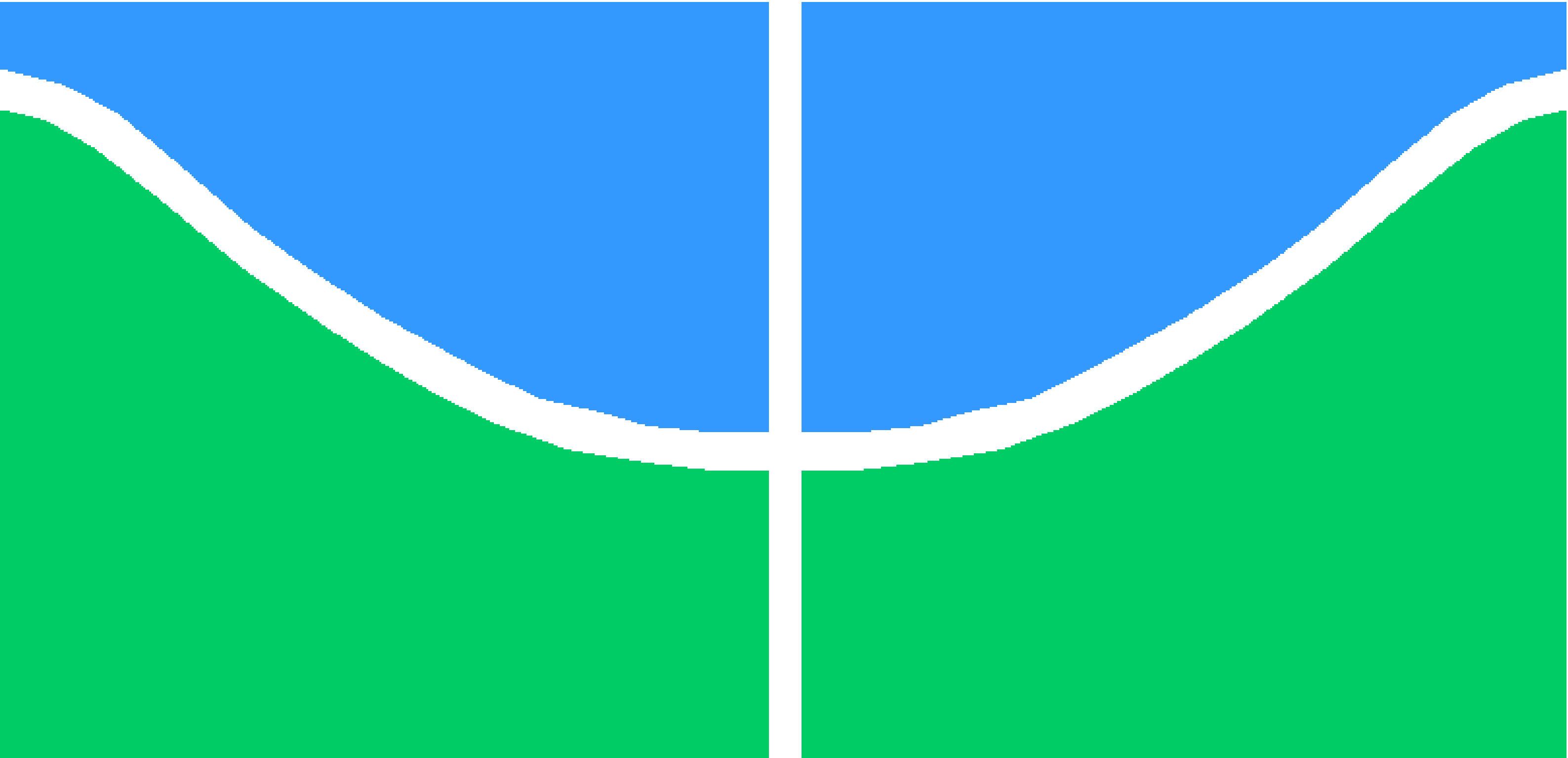


DISSERTAÇÃO DE MESTRADO

RECONSTRUÇÃO DE IMAGENS DE RESSONÂNCIA MAGNÉTICA ACELERADA POR PLACAS DE PROCESSAMENTO GRÁFICO

Thales Henrique Dantas

Brasilia, outubro de 2015

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

RECONSTRUÇÃO DE IMAGENS DE RESSONÂNCIA MAGNÉTICA ACELERADA POR PLACAS DE PROCESSAMENTO GRÁFICO

THALES HENRIQUE DANTAS

ORIENTADOR: JOÃO LUIZ A. DE CARVALHO

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGEA 605/15

BRASÍLIA/DF: OUTUBRO – 2015

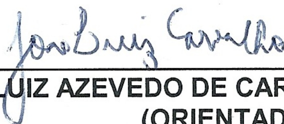
UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

RECONSTRUÇÃO DE IMAGENS DE RESSONÂNCIA MAGNÉTICA ACELERADA POR PLACAS DE PROCESSAMENTO GRÁFICO

THALES HENRIQUE DANTAS

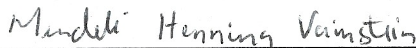
DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:



JOÃO LUIZ AZEVEDO DE CARVALHO, Dr., ENE/UNB
(ORIENTADOR)

RAFAEL MORGADO SILVA, Dr., FGA/UNB
(EXAMINADOR INTERNO)



MENDELI HENNING VAINSTEIN, Dr., IF/UFRGS
(EXAMINADOR EXTERNO)

Brasília, 27 de junho de 2014.

FICHA CATALOGRÁFICA

DANTAS, THALES H.

Reconstrução de Imagens de Ressonância Magnética acelerada por Placas de Processamento Gráfico [Distrito Federal] 2015.

xii, 56p., 210 x 297 mm (ENE/FT/UnB, Mestre, Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.

Departamento de Engenharia Elétrica

1.Imagens Médicas	2.Ressonância Magnética
3.Placas de processamento gráfico	4.GPGPU
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

DANTAS, T. H. (2015). Reconstrução de Imagens de Ressonância Magnética acelerada por Placas de Processamento Gráfico. Dissertação de Mestrado em Engenharia Elétrica, Publicação PPGEA 605/15, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 56p.

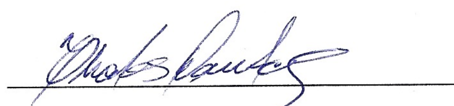
CESSÃO DE DIREITOS

AUTOR: Thales Henrique Dantas.

TÍTULO: Reconstrução de Imagens de Ressonância Magnética acelerada por Placas de Processamento Gráfico.

GRAU: Mestre ANO: 2015

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação de mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte dessa dissertação de mestrado pode ser reproduzida sem autorização por escrito do autor.



Thales Henrique Dantas
SQN 304 bloco H apartamento 105
70736-080 Brasília – DF – Brasil.

Agradecimentos

Gostaria de agradecer o apoio da CAPES e do PROUNI, que contribuíram para viabilizar este trabalho.

Thales Henrique Dantas

Dedicatória

Faço uma dedicação especial a meus pais e irmãos, verdadeiras fortalezas, portos seguros com os quais sempre posso contar. Dedico especialmente a minha esposa, que esteve ao meu lado em todos os momentos. Dedicação especial também ao meu orientador, que acreditou em mim e não me deixou desistir e ao meu sócio, que segurou as pontas em minhas muitas ausências.

Thales Henrique Dantas

RESUMO

RECONSTRUÇÃO DE IMAGENS DE RESSONÂNCIA MAGNÉTICA ACELERADA POR PLACAS DE PROCESSAMENTO GRÁFICO

Autor: Thales Henrique Dantas

Orientador: João Luiz de A. Carvalho

Programa de Pós Graduação em Engenharia Elétrica

Brasília, outubro of 2015

Fourier velocity encoding **(FVE) é útil no diagnóstico de doenças valvulares, visto que pode eliminar os efeitos de volume parcial que podem causar perda de informação de diagnóstico no imageamento de fluxo cardiovascular por contraste de fase. FVE também foi proposto como método para a medição da taxa de cisalhamento da superfície das artérias carótidas. Apesar de o tempo de aquisição para FVE no espaço de Fourier bi-dimensional(2DFT) ser proibitivamente longo, o uso de FVE espiral se mostra bastante promissor, uma vez que este é substancialmente mais rápido. Contudo, a reconstrução dos dados de FVE em espiral é longo, devido à sua multi-dimensionalidade e ao uso de amostragem não Cartesiana. Isso é particularmente importante para aquisições de múltiplos cortes, volumes e(ou) de múltiplos canais. Os conjuntos de dados de FVE em espiral consistem em pilhas de espirais resolvidas no espaço k_x-k_y-k_v. A distribuição de velocidade espaço-temporal, $m(x, y, v, t)$, é tipicamente obtida a partir dos dados no espaço-k, $M(kx, ky, kv, t)$, aplicando uma transformada inversa de Fourier não uniforme ao longo de k_x-k_y, seguida de uma transformada Cartesiana ao longo de k_v. Com esta abordagem, toda a matriz $m(x, y, v, t)$ é calculada. Entretanto estamos tipicamente interessados nas distribuições de velocidade associadas com uma pequena região de interesse dentro do plano x-y. Nós propomos o uso da reconstrução de um único *voxel* usando a transformada direta de Fourier (DrFT) para reconstruir os dados da FVE espiral. Ao passo que o tempo de reconstrução por DrFT de toda a imagem é ordens de magnitude maior que a reconstrução por *gridding* ou *Non Uniform Fast Fourier Transform*(NUFFT), a equação da DrFT permite a reconstrução de *voxels* individuais com uma quantidade consideravelmente reduzida de esforço computacional. Adicionalmente, propomos o uso de placas de processamento gráfico de uso geral (GPGPUs) para acelerar ainda mais a reconstrução e alcançar reconstruções de FVE espirais de maneira aparentemente instantânea. É apresentada também uma proposta para, potencialmente, acelerar também a reconstrução por *gridding* ou NUFFT utilizando o algoritmo de Goertzel para reconstruir uma quantidade limitada de pontos também por estes métodos.**

ABSTRACT

MAGNETIC RESONANCE IMAGE RECONSTRUCTION ACCELERATED BY GENERAL PURPOSE GRAPHIC PROCESSING UNITS

Author: Thales Henrique Dantas

Supervisor: João Luiz de A. Carvalho

Programa de Pós Graduação em Engenharia Elétrica

Brasília, october of 2015

Fourier velocity encoding (FVE) is useful in the assessment of valvular disease, as it eliminates partial volume effects that may cause loss of diagnostic information in phase-contrast imaging. FVE has also been proposed as a method for measuring wall shear rate in the carotid arteries. Although the scan-time of Two-dimensional Fourier Transform (2DFT) FVE is prohibitively long for clinical use, the spiral FVE method shows promise, as it is substantially faster. However, the reconstruction of spiral FVE data is time-consuming, due to its multidimensionality and the use of non-Cartesian sampling. This is particularly true for multi-slice/3D and/or multi-channel acquisitions. Spiral FVE datasets consist of temporally-resolved stacks-of-spirals in k_x-k_y-k_v space. The spatial-temporal-velocity distribution, $m(x, y, v, t)$, is typically obtained from the k-space data, $M(kx, ky, kv, t)$, by first using a non-Cartesian inverse Fourier transform along k_x-k_y, followed by a Cartesian inverse Fourier transform along k_v. With this approach, the entire $m(x, y, v, t)$ matrix is calculated. However, we are typically only interested in the velocity distributions associated with a small region-of-interest within the x-y plane. We propose the use of single-voxel direct Fourier transform (DrFT) to reconstruct spiral FVE data. While whole-image DrFT is orders of magnitude slower than the gridding and Non Uniform Fast Fourier Transform(NUFFT) algorithms, the DrFT equation allows the reconstruction of individual voxels of interest, which considerably reduces the computation time. Additionally, we propose the use of general-purpose computing on graphics processing units (GPGPUs) to further accelerate computation and achieve seemingly instantaneous spiral FVE reconstruction. We also propose a method that shows potential to accelerate reconstructions using gridding and NUFFT by means of using the Goertzel algorithm to reconstruct onlya small number of pixels.

SUMÁRIO

1 Introdução **1**
1.1 Motivação 1
1.2 Estado da arte 2
1.3 Objetivos e contribuições 3
1.4 Abordagem 3
1.5 Apresentação do manuscrito 4

2 Ressonância Magnética **5**
2.1 Princípios 5
2.2 Aquisição 7
2.2.1 Seleção de corte 7
2.2.2 Codificação Espacial 8
2.3 Reconstrução 11
2.4 Amostragem não Cartesiana 11
2.5 Reconstrução não Cartesiana 12
2.6 Imageamento de Fluxo 12
2.6.1 Contraste de Fase 12
2.6.2 *Fourier Velocity Encoding* 13
2.6.3 Multi-dimensionalidade 14
2.6.4 Tempo de reconstrução 15

3 Processamento Paralelo em CUDA **16**
3.1 Histórico 16
3.2 Arquitetura 19
3.3 Definição da linguagem utilizada 22
3.4 Fundamentos da linguagem 23

4 Reconstrução em Cuda **29**
4.1 CUDA DrFT 29
4.2 CUDA *Gridding* 32
4.3 NUFFT 34
4.4 Resultados e Discussão 35

5 Reconstrução de voxel único por DrFT em CUDA **37**

5.1 Proposta 37
5.2 Implementação 38
5.3 Resultados 40

6 Reconstrução de voxel único por gridding em CUDA 41
6.1 Introdução 41
6.2 Algoritmo de Goertzel 42
6.3 Implementação 44
6.4 Avaliação do algoritmo proposto 46

7 Conclusões 48

REFERÊNCIAS BIBLIOGRÁFICAS 49

Anexos 52

LISTA DE FIGURAS

2.1 **Sequências de pulsos e suas trajetórias correspondentes no espaço-k. (a) 2DFT (b) espiral. [Reproduzido de [8]]** 10
2.2 **Sequência de pulsos para aquisição FVE espiral. (a) seleção do corte (b) gradiente bipolar para codificação de velocidade, que muda de intensidade para a codificação das velocidades (c) leitura espiral (d) gradientes de refocalização. [Adaptado de [8]]** 14

3.1 **Crescimento do poder de processamento das CPUs. [Adaptado de [11]].** 17
3.2 **Crescimento do poder de processamento das GPUs comparado ao das CPUs (GFLOPS - Giga Floating point Operations Per Second). [Adaptado de [12]].** 18
3.3 **Modelo de memória hierárquica usada nas GPGPUs. [Adaptado de [12]].** 21
3.4 **Operações executadas em paralelo** 22
3.5 **Cálculo de um resultado da matriz C. [Adaptado de [12]]** 25
3.6 **Cálculo de um conjunto de resultados da matriz C. [Adaptado de [12]]** .. 26

4.1 **Áreas influenciadas pela interpolação de cada ponto da trajetória espiral. [Reproduzido de [14]].** 34
4.2 **Agrupamentos das interpolações por região. [Reproduzido de [14]].** 35

5.1 **Comparação da reconstrução total e da reconstrução de voxel único.** 38
5.2 **Geração de dados de velocidade de fluxo de maneira interativa. Ilustração de um corte axial do pescoço. (a), (c) - aorta (b), (d) - carótida.** 39

6.1 **Representação do Filtro Goertzel** 44

LISTA DE TABELAS

4.1 Tempo de reconstrução para o algoritmo CUDA DrFT 32

5.1 Comparativo de velocidade para DrFT de voxel único 40

6.1 Comparativo de tempo entre FFT, GPUFFT e GPU Goertzel (valores em ms). 46

LISTA DE SÍMBOLOS E ABREVIAÇÕES

FVE	*Fourier Velocity Encoding*
2DFT	*2 Dimensional Fourier Transform*
DrFT	*Direct Fourier Transform*
NUFFT	*Non Uniform Fast Fourier Transform*
PET	*Positron Emission Tomography*
SPECT	*Single Photon Emission Computed Tomography*
GPGPU	*General Purpose Graphics Processing Unit*
FFT	*Fast Fourier Transform*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
TR	*Repetition Time*
MIPS	*Million Instructions Per Second*
GPU	*Graphics Processing Unit*
CMOS	*Complementary Metal Oxide Semiconductor*
ULA	**Unidade Lógica Aritmética**
SIMD	*Single Instruction Multiple Data*
SIMT	*Single Instruction Multiple Thread*
IFFT	*Inverse Fast Fourier Transform*
DFT	*Discrete Fourier Transform*

Capítulo 1

Introdução

Este capítulo apresenta as principais motivações do trabalho, apresentando o estado da arte e a abordagem adotada para criarmos uma contribuição relevante para a área. Também são apresentados os tópicos tratados em cada capítulo do trabalho.

1.1 Motivação

Apesar do ainda alto custo dos equipamentos, o uso da ressonância magnética para o imageamento médico é bastante difundido. O uso de radiação não ionizante se mostra um grande diferencial em relação a outras técnicas de imageamento médico como a tomografia computadorizada, PET (*positron emission tomography*) e SPECT (*single photon emission computed tomography*). A qualidade das imagens obtidas e a possibilidade de, em um único aparelho, ter imagens anatômicas, metabólicas e de fluxo colocam a ressonância magnética na posição de única tecnologia capaz de fornecer um exame completo do ponto de vista de imageamento cardiovascular.

Um aspecto negativo é que as aquisições em ressonância magnética são relativamente mais longas do que as aquisições em outras tecnologias de imageamento, como a tomografia computadorizada e a ultassonografia, especialmente quando estamos interessados em dados multi-corte, 3D e(ou) informações de fluxo e(ou) resolvidos no tempo. Para estes tipos de aquisições, os tempos de aquisição em ressonância magnética se tornam ainda mais longos com o uso da tradicional trajetória 2DFT (*two dimensional Fourier Transform*). Contudo as aquisições em espiral têm se mostrado adequadas para a aquisição rápida de dados.

Os tempos de aquisição podem aumentar ainda mais quando além de informações anatômicas se deseja adquirir informações de fluxo. A técnica de constraste de fase [1] permite a captura de informação de velocidade adicionando apenas mais uma etapa de codificação para cada eixo de velocidade que se quer medir. Essa técnica, contudo, só permite extrair uma informação de velocidade para cada *voxel* adquirido, o que se

mostra inadequado quando um único *voxel* engloba regiões com matérias em diferentes velocidades, como é o caso junto às paredes dos vasos sanguíneos. Essas limitações nas medidas de velocidade podem ser superadas com a técnica de FVE (*Fourier Velocity Encoding*) [2], ao custo de adição de etapas de codificação. O número de etapas adicionais é proporcional à resolução desejada para a informação de velocidade. A *spiral FVE* [3] aplica a trajetória em espiral à FVE para acelerar a aquisição.

A reconstrução de uma grande quantidade de dados amostrados de forma não cartesiana, contudo, se mostra uma tarefa que é computacionalmente muito mais intensa do que a reconstrução de dados uniformemente amostrados. Algoritmos de reconstrução rápidos, como a NUFFT (*non unoform Fourier transform*) [4] tem tomado o lugar da da FFT (*fast Fourier transform*), que era usada para a reconstrução dos dados adquiridos de forma Cartesiana, voltando a dar agilidade na reconstrução das imagens de ressonância magnética.

