UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

PAULO ROBERTO BUENO

RECONSTRUÇÃO DE IMAGENS DE ULTRASSOM ATRAVÉS DE PROBLEMAS INVERSOS USANDO PROCESSAMENTO PARALELO EM GPUS

TESE

CURITIBA

2018

PAULO ROBERTO BUENO

RECONSTRUÇÃO DE IMAGENS DE ULTRASSOM ATRAVÉS DE PROBLEMAS INVERSOS USANDO PROCESSAMENTO PARALELO EM GPUS

Tese apresentada ao Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Doutor em Ciências" – Área de Concentração: Engenharia Biomédica.

Orientador: Prof. Marcelo Victor Wüst Zibetti, Dr. Eng.

Co-orientador: Prof. Joaquim Miguel Maia, Dr. Eng.

CURITIBA

2018

Dados Internacionais de Catalogação na Publicação

B928r 2018

Bueno, Paulo Roberto
Reconstrução de imagens de ultrassom através de problemas inversos usando processamento paralelo em GPUS / Paulo Roberto Bueno.-- 2018.
120 f. : il. ; 30 cm

Texto em português com resumo em inglês
Disponível também via World Wide Web
Tese (Doutorado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Curitiba, 2018
Bibliografia: f. 87-92

1. Ultrassonografia. 2. Exames médicos - Imagem. 3. Exames médicos - Qualidade da imagem. 4. Algoritmos de correção de erros. 5. Processamento paralelo (Computadores). 6. Problemas inversos (Equações diferenciais). 7. Engenharia elétrica - Teses. I. Zibetti, Marcelo Victor Wüst. II. Maia, Joaquim Miguel. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD: Ed. 23 -- 621.3

Biblioteca Central da UTFPR, Câmpus Curitiba
Bibliotecário: Adriano Lopes CRB-9/1429

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Pesquisa e Pós-Graduação

TERMO DE APROVAÇÃO DE TESE Nº 179

A Tese de Doutorado intitulada **"Reconstrução Paralela de Imagens de Ultrassom Através de Problemas Inversos usando GPUs",** defendida em sessão pública pelo(a) candidato(a) **Paulo Roberto Bueno** no dia 21 de setembro de 2018, foi julgada para a obtenção do título de Doutor em Ciências, área de concentração Engenharia Biomédica, e aprovada em sua forma final, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial.

BANCA EXAMINADORA:

Prof(a). Dr(a). Joaquim Miguel Maia - Presidente – (UTFPR)
Prof(a). Dr(a). Álvaro Rodolfo De Pierro – (UNICAMP)
Prof(a). Dr(a). Daniel Rodrigues Pipa – (UTFPR)
Prof(a). Dr(a). Amauri Amorin Assef – (UTFPR)
Prof(a). Dr(a). Solivan Arantes Valente - (UP)

A via original deste documento encontra-se arquivada na Secretaria do Programa, contendo a assinatura da Coordenação após a entrega da versão corrigida do trabalho.

Curitiba, 21 de setembro de 2018.

AGRADECIMENTOS

À minha esposa, pela paciência e compreensão durante esses anos todos.

Aos familiares e amigos, por entenderem minhas ausências e me incentivarem sempre.

Aos meus orientadores, pela dedicação com que me conduziram durante esta jornada.

RESUMO

BUENO, Paulo R.. RECONSTRUÇÃO DE IMAGENS DE ULTRASSOM ATRAVÉS DE PROBLEMAS INVERSOS USANDO PROCESSAMENTO PARALELO EM GPUS. 120 f. Tese – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2018.

Este projeto de pesquisa busca aprimorar os métodos de reconstrução de imagens de ultrassom baseados em problemas inversos para implementação paralela em GPU. Estes métodos são inovações recentes para ultrassom e têm demonstrado obter qualidade de imagem superior às reconstruções convencionais baseados em *beamforming* e *delay-and-sum*. Contudo, o tempo computacional destes algoritmos ainda é alto, não permitindo execução em tempo real, que é uma demanda já estabelecida em ultrassom. Para isso, apresentamos uma modificação dos algoritmos de ultrassom para execução paralela em unidades de processamento gráfico (GPU). Estes processadores têm grande capacidade de processamento, atingindo níveis de instruções por segundo muito maiores que as CPUs. Porém, os algoritmos podem usufruir deste benefício apenas se forem paralelos ou paralelizáveis. O algoritmo reconstrução de imagens de ultrassom proposto é paralelo e pode ser executado em GPU. Contudo, ele requer uma grande quantidade de memória para armazenar a matriz que representa o sistema de aquisição. Isso limita a resolução máxima das imagens que podem ser reconstruídas na GPU. Este trabalho propõe um modelo alternativo de reconstrução que reduz à metade a necessidade de memória aproveitando a simetria das funções de espalhamento de ponto. Com isso apenas metade da matriz do sistema precisa ser armazenada na GPU. Desta forma, pode-se reconstruir imagens de ultrassom com qualidades superiores às atuais, e praticamente no mesmo tempo.

Palavras-chave: Ultrassom, Reconstrução de Imagens, Problemas Inversos, Processamento Paralelo, GPU

ABSTRACT

BUENO, Paulo R.. ULTRASOUND IMAGES RECONSTRUCTION THROUGH INVERSE PROBLEMS USING PARALLEL COMPUTING ON GPUS. 120 f. Tese – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2018.

This project aims to improve the ultrasound image reconstruction methods based on inverse problems for parallel implementation on GPU. These methods are recent innovations for ultrasound and have demonstrated to achieve better image quality upon conventional reconstructions such as, beamforming and delay-and-sum. However, the computing time of these algorithms are still high, not allowing reconstructions in real-time, which is an already established demand in ultrasound. For this, we have modified the algorithms for parallel execution on graphics processing units (GPU). These processors have a very high processing power, much larger than a CPU. But the algorithms can take advantage of this benefit only if they are parallel or parallelizable. The modified algorithms are parallel and can be executed on GPU. However, they require a large amount of memory to store the system matrix. This fact limits the maximum image resolution that can be obtained by reconstruction on the GPU. This work uses a new and alternative way to reconstruct images through the exploitation of the simmetry of the point spread function, halving the memory size required by the system matrix. Thus, it is possible to reconstruct an ultrasound image with better quality then the current images, and virtually at the same time.

Keywords: Ultrasound, Image Reconstruction, Inverse Problems, Parallel Processing, GPU

LISTA DE FIGURAS

FIGURA 1 – Representação esquemática ilustrativa de um transdutor. 19
FIGURA 2 – Efeitos da variação de impedância sobre a formação de ecos. 20
FIGURA 3 – Resposta eletroacústica ao impulso do transdutor. 22
FIGURA 4 – Sistema de coordenadas do pulso-eco. 23
FIGURA 5 – Pulso-eco e o modo-A de escaneamento. 25
FIGURA 6 – Modo-A e modo-B. 26
FIGURA 7 – Efeitos de focalização baseados em atrasos individuais. 26
FIGURA 8 – Modos de disparos para transdutores multielementos. 28

FIGURA 9 – Modelo esquemático de um sistema real. 31
FIGURA 10 – Curva L ilustrativa. 38
FIGURA 11 – Modelo esquemático de construção da matriz H. 42

FIGURA 12 – Filosofias de projeto para CPUs e GPUs. 45
FIGURA 13 – Classificação Flynn para sistemas de computação. 46
FIGURA 14 – Ilustração da Lei de Amdahl. 48
FIGURA 15 – Exemplo de uma operação de soma paralela. 50
FIGURA 16 – Decomposição e dependência funcional de um processo. 51
FIGURA 17 – Execução sequencial das tarefas. 51
FIGURA 18 – Execução paralela em dois núcleos. 52
FIGURA 19 – Ilustração de escalabilidade 54

FIGURA 20 – Formação da matriz **H** a partir de um modelo simétrico de PSF 58
FIGURA 21 – Permutação dos blocos da **H** para compor o lado simétrico 59
FIGURA 22 – Código C para GPU aplicando simetria 62
FIGURA 23 – Definição da função de integração com o MATLAB 63
FIGURA 24 – Diagrama de componentes do sistema de reconstrução de imagens 64

FIGURA 25 – *Phantom* Fluke 84-317 68
FIGURA 26 – Vista esquemática dos alvos no interior do *Phantom* 68
FIGURA 27 – *Phantoms* sintéticos 69
FIGURA 28 – Vista esquemática dos alvos no interior do *Phantom* 70
FIGURA 29 – Desempenho na multiplicação matriz-matriz em precisão dupla 71
FIGURA 30 – Desempenho na multiplicação matriz-matriz em precisão simples 72
FIGURA 31 – Desempenho na multiplicação matriz-vetor 73
FIGURA 32 – Desempenho na multiplicação matriz-vetor com armazenamento esparso em precisão dupla 74
FIGURA 33 – Erros de reconstrução para os *Phantoms* sintéticos 75

FIGURA 34 – Imagens reconstruídas para os *Phantoms* sintéticos utilizando FISTA 76
FIGURA 35 – Imagens reconstruídas para os *Phantoms* sintéticos utilizando CGNE 77
FIGURA 36 – Erros de reconstrução para os conjuntos de dados reais 81
FIGURA 37 – Resultados das reconstruções com dados reais 82

LISTA DE TABELAS

TABELA 1 – Quadros por segundo por linhas de sinal. 29

TABELA 2 – Configurações do computador utilizado. 66
TABELA 3 – Especificações técnicas das GPUs. 67
TABELA 4 – Parâmetros para o sistema de imageamento por ultrassom. 67
TABELA 5 – Conjunto de dados reais utilizados. 78
TABELA 6 – Características dos conjuntos de dados. 78
TABELA 7 – Tempos das operações, em segundos. 78
TABELA 8 – Tempos medianos para o CGNE por iteração, em segundos. 79
TABELA 9 – Tempos medianos para o CGNE com 5 iterações, em segundos. 79
TABELA 10 – Tempos medianos para o FISTA por iteração, em segundos. 79
TABELA 11 – Tempos medianos para o FISTA com 5 iterações, em segundos. 80

LISTA DE SIGLAS

CGNE	*Conjugate Gradient for Normal Equations*
FISTA	*Fast Iterative Shrinkage-Thresholding Algorithm*
GPU	*Graphics processing unit*
CPU	*Central processing unit*
ROI	*Region Of Interest*
PSF	*Point Spread Function*
MHz	MegaHertz
modo-B	Escaneamento em modo brilho
modo-A	Escaneamento em modo amplitude
fps	*Frames per Second*
DAS	*Delay-and-Sum*
END	Ensaio não destrutivo
ISTA	*Iterative Shrinkage-Thresholding*
DRAM	*Dynamic Random-Access Memory*
GB	*Gigabytes*
s	Segundos
SISD	*Single Instruction Single Data*
SIMD	*Single Instruction Multiple Data*
MISD	*Multiple Instruction Single Data*
MIMD	*Multiple Instruction Multiple Data*
SM	*Streaming Multiprocessor*
BLAS	*Basic Linear Algebra Subprograms*
GEMM	*General Matrix Multiply*
MATLAB	*MATrix LABoratory*
MAGMA	*Matrix Algebra on GPU and Multicore Architetures*
FLOPS	*Foating-point Operations Per Second*
NRMSE	*Normalized Root Mean Square Error*
IEEE	*Institute of Electrical and Electronics Engineers*
SMX	*Streaming Multiprocessor Kepler*
CUDA	*Compute Unified Device Architecture*
GPC	*Graphics Processing Clusters*
SMM	*Streaming Multiprocessor Maxwell*
CUBLAS	CUDA *Basic Linear Algebra Subprograms*
MAGMA	*Matrix Algebra on GPU and Multicore Architetures*

LISTA DE SÍMBOLOS

Z	Impedância acústica
ρ	Densidade do meio
c	Velocidade do som no meio
R	Coeficiente de Reflexão
I	Intensidade de reflexão
$u(t)$	Sinal elétrico de excitação aplicado ao transdutor
$p_k(t)$	Pressão acústica gerada pelo transdutor
k	Elemento do transdutor
$h_k^{ef}(t)$	Resposta ao impulso eletroacústica de emissão
$*$	Convolução no tempo
β	Largura de banda fracionária
ω	Frequência central do transdutor
$\psi(\mathbf{r},t)$	Potencial de velocidade no ponto $\mathbf{r}$, no instante t.
A	Conjunto de todos os pontos da superfície do transdutor
$v_n(\mathbf{r}_2,t)$	Velocidade normal à superfície do transdutor
$h_k^{f-SIR}(\mathbf{r},t)$	Resposta ao impulso espacial de emissão
$p(\mathbf{r},t)$	Pressão total no ponto $\mathbf{r}$, no instante t
K	Número de elementos utilizados do transdutor
$f(\mathbf{r})$	Coeficiente de reflexão do ponto $\mathbf{r}$
$h_k(\mathbf{r},t)$	Resposta ao impulso total
$h_k^{b-SIR}(\mathbf{r},t)$	Resposta ao impulso espacial de retorno
$g_k(\mathbf{r},t)$	Sinal medido no k-ésimo elemento do transdutor proveniente do ponto $\mathbf{r}$, no instante t
T	Taxa de quadros por segundo
P	Profundidade de interesse
L	Número de linhas de sinal em modo-A
m/s	Metros por segundo
μs	Microsegundos
cm	Centímetro
S	Número de amostras discretizadas
M	Total de funções base/pixels que compõem a imagem
t_i	Instante de tempo da i-ézima amostra
$g_k[t_i]$	Sinal elétrico no transdutor discretizado
ℓ_p	Norma vetorial p
λ	Parâmetro de regularização
U	*Speedup* (Computação Paralela)
p	Número de processadores ativos utilizados(Computação Paralela)
w	Trabalho realizado (Computação Paralela)
T	Tempo de execução de um algoritmo (Computação Paralela)
τ	Fração paralelizável de um algoritmo
E	Eficiência de um algoritmo paralelo (Computação Paralela)

SUMÁRIO

1 INTRODUÇÃO 14
2 IMAGENS DE ULTRASSOM 18
2.1 FORMAÇÃO DE IMAGEM DE ULTRASSOM VIA PULSO-ECO 18
2.1.1 Princípio básico do Pulso-Eco 18
2.1.2 Emissão do pulso de ultrassom 21
2.1.3 Recepção dos ecos de ultrassom 24
2.2 SISTEMAS DE IMAGEM DE ULTRASSOM 25
2.2.1 Modos de escaneamento em ultrassom 25
2.2.2 O Imageamento em Tempo Real 27
2.3 CONCLUSÃO DO CAPÍTULO 29
3 RECONSTRUÇÃO BASEADA EM PROBLEMAS INVERSOS 31
3.1 FORMULAÇÃO DO PROBLEMA DIRETO 31
3.2 REPRESENTAÇÃO MATRICIAL DISCRETA 33
3.3 RECONSTRUÇÃO 34
3.3.1 Regularização 37
3.3.2 Algoritmos iterativos 39
3.4 MODELO DE AQUISIÇÃO 41
3.5 CONCLUSÃO DO CAPÍTULO 43
4 PROCESSAMENTO PARALELO EM GPU 44
4.1 FUNDAMENTOS DO PROCESSAMENTO PARALELO 44
4.1.1 Tipos de concorrência e granularidade 45
4.1.2 Medidas básicas de desempenho 47
4.2 PROJETO DE ALGORITMOS PARALELOS 50
4.2.1 Arquiteturas físicas das GPUs 52
4.3 PROGRAMAÇÃO PARALELA COM CUDA C 53
4.4 ALGORITMOS PARALELOS PARA ÁLGEBRA LINEAR 54
4.5 CONCLUSÃO DO CAPÍTULO 56
5 MODELO DE AQUISIÇÃO DE ULTRASSOM COM PSF SIMÉTRICA 57
5.1 PROPOSTA DE UTILIZAÇÃO DA SIMETRIA DA PSF 57
5.2 CUSTOMIZAÇÃO DAS OPERAÇÕES DE MULTIPLICAÇÃO PARALELA EM GPU 61
5.3 REDUÇÕES BÁSICAS 64
5.4 CONCLUSÃO DO CAPÍTULO 65
6 MATERIAIS E MÉTODOS - EXPERIMENTOS E RESULTADOS 66
6.1 AMBIENTE EXPERIMENTAL 66
6.2 EXPERIMENTO 1: AVALIAÇÃO DE DESEMPENHO EM OPERAÇÕES BÁSICAS 70
6.3 EXPERIMENTO 2: AVALIAÇÃO DE DESEMPENHO COM MATRIZES ESPARSAS 71
6.4 EXPERIMENTO 3: AVALIAÇÃO DO MODELO SIMÉTRICO 72

6.4.1 Testes com dados sintéticos 74
6.4.2 Testes com dados reais 75
6.5 CONCLUSÃO DO CAPÍTULO 81
7 CONSIDERAÇÕES FINAIS E CONCLUSÕES 84
7.1 TRABALHOS FUTUROS 86
REFERÊNCIAS 87
Apêndice A – DESEMPENHO EM OPERAÇÕES BÁSICAS 93
Apêndice B – DESEMPENHO COM MATRIZES ESPARSAS 95
Apêndice C – ARQUITETURAS DAS GPUS 96
C.1 ARQUITETURA KEPLER 96
C.2 ARQUITETURA MAXWELL 98
Apêndice D – CODIFICAÇÃO EM CUDA 100
Apêndice E – ALGORITMOS PARALELOS PARA ÁLGEBRA LINEAR 102
E.1 ALGORITMO DE STRASSEN-WINOGRAD 102
Apêndice F – MULTIPLICAÇÃO MATRIZ-VETOR PARALELA E SIMÉTRICA . 105
Apêndice G – MULTIPLICAÇÃO MATRIZ-MATRIZ PARALELA E SIMÉTRICA 113

1 INTRODUÇÃO

Os sistemas de imagem por ultrassom são úteis em diversas áreas da atividade humana. Os exemplos mais comuns vão desde a área da saúde, com exames de imagem para diagnóstico médico (STERGIOPOULOS, 2009), até na produção industrial, em inspeção de qualidade de produtos e na busca por falhas e defeitos (GROOVER, 1987; GUARNERI et al., 2015).

