar uma simulação dos efeitos que podem acontecer no paciente com a aplicação de um medicamento. Com a injeção de uma substância no paciente, é possível simular o caminho de percorrimento, velocidade, problemas ou efeitos colaterais. Em tratamento de fraturas, torna-se mais fácil decidir o tratamento e a forma como ele será executado. Isso também ocorre em cirurgias neurológicas, abdominais e cardíacas, por exemplo.

- **Suporte Intraoperativo:** O cirurgião tem ajuda para orientação, navegação e localização dentro do corpo do paciente, com modelos 3D vistos em um monitor. A Figura 1.2a mostra o modelo criado antes da cirurgia, que é observado pelo médico durante o procedimento, como na Figura 1.2b.



(a) Crânio



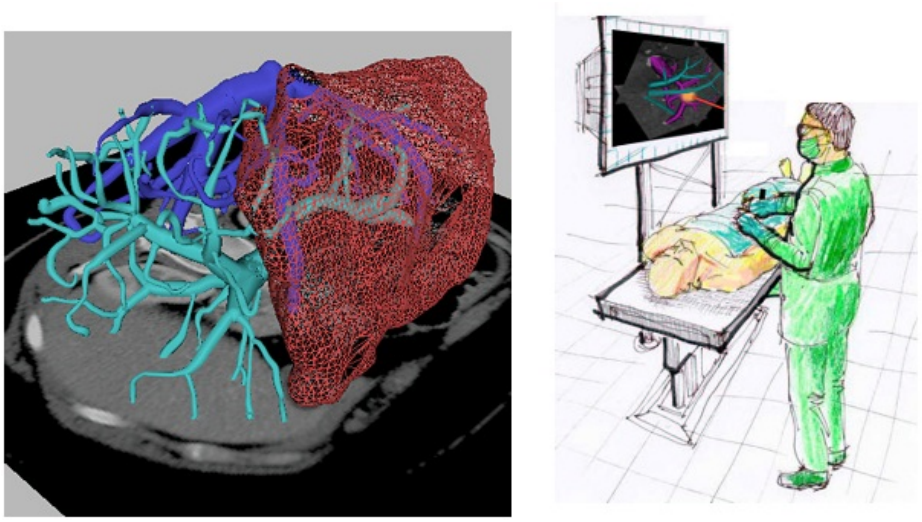
(b) Localização de um Tumor

Figura 1.1: (a) Crânio e algumas de suas partes [Preim & Bartz 2007]. (b) Localização de um tumor (esfera vermelha) no cérebro [Rúbio 2003].

Dentre as quatro categorias de aplicação descritas, estamos mais interessados na categoria de diagnóstico por imagem. Considere, por exemplo, o angiograma que é um exame de diagnóstico por imagem utilizado como avaliação para várias enfermidades de causa vascular. O angiograma consiste em injetar um contraste radiopaco através de um catéter em uma veia ou artéria, para realçar essas estruturas e facilitar a sua observação pelo médico.

O angiograma e outros exames são usados no tratamento do infarto do miocárdio, estenose, retinopatia diabética [Staal et al. 2004], retinopatia da prematuridade [Heneghan et al. 2002], acidente vascular cerebral, aneurisma, embolia pulmonar, detecção de região de mácula, má formação de vasos, diâmetro de vasos para diagnóstico de hipertensão [Martínez-Perez et al. 2002].

No entanto, a estrutura vascular em certas situações é difícil de ser visualizada. Quando o vaso é muito fino ou a sua intensidade é muito semelhante à de tecidos vizinhos ou existe



(a) Crânio (b) Localização de um Tumor

Figura 1.2: (a) Modelo 3D dos órgãos do paciente criados antes da cirurgia. (b) Cirurgião durante o procedimento. [Aylward et al. 2002]

ruído próximo, a forma anatômica do vaso confunde-se com a forma de bordas ou outros componentes da imagem. O vaso então torna-se quase imperceptível a olhos médicos, podendo levar a um diagnóstico falho.

Uma forma de facilitar o processo de segmentação, reduzindo os erros de diagnóstico, é o uso de ferramentas baseadas em técnicas de segmentação de imagens médicas, tornando o processo semi-automático. Estas ferramentas auxiliam o processo de separação da estrutura vascular dos demais tecidos e de acordo com a segmentação a ser realizada, é necessária a indicação de pontos iniciais para a segmentação. Porém, a seleção dos pontos iniciais pode ser é um fator limitante no resultado da segmentação. As condições para que os pontos iniciais sejam bons para um ótimo resultado, às vezes os tornam difíceis de serem determinados pelo usuário, como pontos que devem estar os mais próximos possíveis do centro do vaso objetivado [Abeysinghe & Ju 2009]. Para contornar este problema, uma possível solução seria selecionar uma região e não apenas um ponto, com isso seriam processados todos os pontos da região, o que necessita de muito processamento. Este grande volume de processamento seria então executado em paralelo, para se evitar um maior tempo para obtenção da segmentação

As ferramentas de auxílio à análise de imagens médicas implementam métodos de segmentação que usam conceitos como região de crescimento, casamento de filtros, contornos ativos e *height ridges*. Esta última abordagem usada em [Eberly et al. 1994], visa encontrar pontos na imagem cujo valor local seja maior que seus vizinhos. Estes pontos correspondem a picos na imagem. Em imagens médicas, tais picos podem ser relacionados a vasos sanguíneos, uma vez que a intensidade dos pontos pertencentes a vasos aumentam das bordas para o centro (pico). Por fim, métodos que seguem a abordagem *height ridges* selecionam os pontos de pico de maior intensidade de maneira a identificar

linhas centrais ou *centerlines*, que provavelmente correspondem a vasos sanguíneos. As linhas centrais são formadas por pontos que se encontram exatamente no centro do vaso e são úteis para determinar a estrutura vascular e medições como a espessura do vaso.

As técnicas de segmentação baseadas em CPU (*Central Processing Unit*) podem demorar às vezes muitos segundos, ou mesmo minutos para os padrões de equipamentos atuais. No entanto, com um menor tempo de execução para obtenção dos resultados, o diagnóstico pode ser dado mais rapidamente e possivelmente o atendimento ao paciente será dado em um intervalo de tempo mais curto. O tempo de processamento dessas imagens pode ser reduzido com a sua paralelização e uso de uma arquitetura apropriada. CUDA (*Compute Unified Device Archtecture*) é uma arquitetura que proporciona programação genérica em GPU (*Graphics Processing Unit*), permitindo o processamento em paralelo de informações diretamente na placa gráfica, informações que comumente seriam executadas em CPU.

Considerando as questões levantadas anteriormente, o presente trabalho propõe a paralelização do algoritmo de segmentação vascular baseado em *height ridges* de Aylward com o objetivo de (i) facilitar a seleção do objeto desejado por parte do usuário e (ii) diminuir o tempo para a segmentação das linhas centrais.

O método foi executado sobre dados artificiais, que simulavam a intensidade dos vasos e em imagens médicas da região peitoral, mais especificamente do coração e vasos próximos. Para validação do trabalho foi medido o tempo de execução, para obtenção das linhas centrais em função da quantidade de voxels processados a cada milissegundos. O ganho do método proposto foi 873 vezes mais rápido sendo executado em GPU e 150 vezes mais rápido sendo executado em CPU do que o original de Aylward em CPU.

Este texto está organizado da seguinte forma, no Capítulo 2 introduzimos alguns conceitos básicos, necessários para o entendimento deste trabalho. O capítulo 3 apresenta detalhes da implementação relacionados à GPU. O Capítulo 4 contém a metodologia utilizada para avaliar o trabalho, cujos resultados são discutidos no capítulo 5. Por fim, o Capítulo 6 apresenta as conclusões desta dissertação juntamente com algumas indicações de trabalho futuros.

Trabalhos Relacionados

As *centerlines* podem ser geradas por pós-processamento, de modo explícito ou implícito [Aylward & Bullitt 2002]. Os métodos de pós-processamento realizam inicialmente algum tipo de tarefa antes da obtenção das *centerlines*, como por exemplo um limiarização e é normalmente utilizado quando a identificação das linhas centrais é feita voxel por voxel. Em [Wilson & Noble 1999] é realizado uma limiarização no histograma para separação delas. No trabalho de [Verscheure et al. 2010] uma separação da estrutura vascular do cérebro é executada e em seguida é feita uma esqueletonização para determinação das *centerlines*, usando o algoritmo de Dijkstra [Wan et al. 2002]. [Egger et al. 2007] também usa o Dijkstra, mas antes realiza uma limiarização, em seguida coloca o ponto inicial determinado pelo usuário no centro do vaso, usando um poliedro de forma recursiva e no final é necessária uma suavização e correção da posição da linha central.

Para medição do comprimento de vasos abdominais [Babin et al. 2009] segmenta as

centerlines identificando os pontos de maior distância dos pontos pertencentes ao fundo da imagem. O comprimento e o volume das coronários é medido em [Casciaro et al. 2010]. Um dos principais problemas dos métodos que utilizam esqueletonização é a necessidade da limiarização anterior, que é uma segmentação muito simples e sofre muito com ruídos que prejudica os resultados das *centerlines*. Os métodos de pós-processamento podem demandar muito tempo e necessitar de muito processamento para apresentar as linhas centrais, visto que para alcançar o resultado muitas etapas podem ser feitas até a segmentação propriamente dita das *centerlines*.

Os métodos implícitos não objetivam a determinação das linhas centrais, mas podem ser usados para tal. Estes métodos são focados para a separação de estruturas tubulares. Dentre os métodos estão os de regiões de crescimento [Yim et al. 2000], contornos ativos [McInemey & Terzopoulos 1999], casamento de filtros [Shang et al. 2010] e *level-sets* [Hongmei et al. 2003]. Em [Bansal et al. 2010] o casamento de filtros com muitas rotações é feito. O trabalho de [Lorigo et al. 2000] usa *level-sets* para separação de estruturas tubulares e com bons resultados para segmentação de vasos. Com [Pock et al. 2005] a determinação das linhas centrais é conseguido executando um filtro sobre a imagem para separar a estrutura tubular e em seguida *level-sets*. As *centerlines* obtidas por *level-sets* estão sendo aplicadas também para descoberta de problemas na colonoscopia virtual [Van Uitert et al. 2006, Van Uitert & Summers 2007]. Um outro exemplo de *level-sets* usado para *centerlines* é [Xu et al. 2009]. Estes métodos que utilizam *level-sets*, no entanto, podem ser difíceis de se controlar, no sentido de determinar a condição de parada e os parâmetros de contorno e determinação das áreas almejadas. Por não terem sido desenvolvidos para segmentação das linhas centrais, os métodos implícitos podem precisar de mudanças ou de muitas etapas para conseguir bons resultados, podendo com isto prejudicar os resultados e aumentando o tempo de execução do método original. Um exemplo disto é um mesmo filtro sendo executado várias vezes em diferentes rotações [Bansal et al. 2010].

Os explícitos têm como objetivo principal a busca das linhas centrais. Em [Frangi et al. 1999] é feita uma classificação das estruturas usando os autovalores da matriz hessiana, e a linha central é gerada a partir de dois pontos indicados pelo usuário. A seleção dos dois pontos pode se tornar difícil para o usuário, dependendo do vaso que se objetiva, o que possivelmente prejudicará o resultado. Uma melhoria do trabalho anterior é feita em [Bitter et al. 2001] em que os pontos iniciais não são mais necessários, no entanto são realizados muitos passos para conseguir as linhas centrais, o que causa um maior tempo para alçancar as linhas centrais. Filtros são executados em [Gerig et al. 1993] para procurar as linhas centrais. Em [Mendonca & Campilho 2006] é utilizado um filtro de deslocamento de diferença gaussiana (*difference of offset Gaussians filters*), onde é feita uma classificação de cada ponto de acordo com o resultado do filtro, para uma pós validação, o que demanda mais tempo para apresentação dos resultados.

O trabalho de [Aylward & Bullitt 2002] obtém as linhas centrais usando *Height Ridges* e é a principal referência deste trabalho. Nesse método, o usuário define um ponto inicial para a segmentação, que deve estar o mais próximo do centro do vaso, então para esse ponto é verificado se ele se encontra no centro do vaso, o que ocorre se o ponto for um *Height Ridge* e obedecer a uma relação. Em caso afirmativo é realizado um percorri-

mento em duas direções opostas para a seleção dos pontos que formarão a linha central. Em caso negativo o autovalor mais negativo e seu respectivo autovetor da matriz hessiana do ponto, são usados para encontrar um próximo ponto como candidato a linha central. O método usa multi-escala para encontrar linhas centrais dos vasos de diâmetros diferentes e diminuir o tempo de processamento. A seleção de um ponto inicial pode ser uma complicação para o usuário. O percorrimento necessita de algumas verificações para se evitar que a linha central seja descontinuada ou para seguir em bifurcações.

Com o método proposto, a seleção da referência inicial torna-se uma tarefa mais simples, pois o usuário seleciona uma região do volume e não apenas um ponto na imagem. Caso o objetivo seja a linha central de um vaso muito fino, a seleção de um ponto próximo a linha central é complicada, mas a seleção da região ao redor do vaso desejado é mais simples. O processamento paralelo do volume selecionado diminui muito o tempo de segmentação. Além disso, executando-se o algoritmo sobre todo o volume selecionado, elimina-se a necessidade do percorrimento e das verificações para evitar a descontinuidade da linha e a continuação em bifurcações, o que reduz o tempo para visualização das linhas centrais.

Capítulo 2

Fundamentação Teórica

Neste capítulo são discutidos conceitos básicos de imagens médicas, o padrão DICOM e a aquisição de imagens médicas com a tomografia computadorizada que é o aparelho que gera imagens de angiogramas. Além disso, são discutidas informações de alguns conceitos matemáticos e a arquitetura CUDA de programação paralela, usada para executar boa parte do método.

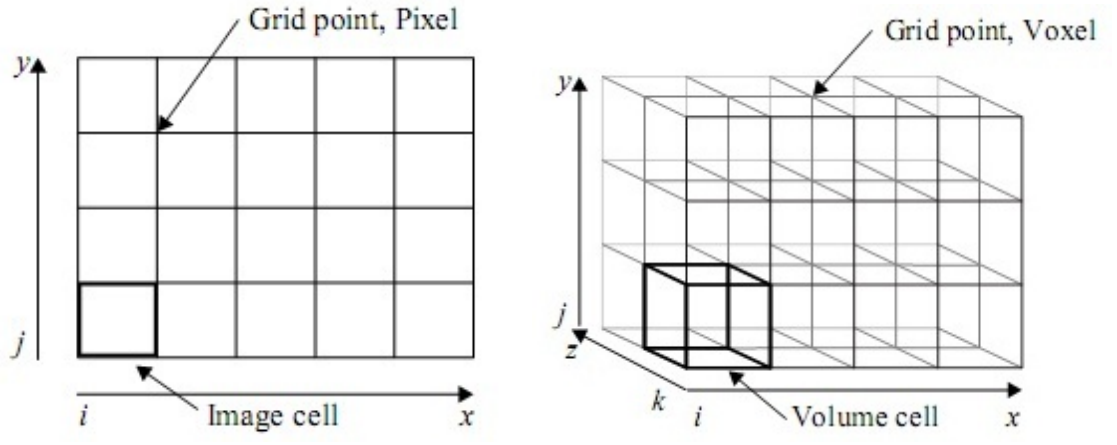
2.1 Conceitos Básicos sobre imagens médicas

Os dados fornecidos pelos aparelhos de aquisição de imagem formam um conjunto de imagens sobrepostas, umas sobre as outras, separadas por uma pequena distância em milímetros. Cada imagem contém informações correspondentes a uma fina fatia do corpo. A imagem é composta por pixels (*picture elements*) dispostos como uma grade (*grid*) regular nas coordenadas x e y. Um pixel é o menor componente de uma imagem digital, em que comumente é atribuída uma cor. Se consideramos i valores para coordenada x e j para a y, podemos acessar diretamente cada pixel da imagem, conforme exemplificado na Figura (2.1a).

Pela organização em pilha das fatias (*slices*), é possível formar uma relação tridimensional. Esta nova organização tridimensional possui um componente unitário correspondente ao pixel, denominado de voxel (volumetric pixel), que possui uma coordenada Cartesiana a mais, z, conforme ilustrado na Figura 2.1b. Voxels são paralelepípedos formados por planos paralelos aos eixos Cartesianos que definem o espaço do volume, sendo a menor unidade que compões os objetos 3D, similar a uma célula (volume *cell*).

Assim, a localização de cada voxel é feita com os valores (i,j,k), sendo k o valor para a dimensão z. A distância entre uma fatia e outra é chamada de distância de fatia (*slice distance*). A dimensão do voxel relativa aos três eixos cartesianos é o espaçamento de voxel (*voxel spacing*). Quando este espaço é igual para os três eixos cartesianos, é denominado uma grade isotrópica, caso contrário, anisotrópica. Na maioria dos casos o conjunto de imagens é anisotrópico, porque frequentemente a distância entre fatias (eixo z) é maior que a distância entre pixels [Preim & Bartz 2007].

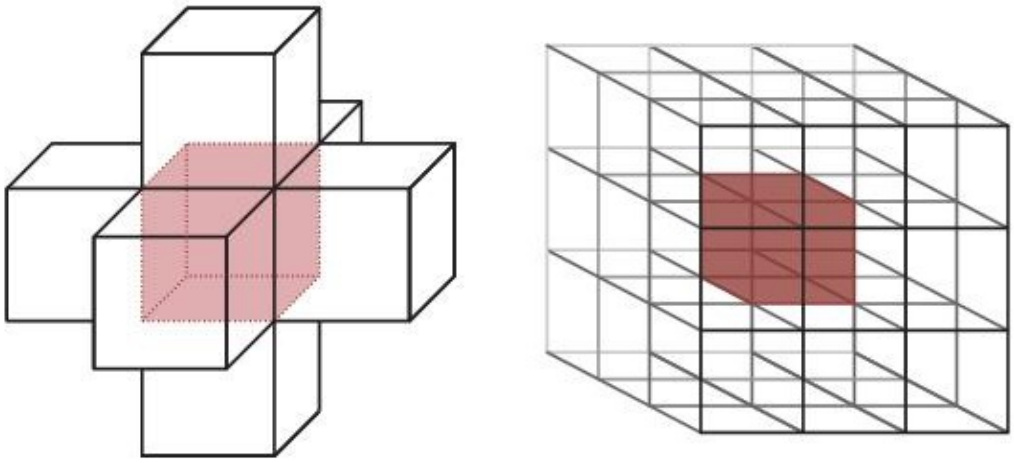
Como os voxels estão agrupados próximos um dos outros, um conceito para referenciar um grupo de voxels é o de vizinhança (*neighborhood*), que consiste nos voxels próximos ao que está em foco. Quando se usa o termo 6-*neighborhood* (Figura 2.2a)



(a) Grade de pixels (b) Grade de voxels

Figura 2.1: (a) Grade de pixels. (b) Grade de voxels [Preim & Bartz 2007].

estamos nos referindo aos voxels que possuem uma face compartilhada com o voxel principal. O grupo de voxels que possuem algum vértice, aresta ou face compartilhada com o voxel focado é chamado de 26-*neighborhood* (Figura 2.2b). E 8-*neighborhood* são os voxels vizinhos ao voxel em destaque e que estão em uma mesma fatia.



(a) 6-*neighborhood* (b) 26-*neighborhood*

Figura 2.2: Voxel principal em vermelho (mais escuro). (a) 6-*neighborhood*. (b) 26-*neighborhood* [Preim & Bartz 2007].

Existem imagens médicas coloridas, mas a maioria são em tons de cinza. Cada pixel da imagem é representado por um certo número de bits, alguns deles são relacionados a posição e outros ao nível de cinza. Com 3 bits têm-se 8 tons de cinza, 4 bits 16 tons de cinza, o Tomografo Computadorizado é capaz de criar 4096 valores distintos de cinza,

enquanto que o RM produz até 65.536 [Preim & Bartz 2007]. Dessa maneira consegue-se representar a intensidade de cada pixel utilizando menos espaço do que uma imagem colorida, onde normalmente a cor é representada pelas três cores primárias, vermelho, verde e azul, o que ocuparia três vezes mais espaço.

Um outro motivo é o fato do sistema humano ser mais perceptivo ao contraste com a cor cinza, conseguindo distinguir aproximadamente 100 valores diferentes de cinza [Rheingans 1992]. No entanto, a maioria dos monitores é capaz de mostrar essas imagens de 8 bits (256 tons). O contraste é uma característica importante para a qualidade da imagem, pois quanto melhor o contraste, mais fácil será distinguir as estruturas da imagem, detectar a fronteira entre órgãos, encontrar vasos ou aneurismas.

2.2 Padrão DICOM

As imagens médicas são criadas sob o padrão DICOM (*Digital Imaging and Communications in Medicine*) que surgiu a partir do padrão ARC-NEMA criado pelo *American College Radiology* (ACR) e da *National Electrical Manufacturer's Association* (NEMA) em 1985 [Committee 1985]. A segunda versão deste padrão foi lançada em 1988 [Committee 1988] e a terceira versão que corresponde ao DICOM, foi lançada em 1993 [Committee 1993]. Os principais objetivos do DICOM são [von Land et al. 1997]:

- Aumentar a capacidade de suas funcionalidade para estimular a sua implementação;
- Aumentar a ajuda aos usuários e desenvolvedores que irão utilizar o padrão;
- Incorporar características do SPI (*Standard Product Interconnect*) que é um dialeto do ARC-NEMA;
- Manter compatibilidade com versões anteriores quando possível, e;
- Adicionar descrições orientadas a objetos.

A biblioteca VTK (Visualization Toolkit) [Kitware 2006] foi usada para exibir as imagens dos arquivos DICOM, além dos resultados da segmentação das linhas centrais.

Cada arquivo DICOM possui um identificador único, *Unique IDentifier* (UID). O padrão cobre dois aspectos, a comunicação e a estrutura dos dados. O primeiro utiliza o padrão PACS (*Picture Archiving and Communication Systems*) que possibilita a troca de dados sobre o protocolo TCP/IP. O PACS cria uma rede de nós que adquirem, arquivam, visualizam e recuperam informações dos arquivos médicos.

O segundo aspecto é divido em duas partes, cabeçalhos (*headers*) e dados. As informações dos cabeçalhos especificam em que consistem os dados. Estes são organizados em etiquetas (*tags*) chamadas de IODs (*Information Object Definitions*), que por sua vez são reunidas em grupos de acordo com os significados. Como exemplo destes grupos, existe o grupo oito que trata do exame e da modalidade (Ultra Sonografia, Tomografia Computadorizada, etc), enquanto o grupo dez guarda as informações do paciente, como nome, sexo, idade e data de nascimento. O grupo dezoito tem informações sobre a área

de interesse objetivada no exame [Schaefer et al. 2006]. Existem muitas outras etiquetas, com outras informações relacionadas ao exame e as imagens. As etiquetas pertencentes ao padrão DICOM são chamadas públicas. Os fabricantes dos aparelhos de exames médicos podem adicionar *tags* que são denominadas privadas.

2.3 Aquisição de imagens médicas

Imagens médicas são adquiridas por vários motivos: diagnóstico, planejamento terapêutico, navegação intra-operativa e acompanhamento pós-operatório, por exemplo. O principal uso é para diagnóstico como parte da busca para encontrar a causa de um problema. O diagnóstico por imagem é muito útil quando os sintomas relatados pelo paciente ou observados pelo médico não são suficientes para detectar o problema. De acordo com a parte do corpo que se objetiva e a informação que se queira, escolhe-se um método de aquisição adequado. Um dos métodos de aquisição é a Tomografia Computadorizada (TC).

A Tomografia Computadorizada foi criada por Godfrey Hounsfield em 1968 [Susskind 1980] [Hounsfield 1975] e é um grande marco para a aquisição de imagens médicas, pois foi a primeira tecnologia capaz de visualizar tridimensionalmente aspectos internos do paciente. As imagens do corpo humano são produzidas a partir de informações da medida de atenuação dos raios-X que atravessam o paciente. O tomógrafo consiste basicamente de um emissor que envia um feixe de raio-X que atravessa o paciente e um detector localizado a frente do emissor. Estes rotacionam ao redor do corpo para obter dados de vários ângulos diferentes e translada para conseguir a próxima fatia. Um modelo de TC é visto na Figura 2.3, enquanto que um diagrama do sistema do TC é vista na Figura 2.4.

Figura 2.3: Modelo de TC [Cubero 2007].

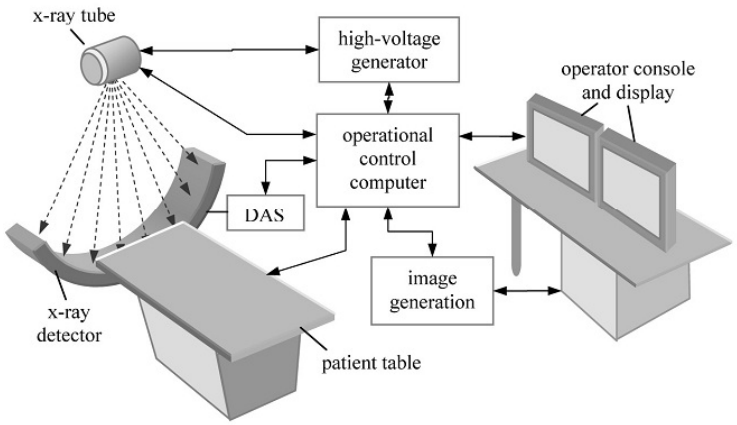
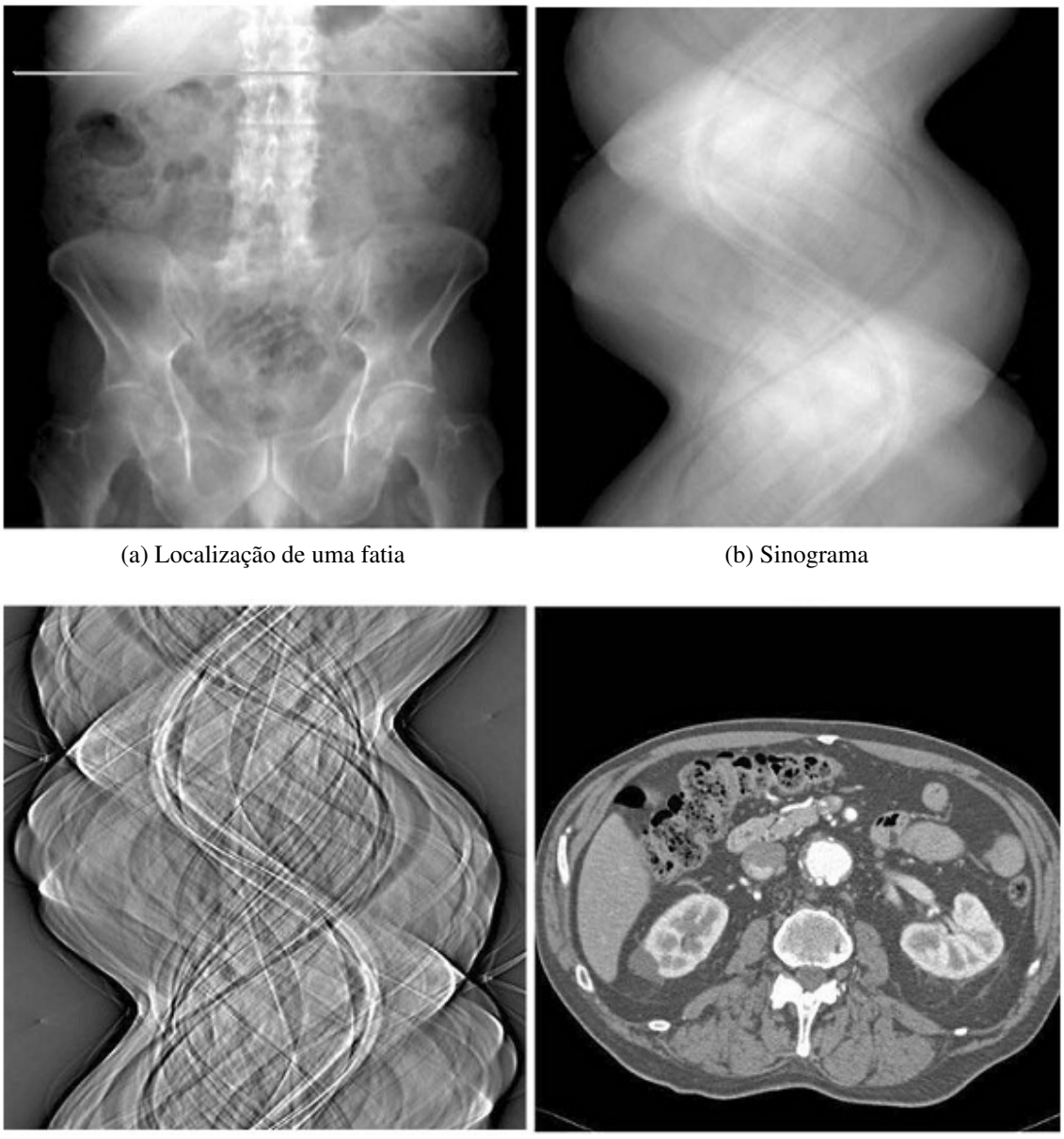


Figura 2.4: Modelo de TC [Hsieh 2009]. O DAS é a sigla para Sistema de Aquisição de Dados (Data Acquisition System).