Nos casos em que a ressonância magnética é utilizada para estudar fluxo sanguíneo, a quantidade de dados a serem reconstruídos aumenta em até 6 vezes para contraste de fase e mais ainda para FVE, pois passa a ser necessária a reconstrução dos dados referentes à codificação de velocidade. Nestes casos, geralmente estamos interessados na informação de fluxo sanguíneo apenas para alguns pontos de interesse no plano x-y, onde temos veias e artérias. Para a reconstrução deste tipo de informação, algoritmos como a NUFFT "desperdiçam" esforço computacional para reconstruir as informações de fluxo para todo o plano x-y, ou seja, calculam e armazenam as informações de velocidade para regiões da imagem que na verdade representam tecidos estáticos ou ausência de sinal. O esforço computacional desnecessário se torna ainda mais evidente quanto estudamos dados tridimensionais e(ou) resolvidos no tempo. Tudo isso faz com que, mesmo com o emprego dos algoritmos rápidos, a reconstrução das informações possa levar minutos.

1.2 Estado da arte

Utilizando a trajetória Cartesiana tradicional, que na área de ressonância magnética é chamada de 2DFT, com codificação dos dados de velocidade, o tempo de aquisição para vários quadros temporais de múltiplos cortes se torna proibitivo no caso de uso de FVE. Como os tempos de aquisição são longos, para capturar um determinado quadro temporal de um batimento cardíaco, por exemplo, são necessárias várias aquisições parciais. Através da técnica CINE, é possível sincronizar as aquisições parciais com o batimento cardíaco para fazer uma reconstrução completa [5]. Contudo, é necessário que durante todo o tempo de captura o paciente permaneça imóvel. Os pacientes geralmente têm relativa facilidade em atender este requisito no que diz respeito a sua parte motora, podendo permanecer imóveis durante vários minutos. Este requisito se torna mais complexo no que diz respeito à parte respiratória.

É possível segurar o fôlego, interrompendo o movimento do tórax, mas aquisições que durem mais que 10 segundos podem se apresentar como irrealizáveis para alguns pacientes.

Carvalho e Nayak mostraram que, utilizando trajetórias de aquisição em espiral com codificação de velocidade de Fourier (*spiral* FVE)[3], foram alcançados tempos de aquisição adequados. Utilizando técnicas de aceleração para a aquisição, reduziram o tempo necessário para aquisição de um volume de referência de 216 batimentos cardíacos para apenas 12 batimentos. Isso tornou possível a aquisição em apenas uma curta retenção de fôlego de aproximadamente 10 segundos.

Os dados não uniformemente amostrados, contudo, impossibilitam o uso direto das tradicionais transformadas rápidas de Fourier para a reconstrução dos dados, pois estas pressupõem amostras uniformemente espaçadas. A reconstrução de amostras não uniformemente espaçadas pode ser feita aplicando a chamada a transformada direta de Fourier (DrFT)[6], mas o esforço computacional é muito maior (proporcional a N^2, contra $Nlog(N)$ da FFT, em que N é o número de amostras no espaço-k). Para acelerar a reconstrução, os dados são interpolados para uma grade uniforme (processo conhecido como *gridding*)[7], para então ser aplicada a FFT.

Na técnica de *gridding*, a escolha do *kernel* de interpolação afeta consideravelmente a qualidade da imagem reconstruída. É possível utilizar técnicas de minimização de erros para ajustar o *kernel* de interpolação para melhorar a qualidade da reconstrução, como faz a técnica de NUFFT.

1.3 Objetivos e contribuições

O objetivo deste trabalho é mostrar que é possível uma reconstrução rápida e analiticamente exata, explorando as propriedades da reconstrução por DrFT e o poder de processamento paralelo das placas de processamento gráfico de uso geral ou GPGPUs (*General Purpose Graphics Processing Units*). Também propomos uma técnica para acelerar a reconstrução por *gridding*, quando a região de interesse para a reconstrução é limitada a poucos *voxels*, utilizando o algoritmo de Goertzel também em GPGPU.

1.4 Abordagem

Para alcançar os objetivos propostos, foi feito um estudo sobre a tecnologia de ressonância magnética e também sobre suas técnicas de aquisição e reconstrução de imagens. Vários algoritmos de reconstrução de imagens simples foram implementados em MATLAB e em linguagem C para aprofundar o conhecimento e testar algumas hipóteses para o processo de reconstrução acelerada.

Foram estudadas então as tecnologias de programação para processadores paralelos

e foi feita a opção pelo aprofundamento no estudo da linguagem CUDA, que é a linguagem de programação criada pelo fabricante de placas de processamento gráfico Nvidia para o uso do poder de processamento paralelo de suas placas de processamento gráfico. Os algoritmos de reconstrução foram implementados em CUDA e comparados com os resultados originais em MATLAB e em linguagem C para validação. Após as implementações e testes iniciais, onde foram utilizados dados simulados, passamos a utilizar um conjunto de dados de FVE espiral da artéria carótida, com vários cortes, quadros temporais e canais de aquisição.

1.5 Apresentação do manuscrito

No capítulo 2 é feita uma revisão sobre a tecnologia de ressonância magnética, seus princípios físicos e a aquisição dos sinais.

Em seguida, no capítulo 3 é apresentada a tecnologia de processamento paralelo disponível nas placas de processamento gráfico de uso geral atualmente disponíveis no mercado, assim como conceitos de programação para o uso dessa tecnologia com foco na linguagem CUDA.

No capítulo 4 se discute a implementação de alguns algoritmos de reconstrução de imagens de ressonância magnética em CUDA e discute os resultados, comparando com as implementações tradicionais em CPU (*central processing unit*).

O capítulo 5 traz uma proposta de reconstrução de apenas um *voxel* em CPU e em GPGPU (*general purpose graphics processing unit*), comparando os resultados obtidos com os resultados do capítulo 4.

Uma alternativa para acelerar a reconstrução por *gridding* é discutida no capítulo 6 para imagens onde as regiões de interesse se limitam a poucos *voxels*. Os resultados são comparados com os resultados dos dois capítulos anteriores.

Os resultados gerais são analisados e as conclusões são apresentadas no capítulo 7.

Capítulo 2

Ressonância Magnética

Este capítulo faz uma revisão sobre os princípios físicos da ressonância magnética. São mostrados os conceitos tanto pasa as aquisições tradicionais quanto para as aquisições aceleradas. São discutidas as principais técnicas de reconstrução dos dados adquiridos através de ressonância magnética. São apresentadas também as tecnologias de aquisição de dados para estudos de fluxo sanguíneo, assim como as principais limitações para a aquisição e reconstrução deste tipo de dado.

2.1 Princípios

Atualmente, a ressonância magnética nuclear é a única tecnologia de imageamento médico que tem capacidade de fornecer tanto dados de anatomia e metabolismo quanto de fluxo. A aquisição se baseia no uso de fortes campos magnéticos e energia na faixa de rádio-frequência(RF). Tanto os campos magnéticos quanto a energia de RF são não-ionizantes e são consideradas completamente seguros para os pacientes. É uma tecnologia que, apesar do ainda alto custo, já é bastante difundida.

A descrição da mecânica quântica é complexa e não está diretamente ralacionada ao objetivo desta dissertação. Uma vez que a menor quantidade de matéria representada pela resolução atual máxima das técnicas de ressonância magnética é grande o suficiente, será apresentado um tratamento semi-clássico para os princípios físicos utilizados.

Átomos que possuem *spin* não nulo, como núcleos de hidrogênio, possuem um pequeno campo magnético. Em condições normais, os núcleos apontam os eixos de seus campos magnéticos em direções aleatórias, resultando em uma magnetização total nula. Entretanto, na presença de uma campo magnético externo B_0, os núcleos

tendem a alinhar os eixos de seus campos com o eixo desse campo externo. Aproximadamente metade dos núcleos irão apontar seus eixos na mesma direção e sentido de B_0, ao passo que o restante dos núcleos irão apontar seus eixos no sentido oposto. Quanto maior for a intensidade do campo B_0, mais núcleos apontarão no mesmo sentido deste, criando uma magnetização total resultante passível de ser detectada. Esta magnetização resultante será tão maior quanto for a intensidade do campo externo B_0.

Os equipamentos de ressonância magnética usam supercondutores para gerar um forte campo magnético (comumente de 1,5 ou 3 Tesla). Este campo permanece ligado mesmo quando o equipamento não está em uso. Estes equipamentos contam também com bobinas para gerar perturbações intencionais neste campo na forma de gradientes de campos magnéticos. Tipicamente são utilizadas 3 bobinas, uma para cada direção espacial. Os gradientes gerados por estas bobinas (G_x, G_y e G_z) geram variações lineares em cada direção no campo magnético total.

Outro efeito do campo magnético externo sobre os *spins* é que ele faz com os eixos magnéticos dos mesmos precessem ao redor do eixo magnético do campo externo. A frequência de precessão dos eixos magnéticos(ω) dos núcleos varia de acordo com a intensidade do campo(B), segundo a fórmula:

$$\omega = \gamma B \tag{2.1}$$

em que γ é a constante giromagnética (42,6 MHz/Tesla para prótons ^{1}H). A frequência de precessão quando $B = B_0$ é chamada de frequência de Larmor. Para estudos de ressonância magnético em tecidos humanos, nos concentramos em estudar o comportamento do hidrogênio, por ser parte das moléculas de água, que é muito abundante no corpo.

Usando os gradientes, podemos variar os campos magnéticos percebidos pelos núcleos de hidrogênio em diferentes posições do material sob estudo, efetivamente codificando informação espacial na frequência ou na fase dos *spins*.

O terceiro componente de um equipamento de ressonância magnética é a bobina de RF. Esta bobina é utilizada para emitir pulsos de RF na mesma frequência que a frequência de precessão dos núcleos de hidrogênio que se deseja excitar. Na presença dos gradientes, diferentes posições do material sob estudo sentem campos magnéticos diferentes. Com isso, prótons em posições diferentes precessam em frequências diferentes. Isso permite que possamos excitar regiões específicas utilizando pulsos de RF em frequências adequadas.

Os núcleos que precessam na mesma frequência do pulso de RF estão em ressonância com este campo e absorvem energia do pulso. Ao absorverem a energia do pulso de RF, os núcleos aumentam o angulo entre a direção do campo B_0 e a direção de seu eixo magnético. Por definição, o campo B_0 aponta na direção do eixo z. Ao aplicarmos

o pulso de RF, os núcleos excitados deslocam seu eixo magnético na direção do plano x-y, criando a chamada magnetização transversal.

A magnetização transversal diminui de forma diferente para cada tecido após o desligamento do pulso de RF. Devido à interação entre os campos dos próprios *spins*, eles vão saindo de fase e a magnetização líquida diminui segundo uma constante de tempo denominada relaxamento *spin-spin* ou relaxamento T_2. Existe ainda o tempo de relaxamento T_2*, que é ainda menor que o tempo de relaxamento T_2 e se deve a não homogeneidades nos campos observados pelos *spins*. Este relaxamento pode ser corrigido aplicando um pulso de 180°, que força um realinhamento no sentido inverso. A magnetização no eixo z é recuperada segundo uma terceira constante de tempo diferente, denominada relaxamento *spin-lattice* ou relaxamento T_1. O tempo de relaxamento T_1 é mais longo que o tempo de relaxamento T_2. Isso se deve ao fato de que antes dos *spins* realinharem seus eixos magnéticos com o campo magnético externo (relaxamento T_1), eles perdem a coerência de fase, o que diminui a magnetização líquida (relaxamento T_2).

A bobina utilizada para a transmissão do pulso de RF também é muitas vezes responsável por captar o sinal. A bobina de RF consegue captar a onda eletromagnética resultante da variação do campo magnético no plano x-y devido à precessão dos núcleos. É comum o uso de várias bobinas em posições diferentes para realizar a captura dos sinais.

2.2 Aquisição

As bobinas de recepção captam o sinal combinado, resultante da oscilação de todos os prótons excitados. Para obtermos informação espacial sobre os prótons, temos que selecionar o corte a ser adquirido e realizar codificação espacial em frequência e em fase.

2.2.1 Seleção de corte

Se aplicarmos, por exemplo, um gradiente na direção z (G_z) no momento do pulso de RF, teremos que os núcleos ao longo do eixo z precessarão em frequências diferentes segundo

$$\omega(z) = \gamma(B_0 + G_z z). \tag{2.2}$$

Com isso o pulso de RF excitará apenas os núcleos que estejam precessando na mesma frequência do pulso. Controlando a frequência central do pulso, podemos selecionar a fatia ao longo do eixo z que iremos excitar e, controlando a largura de banda do pulso ($\Delta\omega$) e a intensidade de gradiente (G_z), podemos definir a largura do

corte conforme

$$\Delta z = \frac{\Delta\omega}{\gamma G_z} \tag{2.3}$$

Após o pulso de RF, apenas os *spins* do corte selecionado irão ter componentes de magnetização transversal. Isso implica que apenas estes *spins* contribuirão para o sinal captado pela bobina de RF.

Para fins didáticos, vamos supor que este primero pulso faz com que o eixo magnéticos dos núcleos sofram um deslocamento de 90° na direção do plano x-y, sendo assim denominado pulso de 90°.

2.2.2 Codificação Espacial

2.2.2.1 Codificação em frequência

Após dois pulsos de 90°, separados por um intervalo de tempo TR (*repetition time*), sendo o primeiro em $t = -\text{TR}$ e o segundo em $t = 0$, a magnetização transversal nas posições (x,y) do corte selecionado será dada por:

$$M_{xy}(t) = M_0(x,y)(1 - e^{\frac{-TR}{T_1}})e^{-\frac{t}{T_2^*}}. \tag{2.4}$$

Vamos supor agora que, no momento da aquisição, aplicamos um gradiente na direção x (G_x). Isso fará com que, ao longo do eixo x, os núcleos precessem com frequências diferentes, variando linearmente suas frequências de precessão em função de suas posições ao longo do eixo x, segundo:

$$\omega(x) = \gamma(B_0 + G_x x). \tag{2.5}$$

Consequentemente, a magnetização transversal será dada por:

$$M_{xy}(t) = M_0(x,y)(1 - e^{\frac{-TR}{T_1}})e^{-\frac{t}{T_2^*}}e^{-i\gamma\int_0^t G_x(\tau)d\tau x}e^{-i\gamma B_0}. \tag{2.6}$$

A bobina de RF captará então não mais uma única frequência, mas uma banda de frequências. A intensidade do sinal em cada frequência dará a quantidade de sinal gerada pelos núcleos de uma determinada posição ao longo do eixo x.

2.2.2.2 Codificação em fase

Após a emissão do pulso de 90°, todos os spins excitados precessam com a mesma fase ϕ_0. Se, antes do momento da aquisição, aplicamos um gradiente na direção y (G_y) por um período T_f, isso fará com que os núcleos ao longo do eixo y precessem com

frequências diferentes. Estas frequências variam linearmente com a posição ao longo do eixo y. Ao desligarmos o gradiente na direção y, todos os núcleos voltam a oscilar na mesma frequência, mas devido ao período em que ficaram oscilando em frequências diferentes, teremos diferenças de fase entre os núcleos ao longo do eixo y. A fase final correspondente a cada posição no eixo y pode ser calculada integrando a frequência de precessão ao longo do período T_f :

$$\phi(y) = \phi_0 + \gamma \int_0^{T_f} G_y(t)\, dy. \tag{2.7}$$

Com isso, podemos determinar a quantidade de sinal gerada pelos prótons de uma determinada posição ao longo do eixo y, de acordo com a informação de fase do sinal capturado pela bobina de RF.

2.2.2.3 Espaço-k

Quando os gradientes são reproduzidos, o sinal recebido pela bobina em um determinado instante de tempo, representa a soma de diferentes sinais senoidais gerados por núcleos em diferentes partes do corte excitado, cada um com frequência e fase correspondentes à sua localização espacial. Tomando como exemplo uma excitação que selecione um corte axial do corpo, o valor do sinal demodulado em um dado instante t representa uma amostra da transformada de Fourier $M(k_x, k_y)$ da imagem $m(x, y)$ segundo:

$$M(k_x, k_y) = \int_x \int_y m(x, y) e^{-i2\pi(k_x x + k_y y)}\, dx\, dy. \tag{2.8}$$

em que as coordenadas k_x e k_y variam segundo os gradientes de leitura:

$$k_x(t) = \frac{\gamma}{2\pi} \int_0^t G_x(\tau) d\tau \tag{2.9}$$

$$k_y(t) = \frac{\gamma}{2\pi} \int_0^t G_y(\tau) d\tau. \tag{2.10}$$

e

$$m(x, y) = M_0(x, y)(1 - e^{-\frac{TR}{T_1}}) e^{-\frac{t}{T_2}} e^{-\frac{t}{T_2^*}}. \tag{2.11}$$

O termo

$$e^{-i\gamma B_0} \tag{2.12}$$

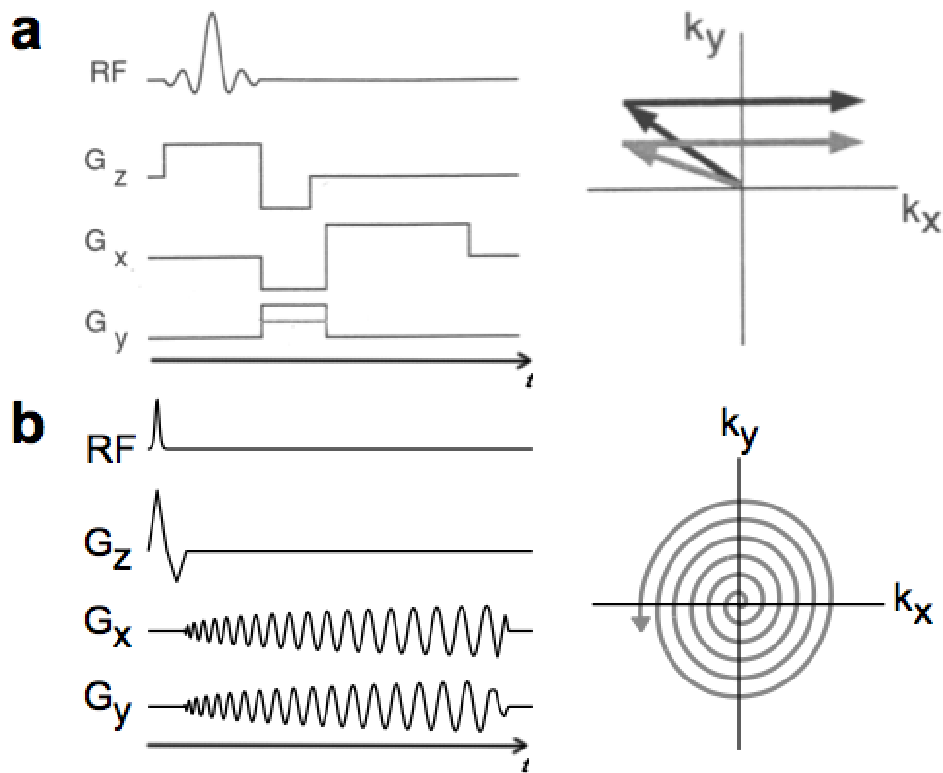


Figura 2.1: Sequências de pulsos e suas trajetórias correspondentes no espaço-k. (a) 2DFT (b) espiral. [Reproduzido de [8]]

é calcelado pela demodulação de $M_{x,y}(t)$.