Os constantes avanços têm consolidado os sistemas de imageamento médico por ultrassom como uma solução confiável, de custo relativamente baixo e que oferece maior conforto ao paciente (AZHARI, 2010). Dois aspectos importantes, que vêm continuamente sendo melhorados, são: 1) a qualidade da imagem, onde a resolução espacial[1] é importante (ELLIS et al., 2010); e 2) o tempo de tratamento da imagem, onde é relevante a velocidade com que os dados são capturados e o tempo de reconstrução.

Os métodos de reconstrução de imagens de ultrassom baseados em problemas inversos são inovações recentes para ultrassom e têm demonstrado obter qualidade de imagem superior às reconstruções convencionais baseadas em *beamforming* e *delay-and-sum* (ZANIN, 2011; LU et al., 1994; SYNNEVAG et al., 2007). Contudo, o custo computacional desses algoritmos é alto, não permitindo execução em tempo real, que é uma demanda já estabelecida em ultrassom (AZHARI, 2010; FONTES et al., 2011; DAI et al., 2010).

Existe uma variedade de métodos para reconstrução por problemas inversos (VALENTE et al., 2017): dentre eles, foram escolhidos dois para serem objeto de avaliação e aplicação neste trabalho: o *Conjugate Gradient for Normal Equations* (CGNE) e o *Fast Iterative Shrinkage-Thresholding Algorithm* (FISTA). O primeiro por ser considerado como um

[1] Resolução Espacial é definida como a menor distância que separa dois objetos de mesmas propriedades e que podem ser visualizados como objetos separados (AZHARI, 2010).

método clássico (SAAD, 2003) e o segundo por ser objeto de diversos estudos recentes (BECK; TEBOULLE, 2009a; LIN et al., 2015).

Embora estes métodos apresentem uma melhora na qualidade da imagem, ainda não são utilizados na prática devido ao seu custo computacional. Neste trabalho conseguimos contornar este problema através do aprimoramento destes métodos para implementação paralela em Unidades de Processamento Gráfico (GPU). Estes processadores possuem grande capacidade computacional, atingindo níveis de instruções por segundo muito maiores que as Unidades de Processamento Central (CPU) (SCHIWIETZ et al., 2006ba; NVIDIA Corporation, 2014c, 2014ba).

Um aspecto especialmente importante é a quantidade de memória disponível em uma GPU. Geralmente, este recurso é limitado e impacta diretamente no tamanho da região visualizada e na resolução da imagem. Os modelos empregados na reconstrução de imagens por problemas inversos normalmente requerem muita memória, maior que a disponível em muitas GPUs disponíveis (ZANIN, 2011). Considerando os avanços recentes dos métodos de reconstrução de ultrassom por problemas inversos, a solução deste problema é importante, sendo um fator motivador para o desenvolvimento deste trabalho.

O principal objetivo deste trabalho é a implementação de uma versão customizada dos algoritmos selecionados, capaz de executar a reconstrução de uma imagem de ultrassom em tempo real, ou próximo disto, em GPU. Para alcançar este objetivo é necessário atingir os seguintes objetivos específicos:

1. Preparação de um ambiente computacional apropriado para a realização de experimentos com dados de ultrassom sintéticos e reais em GPU;
2. Desenvolvimento de um modelo do problema de acordo com os conceitos de problemas inversos e capaz de reduzir o consumo de memória em sua implementação;
3. Desenvolvimento de algoritmos de reconstrução de imagens paralelos que possam ser executados com eficiência em GPU.

Os três objetivos específicos propostos foram alcançados como passamos a descrever. O ambiente computacional foi construído empregando duas configurações distintas de GPU e uma mesma configuração de CPU. O emprego de duas GPUs com capacidades distintas permitiu uma avaliação do comportamento dos algoritmos em condições de processamento diferentes. Os resultados obtidos pelo processamento em CPU serviram como referência para a medição do ganho de desempenho e do erro na reconstrução das imagens.

A redução do consumo de memória foi alcançada com uma modificação na modelagem do problema, introduzindo o conceito de simetria. O modelo convencional é construído a partir do estabelecimento de uma Região de Interesse (ROI) hipotética (VIOLA et al., 2008) e da captura dos sinais esperados para cada ponto definido através de um método para simulação dos sinais de ultrassom (JENSEN, 2004).

No modelo simétrico assumiu-se que as Funções de Espalhamento de Ponto (PSF) do sistema de aquisição são lateralmente simétricas (em relação ao eixo central de propagação). Isto permitiu o armazenamento de apenas metade das PSFs. As metdades obtidas podem ser facilmente produzidas com as armazenadas, reduzindo significativamente o consumo de memória do modelo.

Finalmente, o último objetivo específico foi alcançado com a customização dos algoritmos e com a implementação de um conjunto de funções, detalhadas nos Anexos F e G, para as operações básicas com matrizes. Estas funções, implementadas em linguagem de programação C, permitiram a manipulação da matriz reduzida do modelo, ampliando-a e obtendo os mesmos resultados que seriam obtidos com uma matriz não reduzida. No desenvolvimento destas funções aplicaram-se conceitos de paralelização similares aos empregados em bibliotecas atualmente disponíveis como CUBLAS (NVIDIA Corporation, 2014ba) com a adição de técnicas para o ajuste do paralelismo de acordo com as características físicas da matriz (XU et al., 2012; ABDELFATTAH et al., 2014).

Este trabalho está dividido como segue:

- Capítulo 2: trata dos temas fundamentais da formação de imagens por ultrassom e a

forma como ela é representado a partir do modelo contínuo até o modelo discreto na forma matricial.

- Capítulo 3: são abordadas as condições para reconstrução através de problemas inversos, bem como, são apresentados alguns métodos avaliados para possível implementação em GPU.

- Capítulo 4: apresenta os conceitos fundamentais do processamento paralelo e os aspectos que influenciam e determinam o desempenho dos algoritmos; são apresentados, também, os detalhes técnicos das GPUs empregadas nesta pesquisa.

- Capítulo 5: apresenta o modelo simétrico proposto neste trabalho e os detalhes de sua implementação.

- Capítulo 6: são apresentados o ambiente experimental utilizado neste trabalho, os principais experimentos realizados e seus resultados.

- Capítulo 7: são apresentadas as conclusões gerais.

2 IMAGENS DE ULTRASSOM

Os sistemas de imagem por ultrassom são baseados na propagação e reflexão de ondas acústicas através das áreas observadas (HEDRICK et al., 1995). Estas ondas são de natureza mecânica e não ionizante, portanto mais seguras para aplicação em diagnóstico e tratamento médico (JENSEN, 2002; AZHARI, 2010). Devido a rápida velocidade de propagação do som nos tecidos, esse técnica de imageamento permite uma alta taxa de amostragem espaço-temporal, sendo possível a observação em tempo real dos tecidos (HEDRICK et al., 1995; AZHARI, 2010).

Neste trabalho optamos pelo uso do modo pulso-eco dos sistemas de ultrassom (AZHARI, 2010), que se caracteriza pela emissão e recepção em um mesmo transdutor. O pulso-eco é largamente utilizado para imageamento médico, justificando, assim, sua escolha (HEDRICK et al., 1995; WEBB, 2003). Para os fins deste trabalho supõem-se que o sistema é linear e invariante no tempo (VIOLA et al., 2008; ZEMP et al., 2003; LINGVALL; OLOFSSON, 2007; ZANIN et al., 2011).

2.1 FORMAÇÃO DE IMAGEM DE ULTRASSOM VIA PULSO-ECO

A seguir, serão abordados os conceitos fundamentais do ultrassom e como obter imagens a partir de seus sinais.

2.1.1 PRINCÍPIO BÁSICO DO PULSO-ECO

Tipicamente, para formação de imagens utilizam-se frequências acústicas entre 1 megahertz (MHz) e 20 MHz. A emissão consiste na aplicação de um sinal elétrico de excitação

a um transdutor, que o converte em uma onda mecânica. Esta onda viaja pelo meio que se pretende observar, sendo refletida ao encontrar variações na impedância acústica do meio, retornando ao transdutor onde é reconvertida em sinal elétrico (HEDRICK et al., 1995; ZANIN, 2011), sendo o sinal emitido e recebido pelo mesmo transdutor (AZHARI, 2010).

O transdutor, principal elemento de um sistema de ultrassom, é responsável pela conversão de um sinal elétrico em onda acústica e pela conversão da onda recebida em sinal elétrico. Ele é composto pela cerâmica piezoelétrica, por uma camada de amortecimento, uma de compatibilização acústica e pelo contato metálico (HEDRICK et al., 1995), ilustrado na Figura 1.

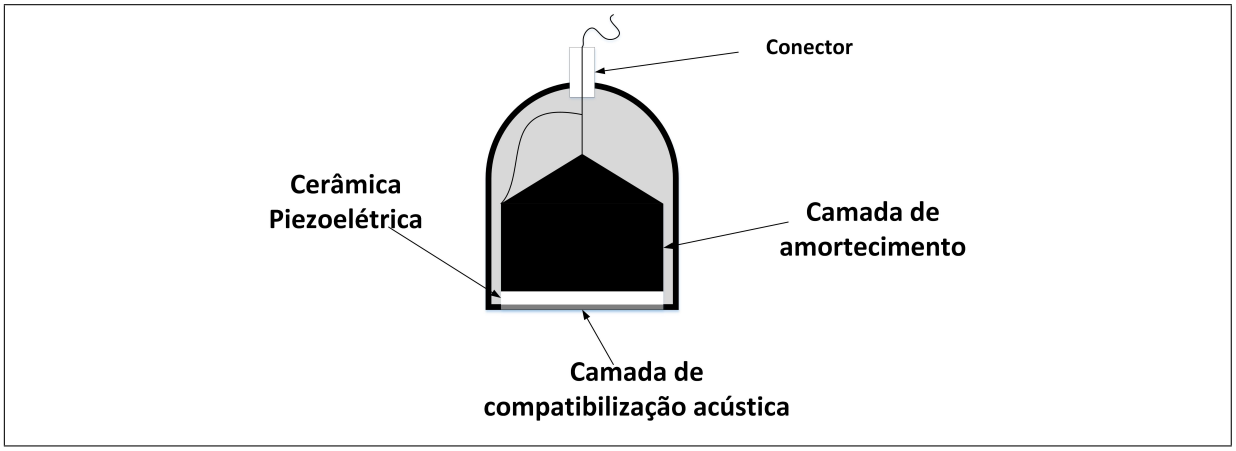


Figura 1: Representação esquemática ilustrativa da montagem de um transdutor monoelemento de ultrassom.
Fonte: Adaptado de (HEDRICK et al., 1995).

Quando uma onda plana é gerada e disparada perpendicularmente aos tecidos, sempre que houver uma diferença na impedância acústica, uma parte da sua energia será refletida de volta ao transdutor (HEDRICK et al., 1995). Este efeito está ilustrado na Figura 2. A impedância acústica Z, pode ser definida pelo produto entre a densidade ρ e a velocidade do som no meio c (AZHARI, 2010; HEDRICK et al., 1995; ZANIN, 2011):

$$Z = \rho c \tag{1}$$

No caso de incidência perpendicular e considerando as diferenças de impedância é possível definir o coeficiente de reflexão R, sendo Z_1 a impedância acústica do meio 1 e Z_2 a

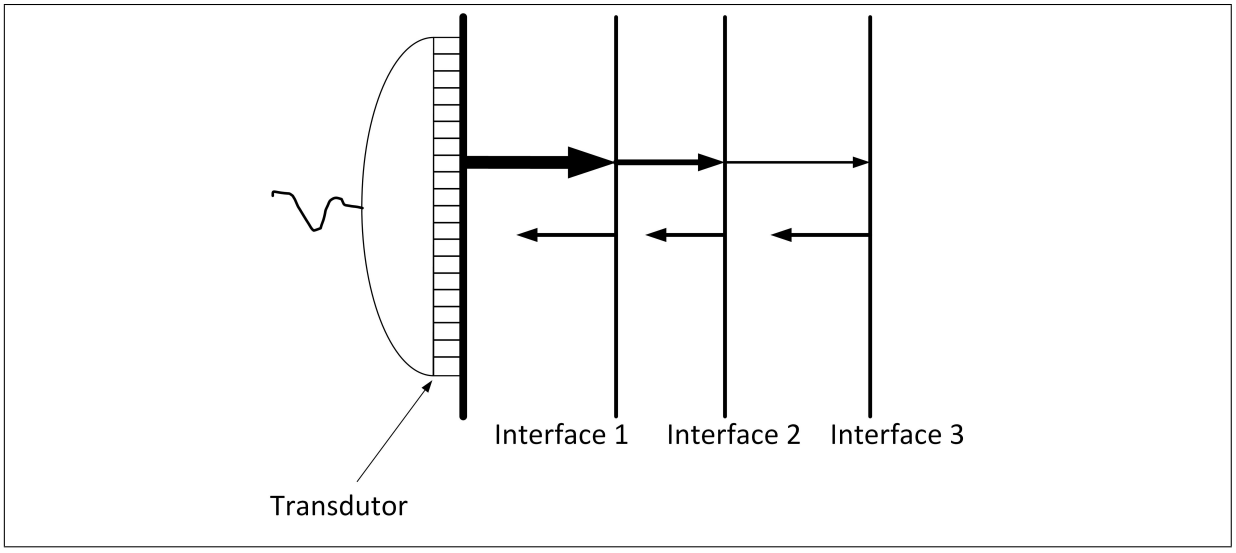


Figura 2: Efeitos da variação de impedância sobre a formação de ecos, a cada interface corresponde uma mudança na impedância do meio ou tecido.
Fonte: Adaptado de (HEDRICK et al., 1995).

impedância acústica do meio 2 (WEBB, 2003; BUSHBERG et al., 2002):

$$R = \frac{(|Z_2| - |Z_1|)}{(|Z_2| + |Z_1|)} \tag{2}$$

e a intensidade I de reflexão é dada por:

$$I = \left(\frac{|Z_2| - |Z_1|}{|Z_2| + |Z_1|} \right)^2 \tag{3}$$

A partir do coeficiente R observam-se três situações (AZHARI, 2010; HEDRICK et al., 1995):

- Quando $Z_1 = Z_2$, não há onda refletida;
- Quando $Z_2 > Z_1$, a onda será refletida em fase com à onda incidente;
- Quando $Z_2 < Z_1$, a onda será refletida com 180°de defasagem em relação à onda incidente.