A localização de uma fatia em relação ao corpo do paciente pode ser vista na Figura 2.5a, como uma linha horizontal. Os dados são capturados por um certo número de posições do emissor e do detector, chamadas de visualizações (*views*). Cada detector obtém uma leitura de cada visualização, o conjunto de dados conseguido por todos os detectores de todas as visualizações é o sinograma (*sinogram*) observado na Figura 2.5b. A intensidade do sinograma é proporcional ao valor do coeficiente de atenuação do raio-X entre o correspondente emissor e a posição dos detectores. Os diferentes tecidos do corpo humano possuem valores variados de coeficiente de atenuação de Raio X. Por este motivo, é possível distinguir tecidos distintos, separá-los de tumores ou mesmo conseguir determinar suas fronteiras. Dessa maneira o tomógrafo consegue fatias de *cross section* do corpo do paciente, como na Figura 2.5c e produz uma fatia como na Figura 2.5d.

A maneira como os dados de uma combinação emissor e detector são obtidos no TC são mostrados na Figura 2.6. Para essa combinação duas medidas físicas são levadas em conta: a medida de calibração (*calibration measurement*) e a medida real (*actual measurement*). Durante a calibração, o objeto cuja a *cross section* queremos não está no caminho de todo o feixe de raios, entre emissor e detector. A parte do feixe de raios que intersecta o objeto focado é chamado de região de reconstrução (*reconstruction region*). O ar e a água, que são atravessados pelo feixe, são chamados de material de referência e preenche a região de reconstrução inicialmente.

A medida de calibração serve para conhecermos o número de fótons que deixam o emissor e chegam ao detector. O detector de referência (*reference detector*) é usado para compensar a flutuação na força do emissor. Durante a medida real o objeto de interesse é inserido na região de reconstrução, substituindo o material de referência. Isto é importante para evitar que o objeto de interesse fique fora da região de reconstrução e permite que outros objetos fora da região de construção ocupem uma posição fixa durante a medição de calibração e real. Um exemplo disso é o objeto marcado como compensador (*compensator*). O tamanho da medida real em relação ao tamanho da medida de calibração depende da absorção de fótons e das propriedades de espalhamento do objeto a ser



(a) Localização de uma fatia

(b) Sinograma

(c) Sinograma convoluído

(d) Fatia

Figura 2.5: (a) Imagem renderizada mostrando a localização da fatia em um conjunto de imagens. Ela também corresponde as imagens a seguir. (b) Sinograma da fatia. (c) Sinograma convoluído. (d) A fatia gerada pelo TC. [Herman 2010]

reconstruído [Herman 2010].

As medidas de calibração e real são aplicadas para cada combinação emissor e detector e são usadas para gerar um terceiro conjunto de valores, os números de TC (CT *numbers*). Estes são transformados em valores de tons de cinza para a construção das imagens de TC. De um modo geral, um número de TC é proporcional à média relativa da atenuação linear de um voxel. Os dados são enviados para um computador que monta as fatias.

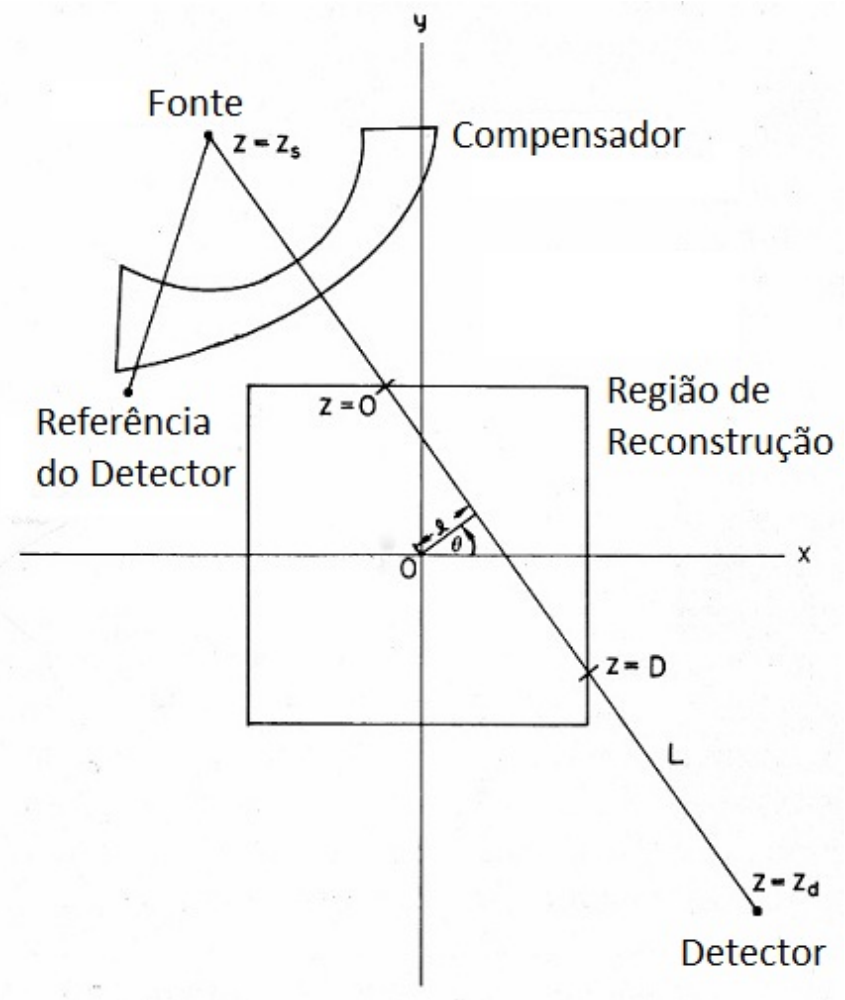


Figura 2.6: Obtenção de dados pelo TC (Adapatado de [Herman 2010]).

O gantry é parte do tomógrafo onde o paciente é posicionado, atualmente alguns têm a forma de "C", onde um dos objetivos é evitar o incomodo para o paciente com problemas de claustrofobia.

O tomógrafo EMI foi o primeiro aparelho introduzido no uso clínico em 1972, utilizava um feixe em pincel e cristais detectores de iodeto de sódio, movendo-se transversalmente em relação ao paciente. O conjunto emissor e detector obtinha uma projeção e então era rotacionado em 1 grau, assim 180 projeções eram obtidas num intervalo de 180

graus. Uma imagem demorava cerca de cinco minutos para ser gerada. Dessa forma o exame poderia durar horas devido a quantidade de feixes necessários para se conseguir as imagens com uma boa qualidade. Os TCs com essa arquitetura são chamados de sistemas de primeira geração.

Nos modelos da segunda geração foram adicionados mais detectores, que escaneavam paralelamente e que rotacionavam menos vezes, assim o tempo dos exames diminuiu e uma fatia era criada entre 10 e 60 segundos. Na geração seguinte era usado um feixe em leque rotativo e detectores rotativos, chamado sistema rotatório-rotatório. A quarta geração possui um anel rotatório com o emissor e um conjunto de detectores fixos, chamado sistema rotatório-fixo.

Os tomógrafos da terceira e quarta geração adquirem uma imagem em cerca de 1 ou 2 segundos e com qualidade aproximadamente semelhante. Um fator limitante é os cabos que fornecem a voltagem para ao tubo de raios-X, que diminuem a capacidade de rotação do emissor. Essa limitação foi superada nos TCs que passaram a usar a tecnologia do anel de deslizamento, em que a energia para o emissor é fornecida com um anel em contato com o gantry. Com isso o tubo de raios-X poderia rotacionar livremente, sem a limitação de 360^{o} como anteriormente, conseguindo várias projeções em um curto intervalo de tempo.

Os TC com o anel de deslizamento realizam a tomografia computadorizada chamada helicoidal ou espiral onde o emissor rotaciona constantemente enquanto o paciente é movimentado perpendicularmente no centro do aparelho, conforme ilustrado na Figura 2.7. Desta forma o volume completo é obtido com ausência de falhas espaciais ou temporais. A combinação do movimento em espiral com vários emissores permite a criação de até 64 fatias nos aparelhos mais modernos.

Na quinta geração é usado um canhão de elétrons que gera um feixe de alta velocidade em um arco de 210 graus. Existem anéis de detectores múltiplos que possibilita a obteção de múltiplas imagens. Não existem partes móveis e uma imagem é obtida entre 50 e 100 ms, diminui problemas de artefatos de movimento. Isso e muito útil para imagens cardíacas e para pacientes que não podem suspender a respiração por um intervalo de tempo, como crianças pequenas ou de trauma.

As vantagens do TC em relação ao raio-X são:

- **Localização de estruturas:** Pelo mapeamento de fatias que fornece um mapeamento tridimensional das estruturas analisadas, é mais simples a sua localização. Ao contrário do raio-X, onde a posicionamento é feito pela combinação de duas imagens uma lateral e outra frontal.

- **Sensibilidade:** O TC é mais sensível do que o raio-X, pois consegue distinguir mais claramente tecidos mole.

- **Medições quantitativas:** O aparelho de TC é capaz de medir a quantidade de radiação absorvida por cada região. Essa é uma maneira eficaz de distinguir cada região e é útil para detectar algumas doenças, que de acordo com o nível de absorção, pode determinar o seu estágio de desenvolvimento como a osteoporose [Preim & Bartz 2007].

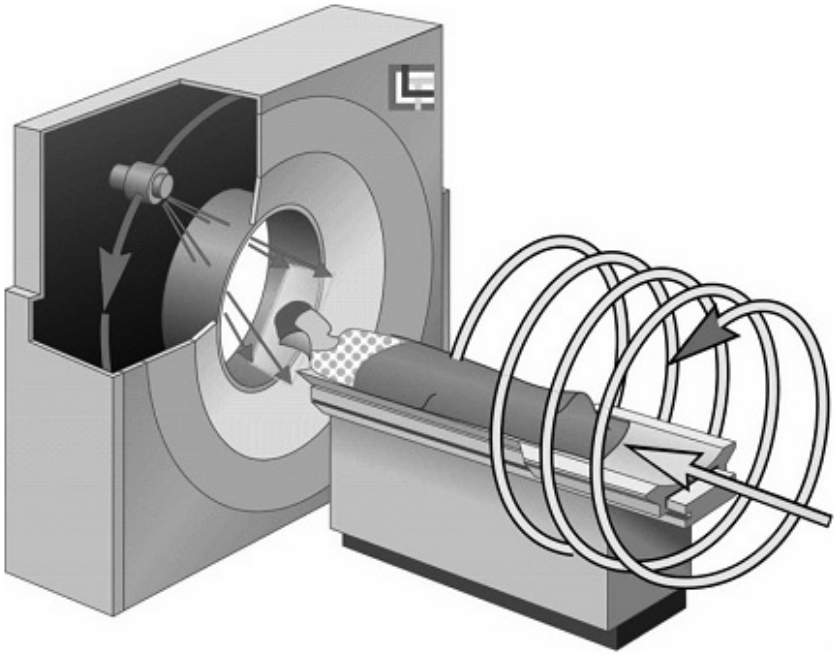


Figura 2.7: Movimento helicoidal feito pelo emissor nos TC modernos [Herman 2010].

Mas o TC tem alguns problemas [Paiva et al. 1999]:

- **Pequena resolução temporal:** As imagens adquiridas a partir de órgão em movimentos rápidos, como por exemplo as contrações cardíacas, podem ficar um pouco desfocadas.
- **Artefatos inerentes ao método de aquisição:** Existem alguns elementos que podem causar defeitos ou imprecisões nas imagens geradas como retroprojeção, truncamento, zebra e deslocamento químico.
- **Artefatos inerentes ao paciente:** Também o paciente pode conter elementos que afetam a imagem por exemplo, artefatos de alta densidade ou materiais metálicos.
- **Resolução espacial:** O TC tem resolução espacial relativamente pequena.
- **Inabilidade de detecção:** Algumas doenças em estado incipientes podem não ser detectadas pelo TC caso não tenha provocado alterações maiores no coeficiente do tecido objetivado.

Alguns parâmetros definem um exame de TC, como a distância entre voxels, o número de pixels no volume em cada direção x, y e z. Um exemplo do número de pixels é um volume dito como 512 x 512 x 250, ou seja, ele tem 250 fatias de dimensões 512 x 512. Outros parâmetros são a inclinação do gantry e a velocidade da mesa.

Os número de TC ou coeficientes de atenuação relativa (μ) é expresso em HUs (Hounsfield units). É uma maneira de definir quais regiões pertencem ao objeto visualizado. A

Tecido	Intervalo de valores (HU)
Ar	-1000
Pulmão	-900...-170
Gordura	-220...-30
Água	0
Pâncreas	10...40
Fígado	20...60
Coração	20...50
Rim	30...50
Ossos	45...3000
Sangue	40
Músculos	30...50

Tabela 2.1: Valores de densidade de tecidos em HU [Preim & Bartz 2007].

água tem valor zero, o ar -1000 e os pulmões variam entre -900 e 170, conforme descrito na Tabela 2.1.

A qualidade da imagem criada pelo TC pode usar como parâmetros o contraste, o ruído e a resolução espacial. O contraste é a diferença entre valores de HU de tecidos próximos. Com o aumento de kVp (*thousands of volts at peak voltage*) que é a voltagem do TC, aumenta o contraste, pois aumenta o espectro da energia dos fótons.

O ruído é principalmente definido pelo número de fótons utilizado para criação da imagem (ruído quântico), e diminui com o crescimento do número de fótons. O ruído pode ser diminuído com o aumento do tamanho do voxel ou com o aumento da dosagem do contraste. A resolução espacial é a capacidade de discriminar objetos adjacentes, e é influenciado pelo tamanho do pixel, quando menor o tamanho do pixel, melhor a resolução espacial. O nível de radiação usado no exame também é um fator que influência na qualidade do exame.

A intensificação de estruturas do corpo como coração, vasos e artérias é necessária para facilitar a sua observação, visto que muitas vezes elas são finas ou quase não aparecem nas imagens devido a cor muito semelhante a de tecidos próximos. Assim um contraste é injetado no paciente para realçar essas estruturas. O angiograma é o exame feito para visualizar essas estruturas.

2.4 Arquitetura CUDA

Um dos motivos para a melhoria na qualidade gráfica de jogos e efeitos de vídeo é o grande poder de processamento das GPUs (*Graphics Processing Units*), que conseguem executar um grande volume de processamento em paralelo e uma rápida leitura de dados armazenados na memória das placas gráficas de vídeo. A execução de filtros [Fernando 2004] e reconstrução de imagens médicas [Matt Pharr 2005], que são muito usados nos métodos de segmentação, são aceleradas com CUDA. A evolução do hard-

ware nos últimos anos em relação à CPU é vista na Figura 2.8. Nesta Figura observa-se o crescimento do volume de processamento das placas NVIDIA GeForce em relação a processadores Intel. Enquanto a GeForceGTX 480 possui teóricos 1375 GFLOPS/segundo aproximadamente, um processador Intel tem cerca de 125 GFLOPS/segundo.

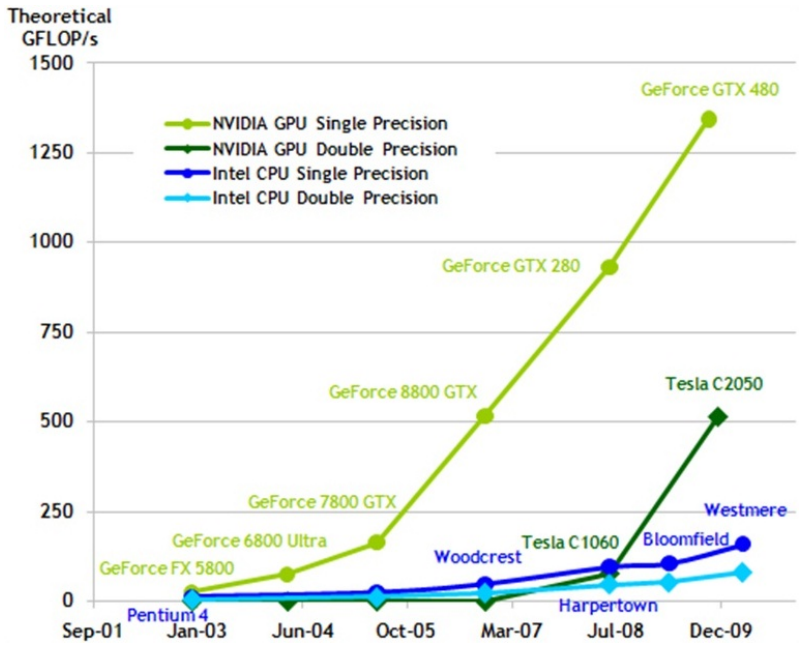


Figura 2.8: Histórico evolutivo do poder de processamento das GPUs e CPUs [NVIDIA 2010].

Assim, elas estão sendo utilizadas para outros fins além de gerar imagens, o que está sendo chamado de GPGPU (*General-purpose Computing on Graphics Processing Units*). São exemplos de aplicação de GPUGPU: simulação dinâmica de fluídos, análise sísmica, resolução de problemas matemáticos e segmentação de imagens.

Isso é possível graças a capacidade de programação de instruções para serem executadas diretamente em GPU. Existem algumas linguagens de alto nível para GPU, por exemplo GLSL (*OpenGL Shading Language*) [John Kessenich 2010], HLSL (*High Level Shading Languag*) [Reed 2010], Cg (*C for Graphics*) [Randima Fernando 2003] e CUDA (*Compute Unified Device Archtecture*) [NVIDIA 2010].

Existem duas questões importantes para criação de softwares que utilizam CPU e GPU afim de otimizar o desempenho. A primeira a definir são quais instruções serão executadas em cada uma. A segunda é, quais os dados e a forma como estes serão armazenados na memória da GPU para evitar constantes trocas de informações entre a memória RAM do computador e a memória da placa gráfica. Este fato é importante pois a forma como os dados são tratados em linguagens de programação em GPU é diferente da forma que a memória é manipulada em linguagens de programação para CPU. Outros problemas da programação em GPU é o conjunto limitado de instruções, onde a maioria consistem de funções matemáticas e a limitação de memória, que precisa ser gerenciada para evitar problemas de execução.

A linguagem CUDA foi criada pela NVIDIA para programação genérica em GPU sem necessidade de conhecimento de OpenGL, hardware gráfico ou do uso de textura para tráfego de informações. A primeira versão foi lançada em novembro de 2006 e as placas da série GeForce 8, Tesla e Quadro foram as primeiras a suportar a linguagem.

As camadas de software de CUDA possuem uma API (*Application Programming Language*), suporte ao *runtime* e ao *driver*. A API de CUDA (CUDA *libraries*) conta com suporte a diversas funções matemáticas, primitivas de computação gráfica, bibliotecas e um conjunto de marcadores e diretivas especiais para a linguagem C, que permitem o compilador de CUDA (nvcc) reconhecer se o código deve ser executado em GPU ou CPU. Tipos adicionais e subconjunto de funções da biblioteca padrão C que podem ser usados no *host* e no *device*, também são pertencentes a API. Sua sintaxe é semelhante a de C/C++ e possui versões para Fortran, Java (JaCUDA), Python (PyCUDA) e .Net (CUDA.NET).

A complexidade da GPU da NVIDIA é escondida por essa API. Desta maneira os programadores podem escrever códigos para a placa de vídeo sem a necessidade de conhecer funções gráficas pré-definidas. A primeira vantagem é que o programador não precisa se preocupar com detalhes de *hardware* da GPU. Outra vantagem, de acordo com a NVIDIA, é uma maior flexibilidade que permite que a arquitetura da GPU seja modificada sem tornar a API obsoleta e impossibilitar softwares já existentes de serem executados nas novas placas gráficas.

A camada de *runtime* de CUDA gerencia as operações da CPU (*host*), a comunicação da CPU com a GPU (*device*) e as funções específicas suportas pela GPU. O *driver* de CUDA otimiza e gerencia o uso dos recursos da GPU, como a memória da placa gráfica. Na Figura 2.9 é mostrada a pilha de camadas e sua relação com a CPU.

Em CUDA o modelo de programação usa *Stream Processor* que é composto por dois princípios básicos: *streams* e *kernels* [Lopes & de Azevedo 2008]. *Streams* são os fluxos de dados que servem de entrada para os *kernels*. Os *kernels*, que são os códigos executados em GPU, recebem esses dados, executam um processamento sobre eles e geram novos dados de saída. Esses, por sua vez, podem ser usados para alimentar outros *kernels* para um novo processamento.

Uma aplicativo que usa CUDA tem sua execução tanto em CPU (*host*) quanto em GPU (*device*). Apenas o código que contém a sinalização que deve ser executado em GPU é executado nela, as demais instruções do algoritmo é realizadas em CPU.

Em CUDA cada *kernel* é executado por uma *thread*. Estas por sua vez são agrupadas em blocos. Cada bloco pode conter no máximo 512 *threads* que podem ser organizadas de maneira matricial, como uma matriz unidimensional de tamanho 512, uma matriz bidimensional 22×22 ou tridimensional $8 \times 8 \times 8$. Os blocos são organizados em *grids*, que podem ser arrumados, no máximo, em uma matriz bidimensional cujo tamanho normalmente não deve exceder 65535×65535.

A definição da quantidade e da organização das *threads* é feita pelo programador, que deve fazer isso de maneira que otimize a execução em GPU. Esse dimensionamento é realizado atribuindo valores para variáveis do tipo `dim3`, que são como vetores de dimensão 3. Então elas são passadas em diretivas para a execução do *kernel*. Na Figura 2.10 tem-se a visualização de um dimensionamento das *threads*, que em CUDA é declarado da

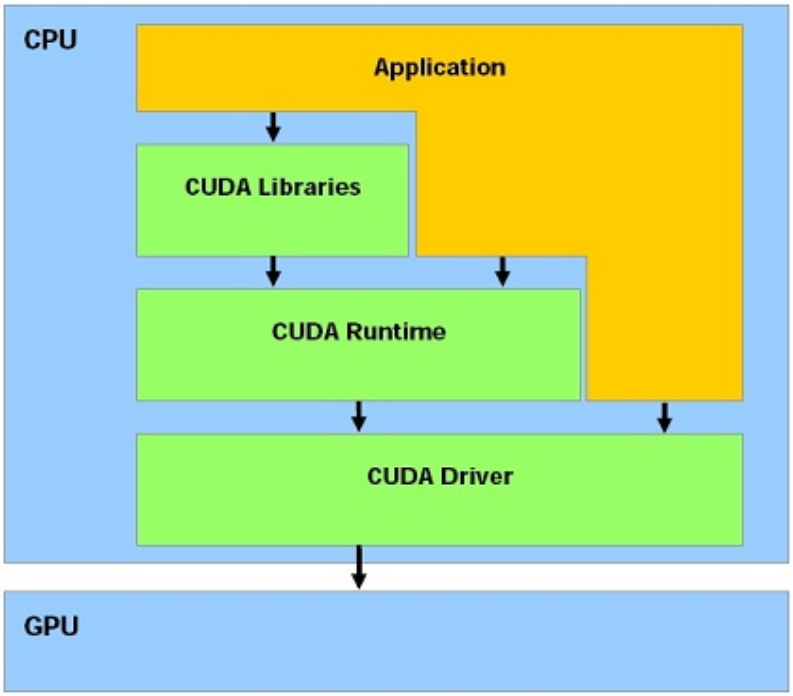


Figura 2.9: As camadas que constituem CUDA e sua relação com a CPU [Lopes & de Azevedo 2008].

seguinte forma:

$$\texttt{dim3 dimBlock}(4,3)$$

$$\texttt{dim3 dimGrid}(3,2)$$

A variável `dimBlock` é a dimensão do bloco e `dimGrid` a dimensão do *grid*. Os parâmetros são as dimensões nos eixos x e y respectivamente.

Estes dimensionamentos são dois dos quatro parâmetros que constituem a configuração de execução. Os outros dois parâmetros são o número de *bytes* na memória compartilhada alocados dinamicamente por bloco e um *stream* adicional.

Cada *thread* e bloco possui um índice único para sua identificação e localização, que pode ser obtido usando uma fórmula básica.

```
indice = dimensaoDoBloco * indiceDoBloco + indiceDaThread
```

A dimensão e o índice do bloco, além do índice de *threads* são conseguidos utilizando variáveis pré-definidas em CUDA. A dimensão do bloco em cada dimensão x, y e z, é obtida com `blockDim.x`, `blockDim.y` e `blockDim.z`, assim como a dimensão do *grid*, `gridDim.x` e `gridDim.y`, que podem ser úteis de acordo com a organização das *threads*. O índice de cada bloco é acessado com `blockIdx.x` e `blockIdx.y`, para as dimensões x e y respectivamente, enquanto que os de cada *thread* é obtido com `threadIdx.x`, `threadIdx.y` e `threadIdx.z`.