Na prática, os gradientes permitem o deslocamento no chamado domínio de Fourier ou espaço-k. As sequências de gradientes e pulsos são ilustradas em um diagrama chamado de sequência de pulsos. O projeto das sequências de pulsos é feito de forma a causar uma determinada trajetória para a aquisição das amostras no espaço-k. Dessa forma, é possível personalizar o percurso no espaço-k durante a aquisição. A Fig. 2.1 ilustra algumas sequências de pulsos e suas correspondentes trajetórias. O diagrama de pulso que representa a aquisição 2DFT mostra sequência necessária para percorrer uma linha da aquisição. A linha mais clara na representação do campo G_y ilustra como seria a aquisição de outra linha, que tem seu percurso correspondente representado na trajetória também pela linha mais clara.

Após a aquisição de todos os dados no domínio de Fourier, podemos obter uma imagem aplicando a transformada de Fourier inversa. A imagem resultante representa a distribuição de densidade de prótons ponderada por duas funções. A primeira função descreve o crescimento da componente longitudinal (relaxamento T_1) e a segunda função descreve o decaimento da componente transversal (relaxamento T_2 e T_2^*). Isso implica o fato de que as imagens de ressonância magnética não representam puramente a densidade de prótons, mas sim uma ponderação da densidade pelos fatores T_1 e T_2, que dependem das características dos tecidos, e dos parâmetros TR (tempo de

repetição) e TE (tempo de leitura), os quais são definidos pelo operador.

2.3 Reconstrução

Como visto anteriormente, as informações geradas através de ressonância magnética são captadas pela bobina de RF na forma de intensidades, frequências e fases, no que chamamos de espaço-k. Foi também visto como é feita a seleção dos cortes e como são codificadas as informações do plano x-y ao longo deste espaço-k. Para reconstruir as imagens adquiridas no espaço-k utilizamos a transformada inversa de Fourier.

Se os dados são uniformemente amostrados, ou seja, em uma grade uniforme no espaço-k, podemos aplicar diretamente as transformadas rápidas de Fourier, ou FFTs. Existem várias algoritmos para o cálculo de FFTs e é comum a várias arquiteturas de processadores a existência de hardware dedicado para a aceleração do cálculo de FFTs.

Na aquisição cartesiana dos dados, cada linha da grade de amostragem no espaço-k é adquirida em um tempo TR. O tempo necessário então para uma aquisição completa é de tantos períodos TR quantas linhas forem necessárias para a resolução necessária.

2.4 Amostragem não Cartesiana

A aquisição dos dados em uma grade Cartesiana uniforme facilita a reconstrução dos dados através da aplicação da FFT. Este tipo de aquisição, contudo, representa uma escolha pobre no que diz respeito ao tempo de aquisição.

Assumindo a abstração de que, para adquirir todo o espaço-k, devemos percorrê-lo linha por linha de maneira uniforme, encontramos dificuldades práticas para definir a sequência de pulsos necessária para criar este caminho de maneira rápida.

Um longo tempo de aquisição dificulta ou até mesmo impossibilita a aplicação da ressonância magnética para estudos cardíacos, em que existe a presença de movimento no tecido a ser estudado. É possível a aplicação de técnicas que sincronizam a aquisição dos dados com o ciclo cardíaco, como a técnicas CINE. Através destas técnicas, é possível adquirir uma determinada posição do ciclo através de aquisições parciais feitas na mesma posição relativa ao longo de vários batimentos. Ainda assim, a influência de outros movimentos, como o respiratório, quando a aquisição é mais longa que a capacidade de segurar o fôlego do paciente, prejudicam a qualidade final das imagens que são geradas através destes métodos.

Para acelerar a aquisição dos dados, pode-se realizar amostragens não Cartesianas, onde os percursos gerados são mais fáceis de serem realizados através dos gradientes e campos das máquinas de ressonância magnética de forma rápida.

Através da aquisição não-Cartesiana, é possível capturar todas as fases do ciclo cardíaco em apenas um curto fôlego, minimizando assim os artefatos gerados pelos movimentos de respiração.

2.5 Reconstrução não Cartesiana

A reconstrução de dados amostrados de forma não Cartesiana não pode ser feita através da aplicação direta da FFT, uma vez que tais algoritmos tiram vantagem exatamente das simetrias decorrentes do espaçamento regular das amostras.

A aplicação da transformada direta de Fourier (DrFT) é possível, uma vez que não impõe sobre os dados qualquer restrição quanto ao espaçamento das amostras.

Temos como alternativa a utilização dos algoritmos de *gridding*, onde as amostras não uniformes são interpolados para uma grade uniforme, para então serem reconstruídos por FFT.

Os algorítmos de DrFT e *gridding* são abordados em maior detalhe no capítulo 4.

2.6 Imageamento de Fluxo

Através da ressonância magnética, além da aquisição de informações anatômicas, é possível também adquirir informações de fluxo sanguíneo. A avaliação do fluxo sanguíneo através de ressonância magnética apresenta algumas vantagens sobre ultrassonografia Doppler.

A ultrassonografia Doppler ainda é o método mais usual para estudos de fluxo sanguíneo pela comunidade médica. Esta tecnologia é bastante barata, mas apresenta algumas limitações práticas como a dificuldade para estudar vasos mais profundos e a necessidade de uma janela acústica livre desde a pele até os vasos a serem estudados. Outro fator importante para o uso da ultrassonografia Doppler é a habilidade do operador para conseguir realizar os estudos. Através da ressonância magnética, é possível avaliar fluxo em vasos profundos, ainda que em regiões oclusas por ossos e de maneira menos dependente do operador.

2.6.1 Contraste de Fase

Na técnica imageamento de fluxo por contraste de fase, um gradiente bipolar alinhado com o eixo do fluxo é usado para se obter uma medida de velocidade para cada *pixel*, ou *voxel*, da imagem. Na prática, são feitas duas aquisições, variando o primeiro momento do gradiente bipolar. Com esta técnica, *spins* estacionários sofrem um defasamento total nulo, uma vez que ambos os pulsos do gradiente bipolar têm a mesma amplitude e duração. Já os *spins* que estão em movimento irão sofrer um

defasamento não nulo, já que, com o deslocamento, se encontram em uma posição diferente ao longo do gradiente quando percebem o segundo pulso, causando uma diferença de fase acumulada. A informação de velocidade é obtida diretamente a partir dessa diferença de fase.

É possível combinar o contraste de fase com CINE, para gerar imagens ao longo de um ciclo cardíaco, e mostrar informações de movimento e fluxo sanguíneo. O contraste de fase também pode ser usado para aquisições em tempo real usando codificações sequenciais, periódicas e contínuas.

A técnica de constraste de fase apresenta problemas de inconsistência de dados, efeitos de volume parcial e dispersão de fase dentro do *voxel*, o que pode levar a medidas subestimadas de velocidade de pico [1]. Com o uso de *voxels* grandes, *spins* estacionários e em movimento coexistem. Isso resulta em perda de sinal, distorção e erros de estimação de velocidade, uma vez que o sinal de fase medido representa uma média das velocidades de todos os *spins* de um *voxel*. Por estas razões, o contraste de fase não se mostra adequado para a medição precisa de velocidades de pico, especialmente em jatos de fluxo complexo ou turbulentos, como os normalmente observados em válvulas com estenose ou refluxo.

2.6.2 *Fourier Velocity Encoding*

A técnica *Fourier Velocity Encoding*, ou simplesmente FVE[9], pode ser considerada o equivalente em ressonância magnética da ultrassonografia espectral Doppler. Nesta técnica, todo o espectro de velocidades dos *spins* em cada *voxel* é medido, codificando em fase no domínio de Fourier (k_v) a informação de velocidade. Com isso, as limitações mencionadas para a técnica de contraste de fase são superadas.

Esta técnica ainda não é utilizada clinicamente pois, em princípio, o seu tempo de aquisição é consideravelmente mais longo que o tempo necessário para a aquisição por contraste de fase, porém várias técnicas para acelerar a aquisição do FVE têm sido propostas.

Enquanto a técnica de contraste de fase provê apenas a média das velocidades dos *spins* de um determinado *pixel*, FVE permite a aquisição de um histograma das velocidades dos *spins* que compõem cada *pixel*. Isso é feito aplicando etapas adicionais de codificação de fase ao longo da dimensão de velocidade. Isso adiciona a dimensão de velocidade(k_v) ao espaço-k, demandando várias aquisições adicionais para percorrê-la.

Para mover as aquisições ao longo da dimensão de velocidade no espaço-k, gradientes bipolares são aplicado antes das aplicações dos gradientes de leitura espacial de cada aquisição. A aplicação de gradientes bipolares consistem em aplicar um gradiente, fazendo com que as frequências de precessão variem ao longo do eixo z por um curto período de tempo e depois aplicar o mesmo gradiente com sinal invertido ao longo do mesmo eixo para que as frequências retornem ao seu valor inicial. Isso faz

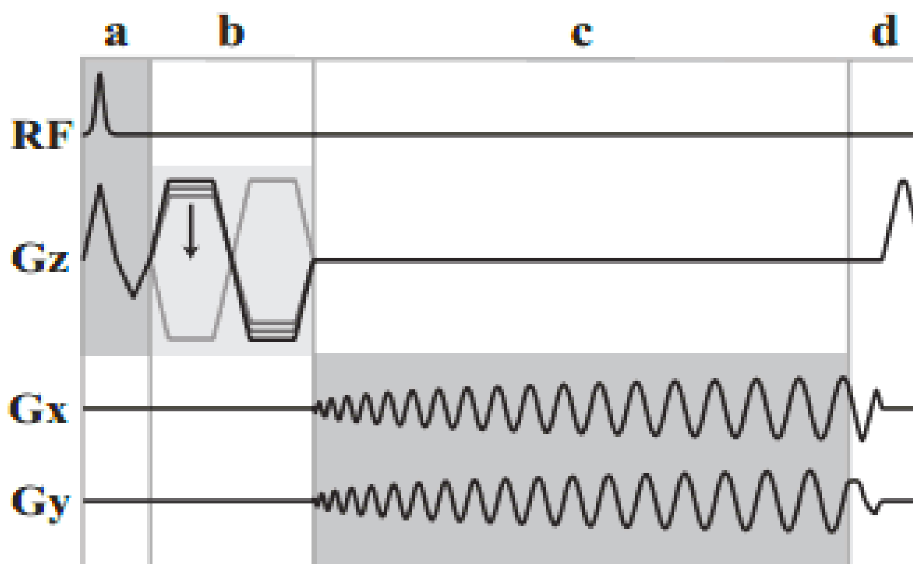


Figura 2.2: Sequência de pulsos para aquisição FVE espiral. (a) seleção do corte (b) gradiente bipolar para codificação de velocidade, que muda de intensidade para a codificação das velocidades (c) leitura espiral (d) gradientes de refocalização. [Adaptado de [8]]

com que surja uma diferença de fase ao longo do eixo do gradiente. Cada aquisição ao longo de k_v é chamada de uma codificação de velocidade. O número de codificações de velocidade em k_v depende da resolução de velocidades que se deseja alcançar e da faixa de velocidades que se quer medir.

A Fig. 2.2 mostra uma sequência de pulsos para aquisição em espiral com FVE. Na figura é apresentada a aquisição de uma codificação de velocidade, com as linhas mais claras e a seta indicando a variação da intensidade do pulso de 180° para cada codificação de velocidade.

2.6.3 Multi-dimensionalidade

Para o estudo de fluxo, ao invés de capturar simples imagens bidimensionais, temos que adquirir dados de velocidade. Além disso, geralmente adquirimos também vários cortes e vários quadros temporais. Também utilizamos informações de 4 bobinas de captação. Com isso os nossos dados são do tipo $M(k_x, k_y, z, k_v, t)$. A captura dos sinais pode ser feita utilizando mais de uma bobina de leitura, resultando em um sinal do tipo $M(k_x, k_y, z, k_v, t, b)$ em que b denota o número do canal.

O volume de dados para estudo do fluxo sanguíneo, por exemplo, é de aproximadamente 1GB de dados, para 32 codificações de velocidade, 5 cortes, 4 bobinas de captação, 43 quadros temporais e com resolução espacial de 115×115 *pixels*. Essa quantidade de dados gera implicações não só para o processamento, mas também para o armazenamento e para a transmissão.

2.6.4 Tempo de reconstrução

O tempo de reconstrução de uma imagem de ressonância magnética, mesmo que sem a utilização de algoritmos rápidos, não seria tão relevante se analisada de forma isolada. Contudo, apesar do impacto no tempo de reconstrução para apenas uma imagem não ser sensível para o usuário mesmo que leve alguns segundos, a quantidade de reconstruções necessárias para dados com tantas dimensões passa a ser bastante significativo. Caso a amostragem não seja feita em grade uniforme e não utilizemos técnicas aceleradas de reconstrução, a reconstrução da informação de fluxo sanguíneo do *dataset* de 1 GB mencionado anteriormente, pode levar até 52 horas em um computador com processador Intel Core i7 operando a 2,9 GHz e 8 GB de memória RAM executando a DrFT em Matlab.

Dessa forma, fica evidente a importância do desenvolvimento de algoritmos e ferramentas que possibilitem a reconstrução rápida das imagens de ressonância magnética, uma vez que são esperados incrementos nas resoluções de todas as dimensões dos dados adquiridos e, consequentemente, do volume de dados.

Capítulo 3

Processamento Paralelo em CUDA

Este capítulo se inicia com um pequeno histórico da tecnologia de processamento paralelo, juntamente com argumentos que compelem à migração das arquiteturas seriais para arquiteturas paralelas de processamento. São apresentados então os fundamentos da linguagem de programação CUDA.

3.1 Histórico

Em 1965, Gordon Moore criou o que se tornaria mais tarde conhecido como a Lei de Moore, em que predizia que o número de transistores em um circuito integrado dobraria a cada ano, estimativa que mais tarde foi revisada para 18 meses. O que muitos pensam hoje, contudo, é que a lei de Moore diz que o poder de processamento dos processadores dobraria a cada ano. O que aconteceu é que com mais transistores, os engenheiros conseguiram por vários anos melhorar o desempenho dos processadores através de mudanças e melhorias de uma mesma arquitetura. Estas melhorias foram em grande parte viabilizadas pela maior capacidade de integração dos dispositivos semicondutores. Foram desenvolvidas novas tecnologias que permitiram executar instruções em menos ciclos de processamento, aumento da velocidade das portas lógicas nos semicondutores, etc. Somando todos esses fatores, temos o ganho de desempenho de aproximadamente 52% ao ano para processadores *single-threaded* [10].

Contudo, este crescimento começou a diminuir para o patamar de 20% ao ano por volta de 2001 [10] e hoje gira em torno de 5% a 10%. A figura Fig. 3.1 mostra a evolução da razão entre MIPS (*million instructions per second*) e o *clock* dos processadores ao longo do tempo.

Vários outros fatores além da frequência de operação do processador contribuem para o seu desempenho, como velocidade das memórias e largura dos barramentos. São exatamente esses outros fatores que ainda vêm permitindo a evolução na capacidade

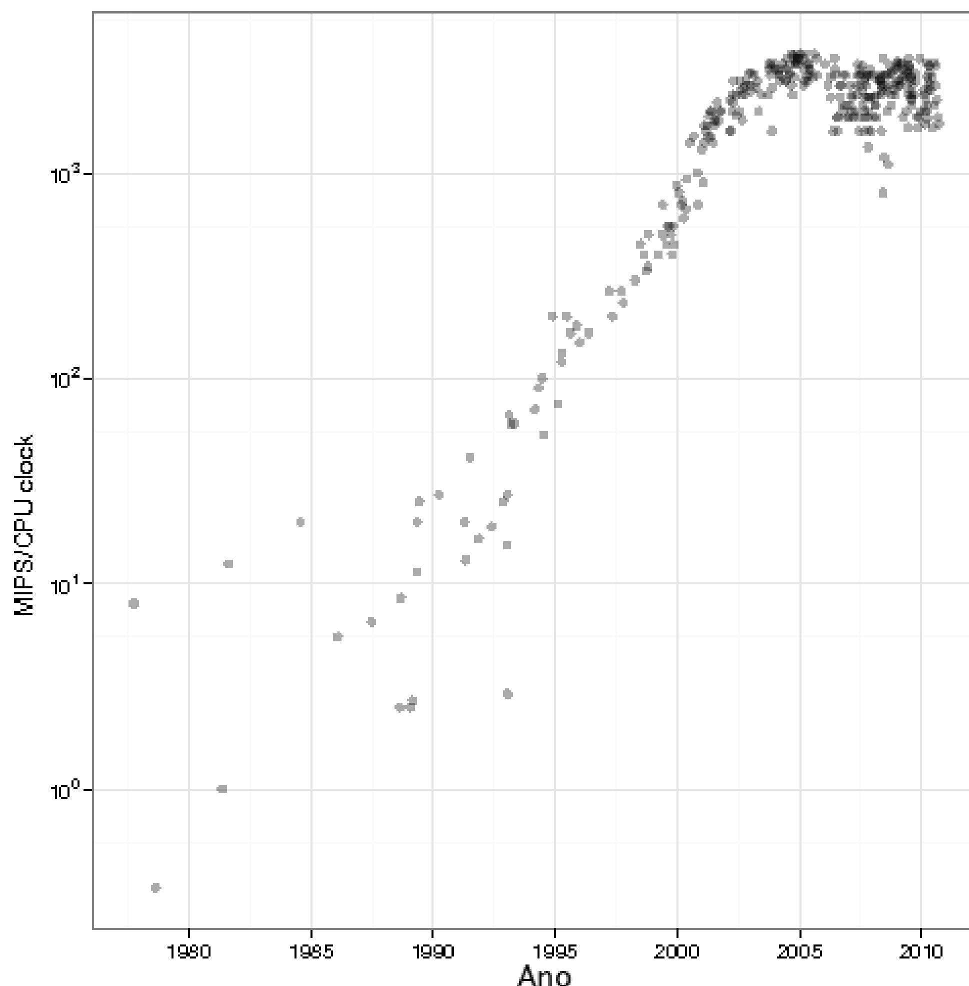


Figura 3.1: Crescimento do poder de processamento das CPUs. [Adaptado de [11]].

de processamento das CPUs nos últimos anos, já que a frequência de operação tem se mantido constante. Ainda assim, a frequência de operação é um dos principais fatores a serem considerados ao avaliar o desempenho de computadores.