Assim, é possível identificar as diversas interfaces que compõem o tecido e além disso, considerando-se os tempos de emissão e recepção dos sinais, é possível calcular a profundidade das mesmas. Neste trabalho foi utilizada a formação de onda plana, perpendicular e de um único disparo. Embora a impedância acústica seja uma grandeza complexa, neste trabalho foi empregada apenas sua parte real.

2.1.2 EMISSÃO DO PULSO DE ULTRASSOM

Quando aplicada uma excitação elétrica à cerâmica piezoelétrica, $u(t)$, a mesma gera uma pressão acústica que será transmitida ao meio que se deseja visualizar. A pressão acústica gerada é definida por (AZHARI, 2010; DEMIRLI; SANIIE, 2001; ZANIN et al., 2011):

$$\begin{aligned} p_k(t) &= \int_{-\infty}^{+\infty} u(\tau) h_k^{ef}(t-\tau) d\tau \\ p_k(t) &= h_k^{ef}(t) * u(t) \end{aligned} \tag{4}$$

onde $p_k(t)$ é a pressão acústica gerada pelo k-ésimo elemento do transdutor, $h_k^{ef}(t)$ é a resposta eletroacústica de emissão ao impulso e o símbolo $*$ representa a convolução no tempo. Neste trabalho foi empregada a expressão da Equação 5 para definir a resposta ao impulso do transdutor (AZHARI, 2010; LU et al., 2014):

$$h^{ef}(t) = e^{-\beta t^2} \cos(2\pi\omega t) \tag{5}$$

onde β é a largura de banda fracionária e ω é a frequência central do transdutor. Por apresentar uma envoltória Gaussiana, este modelo também é chamado de modelo de ecos Gaussianos (DEMIRLI; SANIIE, 2001). Na Figura 3 está ilustrada esta resposta no domínio do tempo e a sua envoltória, que pode ser extraída aplicando-se a transformada de Hilbert, como descrito por Azhari (2010).

A pressão gerada no elemento do transdutor é propagada através do meio de densidade homogênea ρ_0 atingindo um ponto $\mathbf{r}$ afastado do transdutor, cuja geometria está ilustrada na Figura 4, onde $\mathbf{r}_1$ representa um ponto no qual se deseja observar a pressão acústica e $\mathbf{r}_2$ representa a posição espacial do transdutor em relação à origem. Pelo princípio de Huygens (AZHARI, 2010; JENSEN, 1991, 1992b), cada ponto infinitesimal irradiante do transdutor gera uma onda esférica que se afasta em velocidade constante c; assim, a pressão resultante será dependente de ambas as posições, de emissão e recepção calculada por ($\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1$). O equacionamento que segue assume um meio linear e homogêneo, apesar de ser incluída no modelo a atenuação não foi considerada neste trabalho.

Primeiramente, é necessário calcular o potencial de velocidade, $\psi(\mathbf{r},t)$, do campo

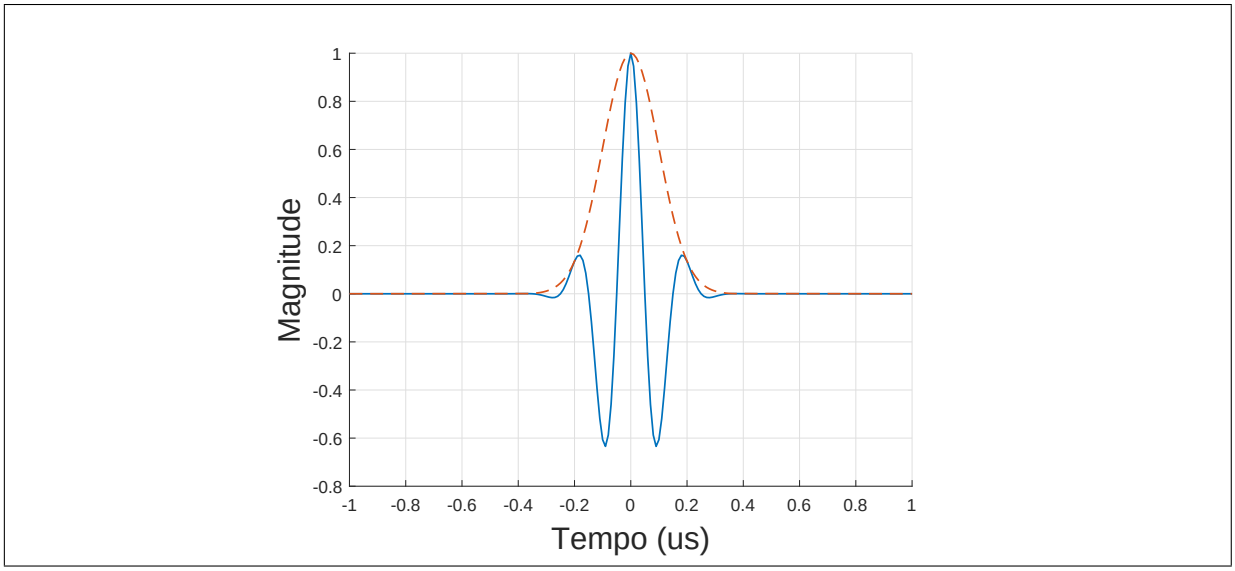


Figura 3: Resposta eletroacústica ao impulso do transdutor no domínio do tempo e sua envoltória.
Fonte:Própria.

gerado pela contribuição de todos os pontos infinitesimais da superfície do transdutor (JENSEN, 2015; STEPANISHEN, 2005; JENSEN, 1992a):

$$\psi(\mathbf{r}_1,t) = \int_A \frac{v_n\left(\mathbf{r}_2, t - \frac{|\mathbf{r}_1-\mathbf{r}_2|}{c}\right)}{2\pi|\mathbf{r}_1-\mathbf{r}_2|} d\mathbf{r}_2 \tag{6}$$

onde A representa o conjunto de todos os pontos infinitesimais da superfície do transdutor, nas posições $\mathbf{r}_2$, e $v_n(\mathbf{r}_2,t)$ é a velocidade normal à superfície do transdutor.

Se convolvermos Equação (6) no domínio do tempo com a função Delta de Dirac, $\delta\left(t - \frac{|\mathbf{r}_1-\mathbf{r}_2|}{c}\right)$, é possível separar a velocidade da geometria do transdutor, como segue:

$$\psi(\mathbf{r}_1,t) = \int_A \int_T \frac{v_n(\mathbf{r}_2,t_2)\delta\left(t - t_2 - \frac{|\mathbf{r}_1-\mathbf{r}_2|}{c}\right)}{2\pi|\mathbf{r}_1-\mathbf{r}_2|} dt_2 d\mathbf{r}_2 \tag{7}$$

onde T é o intervalo de tempo para propagação da onda e t_2 são todos os instantes de tempo em T.

Assumindo que a velocidade é uniforme em toda a superfície do transdutor podemos

simplificar a Equação (7), através de:

$$\begin{aligned} v_n(t) &= v_n(\mathbf{r}_2, t_2) \\ \psi(\mathbf{r}_1, t) &= v_n(t) * \int_A \frac{\delta\left(t - \frac{|\mathbf{r}_1 - \mathbf{r}_2|}{c}\right)}{2\pi|\mathbf{r}_1 - \mathbf{r}_2|} d\mathbf{r}_2 \end{aligned} \tag{8}$$

A integral na Equação (8) segue o princípio de Huygens de espalhamento esférico e representa o comportamento do espalhamento da velocidade no meio, também chamado de resposta espacial de emissão ou direta (VALENTE et al., 2017; JENSEN, 2004, 2015; STEPANISHEN, 2005):

$$\begin{aligned} h^{f-SIR}(\mathbf{r}_1, t) &= \int_A \frac{\delta\left(t - \frac{|\mathbf{r}_1 - \mathbf{r}_2|}{c}\right)}{2\pi|\mathbf{r}_1 - \mathbf{r}_2|} d\mathbf{r}_2 \\ h_k^{f-SIR}(\mathbf{r}, t) &= \int_A \frac{\delta\left(t - \frac{|\mathbf{r}|}{c}\right)}{2\pi|\mathbf{r}|} d\mathbf{r}_2 \end{aligned} \tag{9}$$

onde $h_k^{f-SIR}(\mathbf{r}, t)$ é a resposta ao impulso espacial de emissão, $\delta\left(t - \frac{|\mathbf{r}|}{c}\right)$ é a função Delta de Dirac, que repesenta a contribuição do *k*-ésimo elemento da superfície *A* do transdutor apenas quando sua onda estimula o ponto **r**.

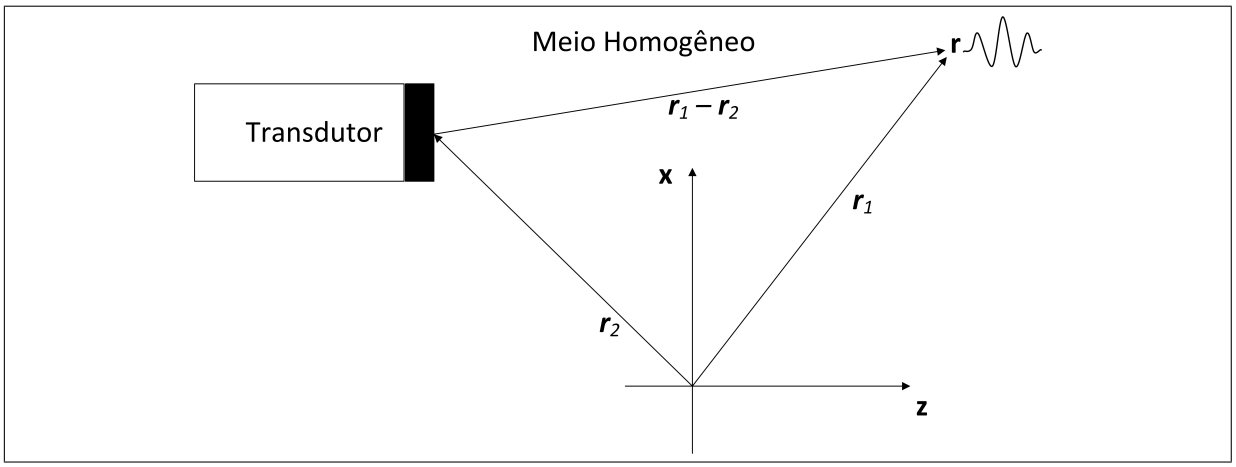


Figura 4: Sistema de coordenadas para o cálculo da pressão acústica em um determinado ponto.
Fonte: Adaptado de (JENSEN, 2015).

Finalmente, podemos calcular a pressão acústica no ponto **r**, gerada pelo *k*-ésimo elemento do transdutor, através das Equações (4) e (9) como:

$$p_k(\mathbf{r}, t) = h_k^{f-SIR}(\mathbf{r}, t) * h_k^{ef}(t) * u_k(t) \tag{10}$$

A pressão total $p(\mathbf{r},t)$ sofrida pelo ponto $\mathbf{r}$ pode ser definida como a soma de todas as contribuições individuais dos K elementos do transdutor ao longo do tempo:

$$\begin{aligned} p(\mathbf{r},t) &= \sum_{k=1}^{K} h_k^{f-SIR}(\mathbf{r},t) * h_k^{ef}(t) * u_k(t) \\ p(\mathbf{r},t) &= \sum_{k=1}^{K} p_k(\mathbf{r},t) \end{aligned} \tag{11}$$

2.1.3 RECEPÇÃO DOS ECOS DE ULTRASSOM

Considerando que cada ponto de uma região imageada responde de forma diferente de acordo com as variações de impedância acústica, por limites entre órgãos ou mudanças no tipo de material, ele poderá ou não refletir (total ou parcialmente) a onda incidente. Quando um ponto de interesse reflete a onda incidente ele pode ser considerado como um emissor de ondas acústicas. Esta onda refletida (eco) se propaga através do meio de volta ao transdutor onde, será convertida em sinal elétrico.

Assumindo que o meio seja linear é aceitável considerar que a resposta ao impulso espacial de retorno será igual a resposta espacial de emissão (ZANIN et al., 2011). A onda refletida pelo ponto $\mathbf{r}$ será ponderada pela sua força de reflexão $f(\mathbf{r})$ conforme a equação (ZEMP et al., 2003; AZHARI, 2010; HEDRICK et al., 1995; LINGVALL; OLOFSSON, 2007):

$$f(\mathbf{r}) = -2\frac{Z(\mathbf{r}) - Z_0}{Z_0} \tag{12}$$

onde Z_0 é a impedância do meio de onde a onda vem e $Z(\mathbf{r})$ é a impedância do meio para onde a onda vai, o ponto $\mathbf{r}$ faz parte da superfície de fronteira entre esses meios.

A resposta ao impulso total para o k-ésimo elemento do transdutor $h_k(\mathbf{r},t)$ será o resultado da convolução entre a resposta acústica-elétrica do elemento $h_k^{eb}(t)$ convolvida com a resposta ao impulso espacial de retorno $h_k^{b-SIR}(\mathbf{r},t)$, convolvida com a pressão acústica no ponto $\mathbf{r}$:

$$h_k(\mathbf{r},t) = h_k^{eb}(t) * h_k^{b-SIR}(\mathbf{r},t) * p(\mathbf{r},t) \tag{13}$$

Por fim, o sinal elétrico $g_k(\mathbf{r},t)$ resultante do eco recebido no k-ésimo elemento do

transdutor será:

$$g_k(\mathbf{r},t) = h_k(\mathbf{r},t) f(\mathbf{r}) \tag{14}$$

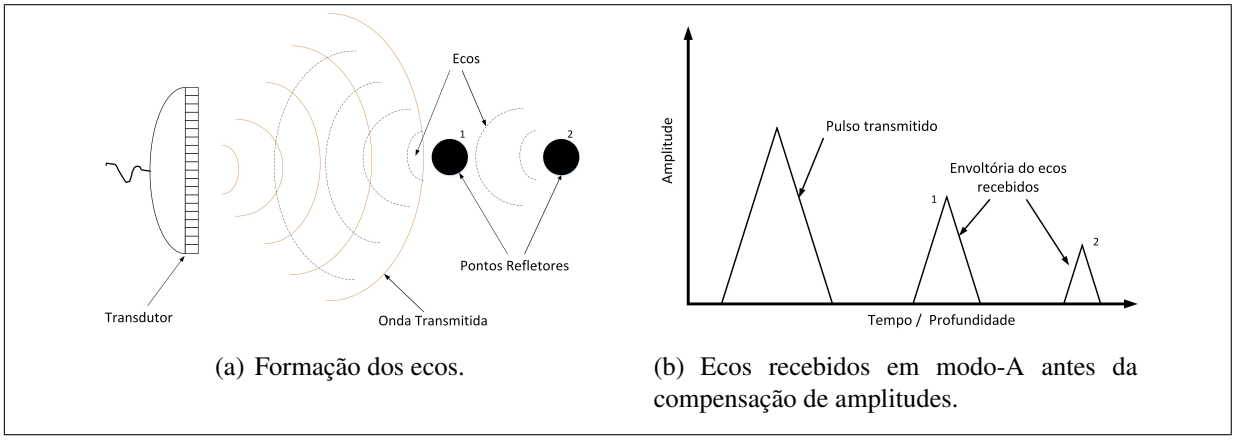
2.2 SISTEMAS DE IMAGEM DE ULTRASSOM

Os sistemas de imagem de ultrassom compreendem todos os componentes necessários para a aquisição, tratamento e processamento dos sinais de ultrassom e sua visualização pelos usuários (BUSHBERG et al., 2002). Nesta seção são abordados alguns aspectos de maior relevância para o desenvolvimento deste trabalho.