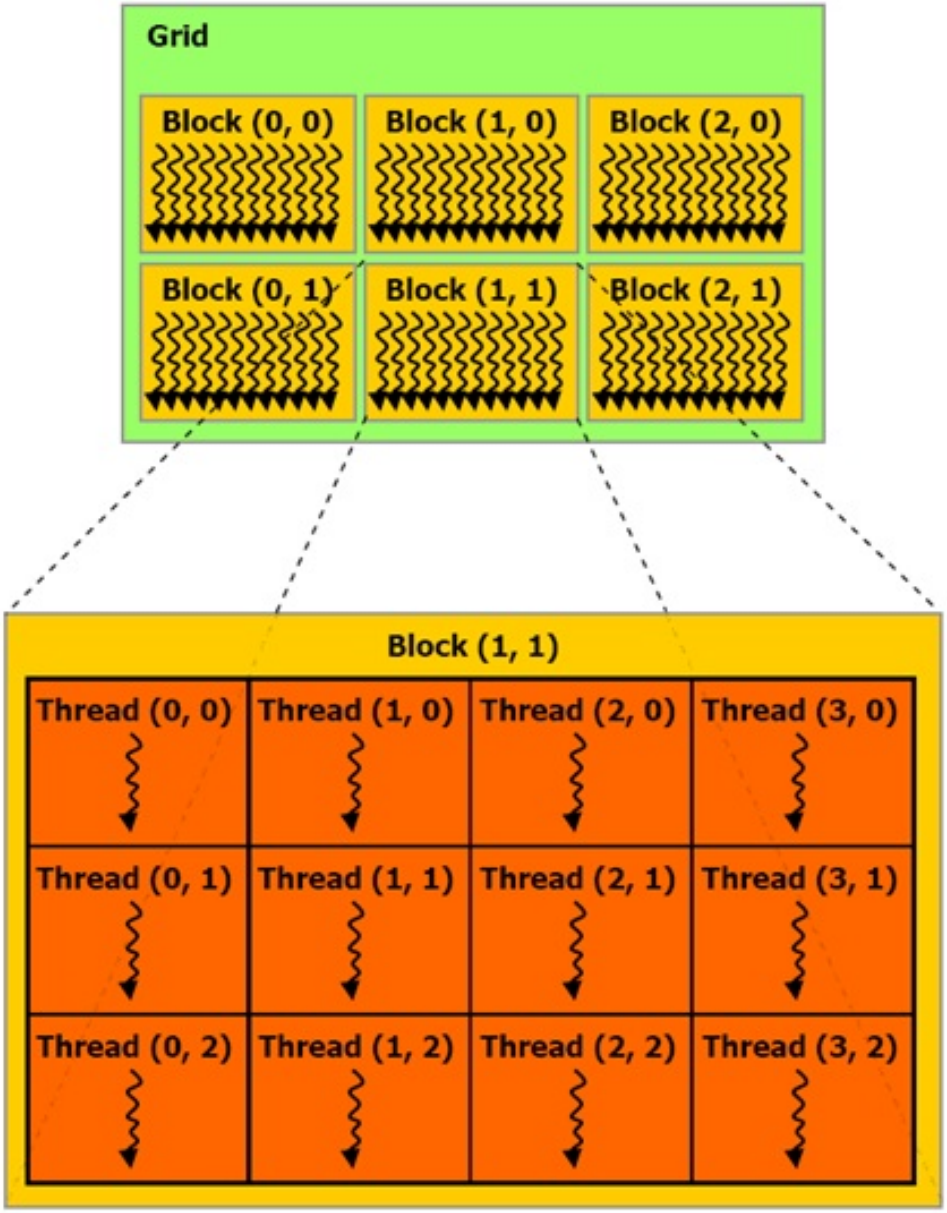


Figura 2.10: No exemplo o *grid* tem dimensão x igual a 2 e y igual a 3. O bloco tem o tamanho para x igual a 4 e para y igual a 3 [NVIDIA 2010].

Um trecho de código para que seja executado na GPU deve conter um marcador especial `_global_` caso a função seja invocado por uma função executada pelo *host* e usar o marcador `_device_` caso a função seja chamada por outra função realizada em GPU. As funções a serem executadas em CPU podem ter o marcador `_host_` ou não. Algumas limitações para funções com o marcador `_global_` são que elas devem retornar obrigatoriamente `void` e ter uma configuração de execução.

Existe uma hierarquia de memórias em CUDA (Figura 2.11). Cada *thread* possui uma memória local (*local memory*) com registradores. Cada bloco possui uma memória compartilhada (*shared*) que é acessada somente pelas *threads* do bloco. Existe também a memória global (*global memory*) da placa de vídeo que é acessada por todas as *threads*. Deve haver um cuidado com a quantidade de dados copiados para a placa gráfica para evitar falta de memória.

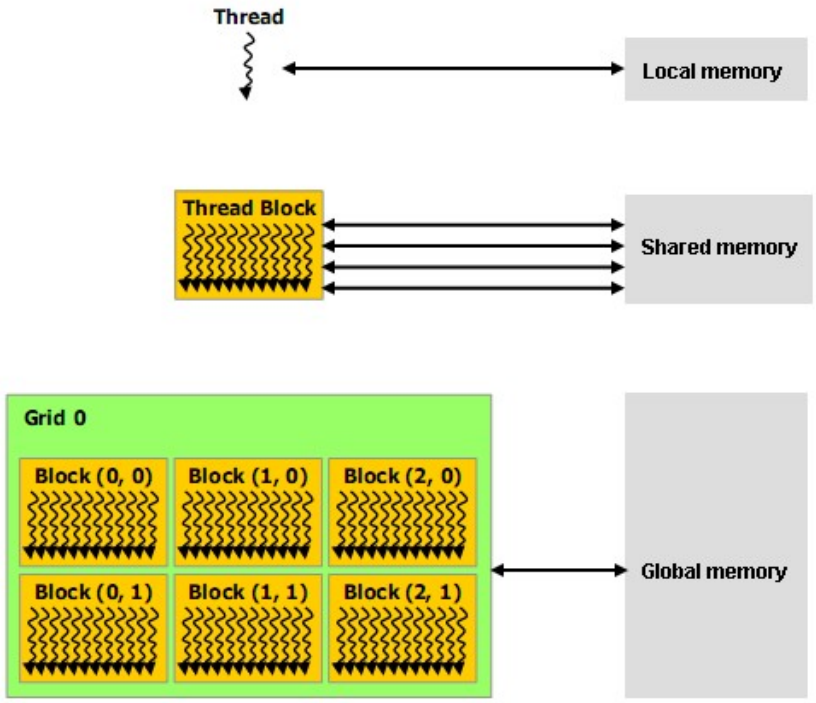


Figura 2.11: Arquitetura de CUDA, mostrando a organização das *threads*, blocos, *grids* e memórias [NVIDIA 2010].

Existem marcadores especiais para variáveis que são:

- `_device_` A variável deve residir no *device*.
- `_constant_` reside no espaço de memória de constantes que se encontram no *device*, durante toda a vida da aplicação.
- `_shared_` é alocada no espaço de memória de um bloco de *threads*, com o tempo de vida do bloco e somente acessíveis pelas *threads* do mesmo.

A arquitetura CUDA não permite recursão. Por isso, pode ser problemático gerar versões de algoritmos recursivos para serem executados em GPU. O tempo de execução costuma ser na casa de milissegundos, por isso CUDA possui funções próprias para medição do tempo com a precisão necessária. Estas funções foram usadas para a medição do tempo de execução do método, para uma medição mais precisa do tempo.

As informações contidas neste capítulo são importantes para compreender conceitos importantes de imagens médicas necessários para entendimento de partes do método, como a organização das fatias. A forma de obtenção das imagens médicas pelo TC ajuda a visualizar como as informações do corpo do paciente são transformadas nas imagens que formam o volume de dados usados no método. O conhecimento da arquitetura e de conceitos de CUDA são fundamentais para entender o código implementado do método.

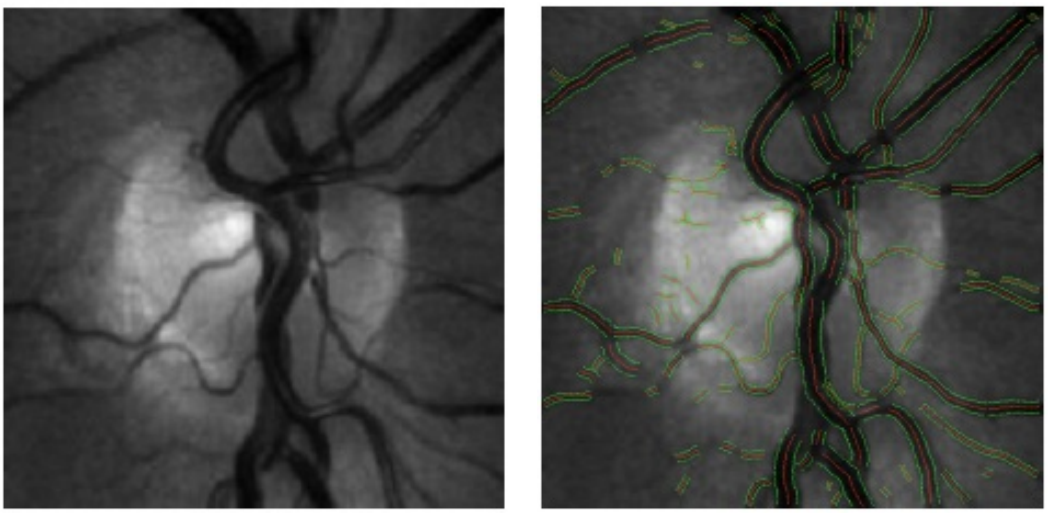
2.5 Segmentação Vascular

O processo de particionar uma imagem em regiões é chamado de segmentação [Gonzalez & Woods 2002]. Em alguns casos é necessário um pré-processamento da imagem, devido a ruídos ou como pré-requisitos para a aplicação de determinadas técnicas de segmentação, como por exemplo limiarização e esqueletonização. Um pré-processamento comum é o uso de filtros por convolução, principalmente para a redução de ruídos. São exemplos disso, os filtros Linear, Médio e de Wiener [Radha & Krishnaveni 2009]. Um outro exemplo de pré-processamento consiste em extrair o canal verde de imagens que usem o padrão RBG para a segmentação, pois o canal verde tem a melhor qualidade da imagem, em relação ao canal vermelho e azul [Vlachos & Dermatas 2010, Ricci & Perfetti 2007].

A segmentação é utilizada na área médica para muitos fins. São exemplos destes fins: o registro de imagens médicas [Shen et al. 2003] [Can et al. 2002], monitoramento em vídeo [Koozekanani et al. 2003] e detecção das paredes e linhas centrais dos vasos [Bansal et al. 2010]. Na Figura 2.12a temos uma região de uma imagem do olho humano, e na Figura 2.12b temos essa imagem segmentada, onde as linhas verdes indicam a parede dos vasos sanguíneos e as linhas vermelhas as linhas centrais. Com a separação das paredes dos vasos, é mais fácil a visualização dos mesmo em meio a outras estruturas e com as linhas centrais, é possível determinar a estrutura vascular de uma maneira mais simplificada.

Existem vários métodos de segmentação que definem o critério e a maneira como ela será feita, alguns objetivam apenas um tipo de estrutura, como a vascular. Frequentemente são utilizados filtros nos métodos de segmentação, como por exemplo para obtenção do gradiente, do laplaciano, detecção de bordas e realce da imagem [Pratt 2007]. O resultado dos filtros é usado como entrada para alguma parte do método. Alguns métodos necessitam de uma referência inicial, que pode ser um ou mais pontos (*seed points*) ou mesmo uma região da imagem. A referência inicial frequentemente deve obedecer algum critério, como estar dentro do vaso que se deseja segmentar.

A maneira mais simples de se realizar a segmentação é executá-la manualmente, selecionando os pontos na imagem que pertencem a parte desejada. Isto é chamado de segmentação manual. Esta segmentação é muito utilizada para separação de objetos que são difíceis de delimitar suas formas devido ao baixo contraste na imagem ou por a sua forma



(a) Imagem original (b) Imagem segmentada

Figura 2.12: (a) Imagem de uma região do olho humano. (b) Resultado da segmentação com as paredes dos vasos indicadas pela cor verde e o centro dos vasos pela cor vermelha [Bansal et al. 2010].

ser muito complexa [Preim & Bartz 2007]. O principal problema da segmentação manual é o grande tempo necessário para ser concretizada. A segmentação semi-automática procura resolver este problema realizando a segmentação de imagens com pouca interferência do usuário. Esta interferência em muitos métodos limita-se a determinação dos pontos de referência por parte do usuário.

Frequentemente, os métodos de segmentação são combinados para melhorar o resultado final. Um exemplo desta combinação consiste em utilizar a segmentação resultante de um método como entrada para o outro método. Um método pode realçar algumas características em uma imagem o que melhora o resultado em um método utilizado em seguida no mesmo conjunto de imagens. Como exemplo de características usadas para segmentação temos a intensidade de um ponto, textura e magnitude do gradiente [Dougherty 2009].

Os problemas mais comuns de segmentação de imagens médicas, que prejudicam o resultado são ruídos, baixo contraste, objetos com variação de níveis de intensidade e de formas complexas.

Alguns dos métodos de segmentação vascular são:

- **Limiarização (*Thresholding*):** É o método mais simples de segmentação, que divide a imagem em áreas que possuem em comum um valor ou estão em um intervalo de valores [Jiang & Mojon 2003]. A limiarização é rápida e fácil de ser implementada. No entanto, devido a sua simplicidade os resultados são muito prejudicados por existirem na imagem ruídos, baixo contraste e muitas áreas de intensidade com valores próximos, porque estes fatores acabam agregando ao resultado regiões que não pertencem ao objeto desejado. Assim a limiarização é muitas vezes usada como

um selecionador de pontos na imagem que serão processados por outros métodos posteriormente, como neste trabalho.

- **Esqueletonização (*Skeletonization*):** Consiste em criar as linhas centrais dos vasos, em seguida uni-las e montar o esqueleto da estrutura vascular, representando assim a forma da estrutura com poucos pixels. Normalmente é realizada uma limiarização antes de ser executada e baseiam-se nas fronteiras das formas as quais estão sendo aplicadas [Kudelski et al. 2010]. Alguns métodos baseiam-se em uma máscara, e usam informações dos vizinhos ao ponto processado [Kwon et al. 2001]. Estes métodos podem usar um ponto inicial definido pelo usuário. Dois problemas comuns ocorrem nesse tipo de método. O primeiro é a falha em curvas do esqueleto resultante, quando pontos deixam de ser segmentados. O segundo problema é o resultado não conter pontos próximos as extremidades dos vasos.

- **Região de crescimento (*Region growing*):** As características de um ponto inicial são referência para que outros pontos ao redor dele sejam agrupados formando uma região. Este agrupamento ocorre até que não seja mais encontrado nenhum ponto que se enquadre nas características do ponto inicial. Assim podem ser criadas várias regiões de características diferentes, caso forem usadas diferentes combinações de características. A necessidade de intervenção do usuário é uma desvantagem desse método, pois o usuário necessita definir um ou mais pontos iniciais (*seed points*) para cada região desejada para a segmentação [Martínez-Pérez et al. 1999]. Os pontos de início do algoritmo podem ser escolhidos randomicamente de maneira automática ou usando uma heurística para um problema específico, mas alguns pontos podem segmentar regiões que não estão entre as desejadas. O ruído e o baixo contraste também são problemas que prejudicam o resultado.

- **Casamento de filtros (*Matching filter*):** São usados filtros para buscar na imagem estruturas que se assemelham às contidas nele, quando encontradas são segmentadas. Isso envolve a convolução da imagem com múltiplos filtros em diferentes direções [Miles & Nuttall 1993] [Hoover et al. 2000]. O problema desses métodos é ter o filtro adequado para o objeto que se deseja, caso contrário os resultados podem não ser muito satisfatórios.

- **Contornos ativos (*Active contours*):** É conhecido também como *Snakes*. É criada uma curva paramétrica ao redor da estrutura que deseja-se segmentar, então forças externas à região selecionada buscam se adequar à estrutura a ser segmentada, enquanto forças internas suavizam a região selecionada [Chiu et al. 2010] [Derraz et al. 2004]. Essas curvas são constituídas por vários pontos, onde cada um possui uma energia associada que aumenta ou diminui de acordo com a força nele aplicada. O controle destas forças é um fator importante para o resultado, e é um problema porque quando controle não é bem definido, os contornos podem não segmentar as fronteiras do objeto alvo adequadamente. Os pontos iniciais selecionados pelo usuário também influenciam na facilidade do algoritmo segmentar corretamente, e dependendo do tipo de objeto almejado, pode ser difícil a seleção adequada pelo

usuário. A principal vantagem dos métodos de contornos ativos é que o resultado é bem coerente, fechado e suavizado em relação a fronteira do objeto focado [Dougherty 2009].

- **Multi-escalares (*Multiscale*):** Usam várias resoluções da imagem. Características de distinção mais fácil, como vasos de maiores dimensões, são processados em uma baixa resolução, enquanto que os de menor dimensão e que necessitam de mais detalhes, usam uma resolução maior [Li et al. 2006]. É muito boa para eficácia de detecção de estruturas que variam muito de tamanho, como vasos. Um problema é a necessidade de executar o algoritmo várias vezes em diferentes resoluções, o que pode demandar muito tempo para obtenção dos resultados.

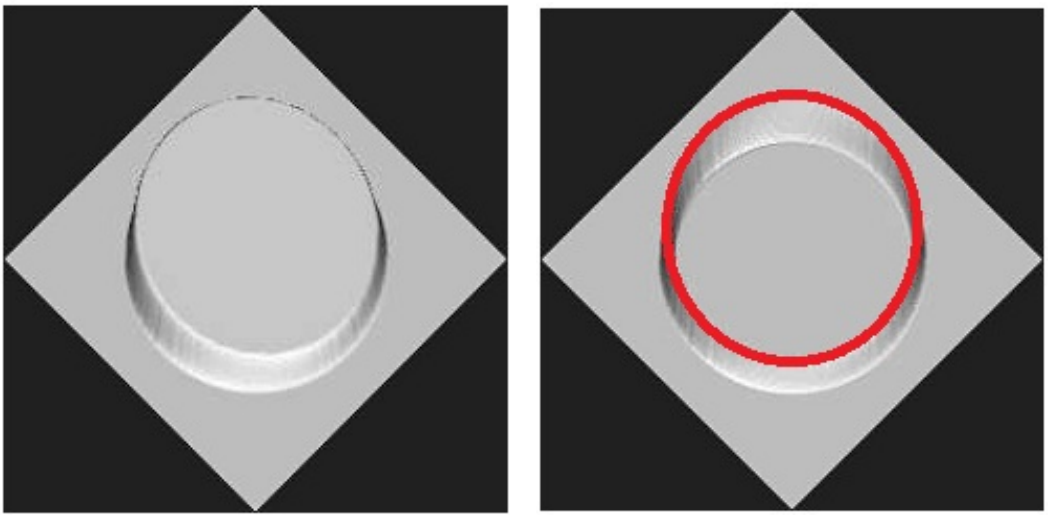
- ***Height ridges*:** É definido como o local onde a intensidade assume um valor máximo local. Utilizando este conceito, os métodos de *height ridges* procuram pontos de maior intensidade em relação aos seus vizinhos. A busca de linhas centrais é um exemplo de aplicação deste método. O *height ridges* será mais detalhado na próxima seção.

2.6 *Height Ridges*

Métodos para representar formas de objetos em imagens de tons de cinza podem ser divididos em duas categorias: baseadas em bordas e em regiões. Os baseados em bordas usam a informação do gradiente, onde grandes valores dele indicam a presença de borda. A propriedade de borda de um ponto é determinada pela medida de desigualdade entre a intensidade do ponto e a dos vizinhos. O valor de magnitude do gradiente é um exemplo de valor usado para a detecção. Esses métodos devem trabalhar com a orientação da borda, a sua intensidade e a conexão dela [Eberly et al. 1994] . O método de *ridges* se enquadra na detecção de bordas.

Um exemplo de detecção de borda está na Figura 2.13. A representação tridimensional dos níveis de cinza de uma imagem, onde a altura representa a intensidade do nível de cinza na imagem é mostrado na Figura 2.13a. Enquanto na Figura 2.13b é mostrado tridimensionalmente a intensidade da magnitude do gradiente, onde quanto mais alto o ponto, maior o valor de magnitude. Muitos métodos de detecção de bordas tem o seu resultado prejudicado pela presença de ruído, no entanto os métodos que utilizam *ridges* **são menos afetados por ruídos [Aylward et al. 1996].**

É desejável que *ridges* obedeçam algumas propriedades. Primeiro, o processo deve ser local, o *ridge* deve ser obtido por informações do ponto e sua vizinhança. Segundo, um *ridge* deve ser invariável com relação as seguintes transformações: translação, rotação e ampliação. Estas invariações garantem uma boa robustez para *ridges* para diferentes objetos desejados. As duas primeiras transformações, definem que as operações sobre o *ridge* devem ser comutativas, ou seja, o *ridge* de um objeto rotacionado e transladado também devem ser rotacionados e transladados da mesma forma e direção, mantendo-se da mesma maneira como no objeto original. A terceira invariação significa que o *ridge* deve ser independente da unidade de medida usada. Desta forma, mudanças na unidade



(a) Intensidade dos pontos (b) Magnitude do gradiente

Figura 2.13: (a) Gráfico do valor de intensidade dos pontos, (b) Gráfico de intensidade de magnitude, com o pico de intensidade marcado em vermelho. (Adaptado de [Eberly et al. 1994]).

de medida devem manter o *ridge* uniforme e independente do tamanho do objeto. Caso ele dobre de tamanho, o *ridge* deve dobrar também.

Os *ridges* de f são definidos, considerando f uma função $\in C^2(\Re^n, \Re)$. *Df(x)* e $D^2f(x)$ o gradiente e o laplaciano do ponto x. Além disso $\lambda_i, 1 \leq i \leq n$ sendo os autovalores da matriz Hessiana com $\lambda_1 \leq ... \leq \lambda_n$. Considerando $v_i, 1 \leq i \leq n$, os correspondentes autovetores. Assim, um ponto x é um *d-dimensional ridge* se x é um ponto de máximo local do tipo d com respeito a $V = [v_1, ..., v_{n-d}]$ e seus autovalores ordenados, se obedecer as condições: $V^t Df(x) = 0$ e $\lambda_{n-d} < 0$ [Eberly 1996].

Para um volume, n = 3, com um ponto x, os autovalores da matriz Hessiana do ponto x ordenados como $\lambda_1 \leq \lambda_2 \leq \lambda_3$, os autovetores v_1, v_2, v_3 e d = 1, o ponto x é um *Height Ridge* se obedecer as condições:

- $\lambda_1 < 0$ e $\lambda_2 < 0$
- $v_1 * Df(x) = 0$ e $v_2 * Df(x) = 0$

O escalar $\lambda \in C$ é um autovalor (*eigenvalue*) de uma matriz A n x n, cujos elementos são números complexos, se existir um vetor v diferente de zero tal como Av = λv. Nesse caso, v é o autovetor (*eigenvector*) correspondente a λ [Anton 2004].

Graficamente o principio básico pode ser visto na Figura 2.14. Seja a imagem (a) formada por um retângulo com dois vetores, essa imagem sofre uma transformação (T) apenas na horizontal, resultando em uma imagem de formato retangular (b). O vetor v2 passa a ser v2', que não tem a mesma direção que v2. Dessa forma v2' não é a representação de v2 multiplicado por um escalar. Olhando para v1', o vetor tem a mesma

direção que v1, e por isso é representado por v1 multiplicado por um escalar. Diz-se então que v1 é um autovetor de T e o escalar é o autovalor correspondente.

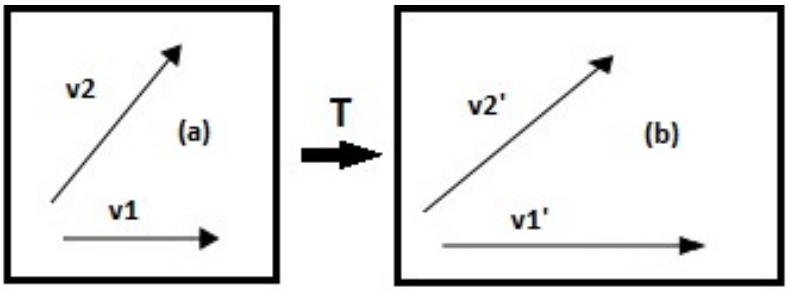


Figura 2.14: Visualização gráfica do conceito básico de autovalores e autovetores.

Os *ridges* são utilizados em aplicações para:

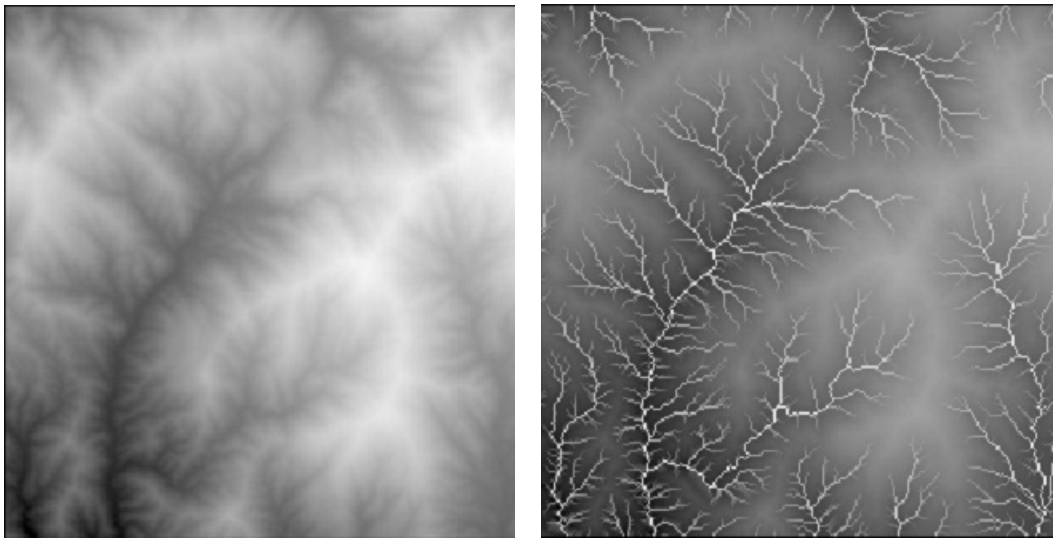
- **Extração de *centerlines*:** Extraem as centerlines de objetos de um volume anisotrópico em tons de cinza.
- **Aproximações de medianas:** Consiste em determinar o quão próximo esta um ponto da *centerline* de um objeto [Lopez et al. 1999].
- **Segmentação:** Os *ridges* e vales obtidos com os autovalores podem ser usados para separar uma imagem em regiões.
- **Extração de padrões de drenagem:** A delineação do caminho de rios na superfície da terra é um exemplo dessa aplicação, como visto na Figura 2.15.

Existem outras ações realizadas para a segmentação de imagens que são utilizadas nas etapas deste trabalho. Na primeira etapa o filtro gaussiano é processado sobre a imagem para reduzir o ruído. Em seguida, na segunda e terceira etapa são calculados o vetor gradiente e a matriz hessiana respectivamente, em cada ponto da imagem. Estes conceitos serão discutidos em seguida.

2.7 Conceitos Matemáticos

2.7.1 Filtro Gaussiano

Existem vários filtros passa baixa usados em processamento de imagens, como por exemplo filtros de média. Um dos mais utilizados é o *gaussian blur* ou gaussiano passa baixa, e tem como vantagens realizar uma boa suavização e ser circularmente simétrico, assim as bordas e linhas nas várias direções são tratadas igualmente. Frequentemente os operadores de suavização, que são usados em processamento de imagens atuam em uma vizinhança pequena, como 3x3, 5x5. Os valores do operador usualmente usam os números do triângulo de Pascal, como na Figura 2.16 [Waltza & Miller 1998].



(a) Imagem original (b) Padrao de drenagem

Figura 2.15: (a) Imagem de elevação digital [Lopez et al. 1999]. (b) Imagem com o padrão de drenagem obtida [Lopez et al. 1999].

Índice N	Coeficientes	Soma dos coeficientes = 2^N
0	1	1
1	1 1	2
2	1 2 1	4
3	1 3 3 1	8
4	1 4 6 4 1	16
5	1 5 10 10 5 1	32
6	1 6 15 20 15 6 1	64
7	1 7 21 35 35 21 7 1	128
8	1 8 28 56 70 56 35 8 1	256
9	1 9 36 84 126 126 84 36 9 1	512
10	1 10 45 120 210 252 210 120 45 10 1	1024
11	1 11 55 165 330 462 462 330 165 55 11 1	2048

Figura 2.16: Triângulo de Pascal que é usado como base para montagem dos operadores do gaussiano [Waltza & Miller 1998].