Uma nova abordagem para o aumento do poder de processamento é o paralelismo, ou seja, utilizar vários processadores e distribuir o processamento para realizar a mesma tarefa em menos tempo.

Os primeiros produtos de massa a utilizar os princípios de processamento paralelo e memória hierárquica distribuída foram as placas de processamento gráfico (*graphics processing unit* - GPU) para computadores. A quantidade de cálculos necessários para a construção de cenas ou objetos em três dimensões se torna muito grande quando o número de polígonos cresce. Para tirar o peso destes cálculos da CPU, foram criadas placas dedicadas que criavam imagens a partir das informações dos vértices dos polígonos, cores e de imagens de textura. Estas placas utilizavam a arquitetura de processamento paralelo, mas como tinham uma finalidade muito específica, seus algoritmos eram definidos no próprio silício dos circuitos integrados. Isso permitia maiores

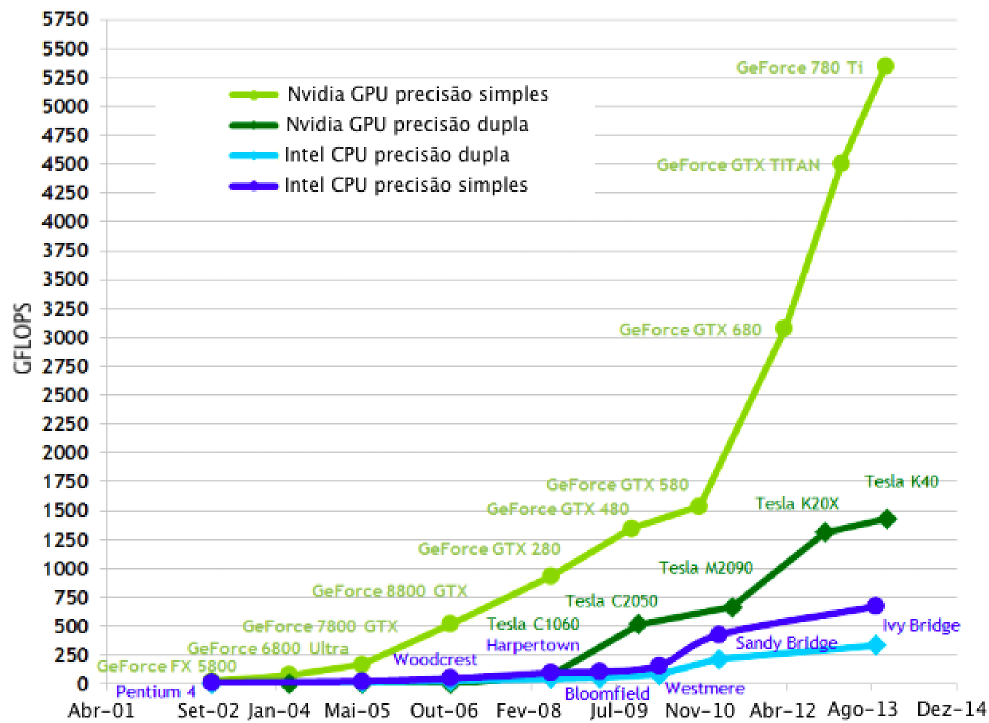


Figura 3.2: Crescimento do poder de processamento das GPUs comparado ao das CPUs (GFLOPS - Giga Floating point Operations Per Second). [Adaptado de [12]].

velocidades de operação, mas representava um problema evolutivo, uma vez que cada novo recurso ou algoritmo desenvolvido para a geração das imagens demandava um novo projeto de circuito.

Em paralelo a isso, alguns pesquisadores começaram a tentar utilizar o mecanismo interno das placas de vídeo para realizar operações que não estivessem relacionadas a gráficos tridimensionais, substituindo as texturas e polígonos enviados para a GPU por matrizes cuidadosamente criadas de forma que quando a mesma operasse nesses dados, o resultado fosse, por exemplo, a transformada de Fourier de uma matriz.

A própria indústria de placas gráficas viu o potencial para criar placas que utilizassem a arquitetura paralela, mas que permitissem uma programação dos algoritmos a serem executados. Dessa forma, novos algoritmos para computação gráfica podiam ser incorporados através de uma atualização de *drivers*. Isso acabou permitindo que a comunidade científica escrevesse seus próprios algoritmos genéricos para fins diversos utilizando a arquitetura paralela. Com isso nasceram as Unidades de Processamento Gráfico de Uso Geral (*General Purpose Graphics Processing Units* - GPGPUs). Com o tempo, outras funções computacionalmente intensas foram sendo delegadas de forma nativa para as GPGPUs, como a codificação e decodificação de vídeo.

Desde seu surgimento, o crescimento do poder de processamento das GPGPUs vem crescendo de forma bastante acelerada, aumentando cada vez mais a vantagem em relação às CPUs, como mostrado na Fig. 3.2.

Um dos principais fatores para a dificuldade em aceitação das arquiteturas paralelas é a necessidade de se escrever os algoritmos voltados especificamente para execução paralela. Durante vários anos os fabricantes de CPUs se esforçaram ao máximo para que códigos escritos para uma geração anterior de CPUs pudessem ser facilmente recompilados para serem executados em uma nova geração de CPUs, sem que fossem necessárias modificações, mas fazendo uso do novo poder de processamento. Vários recursos foram incorporados às CPUs para que além do ganho na frequência do *clock*, os programas pudessem ser executados mais rapidamente de forma transparente ao desenvolvedor. Estruturas como *caches* e *pipelines* tentam melhorar o desempenho do processador criando formas de manter tanto os barramentos de dados e de endereços, como a CPU ocupados, para acelerar a execução de programas e dos sistemas operacionais.

Contudo, para utilizar o processamento paralelo, os programas têm que ser escritos levando em consideração este novo paradigma. Com isso, deve-se investir muito tempo por parte dos desenvolvedores tanto para aprender esta nova tecnologia quanto para reescrever os programas. Este investimento de tempo é facilmente justificado, uma vez que algoritmos reescritos de forma paralela podem chegar a ser centenas de vezes mais rápidos que seus equivalentes em CPU.

Da mesma forma que para CPUs, o desempenho de algoritmos em GPU dependem do hardware onde o algoritmo é executado, assim como da habilidade do programador em escrever um código bem organizado e otimizado. Para a programação em GPGPU, contudo, existe um outro fator que tem bastante peso: o quanto um algoritmo é paralelizável. Algoritmos em que vários cálculos podem ser executados de forma descorrelacionada ou paralela apreciam uma grande aceleração quando escritos para GPGPU, ao passo que algoritmos onde existe uma característica muito serializada podem se beneficiar menos das capacidades das GPGPUs.

3.2 Arquitetura

A abordagem para criar circuitos integrados com grande poder de processamento paralelo não consiste em simplesmente aumentar o número de processadores, pois hoje um dos principais fatores limitantes para o aumento do número de processadores em um chip é a potência dissipada. Para atacar este problema, foi analisado como a potência é consumida em um processador para então estudar as possibilidades para minimizá-la. Dessa forma, a evolução não poderia mais ficar preocupada apenas com o desempenho final das unidades de processamento, mas também com a eficiência de cada uma.

Em um circuito integrado CMOS, o consumo de energia se dá primariamente nas transições de estado dos transistores, onde os transistores variam entre os estados ligado e desligado, ou conduzindo e não conduzindo. Quando ocorre uma transição

no terminal de controle do transistor, antes desse novo estado ser traduzido para sua saída, é necessário que exista um fluxo de elétrons para carregar as capacitâncias parasitas formadas entre os condutores que levam os sinais da saída de uma porta até as entradas dos transistores conectados. Essas capacitâncias crescem com as áreas dos capacitores. Como dentro de uma unidade lógica aritmética (ULA) as distâncias entre as portas são muito pequenas, também são pequenas as capacitâncias parasitas presentes nessas estruturas. Contudo, quando queremos mover os operandos ou os resultados, os sinais enfrentam uma capacitância que cresce linearmente com a distância a ser percorrida, pois estamos aumentando uma das dimensões da capacitância da linha.

Verificou-se que uma unidade lógica aritmética consome 50 pJ para realizar uma operação de multiplicação e acumulação em operandos de ponto flutuante de 64 bits e que a energia necessária para mover tais palavras de 64 bits (operandos e resultado) é de aproximadamente 25 pJ/mm dentro do chip. Caso os dados precisem ser movidos para fora do circuito integrado, o consumo chega a 1 nJ [10]. Assim, gasta-se muitas vezes mais energia movendo os operandos e os resultados do que propriamente realizando os cálculos. Essa discrepância é ainda maior se usarmos operandos de precisão simples ou inteiros, uma vez que a complexidade e o consumo da ULA diminuem, mas a energia para mover os dados não diminui. Para resolver esse problema foi necessário o desenvolvimento de arquiteturas que minimizassem a necessidade de mover os dados por longas distâncias.

Para suprir essas necessidades, foram criadas arquiteturas onde cada circuito integrado tem vários processadores, mas com memórias distribuídas e conectadas de forma hierárquica. Isso permite que um processador possa operar em um conjunto de dados e passar os resultados para outro processador através de uma memória compartilhada próxima aos mesmos. Isso evita que os resultados intermediários precisem passar por uma memória externa mais distante, diminuindo em muito a distância percorrida por estes dados e consequentemente a potência dissipada no processo.

Nas arquiteturas atuais, as unidades de processamento das GPUs são bem mais simples que nas CPUs tradicionais, mas aparecem em grande número e com uma estrutura de memória hierárquica, em que cada unidade de processamento conta com uma pequena quantidade de memória. Cada grupo de unidades de processamento conta com um pouco de memória compartilhada e a placa de vídeo conta com uma memória global acessível por todas as unidades de processamento, como ilustrado na Fig. 3.3. Existem também memórias com características de acesso diferenciadas, para atender determinados algoritmos que têm padrões de acesso à memória muito específicos. O tempo de acesso para cada uma destas memórias é diferenciado, sendo que as memórias mais próximas aos processadores são mais rápidas (e consomem menos energia) que as memórias mais distantes.

Um componente muito sofisticado nas arquiteturas atuais é o escalonador de ta-

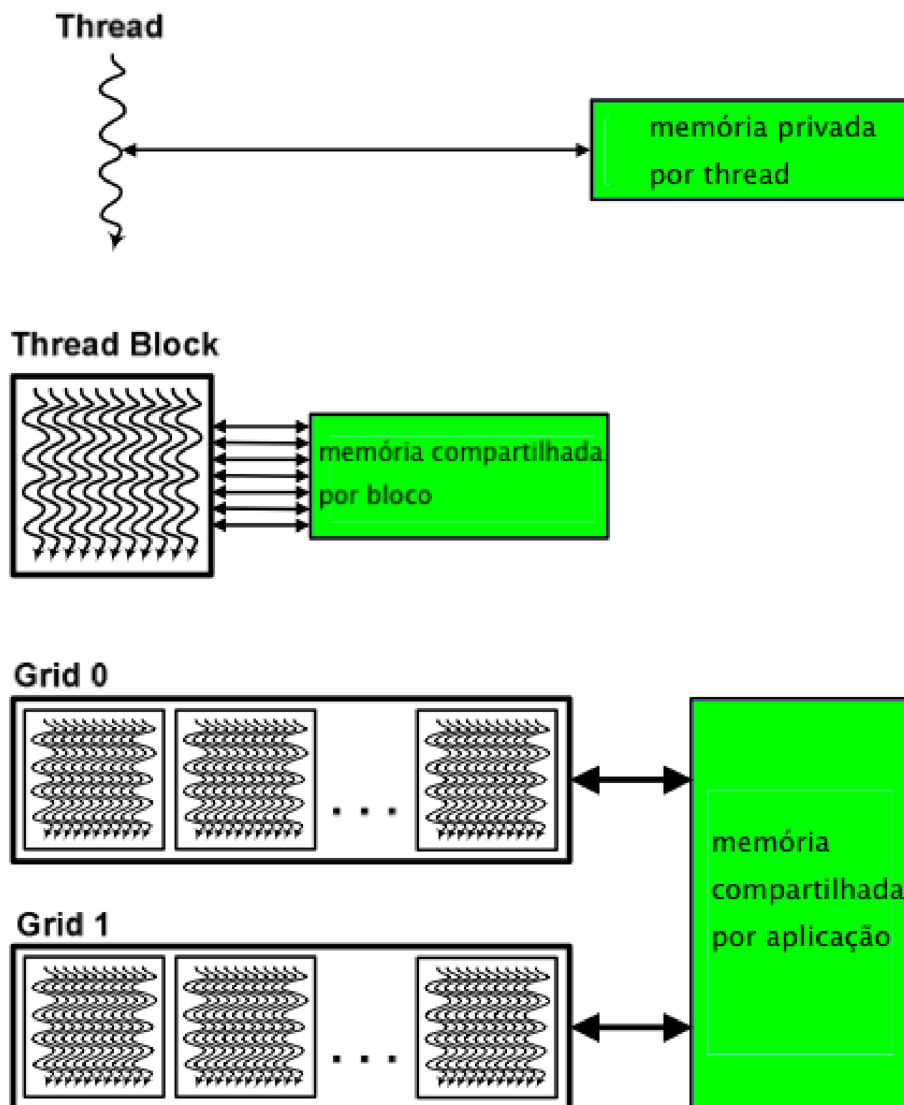


Figura 3.3: Modelo de memória hierárquica usada nas GPGPUs. [Adaptado de [12]].

refas. Ele é o responsável por gerenciar como os trechos de código, ou *kernels*, são distribuídos e executados pelas unidades de processamento. Em uma placa de processamento gráfico de uso geral, existe um escalonador para cada conjunto de um número determinado de unidades de processamento, ou *thread block*. Os escalonadores controlam a distribuição das tarefas para cada *thread* e também os passos de sua execução.

Várias arquiteturas de CPUs apresentam estruturas conhecidas como SIMD (*single instruction multiple data*), em que o processador consegue operar a mesma instrução em um conjunto de dados de tamanho definido, de forma simultânea. As GPGPUs funcionam com estruturas do tipo SIMT (*single instruction multiple threads*). Ao passo que em SIMD, o processador executa uma única instrução em múltiplos dados de forma paralela, em SIMT pode-se paralelizar programas inteiros, com operações,

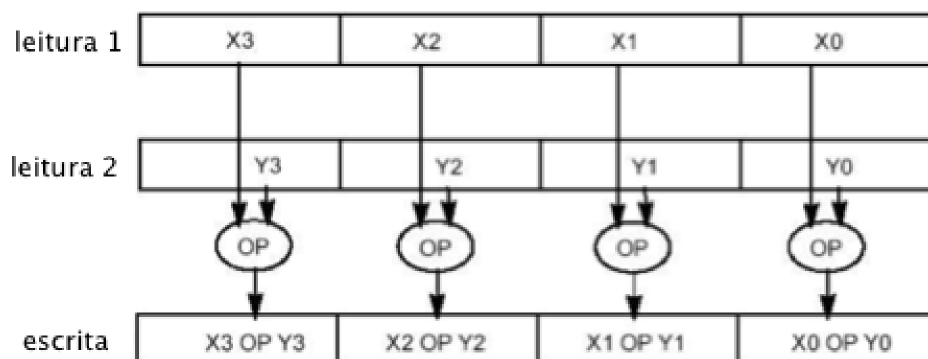


Figura 3.4: Operações executadas em paralelo

comparações e execução condicional.

Acessos de leitura podem ser feitos de forma simultânea por várias *threads*, uma vez que isso não traz prejuízo à integridade e à coerência dos dados. Do ponto de vista de projeto de circuito integrado, isso também não gera nenhum tipo de dificuldade técnica. Contudo, o acesso simultâneo para escrita é um fator crítico para o conceito de processamento paralelo. Sem nem ao menos nos preocupar com a implementação do circuito integrado, o próprio conceito de acesso paralelo para escrita já se mostra inviável. Se duas ou mais *threads* tentam escrever resultados diferentes em uma mesma posição de memória, não há como arbitrar o valor final a ser armazenado. Por isso, já na concepção dos algoritmos paralelos, os acessos de escrita devem ser pensados de forma a não causar conflitos. São admitidas escritas simultâneas nas memórias, porém não no mesmo endereço. As escritas paralelas devem acontecer como ilustrado na Fig. 3.4.

No caso de execuções condicionais que desviam o fluxo de um código, como, por exemplo, em um *if...else*, enquanto as *threads* que avaliaram a condição *if* como verdadeira executam o código correspondente, as *threads* que avaliaram a condição como falso ficam congeladas. Quando as *threads* que tomaram o ramo verdadeiro da condicional terminarem de executar as instruções correspondentes, estas ficam congeladas enquanto as demais *threads* executam as instruções relativas ao ramo falso da condicional. Esta coerência na execução só é necessária dentro de um *thread block*. *Thread blocks* diferentes podem executar códigos diferentes de forma simultânea, ficando o escalonador de maior nível responsável pelo agendamento de execução das *thread blocks*.

3.3 Definição da linguagem utilizada

Durante o processo evolutivo dos processadores paralelos, várias linguagens — ou simples extensões para outras linguagens — foram criadas para suportar estas arquiteturas, mas a primeira linguagem a se destacar nesse cenário, por acompanhar

um produto realmente massificado, foi a linguagem CUDA (*Compute Unified Device Architecture*), criada pela NVIDIA para suas placas de processamento gráfico. Com um produto de prateleira amplamente disponível, a adoção da linguagem foi rápida tanto no meio acadêmico quanto no meio comercial.

O segundo grande fabricante de placas gráficas, a ATI Technologies, mais tarde comprada pela AMD, lançou a linguagem ATI Stream, mas sua aceitação foi baixa. Mais tarde, já após sua aquisição pela AMD, as placas gráficas da ATI passaram defender o padrão OpenCL, proposto pela Apple Computer. O OpenCL hoje é suportado por praticamente todos os maiores fabricantes de CPUs e GPUs e representa para o processamento paralelo o que o openGL representa para a aceleração de gráficos em duas e três dimensões.

À época do início deste trabalho, a linguagem CUDA se apresentava como a mais madura e já contava com mais ferramentas, bibliotecas e disseminação na comunidade científica. Porém estas desvantagens vêm sendo reduzidas rapidamente e, hoje, o OpenCL tem apresentado grande adoção devido principalmente a sua natureza multiplataforma, estando presente, já há algum tempo, até mesmo em equipamentos embarcados como *smart phones* e *tablets*.

Em openCL, detalhes específicos das arquiteturas onde os códigos serão executados são transparentes para o programador. Este grau de abstração facilita a portabilidade do mesmo código para diversas plataformas de computação paralela diferentes. Existe, inclusive, a capacidade de distribuição do processamento em ambientes com processamento heterogêneo, paralelizando execuções entre CPUs e GPUs.