2.2.1 MODOS DE ESCANEAMENTO EM ULTRASSOM

O modo-B de escaneamento é a forma mais usada para geração de imagens médicas de ultrassom (AZHARI, 2010; CHRISTENSEN, 1998; WEBB, 2003). O modo-B, também chamado de modo brilho, é composto por múltiplas linhas de modo-A, que é chamado modo amplitude.

A Figura 5 representa o processo de escaneamento por pulso-eco e a formação das linhas do modo-A (HEDRICK et al., 1995; AZHARI, 2010; WEBB, 2003; CHRISTENSEN, 1998).



(a) Formação dos ecos.

(b) Ecos recebidos em modo-A antes da compensação de amplitudes.

Figura 5: Pulso-eco e o modo-A de escaneamento.
Fonte: Adaptado de (HEDRICK et al., 1995).

Os sistemas de ultrassom operando no modo-B convertem as amplitudes recebidas em *pixels* representados em escala de cinza na visualização. Neste caso a imagem gerada

é bidimensional. Uma imagem no modo-B é constituída por diversas linhas do modo-A (HEDRICK et al., 1995; AZHARI, 2010; ZANIN, 2011), isto pode ser visto na Figura 6.

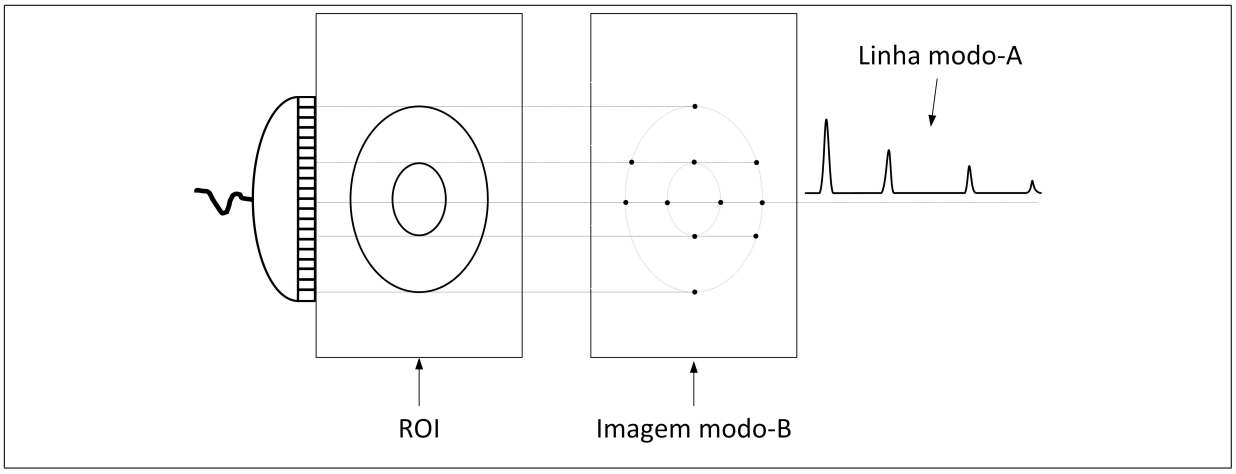


Figura 6: O modo-A é representado como uma linha com amplitude. No modo-B a amplitude controla o brilho.
Fonte: Adaptado de (HEDRICK et al., 1995).

Os equipamentos de ultrassom podem utilizar transdutores multielementos; assim, os sinais podem ser melhorados através da geração de múltiplas linhas, focalização e direcionamento do feixe. Estes dois últimos efeitos podem ser obtidos pela inserção de atrasos no disparo de cada elemento individualmente, os quais estão ilustrados na Figura 7 (AZHARI, 2010; HEDRICK et al., 1995; ZANIN, 2011).

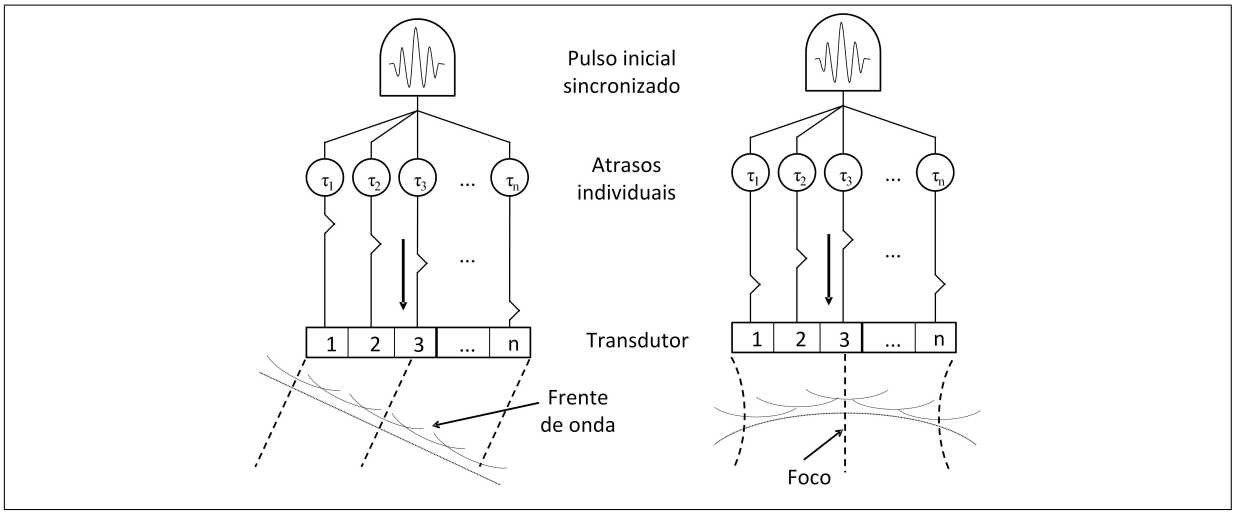


Figura 7: Efeitos de focalização baseados em atrasos individuais.
Fonte: Adaptado de (HEDRICK et al., 1995).

2.2.2 O IMAGEAMENTO EM TEMPO REAL

A geração de imagens médicas em tempo real permite diagnóstico mais rápido e melhor acompanhamento de pacientes (HEDRICK et al., 1995). Para obter o imageamento em tempo real, é preciso considerar o tempo de aquisição do sinal e, também, o tempo de reconstrução da imagem.

No modo de operação pulso-eco o tempo de aquisição é limitado pelo tempo que a onda leva para atingir a profundidade de interesse e retornar ao transdutor. Além disso, a formação de uma imagem no modo-B depende da captura de múltiplas linhas de sinal. O número dessas linhas influencia na resolução espacial. Deste modo, para se obter boas imagens em tempo real é preciso ajustar as necessidades de resolução espacial com o tempo necessário para as ondas viajarem até a profundidade de interesse e voltarem para o transdutor (BUSHBERG et al., 2002). Estes fatores impõem restrições quanto à taxa de quadros por segundo (fps), que pode ser calculada por (HEDRICK et al., 1995; BUSHBERG et al., 2002):

$$T = \frac{c}{2PL} \tag{15}$$

onde: T é a taxa que quadros por segundo, c é a velocidade do som no meio, P é a profundidade de interesse, L é o número de linhas do sinal por quadro.

Pela Equação (15), observa-se que quando a profundidade de interesse e/ou o número de linhas aumenta, a taxa máxima de quadros possível diminui. Considerando, por exemplo, a velocidade do som nos tecidos como 1.540 m/s, a onda acústica leva 13 μs para percorrer 1 cm de tecido. Desta forma, para se escanear um tecido a 10 cm de profundidade em modo-B, com 100 linhas de sinais, a taxa máxima de quadros possível, seria de 77 fps.

No caso de um transdutor multielementos linear sequencial, no qual cada elemento é disparado individualmente de maneira sequencial, o intervalo de disparo é determinado pela profundidade de interesse. De forma similar é possível determinar a taxa máxima de quadros para transdutores multielementos segmentados. Nestes, os elementos são disparados de maneira agrupada, onde cada grupo corresponde a uma linha de sinal.

Assim, se no caso anterior os elementos fossem excitados em grupos de dez elementos teríamos um total de 13 linhas de sinal, considerando um total de 130 elementos. A principal desvantagem neste caso é a perda de resolução espacial devido ao baixo número de linhas de sinal. Os diversos tipos de disparos podem ser observados na Figura 8.

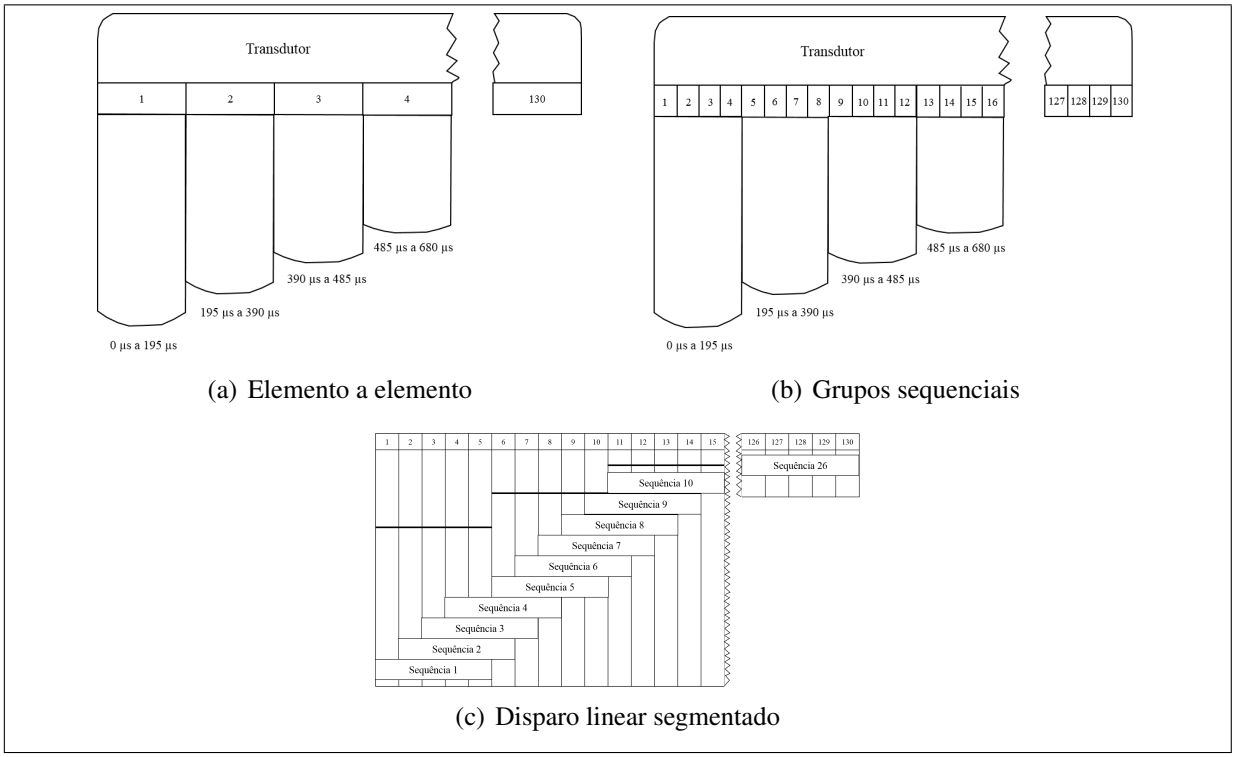


Figura 8: Modos de disparos para transdutores multielementos: (a) os elementos são disparados individualmente, (b) os elementos são disparados agrupados e de maneira sequencial e (c) os elementos são disparados de forma segmentada.
Fonte: Adaptado de (HEDRICK et al., 1995).

Para melhorar a resolução espacial com transdutores multielementos e alcançar ganhos em termos de taxa máxima, pode-se disparar grupos de elementos com diferença de passo de um elemento em cada disparo (HEDRICK et al., 1995; WEBB, 2003). Com esta estratégia é possível aumentar a resolução espacial e, ao mesmo tempo obter uma taxa máxima melhor, que no caso anterior nos daria 121 linhas de sinal. Na Tabela 1, temos uma comparação da taxa máxima de quadros para cada configuração de linhas de sinal e profundidade de interesse.

Como observado, diversos fatores precisam ser levados em conta no projeto de um equipamento de ultrassom para formação de imagem em tempo real. A definição da quantidade de elementos no transdutor, a forma de disparo destes elementos, as expectativas em termos de

Tabela 1: Quadros por segundo por linhas de sinal.

	Linhas de Sinal		
Profundidade de Interesse	130	100	13
5 cm	118 fps	154 fps	1.184 fps
10 cm	59 fps	77 fps	592 fps
15 cm	39 fps	51 fps	394 fps

Fonte: Própria.

resolução espacial, tempos para aquisição dos sinais e as profundidades de interesse máximas a serem alcançadas. Do equilíbrio destas variáveis irá depender a qualidade final da imagem apresentada.

Os tempos de reconstrução das imagens são determinados, de maneira geral, pelo método de reconstrução empregado e pela capacidade do *hardware* disponível. Após o processo de aquisição dos sinais é necessário seu processamento para a obtenção da imagem desejada. Neste caso é necessário analisar um novo conjunto de fatores, além da profundidade de interesse e da quantidade de linhas de sinal. De acordo com Hedrick et al. (1995), a taxa de quadros por segundo desejável para exames em tempo-real de ultrassom é de aproximadamente 30 fps.

O método mais utilizado para reconstrução de imagens em ultrassom é o *beamforming*. Como em outros métodos, ele baseia-se numa estimativa dos sinais recebidos de um determinado ponto no espaço, ou de uma direção, através do conjunto de sinais capturados nos diversos sensores. São considerados atrasos, ponderações e soma dos sinais recebidos em cada transdutor. Em ultrassom médico a forma mais empregada de *beamforming* é a *Delay-and-Sum* (DAS) (LU et al., 1994; WEBB, 2003; SYNNEVAG et al., 2007; ZANIN, 2011). Este método pode ser visto em detalhes por Lu et al. (1994). Para o caso de reconstrução baseada em problemas inversos o tempo de processamento é um obstáculo a ser superado.

2.3 CONCLUSÃO DO CAPÍTULO

Neste capítulo abordaram-se os temas fundamentais da formação de imagens por ultrassom. Foram apresentados os princípios de funcionamento da geração e captura do pulso ultrassônico, e o modo como os ecos são formados e detectados pelo transdutor. Foi mostrado

como uma imagem bidimensional pode ser obtida através de diversos sinais do modo-A. Foram discutidas as condições para a reconstrução de imagens em tempo real com suas características e limitações físicas. No próximo capítulo são apresentados detalhes do modelo matemático para a implementação de algoritmos de reconstrução baseados em problemas inversos.

3 RECONSTRUÇÃO BASEADA EM PROBLEMAS INVERSOS

Neste capítulo são apresentados os fundamentos de reconstrução de imagens, considerado como um problema inverso, e o modelo do sistema de aquisição utilizado na reconstrução de ultrassom no modo pulso-eco. A partir do modelo analógico apresentado no capítulo anterior são introduzidos os princípios para sua discretização, a fundamentação teórica sobre problemas inversos e as maneiras de solucioná-lo.

3.1 FORMULAÇÃO DO PROBLEMA DIRETO

Um sistema de ultrassom pode ser modelado como um mapeamento de entradas e saídas segundo a Figura 9. As entradas representam as reflexões do meio que se deseja visualizar, e as saídas são os ecos produzidos.