Cada linha do triângulo e usada para compor a máscara (*kernels*) do gaussiano. A linha de índice 2 (1 2 1) por exemplo, é usada para montar uma máscara 3x3 ou 3x3x3. Os *kernels* podem ser executados em cada direção separadamente e o resultado será o mesmo que executado conjuntamente.

Por exemplo a máscara:

$$\begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix}$$

Terá o mesmo resultado caso seja passada sobre a imagem na direção X e depois na Y, como abaixo:

$$\begin{vmatrix} 1 & 2 & 1 \end{vmatrix} \qquad \begin{vmatrix} 1 \\ 2 \\ 1 \end{vmatrix}$$

O filtro gaussiano de uma dimensão tem a forma:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}, \tag{2.1}$$

enquanto que o filtro de duas dimensões é:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{2.2}$$

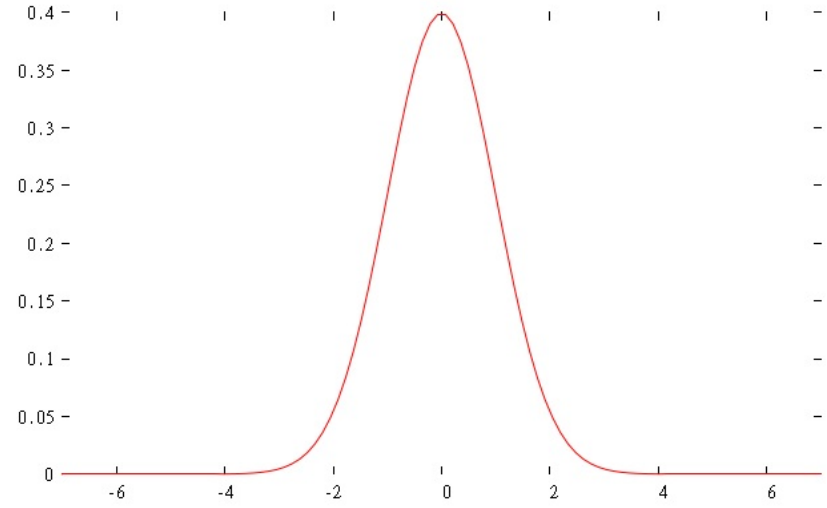
e o de três dimensões é definido como [Sage et al. 2005]:

$$G(x,y,z) = \frac{1}{(2\pi)^{3/2}\sigma^3} e^{-\frac{x^2+y^2+z^2}{2\sigma^2}} \tag{2.3}$$

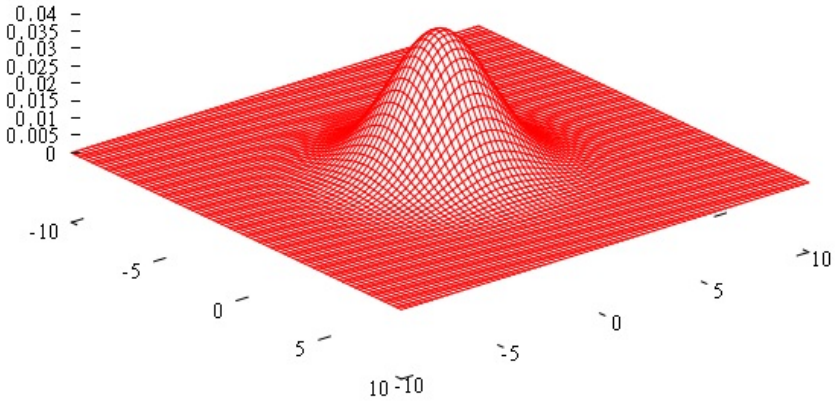
Graficamente, o filtro Gaussiano com uma e duas dimensões é visto na Figura 2.17. Considerando que uma imagem, é como uma coleção de pixels discretos, nós precisamos produzir uma maneira discreta do filtro gaussiano para obter a máscara, o que teoricamente seria infinito. No entanto, na prática ela é zerada três unidades do centro.

O desvio padrão (σ) de uma máscara pode ser calculado obtendo o valor da função onde os valores de X, Y são zero, ou seja, o centro da máscara. Em seguida substitui-se os valores na equação do filtro gaussiano. No *kernel* 3x3 mostrado acima, deve-se normalizalo multiplicando-o por 1/16. Assim o valor da função no centro da máscara é 4/16 = 0.25. Substituindo na equação de duas dimensões temos: $0.25 = 1/2\pi\sigma^2$, resultando em um valor para σ = 0.798. Máscaras com desvio padrão maiores podem ser montadas ou uma mesma máscara pode ser passada sobre a imagem várias vezes.

A suavização de uma imagem pelo filtro Gaussiano ocorre quase da mesma maneira que o filtro de média, mas o resultado será mais suave a medida que o valor do desvio padrão aumente. Quanto maior o tamanho da máscara melhor tende a ser o resultado. O gaussiano, por não ter todos os pesos iguais como o filtro de média, realiza uma suavização mais delicada preservando melhor os contornos.



(a) Uma dimensão



(b) Duas dimensões

Figura 2.17: (a) Forma gráfica do Filtro Gaussiano para uma dimensão, (b) para duas dimensões. [Williams et al. 2011]

2.7.2 Gradiente no Contexto de Processamento de Imagens

A primeira derivada de uma função é chamada de gradiente. No processamento de imagens digitais é usada a magnitude do gradiente. Em um ponto qualquer de uma imagem, o gradiente é definido como um vetor:

$$\nabla f = \begin{vmatrix} G_x \\ G_y \end{vmatrix} \tag{2.4}$$

Onde G_x é a primeira derivada na direção X e G_y na direção Y. Cada um dos componentes do vetor é também um operador e são definidos na forma de derivadas parciais como:

$$G_x = f(x+1,y) - f(x,y) \tag{2.5}$$

$$G_y = f(x,y+1) - f(x,y) \tag{2.6}$$

A magnitude do vetor gradiente é dada por:

$$\nabla f = [G_x^2 + G_y^2]^{1/2} \tag{2.7}$$

A magnitude do vetor gradiente é invariável à rotação, ou seja, é isotrópico. Na prática a magnitude do gradiente é obtida usando os valores absolutos dos componentes.

$$\nabla f \approx |G_x| + |G_y| \tag{2.8}$$

É uma maneira mais simples de implementação computacional e preserva as mudanças em tons de cinza, mas a propriedade isotrópica é perdida em sua maioria. Muitas das máscaras usadas para obtenção do gradiente são isotrópicas somente para ângulos múltiplos de 90^o, ou seja, nas direções X e Y.

O gradiente é usado no processamento de imagens digitais para encontrar descontinuidades na imagem, que podem ser um ponto, uma linha ou uma borda. Para isso são analisados os valores em cada ponto da imagem, e os pontos de descontinuidade possuem valores diferentes de zero.

Um dos operadores mais usados para o processamento do gradiente é o de Sobel, que usa uma máscara em cada direção de tamanho 3x3. As máscaras na direção x e y são:

$$G_x = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix} \qquad G_y = \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$$

Computacionalmente consideramos um conjunto de pontos como:

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Aplicamos a máscara de Sobel com o objetivo de calcular o gradiente do ponto z_5

usando a vizinhança e as fórmulas [Gonzalez & Woods 2002]:

$$G_x \approx |(z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)| \tag{2.9}$$

$$G_y \approx |(z_1 + 2z_4 + z_7) - (z_3 + 2z_6 + z_9)| \tag{2.10}$$

Algumas propriedades são obtidas para cada ponto da imagem usando o gradiente:

- A direção do gradiente de um ponto indica a direção de maior crescimento da função.
- A magnitude do gradiente indica o valor de quanto é o aumento.
- A direção do gradiente é a normal do ponto.
- O gradiente define o plano tangente ao ponto.

2.7.3 Laplaciano no Contexto de Processamento de Imagens

O Laplaciano é a soma de todas as derivadas parciais de segunda ordem. Para uma função f de três dimensões, é definido como o Laplaciano:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \tag{2.11}$$

Devido as derivadas em qualquer ordem serem operadores lineares, o Laplaciano é um operador linear. Realizando a segunda derivada completa para todas as variáveis da função, é gerada uma matriz, chamada de matriz Hessiana da forma [Sato et al. 2000]:

$$\nabla^2 f = \begin{vmatrix} f_{xx} & f_{xy} & f_{xz} \\ f_{yx} & f_{yy} & f_{yz} \\ f_{zx} & f_{zy} & f_{zz} \end{vmatrix} \tag{2.12}$$

O valor de cada componente da matriz pode ser obtido por derivadas parciais com:

$$f_{xx} = f(x+1,y,z) - 2f(x,y,z) + f(x-1,y,z) \tag{2.13}$$

$$f_{yy} = f(x,y+1,z) - 2f(x,y,z) + f(x,y-1,z) \tag{2.14}$$

$$f_{zz} = f(x,y,z+1) - 2f(x,y,z) + f(x,y,z-1) \tag{2.15}$$

$$f_{xy} = f_{yx} = \frac{f(x-1,y+1,z) - f(x+1,y+1,z) + f(x+1,y-1,z) - f(x-1,y-1,z)}{4} \tag{2.16}$$

$$f_{xz} = f_{zx} = \frac{f(x-1,y,z+1) - f(x+1,y,z+1) + f(x+1,y,z-1) - f(x-1,y,z-1)}{4} \tag{2.17}$$

$$f_{zy} = f_{yz} = \frac{f(x,y+1,z-1) - f(x,y+1,z+1) + f(x,y-1,z+1) - f(x,y-1,z-1)}{4} \tag{2.18}$$

Uma máscara usada para a obtenção do Laplaciano é:

0	1	0
1	-4	1
0	1	0

Uma outra que inclui as diagonais é:

1	1	1
1	-8	1
1	1	1

Essas máscaras são isotrópicas para rotações com ângulos múltiplos de 90º. O operador do Laplaciano é usado para destacar continuidades em uma imagem. As descontinuidades são definidas com maior precisão, e são os pontos em que ocorre o *zero-crossing*, ou seja, a função muda de valores positivos para negativos.

Comparando o resultado de detecção de descontinuidades usando a primeira e a segunda derivada, observa-se que o Gradiente produz geralmente bordas grossas e tem uma resposta mais forte a cada passo dos valores de tons de cinza. Por outro lado, o Laplaciano apresenta uma melhor resposta para finos detalhes como pontos e linhas estreitas, além de gerar uma dupla resposta nas mudanças dos valores de tons de cinza. Uma comparação analisando o comportamento gráfico da função, do gradiente e do Laplaciano, pode ser visto na Figura 2.18.

Algumas informações são extraídas da matriz Hessiana através dos seus autovalores e autovetores.

- O autovetor de maior valor absoluto indica a direção de maior curvatura da função.
- A direção da menor curvatura, por sua vez é aponta pelo autovetor de menor valor absoluto.
- Os respectivos autovalores indicam o valor dessas curvaturas.
- Os autovetores são sempre ortogonais.
- Os autovalores são invariáveis com relação a rotação.

Um método muito utilizado para obtenção dos autovalores e autovetores é o algoritmo QR.

2.7.4 Algoritmo QR

O algoritmo QR tem como um componente importante a decomposição QR. Esta consiste em decompor uma matriz quadrada A em uma matriz ortogonal Q e uma triangular superior R, usando para isso o processo Gram-Schmidt de ortogonalização.

Neste processo assume-se que são conhecidas algumas bases $w_1, ..., w_n$ de um vetor v de n dimensões e serão criadas as bases ortogonais $u_1, ..., u_n$. Uma base é o conjunto de vetores que gera o espaço do qual ele é base. E uma base é chamada ortogonal se o produto $u_i \cdot u_j$ for igual a zero, para todo $i \neq j$.

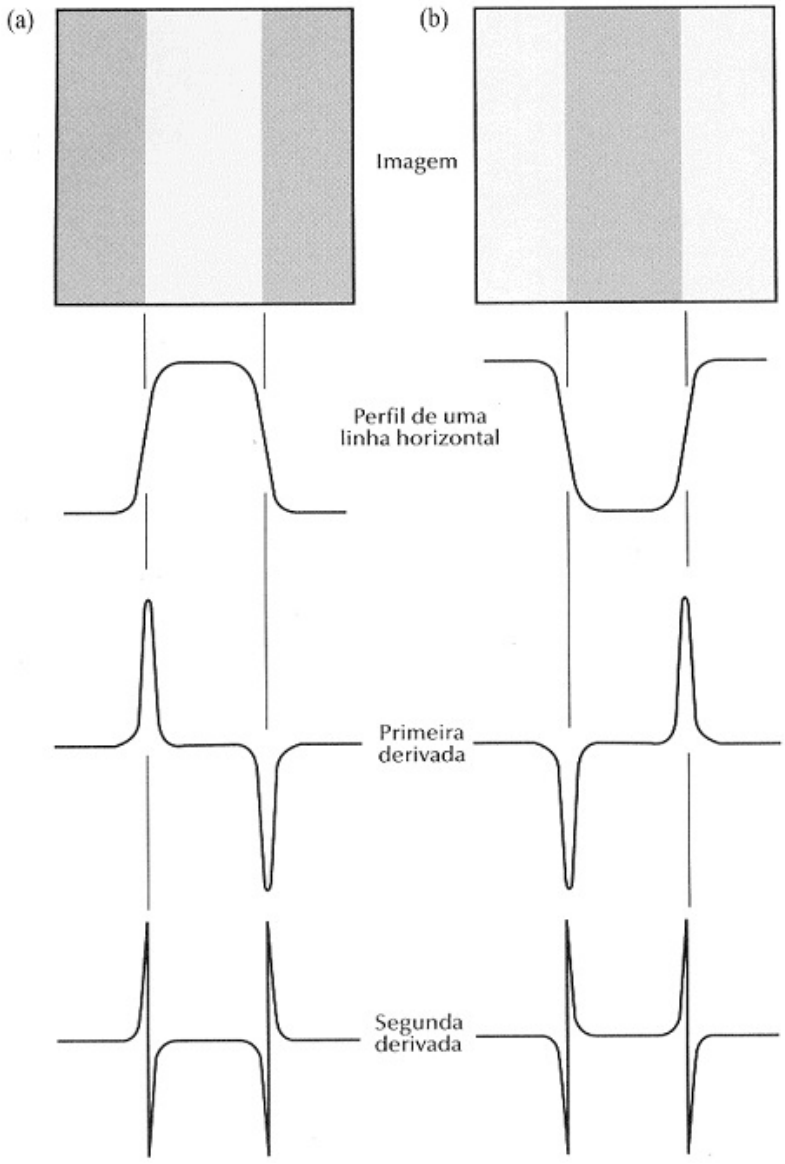


Figura 2.18: Comportamento grafico da função que representa uma imagem, a forma da função gradiente e do Laplaciano, assim como o ponto de localização da borda no gráfico [Gonzalez & Woods 2000].

Inicialmente para o processo de Gram-Schmidt obtemos u_1 com:

$$u_1 = \frac{w_1}{||w_1||}$$

Onde $||w_n|| = \sqrt{w_n \cdot w_n}$. Em seguida subtraem-se os múltiplos adequados de u_1 de todos os vetores bases restantes afim de organizar a ortogonalidade de u_1. Isso é realizado da forma:

$$w_k^{(2)} = w_k - w_k \cdot u_1 u_1, \qquad k = 2, ..., n.$$

A segunda base ortogonal é então conseguida com:

$$u_2 = \frac{w_2^{(2)}}{||w_2^{(2)}||}$$

Para a próxima base ortogonal realizamos o seguinte cálculo:

$$w_k^{(3)} = w_k^{(2)} - w_k^{(2)} \cdot u_2 u_2, \qquad k = 3, ..., n.$$

Então $u_3 = w_3^{(3)}/||w_3^{(3)}||$. Todo o processo é generalizado da seguinte maneira [Olver 2010]:

$$u_j = \frac{w_j^{(j)}}{||w_j^{(j)}||}, \quad w_k^{(j+1)} = w_k^{(j)} - w_k^{(j)} \cdot u_j u_j, \quad k = j+1, ..., n, \quad j = 1, ..., n.$$

Considerando $w_1, ..., w_n$ serem bases de $\Re^n$ e $u_1, ..., u_n$ serem as correspondentes bases ortogonais resultantes do processo Gram-Schmidt. Assim podemos montar com esses vetores as colunas de duas matrizes, da seguinte forma:

$$A = (w_1 w_2 ... w_n) \qquad Q = (u_1 u_2 ... u_n)$$

Com isso u_i forma uma base ortogonal e Q é uma matriz ortogonal. Utilizando o processo de Gram-Schmidt a matriz A pode ser decomposta em uma matriz Q e outra R triangular superior.

$$A = QR, \qquad onde \qquad R = \begin{vmatrix} r_{11} & r_{12} & ... & r_{1n} \\ 0 & r_{22} & ... & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & r_{nn} \end{vmatrix}$$

Assim, qualquer matriz não singular A, pode ser fatorada em A = QR, ou seja, no produto de uma matriz ortogonal Q e uma matriz triangular superior R, onde R_{ij}, $0 \leq i \leq n$, $i \leq j \leq n$ são os elementos diferentes de zero que estão na diagonal principal e acima dela. A decomposição é única se todas as diagonais de entrada de R são assumidas positivas. Um pseudocódigo da decomposição QR está no Algoritmo 1.

Algorithm 1 Um pseudocódigo da decomposição QR [Olver 2010].

for j= 0 to 10 **do**
  $r_{jj} = \sqrt{a_{1j}^2 + \cdots + a_{nj}^2}$
  if $r_{jj} == 0$ **then**
    print "É uma coluna linearmente dependente"
  else
    for i = 1 to n **do**
      $a_{ij} = a_{ij}/r_{jj}$
    end for
  end if
  for k = j + 1 to n **do**
    $r_{jk} = a_{1j}a_{1k} + \cdots + a_{nj}a_{nk}$
    for i = 1 to n **do**
      $a_{ik} = a_{ik} - a_{ij}r_{jk}$
    end for
  end for
end for

O algoritmo QR foi proposto inicialmente por Francis e Kublanovskaya trabalhando independentemente em 1961. A idéia é simples e baseia-se em uma sequência de decomposições QR. O primeiro passo é fatorar a matriz A.

$$A = A_1 = Q_1 R_1$$

Em seguida multiplica-se as matrizes Q_1 e R_1 em ordem inversa para se obter uma nova matriz.

$$R_1 Q_1 = A_2$$

Repetindo a fatoração, consegue-se novas matrizes Q e R.

$$A_2 = Q_2 R_2$$

Realizando esses passos sucessivamente, podemos resumir o algoritmo da seguinte maneira.

$$A = Q_1 R_1 \qquad R_k Q_k = A_{k+1} = Q_{k+1} R_{k+1}, \qquad k = 1, 2, 3, ...$$

Onde Q_k, R_k são do passo anterior, e Q_{k+1} e R_{k+1} são as matrizes do passo atual. O algoritmo é executado até que os elementos da diagonal principal de Q_k sejam todos iguais a 1. Quando isso acontecer os elementos da diagonal principal de A_k serão os autovalores procurados da matriz A. Para obtermos os autovetores, realizamos a multiplicação das matrizes Q de cada passo do algoritmo.

$$S_1 = Q_1, \qquad S_k = S_{k-1} Q_k = Q_1 Q_2 ... Q_{k-1} Q_k, \qquad k > 1$$

Com isso a matriz S ($S_k \longrightarrow S$) é uma matriz ortogonal em que as suas colunas são os autovetores dos autovalores da matriz A_k.

O conhecimento dos grupos em que os métodos de segmentação vascular são classificados e uma visão geral dos métodos, são importantes para conhecer um pouco sobre a área de segmentação vascular e entender as vantagens e desvantagens de cada método. Uma maior abordagem ao método de *height ridges* é necessária para compreender alguns pontos do método proposto, como a separação dos pontos analisados. O gradiente e o laplaciano são utilizados para obtenção de dados a serem usados no método para verificar condições para remover pontos que não pertencem a linhas centrais. O algoritmo QR é usado para obtenção dos autovalores e autovetores da matriz hessiana, e que também são necessários no método deste trabalho.

Capítulo 3

O Método de *Height Ridge* em Paralelo

Neste capítulo são descritos os detalhes de implementação do método de *Height Ridge* em Paralelo, que é dividido em 7 etapas. Na etapa inicial uma limiarização é realizada para separar os voxels que são pertencentes aos vasos e reduzir a quantidade de pontos processados. Na segunda etapa um filtro gaussiano reduz ruídos no volume selecionado pelo usuário. A terceira etapa executa novamente uma limiarização. A quarta e quinta etapas calculam, respectivamente, os componentes do vetor gradiente e a matriz hessiana do voxel processado por uma *thread* de CUDA. Os autovalores e os autovetores da matriz hessiana são obtidos na sexta etapa. Na sétima etapa são realizadas as verificações para segmentar os pontos que são linhas centrais. A organização das etapas do método proposto são vistas na Figura 3.1. Vale salientar que a etapa de percorrimento existente no método de Aylward, foi removida no método proposto deste trabalho.

Na primeira e terceira etapas, uma limiarização é realizada sobre os voxels do volume selecionado pelo usuário, para separar apenas os voxels que compõem os vasos e assim diminuir a quantidade de voxels processados e o tempo de execução do método. Os valores de limiar inferiores e superiores variam de acordo com os dados analisados, assim nos dados de teste deste trabalho, os limiares inferior e superior foram respectivamente 30 e 400.

Na segunda etapa, um filtro gaussiano passa baixa é executado no volume selecionado pelo usuário para diminuir o ruído e melhorar a qualidade da segmentação. Inicialmente no trabalho original o usuário seleciona um ponto inicial (*seed*) que será verificado se é um ponto pertencente à *centerline* do vaso. Em contraste, no método proposto, o usuário seleciona uma parte do volume total, cujo limite depende da quantidade de memória da placa gráfica, para que a verificação das linhas centrais seja feita em cada voxel do volume selecionado. Este volume é então copiado para um vetor unidimensional, com uma quantidade de índices suficientes para comportar todos os voxels do volume determinado pelo usuário. A cópia foi feita com a função `memcpy` da biblioteca *string* [Schildt 1996], copiando linha por linha do volume, evitando assim a cópia voxel por voxel.

Com os dados selecionados pelo usuário, alocados no vetor, eles são então copiados para a memória da placa gráfica (*device*) e a organização das *threads* por bloco é determinada. Esta organização contou com um bloco de dimensões $2 \times 2 \times 2$, ou seja, oito *threads* por bloco, pois um número maior que oito de *threads* causava o uso de registradores por *kernel*, maior que o permitido por CUDA. O *grid* de blocos tem dimensões `X` e `Y` respectivamente determinado pelas equações 3.1 e 3.2. A quantidade de blocos utilizada

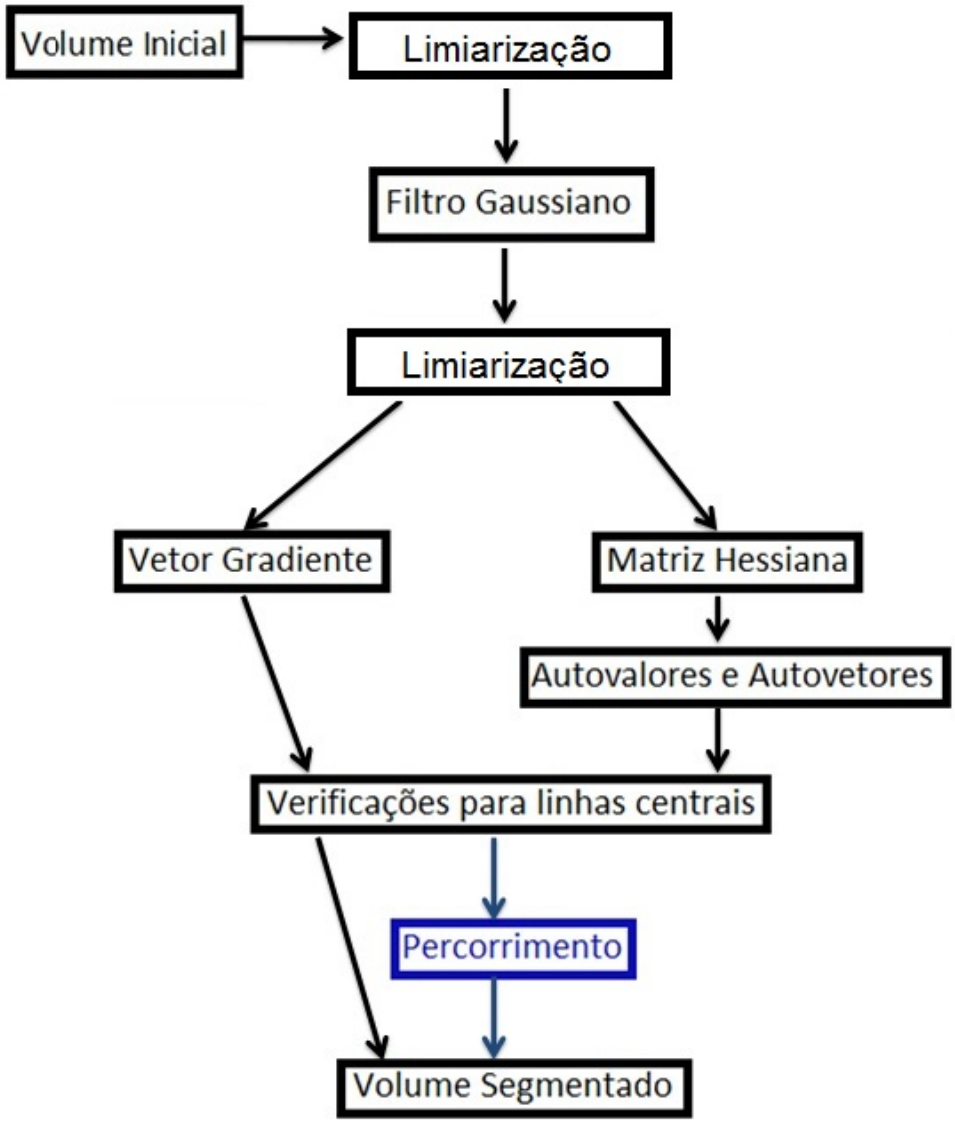


Figura 3.1: Organização das etapas do método proposto. A etapa de percorrimento, destacada em azul, existente no método de Aylward foi removida neste trabalho.

é o produto entre os valores X e Y das equações 3.1 e 3.2.

$$\mathtt{X} = \frac{\mathtt{larguraDoVS}}{\mathtt{dimensãoDoBloco}} \tag{3.1}$$

$$\mathtt{Y} = \left(\frac{\mathtt{larguraVS}}{\mathtt{dimensãoDoBloco}}\right) * \left(\left(\frac{\mathtt{profundidadeDoVS}}{\mathtt{dimensãoDoBloco}}\right) + 1\right) \tag{3.2}$$

Onde larguraDoVS e profundidadeDoVS são respectivamente a largura e a profundidade do volume selecionado pelo usuário, enquanto que dimensãoDoBloco é a dimensão do bloco de *threads*. A soma do valor 1 na equação 3.2 é necessária para a situação em que o resultado da divisão do segundo termo da multiplicação seja um valor ímpar, o que pode causar falha na busca de índice do vetor de dados durante a execução do *kernel*, por procurar um índice em uma posição maior que a existente no vetor de dados.