Originalmente, na linguagem CUDA o programador tinha mais controle sobre a arquitetura, podendo, por exemplo, definir como será o uso da memória hierárquica. Eventualmente, o OpenCL passou a suportar o uso de *flags* específicas para melhor aproveitar algumas características específicas do hardware.

3.4 Fundamentos da linguagem

A linguagem CUDA é, na verdade, uma extensão da linguagem C. Programas escritos em C ou C++ utilizam uma *Application Programming Interface*, ou API, para copiar dados para a GPGPU e configurar a execução das unidades de processamento. Os programas em C ou C++ dispara através das APIs a execução dos reais algoritmos paralelos na GPU. O compilador *nvcc* da NVIDIA traduz os códigos escritos para o processamento paralelo para o formato executável, que é carregado para a placa gráfica no momento da execução. Após a execução na GPGPU, o programa em C retoma o controle de execução e busca os resultados na GPGPU através da mesma API da NVIDIA.

Atualmente, as APIs para o acesso aos recursos de programação paralela estão dis-

poníveis também em outras linguagens. É possível fazer as transferências de memória, assim como configurar e executar os *kernels*, a partir de linguagens de mais alto nível, como Python ou até mesmo a partir do Matlab.

Os algoritmos que serão executados na GPGPU devem ser escritos na forma de *kernels*, que são basicamente os trechos de código que cada unidade de processamento executará em paralelo. Todas as unidades de processamento executam o mesmo código, apenas diferenciando os operandos. A linguagem de programação provê para os *kernerls* variáveis que aparecem com valores diferentes para cada uma das instâncias da execução. Essas variáveis são usadas para decidir, por exemplo, qual parte dos dados cada *kernel* deve processar e onde guardar os resultados.

O *kernel* é o código escrito, enquanto que e as instâncias desse código, enquanto executadas por uma unidade de processamento, são chamadas de *threads*. Cada *kernel* pode receber argumentos como valores ou ponteiros para memórias globais. A variável blockIdx.x disponibiliza o identificador do bloco da *thread* corrente, `threadIdx.x` possui o identificador da *thread* dentro do bloco e `blockDim.x` traz a dimensão total do bloco. Assim podemos calcular `blockIdx.x*blockDim.x+threadIdx.x` para identificar a posição da *thread* dentro da cadeia de processamento total. Cada uma dessas constantes podem ter valores em x, y e z. Esta separação em três dimensões é meramente lógica e serve para facilitar o trabalho do programador no momento da concepção dos algoritmos e da codificação dos *kernels*.

Para ilustrar este conceito, tomemos uma multiplicação de duas matrizes A e B, com o resultado em uma matriz C, toda com dimensões $N \times N$. Para calcular o resultado da multiplicação na coluna x, linha y, ou seja, na posição (x,y), precisamos multiplicar toda a linha y da matriz A por toda a coluna x da matriz B e somar os resultados. Para calcular o resultado na posição $(x,y+1)$, precisamos multiplicar toda a linha $y+1$ da matriz A por toda a coluna x da matriz B e somar os resultados. Podemos ilustrar esse processo conforme mostrado na Fig. 3.5.

Perceba que, do ponto de vista de acesso de leitura aos dados, essas operações são passíveis de serem executadas em paralelo, uma vez que ambos os cálculos acessam os mesmos valores simultaneamente (coluna x da matriz A). Os resultados intermediários (acúmulo das multiplicações das células) são guardados nas memórias locais a cada *thread*. Do ponto de vista de acesso de memória para escrita dos resultados finais, essas *threads* também podem ser executadas em paralelo, já que, ao final dos cálculos, cada *thread* escreve seu resultado em uma posição diferente da memória. Dessa forma, podemos atribuir o cálculo de cada valor da matriz C a uma unidade de processamento da nossa placa de processamento gráfico. Se nossa placa tem mais de $N \times N$ unidades de processamento, podemos realizar toda a multiplicação em paralelo. Caso nossa placa de processamento gráfico tem menos de $N \times N$ unidades de processamento, podemos calcular o resultado C por partes, como ilustrado na figura Fig. 3.6.

Segue abaixo o código de um *kernel* que executa a multiplicação discutida acima.

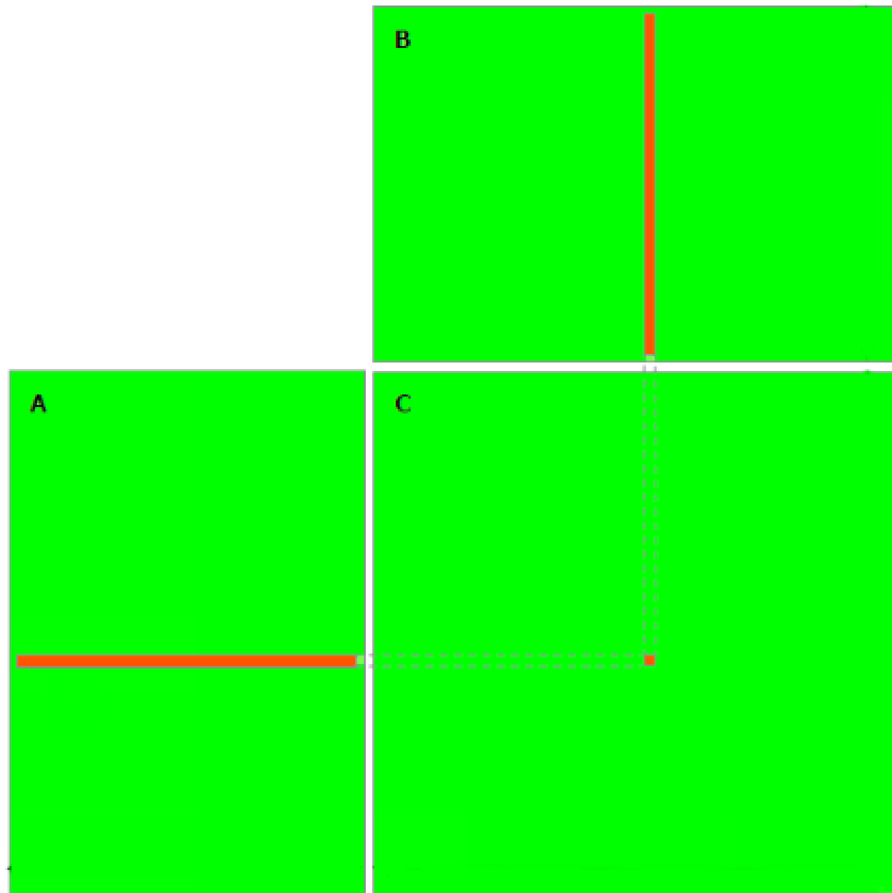


Figura 3.5: Cálculo de um resultado da matriz C. [Adaptado de [12]].

```
// Matrizes armazenadas linha a linha
// M(linha, coluna) = *(M.valores + linha * M.largura + coluna)
typedef struct
{
   int largura;
   int altura;
   int* valores;
} Matriz;

__global__ void MatMulKernel(Matriz A, Matriz B, Matriz C)
{ // cada thread calcula um elemento da matriz C acumulando os resultados na variavel
    tempC.
   int tempC = 0;
   int linha    = blockIdx.y * blockDim.y + threadIdx.y;
```

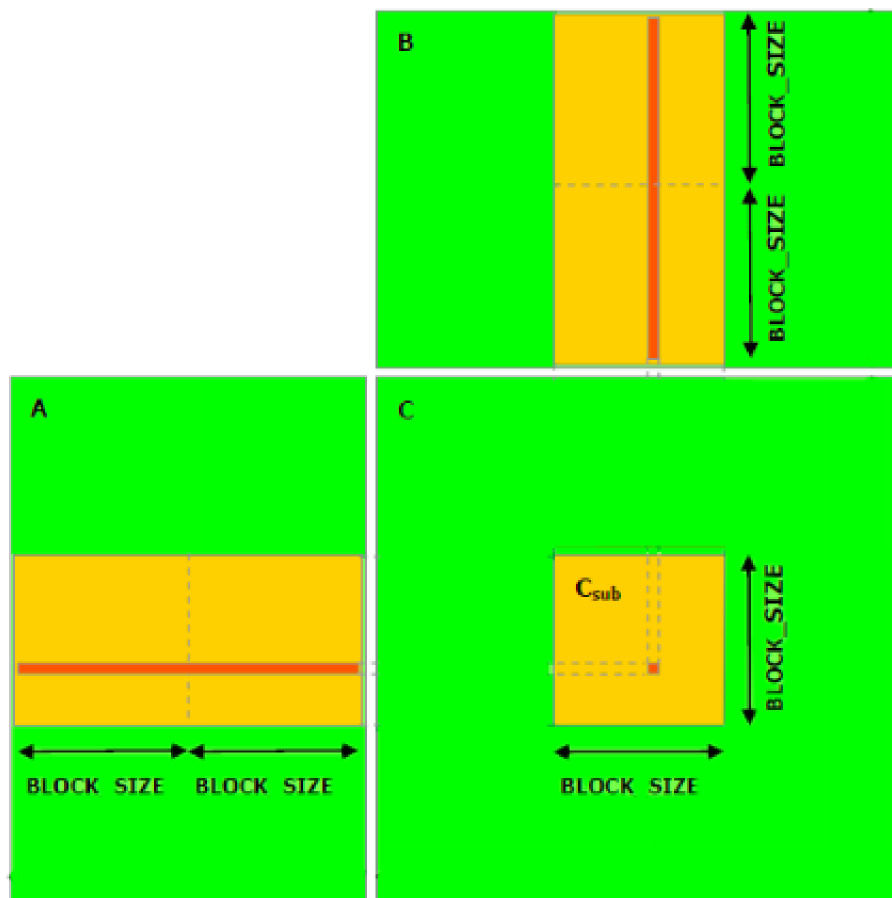


Figura 3.6: Cálculo de um conjunto de resultados da matriz C. [Adaptado de [12]].

```
    int coluna = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = 0; i < A.largura; i++)
    {
        tempC += A.valores[linha * A.largura + i] * B.valores[i * B.largura + coluna];
    }
    C.valores[linha * C.largura + coluna] = tempC;
}
```

Os dados a serem processados devem ser transferidos para a memória da GPGPU para poderem ser acessados pelas unidades de processamento. Esta transferência é realizada através de APIs específicas da linguagem. Da mesma maneira, os resultados são colocados na memória da GPGPU e devem ser resgatados para a memória da CPU através de API específica. Segue abaixo um exemplo de código em linguagem C para realizar as transferências de dados para a GPGPU, disparar a execução do processamento e recuperar os resultados de volta para a memória da CPU.

```
// Definicao do tamanho do bloco de threads
#define BLOCK_SIZE 16

// prototipo de funcao do kernel CUDA
__global__ void MatMulKernel(const Matriz, const Matriz, Matriz);

// Codigo da CPU
// para simplificar a ilustracao, as matrizes sao consideradas
// como tendo dimensoes multiplas de BLOCK_SIZE

void MatMul(const Matriz A, const Matriz B, Matriz C)
{
   // Carrega as matrizes A e B para a memoria da GPGPU
   Matriz d_A;
   d_A.largura = A.largura;
   d_A.altura = A.altura;
   size_t size = A.largura * A.altura * sizeof(int);
   cudaMalloc(&d_A.valores, size);
   cudaMemcpy(d_A.valores, A.valores, size, cudaMemcpyHostToDevice);

   Matriz d_B;
   d_B.largura = B.largura;
   d_B.altura = B.altura;
   size = B.largura * B.altura * sizeof(int);
   cudaMalloc(&d_B.valores, size);
   cudaMemcpy(d_B.valores, B.valores, size, cudaMemcpyHostToDevice);

   // Aloca memoria para a matriz C na GPGPU
   Matriz d_C;
   d_C.largura = C.largura;
   d_C.altura = C.altura;
   size = C.largura * C.altura * sizeof(int);
   cudaMalloc(&d_C.valores, size);

   // dispara a execucao do kernel
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
   dim3 dimGrid(B.largura / dimBlock.x, A.altura / dimBlock.y);
   MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

   // Le os resultados da matriz C de volta para a memoria da CPU
   cudaMemcpy(C.valores, Cd.valores, size, cudaMemcpyDeviceToHost);

   // libera as memorias alocadas na GPGPU
   cudaFree(d_A.valores);
   cudaFree(d_B.valores);
   cudaFree(d_C.valores);
}
```

Seria ainda possível realizar alterações no código para melhorar seu desempenho, fazendo uso mais intenso da memória compartilhada [12], para diminuir o tempo de acesso total a valores que são recorrentemente utilizados. Além disso, seriam necessárias alterações para o uso de matrizes com dimensões arbitrárias.

É possível utilizar os dados resultantes de uma etapa de processamento realizado na GPGPU como entrada para uma etapa seguinte de processamento paralelo, sem a necessidade de transferir esses resultados intermediários de volta para a CPU e novamente para a GPGPU. Isso permite melhorar a performance de algoritmos mais complexos, onde é necessário a execução de *kernels* diferentes para alcançar o objetivo final. Para a apresentação e visualização de grandes volumes de dados, estes podem ser mostrados diretamente da placa de vídeo, para visualização em três dimensões de um conjunto de dados de ressonância magnética, por exemplo.

Capítulo 4

Reconstrução em Cuda

Este capítulo discute a implementação dos algorítmos de reconstrução de imagens de ressonância magnética em linguagem de programação CUDA, fazendo uma análise dos resultados obtidos.

4.1 CUDA DrFT

Como discutido no Capítulo 2, os dados adquiridos por um equipamento de ressonância magnética estão no espaço-k, ou domínio de Fourier. Para obtermos imagens a partir dos dados adquiridos precisamos aplicar a transformada inversa de Fourier. Como os dados adquiridos são discretos, devemos utilizar, na verdade, a transformada discreta de Fourier inversa (*inverse discrete Fourier transform* - IDFT).

A IDFT é dada por:

$$s_n = \frac{1}{N}\sum_{k=0}^{N-1} S_k e^{i2\pi \frac{k}{N} n}, \tag{4.1}$$

onde s é a amostra reconstruída no domínio da imagem, S é a amostra no domínio de Fourier e N é o número total de amostras adquiridas.

Existem algoritmos rápidos para o cálculo das transformadas de Fourier. Tais algoritmos são chamados de *Fast Fourier Transforms* ou FFTs. Como a transformada inversa de Fourier pode ser expressada em termos da transformada direta, tais algoritmos também se aplicam à transformada inversa, com o nome de *Inverse Fast Fourier Transform* ou simplesmente IFFTs.

Tais algoritmos exploram propriedades da matriz da DFT para obter o mesmo resultado com um número muito menor de operações. Ao passo que a transformada, se calculada simplesmente pela fórmula, tem complexidade $O(N^2)$, as transformadas rápidas têm complexidade $O(N\log(N))$.

Alguns dos algoritmos mais conhecidos operam em conjuntos com uma quantidade

de dados que seja uma potência de 2. Entretanto, existem versões de algoritmos que operam em conjuntos de dados com quantidades quaisquer de amostras com o mesmo grau de complexidade computacional.

No caso de imagens, utilizamos a versão bidimensional da transformada inversa, que, para dados uniformemente amostrados em uma grade $N \times M$, é dada por:

$$s_{xy} = \frac{1}{NM} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} S_{ij} \cdot e^{i2\pi \frac{nx}{N}} \cdot e^{i2\pi \frac{my}{M}} \tag{4.2}$$

No caso de dados não uniformemente amostrados, com M amostras do espaço-k, a equação para a reconstrução é dada pela DrFT inversa:

$$s_n = \frac{1}{M} \sum_{m=0}^{M-1} W_m S_m e^{i2\pi[k_x(m)x(n)+k_y(m)y(n)]}. \tag{4.3}$$

Na equação acima, os termos W_m representam os fatores de ponderação das amostras, que são utilizados para compensar a densidade não uniforme das amostras (ponderação). Os fatores de ponderação são calculados fazendo um diagrama de Voronoi, que define regiões baseadas nas distâncias entre as localizações no espaço-k das amostras, e tomando o inverso do valor da área de cada célula resultante. Se a grade de amostragem não for uniforme, as simplificações exploradas pela FFT não mais são válidas e devemos realizar todas as operações da equação original para obter a transformada.

A reconstrução direta por Fourier é um algoritmo altamente paralelizável, uma vez que cada *pixel* s_{xy} da imagem pode ser reconstruído de forma totalmente independente dos demais *pixels* (do ponto de vista de acessos de escrita na memória). Isso favorece a utilização de GPGPUs para reconstrução paralela.

Em CUDA, escrevemos os chamados *kernels*, que são executados por cada unidade de processamento da GPGPU de forma independente. Cada instância em execução do *kernel* é chamada de *thread*, que têm a capacidade de saber suas “posições"em relação às demais *threads*, de forma que é possível determinar qual *pixel* cada *kernel* deve reconstruir. Como cada *pixel* é reconstruído de forma independente, cada *thread* escreve um *pixel* diferente na matriz de resultados, evitando conflitos de escrita. O acesso paralelo para leitura é permitido.

São criados tantas *threads* quantos *pixels* existirem na imagem reconstruída. A linguagem de programação é responsável por agendar a execução das *threads* nas unidades de processamento disponíveis na GPGPU, de forma que o desenvolvedor não precisa se preocupar com detalhes específicos do *hardware*, como o modelo de GPGPU que está sendo utilizado e o número máximo de *threads* simultâneas. O desenvolvedor paraleliza o algoritmo de uma forma que faça sentido do ponto de vista lógico e a linguagem de programação paraleliza isso de forma física na GPGPU. Isso permite

que a velocidade de execução dos algoritmos possa ser escalonada automaticamente quando utilizamos GPGPUs com mais unidades de processamento. Os *kernels* só operam em memória que seja interna à GPGPU; logo, todos os dados necessários para os cálculos devem ser copiados para a GPGPU antes de se iniciar o processamento e o resultado deve ser copiado para fora da GPGPU.

Para a reconstrução por DrFT, foi implementado um algoritmo onde os *kernels* são organizados logicamente em uma matriz de dimensões iguais às dimensões da imagem a ser reconstruída. O *kernel* calcula os coeficientes de Fourier que irá utilizar, baseando-se na posição do pixel a ser reconstruído no espaço x-y e nas posições das amostras no espaço k_x-k_y. As amostras são multiplicadas pelos pesos correspondentes às suas posições (ponderação) e, então, pelos coeficientes de Fourier. Todos os ensaios foram realizados em um computador com processador Intel Core i7 operando a 2,9GHz e 8GB de memória RAM com uma placa de processamento gráfico nVIDIA GTX570.