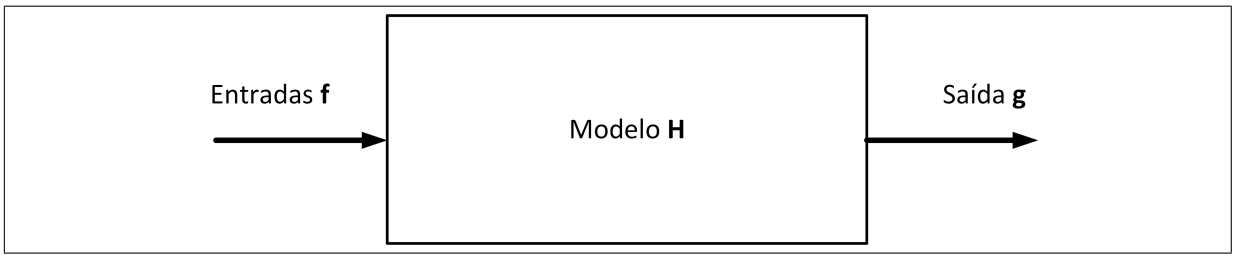


Figura 9: Modelo esquemático ilustrativo para um sistema real. Um objeto **f** através do modelo do sistema **H** produz um sinal de saída **g**.
Fonte: Própria.

Segundo Bertero e Boccacci (2008), quando procuramos o sinal **g** a partir do objeto **f** conhecido temos o que é chamado de problema direto. Por outro lado, quando conhecemos apenas o sinal **g** e queremos determinar os objetos que o originaram, temos um problema inverso. Esta modelagem pode ser usada tanto para inspeção não destrutiva (END), quanto para imagens médicas porque os objetos não são conhecidos, apenas as ondas acústicas refletidas

por eles.

Se consideramos que este sistema é linear e contínuo, podemos relacionar suas entradas e saídas pela integral de Fredholm de primeiro tipo, com entrada e saída em função da variável x (BERTERO; BOCCACCI, 2008; RAMM, 2005; GUARNERI, 2015):

$$g(x) = \int_{-\infty}^{+\infty} h(x;x')f(x')dx' \tag{16}$$

onde $g(x)$ representa a saída do sistema, $h(x;x')$ é a função que modela o sistema, também conhecida como *kernel*, neste trabalho também será chamada de PSF, e $f(x)$ é a entrada do sistema. Para o caso do sistema invariante em relação a x, temos que $h(x;x') = h(x-x')$, e a Equação (16) torna-se uma integral de convolução (GUARNERI, 2015; KARL, 2000).

Considerando que os dados serão processados digitalmente é necessário discretizar o modelo. Para S amostras, podemos escrever a Equação (16) como (KARL, 2000):

$$g_s = \int_{-\infty}^{+\infty} h_s(x')f(x')dx', \qquad 1 \leq s \leq S \tag{17}$$

onde $h_s(x') = h(x_s;x')$ é a PSF correspondente à s-ésima amostra.

A imagem $f(x)$ pode ser representada por um conjunto de coeficientes discreto e finito. Assumindo que a função de entrada pode ser adequadamente definida como uma soma ponderada de M funções base $\phi_m(x), m = 1,\ldots,M$, temos:

$$f(x) = \sum_{m=1}^{M} f_m \phi_m(x) \tag{18}$$

Pela Equação (18) vemos que a função de entrada $f(x)$, mesmo sendo contínua, é representada por um conjunto finito de coeficientes f_m. A quantidade total de coeficientes M determina o tamanho da imagem em *pixels*. Este conjunto pode ser organizado como um vetor coluna $\mathbf{f} = [f_1, f_2, \ldots, f_M]^T$ que após a resolução pode ser recomposto em uma imagem bidimensional para sua visualização.

Substituindo a Equação (18) na Equação (17) temos a forma discreta da relação entre

as amostras g_s observadas e os coeficientes f_m da imagem que se deseja reconstruir:

$$g_s = \sum_{m=1}^{M} H_{sm} f_m, \qquad 1 \leq s \leq S \tag{19}$$

onde H_{sm} é definida por:

$$H_{sm} = \int_{-\infty}^{+\infty} h_s(x')\phi_m(x')dx', \qquad 1 \leq s \leq S, \quad 1 \leq m \leq M \tag{20}$$

A partir das Equações (14), (18) e (19) é possível expressar o processo em sua forma matricial completa e discreta como será visto na próxima seção.

3.2 REPRESENTAÇÃO MATRICIAL DISCRETA

Na seção anterior foram apresentadas as definições para o modelamento do sistema considerando apenas um elemento sensor. Entretanto, nas aplicações de ultrassom os transdutores possuem vários elementos sensores, sendo possível selecionar sua quantidade de acordo com as necessidades de investigação (HEDRICK et al., 1995). Assim, é necessário expandir as definições para contemplar múltiplos elementos no modelo. Observando as Equações (14), (18) e (19) é possível considerar os K elementos do transdutor através de (KARL, 2000; VIOLA et al., 2008; ZANIN, 2011):

$$g_k[\mathbf{r}, t_i] = h_k[\mathbf{r}, t_i] f[\mathbf{r}] \tag{21}$$

onde $g_k[\mathbf{r}, t_i]$ são os ecos emitidos pelo ponto $\mathbf{r}$ e recebidos pelo elemento k do transdutor para todos os tempos t_i.

Para uma ROI de tamanho $M_1 \times M_2$, no plano cartesiano, com $M = M_1 \times M_2$ pontos a serem observados, modela-se o sinal $g_k[t_i]$ para o instante t_i como:

$$g_k[t_i] = \sum_{\mathbf{r} \in ROI} h_k[\mathbf{r}, t_i] f[\mathbf{r}] \tag{22}$$

onde $\mathbf{r} \in ROI$ são todos os pontos do espaço investigado.

A seguir, a Equação (22) é representada na sua forma matricial:

$$\mathbf{g}_k = \mathbf{H}_k \mathbf{f} \tag{23}$$

onde $\mathbf{g}_k = \left[g_k[1], g_k[2], ..., g_k[S] \right]^T$ constitui o vetor com todas as S amostras discretizadas no tempo e capturadas pelo k-ézimo elemento do transdutor. O vetor $\mathbf{f}$ representa a imagem com todos os pontos M, ordenados de tal forma que: $\mathbf{f} = \left[f[1,1], ..., f[M_1,1], f[1,2], ..., f[M_1,M_2] \right]^T$ e a matriz $\mathbf{H}_k$, de dimensões $S \times M$, representa os ecos dos M elementos da ROI capturados pelo k-ézimo elemento do transdutor.

A representação matricial do sistema pode ser descrita como:

$$\begin{bmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_K \end{bmatrix} = \begin{bmatrix} \mathbf{H}_1 \\ \vdots \\ \mathbf{H}_K \end{bmatrix} \mathbf{f}$$
$$\mathbf{g} = \mathbf{H}\mathbf{f} \tag{24}$$

onde $\mathbf{g}$ representa o vetor de dados, de dimensões $N = K \cdot S$, contendo os sinais vindos dos elementos do transdutor.

Em aplicações práticas precisamos considerar ainda que os dados podem ser distorcidos por pertubações ou ruído, assim, a forma completa pode ser expressa como (KARL, 2000):

$$\mathbf{g} = \mathbf{H}\mathbf{f} + \eta \tag{25}$$

onde η é um vetor com o ruído ou perturbações sobrepostas aos dados.

3.3 RECONSTRUÇÃO

Considerando a Equação (25), e que o vetor $\mathbf{g}$ incorpora os ruídos e perturbações coletados nas medições e a matriz $\mathbf{H}$ for quadrada e inversível, o vetor $\mathbf{f}$ pode ser encontrado a partir de:

$$\mathbf{f} = \mathbf{H}^{-1}\mathbf{g} \tag{26}$$

Contudo, esta solução é, geralmente, de pouca utilidade prática pois na maioria dos sistemas reais acontece alguma das seguintes condições, conhecidas como Condições de Hadamard (HANSEN, 1998; BERTERO; BOCCACCI, 2008):

1. inexistência de um vetor de entrada **f** capaz de gerar o vetor de saída **g**, seja pela existência de ruído ou quando o sistema de equações lineares é sobredeterminado, ou seja, quando a dimensão do vetor de entrada é menor que a do vetor de saída ($M < N$);

2. existência de infinitos vetores de entrada **f** capazes de gerar o vetor de saída **g**, pelo fato do sistema de equações lineares ser subdeterminado, ou seja, a dimensão do vetor de entrada é maior do que a do vetor de saída ($M > N$);

3. devido a matriz **H** ser mal-condicionada, pequenas variações no ruído tornam a solução instável, gerando soluções para a Equação (26) completamente diferentes [1].

Se alguma delas acontece o problema é dito mal-posto. De maneira geral, nos sistemas de ultrassom são identificadas as condições 1 e 3. Além do que, nos modelos de ultrassom a matriz H normalmente não é quadrada, de modo que não tem inversa. Assim, a solução através da Equação (26) (KARL, 2000) é inviável para aplicações práticas.

Uma alternativa para resolver a primeira condição de Hadamard seria considerar que inexistindo uma solução exata para a Equação (25), ainda assim é possível encontrar uma solução aproximada. Podemos utilizar o conceito de norma vetorial apresentada por Barrett et al. (2004) como uma medida de distância entre dois vetores, definida como $\|\mathbf{f}_1 - \mathbf{f}_2\|_p$ e onde $\|\cdot\|_p$ é a norma ℓ_p do vetor, representada genericamente como:

$$\|\mathbf{f}\|_p = \left[\sum_{m=1}^{M} |v_m|^p \right]^{\frac{1}{p}} \tag{27}$$

Desta forma, a melhor solução seria aquela onde a distância fosse mínima, ou seja, aquela na qual a distância entre a solução ideal e a aproximada fosse a menor possível. Assim,

[1] O condicionamento de uma matriz não singular **A** é medido pelo número de condicionamento, k, através de: $k_p(\mathbf{A}) = \|\mathbf{A}\|_p \|\mathbf{A}^{-1}\|_p$, onde p é a norma escolhida.Quando o k for grande a matriz é considerada mal-condicionada e quando ele for um ou próximo disso será dita bem-condicionada. O efeito disso sobre o sistema é que pequenas variações nos coeficientes da matriz geram grandes diferenças no resultado (BERTERO; BOCCACCI, 2008).

o problema passa ser uma questão de minimização que pode ser descrita como:

$$\hat{\mathbf{f}} = \arg\min_{\mathbf{f}} \|\mathbf{g} - \mathbf{H}\mathbf{f}\|_p^p \tag{28}$$

onde $\|\mathbf{g} - \mathbf{H}\mathbf{f}\|_p^p$ é chamada de função custo do problema.

Uma aplicação deste conceito é o método dos Quadrados Mínimos (*Least Squares*) (BARRETT et al., 2004; BOVIK, 2000; VOGEL, 2002) onde se utiliza a norma ℓ_2 na função custo:

$$\hat{\mathbf{f}}_{LS} = \arg\min_{\mathbf{f}} \|\mathbf{g} - \mathbf{H}\mathbf{f}\|_2^2 \tag{29}$$

Neste caso, tem-se um problema de otimização quadrática e, por ser um problema estritamente convexo, sua solução pode ser encontrada igualando-se seu gradiente a zero (VOGEL, 2002; BARRETT et al., 2004):

$$\begin{aligned} \nabla_{\mathbf{f}} \left[\|\mathbf{g} - \mathbf{H}\mathbf{f}\|_2^2 \right] &= 0 \\ 2\mathbf{H}^T\mathbf{H}\mathbf{f} - 2\mathbf{H}^T\mathbf{g} &= 0 \\ \mathbf{H}^T\mathbf{H}\mathbf{f} &= \mathbf{H}^T\mathbf{g} \end{aligned} \tag{30}$$

Assim, é possível estimar a imagem de ultrassom através de:

$$\hat{\mathbf{f}}_{LS} = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{g} \tag{31}$$

Embora a solução por Quadrados Mínimos seja uma maneira de contornar a primeira condição de Hadamard, se a matriz $\mathbf{H}$ for mal-condicionada a solução é instável na presença de ruídos. Fazendo $\mathbf{H}^+ = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T$, também chamada de pseudoinversa, temos (VOGEL, 2002):

$$\begin{aligned} \hat{\mathbf{f}}_{LS} &= \mathbf{H}^+(\mathbf{H}\mathbf{f} + \eta) \\ \hat{\mathbf{f}}_{LS} &= \mathbf{H}^+\mathbf{H}\mathbf{f} + \mathbf{H}^+\eta \\ \hat{\mathbf{f}}_{LS} &= \mathbf{f} + \mathbf{H}^+\eta \end{aligned} \tag{32}$$

onde é possível ver a amplificação do ruído, expresso por $\mathbf{H}^+\eta$. Uma forma de contornar este problema é acrescentar alguma informação prévia sobre a solução desejada, também

chamada de regularização, de maneira a estabilizar o problema para se obter uma solução estável (HANSEN, 1998).

3.3.1 REGULARIZAÇÃO

O problema da contaminação da solução pelo ruído presente nos dados ou do mal condicionamento da matriz precisa ser tratado. A adição de um parâmetro de regularização resolve este problema, por adicionar este parâmetro os algoritmos são chamados de regularizados.

Para Hansen (1998), exitem várias formas de se aplicar a regularização, porém, as dominantes são as que permitem algum resíduo na solução regularizada. De modo geral, possuem a seguinte forma (GUARNERI, 2015):

$$\hat{\mathbf{f}}(\lambda) = \arg\min_{\mathbf{f}} \{Q_1(\mathbf{f}) + \lambda Q_2(\mathbf{f})\} \tag{33}$$

onde λ representa o parâmetro de regularização e os termos $Q_1(\mathbf{f})$ e $Q_2(\mathbf{f})$ são, respectivamente, o termo que visa assegurar a fidelidade da resposta com os dados medidos e o termo regularizador, que adiciona informação prévia visando a estabilização da resposta.

O parâmetro λ é responsável pelo equilíbrio entre os termos, de modo que valores próximos a zero diminuem o efeito da regularização, resultando numa imagem reconstruída mais ruidosa e valores maiores aumentam a regularização, produzindo uma imagem mais suave (HANSEN, 1998).

Um método conveniente para se determinar o parâmetro de regularização e que foi popularizado por Hansen (1998), é o do gráfico da Curva L. Trata-se da plotagem da curva da norma do termo $Q_1(\mathbf{f})$ em função da norma do termo $Q_2(\mathbf{f})$, em escala logarítmica. Aqui o que se deseja é identificar um ponto da curva que represente o melhor equilíbrio entre os termos. Para Hansen (1998) este ponto se encontra na maior curvatura da região central, conforme ilustrado na Figura 10. Outra opção foi proposta por Zibetti et al. (2008) onde se utiliza o critério proposto por Reginska (2012), onde o ponto ótimo é o cruzamento da curva com uma reta de decaimento negativo, e a escala raiz quadrada. Um método que utiliza estes conceitos e segundo

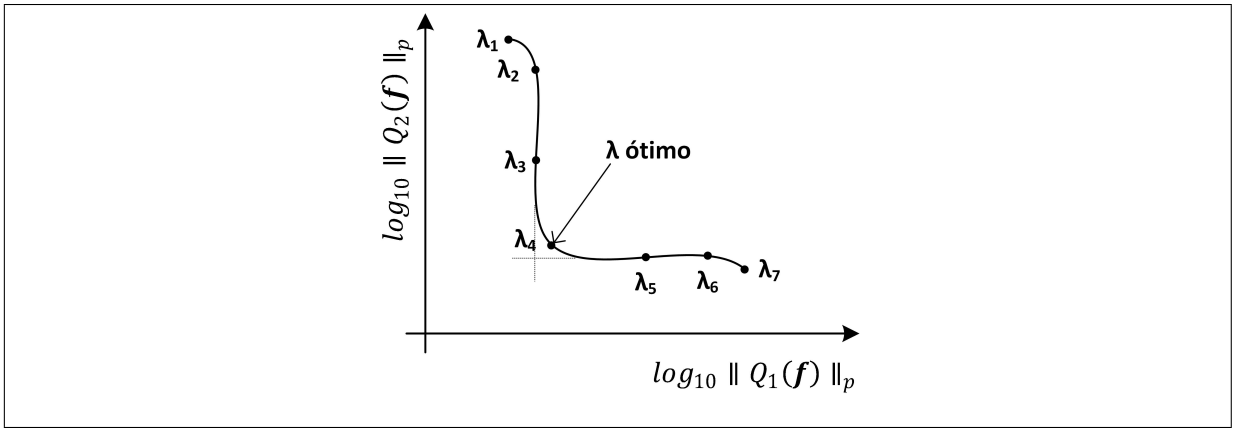


Figura 10: Curva L ilustrativa para $\lambda_1 < \lambda_2 < ... < \lambda_7$.
Fonte: Própria.