O *kernel* do filtro gaussiano é então processado sobre cada índice do vetor unidimensional de dados em paralelo, onde cada índice corresponde a uma posição no volume. A relação da posição do voxel no volume com seu respectivo índice no vetor é feita com as equações.

$$\mathtt{x = blockIdx.x * tamanho + threadIdx.x} \tag{3.3}$$

$$\mathtt{y} = \left(\mathtt{blockIdx.y} - \left(\left(\frac{\mathtt{yDim}}{2}\right) * \left(\frac{\mathtt{blockIdx.y}}{(\mathtt{yDim}/2)}\right)\right)\right) * \mathtt{tamanho} + \mathtt{threadIdx.y} \tag{3.4}$$

$$\mathtt{z} = \left(\left(\frac{\mathtt{blockIdx.y}}{(\mathtt{yDim}/2)}\right) * \mathtt{tamanho}\right) + \mathtt{threadIdx.z} \tag{3.5}$$

$$\mathtt{zInc = yDim * xDim} \tag{3.6}$$

$$\mathtt{indice = (x + (y*yDim))+(z*zInc)} \tag{3.7}$$

As três primeiras equações, 3.3, 3.4 e 3.5 são usadas para encontrar a posição do voxel no volume, processado por uma determinada *thread*. Nestas equações o tamanho é a dimensão do bloco de *threads*, yDim e xDim são respectivamente a altura e a largura do volume selecionado pelo usuário. A variável zDim é a quantidade de voxels em uma fatia do volume selecionado pelo usuário, enquanto que as demais são variáveis pré-definidas por CUDA. A Equação 3.7, calcula o índice do voxel da posição x, y e z do volume. A determinação da correspondência entre a posição de um voxel e o índice do vetor de dados é necessária para o conhecimento dos vizinhos do voxel, para a execução do filtro gaussiano.

A dimensão do filtro gaussiano é $3 \times 3 \times 3$, pelo fato de estar sendo executado em um volume tridimensional, e é visto na Figura 3.2

As próximas etapas do algoritmo são executados em um único *kernel*, que também processa todos os voxels do volume selecionado pelo usuário em paralelo, utilizando a mesma organização de *threads* e de dados definidos no *kernel* do filtro gaussiano.

A quarta etapa consiste em calcular os componentes do vetor gradiente, com as equações de derivadas parciais:

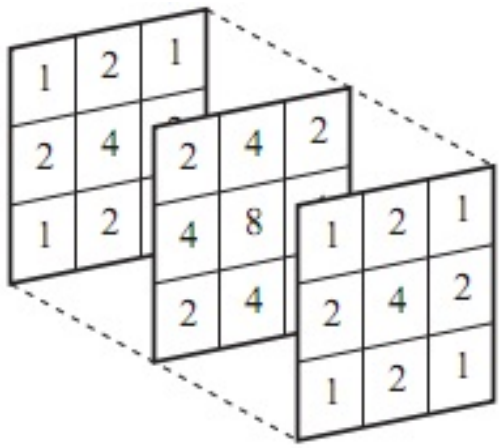


Figura 3.2: Visualização do filtro gaussiano usado no nosso método [Waltza & Miller 1998].

$$G_x = f(x+1, y, z) - f(x, y, z) \tag{3.8}$$

$$G_y = f(x, y+1, z) - f(x, y, z) \tag{3.9}$$

$$G_z = f(x, y, z+1) - f(x, y, z) \tag{3.10}$$

Em busca de melhorias no segmentação do método proposto, foi testada a obtenção dos valores dos componentes do vetor gradiente utilizando, os filtros de Sobel [Pradabpet et al. 2009] e por diferença centrais. No entanto, os filtros de Sobel não possuíam uma máscara para o componente z do vetor gradiente. Para superar esta dificuldade as máscaras dos componentes x e y foram rotacionadas para se adaptarem para o componente z. Entretanto, os resultados apresentavam um pouco de ruído e linhas centrais mais grossas do que os vistos com derivadas parciais. Este último problema também foi observado quando se utilizou diferenças centrais para o cálculo dos componentes do vetor gradiente.

Na quinta etapa os componentes da matriz hessiana (2.12) são obtidos com as equações 2.13, 2.14, 2.15, 2.13, 2.16, 2.17 e 2.18 (ver capítulo 2 2).

Na sexta etapa os autovalores e autovetores da matriz hessiana são calculados com o algoritmo QR e em seguida são ordenados de maneira crescente, para a próxima etapa.

A sétima etapa verifica se um voxel é a linha central de um vaso. Esta verificação consiste em três condições que utilizam informações do vetor gradiente (∇f) dos autovalores ordenados como $\lambda_1 \leq \lambda_2 \leq \lambda_3$, e autovetores v_1, v_2, v_3 da matriz hessiana do voxel. As duas primeiras são relacionadas a detecção de *height ridges*.

$$\lambda_1 \leq \lambda_2 < 0$$

$$v_1 \cdot \nabla f = 0 \quad e \quad v_2 \cdot \nabla f = 0$$

A terceira condição define se o *ridge* é centro ou não. É importante porque em alguns casos o vaso é muito fino e se confunde com as bordas. O valor de r varia entre $0 < r < 1$

e é um dos critérios para se designar a quantidade de pontos segmentados.

$$\frac{\lambda_2}{\lambda_1} \geq \mathtt{r}$$

As três condições acima são as testadas neste trabalho, visto que as demais etapas do método de [Aylward & Bullitt 2002] buscam outros pontos que compõem a linha central, realizando o percorrimento citado na Figura 3.1. O percorrimento é realizado de maneira diferente, de acordo com a classificação do voxel processado em *centerline* ou não. Se o voxel processado não for *centerline* é verificado se o menor autovalor (λ_1) é menor que zero, caso seja, o autovetor v_1 é utilizado para definir a direção da localização do próximo voxel a ser testado. Caso o autovalor (λ_1) não for maior que zero, é retornado que nenhum *ridge* foi encontrado. Se o voxel processado for *centerline*, o autovalor v_3 é usado como vetor tangente inicial e indica a direção da posição do próximo ponto candidato a linha central.

Este percorrimento não é necessário em nosso método porque toda região ao redor da área desejada é processada de uma só vez e rapidamente, facilitando a seleção por parte do usuário e obtenção dos resultados. Além disso, verificações feitas no trabalho de [Aylward & Bullitt 2002] para conectar linhas centrais que podem ficar descontínuas durante a segmentação ou ligar bifurcações, também são desnecessárias pois todos os pontos do volume são testados e segmentados como *centerlines* se assim forem, construindo as linhas centrais naturalmente.

Devido os cálculos executados na implementação resultarem em valores aproximados, o vetor gradiente tem seus resultados aproximados. Assim, nas verificações são feitas mudanças nos valores dos componentes do vetor gradiente para zero desde que estejam dentro de uma faixa de valores. As faixas da aproximação dos componentes do gradiente são:

$$|G_x| < \mathtt{k}, \qquad |G_y| < \mathtt{k} \quad e \quad |G_z| < \mathtt{k},$$

onde `i`=1,2,3, `k` são os valores de referência para a aproximação que devem ser positivos. Quanto mais próximo de zero forem os valores de `k` menos pontos são segmentados. Caso não seja realizado o arredondamento nenhum ponto é selecionado. Isto é causado pelos cálculos executados no método que retornam valores aproximados e não os valores precisos da função gradiente.

Capítulo 4

Metodologia

Neste capítulo é descrita a metodologia de análise do método explicado no capítulo anterior.

4.1 Materiais

Conforme descrito no Capítulo 1, o método usado no trabalho objetiva reduzir o tempo de processamento para obtenção de *centerlines*, usando uma abordagem paralela implementada em CUDA. O trabalho de Aylward [Aylward & Bullitt 2002] é utilizado como principal referência para o trabalho por mostrar-se eficaz com problemas de ruído e possuir uma formato paralelizável. Os experimentos foram realizados em dados artificiais e em imagens médicas reais.

4.1.1 Dados Artificiais

Os dados artificiais correspondem a quatro estruturas tubulares de diâmetros diferentes e com a forma de um cilindro. Estas estruturas foram criadas para imitar a intensidade dos vasos, ou seja, maiores valores de intensidade no centro diminuindo gradativamente para as bordas. Os diâmetros das quatro estruturas são respectivamente 3, 5, 9 e 17 voxels. Estes valores foram escolhidos por serem próximos aos diâmetros de muitos vasos das imagens médicas, como pode ser visto na Figura 4.5. A Figura 4.1 mostra o topo dos cilindros criados após a aplicação de zoom na imagem, de maneira a ilustrar a diminuição da intensidade do centro para as bordas. Na Figura 4.2 são vistas as paredes dos cilindros de forma tridimensional com um zoom na imagem.

A função utilizada para gerar os dados artificiais originalmente é $z = x^2 + y^2$, que graficamente é vista na Figura 4.3.

No entanto, para formar o cilindro com as intensidades desejada ela foi modificada para:

$$z = h - \sqrt{\frac{m * x^2}{a} + \frac{n * y^2}{b}}$$

Assim o gráfico da função foi invertido e em seguida elevado em relação ao plano formato pelos eixos x e y. Desta forma, os pontos de maior intensidade ficam no centro e os demais vão gradativamente reduzindo de maneira uniforme até as paredes dos vasos.

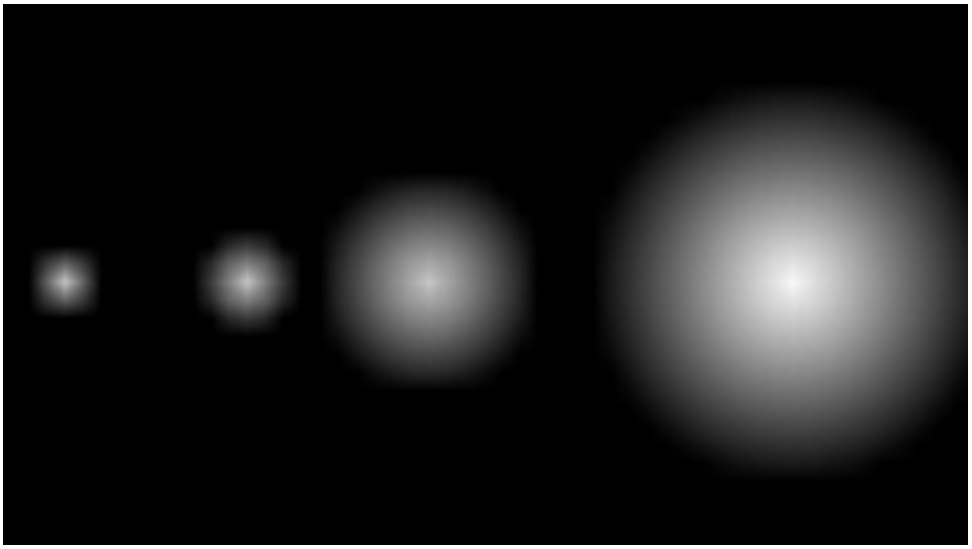


Figura 4.1: Dados artificiais de formato cilíndrico criados para experimento, com raios da esquerda para a direita, respectivamente 3, 5, 9 e 17 voxels.

Na fórmula, h controla a elevação da função em relação ao plano Cartesiano formado pelos eixos x e y. A raiz quadrada serve para suavizar as bordas da função, para que ela tenha um formato circular. Os valores de m e n afunilam mais o gráfico nos eixos x e y respectivamente, com isso controla-se a intensidade dos pontos centrais. Com a e b tem-se controle sobre a largura do cilindro. As quatro equações que geraram os cilindros dos dados artificiais de são:

$$\texttt{raio} = 3 \quad z = 200 - \sqrt{\frac{500 * x^2}{0.05} + \frac{500 * y^2}{0.05}} \tag{4.1}$$

$$\texttt{raio} = 5 \quad z = 200 - \sqrt{\frac{500 * x^2}{0.1} + \frac{500 * y^2}{0.1}} \tag{4.2}$$

$$\texttt{raio} = 9 \quad z = 200 - \sqrt{\frac{100 * x^2}{0.1} + \frac{100 * y^2}{0.1}} \tag{4.3}$$

$$\texttt{raio} = 17 \quad z = 250 - \sqrt{\frac{500 * x^2}{1} + \frac{500 * y^2}{1}} \tag{4.4}$$

A equação que gera o cilindro de raio 9 dos dados artificiais é vista graficamente na Figura 4.4.

4.1.2 Dados Médicos

Os arquivos médicos são imagens DICOM de TC de um angiograma cerebral de dimensões $512 \times 512 \times 132$, com espaçamento entre voxels de 0,39 milímetros para x e y, e para z 0,6 milímetros. Uma fatia das imagens é vista na Figura 4.5, nesta figura é pos-

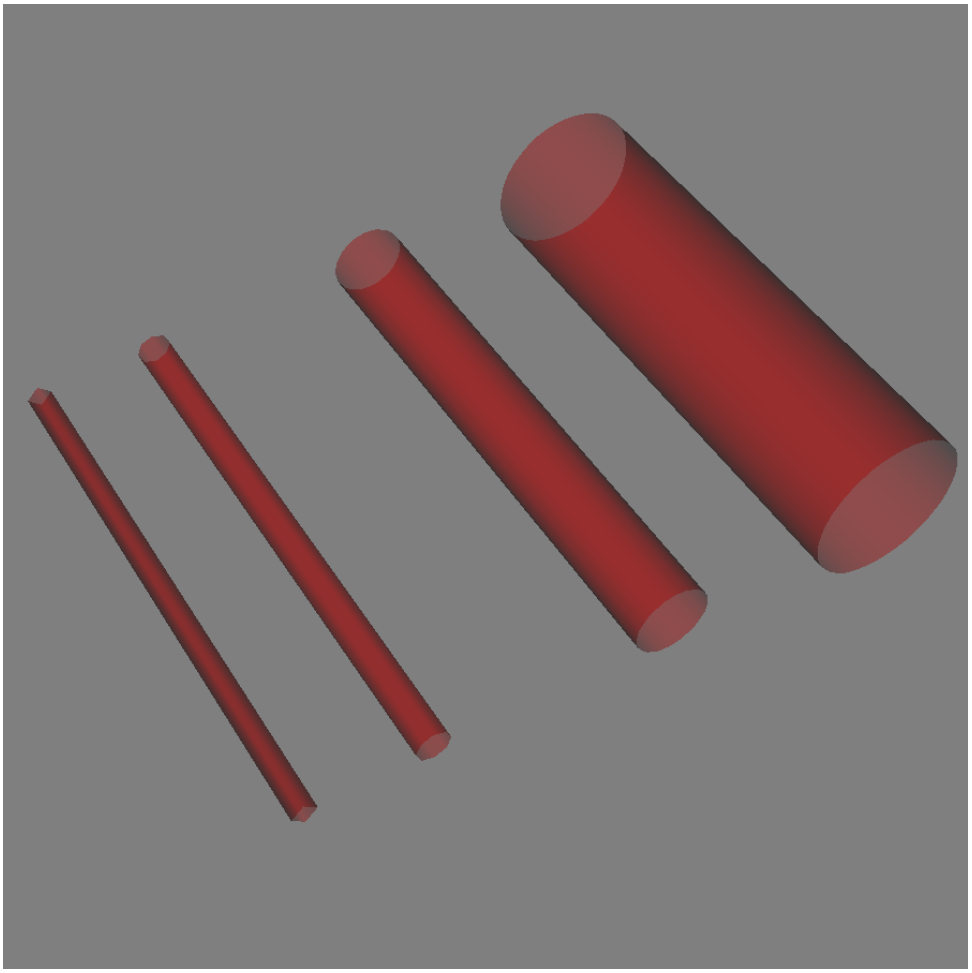


Figura 4.2: Os dados artificiais vistos de maneira tridimensional.

sível ver o topo dos quatro cilindros dos dados artificiais dentro do retângulo vermelho e observar que os seus diâmetros são próximos a de muitos vasos presentes na imagem médica.

4.1.3 Sistema Computacional

Para a implementação do método deste trabalho foi utilizada a linguagem C++, VTK e CUDA. A leitura dos arquivos DICOM usou a classe vtkDICOMImageReader através da variável `reader` e em seguida, um backup dos arquivos DICOM foram criados para a variável `dataCenterline`, para que dessa maneira a manipulação dos dados vindos de `reader` seja feita sem impedir a visualização dos dados originais gravados na variável `dataCenterline`, como é visto no algoritmo 4.1.

```
1 //ler arquivos DICOM
2 vtkDICOMImageReader *reader = vtkDICOMImageReader::New();
```

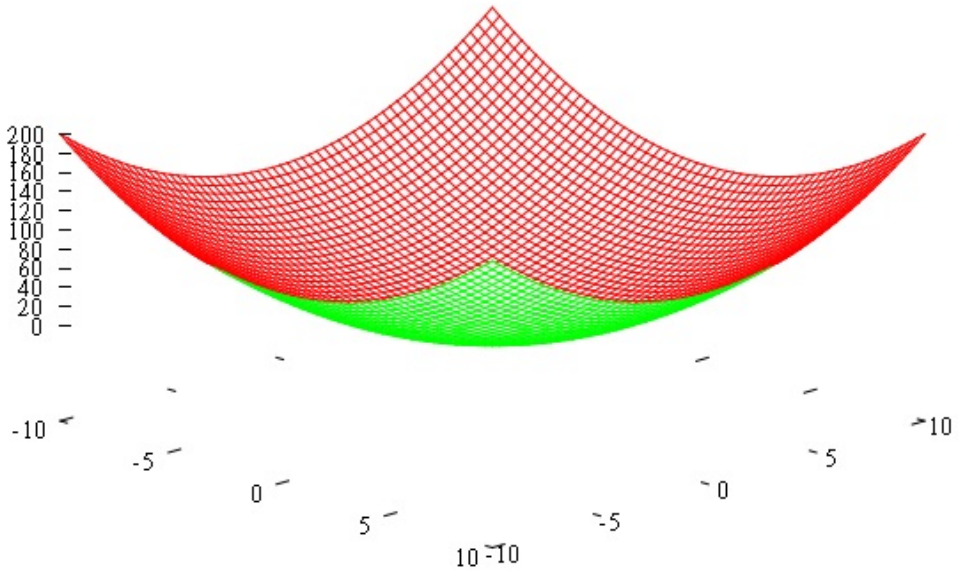


Figura 4.3: Gráfico da função original usada para gerar os dados artificiais [Williams et al. 2011].

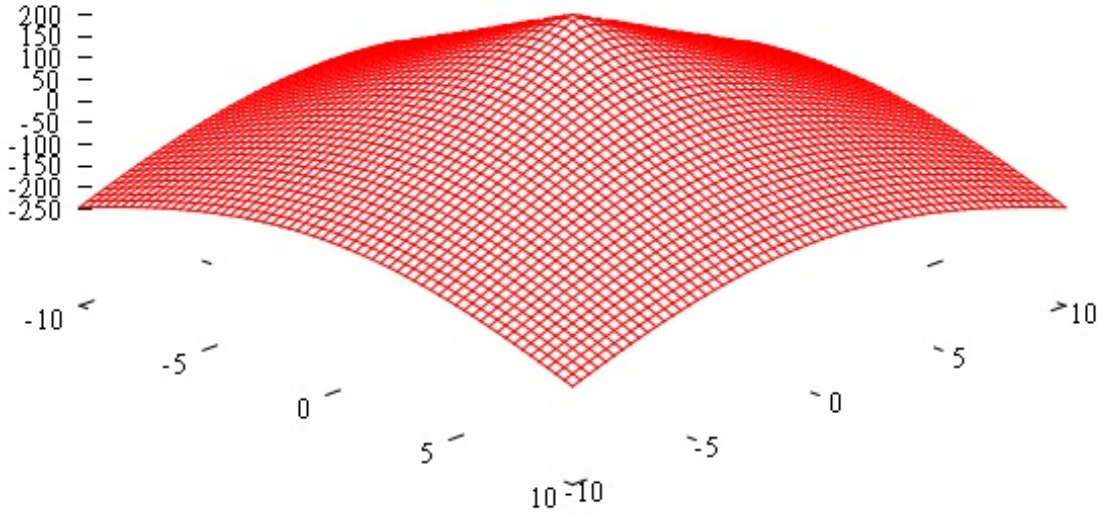


Figura 4.4: Gráfico da equação do cilindro de raio 9 dos dados artificiais [Williams et al. 2011].

Figura 4.5: Uma fatia do conjunto de imagens DICOM utilizada nos experimentos do método. O topo dos cilindros dos dados artificiais são vistos dentro do retângulo vermelho.

```
3  reader->SetDirectoryName("C:/cerebro3");
4  reader->Update();
5
6  //backup dos dados originais
7  int extend[6];
8  vtkImageData *dataCenterline = vtkImageData::New();
9  reader->GetOutput()->GetExtent(extend);
10 dataCenterline->SetExtent(extend);
11 dataCenterline->Update();
12 dataCenterline->DeepCopy(reader->GetOutput());
```

Listing 4.1: Leitura dos arquivos DICOM.

Em seguida são criadas e inicializadas as variáveis `H`, `W` e `D` que guardaram respectivamente a altura, largura e profundidade do volume de dados originais, como mostrado no algoritmo 4.2.

```
1 //H altura, W largua e D profundidade do volume
2 int H = reader->GetOutput()->GetDimensions()[0];
3 int W = reader->GetOutput()->GetDimensions()[1];
```

```
4 int D = reader->GetOutput()->GetDimensions()[2];
```

Listing 4.2: Dimensões do volume de dados originais.

O volume selecionado pelo usuário é definido em código para que seja mantido sempre o mesmo volume, a fim de se comparar as variações no resultado da segmentação durante os testes. A altura, largura e profundidade do volume selecionado pelo usuário é definido nas variáveis `Ht`, `Wt` e `Dt` respectivamente.

No algoritmo 4.3 também são mostradas a declaração das variáveis que guardam os dados iniciais (`dados`), os dados iniciais copiados para a placa gráfica (`dadosDevice`), os dados finais resultantes do processamento na placa gráfica (`dadosFinalDevice`), a variável `tIn` que guarda temporariamente o volume selecionado pelo usuário antes de ser copiado para a placa gráfica e `tOut` que guarda temporariamente o resultado da segmentação após ser copiado da placa gráfica para a memória RAM do computador para em seguida ser usada na visualização do resultado da segmentação. O volume selecionado originalmente pelo usuário é copiado para a variável `tInOrig`, afim de se criar a visualização dos dados originais juntamente com os das linhas centrais.

```
1 //tamanho do volume selecionado pelo usuario
2 int Ht = 80, Wt = 80, Dt = 30;
3 short *dados, *dadosDevice, *tIn, *tInOrig;
4 float *dadosFinalDevice, *tOut;
```

Listing 4.3: Dimensionamento do volume selecionado pelo usuário e declaração das variáveis que guardaram os dados a serem processados.

A cronometragem do tempo de execução do método usou funções de CUDA. A variável `tempoParcial` guarda o tempo parcial da execução e `tempoTotal` a quantidade do tempo total de execução e `timer` é a incrementadora do tempo. É necessário inicializar a API de CUDA antes da contagem de tempo para uma correta medição do tempo total de execução do método, pois CUDA demora um certo tempo para ser inicializada. Para a inicializar CUDA é criada a variável `inicializar` e é alocada uma pequena quantidade de memória na placa gráfica. A função `cutStartTimer()` começa a marcação do tempo, como visto no algoritmo 4.4.

```
1 //variaveis para contagem de tempo
2 float tempoParcial;
3 float tempoTotal = 0;
4 unsigned int timer = 0;
5 cutCreateTimer(&timer);
6
7 //iniciar a execução da API de CUDA
8 float* iniciarlizar;
```

```
9  CUDA_SAFE_CALL( cudaMalloc((void**) &iniciarlizar, sizeof(float)));
10
11 //inicia o cronometro
12 cutStartTimer(timer);
```

Listing 4.4: Declaração das variáveis para cronometragem do tempo e inicialização da API de CUDA.

O próximo passo é visto no algoritmo 4.5 onde é alocada memória para as variáveis que guardaram os dados e é mostrado o tempo de duração da alocação.

```
1  //aloca memoria no host
2  dados = (short*) malloc(sizeof(short)*H*W*D);
3  tIn = (short*) malloc(sizeof(short)*Ht*Wt*Dt);
4  tInOrig = (short*) malloc(sizeof(short)*Ht*Wt*Dt);
5  tOut = (float*) malloc(sizeof(float)*Ht*Wt*Dt);
6
7  //aloca memoria no device
8  cudaMalloc((void**)&dadosDevice, sizeof(short)*Ht*Wt*Dt);
9  cudaMalloc((void**)&dadosFinalDevice, sizeof(float)*Ht*Wt*Dt);
10
11 tempoParcial = cutGetTimerValue(timer);
12 tempoTotal += tempoParcial;
13 cout << "Inicializa␣memória␣GPU␣em:␣" << tempoParcial << "␣ms" << end
```

Listing 4.5: Alocação de memória para as variáveis que guardaram os dados processados e calculo do tempo usado na alocação.

Em seguida os cilindros dos dados artificiais são criados em uma pequena parte do volume dos dados originais de acordo com o mostrado no algoritmo 4.6, usando as equações 4.1, 4.2, 4.3 e 4.4.

```
1  //cria dados artificiais
2  for(int x=-5; x<6; x++){
3    for(int y=-5; y<6; y++){
4      for(int z=0; z<30; z++){
5
6  float vf0 = 200-sqrt(500.0*x*x/0.05+500*y*y/0.05);
7  reader->GetOutput()->
8  SetScalarComponentFromFloat(x+15, y+50, z+10, 0, vf0);
9
10     }
11   }
```

```
12 }
13
14 for(int x=-5; x<6; x++){
15   for(int y=-5; y<6; y++){
16     for(int z=0; z<30; z++){
17
18 float vf1 = 200-sqrt(500.0*x*x/0.1+500*y*y/0.1);
19 reader->GetOutput()->
20 SetScalarComponentFromFloat(x+25, y+50, z+10, 0, vf1);
21
22     }
23   }
24 }
25
26 for(int x=-5; x<6; x++){
27   for(int y=-5; y<6; y++){
28     for(int z=0; z<30; z++){
29
30 float vf2 = 200-sqrt(100.0*x*x/0.1+100*y*y/0.1);
31 reader->GetOutput()->
32 SetScalarComponentFromFloat(x+45, y+50, z+10, 0, vf2);
33
34     }
35   }
36 }
37
38 for(int x=-10; x<11; x++){
39   for(int y=-10; y<11; y++){
40     for(int z=0; z<30; z++){
41
42 float vf3 = 250-sqrt(500.0*x*x+500*y*y);
43 reader->GetOutput()->
44 SetScalarComponentFromFloat(x+65, y+50, z+10, 0, vf3);
45
46     }
47   }
48 }
```

Listing 4.6: Criação dos cilindros dos dados artificiais.

O cronometro é zerado com a função de CUDA `cutResetTimer()` para medir o tempo de copia dos dados do leitor de arquivos DICOM (`reader`) para um vetor unidimensional. A copia é importante pois quando os dados estão em um vetor unidimensional o processamento dos dados em CUDA é mais simples. A classe vtkImageExport copia

os dados do leitor para o vetor unidimensional `dados`. O tempo de duração desta copia é então obtido e mostrado na tela, como está no algoritmo 4.7.

```
1  cutResetTimer(timer);
2
3  //copia os dados do volume selecionado pelo usuario para o vetor "dado
4  vtkImageExport *exporter = vtkImageExport::New();
5  exporter->SetInput(reader->GetOutput());
6  exporter->ImageLowerLeftOn();
7  exporter->Update();
8  exporter->Export(dados);
9
10 tempoParcial = cutGetTimerValue(timer);
11 tempoTotal += tempoParcial;
12 cout << "Copia para vetor em: " << tempoParcial << endl;
```

Listing 4.7: Copia dos dados DICOM para um vetor unidimensional e medição do tempo para realização da copia.