O código fonte abaixo traz uma versão simplificada do *kernel* criado para realizar a DrFT inversa de uma imagem.

```
__global__ ivoid drft( const float * datar, const float * datai, const float * kx,
    const float * ky, const int dataLength, const int Nim, float * mr, float * mi )
{
   // datar - parte real dos dados de entrada
   // datai - parte imaginaria dos dados de entrada
   // kx - posicao ao longo do eixo x das amostras
   // ky - posicao ao longo do eixo y das amostras
   // Nim - tamanho da imagem (lado)
   // mr - parte real do resultado
   // mi - parte imaginaria do resultado

   // definicao da posicao da thread atual dentro da imagem
   int x = (blockIdx.x * blockDim.x) + threadIdx.x;
   int y = (blockIdx.y * blockDim.y) + threadIdx.y;

   uint sample = 0; // variavel temporaria para controlar a posicao

    float tempValue;
    float tempCos; // valor temporario do coseno
    float tempSin; // valor temporario do seno
    int tempPos = x + y * Nim; // posicao linear do pixel
    float Nim2 = Nim/2; // meio tamanho da imagem

     mr[tempPos] = 0;
     mi[tempPos] = 0;

     for(sample = 0; sample < dataLength; sample++)
     {
```

Tabela 4.1: Tempo de reconstrução para o algoritmo CUDA DrFT

Resolução	Tempo médio (ms)
115×115	29
128×128	35
256×256	140
512×512	520

```
        tempValue = 2 * M_PI * ((kx[sample] * (x-Nim2)) + (ky[sample] * (y-Nim2))); //
            calculo de omega
        tempCos = cos(tempValue);
        tempSin = sin(tempValue);
        mr[tempPos] += (datar[sample] * tempCos) + (datai[sample] * tempSin); // parte
            real
        mi[tempPos] += (datai[sample] * tempCos) - (datar[sample] * tempSin); // parte
            imaginaria
    }
}
```

O código simplificado não mostra algumas otimizações como, por exemplo, o cálculo prévio dos coeficientes para a memória compartilhada como forma de otimizar os tempos de acesso a valores recorrentes. Também não são mostradas as verificações dos limites do tamanho da imagem, que impedem que *threads* calculem valores fora das dimensões da imagem.

A tabela 4.1 mostra os tempos médios (100 repetições) de reconstrução para imagens de diversas resoluções pelo código DrFT em CUDA. O tempo total de reconstrução para 32 codificações de velocidade, 5 cortes, 4 bobinas de captação, 43 quadros temporais e com resolução de 115×115 *pixels* utilizando a DrFT é de 18 minutos. Para referência, o tempo de reconstrução para os mesmos dados utilizando DrFT em Matlab foi de 52 horas.

4.2 CUDA *Gridding*

A reconstrução de imagens de ressonância magnética por *gridding* tenta aproveitar o ganho de esforço computacional dado pelo uso da FFT, convertendo os dados amostrados de forma não uniforme em dados uniformemente amostrados para, então, aplicação da IFFT.

Isso é feito através da interpolação bidimensional dos dados não uniformemente amostrados com um função de interpolação, também chamada de *kernel*, para uma grade uniforme. Obviamente, os dados da grade uniforme são aproximações que serão

tão boas quanto a qualidade da função de interpolação utilizada. Podem ser utilizados *kernels* triangulares, que simplificam os cálculos ao custo de pior qualidade de imagem resultante, assim como *kernels* do tipo Keiser–Bessel [13], que apresentam maior dificuldade computacional, mas com ganho na qualidade da imagem resultante.

Para a reconstrução por *gridding*, é necessária a compensação da densidade irregular original das amostras antes de convolução com o *kernel* de interpolação. Esta etapa é chamada de ponderação e minimiza os efeitos de que, em uma trajetória em espiral, existem geralmente muito mais amostras no centro do espaço-k do que em sua periferia. Como etapa final da reconstrução por gridding, é necessário corrigir o efeito causado pela convolução com o *kernel* de interpolação. Esta etapa é chamada de deapodização.

Do ponto de vista de paralelização, o processo de interpolação para uma grade uniforme incorre em um problema para a implementação, uma vez que os resultados (pontos na grade uniforme) sofrem influência da interpolação de mais de um ponto da trajetória não uniforme, como ilustrado na Fig. 4.1. Se pensarmos uma implementação do *kernel* em que cada *thread* fica responsável por calcular a influência na matriz final de um ponto da trajetória original, teremos conflito no acesso de escrita da memória, uma vez que várias *threads* podem contribuir para um mesmo ponto.

A aceleração em CUDA da etapa de convolução com os *kernels* de interpolação já é obra de estudo por parte da comunidade científica. Em 2008, Sorensen [15] mostrou os obstáculos envolvidos em otimizar o algoritmo para o uso do paralelismo das GPGPUs. Propôs metodologias para minimizar o problema de colisão de escrita, agrupando a interpolação dos pontos por região. Cada região é calculada de forma independente em memórias separadas e posteriormente as áreas de interferência são resolvidas. Este processo é ilustrado na Fig. 4.2. Ganhos de até $85\times$ na velocidade em relação à interpolação em CPU foram reportados.

Mais tarde, Gregerson [14] reportou ganhos de até $114\times$ em relação ao tempo de execução em CPU através de otimizações que tiravam melhor proveito da arquitetura das GPGPUs, como uso de memória compartilhada e calculo prévio do *kernel* de convolução.

Os algoritmos de FFT tentam acelerar o cálculo final utilizando estratégias para reduzir o número de operações necessárias, geralmente eliminando operações repetidas e combinando resultados. Esses algorítmos também são paralelizáveis, mas não totalmente. Utilizando uma forma de FFT que fatora recursivamente uma FFT de N amostras de comprimento em 2 FFTs de comprimento $N/2$, e assim por diante. Dessa forma, fica claro que o cálculo tem uma característica serial, onde uma etapa do processamento depende do resultado da etapa anterior, prejudicando a capacidade de paralelilzação do algoritmo.

Tendo em vista os trabalhos realizados anteriormente, este trabalho não se preocupou em repetir esses experimentos. A técnica de *gridding* é apresentada como

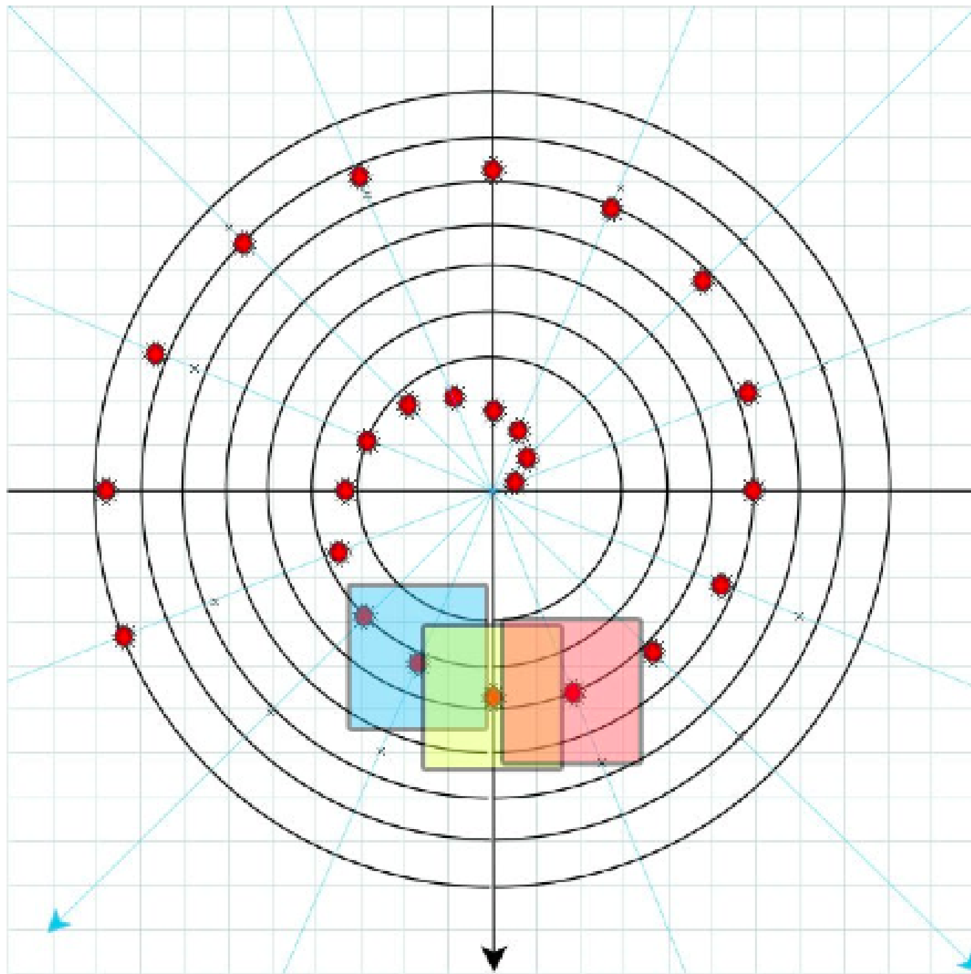


Figura 4.1: Áreas influenciadas pela interpolação de cada ponto da trajetória espiral. [Reproduzido de [14]].

fundamento para a técnica de NUFFT apresentada a seguir.

4.3 NUFFT

Com uma metodologia difundida principalmente por Fessler [4] [16], é possível melhorar a qualidade de imagens de ressonância magnética reconstruídas a partir de dados adquiridos em trajetórias não uniformes. Este método, conhecido como NUFFT (*non-uniform* FFT), otimiza os parâmetros do *kernel* de interpolação utilizado no *gridding* de maneira iterativa, através de técnicas de minimização de erro. A NUFFT foi bastante difundida, talvez pelo fato de que Fessler e seus contribuidores disponibilizaram uma biblioteca em Matlab com várias ferramentas de reconstrução que, entre outras coisas, continha as rotinas da NUFFT.

Imagens de referência reconstruídas utilizando a NUFFT apresentam erros mínimos, comparáveis aos erros de quantização dos dados. Neste trabalho a NUFFT é

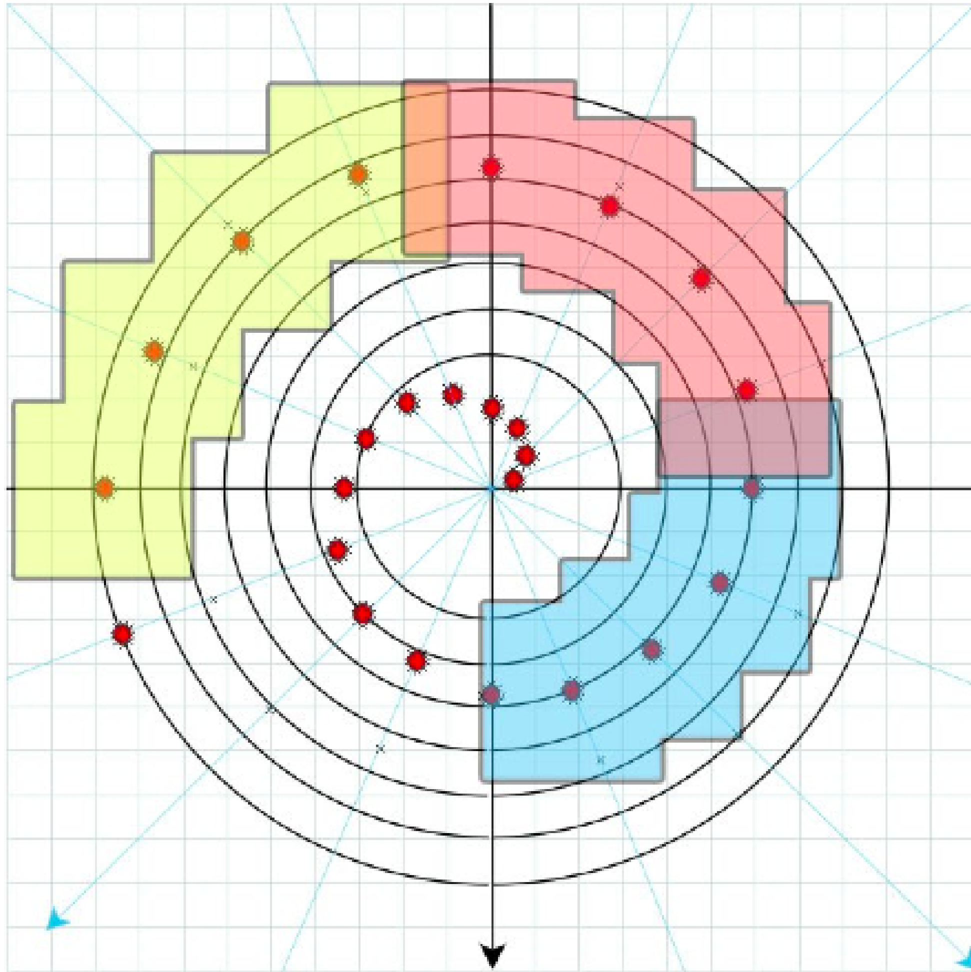


Figura 4.2: Agrupamentos das interpolações por região. [Reproduzido de [14]].

utilizada como base de comparação no que diz respeito a tempo de reconstrução. Não consideramos que faz sentido comparar o tempo de reconstrução com *gridding*, pois a NUFFT é *gridding* executado iterativamente para obter melhor qualidade de imagem. Faz sentido sim compará-la com DrFT, pois a DrFT, que é a solução analítica, também está preocupada com a qualidade final da imagem reconstruída.

O tempo total de reconstrução para 32 codificações de velocidade, 5 cortes, 4 bobinas de captação, 43 quadros temporais e com resolução de 115×115 *pixels*, utilizando o algoritmo NUFFT, disponibilizado para Matlab por Fessler, foi de 5 minutos.

4.4 Resultados e Discussão

Como mostrado neste capítulo, a reconstrução de imagens de maneira acelerada apresenta grande vantagem de tempo quando comparada com métodos não acelerados. Reconstruções analiticamente exatas com dias de duração afetam negativamente tanto as análises clínicas quanto o estudos destes dados no ambiente acadêmico. Com isso

também fica claro o motivo da técnica NUFFT, apresentando resultados com baixo erro e com tamanha aceleração.

A reconstrução por DrFT em CUDA, apesar de se mostrar mais lenta do que a NUFFT, ainda se mostra como alternativa viável para a reconstrução exata dos dados, dada a sua aceleração em relação ao método não acelerado.

Capítulo 5

Reconstrução de voxel único por DrFT em CUDA

Este capítulo discute a implementação de um algoritmo para reconstrução de pixel único utilizando DrFT em linguagem de programação CUDA. Os resultados obtidos são comparados com os resultados obtidos no capítulo anterior.

5.1 Proposta

Como discutido no capítulo 2, quando não adquirimos os dados em uma grade uniforme, ficamos impossibilitados de aplicar diretamente a FFT para a reconstrução dos mesmos. Como mostrado no capítulo 4, mesmo se utilizarmos aceleração por GPU, a reconstrução de toda a imagem por DrFT ainda é mais lenta que os demais métodos. Contudo, se o foco do nosso estudo é fluxo sanguíneo, temos regiões de interesse bem específicas, que englobam basicamente as artérias e veias.

Também como mostrado no capítulo 4, o algoritmo mais rápido e com os melhores resultados é a NUFFT. Este algoritmo, contudo, precisa reconstruir toda a imagem, ou seja, gasta-se bastante esforço computacional com reconstrução de dados referentes a tecidos estáticos e dados que não são de interesse clínico. Olhando diretamente para os *pixels*, podemos reconstruir as informações de fluxo diretamente dos dados amostrados, sem necessariamente passar pela etapa de reconstruir as imagens, conforme ilustrado na Fig. 5.1.

A DrFT, apesar de ser mais lenta para a reconstrução de toda a imagem, permite a reconstrução de *pixels* individuais. Apenas esse fato gera uma redução de esforço computacional proporcional à área (em *pixels*) da imagem reconstruída. Ou seja, para uma imagem de 115×115 *pixels*, a redução seria de 13.225 vezes. A redução é tão significativa que, mesmo que precisemos reconstruir dezenas de pontos, o esforço computacional ainda é muito menor.

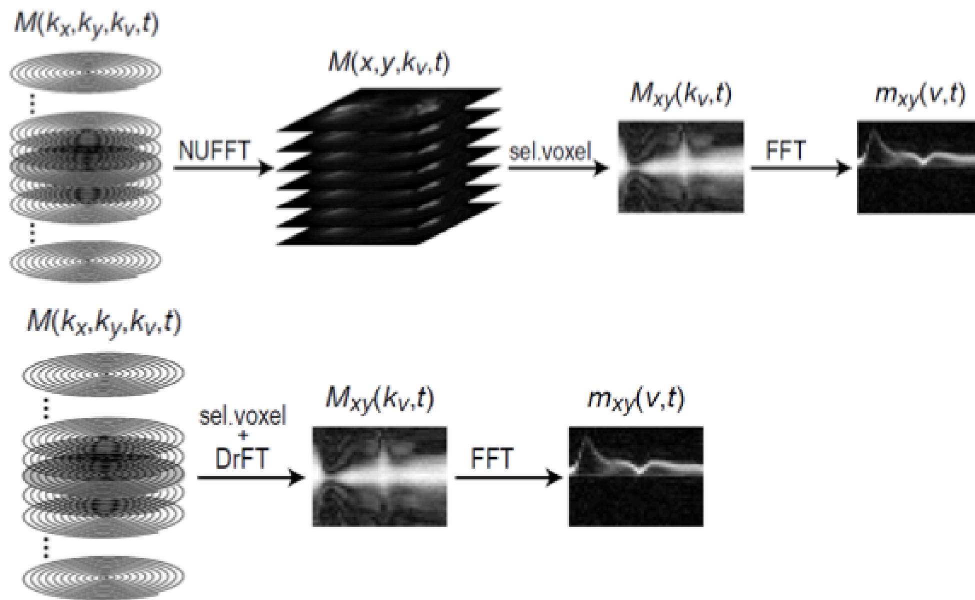


Figura 5.1: Comparação da reconstrução total e da reconstrução de voxel único.

5.2 Implementação

Como o ganho de esforço computacional se baseia na reconstrução de pontos de interesse, precisamos apresentar para o usuário pelo menos 1 imagem completamente reconstruída para que ele possa selecionar os *pixels* para a reconstrução dos dados de FVE espiral.

Foi implementado um programa onde uma imagem do corte intermediário e $k_v =$ 0s/cm é reconstruída e mostrada para o usuário utilizando DrFT. O usuário então seleciona na imagem o *pixel* de interesse. Um *kernel* CUDA é criado, em que cada *thread* realiza a reconstrução daquele *pixel* para todas as bobinas de cada quadro temporal de cada corte de cada codificação de velocidade utilizando a DrFT. É aplicada uma FFT ao longo de k_v para recuperar a informação de velocidade e o resultado é apresentado ao usuário. A figura Fig. 5.2 mostra exemplos de imagens geradas por este método.