Karl (2000) é um dos mais referenciados, é o método de Tikhonov. A regularização de Tikhonov tem como ideia central o uso de informação *a priori* sobre a imagem a ser reconstruída. Isto é alcançado pela adição de um termo regularizador (*prior*) à função custo quadrática na Equação (29). Sua forma geral é (HANSEN, 1998; BARRETT et al., 2004; BERTERO; BOCCACCI, 2008):

$$\hat{\mathbf{f}}_{Tk} = \arg\min_{\mathbf{f}} \left[\|\mathbf{g} - \mathbf{H}\mathbf{f}\|_2^2 + \lambda^2 \|\mathbf{L}\mathbf{f}\|_2^2 \right] \tag{34}$$

onde $\mathbf{L}$, em geral, é um operador derivativo ou gradiente, de maneira que o termo de regularização indica uma medida de variabilidade da entrada $\mathbf{f}$; se os espaços nulos de $\mathbf{H}$ e $\mathbf{L}$ forem distintos, $N(\mathbf{H}) \cap N(\mathbf{L}) = \emptyset$, a solução existe e será única. Sua forma fechada pode ser expressa como (KARL, 2000):

$$\hat{\mathbf{f}}_{Tk} = \left(\mathbf{H}^T\mathbf{H} + \lambda \mathbf{L}^T\mathbf{L} \right)^{-1} \mathbf{H}^T\mathbf{g} \tag{35}$$

Fazendo $\mathbf{L} = \mathbf{I}$, sendo $\mathbf{I}$ a matriz identidade, o termo de regularização na Equação (34) estará medindo a energia de $\mathbf{f}$ e isto irá prevenir que os valores de $\mathbf{f}$ se tornem muito grandes como acontecia na solução não regularizada (BOVIK, 2000). Outra consequência desta escolha é que não será possível tentar reconstruir componentes da imagem que não estejam nos dados (KARL, 2000). Para o caso de $\mathbf{L} \neq \mathbf{I}$, a matriz $\mathbf{L}$ pode ser obtida, por exemplo, pela operação de diferenças finitas sobre o vetor $\mathbf{f}$, desta forma, o termo de regularização representa a medida de variação ou rugosidade da imagem e forçará uma limitação da energia das componentes de alta frequência, assumindo, assim, que a imagem deve ser suave.

A solução regularizada de Tikhonov permite uma melhor aproximação e maior estabilidade da solução. Porém, a melhor escolha do λ e da função de regularização são questões ainda em aberto para o caso do ultrassom.

Passarin et al. (2012) oferecem uma alternativa para isto empregando a norma ℓ_1, ao invés da norma ℓ_2, o que tem se mostrado interessante no problema de reconstrução de imagens de ultrassom. Neste caso, foi assumida uma distribuição Laplaciana para o *pior*, e a função de minimização pode ser descrita como (BOVIK, 2000; KARL, 2000; PASSARIN et al., 2012):

$$\hat{\mathbf{f}} = \arg\min_{\mathbf{f}} \left[\|\mathbf{g} - \mathbf{H}\mathbf{f}\|_2^2 + \lambda \|\mathbf{f}\|_1 \right] \tag{36}$$

Por apresentarem o termo de fidelidade quadrático, estes métodos são problemas de otimização quadráticos e a solução pode ser calculada se aplicando algoritmos iterativos, como será visto na próxima seção. A norma ℓ_1 mantém o problema convexo, mas ele deixa de ser estritamente convexo.

3.3.2 ALGORITMOS ITERATIVOS

Os métodos iterativos são uma alternativa para encontrar a solução da Equação 33. Segundo Karl (2000), eles oferecem algumas vantagens sobre os métodos de inversão direta: (i) soluções aproximadas podem ser obtidas com poucas iterações, consequentemente, com um custo computacional menor; (ii) eles costumam consumir menos memória do que as fatorizações ou inversões necessárias nos métodos exatos e (iii) muitos métodos iterativos são candidatos naturais para a paralelização possibilitando ganhos de desempenho adicionais.

A adoção de métodos iterativos permite a avaliação de um método com regularização ℓ_2-ℓ_2 e outro com regularização ℓ_2-ℓ_1, oportunizando a análise dos resultados obtidos por algoritmos distintos, tanto do ponto de vista de ultrassom como da eficiência em termos de paralelização.

Gradiente conjugado por equações normais (CGNE)

A Equação (29) dá a solução para o sistema linear de equações dado por:

$$\mathbf{H}^T\mathbf{H}\mathbf{f} = \mathbf{H}^T\mathbf{g}. \tag{37}$$

Esta forma é conhecida como Sistema de Equações Normais e está associada ao problema dos quadrados mínimos. Ela é tipicamente empregada quando o sistema é sobredeterminado.

O CGNE está descrito no Algoritmo (1):

Algoritmo 1: Gradiente Conjugado por Equações Normais pelo Erro

$\mathbf{r}_0 := \mathbf{g} - \mathbf{H}\mathbf{f}_0$;
$\mathbf{p}_0 := \mathbf{H}^T\mathbf{r}_0$;
for $i := 0, 1, \ldots,$ *until convergence* **do**
  $\alpha_i := \dfrac{\mathbf{r}_i^T\mathbf{r}_i}{\mathbf{p}_i^T\mathbf{p}_i}$;
  $\mathbf{f}_{i+1} := \mathbf{f}_i + \alpha_i\mathbf{p}_i$;
  $\mathbf{r}_{i+1} := \mathbf{r}_i - \alpha_i\mathbf{H}\mathbf{p}_i$;
  $\beta_i := \dfrac{\mathbf{r}_{i+1}^T\mathbf{r}_{i+1}}{\mathbf{r}_i^T\mathbf{r}_i}$;
  $\mathbf{p}_{i+1} := \mathbf{H}^T\mathbf{r}_{i+1} + \beta_i\mathbf{p}_i$;

No CGNE a atualização do vetor da imagem é $\mathbf{f}_{i+1} = \mathbf{f}_i + \alpha_i\mathbf{p}_i$. O vetor com o resíduo é atualizado por $\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i\mathbf{H}\mathbf{p}_i$ e a direção de busca através de $\mathbf{p}_{i+1} = \mathbf{H}^T\mathbf{r}_{i+1} + \beta_i\mathbf{p}_i$. Esta versão de algoritmo é conhecida como Método de Craig (SAAD, 2003). Ele minimiza energia do erro (dada pelo quadrado da sua norma ℓ_2) no espaço de Krylov $\mathscr{K}_i$.

***Fast iterative shrinkage-thresholding algorithm** (FISTA)*

Este algoritmo é derivado da família de algoritmos *Iterative Shrinkage-Thresholding* (ISTA) e busca solucionar a Equação (36). Ele é baseado em métodos de otimização convexa que empregam um operador de *Shrinkage-Thresholding* definido como (BECK; TEBOULLE, 2009b; ZIBULEVSKY; ELAD, 2010; VALENTE, 2017):

$$S_\alpha(x) = \begin{cases} 0 & , \alpha \geq |x| \\ x - \alpha \cdot sgn(x) & , \alpha < |x| \end{cases} \tag{38}$$

Por ser uma família de algoritmos considerados lentos muitas propostas de aceleração têm sido apresentadas. Beck e Teboulle (2009a) apresentaram uma forma de aceleração que garante uma taxa de convergência da ordem de $O(1/i^2)$, sendo i o número da iteração. Este algoritmo foi chamado de FISTA (BECK; TEBOULLE, 2009b) e está representado no Algoritmo (2):

Algoritmo 2: *Fast Iterative Shrinkage-Thresholding Algorithm*

$\mathbf{f}_0 := 0;$
$\mathbf{y}_0 = \mathbf{f}_0;$
$\alpha_0 = 1;$
for $i := 0, 1, \ldots,$ *until convergence* **do**
$$\mathbf{f}_{i+1} = S_{\lambda/c}\left(\mathbf{y}_i + \tfrac{1}{c}\mathbf{H}^T(\mathbf{g} - \mathbf{H}\mathbf{y}_i)\right);$$
$$\alpha_{i+1} = \frac{1+\sqrt{1+4\alpha_i^2}}{2};$$
$$\mathbf{y}_{i+1} = \mathbf{f}_{i+1} + \left(\frac{\alpha_i - 1}{\alpha_{i+1}}\right)(\mathbf{f}_{i+1} - \mathbf{f}_i);$$

De acordo com Elad (2010) o método FISTA e suas variações têm se mostrado eficientes na minimização ℓ_2-ℓ_1, principalmente em problemas de grandes dimensões, o que faz deles bons candidatos para utilização em problemas de ultrassom. Todos empregam os mesmo passos básicos: calculam o gradiente da parte quadrática da função custo e o operador proximal de um ponto especial (termo entre parênteses no primeiro passo do laço do algoritmo), que se resume ao *Shrinkage/Thresholding* no caso da norma L1. Tanto para o CGNE quanto para o FISTA, as condições de parada, número total de iterações ou convergência, estão descritos no Capítulo 6.

3.4 MODELO DE AQUISIÇÃO

Neste estudo, a construção da matriz de aquisição **H** é feita através do estabelecimento de uma ROI hipotética que representa todos os possíveis alvos dentro de uma determinada área. Os parâmetros para construção variam de acordo com as necessidades de imageamento, ou seja, quantidade total de pixels a serem reconstruídos, dimensão da área alvo de captura, número de elementos do transdutor a serem utilizados, frequência de amostragem e a quantidade de

disparos efetuados. Estes parâmetros são determinantes para a dimensão final da matriz.

Na Figura 11 é apresentado o modelo esquemático de construção da matriz **H**, onde para cada alvo hipotético (c,l) se captura uma PSF sintética simulada através da biblioteca *Field II* desenvolvida por Jensen (2004).

A resposta espacial **h** gerada para cada um dos pontos da ROI hipotética, com tamanho $K \times S$ é ordenada como um vetor de tamanho $(K \cdot S) \times 1$ e adicionada à coluna m da matriz **H**. Ao final a matriz **H** guarda as características físicas do modelo observado (VIOLA et al., 2008; ZANIN, 2011).

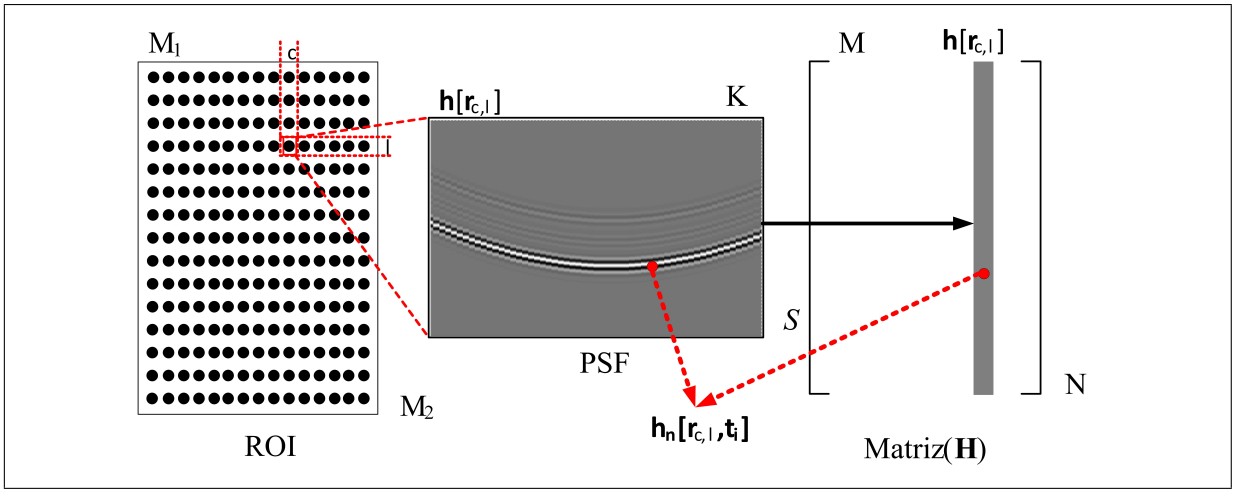


Figura 11: Modelo esquemático de construção da matriz **H**. Um ponto qualquer (c,l) gera uma PSF que é adicionada à coluna m da matriz **H**.
Fonte: Adaptado de (VIOLA et al., 2008).

Este modelo matricial foi empregado nas reconstruções de imagens no escopo deste trabalho. A avaliação compreende a utilização de dois conjuntos de dados distintos. Num primeiro momento se utiliza um conjunto de dados sintéticos construídos para validar os conceitos apresentados no Capítulo 5. Em seguida, o mesmo modelo aquisição é avaliado através de um conjunto de dados experimentais coletados por equipamento de ultrassonografia, obtidos pelo imageamento de um *phantom* de ultrassom para uso profissional, o *Phantom Fluke 84-317* (FLUKE-BIOMEDICAL, 2005).

3.5 CONCLUSÃO DO CAPÍTULO

Neste capítulo foram abordados os conceitos para a formulação do problema inverso, suas condições básicas de solução, algumas alternativas para se contornar as condições de Hadamard. Foram abordados, também, diversos métodos para se efetuar a reconstrução, como a regularização de Tikhonov, além dos métodos iterativos CGNE e FISTA. Descreveu-se, também, o modelo de aquisição empregado neste estudo. Procurou-se enfatizar as condições computacionais para a aplicação, bem como, as questões relativas à complexidade dos algoritmos envolvidos.

4 PROCESSAMENTO PARALELO EM GPU

Neste capítulo serão apresentados os conceitos fundamentais do processamento paralelo e suas características para desenvolvimento de soluções em GPU.

4.1 FUNDAMENTOS DO PROCESSAMENTO PARALELO

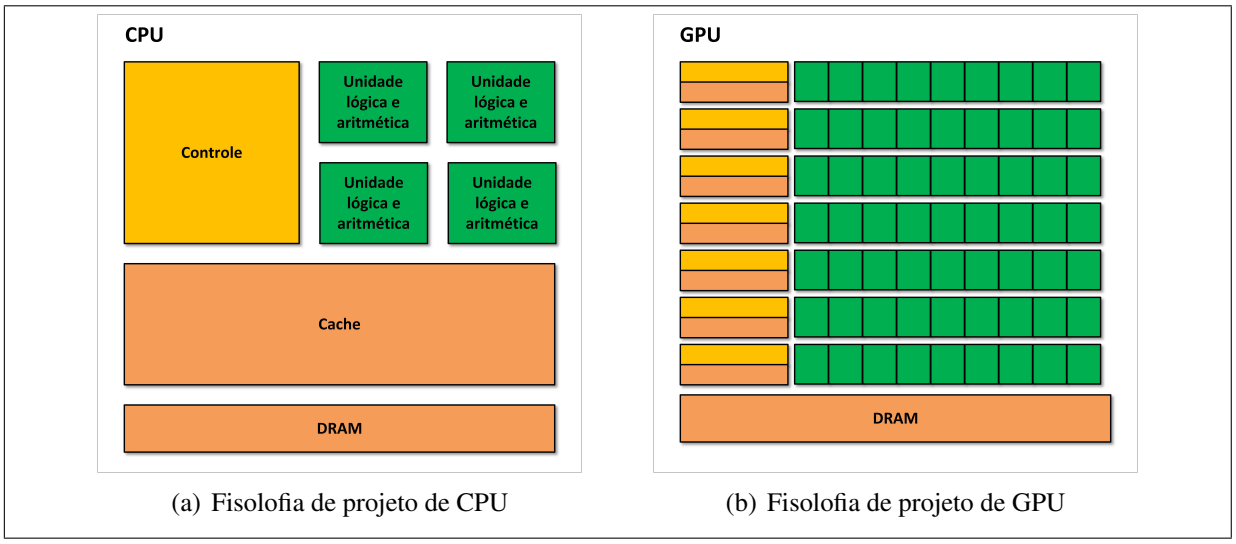
Os processadores baseados em uma única CPU obtiveram ganhos de desempenho e de redução de custos significativos por mais de duas décadas, mas estes avanços diminuíram nos últimos tempos por problemas com o consumo de energia e a dissipação de calor. Desde então, os fabricantes pesquisam modelos com múltiplas unidades de processamento, conhecidas como multi-núcleos (KIRK; HWU, 2010; NAVAUX, 2011).