O algoritmo 4.8 copia o volume selecionado pelo usuário para a variável `tIn`. A copia é feita coluna por coluna. As variáveis X, Y e Z contém as coordenadas de inicio do volume selecionado, então é feito o backup desse volume. É mostrado também o tempo levado para copia do volume selecionado pelo usuário.

```
1  cutResetTimer(timer);
2
3  //copiando o volume selecionado pelo usuario para uma variavel a
4  //ser utilizada para copiar os dados para o device
5  for(int k=0; k<Dt; k++){
6    for(int i=0; i<Ht; i++){
7      memcpy(&tIn[(i*Ht)+(k*Ht*Wt)],
8        &dados[(X + ((Y+i)*H))+((Z+k)*zInc)],
9        sizeof(short)*Ht
10     );
11   }
12 }
13
14 memcpy(tInOrig, tIn, sizeof(short)*Ht*Wt*Dt);
15
16 tempoParcial = cutGetTimerValue(timer);
17 tempoTotal += tempoParcial;
18 cout << "Obtem volume selecionado: " << tempoParcial << endl;
```

Listing 4.8: Copia do volume selecionado pelo usuário para uma variável temporária para em seguida o volume ser copiado para a placa gráfica. O tempo usado na copia também é medido e mostrado.

O volume selecionado pelo usuário, que é copiado para a placa gráfica com a função `cudaMemcpy()`, assim como a definição das dimensões dos blocos e do grid de CUDA usando as funções `dimBlock()` e `dimGrid()` respectivamente são mostrados no algoritmo 4.9. Cada bloco foi definido com dimensões $2 \times 2 \times 2$, pois com dimensões maiores seriam usadas mais *threads* por bloco do que o suportado por CUDA. As dimensões do grid usaram a equação 3.1 para a dimensão x e a equação 3.2 para a y.

```
1  cutResetTimer(timer);
2
3  //copia os dados para a placa grafica
4  cudaMemcpy(dadosDevice, tIn, sizeof(short)*Ht*Wt*Dt,
5    cudaMemcpyHostToDevice);
6
7  //dimensionamento dos blocos e do grid de CUDA
8  int blocoDim = 2;
9
10 dim3 dimBlock(blocoDim, blocoDim, blocoDim);
11 dim3 dimGrid(Wt/blocoDim, Wt/blocoDim * (Dt/blocoDim));
```

Listing 4.9: Copia do volume selecionado pelo usuário para a memória da placa gráfica e o dimensionamento dos blocos e do grid de *threads*.

Em seguida são chamados os dois *kernels* que executaram na placa gráfica. O *kernel* `gaussiano` processa o filtro gaussiano e o *kernel* `calc` obtem o vetor gradiente, a matriz hessiana, os autovalores e os autovetores de cada voxel do volume selecionado pelo usuário. O primeiro parâmetro de cada *kernel* são os dados de entrada, enquanto que o segundo parâmetros são os dados de saída. Os demais parâmetros vistos no algoritmo 4.10 são utilizados para obtenção dos índice das *threads* que processam o volume selecionado pelo usuário.

```
1  //calcula o gaussiano
2  gaussiano<<<dimGrid, dimBlock>>>(
3    dadosDevice, dadosFinalDevice, blocoDim,
4    Ht, Ht*Wt, Ht, Wt, Dt
5  );
6
7  calc<<<dimGrid, dimBlock>>>(
8    dadosFinalDevice, dadosDevice, blocoDim,
9    Ht, Ht*Wt, Ht, Wt, Dt
```

```
10 );
```

Listing 4.10: Inicia a execução dos dois *kernels*.

A primeira parte do *kernel* `gaussiano` é visto no algoritmo 4.11 onde são mostrados os parâmetros do *kernel*, juntamente com as fórmulas para obtenção da posição x, y e z de um determinado voxel no volume selecionado pelo usuário utilizando as posições das *threads* nos blocos de CUDA. Para em seguida, usar a posição do voxel no volume, e encontrar o índice desse voxel no vetor unidimensional onde os dados estão armazenados.

```
1  __global__ void gaussiano(short *dadosIn, float *dadosOut,
2  int tamanho, int yInc, int zInc, int HtParam, int WtParam,
3  int DtParam){
4
5    //calculo do indice do vetor em relacao a posicao do voxel
6    //no volume tridimensional selecionado pelo usuario
7    int x = blockIdx.x * tamanho + threadIdx.x;
8    int y = ( blockIdx.y - ((yInc/2)*(blockIdx.y/(yInc/2))) )*
9      tamanho + threadIdx.y;
10   int z = ((blockIdx.y/(yInc/2)) * tamanho) + threadIdx.z;
11
12   int indice = (x + (y*yInc))+(z*zInc);
13
14 ...
```

Listing 4.11: Lista de parâmetros do *kernel* e obtenção do índice de um voxel no vetor unidimensional a partir da posição do volume selecionado.

A segunda parte do *kernel* `gaussiano` é mostrada no algoritmo 4.12. O *if* existente em suas seis primeiras condições, evita que o filtro gaussiano seja aplicado em voxels da borda do volume selecionado e as duas últimas são os limiares inferior e superior respectivamente. Os voxels que não estão dentro dos limites da limiarização, ou seja, não pertencem a vasos, tem sua intensidade zerada, afim de se evitar problemas de visualização posteriormente. A limiarização é importante para redução no tempo de processamento pois reduz a quantidade de voxels processados. A função `calcIndice()` usada no algoritmo 4.12 é mostrada no algoritmo 4.13 e calcula o índice de um voxel no vetor de dados unidimensional de acordo com a sua posição x, y e z no volume selecionado pelo usuário.

```
1 ...
2
3 if(x>1 && y>1 && z>1 && x<HtParam-1 && y<WtParam-1 &&
4 z<DtParam-1 && dadosIn[indice]>30 && dadosIn[indice]<400){
5
```

```
6   //passa o filtro gaussiano blur
7   dadosOut[indice] =
8   0.5*((dadosIn[calcIndice(x-1, y+1, z-1, yInc, zInc)] +
9   2 *dadosIn[calcIndice(x, y+1, z-1, yInc, zInc)] +
10  dadosIn[calcIndice(x+1, y+1, z-1, yInc, zInc)] +
11  2 * dadosIn[calcIndice(x-1, y, z-1, yInc, zInc)] +
12  4 * dadosIn[calcIndice(x, y, z-1, yInc, zInc)] +
13  2 * dadosIn[calcIndice(x+1, y, z-1, yInc, zInc)] +
14  dadosIn[calcIndice(x-1, y-1, z-1, yInc, zInc)] +
15  2 *dadosIn[calcIndice(x, y-1, z-1, yInc, zInc)] +
16  dadosIn[calcIndice(x+1, y-1, z-1, yInc, zInc)] +
17
18  2 * dadosIn[calcIndice(x-1, y+1, z, yInc, zInc)] +
19  4 * dadosIn[calcIndice(x, y+1, z, yInc, zInc)] +
20  2 * dadosIn[calcIndice(x+1, y+1, z, yInc, zInc)] +
21  4 * dadosIn[calcIndice(x-1, y, z, yInc, zInc)] +
22  8 * dadosIn[calcIndice(x, y, z, yInc, zInc)] +
23  4 * dadosIn[calcIndice(x+1, y, z, yInc, zInc)] +
24  2 * dadosIn[calcIndice(x-1, y-1, z, yInc, zInc)] +
25  4 * dadosIn[calcIndice(x, y-1, z, yInc, zInc)] +
26  2 * dadosIn[calcIndice(x+1, y-1, z, yInc, zInc)] +
27
28  dadosIn[calcIndice(x-1, y+1, z+1, yInc, zInc)] +
29  2 *dadosIn[calcIndice(x, y+1, z+1, yInc, zInc)] +
30  dadosIn[calcIndice(x+1, y+1, z+1, yInc, zInc)] +
31  2 * dadosIn[calcIndice(x-1, y, z+1, yInc, zInc)] +
32  4 * dadosIn[calcIndice(x, y, z+1, yInc, zInc)] +
33  2 * dadosIn[calcIndice(x+1, y, z+1, yInc, zInc)] +
34  dadosIn[calcIndice(x-1, y-1, z+1, yInc, zInc)] +
35  2 *dadosIn[calcIndice(x, y-1, z+1, yInc, zInc)] +
36  dadosIn[calcIndice(x+1, y-1, z+1, yInc, zInc)])/64);
37
38  } else {
39    dadosOut[indice] = 0;
40  }
41
42 }
```

Listing 4.12: Limiarização e código da máscara do gaussiano.

```
1 //calcular o indice dos elementos nos vetores
2 __device__ int calcIndice(int x, int y, int z, int yInc, int zInc){
3
```

```
4     return (x + (y*yInc))+(z*zInc);
5
6  }
```

Listing 4.13: Método `calcIndice` que calcula o índice de um voxel no vetor de dados unidimensional de acordo com a sua posição no volume selecionado pelo usuário.

Na primeira parte do *kernel* `calc` mostrado no algoritmo 4.14 em que a localização de um voxel no vetor unidimensional de dados, assim como feito no *kernel* `gaussiano`. Em seguida são declaradas as variáveis `gradX`, `gradY` e `gradZ` que terão os valores dos componentes x, y e z respectivamente do vetor gradiente. A matriz Hessiana é representada pela variável `mHessiana`. Os autovalores e autovetores de cada voxel são guardados nas variáveis `eValues` e `eVectors` respectivamente, enquanto que as variáveis `mA`, `mQ`, `mR`, `mS` e `mStemp` são as matrizes A, Q, R, S e uma temporária de S nessa ordem, usadas no método QR.

A variável `valor` é usada para guardar um valor durante o método QR, `cont` é utilizada para uma condição de continuação no método QR. A informação de que um voxel é ou não linha central fica na variável `isCenterline`. O produto entre o vetor gradiente e os autovetor 1 e autovetor 2 é definido nas variáveis `prodV1` e `prodV2` respectivamente.

```
1  __global__ void calc(float *dadosIn, short *dadosOut,
2  int tamanho, int yInc, int zInc, int HtParam,
3  int WtParam, int DtParam){
4
5    //calculo do indice do vetor em relacao a posicao do
6    //voxel no volume tridimensional selecionado pelo usuario
7    int x = blockIdx.x * tamanho + threadIdx.x;
8    int y = ( blockIdx.y - ((yInc/2)*(blockIdx.y/(yInc/2))) )*
9    tamanho + threadIdx.y;
10   int z = ((blockIdx.y/(yInc/2)) * tamanho) + threadIdx.z;
11
12   int indice = (x + (y*yInc))+(z*zInc);
13
14   float gradX, gradY, gradZ;
15
16   //matrizes Hessiana, A, Q, R e S do método QR
17   float mHessiana[3][3], mA[3][3], mQ[3][3], mR[3][3], mS[3][3];
18   //matriz temporaria de S usada no método QR
19   float mStemp[3][3];
20
21   float eValues[3];
22   float eVectors[3][3]; //Eigenvalues e Eigenvectorss
23
24   float valor = 0;
```

```
25   int cont = 0;
26   int isCenterline = 0;
27   float prodV1 = 0;
28   float prodV2 = 0;
```

Listing 4.14: Lista de parâmetros do *kernel* `calc`. Obtenção do índice do voxel e declaração de variáveis usadas no *kernel*.

A parte dois do *kernel* `calc` é vista no algoritmo 4.15. O *if* presente no algoritmo tem as mesmas funcionalidades do *if* existente no *kernel* `gaussiano`. Os componentes do vetor gradiente são calculados usando fórmulas baseadas nas equações 2.5 e 2.6.

```
1  if(x>1 && y>1 && z>1 && x<HtParam-1 && y<WtParam-1 &&
2  z<DtParam-1 && dadosIn[indice]>30 && dadosIn[indice]<400){
3
4    //calcula os gradientes
5    gradX = dadosIn[calcIndice(x+1, y, z, yInc, zInc)] -
6      dadosIn[calcIndice(x, y, z, yInc, zInc)];
7    gradY = dadosIn[calcIndice(x, y+1, z, yInc, zInc)] -
8      dadosIn[calcIndice(x, y, z, yInc, zInc)];
9    gradZ = dadosIn[calcIndice(x, y, z+1, yInc, zInc)] -
10     dadosIn[calcIndice(x, y, z, yInc, zInc)];
```

Listing 4.15: Calculo do gradiente dentro do *kernel* `calc`.

O calculo dos componentes da matriz Hessiana é mostrado no algoritmo 4.16.

```
1    //calcula matriz Hessiana
2
3    //Matriz Hessiana
4    //   0   1   2
5    //0 Dxx Dxy Dxz
6    //1 Dyx Dyy Dyz
7    //2 Dzx Dzy Dzz
8
9    //Dxx
10   mHessiana[0][0] = dadosIn[calcIndice(x+1,y,z, yInc, zInc)] -
11     2 * dadosIn[calcIndice(x,y,z, yInc, zInc)] +
12     dadosIn[calcIndice(x-1,y,z, yInc, zInc)];
13
14   //Dyy
15   mHessiana[1][1] = dadosIn[calcIndice(x,y+1,z, yInc, zInc)] -
16     2 * dadosIn[calcIndice(x,y,z, yInc, zInc)] +
```

```
17      dadosIn[calcIndice(x,y-1,z, yInc, zInc)];
18
19    //Dzz
20    mHessiana[2][2] = dadosIn[calcIndice(x,y,z+1, yInc, zInc)] -
21      2 * dadosIn[calcIndice(x,y,z, yInc, zInc)] +
22      dadosIn[calcIndice(x,y,z-1, yInc, zInc)];
23
24    //Dxy e Dyx
25    mHessiana[0][1] = mHessiana[1][0] =
26      (dadosIn[calcIndice(x-1,y+1,z, yInc, zInc)] -
27      dadosIn[calcIndice(x+1,y+1,z, yInc, zInc)] +
28      dadosIn[calcIndice(x+1,y-1,z, yInc, zInc)] -
29      dadosIn[calcIndice(x-1,y-1,z, yInc, zInc)])/4;
30
31    //Dxz e Dzx
32    mHessiana[0][2] = mHessiana[2][0] =
33      (dadosIn[calcIndice(x-1,y,z+1, yInc, zInc)] -
34      dadosIn[calcIndice(x+1,y,z+1, yInc, zInc)] +
35      dadosIn[calcIndice(x+1,y,z-1, yInc, zInc)] -
36      dadosIn[calcIndice(x-1,y,z-1, yInc, zInc)])/4;
37
38    //Dyz e Dzy
39    mHessiana[1][2] = mHessiana[2][1] =
40      (dadosIn[calcIndice(x,y-1,z+1, yInc, zInc)] -
41      dadosIn[calcIndice(x,y+1,z+1, yInc, zInc)] +
42      dadosIn[calcIndice(x,y+1,z-1, yInc, zInc)] -
43      dadosIn[calcIndice(x,y-1,z-1, yInc, zInc)])/4;
```

Listing 4.16: Calculo da matriz Hessiana.

A implementação do método QR baseado no pseudo código do algoritmo 1 é vista no algoritmo 4.17. A condição cont<15 não permite que o while seja executado mais do que 15 vezes. Este valor foi escolhido devido a média de iterações ter sido 9. No entanto, em alguns casos a quantidade chegou até 15, e dificilmente esse valor era superado. A limitação é necessária para evitar que o algoritmo tenha valores como 0,9999, muito próximo de 1, que é o critério de parada, e sejam feitas iterações desnecessárias.

```
1   //calcula os autovalores e autovetores da matriz hessiana
2
3   //copia os dados para a Matriz A
4   for (int j = 0; j <= 2; j++){
5     for(int i = 0; i <=2; i++){
6       mA[i][j] = mHessiana[i][j];
7     }
```

```
8   }
9
10  //inicializa matriz R
11  for (int i = 0; i <= 2; i++){
12    for(int j = 0; j <=2; j++){
13      mR[i][j] = mS[i][j] = mQ[i][j] = 0;
14    }
15  }
16
17  while(fabs(mQ[0][0]) != 1 && fabs(mQ[1][1]) != 1 &&
18    fabs(mQ[2][2]) != 1 && cont<15){
19
20    cont++;
21
22    //calculo das matrizes Q e R
23    for(int j = 0; j <= 2; j++){
24      valor = 0;
25      for(int t = 0; t <= 2; t++){
26        valor += mA[j][t] * mA[j][t];
27      }
28      mR[j][j] = sqrt(valor);
29
30      if(mR[j][j] == 0){
31        break;
32      } else {
33        for(int i = 0; i <= 2; i++){
34          mA[j][i] = mA[j][i] / mR[j][j];
35        }
36      }
37
38      for(int k = j+1; k <= 2; k++){
39        valor = 0;
40        for(int u = 0; u <= 2; u++){
41          valor += mA[j][u] * mA[k][u];
42        }
43        mR[k][j] = valor;
44          for(int p = 0; p <=2; p++){
45          mA[k][p] = mA[k][p] - (mA[j][p] * mR[k][j]);
46        }
47      }
48    }
49
50    //copiando para a verdadeira matriz Q
51    for (int i = 0; i <= 2; i++){
```

```
52          for(int j = 0; j <=2; j++){
53            mQ[i][j] = mA[i][j];
54            if(cont == 1){
55              mS[i][j] = mA[i][j];
56            }
57          }
58        }
59
60        //calculando a matriz S (caso não seja a primeira iteração)
61        if(cont > 1){
62          for (int i = 0; i <= 2; i++){
63            for(int j = 0; j <=2; j++){
64              mStemp[i][j] = mS[0][j] * mQ[i][0] + mS[1][j] *
65                mQ[i][1] + mS[2][j] * mQ[i][2];
66            }
67          }
68        }
69
70        //copiando para a verdadeira S
71        if(cont > 1){
72          for (int i = 0; i <= 2; i++){
73            for(int j = 0; j <=2; j++){
74              mS[i][j] = mStemp[i][j];
75            }
76          }
77        }
78
79        //nova matriz A
80        for (int i = 0; i <= 2; i++){
81          for(int j = 0; j <=2; j++){
82            mA[i][j] = mR[0][j] * mQ[i][0] + mR[1][j] * mQ[i][1] +
83              mR[2][j] * mQ[i][2];
84          }
85        }
86
87      }
```

Listing 4.17: Implementação do método QR.

A verificação das linhas centrais inicia extraindo-se os autovalores e autovetores respectivamente das matrizes A e S resultantes do método QR. Em seguida os autovalores e seus respectivos autovetores são ordenados como está no algoritmo 4.18.

```
1 //verifica os pontos que são centerline
```

```
2
3   //copia valores para a verificação
4   for(int c = 0; c <= 2; c++){
5     eValues[c] = mA[c][c];
6     for(int d = 0; d <= 2; d++){
7       eVectors[c][d] = mS[d][c];
8     }
9   }
10
11  float temp = 0;
12
13  //ordena os eigens para a avaliação do ponto
14  for(int i = 0; i <= 2; i++){
15    for(int j = i; j <= 2; j++){
16      if(eValues[i] > eValues[j]){
17        temp = eValues[i];
18        eValues[i] = eValues[j];
19        eValues[j] = temp;
20
21        for(int d = 0; d <= 2; d++){
22          temp = eVectors[i][d];
23          eVectors[i][d] = eVectors[j][d];
24          eVectors[j][d] = temp;
25        }
26      }
27    }
28  }
```

Listing 4.18: Preparação para verificar se um voxel é linha central.

O algoritmo 4.19 mostra o restante da verificação, onde primeiro é feita a aproximação dos componentes do vetor gradiente com o valor de `k` igual a 30 e em seguida são testadas as condições de *height ridges* com o valor de r igual a 0,5. No final caso o voxels seja linha central ele mantem o seu valor de intensidade, caso contrario a sua intensidade é zerada. O método `produtoVetores` usado no algoritmo 4.19 é mostrado no algoritmo 4.20 e obtém o produto de dois vetores.

```
1   //aproximacoes do gradiente e dos autovetores
2   //k = 30
3   if(abs(gradX) < 30){ gradX = 0; }
4   if(abs(gradY) < 30){ gradY = 0; }
5   if(abs(gradZ) < 30){ gradZ = 0; }
6
7   //verificação se o ponto é centerline
```

```
8
9   if(eValues[0] < 0 && eValues[1] < 0){
10    prodV1 = produtoVetores(eVectors[0][0], eVectors[0][1],
11      eVectors[0][2], gradX, gradY, gradZ);
12    prodV2 = produtoVetores(eVectors[1][0], eVectors[1][1],
13      eVectors[1][2], gradX, gradY, gradZ);
14
15    if(prodV1 == 0 && prodV2 == 0){
16      //r = 0.5
17      if(eValues[1]/eValues[0] >= 0.5){
18        isCenterline = 1;
19      }
20    }
21  }
22
23  if(isCenterline==1){
24    dadosOut[indice] = dadosIn[indice];
25  } else {
26    dadosOut[indice] = 0;
27  }
28
29 }else{
30   dadosOut[indice] = 0;
31 }
32
33 }
```

Listing 4.19: Verificação se um voxel é linha central.

```
1 __device__ float produtoVetores(float v1x, float v1y, float v1z, floa
2     return v1x * v2x + v1y * v2y + v1z * v2z;
3 }
```

Listing 4.20: Método que calcula o produto de dois vetores.

Após a execução dos *kernels* `gaussiano` e `calc` o tempo de processamento é medido e mostrado. Em seguida os dados são copiados da memória da placa grá?ca(`dadosDevice`) para a variável `tIn` usando a função `cudaMemcpy`, como mostrado no algoritmo 4.21. Após a copia os modelos tridimensionais dos vasos originais e do resultado da segmentação são apresentados ao usuário.

```
1 //mostra o tempo total de execucao dos kernels
```

```
2  gpuTime = cutGetTimerValue (timer );
3  tempoTotal += gpuTime;
4  cout << "Executa em: " << gpuTime << endl;
5  cout << "Tempo total: " << tempoTotal << endl;
6  cutStopTimer (timer );
7
8  //copia os resultados para uma variavel dos host
9  cudaMemcpy (tIn , dadosDevice , sizeof(short)*Ht*Wt*Dt ,
10  cudaMemcpyDeviceToHost )
```

Listing 4.21: Método que calcula o produto de dois vetores.

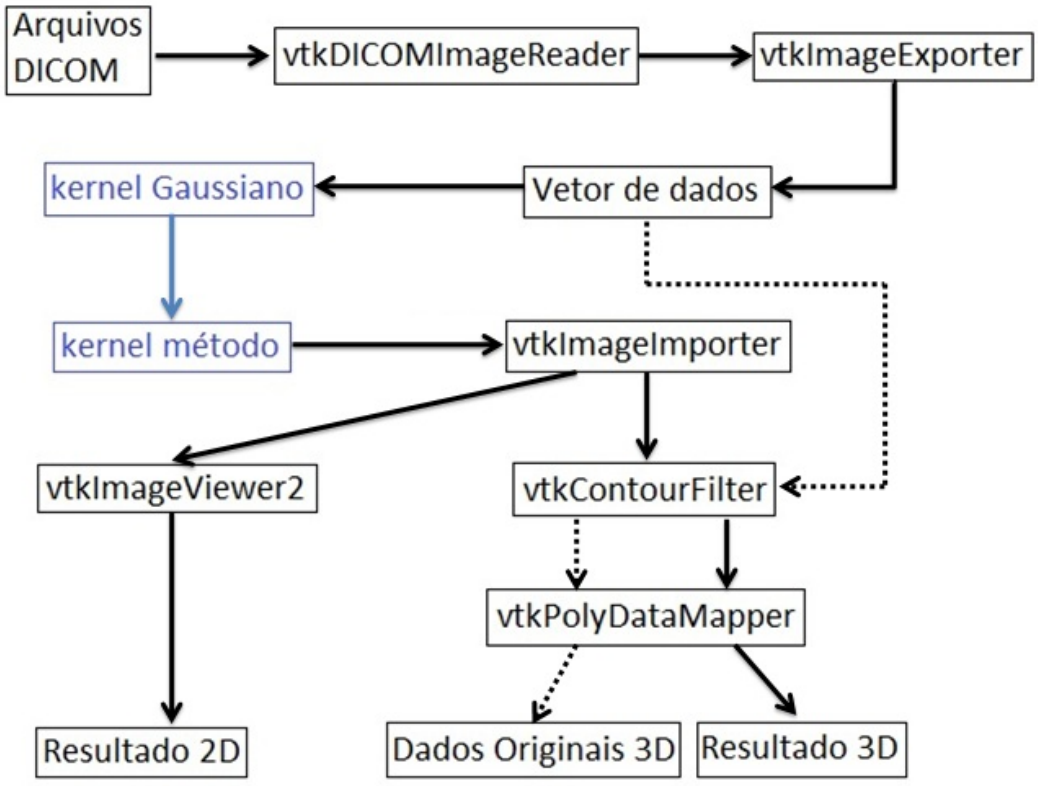


Figura 4.6: Organização das classes da VTK e dos *kernels* de CUDA.

O programa dos testes foi executado em um computador com uma CPU Intel Pentium Core 2 2.66GHZ, 2GB de memória RAM, Windows XP 32 bits e uma placa gráfica NVIDIA GeForce 8800 GTS com 512MB de RAM.

4.2 Método de análise

O método é testado no conjunto de imagens artificiais, que simulam as características de um vaso para assim identificar, se o método proposto obtém as linhas centrais quando os dados são favoráveis. Dados reais de imagens médicas de TC são utilizadas para analisar o desempenho do método proposto na determinação das *centerlines* em condições mais realísticas.

A variação no valor das variáveis independentes `k`, `t` e `r` são avaliadas para se verificar a mudança na quantidade de pontos selecionados como linhas centrais, e uma possível melhora na qualidade nos resultados. Os valores de `k` são 35, 50 e 100 para os dados artificiais e 100 e 50 para as imagens médicas. A escolha destes valores para os dados médicos foi realizada devido a 35 ser um valor mínimo para que apareçam as linhas centrais no cilindro de raio 3 e os parâmetros 50 e 100 afim de analisar a mudança no valor de `k`. Para os dados médicos os valores 50 e 100 foram escolhidos por serem semelhantes aos parâmetros dos dados artificiais. Valores menores que 50 resultavam em poucos voxels segmentados como linhas centrais. A mudança nos valores de `t` foram apenas 0,1 e 0,3 pois se desejava que esta variável tivesse valores os mais próximos possíveis de zero para a visualização dos resultados. Em `r` os valores foram iguais a 0,2, 0,5 e 0,8 para se estudar grandes mudanças no parâmetro desta variável, visto que no trabalho original de Aylward o valor de `r` é fixo em 0,5.

As variáveis dependentes da análise são a velocidade de execução e a quantidade de voxels resultantes da segmentação. Para analisar a velocidade do método proposto, o tempo de segmentação de um determinado volume selecionado pelo usuário é comparado, executando os dados no método deste trabalho em GPU e em CPU. O trabalho de Aylward é executado em CPU. A quantidade de voxels processados por milissegundo (voxels/ms) também é medida para se comparar a velocidade do método deste trabalho em GPU, em CPU e no algoritmo de Aylward. O tempo de execução é medido com funções de CUDA, por terem uma maior precisão em relação as funções padrões de C++. Os tempos em GPU e CPU eram muitos próximos ou idênticos para cada combinação dos valores de `k`, `t` e `r`, quando eram executados várias vezes. Com isto, os valores obtidos nos resultados foram anotados após uma única execução. A quantidade de voxels resultantes de cada segmentação é contabilizada.

Capítulo 5

Resultados

Neste capítulo são apresentados as imagens resultantes da segmentação, o tempo de execução e a quantidade de voxels processados por milissegundo do método proposto. Além disso é feita uma discussão dos resultados obtidos.