O código do *kernel* utilizado para a reconstrução de *voxel* único é mostrado abaixo.

```
__global__ void drftphase( const float * datar, const float * datai, const float * kx,
    const float * ky, const int dataLength, const int Nim, const int xin, const int
    yin, const int nCoils, const int nphases, const int nVE, float * mr, float * mi )
{
   // datar, datai - parte real e imaginaria dos dados de entrada
   // kx,ky - coordenadas x e y da trajetoria
   // datalengh - numero de pontos de uma trajetoria espiral
   // Nim - dimensao da imagem de saida
   // xin, yin - coordenada x e y do ponto de interess
```

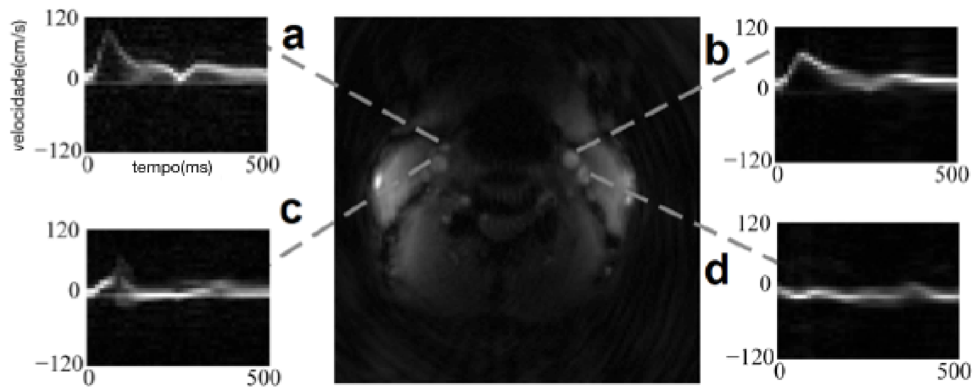


Figura 5.2: Geração de dados de velocidade de fluxo de maneira interativa. Ilustração de um corte axial do pescoço. (a), (c) - aorta (b), (d) - carótida.

```
    // nCoils - numero de bobinas
    // nphases - numero de phase encodings
    // nVE - numero de velocity encodings
    // mr, mi - saida real e imaginaria
    int c = threadIdx.x;
    int p = (blockIdx.x);
    int v = (blockIdx.y);

    float tempValue;
    float tempCos;
    float tempSin;
    float Nim2 = Nim/2;
    int sample;

    int sampleOut = (v * nCoils * nphases) + (p * nCoils) + c;
    int sampleIn = (((c * nVE * nphases) + (p * nVE) + v) * dataLength);
     for(sample = 0; sample < dataLength; sample++)
     {
         tempValue = 2 * M_PI * ((kx[sample] * (xin-Nim2)) + (ky[sample] * (yin-Nim2)));
         tempCos = cos(tempValue);
         tempSin = sin(tempValue);
         mr[sampleOut] += (datar[sampleIn] * tempCos) + (datai[sampleIn] * tempSin);
         mi[sampleOut] += (datai[sampleIn] * tempCos) - (datar[sampleIn] * tempSin);
         sampleIn++;
      }
}
```

Tabela 5.1: Comparativo de velocidade para DrFT de voxel único

	DrFT		NUFFT
	CPU	CUDA	CPU
Completo	52h	18 min.	5 min.
Voxel único	5s	135ms	

5.3 Resultados

A tabela 5.1 mostra o tempo de reconstrução da técnica proposta e compara com os tempos anteriores. Como podemos observar, o tempo de reconstrução é rápido suficiente para parecer instantâneo ao usuário.

A posição vazia da tabela 5.1 nos motivou a avaliar se não seria possível aplicar o paradigma de reconstrução de voxel único para acelerar a NUFFT. Uma possível metodologia é apresentada no capítulo 6.

Capítulo 6

Reconstrução de voxel único por gridding em CUDA

Este capítulo discute a implementação de um algoritmo de DFT (discrete Fourier transform) com reconstrução de pixel único, que poderia ser utilizado com gridding e NUFFT.

6.1 Introdução

A exemplo do que foi proposto no capítulo 5, este capítulo apresenta outra tentativa de acelerar a reconstrução somente dos *pixels* de interesse. O algoritmo de *gridding* tradicionalmente reconstroi toda a imagem, pois usa a FFT, que computa todos os pixels. Contudo, se o número de pixels de interesse for pequeno, podemos utilizar o algoritmo de Goertzel [17] para acelerar a etapa de reconstrução da imagem. A etapa de interpolação para a grade uniforme não seria acelerada por este novo método.

Goertzel propôs um algoritmo que explora as mesmas propriedades de periodicidade que a FFT explora, mas de forma direcionada para reconstrução de apenas um ponto da transformada de Fourier. Seu resultado corresponde exatamente à transformada de Fourier para aquele ponto

Para que a FFT possa ser realmente eficiente, presume-se o uso de tabelas pré-calculadas de seno e cosseno. Para a aceleração em GPU, a quantidade de memória requerida e os tempos de acesso envolvidos para essas tabelas afetam negativamente o desempenho do algoritmo. Se não usarmos tais tabelas, precisaríamos calcular senos e cossenos ao longo do processamento, o que traria uma penalidade de esforço computacional.

O algoritmo de Goertzel utiliza apenas 2 coeficientes que são constantes e pode ser executado de maneira iterativa. Para entradas reais, as operações necessárias para a execução do algoritmo também são reais, com exceção da última iteração. Essas características possibilitam que ele possa ser implementado em GPU de maneira

extremamente eficiente.

6.2 Algoritmo de Goertzel

Se tomarmos a equação para a DFT (*discrete Fourier transform*) do vetor x de N amostras,

$$X_k = \sum_{n=0}^{N-1} (W^k)^n x_n, \tag{6.1}$$

em que $W = e^{-i\frac{2\pi}{N}}$, os coeficientes W^k são obtidos amostrando as funções seno e coseno. Estes números complexos têm a propriedade de que, se ordenados na sequência inversa, obtemos os seus complexos conjugados. As potências negativas também produzem seus complexos conjugados. Utilizando ambas as propriedades ao mesmo tempo, obtemos exatamente a sequência inicial:

$$X_k = \sum_{n=0}^{N-1} (W^{-k})^{N-n} x_n. \tag{6.2}$$

Podemos então interpretar a equação 6.2 como tendo as potências de W como as potências em um polinômio, e os termos x como os coeficientes multiplicativos. Podemos então calcular o polinômio da seguinte forma:

$$X_k = (...(((W^{-k}x_0 + x_1)W^{-k} + x_2)W^{-k} + x_3... + x_{N-1})W^{-k} \tag{6.3}$$

Calculando a sequência de resultados intermediários y, temos:

$$\begin{aligned}
y_{-1} &= 0 \\
y_0 &= W^{-k}y_{-1} + x_0 \\
y_1 &= W^{-k}y_0 + x_1 \\
&... \\
y_{N-1} &= W^{-k}y_{N-2} + x_{N-1} \\
y_N &= W^{-k}y_{N-1}
\end{aligned} \tag{6.4}$$

O resultado y_N é o k-ésimo coeficiente da DFT, isto é, $X_k = y_N$.

Todo o processo pode ser re-escrito de forma iterativa segundo:

$$y_n = \begin{cases} W^{-k}y_{n-1} + x_n, & n = 0, ..., N-1 \\ W^{-k}y_{n-1}, & n = N. \end{cases} \tag{6.5}$$

Para $n = 0, ..., N-1$, a iteração pode ser escrita na forma de função de transferência no domínio da transformada Z, onde o fator z^{-1} representa o atraso de 1 amostra no tempo, conforme:

$$\frac{Y(z)}{X(z)} = \frac{1}{1 - W^{-k}z^{-1}} \tag{6.6}$$

Multiplicando o numerador e o denominador pelo conjugado do denominador, multiplicando os dois lados da equação por $X(z)$ e incorporando a multiplicação para o termo y_N, temos a equação de Goertzel:

$$Y(z) = \left[1 - W^{-k}z^{-1}\right]\left[\frac{1}{1 - 2\text{Re}(W^{-k})z^{-1} + z^{-2}}\right]X(z). \tag{6.7}$$

Com isso, o processo é separado em duas partes cascateadas:

$$Y(z) = \left[1 - W^{-k}z^{-1}\right]S(z), \tag{6.8}$$

em que

$$S(z) = \left[\frac{1}{1 - 2\text{Re}(W^{-k})z^{-1} + z^{-2}}\right]X(z). \tag{6.9}$$

A parte que é aplicada à sequência de entrada usa apenas valores reais (6.9). Além disso, um dos coeficientes é transformado em 1, eliminando metade das multiplicações e o outro coeficiente —2Re(W^k)— não muda, eliminando a necessidade de coeficientes pré calculados como na FFT ou na DFT. Outro fator importante a considerar é que o número de amostras não precisa ser uma potência de 2.

Note que a eq. 6.7 pode ser vista como uma filtragem digital. O segundo estágio do filtro (6.8) é calculado apenas uma vez e produz uma saída complexa. A representação visual do filtro Goertzel é mostrada na Fig. 6.1.

O algoritmo de Goertzel calcula a sequência $s(n)$, dada a sequência $x(n)$, conforme mostrado na Eq. 6.10.

$$s_k(n) = x(n) + 2\cos(\frac{2\pi k}{N})s_k(n-1) - s_k(n-2) \tag{6.10}$$

O único valor de $y(n)$ que realmente nos interessa é o valor final y_N, então buscamos calcular:

$$X_k = y(N) = s_k(N) + s_k(N-1)W^{-k}, \tag{6.11}$$

calculado supondo $x(N) = 0$, uma vez que $x(n)$ é definido apenas para $n = 0, ..., N-1$.

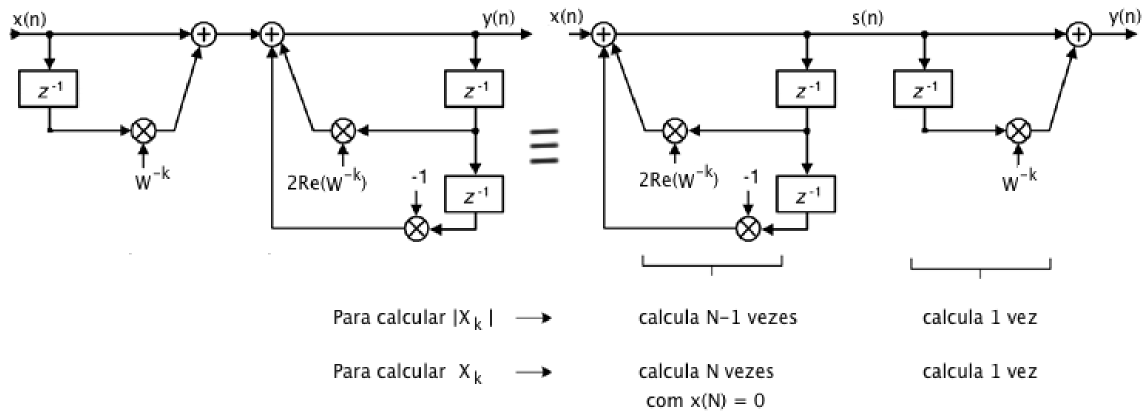


Figura 6.1: Representação do Filtro Goertzel

6.3 Implementação

Se tomarmos uma imagem de dimensões $M \times N$, é necessário executar o algoritmo de Goertzel $M \times N$ vezes na direção x para então executar $M \times N$ vezes na direção y, totalizando $2NM$ execuções do algoritmo. Para reconstruir apenas um ponto, precisamos executar o algoritmo na direção x apenas para a linha de interesse, para então executar o algoritmo apenas uma vez na direção y, ou seja, o total de execuções cai para $N+1$ execuções ficando a razão (r) entre o número de execuções necessárias para calcular toda a imagem e apenas um ponto

$$r = \frac{2NM}{M+1} \tag{6.12}$$

e se $M = N$, temos

$$r = \frac{2N^2}{N+1} \approx 2N. \tag{6.13}$$

Para avaliar o desempenho desta técnica, foram implementados em linguagem CUDA os seguintes *kernels*:

```
__global__ void cudagoec( const float * datar, const float * datai, const int dimC,
    const int dimL, float * mr, float * mi )
{
    // datar, datai - parte real e imaginaria dos dados de entrada
    // dimX - dimensao X e Y da imagem
    // mr, mi - saida real e imaginaria

    int l = threadIdx.x;
    int c = blockIdx.x;

    float w = ((float)c)/dimC;
```

```
    float coef = 2 * cos (2 * M_PI * w);
    float coef2 = sin(2 * M_PI * w);
    float saidar[3] = {0,0,0};
    float saidai[3] = {0,0,0};

    int sample;

    int sampleOut = (c*dimL) + l;
    int sampleIn = l;
    for(sample = 0; sample < dimC; sample++)
    {
        saidar[0] = datar[sampleIn] + coef * saidar[1] - saidar[2];
        saidar[2] = saidar[1];
        saidar[1] = saidar[0];

        saidai[0] = datai[sampleIn] + coef * saidai[1] - saidai[2];
        saidai[2] = saidai[1];
        saidai[1] = saidai[0];

        sampleIn += dimL;
    }

    coef = coef/2;
    mr[sampleOut] = ((coef * saidar[1]) - saidar[2]) - (coef2 * saidai[1]);
    mi[sampleOut] = ((coef * saidai[1]) - saidai[2]) + (coef2 * saidar[1]);
}

__global__ void cudagoel( const float * datar, const float * datai, const int dimC,
    const int dimL, float * mr, float * mi )
{
    int l = threadIdx.x;
    int c = blockIdx.x;

    float w = ((float)l)/dimL;
    float coef = 2 * cos (2 * M_PI * w);
    float coef2 = sin(2 * M_PI * w);
    float saidar[3] = {0,0,0};
    float saidai[3] = {0,0,0};

    int sample;

    int sampleOut = (c*dimL) + l;
    int sampleIn = c*dimL;
    for(sample = 0; sample < dimL; sample++)
    {
        saidar[0] = datar[sampleIn] + coef * saidar[1] - saidar[2];
        saidar[2] = saidar[1];
```

```
        saidar[1] = saidar[0];

        saidai[0] = datai[sampleIn] + coef * saidai[1] - saidai[2];
        saidai[2] = saidai[1]
        saidai[1] = saidai[0];

        sampleIn += 1;
    }

    coef = coef/2;
    mr[sampleOut] = ((coef * saidar[1]) - saidar[2]) - (coef2 * saidai[1]);
    mi[sampleOut] = ((coef * saidai[1]) - saidai[2]) + (coef2 * saidar[1]);
}
```

6.4 Avaliação do algoritmo proposto

Assim como na FFT em duas dimensões, o algorítmo de Goertzel calcula a transformada de todos os pontos em uma direção para então calcular na direção perpendicular. Um *kernel* calcula o algoritmo de Goertzel ao longo das linhas (`cudagoel`) e o outro ao longo das colunas (`cudagoec`). Com estes *kernels*, foram feitos alguns ensaios para avaliar o desempenho do algoritmo. Foram feitas reconstruções completas de imagens de resoluções diferentes, isto é, tamanhos diferentes de matrizes no espaço-k, comparando seu desempenho com os desempenhos da FFT do Matlab e da FFT acelerada por CUDA, também do Matlab. Para cada resolução de imagem estudada, foram realizadas 100 reconstruções.

Tabela 6.1: Comparativo de tempo entre FFT, GPUFFT e GPU Goertzel (valores em ms).

Tamanho	FFT		GPU FFT		GPU Goertzel (img. completa)		GPU Goertzel (voxel único[a])	
	médio	máximo	médio	máximo	médio	máximo	médio	máximo
115×115	1	3,5	1,3	4	0,5	3	$2,17 \cdot 10^{-3}$	$13,0 \cdot 10^{-3}$
128×128	0,4	3	0,6	3	0,6	3	$2,34 \cdot 10^{-3}$	$13,0 \cdot 10^{-3}$
1000×1000	17	23	11	13	74	77	$37,0 \cdot 10^{-3}$	$38,5 \cdot 10^{-3}$
1024×1024	18	24	10	14	76	81	$37,2 \cdot 10^{-3}$	$39,6 \cdot 10^{-3}$

[a] calculado dividindo os tempos da imagem completa por $2N$.

A tabela 6.1 mostra que o tempo médio de reconstrução total de uma imagem utilizando o algoritmo de Goertzel só foi menor que as outras opções de reconstrução para imagens de resolução baixa e não potência de 2.

A mesma tabela também mostra que para imagens muito pequenas, a aceleração

da FFT em GPU é prejudicada provavelmente pelo *overhead* das transferências de e para a GPGPU, apresentando tempos totais maiores para as reconstruções.

A tabela também mostra o prejuízo no desempenho da FFT, quando executado tanto da CPU quanto na GPGPU, quando a resolução da imagem não é potência de 2.

Se assumirmos imagens quadradas, para reconstruir toda uma imagem de lado N, utilizando o algorítmo de Goertzel é necessário executá-lo N^2 vezes para calcular a transformada de Fourier de todos os *pixels* em um sentido, para então executá-lo mais N^2 vezes para calcular a transformada de Fourier de todos os *pixels* no sentido perpendicular, totalizando $2N^2$ execuções. Com o algoritmo de Goertzel podemos reconstruir apenas um ponto, reduzindo o tempo total. Isso é feito calculando a transformada de todos os pontos em uma das direções (N execuções) para então calcular apenas um ponto na outra direção, totalizando $N + 1$ execuções, resultando em um ganho de $2N^2/(N + 1)$ que é aproximadamente 2N vezes na quantidade de execuções do algorítmo de Goertzel. Os resultados apresentados na última coluna da tabela representam este ganho.

Capítulo 7

Conclusões

A utilização de GPGPUs em reconstrução de imagens de ressonância magnética apresenta um futuro promissor. O requisito de alto poder de processamento, principalmente em ressonâncias tridimensionais e com a inclusão de informações de velocidade do fluxo sanguíneo, e o alto grau de paralelismo dos algoritmos utilizados, fazem desta solução uma opção muito interessante para esse tipo de aplicação.

Atualmente, já existem placas com até 4096 unidades de processamento, sendo também possível adicionar mais de uma placa por computador. Esse número tende a crescer, o que justifica ainda mais esse tipo de abordagem. Se fossem utilizadas duas GPGPUs com 4096 unidades de processamento cada uma, esperamos que o tempo de reconstrução completa para dados de *spiral* FVE resolvidos no tempo por DrFT já seria equivalente ou menor que o tempo de reconstrução por NUFFT em CPU. Além do mais, também é possível utilizar *clusters* com várias CPUs e GPUs para se obter um poder de processamento paralelo ainda maior.

Ademais, a metodologia proposta para reconstrução parcial do volume de dados utilizando DrFT em CUDA trazem resultados praticamente instantâneos, e poderiam ser aplicados em visualizações em tempo real, possibilitando a criação de novos procedimentos que seriam anteriormente inviáveis.

Foram mostrados também os ganhos que o uso do algoritmo de Goertzel pode trazer e sua adequação ao uso em GPGPUs para a reconstrução rápida de imagens e, possivelmente, em outros campos. A sua utilização se mostrou vantajosa na reconstrução de imagens completas quando o número de amostras é pequeno e não é uma potência de 2, caso em que a FFT ainda conta com vantagens de desempenho. Também se mostrou vantajoso o seu uso em situações onde não é necessária realizar a transformada de todo o conjunto de dados.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] TANG, D. D. B. C.; PARKER, D. L. Accuracy of phase-contrast flow measurements in the presence of partial-volume effects. *Journal of Magnetic Resonance Imaging*, v. 3(3):377-385, 1993.