Dentre os processadores multi-núcleos, as GPUs despertam interesse devido à sua capacidade de processamento e podem representar uma alternativa para melhorar o desempenho dos métodos de reconstrução de imagens de ultrassom existentes (STONE et al., 2008; DAI et al., 2010; EIDHEIM et al., 2005; JANG; DO, 2009; SCHIWIETZ et al., 2006bb; STONE et al., 2008).

Escrever código para processamento em paralelo de forma eficiente depende da compreensão da arquitetura computacional disponível e das diferentes filosofias de projeto (EIJKHOUT et al., 2014). O projeto de uma CPU é otimizado para o melhor desempenho de código sequencial. Ela utiliza sofisticadas estruturas lógicas de controle para que as instruções de uma *thread*[1] sejam executadas em paralelo, ou mesmo fora de sua ordem sequencial, mas ainda assim mantendo uma aparência de execução sequencial. As GPUs, por outro lado,

[1] É uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrencialmente (NVIDIA Corporation, 2012, 2014ba).

priorizam a execução maciça de *threads* oferecendo desempenho significativo na execução de um grande número de operações em ponto flutuante (KIRK; HWU, 2010). Estas diferenças podem ser observadas na Figura 12.



(a) Fisolofia de projeto de CPU (b) Fisolofia de projeto de GPU

Figura 12: As filosofias de projetos entre CPUs e GPUs são fundamentalmente diferentes. Fonte: Adaptado de (KIRK; HWU, 2010).

As GPUs apresentam uma velocidade de leitura e gravação na memória de acesso dinâmico e direto (DRAM) significativamente altas, como por exemplo, no caso da GPU utilizada neste trabalho que alcança 224 Gigabytes/segundo (GB/s), sendo outro ponto positivo para seu emprego (NVIDIA Corporation, 2014c).

4.1.1 TIPOS DE CONCORRÊNCIA E GRANULARIDADE

Existem duas formas principais de concorrência: a temporal e a de recursos ou espacial. Na concorrência temporal, o ganho de desempenho se dá pela sobreposição das execuções no tempo. Na concorrência de recursos, por sua vez, o ganho é resultado da quantidade de unidades de processamento que trabalham em paralelo (NAVAUX, 2011). M. J. Flynn em 1966, propôs uma classificação para os sistemas de computação, ilustrada na Figura 13 e que se tornou largamente empregada (VOTANO et al., 2004; EIJKHOUT et al., 2014; GRAMA et al., 2003):

- *Single Instruction Single Data* (SISD): uma das arquiteturas mais simples, consiste em um único fluxo de instruções operando sobre um dado;

- *Single Instruction Multiple Data* (SIMD): uma única unidade de controle dispara a execução de instruções para todas as unidades de processamento. A mesma instrução é executada de forma síncrona por todas as unidades. Em outras palavras, nesta arquitetura as unidades de processamento operam sob o controle centralizado de uma unidade de controle. Esta arquitetura também é conhecida como arquitetura matricial;
- *Multiple Instruction Single Data* (MISD) consiste na execução de diferentes instruções sobre um mesmo dado;
- *Multiple Instruction Multiple Data* (MIMD): cada unidade de processamento é capaz de executar diferentes programas de maneira independente umas das outras.

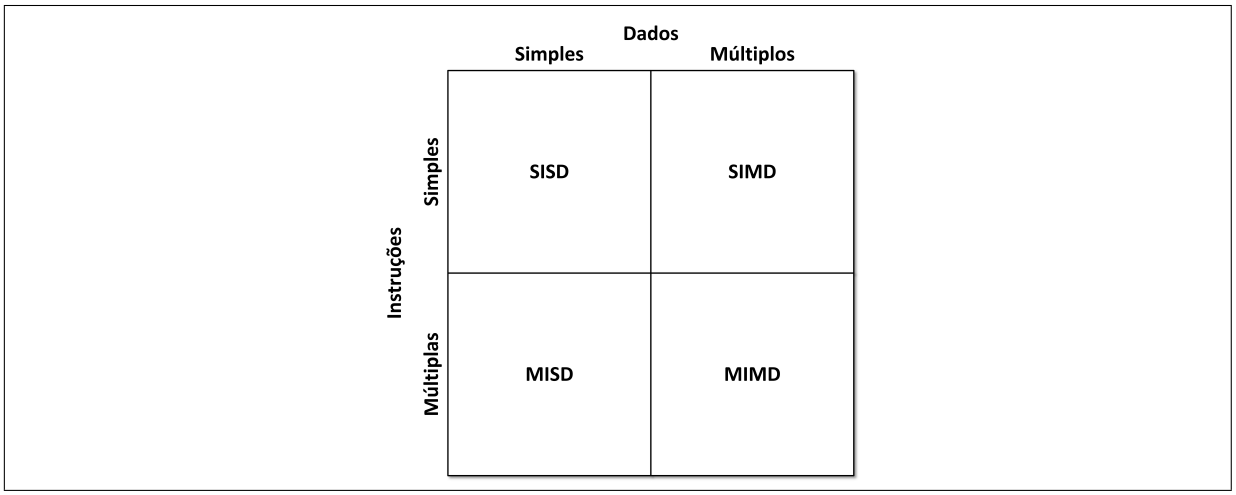


Figura 13: Modelo de classificação proposto por Flynn para sistemas de computação.
Fonte: Adaptado de (VOTANO et al., 2004).

Outro conceito importante, diz respeito aos níveis de granularidade das tarefas, que podem ser (NAVAUX, 2011):

- **Grossa**: decomposição do processo em um pequeno número de grandes tarefas;
- **Média**: decomposição do processo em um número médio de tarefas de tamanho médio;
- **Fina**: decomposição do processo em um grande número de pequenas tarefas;

Neste trabalho, os algoritmos foram projetados para terem granularidade fina, pois serão executados em GPU tirando proveito do paralelismo massivo oferecido pelas mesmas.

4.1.2 MEDIDAS BÁSICAS DE DESEMPENHO

As avaliações dos desempenhos dos algoritmos paralelos e das arquiteturas são realizadas através de certas medidas que indicam a efetividade do conjunto algoritmo-arquitetura escolhido. Basicamente são considerados dois aspectos nessas análises:

1. Desempenho do algoritmo ou da aplicação
2. Desempenho dos canais de interconexão ou de transferência de dados

Speedup

Indica o ganho de desempenho ou aceleração U de um algoritmo paralelo em relação à sua versão sequencial. Calculado através da razão entre seu melhor tempo sequencial e o melhor tempo de sua versão paralela (NAVAUX, 2011; KIRK; HWU, 2010; PACHECO, 2011; VOTANO et al., 2004):

$$U_p(w) = \frac{T(w)}{T_p(w)} \tag{39}$$

onde p é o número de processadores ativos utilizados, w é o trabalho (número total de operações) realizado e T é o tempo de execução (VOTANO et al., 2004).

Pela Lei Amdahl é possível estimar o ganho em *speedup* para um algoritmo de acordo com o possível nível de paralelismo alcançado (VOTANO et al., 2004):

$$U = \frac{1}{\tau + \frac{(1+\tau)}{p}} \tag{40}$$

onde τ representa a fração paralelizável de um algoritmo; esta medida indica o quanto de um algoritmo sequencial pode ser escrito em alguma forma paralela de execução (KIRK; HWU, 2010).

A Figura 14 ilustra o crescimento estimado do *speedup* de acordo com a Lei de Amdahl para diversos níveis de paralelismos. Cada algoritmo possui sua curva (mais ou menos acentuada) que depende da sua fração paralelizável e do número de processadores disponíveis, ressaltando porém, que nem sempre quanto mais processadores melhor (NAVAUX, 2011; VOTANO et al., 2004).

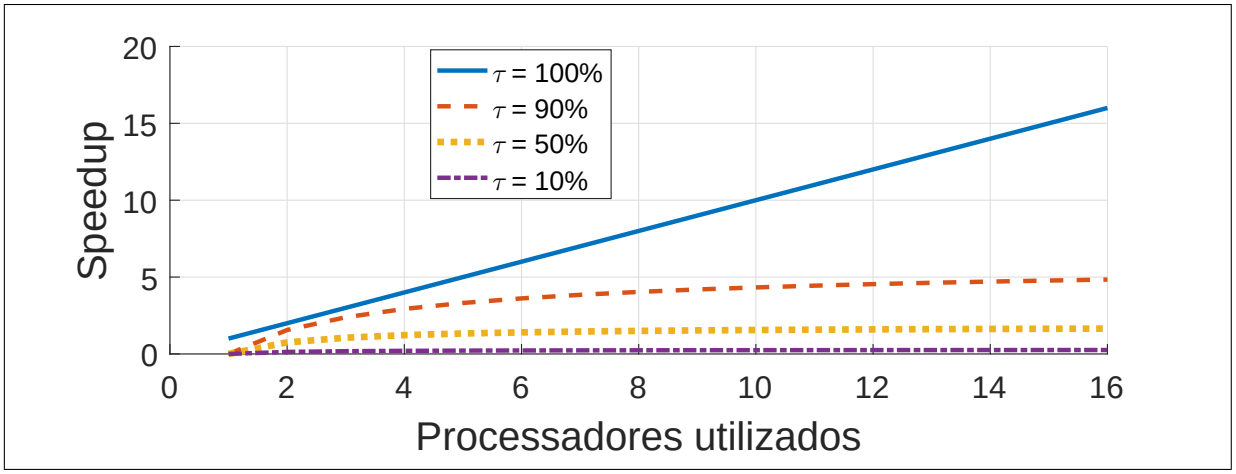


Figura 14: Ilustração do *speedup* possível segundo a Lei de Amdahl considerando $\tau = 100\%$, $\tau = 90\%$, $\tau = 50\%$ e $\tau = 10\%$.
Fonte: Adaptado de (VOTANO et al., 2004).

Eficiência

Este indicador apresenta a taxa de utilização média dos processadores utilizados. Em outras palavras, mostra se os recursos foram bem empregados. É obtido pela razão entre o *speedup* e o número de processadores utilizados (NAVAUX, 2011; KIRK; HWU, 2010; VOTANO et al., 2004):

$$E_p(w) = \frac{U_p(w)}{p} \tag{41}$$

onde o E representa a eficiência de uma implementação paralela.

Cada processador deveria permanecer 100% do tempo ativo, mas normalmente isto não acontece devido à espera por resultados de outros processadores ou transferências de dados. É importante observar que a melhor taxa de utilização não significa, necessariamente, o menor tempo de execução (VOTANO et al., 2004; PACHECO, 2011).

Amdahl em sua proposição não leva em consideração o tamanho do problema. Por outro lado, os pesquisadores Jonh Gustafson e Edwin Barsis propuseram que os programadores deveriam explorar o aumento de recursos para solucionar problemas ainda maiores no mesmo tempo. Ou seja, a eficiência estaria ligada ao tamanho do problema a ser resolvido: quanto maior o problema, mais eficiente a solução. Isto é conhecido como a Lei de Gustafson-Barsis (PACHECO, 2011).

Escalabilidade

Este indicador diz respeito à capacidade que um algoritmo tem de manter a mesma eficiência com o aumento do tamanho do problema, ou seja, a taxa de crescimento do problema é igual a taxa de crescimento da utilização dos recursos (PACHECO, 2011).

Latência

É o tempo que se leva para ter acesso a um bloco de memória, que é geralmente medida em nanosegundos (GRAMA et al., 2003). A latência varia com o aumento da quantidade de dados a serem transferidos, mas não de forma linear, pois o custo de preparo dos dados para envio (empacotamento e desempacotamento) não varia tanto quanto o custo para o envio propriamente dito (NAVAUX, 2011).

Largura de banda (*Bandwidth*)

A velocidade com que os dados podem ser transferidos da memória para o processador define a largura de banda do sistema (GRAMA et al., 2003; NAVAUX, 2011).

A Figura 15 ilustra o grafo computacional para uma árvore de operações em um possível algoritmo paralelo para soma de 16 valores. Assumindo que cada operação de soma consome uma unidade de tempo é possível se calcular medidas de efetividade deste algoritmo. Com $p = 8$, temos:

$$\begin{aligned} w &= 15 \\ T &= 15 \\ T_8(15) &= 4 \\ U_8(15) &= \frac{15}{4} = 3,75 \\ E_8(15) &= \frac{3,75}{8} = 47\% \end{aligned} \tag{42}$$

Cada um dos 8 processadores executa a soma no mesmo nível do grafo na mesma unidade de tempo, começando nas folhas indo até a raiz. A relativa baixa eficiência se deve ao baixo nível de paralelismo próximo da raiz (VOTANO et al., 2004).

Agora, assumindo que as operações perpendiculares são executadas pelo mesmo

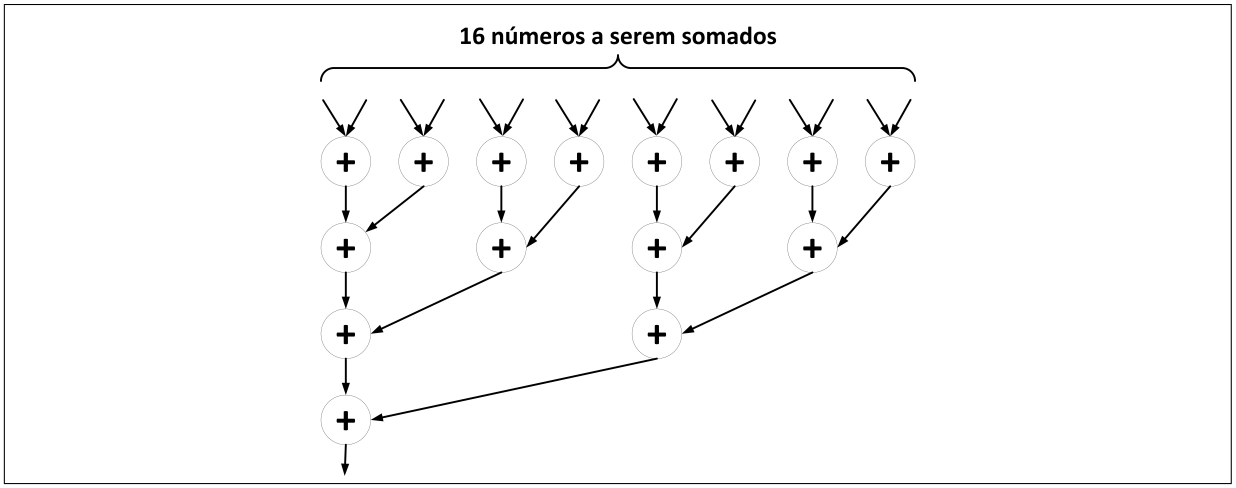


Figura 15: Exemplo da distribuição das operações para a realização de uma soma paralela de 16 números.
Fonte: Adaptado de (VOTANO et al., 2004).

processador, é necessário acrescentar os tempos de transferência dos resultados intermediários ao cálculos dos indicadores. Se cada transferência, representada por uma seta oblíqua, consumir uma unidade de tempo, temos:

$$\begin{aligned} w &= 22 \\ T &= 15 \\ T_8(22) &= 7 \\ U_8(22) &= \frac{15}{7} = 2,14 \\ E_8(22) &= \frac{2,14}{8} = 27\% \end{aligned} \tag{43}$$

Neste caso, a eficiência é ainda pior, pois as operações de transferência de dados constituem apenas uma sobrecarga de trabalho ao invés de operações úteis.