Nas imagens artificiais, o resultado da segmentação é visto nas Figuras 5.1, 5.2 e 5.3. Nas figuras, as paredes dos cilindros dos dados artificiais estão em vermelho, os voxels segmentados como *centerlines* estão em branco e a quantidade de voxels segmentados como *centerlines* (QVC) em cada segmentação. A quantidade de voxels processados que se encontravam dentro dos limites da limiarização é igual a 7992. A visão do topo dos cilindros após a segmentação para os três valores de $\mathtt{k}$ é muito semelhante como é visto na Figura 5.4

Uma das fatias do volume de imagens médicas esta na Figura 5.5a. O modelo tridimensional da região selecionada pelo usuário é visto na Figura 5.5b. Este volume selecionado pelo usuário tem dimensões $150 \times 150 \times 50$, totalizando 1.125.000 voxels.

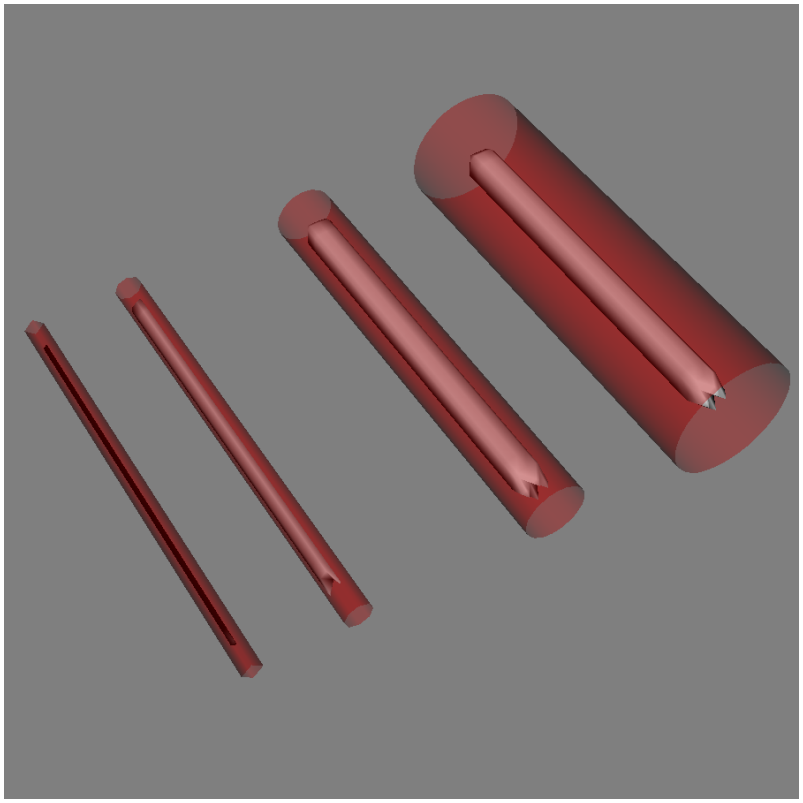
Os resultados da segmentação da região selecionada com diferentes valores de referência para aproximação, são vistos nas Figuras 5.6 e 5.7. A quantidade de voxels processados que estavam dentro dos limites da limiarização aplicada foi de 40433 voxels.

Os tempos médios de cinco execuções para segmentação do volume selecionado pelo usuário das imagens médicas em GPU estão na Tabela 5.1. Em CPU o tempo de execução para o mesmo volume selecionado pelo usuário, com $150 \times 150 \times 50$ voxels é de 23,556 segundos em média, onde foram processados 371479 voxels, sendo que 86980 voxels foram selecionados como linhas centrais. Assim em GPU são processados 87,32 voxels/ms e em CPU são 15,77 voxels/ms. O algoritmo original processou 0,1 voxels/ms. As velocidades são vistas na Tabela 5.2.

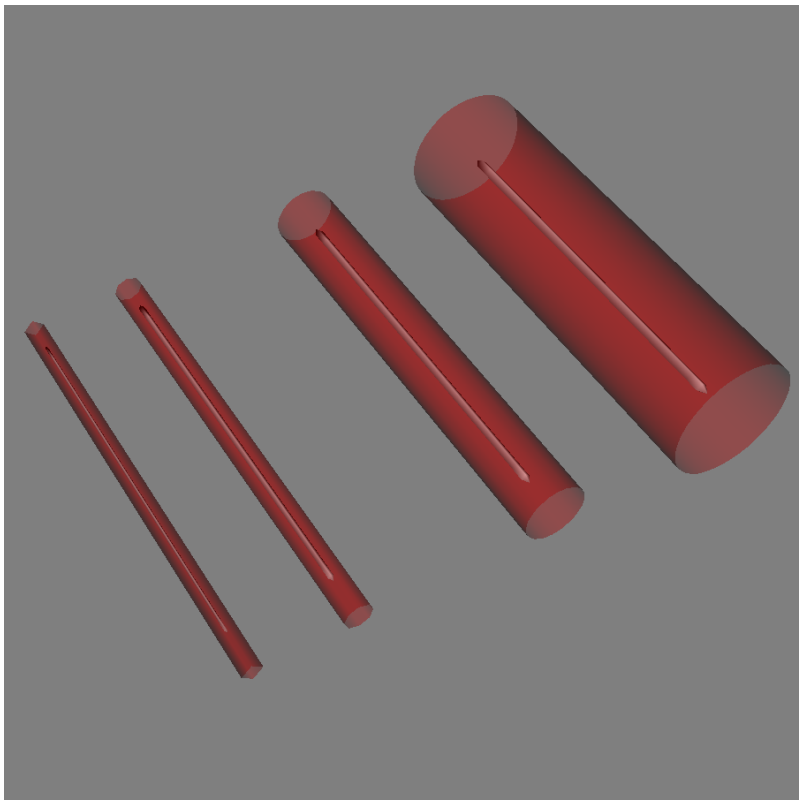
	$\mathtt{k} = 10$	$\mathtt{k} = 20$
$\mathtt{r} = 0,2$	0,463	0,463
$\mathtt{r} = 0,5$	0,467	0,464

Tabela 5.1: Tempos de execução em segundos de segmentação das imagens.

O volume máximo de dados selecionado pelo usuário é limitado pela quantidade de memória da placa de vídeo e pela quantidade máxima de iterações na decomposição QR. Na NVIDIA GeForce 8800 GTS com 512MB de memória o tamanho máximo do volume

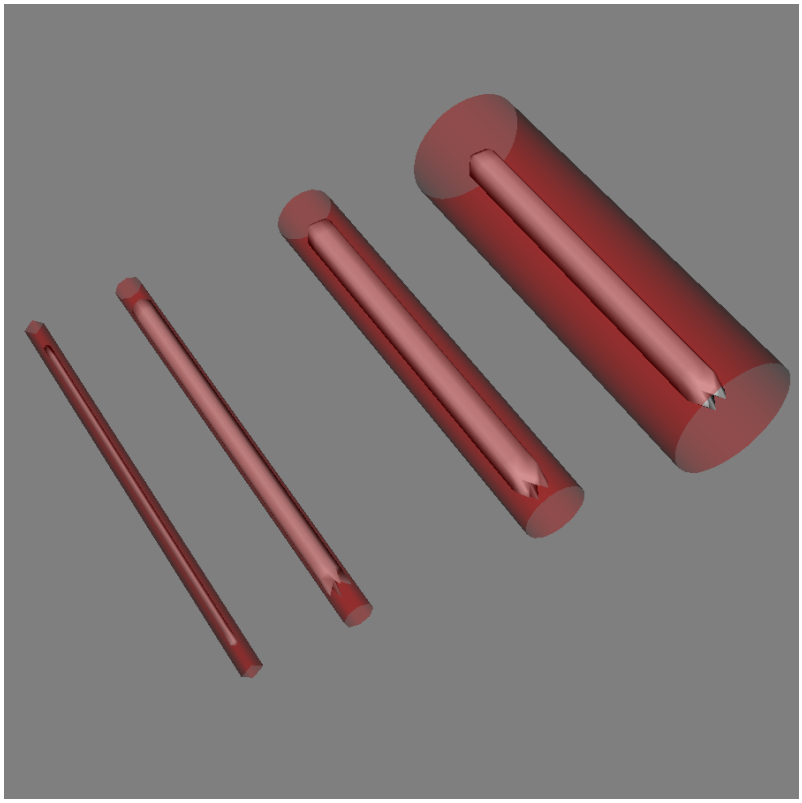


(a) $\mathtt{k} = 25, \quad \mathtt{r} = 0,5, \quad$ QVC = 738.

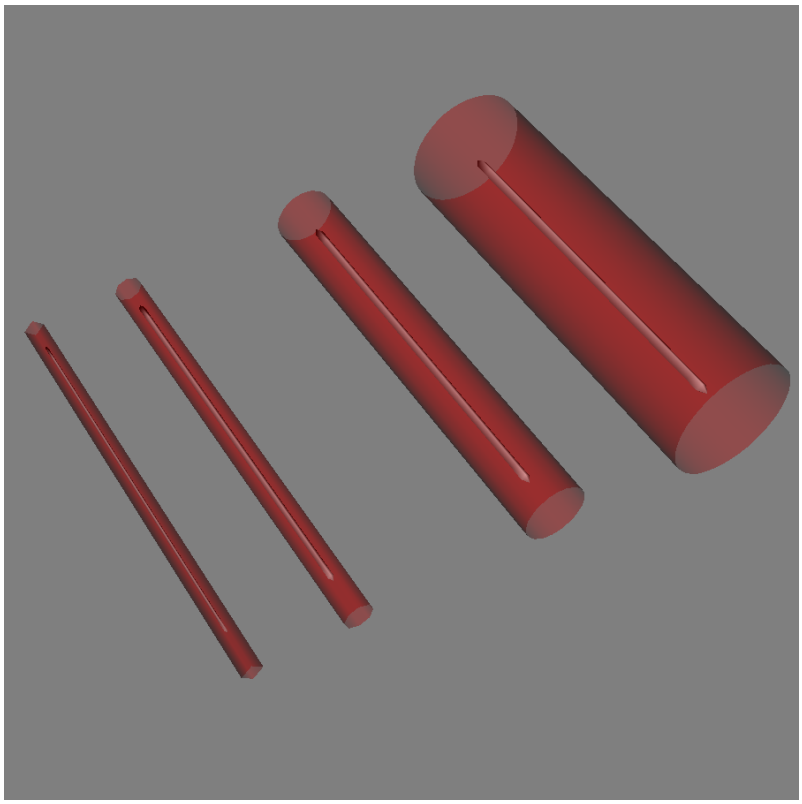


(b) $\mathtt{k} = 25, \quad \mathtt{r} = 0,8, \quad$ QVC = 100.

Figura 5.1: Resultados da segmentação dos dados artificiais para os seguintes valores da variáveis analisadas: (a) $\mathtt{k} = 25, \quad \mathtt{r} = 0,5, \quad$ QVC = 738 e (b) $\mathtt{k} = 25, \quad \mathtt{r} = 0,8, \quad$ QVC = 100.

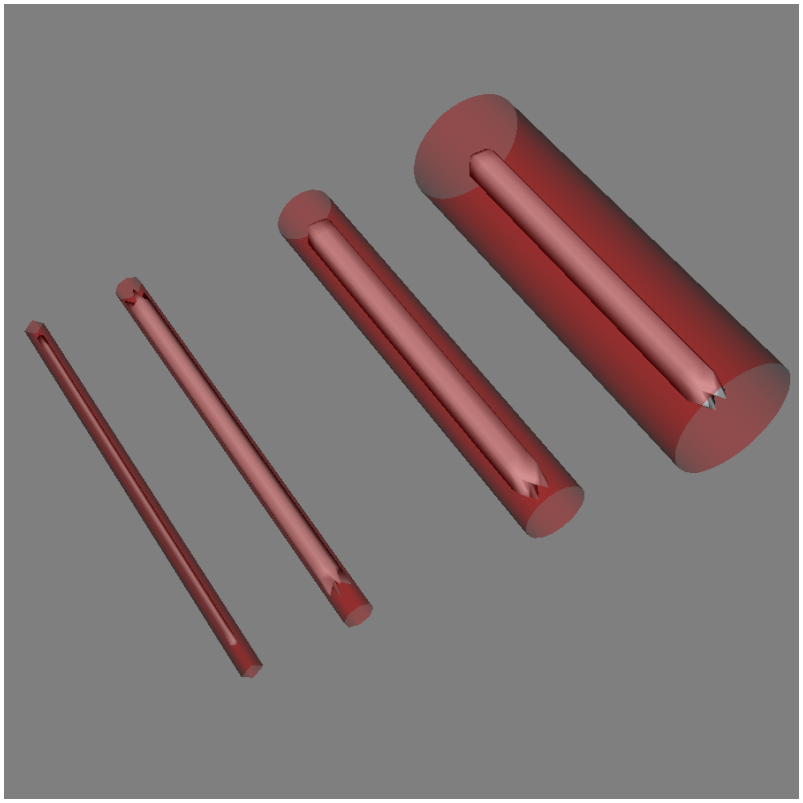


(a) $\mathtt{k} = 50, \quad \mathtt{r} = 0,5, \quad$ QVC = 921.

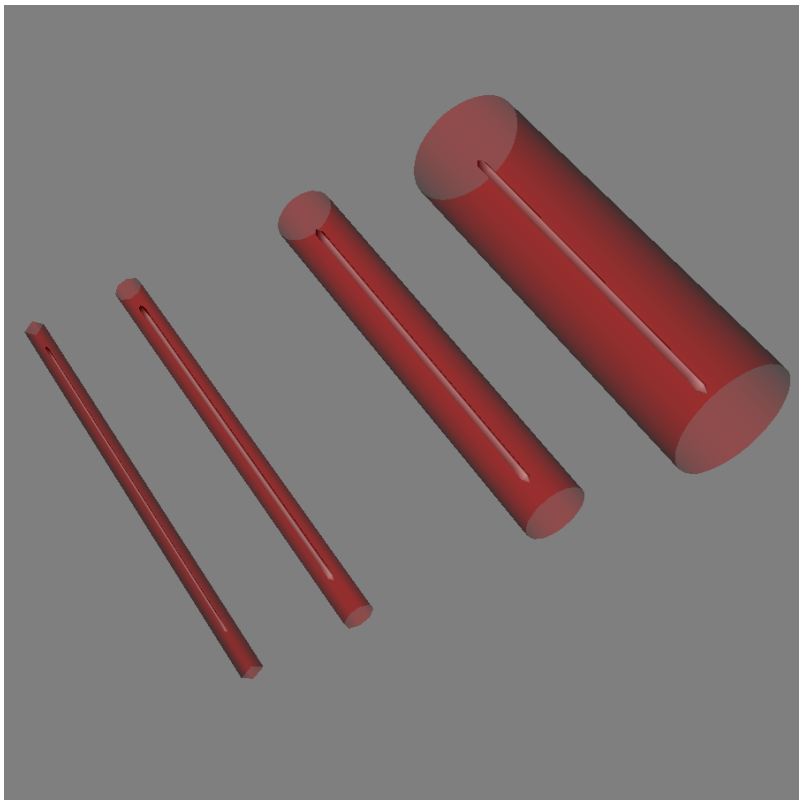


(b) $\mathtt{k} = 50, \quad \mathtt{r} = 0,8, \quad$ QVC = 100.

Figura 5.2: Resultados da segmentação dos dados artificiais para os seguintes valores da variáveis analisadas: (a) $\mathtt{k} = 50, \quad \mathtt{r} = 0,5, \quad$ QVC = 921 e (b) $\mathtt{k} = 50, \quad \mathtt{r} = 0,8, \quad$ QVC = 100.



(a) $\mathtt{k} = 75, \quad \mathtt{r} = 0,5, \quad$ QVC = 926.



(b) $\mathtt{k} = 75, \quad \mathtt{r} = 0,8, \quad$ QVC = 100.

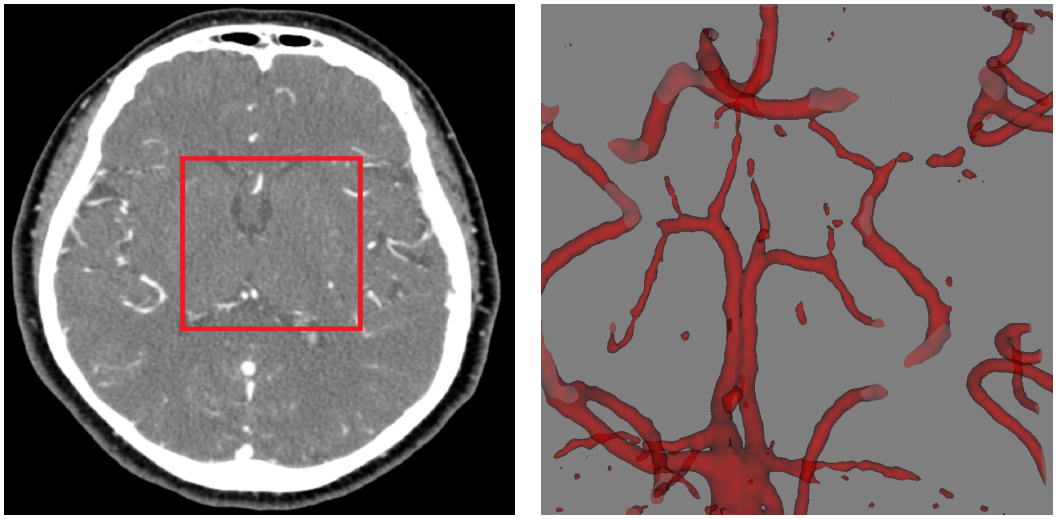
Figura 5.3: Resultados da segmentação dos dados artificiais para os seguintes valores da variáveis analisadas: (a) $\mathtt{k} = 75, \quad \mathtt{r} = 0,5, \quad$ QVC = 926 e (b) $\mathtt{k} = 75, \quad \mathtt{r} = 0,8, \quad$ QVC = 100.

(a) k = 25

(b) k = 50

(c) k = 75

Figura 5.4: Visão do topo dos cilindros após a segmentação para: (a) $\mathtt{k} = 25$, (b) $\mathtt{k} = 50$ e (c) $\mathtt{k} = 75$.



(a) Imagem original (b) Região selecionada

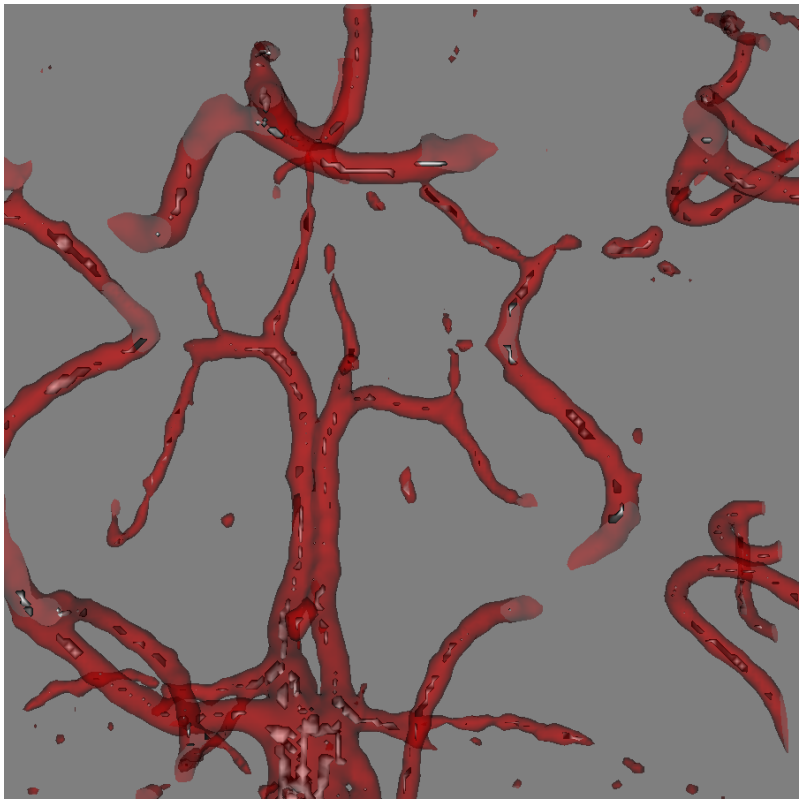
Figura 5.5: (a) Imagem DICOM com a região selecionada pelo usuário nesta fatia do volume. (b) Modelo tridimensional da região selecionada pelo usuário para segmentação.

Método	Velocidade
Proposto em GPU	87,32
Proposto em CPU	15,77
Original em CPU	0,1

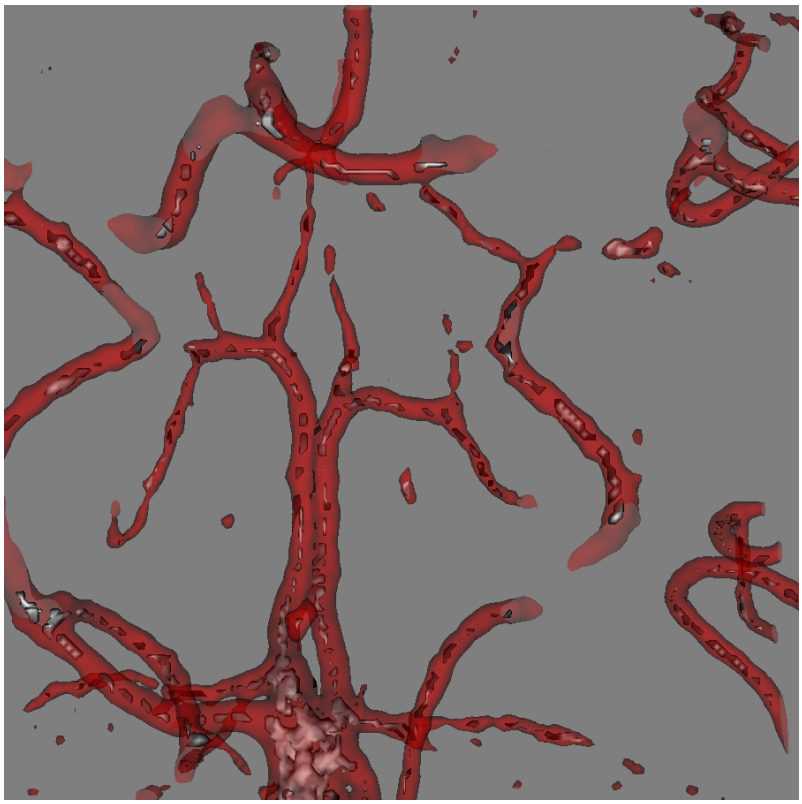
Tabela 5.2: Velocidade de execução em voxels/milissegundo da segmentação das imagens.

foi de $350 \times 350 \times 100$ (12.250.000 voxels), onde foram processados 1.327.789 voxels que estavam dentro dos limites da limiarização e 78.244 voxels foram segmentados como linhas centrais. O tempo médio de execução do volume máximo foi de 7,2718 segundos, ou seja, 182 voxels/ms.

Em todas as segmentações a quantidade máxima de iterações da decomposição QR foi 15. Este valor foi escolhido devido a média de iterações ter sido 9. No entanto, em alguns casos a quantidade chegou até 15, e dificilmente esse valor era superado. A limitação é necessária para evitar que o algoritmo fique com valores como 0,9999, muito próximo de 1, que é o critério de parada, e sejam feitas iterações desnecessárias. Quanto maior o valor máximo de iterações maior será o tempo de execução do algoritmo, o qual não deve ser muito longo, pois pode causar um fechamento forçado por parte do sistema operacional. Este é um problema particular do Windows. No sistema operacional Windows, caso um programa executado em GPU ultrapasse um tempo pré-definido para atualização do vídeo, o programa é abortado. Este valor pode ser alterado no registro do Windows localizado em HKEY_LOCAL_MACHINE/System/CurrentControlSet/Control/GraphicsDrivers criando ou modificando as chaves TdrLevel e TdrDelay. Nessa última chave é atribuído o tempo

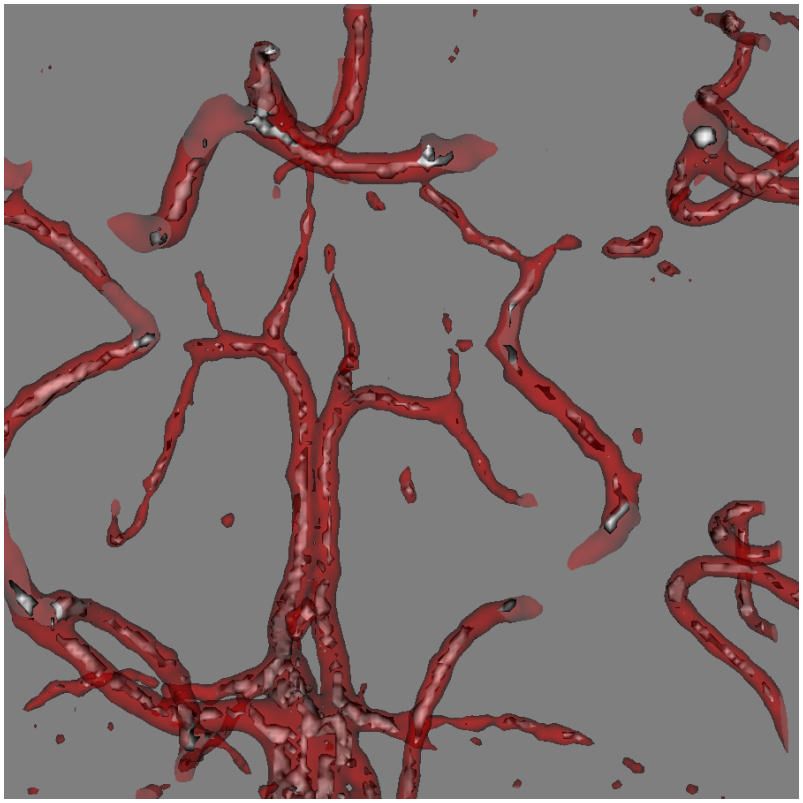


(a) $\mathtt{k} = 10, \quad \mathtt{r} = 0,5, \quad$ QVC = 4820.

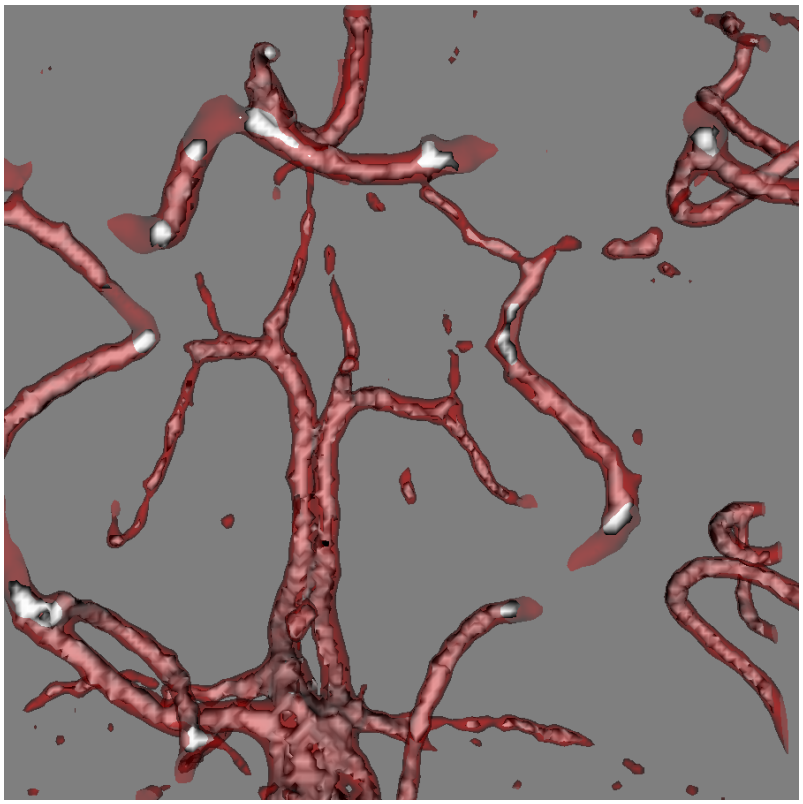


(b) $\mathtt{k} = 10, \quad \mathtt{r} = 0,2, \quad$ QVC = 9591.