[2] FRAYNE, R.; RUTT, B. K. Measurement of fluid-shear rate by fourier-encoded velocity imaging. In: *Magnetic Resonance Medicine*. [S.l.: s.n.], 1995. p. 34(3):378–387.

[3] CARVALHO, J. L. A.; NAYAK, K. S. Rapid quantitation of cardiovascular flow using slice-selective fourier velocity encoding with spiral readouts. In: *Magnetic Resonance Medicine*. [S.l.: s.n.], 2007. p. 57(4):639–646.

[4] FESSLER, J. A.; SUTTON, B. P. Nonuniform fast fourier transforms using min-max interpolation. In: *IEEE Transactions on Signal Processing*. [S.l.: s.n.], 2003. p. 51(2):560–574.

[5] GLOVER, G. H.; PELC, N. J. A rapid-gated cine mri technique. In: *Magnetic Resonance Annu*. [S.l.: s.n.], 1988. p. 299–333.

[6] MAEDA K. SANO, T. Y. A. Reconstruction by weighted correlation for mri with time-varying gradients. In: *IEEE Transactions on Medical Imaging*. [S.l.: s.n.], 1988. p. 7(1):26–41.

[7] SCHOMBERG, H.; TIMMER, J. The gridding method for image reconstruction by fourier transformation. In: *IEEE Transactions on Medical Imaging*. [S.l.: s.n.], 1995. p. 14(3):.

[8] CARVALHO J. F. NIELSEN, K. S. N. J. L. A. Feasibility of in vivo measurement of carotid wall shear rate using spiral fourier velocity encoded mri. *Magnetic Resonance in Medicine*, v. 63(6):1537-1547, 2010.

[9] MORAN, P. A flow velocity zeugmatographic interface for nmr imaging in humans. In: *Magnetic Resonance Imaging*. [S.l.: s.n.], 1982. p. 1:197–203.

[10] DALLY, W. J. The end of denial architecture and the rise of throughput computing. In: *Async Conference UNC*. [S.l.: s.n.], 2009.

[11] GILLESPIE, C. *CPU and GPU Trends Over Time*. Disponível em http://csgillespie.wordpress.com/2011/01/25/cpu-and-gpu-trends-over-time/, acessado em junho/2014.

[12] NVIDIA. *CUDA C Programming Guide*. Disponível em http://docs.nvidia.com/cuda/cuda-c-programming-guide/, acessado em junho/2014.

[13] KAISER, R. W. S. J. F. "on the use of the i0-sinh window for spectrum analysis". In: *IEEE Transactions on Acoustics, Speech and Signal Processing*. [S.l.: s.n.], February 1980. p. 28(1):105–107.

[14] GREGERSON, A. *Implementing Fast MRI Gridding on GPUs via CUDA*. [S.l.], 2008. NVIDIA Tech. Rep. on Med. Imag. Using CUDA.

[15] SORENSEN T. SCHAEFFTER, K. O. N. T. S.; HANSEN, M. S. Accelerating the nonequispaced fast fourier transform on commodity graphics hardware. In: *Medical Images, IEEE Transactions*. [S.l.: s.n.], 2008. p. 538–547.

[16] FESSLER, J. A. On nufft-based gridding for non-cartesian mri. In: *Journal of Magnetic Resonance*. [S.l.: s.n.], 2007. p. 191–195.

[17] GOERTZEL, G. An algorithm for the evaluation of finite trigonometric series. In: *American Mathematical Monthly*. [S.l.: s.n.], 1958. p. 34–35.

[18] JACKSON C. H. MEYER, D. G. N. J. I.; MACOVSKI, A. Selection of a convolution function for fourier inversion using gridding. In: *IEEE Transactions on Medical Imaging*. [S.l.: s.n.], September 1991. p. 10(3):473–478.

[19] MONTIEL, J. A. C. J. M. M.; NEIRA, J.; TAROS, J. D. Sensor influence in the performance of simultaneous mobile robot localization and map building. In: CORKE, P.; TREVELYAN, J. (Ed.). *Experimental Robotics VI*. [S.l.]: Springer Verlag, Inc., 2000. p. 287–296.

[20] LJUNG, L. *System Identification - Theory for the User*. [S.l.]: Prentice-Hall, 1999.

[21] BORGES, G. A. *Acquisition et datation de donees en robotique mobile: une etude de cas utilisant RTX*. Octobre 2000. GDR ISIS OT 6.3 : Journelligents, ENST, Paris.

[22] SWAROOP, R. et al. *BIVARIATE NORMAL, CONDITIONAL AND RECTANGULAR PROBABILITIES: A COMPUTER PROGRAM WITH APPLICATIONS*. [S.l.], 1980. Disponible en ligne sur *www.dfrc.nasa.gov/DTRS/1980/citation.html*.

[23] JULIER, S. J. *Process Models for the Navigation of High-Speed Land Vehicles*. Tese (Doutorado) — University of Oxford, Robotics Research Group, Department of Engineering Science., 1997.

[24] GROUP, C. I. W. *A culminating advance in the theory and practice of data fusion, filtering, and decentralized estimation.* Disponível em http://www.ait.nrl.navy.mil/people/uhlmann/CovInt.html, acessado em junho/2014.

[25] UHLMANN, J. K.; JULIER, S. J.; CSORBA, M. Nondivergent simultaneous map-building and localization using covariance intersection. In: SPEIGLE, S. A. (Ed.). *Navigation and Control Technologies for Unmanned Systems II.* [S.l.]: SPIE, 1997. p. 2–11.

ANEXOS

```
% RECONSTRUCAO ITERATIVA EM MATLAB

clear,clc,close all

slice = 4; % slice a ser reconstruido e apresentadoao usuario para selecao do pixel

% Configura o algoritmo de NuFFT do Fessler
cd fessler
setup
cd ..

% configura os parametros dos dados brutos
rawpath = './rawdata/';
nslices = 5;
ncoils = 4;

% carrega os dados dos arquivos para a memoria
filename = sprintf('%sslice%d.7',rawpath,slice);
[rawdata,usercv,hdr] = rawloadHD_jfn(filename,[],[],[], 1, [], [], []);
maxveloc       = usercv(1);  % valor maximo para a velocidade (velocity FOV/2)
optr           = usercv(2);  % duracao devTR (uS)
nphases        = usercv(4);  % numero de fases cardiacas (frames temporais)
nVE            = usercv(7);  % numero de etapas de codificacao de velocidade
nread  = usercv(8);  % numero de amostras
nintl          = usercv(14); % numero de espirais

% carrega os parametros espaciais da trajetoria espiral
kfile='recon16cm14mm8intl4vd';
spatfov = 160;
spatres = 1.4;
[kxkytraj kxkyweights] = kkread(kfile,nintl,nread);
kxkytraj = kxkytraj / (2*max(abs(kxkytraj(:)))); % normaliza as posicoes entre -0.5 e
    0.5
kxkyweights=kxkyweights/max(kxkyweights(:))/sqrt(2); % normaliza os valores de
    densidade entre 0 e sqrt(2)/2

% INICIALIZA O ALGORITMO DE NUFFT
tic
disp('nufft_init...')
pixels = ceil(spatfov/spatres);
Nd = [pixels,pixels];
Jd = [6,6];
overgridfactor = 2;
om(:,1) = 2*pi*real(kxkytraj(:));
om(:,2) = 2*pi*imag(kxkytraj(:));
st = nufft_init(om, Nd, Jd, overgridfactor*Nd, Nd/2); %initialization
tSetup = toc
```

```
% Reconstroi uma imagem de um slice para apresentar ao usuario
filename = sprintf('%sslice%d.7',rawpath,slice);
xykvt = zeros(pixels,pixels);
p = round(nphases/2);
tic
xykvc = zeros(pixels,pixels,ncoils);
for coil = 1:ncoils,
    rawdata = rawloadHD_jfn(filename,[],[],[], 1,coil,p,1:nintl); % carrega uma fase
        cardiaca de cada bobina
    v = round(nVE/2)
    kxkydata = permute(rawdata(:,v,1,1,:),[1 5 2 4 3]);
    weighteddata = kxkydata(:).*kxkyweights(:); % pondera baseado na densidade de
        amostragem
    m = nufft_adj(weighteddata,st)/pixels; % NUFFT
    xykvc(:,:,coil) = m.';
end;
xykvt(:,:) = combine4channels(xykvc(:,:,1),xykvc(:,:,2),xykvc(:,:,3),xykvc(:,:,4)); %
    combina os dados das bobinas
tRecon1 = toc

% CARREGA TODO O DATASET
tic
rawdata = rawloadHD_jfn(filename,[],[],[], 1,1:ncoils,1:nphases,1:nintl);
rawdata = permute(rawdata,[1 5 2 4 3]);
for p = 1:nphases
    for coil = 1:ncoils
        for v = 1:nVE
            rawdata(:,:,v,p,coil) = rawdata(:,:,v,p,coil) .* kxkyweights;
        end
    end
end
tLoad = toc

% COMPILA O KERNEL CUDA E PREPARA TRANSFERE OS DADOS PARA A GPU
tic
system('/usr/local/cuda/bin/nvcc -ptx cudadrftphase3.cu');

% cria um ponteiro para o kernel CUDA
ck = parallel.gpu.CUDAKernel('cudadrftphase3.ptx', 'cudadrftphase3.cu', 'drftphase');

cudaRes = 8;
% configura como o kernel vai paralelizar a execucao
ck.ThreadBlockSize = [ncoils 1 1];
ck.GridSize = [nphases nVE];

gpu_xykvpcr = gpuArray(zeros(ncoils, nphases,nVE, 'single'));
```

```
gpu_xykvpci = gpuArray(zeros(ncoils, nphases,nVE, 'single'));

% ltransfere os dados para a GPU
gpu_dr = gpuArray(single(real(rawdata(:,:,:,:,:))));
gpu_di = gpuArray(single(imag(rawdata(:,:,:,:,:))));
gpu_kr = gpuArray(single(real(kxkytraj(:))));
gpu_ki = gpuArray(single(imag(kxkytraj(:))));

tCudaSetup = toc

% LOOP PRINCIPAL DA PARTE INTERATIVA
while(1),
    % mostra a imagem de referencia
    tempImg = flipud(fliplr(abs(xykvt(:,:))));
    subplot(211),
    imshow(flipud(fliplr(abs(xykvt(:,:)))),[ ])
    set(gca,'YDir','normal')
    title(sprintf('slice %d',slice))

    % SOLICITA QUE O USUARIO CLIQUE NO VOXEL DESEJADO
    xlabel('CLICK ON A BLOOD VESSEL / CLOSE FIGURE TO STOP')
    [X,Y] = ginput(1);
    X = round(X)
    Y = round(Y)

    xykvc = zeros(nVE,ncoils);
    xykvt_single_pixel = zeros(nVE,nphases);
    xykvt_single_pixel = zeros(nVE, nphases);

    % RECONSTRUCAO POR DRFT
    tic;
    [gpu_xykvpcr_out gpu_xykvpci_out] = feval(ck, gpu_dr, gpu_di, gpu_kr, gpu_ki,
        length(kxkytraj(:)), pixels, X, Y, ncoils, nphases, nVE, gpu_xykvpcr,
        gpu_xykvpci);

    xykvpcr = gather(gpu_xykvpcr_out);
    xykvpci = gather(gpu_xykvpci_out);

    mtemp = complex(xykvpcr, xykvpci);

    for p = 1:nphases
        xykvt_single_pixel(:,p) = combine4channels(mtemp(1,p,:), mtemp(2,p,:),
            mtemp(3,p,:), mtemp(4,p,:));
    end

    xyvt = fftshift(ifft(fftshift(xykvt_single_pixel,1),[],1),1);
    trecon = toc
```

```
    % toma a distribuicao de velocidades do pixel selecionado
    vt = xyvt(:,:);

    % plota o resultado
    taxis = (0:(nphases-1))*optr/1000;
    vaxis = (((0:(nVE-1))/nVE)-0.5)*maxveloc*2;
    subplot(212),imagesc(taxis,vaxis,abs(vt))
    colormap(gray)
    set(gca,'YDir','normal')
    xlabel('time (ms)')
    ylabel('velocity (cm/s)')

end
```

```
// kernel CUDA para reconstrucao por DRFT

__global__ void drftphase( const float * datar, const float * datai, const float * kx,
    const float * ky, const int dataLength, const int Nim, const int xin, const int
    yin, const int nCoils, const int nphases, const int nVE, float * mr, float * mi )
{
    // datar - parte real dos dados de entrada
    // datai - parte imaginaria dos dados de entrada
    // kx - coordenadas x da trajetoria
    // ky - coordenadas y da trajetoria
    // datalengh - numero de pontos de uma trajetoria espiral
    // Nim - dimensao da imagem de saida
    // xin - coordenada x do ponto de interesse
    // yin - coordenada y do ponto de interesse
    // nCoils - numero de bobinas
    // nphases - numero de phase encodings
    // nVE - numero de velocity encodings
    // mr - saida real
    // mi - saida imaginaria
    int c = threadIdx.x;
    int p = (blockIdx.x);
        int v = (blockIdx.y);


    float tempValue;
    float tempCos;
    float tempSin;
    float Nim2 = Nim/2;
    int sample;

    if (c < nCoils)
    {
```

```
        if (p < nphases)
        {
            if (v < nVE)
            {
                int sampleOut = (v * nCoils * nphases) + (p * nCoils) + c;
                int sampleIn = (((c * nVE * nphases) + (p * nVE) + v) * dataLength);
                for(sample = 0; sample < dataLength; sample++)
                {
                    tempValue = 2 * M_PI * ((kx[sample] * (xin-Nim2)) + (ky[sample] *
                        (yin-Nim2)));
                    tempCos = cos(tempValue);
                    tempSin = sin(tempValue);
                    mr[sampleOut] += (datar[sampleIn] * tempCos) + (datai[sampleIn] *
                        tempSin);
                    mi[sampleOut] += (datai[sampleIn] * tempCos) - (datar[sampleIn] *
                        tempSin);
                    sampleIn++;
                    //mr[sampleOut] = c;
                    //mi[sampleOut] = p;
                }
            }
        }
    }
}
```

```
% programa em MATLAB para reconstrucao de imagens utilizando o algoritmo de goertzel
clc, clear, close all

origImg = imread('drft.bmp');

[sizeX sizeY] = size(origImg);

sizeX = 1000;
sizeY = 1000;

for ciclo=1:100

    origImg = round(rand(sizeX,sizeY) * 255);
    tic;
    origfft = fft2(origImg);
    timeFFT(ciclo) = toc;

    D = gpuArray(origImg);
    tic;
    Y = gather(fft2(D));
    timeGPUFFT(ciclo) = toc;
```

```
% compila o kernel CUDA
system('/usr/local/cuda/bin/nvcc -ptx cudagoertzel.cu');

% cria ponteiros para os kernels CUDA
ck = parallel.gpu.CUDAKernel('cudagoertzel.ptx', 'cudagoertzel.cu', 'cudagoec');
ck2 = parallel.gpu.CUDAKernel('cudagoertzel.ptx', 'cudagoertzel.cu', 'cudagoel');

ck.ThreadBlockSize = [sizeX 1];
ck.GridSize = [sizeX];
ck2.ThreadBlockSize = [sizeY 1];
ck2.GridSize = [sizeY];

% aloca memoria na GPU
gpu_datar = gpuArray(zeros(sizeX, sizeY, 'single'));
gpu_datai = gpuArray(zeros(sizeX, sizeY, 'single'));
gpu_dataoutr = gpuArray(zeros(sizeX, sizeY, 'single'));
gpu_dataouti = gpuArray(zeros(sizeX, sizeY, 'single'));

% lcarrega os dados para a GPU
gpu_dr = gpuArray(single(real(origImg(:,:))));
gpu_di = gpuArray(single(imag(origImg(:,:))));

[gpu_datar_out gpu_datai_out] = feval(ck, gpu_dr, gpu_di, sizeX,sizeY, gpu_datar,
    gpu_datai);

reconr = gather(gpu_datar_out);
reconi = gather(gpu_datai_out);

gpu_dr = gpuArray(reconr);
gpu_di = gpuArray(reconi);

[gpu_datar_out gpu_datai_out] = feval(ck2, gpu_dr, gpu_di, sizeX,sizeY, gpu_datar,
    gpu_datai);

reconr = gather(gpu_datar_out);
reconi = gather(gpu_datai_out);

reconfft = complex(reconr, reconi);

% carrega os dados para a GPU
gpu_dr = gpuArray(single(imag(origfft(:,:))));
gpu_di = gpuArray(single(real(origfft(:,:))));

tic;
[gpu_datar_out gpu_datai_out] = feval(ck, gpu_dr, gpu_di, sizeX,sizeY, gpu_datar,
    gpu_datai);
```

```
    reconr = gather(gpu_datar_out);
    reconi = gather(gpu_datai_out);

    gpu_dr = gpuArray(reconr);
    gpu_di = gpuArray(reconi);

    [gpu_datar_out gpu_datai_out] = feval(ck2, gpu_dr, gpu_di, sizeX,sizeY, gpu_datar,
        gpu_datai);
    timeGoe(ciclo) = toc;
    reconr = gather(gpu_datar_out);
    reconi = gather(gpu_datai_out);

    reconImg = complex(reconi, reconr);
    reconImg = reconImg / (sizeX * sizeY);
    reconImg = round(abs(reconImg));

%     figure,
%
%     subplot(221),
%     imshow(origImg,[ ])
%     title('Imagem Original')
%
%     subplot(222),
%     imshow(20*log10(abs(fftshift(origfft))),[min(min(20*log10(abs(origfft))))
    max(max(20*log10(abs(origfft)))) ])
%     title('Espectro Original')
%
%
%     subplot(223),
%     imshow(reconImg,[ ])
%     title('Imagem Reconstruida')
%
%     subplot(224),
%     imshow(20*log10(abs(fftshift(reconfft))),[ min(min(20*log10(abs(origfft))))
    max(max(20*log10(abs(origfft))))])
%     title('Espectro Reconstruido')

    tmp1 = origfft - reconfft;
    tmp2 = single(origImg) - reconImg;

    errMaxFFT(ciclo) = abs(max(max(tmp1))) / abs(max(max(origfft)));
    errMaxImg(ciclo) = abs(max(max(tmp2))) / abs(max(max(origImg)));

    ciclo

end
```

```
figure, plot(abs(errMaxFFT)),title('Erro no espectro');
figure, plot(abs(errMaxImg)),title('Erro na Imagem');
figure, plot(timeFFT),title('Tempo da FFT');
figure, plot(timeGPUFFT),title('Tempo da FFT na GPU');
figure, plot(timeGoe),title('Tempo do Goertzel na GPU');
```