Figura 5.6: Resultados da segmentação do volume selecionado pelo usuário para os seguintes valores da variáveis analisadas: (a) $\mathtt{k} = 10, \quad \mathtt{r} = 0,5, \quad$ QVC = 4820 e (b) $\mathtt{k} = 10, \quad \mathtt{r} = 0,2, \quad$ QVC = 9591.



(a) $\mathtt{k} = 20, \quad \mathtt{r} = 0,5, \quad$ QVC = 8848.



(b) $\mathtt{k} = 20, \quad \mathtt{r} = 0,2, \quad$ QVC = 19296.

Figura 5.7: Resultados da segmentação do volume selecionado pelo usuário para os seguintes valores da variáveis analisadas: (a) $\mathtt{k} = 20, \quad \mathtt{r} = 0,5, \quad$ QVC = 8848 e (b) $\mathtt{k} = 20, \quad \mathtt{r} = 0,2, \quad$ QVC = 19296.

máximo de espera para atualização do vídeo, ou seja, o tempo máximo de execução de um programa em GPU.

Discussão dos Resultados

A segmentação confirma que, quanto mais próximo de zero é o valor de `k`, menor é a quantidade de pontos selecionados, isto ocorre devido os valores obtidos com os cálculos usados serem aproximados. O valor de `r` também influencia, em menor escala, na quantidade de pontos, onde quanto maior o valor de `r` menos pontos são segmentados e quanto menor o valor de `r` mais pontos são selecionados como linhas centrais.

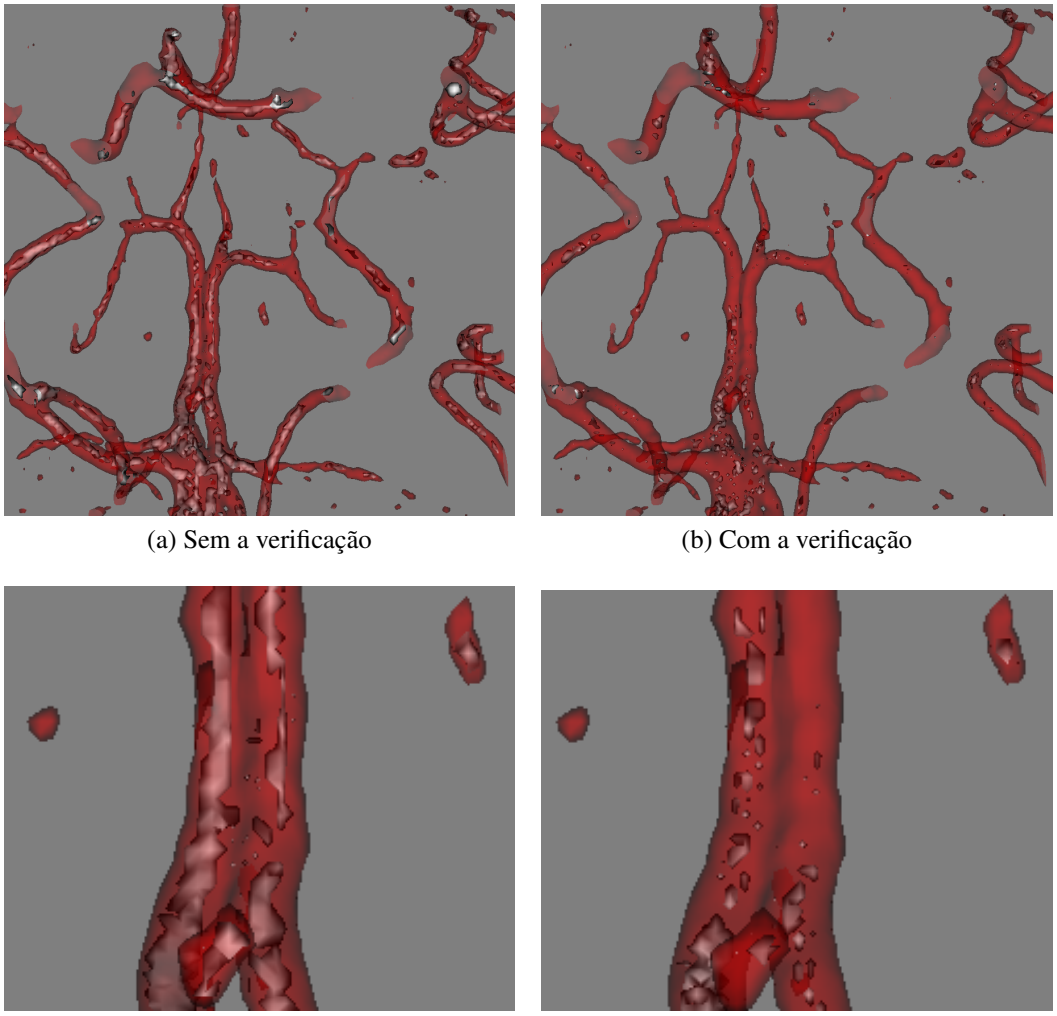
Nos resultados da segmentação dos dados artificiais se observa que para `r`=0,5 a diminuição do valor de `k` reduz o diâmetro das linhas centrais. E para `r`=0,8 mesmo com a variação de `k`, as linhas centrais são segmentadas corretamente com o diâmetro de um voxel. Isto pode ser comprovado pela quantidade de voxels segmentados como linhas centrais que foi igual a 100, e é sempre o mesmo para `r`=0,8. A segmentação de `r`=0,8 pode ser vista tridimensionalmente na Figura 5.3 e bidimensionalmente na Figura 5.4c. A diferença de intensidade dos voxels marcados como *centerlines* na Figura 5.4c pode dar a impressão que as *centerlines* tem diâmetros diferentes, no então elas têm um único voxel de diâmetro. Valores de `r`<0,5 geravam segmentação com linhas centrais mais grossas, então foram descartados.

Na segmentação das imagens médicas com `r`=0,2 as linhas centrais estão mais contínuas e com poucas falhas, em relação a `r`=0,5. O aumento no valor de `k` resulta em mais pontos segmentados como linhas centrais, embora `r` contribua mais para este aumento.

As figuras da segmentação de imagens médicas mostram também que não foram somente os pontos que estão no centro dos vasos que foram segmentados, isto ocorreu devido a mudança no algoritmo original. Na versão original, após um ponto ser marcado para *centerline* era feita uma busca por novos pontos próximos para compor a linha central. No entanto, em muitos casos, um ponto obedecia as condições de central, mas não era encontrado nenhum ponto próximo para ser ligado ao primeiro, sendo assim um único ponto não formava uma linha central. Na versão em paralelo proposta neste trabalho não é feita essa busca após um ponto ser encontrado, mas sim a verificação de cada ponto de maneira independente. Isto pode ocasionar a classificação pontos como *centerline* em alguns vasos da imagem que não se conectam a outros pontos formando uma linha entre si.

A fim de remover os pontos segmentados como *centerlines*, mas que não formam uma linha entre si e diminuir a espessura das linhas centrais, para terem o diâmetro de apenas um voxel, realizou-se uma verificação após a segmentação. Esta verificação consistia em executar um *kernel* sobre cada voxel do volume e usar a direção direta e a direção inversa do vetor tangente ao voxel processado, para então analisar se os dois voxels apontados pela direção direta e inversa do vetor tangente são *centerlines*. Em caso afirmativo para qualquer uma das duas direções, o voxel verificado era pertencente a linha central do vaso, caso contrário ele não se conectava a outro voxel do volume e assim o voxel processado não era mais considerado como parte da linha central. No entanto, após essa verificação foram removidos muitos voxels do resultado e as linhas centrais tornaram-se mais des-

contínuas e por estes decidiu-se remover esta etapa do método. Na Figura 5.8 o resultado da segmentação antes e depois da verificação é visto respectivamente nas imagens 5.8a e 5.8b. Nas Figuras 5.8c uma região sem a verificação é destacada e na Figura 5.8d esta mesma região é apresentada após a verificação



(a) Sem a verificação (b) Com a verificação

(c) Destaque sem a verificação (d) Destaque com a verificação

Figura 5.8: Resultado da segmentação das linhas centrais (a) sem a verificação e (b) após a verificação. Em (c) uma região sem a verificação é destacada e em (d) esta mesma região é apresentada após a verificação.

Capítulo 6

Conclusão

A segmentação da estrutura vascular é importante para ajudar no diagnóstico de várias doenças vasculares. A determinação das linhas centrais é útil para a construção e visualização da estrutura vascular e seus problemas. Existem vários métodos para a obtenção das linhas centrais, onde é necessário em alguns, muitos passos para alcançar o objetivo, causando uma maior demora na obtenção dos resultados. Um outro problema de alguns métodos é a seleção da referência inicial do algoritmo por parte do usuário.

Este trabalho apresentação uma adaptação do trabalho de Aylward para tentar solucionar os problemas antes citados. Com relação a diminuição do tempo para alcançar as linhas centrais, procurou-se criar um método de segmentação paralelo, utilizando o grande poder de processamento paralelo da GPU, através de CUDA. O ganho do método proposto foi 873 vezes mais rápido sendo executado em GPU e 150 vezes mais rápido sendo executado em CPU do que o original de Aylward em CPU.

Para facilitar a seleção pelo usuário, é solicitado que uma região ao redor do objeto desejado seja indicada para segmentação e não pontos difíceis de serem selecionados. Após remover os voxels que não são pertencentes a vasos, ocorre o processados em paralelo dos voxels pertencentes a vasos e aqueles que pertencem a linha central de um vaso são segmentados, ou seja, pontos pertencentes a bifurcações são analisados sem a necessidade de outras verificações específicas e as linhas centrais são ligadas naturalmente.

Trabalhos Futuros

Em termos de trabalhos futuros ainda existem melhorias a serem feitas no método neste trabalho, como gerar linhas centrais com diâmetro de um único voxel de diâmetro para imagens médicas. Será realizado uma análise qualitativa dos resultados de segmentação obtidos comparando-os, por exemplo, com outas técnicas de segmentação aplicadas ao mesmo conjunto de dados médicos. Além disso, seria importante obter uma avaliação por parte de médicos especialistas quanto aos resultados obtidos, de maneira a identificar se, de fato, a segmentação pode ser considerada satisfatória.

Por fim, devido a grande velocidade do método, uma implementação mais iterativa com um processamento subvoxel, poderá ser refinada com uma rápida reposta, com mudanças feita nos parâmetros das variáveis k e r por um usuário.

Referências Bibliográficas

Abeysinghe, Sasakthi & Tao Ju (2009), 'Interactive skeletonization of intensity volumes', *The Visual Computer* **25**, 627–635. 10.1007/s00371-009-0325-5.

Anton, Howard A. (2004), *Elementary Linear Algebra*, 9ª edição, John Wiley & Sons.

Aylward, S.R. & E. Bullitt (2002), 'Initialization, noise, singularities, and scale in height ridge traversal for tubular object centerline extraction', *Medical Imaging, IEEE Transactions on* **21**(2), 61 –75.

Aylward, Stephen R., Julien Jomier, Jean-Philippe Guyon & Susan Weeks (2002), Intra-operative 3d ultrasound augmentation, *em* 'In: IEEE International Symposium on Biomedical Imaging', pp. 421–424.

Aylward, Stephen, Stephen Pizer, David Eberly & Elizabeth Bullitt (1996), Intensity ridge and widths for tubular object segmentation and description, *em* 'Proceedings of the 1996 Workshop on Mathematical Methods in Biomedical Image Analysis (MMBIA '96)', MMBIA '96, IEEE Computer Society, Washington, DC, USA, pp. 131–138.

Babin, D., E. Vansteenkiste, A. Pizurica & W. Philips (2009), Segmentation and length measurement of the abdominal blood vessels in 3-D MRI images, *em* 'Engineering in Medicine and Biology Society (EMBC), 2009. EMBC 2009. Annual International Conference of the IEEE', pp. 4399 –4402.

Bansal, M., S. Kuthirummal, J. Eledath, H. Sawhney & R. Stone (2010), Automatic blood vessel localization in small field of view eye images, *em* 'Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE', pp. 5644–5648.

Bitter, I., A.E. Kaufman & M. Sato (2001), 'Penalized-distance volumetric skeleton algorithm', *Visualization and Computer Graphics, IEEE Transactions on* **7**(3), 195 –206.

Can, A., C.V. Stewart, B. Roysam & H.L. Tanenbaum (2002), 'A feature-based, robust, hierarchical algorithm for registering pairs of images of the curved human retina', *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **24**(3), 347–364.

Casciaro, M.E., D. Craiem, S. Graf, E.P. Gurfinkel & R.L. Armentano (2010), Estimation of coronary length-volume allometric relations of human arteries invivo using CT, *em* 'Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE', pp. 5716 –5719.

Chiu, Yun-Jen, Van-Truong Pham, Thi-Thao Tran & Kuo-Kai Shyu (2010), Evaluation of active contour on medical inhomogeneous image segmentation, *em* 'Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on', Vol. 1, pp. 311 –314.

Committee, ACR-NEMaA (1985), 'Digital imaging and communications standard'.

Committee, ACR-NEMaA (1988), 'Digital imaging and communications standard: Version 2.0'.

Committee, ACR-NEMaA (1993), 'Digital imaging and communications in medicine (DICOM): Version 3.0'.

Cubero, Carla (2007), Arquitetura de centros de diagnósticos: O Caso de um Centro de Bioimagem, Relatório técnico, Universidade Federal da Bahia. p 11.

Derraz, F., M. Beladgham & M. Khelif (2004), Application of active contour models in medical image segmentation, *em* 'Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on', Vol. 2, pp. 675 – 681.

Dougherty, Geoff (2009), *Digital Image Processing for Medical Applications*, 1ª edição, Cambridge University Press, New York.

Eberly, David (1996), *Ridges in image and data analysis*, The Netherlands: Kluwer Academic.

Eberly, David, Robert Gardner, Bryan Morse, Stephen Pizer & Christine Scharlach (1994), Ridges for image analysis, Relatório técnico, Chapel Hill, NC, USA.

Egger, J., Z. Mostarkic, S. Grosskopf & B. Freisleben (2007), A fast vessel centerline extraction algorithm for catheter simulation, *em* 'Computer-Based Medical Systems, 2007. CBMS '07. Twentieth IEEE International Symposium on', pp. 177 –182.

Fernando, Randima (2004), *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Addison-Wesley Professional.

Frangi, A.F., W.J. Niessen, R.M. Hoogeveen, T. van Walsum & M.A. Viergever (1999), 'Model-based quantitation of 3-D magnetic resonance angiographic images', *Medical Imaging, IEEE Transactions on* **18**(10), 946 –956.

Gao, Weixiao & Jingmiao Zhang (2009), Estimation of arteriosclerosis based on fuzzy support vector machine, *em* 'Computer Network and Multimedia Technology, 2009. CNMT 2009. International Symposium on', pp. 1–4.

Gerig, Guido, Thomas Koller, Gábor Székely, Christian Brechbühler & Olaf Kübler (1993), Symbolic description of 3-D structures applied to cerebral vessel tree obtained from MR angiography volume data, *em* 'Proceedings of the 13th International Conference on Information Processing in Medical Imaging', IPMI '93, pp. 94–111.

Gonzalez, Rafael C. & Richard E. Woods (2002), *Digital Image Processing (2nd Edition)*, Prentice Hall.

Gonzalez, Rafael & Richard E. Woods (2000), *Processamento de Imagens Digitais*, Edgar Blucher.

Heneghan, Conor, John Flynn, Michael O'Keefe & Mark Cahill (2002), 'Characterization of changes in blood vessel width and tortuosity in retinopathy of prematurity using image analysis', *Medical Image Analysis* **6**(4), 407 – 429.

Herman, Gabor T. (2010), *Fundamentals of Computerized Tomography: Image Reconstruction from Projections*, 2ª edição, Springer London.

Hongmei, Zhang, Bian Zhengzhong, Jiang Dazong, Yuan Zejian & Ye Min (2003), Level set method for pulmonary vessels extraction, *em* 'Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on', Vol. 3, pp. II – 1105–8.

Hoover, A.D., V. Kouznetsova & M. Goldbaum (2000), 'Locating blood vessels in retinal images by piecewise threshold probing of a matched filter response', *Medical Imaging, IEEE Transactions on* **19**(3), 203 –210.

Hounsfield, Godfrey (1975), *Method of and apparatus for examining a body by radiation such as X or gamma radiation*. Patent number 3919552.

Hsieh, Jiang (2009), *Computed Tomography: Principles, Design, Artifacts, and Recent Advances*, 2ª edição, Wiley.

Jiang, Xiaoyi & D. Mojon (2003), 'Adaptive local thresholding by verification-based multithreshold probing with application to vessel detection in retinal images', *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **25**(1), 131 –137.

John Kessenich, Dave Baldwin, Randi Rost (2010), *The OpenGL® Shading Language*, The Khronos Group Inc.

Kitware, Inc. (2006), *VTK User's Guide*, 5ª edição, Kitware, Inc.

Koozekanani, D., K.L. Boyer & C. Roberts (2003), 'Tracking the optic nervehead in OCT video using dual eigenspaces and an adaptive vascular distribution model', *Medical Imaging, IEEE Transactions on* **22**(12), 1519–1536.

Kudelski, D., J.-L. Mari & S. Viseur (2010), 3D feature line detection based on vertex labeling and 2D skeletonization, *em* 'Shape Modeling International Conference (SMI), 2010', pp. 246 –250.

Kwon, Jun-Sik, Jun-Woong Gi & Eung-Kwan Kang (2001), An enhanced thinning algorithm using parallel processing, *em* 'Proceedings In International Conference on Image Processing', Vol. 3, pp. 752 –755.

Li, Qin, J. You, Lei Zhang, D. Zhang & P. Bhattacharya (2006), A new approach to automated retinal vessel segmentation using multiscale analysis, *em* 'Pattern Recognition, 2006. ICPR 2006. 18th International Conference on', Vol. 4, pp. 77 –80.

Linguraru, Marius George, Babak J. Orandi, Robert L. Van Uitert, Nisha Mukherjee, Ronald M. Summers, Mark T. Gladwin, Roberto F. Machado & Bradford J. Wood (2008), Ct and image processing non-invasive indicators of sickle cell secondary pulmonary hypertension, *em* 'Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE', pp. 859 –862.

Lopes, Bruno Cardoso & Rodolfo Jardim de Azevedo (2008), Computação de alto desempenho utilizando cuda. p. 42.

Lopez, A.M., F. Lumbreras, J. Serrat & J.J. Villanueva (1999), 'Evaluation of methods for ridge and valley detection', *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **21**(4), 327 –335.

Lorigo, L.M., O. Faugeras, W.E.L. Grimson, R. Keriven, R. Kikinis, A. Nabavi & C.-F. Westin (2000), Codimension-two geodesic active contours for the segmentation of tubular structures, *em* 'Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on', Vol. 1, pp. 444 –451.

Martínez-Perez, M.E., A.D. Highes, A.V. Stanton, S.A. Thorn, N. Chapman, A.A. Bharath & K.H. Parker (2002), 'Retinal vascular tree morphology: A semi-automatic quantification', *Biomedical Engineering, IEEE Transactions on* **49**(8), 912 –917.

Martínez-Pérez, M. Elena, Alun D. Hughes, Alice V. Stanton, Simon A. Thom, Anil A. Bharath & Kim H. Parker (1999), Retinal blood vessel segmentation by means of scale-space analysis and region growing, *em* 'Proceedings of the Second International Conference on Medical Image Computing and Computer-Assisted Intervention', MICCAI '99, pp. 90–97.

Matt Pharr, Randima Fernando (2005), *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley Professional.

McCormick, B.H., T.A. DeFanti & M.D. Brown (1987), *Visualization In Scientific Computing*, Vol. 21, ACM SIGGRAPH.

McInemey, T. & D. Terzopoulos (1999), 'Topology adaptive deformable surfaces for medical image volume segmentation', *Medical Imaging, IEEE Transactions on* **18**(10), 840 –850.

Mendonca, A.M. & A. Campilho (2006), 'Segmentation of retinal blood vessels by combining the detection of centerlines and morphological reconstruction', *Medical Imaging, IEEE Transactions on* **25**(9), 1200 –1213.

Miles, F.P. & A.L. Nuttall (1993), 'Matched filter estimation of serial blood vessel diameters from video images', *Medical Imaging, IEEE Transactions on* **12**(2), 147 –152.

NVIDIA (2010), *NVIDIA CUDA C Programming Guide*, 3.2ª edição, NVIDIA.

Olver, Peter J. (2010), Orthogonal bases and the QR algorithm.

Paiva, Anselmo C., Roberto de Beauclair Seixas & Marcelo Gattass (1999), Introdução a visualização volumétrica, Monografia em Ciência da Computação 03/99, Pontíficia Universidade Católica do Rio de Janeiro - PUC-Rio.

Pock, Thomas, Christian Janko, Reinhard Beichel & Horst Bischof (2005), Multiscale medialness for robust segmentation of 3D tubular structures, *em* 'in Proceedings of the Computer Vision Winter Workshop 2005', pp. 93–102.

Pradabpet, C., N. Ravinu, S. Chivapreecha, B. Knobnob & K. Dejhan (2009), An efficient filter structure for multiplierless sobel edge detection, *em* 'Innovative Technologies in Intelligent Systems and Industrial Applications, 2009. CITISIA 2009', pp. 40 –44.

Pratt, William K. (2007), *Digital Image Processing*, 4ª edição, Wiley-Interscience.

Preim, Bernhard & Dirk Bartz (2007), *Visualization in Medicine: Theory, Algorithms, and Applications (The Morgan Kaufmann Series in Computer Graphics)*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Radha, V. & M. Krishnaveni (2009), Threshold based segmentation using median filter for sign language recognition system, *em* 'Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on', pp. 1394 –1399.

Rahman, A., S. Selmi, C. Papadopoulos & G. Papaioannou (2009), CT-scan based FEA for the assessment of the effect of bone density on femur's fracture, *em* 'Information Technology and Applications in Biomedicine, 2009. ITAB 2009. 9th International Conference on', pp. 1 –2.

Randima Fernando, Mark J. Kilgard (2003), *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley Professional.

Rúbio, Cássio Augusto (2003), Estilização e visualização tridimensional de tumores intracranianos em exames de tomografia computadorizada, Dissertação de mestrado, Universidade Federal do Paraná, p. 99.

Reed, Aaron (2010), *Learning XNA 4.0: Game Development for the PC, Xbox 360, and Windows Phone 7*, first editionª edição, O'Reilly Media.

Rheingans, P. (1992), Color, change, and control of quantitative data display, *em* 'Visualization, 1992. Visualization '92, Proceedings., IEEE Conference on', pp. 252 –259.

Ricci, E. & R. Perfetti (2007), 'Retinal blood vessel segmentation using line operators and support vector classification', *Medical Imaging, IEEE Transactions on* **26**(10), 1357 –1365.

Sage, Daniel, Franck R. Neumann, Florence Hediger, Susan M. Gasser & Michael Unser (2005), 'Automatic tracking of individual fluorescence particles: Application to the study of chromosome dynamics', *IEEE Transactions on Image Processing* **14**, 1372–1383.

Sato, Y., C. Westin, A. Bhalerao, S. Nakajima, N. Shiraga, S. Tamura & R. Kikinis (2000), 'Tissue classification based on 3D local intensity structures for volume rendering', *Visualization and Computer Graphics, IEEE Transactions on* **6**(2), 160 –180.

Schaefer, G., J. Huguet, S.Y. Zhu, P. Plassmann & F. Ring (2006), Adopting the DICOM standard for medical infrared images, *em* 'Engineering in Medicine and Biology Society, 2006. EMBS '06. 28th Annual International Conference of the IEEE', pp. 236 –239.

Schildt, Herbert (1996), *C - Completo e Total*, Makron Books.

Shang, Y., R. Deklerck, E. Nyssen, A. Markova, J. de Mey, X. Yang & K. Sun (2010), 'Vascular active contour for vessel tree segmentation', *Biomedical Engineering, IEEE Transactions on* (99), 1.

Shen, Hong, C.V. Stewart, B. Roysam, Gang Lin & H.L. Tanenbaum (2003), 'Frame-rate spatial referencing based on invariant indexing and alignment with application to online retinal image registration', *Pattern Analysis and Machine Intelligence, IEEE Transactions on* **25**(3), 379 – 384.

Staal, J., M.D. Abramoff, M. Niemeijer, M.A. Viergever & B. van Ginneken (2004), 'Ridge-based vessel segmentation in color images of the retina', *Medical Imaging, IEEE Transactions on* **23**(4), 501 –509.

Susskind, C. (1980), 'Godfrey Newbold Hounsfield, nobel laureate', *Proceedings of the IEEE* **68**(1), 3.

Taylor, II, Russell M. (2000), 'Practical scientific visualization examples', *SIGGRAPH Comput. Graph.* **34**(1), 74–79.

Van Uitert, R., I. Bitter, M. Franaszek & R. Summers (2006), Automatic correction of level set based subvoxel accurate centerlines for virtual colonoscopy, *em* 'Biomedical Imaging: Nano to Macro, 2006. 3rd IEEE International Symposium on', pp. 303 –306.

Van Uitert, R.L. & R.M. Summers (2007), 'Automatic correction of level set based subvoxel precise centerlines for virtual colonoscopy using the colon outer wall', *Medical Imaging, IEEE Transactions on* **26**(8), 1069 –1078.

Verscheure, L., L. Peyrodie, N. Makni, N. Betrouni, S. Maouche & M. Vermandel (2010), Dijkstra's algorithm applied to 3D skeletonization of the brain vascular tree: Evaluation and application to symbolic, *em* 'Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE', pp. 3081 –3084.

Vlachos, Marios & Evangelos Dermatas (2010), 'Multi-scale retinal vessel segmentation using line tracking', *Computerized Medical Imaging and Graphics* **34**(3), 213 – 227.

von Land, C.D., V. Lashin, A. Oriol & J.J. Villanueva (1997), Object-oriented design of the DICOM standard and its application to cardiovascular imaging, *em* 'Computers in Cardiology 1997', pp. 645 –648.

Waltza, Frederick M. & John W. V. Miller (1998), An efficient algorithm for gaussian blur using finite-state machines, *em* 'Conf. on Machine Vision Systems for Inspection and Metrology VII', p. 8.

Wan, Ming, Zhengrong Liang, Qi Ke, Lichan Hong, I. Bitter & A. Kaufman (2002), 'Automatic centerline extraction for virtual colonoscopy', *Medical Imaging, IEEE Transactions on* **21**(12), 1450 –1460.

Williams, Thomas, Colin Kelley, Russell Lang, Dave Kotz, John Campbell, Gershon Elber & Alexander Woo (2011), 'Gnuplot', www.gnuplot.info.

Wilson, D.L. & J.A. Noble (1999), 'An adaptive segmentation algorithm for time-of-flight mra data', *Medical Imaging, IEEE Transactions on* **18**(10), 938 –945.

Xu, Jing, Daming Feng, Jian Wu & Zhiming Cui (2009), Robust centerline extraction for tree-like blood vessels based on the region growing algorithm and level-set method, *em* 'Fuzzy Systems and Knowledge Discovery, 2009. FSKD '09. Sixth International Conference on', Vol. 4, pp. 586 –591.

Yim, P.J., P.L. Choyke & R.M. Summers (2000), 'Gray-scale skeletonization of small vessels in magnetic resonance angiography', *Medical Imaging, IEEE Transactions on* **19**(6), 568 –576.