4.2 PROJETO DE ALGORITMOS PARALELOS

O desenvolvimento de algoritmos paralelos tem sido considerado como uma atividade intensiva em tempo e esforço. Isto pode ser atribuído à complexidade inerente de definir e coordenar tarefas concorrentes (GRAMA et al., 2003; VOTANO et al., 2004).

Considerando um processo hipotético como o observado no exemplo da Figura 16, que pode ser decomposto em tarefas menores a serem executadas em determinada ordem, ela ilustra

a decomposição ideal para este caso. É possível a execução concorrente de tarefas, como nos estágios 2 e 3, pelo fato destas serem independentes funcionalmente umas das outras.

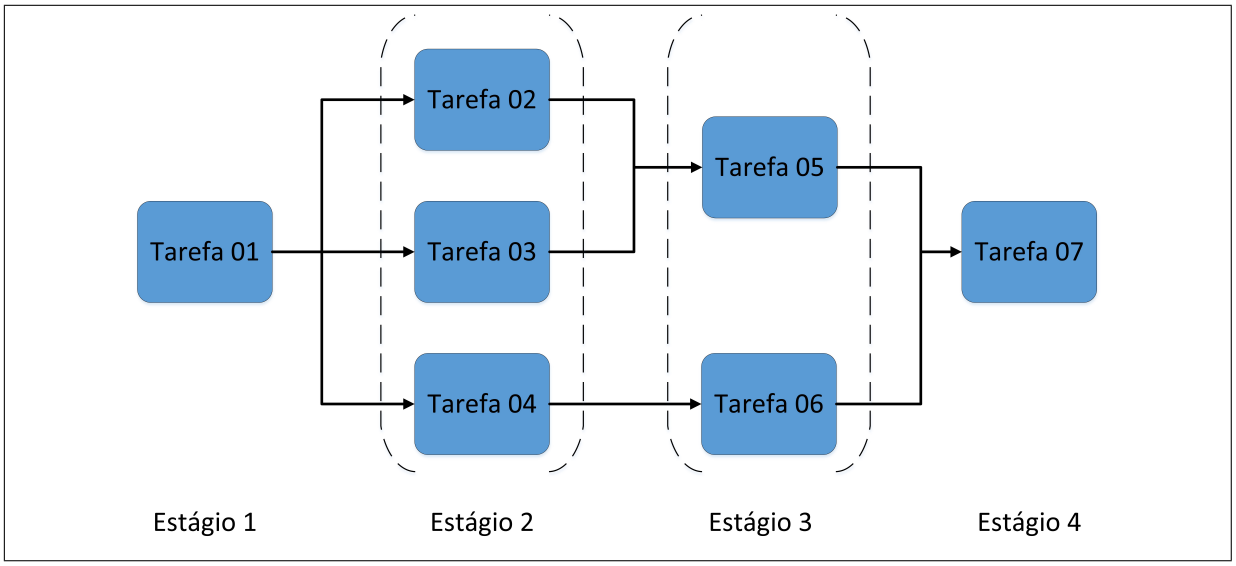


Figura 16: Decomposição e dependência funcional de um processo.
Fonte: Própria.

Supondo um processador hipotético com um único núcleo, este problema poderia ser resolvido pela execução sequencial como ilustrado na Figura 17, neste caso cada tarefa seria executada de forma independente mas na ordem apropriada, resultando numa operação em 7 estágios.

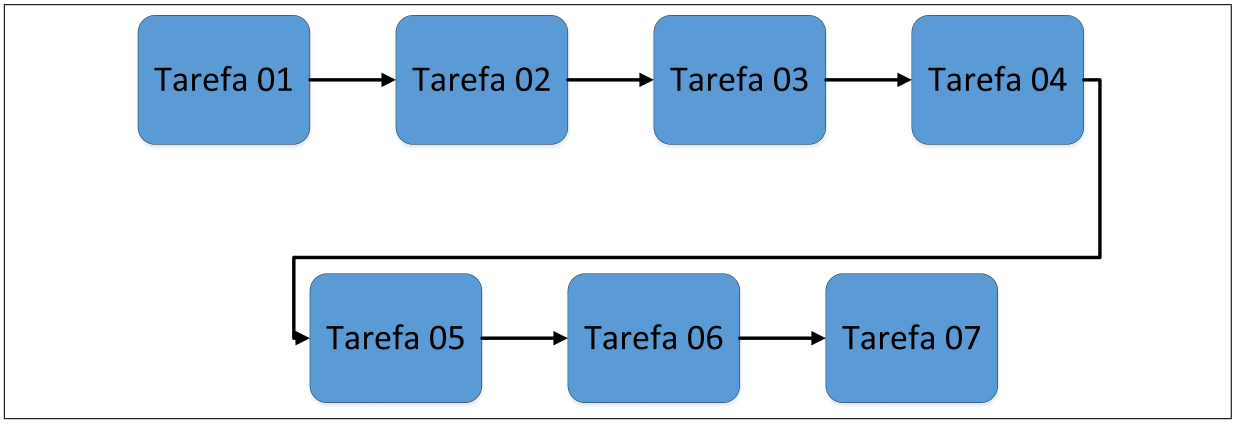


Figura 17: Execução sequencial das tarefas.
Fonte: Própria.

Se fosse um processador com 2 núcleos, seria necessário o ordenamento das tarefas de acordo com sua dependência funcional, isto é, caso uma tarefa dependa do resultado de outra elas devem ser executada de forma sequencial. Isto é visto na Figura 18, onde a nova ordem

das tarefas resultou em uma operação em 5 estágios. Esta atividade de definição da ordem e de como melhor aproveitar os recursos da arquitetura não é trivial, pois precisa levar em consideração a escalabilidade do paralelismo (VOTANO et al., 2004; GRAMA et al., 2003). É sempre desejável que um algoritmo paralelo possa ser escalável, ou seja, caso ele seja executado em uma arquitetura com um número maior ou menor de núcleos, ele precisa se adaptar a esta limitação, sem necessariamente envolver nova versão de código.

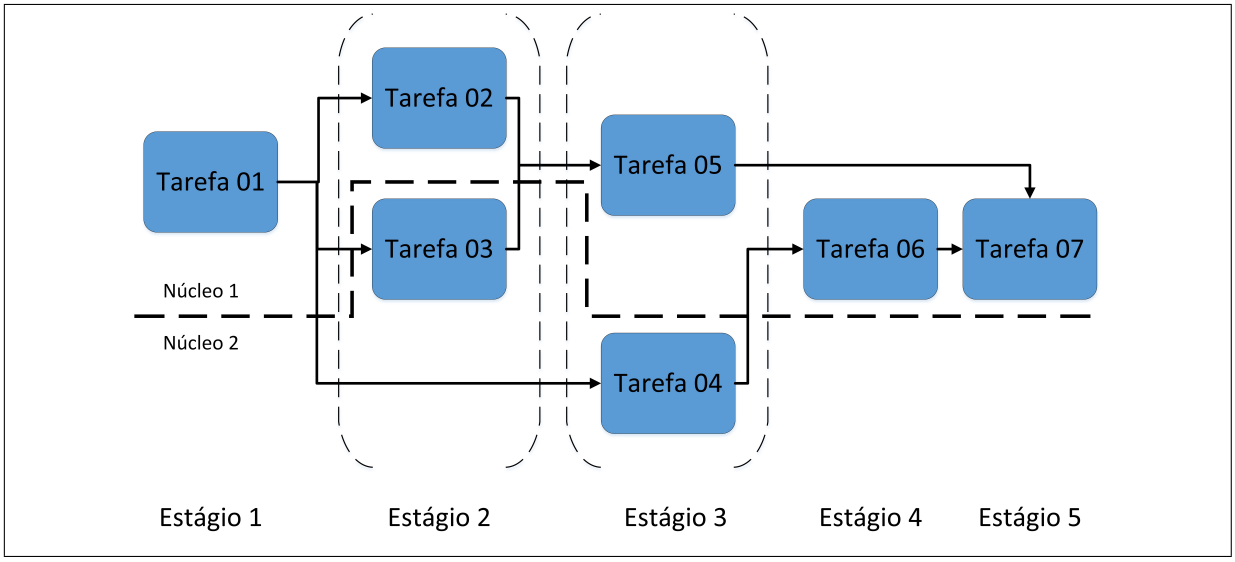


Figura 18: Execução paralela em dois núcleos.
Fonte: Própria.

Além da dependência funcional entre as tarefas, existe também a dependência de dados, isto é, as tarefas podem operar com dados independentes uns dos outros ou podem operar com o mesmo conjunto de dados.

4.2.1 ARQUITETURAS FÍSICAS DAS GPUS

Impulsionadas pelas necessidades da computação gráfica em tempo real e pelos avanços dos processadores multi-núcleos, que enfatizam mais o aumento do paralelismo do que das taxas de *clock*, as GPUs modernas estão na fronteira do desenvolvimento do paralelismo no nível de *chip* (GARLAND et al., 2008).

O processamento de imagens médicas, tanto em tempo real quanto no pós-processamento e diagnóstico se beneficiam diretamente destes avanços (DAI et al., 2010; JANG;

DO, 2009; STONE et al., 2008; SCHIWIETZ et al., 2006bb).

Neste trabalho foram empregados os modelos GTX 780 (Arquitetura Kepler) (NVIDIA Corporation, 2018a) e GTX 970 (Arquitetura Maxwell) (NVIDIA Corporation, 2014bb), ambos da NVIDIA Corporation, com processadores multi-núcleos com 2304 e 1664 núcleos, respectivamente. As arquiteturas físicas para ambas estão descritas em detalhes no Apêndice C.

4.3 PROGRAMAÇÃO PARALELA COM CUDA C

Desenvolvida pela NVIDIA Corporation (2014ba) e disponibilizada em 2006, a plataforma CUDA C de desenvolvimento paralelo é constituída por um modelo de programação e um ambiente de software que permite aos programados a criação de algoritmos paralelos escaláveis através de uma extensão da linguagem C (NVIDIA Corporation, 2014ba).

Com recursos para o desenvolvimento de programas com granularidade fina oferece a possibilidade de utilização massiva de *threads* em GPUs ao mesmo tempo que provê escalabilidade através dos recursos físicos disponíveis nas mais diversas GPUs atuais (GARLAND et al., 2008; NVIDIA Corporation, 2014ba).

O modelo de programação CUDA se baseia em dois pontos fundamentais: (i) oferecer recursos para programação paralela através da extensão de uma linguagem sequencial tradicional, neste caso o C/C++, com o mínimo de abstrações necessárias e (ii) escrever código altamente escalável capaz de executar milhares de *threads* simultâneas em centenas de núcleos de processamento (GARLAND et al., 2008; NVIDIA Corporation, 2014ba).

Do ponto de vista de escalabilidade, a Figura 19 ilustra o processo de escalonar um algoritmo dependendo do número de *Streaming Multiprocessors* (SMs), com o ganho de tempo em GPUs com mais SMs disponíveis (NVIDIA Corporation, 2014ba).

Um programa CUDA está organizado em uma parte do código chamada *host*, que é composta de uma ou mais funções sequenciais executando na CPU hospedeira e uma ou mais funções paralelas chamadas *kernel* configuradas para execução em GPUs. Um *kernel* executa uma porção de código sequencial em um conjunto de *threads* paralelas (NVIDIA Corporation,

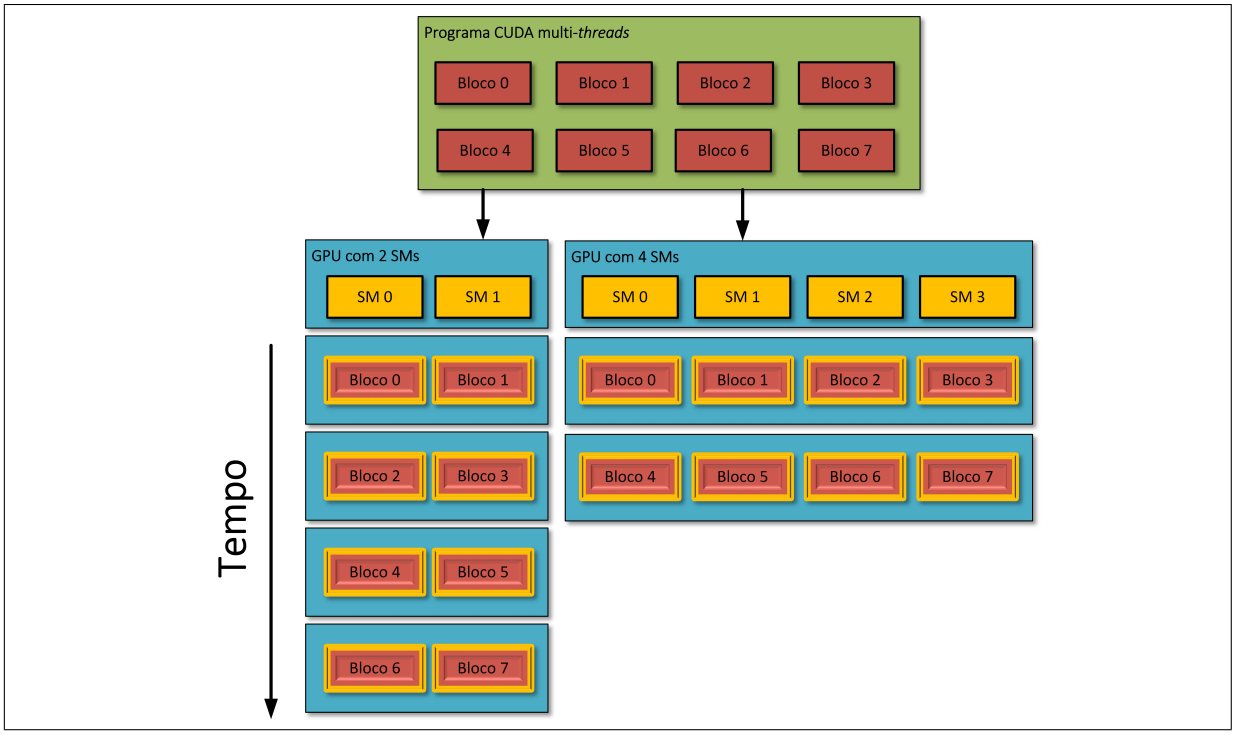


Figura 19: Ilustração de escalabilidade, onde uma mesma quantidade de *threads* pode ser executada em maior ou menor tempo dependendo da quantidade de SMs disponíveis na GPU. Fonte: Adaptado de NVIDIA Corporation (2014ba).

2014ba; GARLAND et al., 2008). Mais detalhes de utilização estão disponíveis no Apêndice D.

A CUDA tem se mostrado uma ferramenta importante no desenvolvimento de aplicações paralelas nas mais diversas áreas, oferecendo recursos para os pesquisadores otimizarem as suas soluções (GARLAND et al., 2008; NVIDIA Corporation, 2014ba).

4.4 ALGORITMOS PARALELOS PARA ÁLGEBRA LINEAR

Em Álgebra Linear existem muitas operações que podem ser otimizadas através de paralelismo, principalmente em GPUs. As operações de multiplicação de matrizes e vetores são, naturalmente, algoritmos candidatos para implementações paralelas. Por serem operações fundamentais em muitas aplicações de Álgebra Linear são objeto de melhorias constantes (GARLAND et al., 2008; NATH et al., 2010; LAI et al., 2013; TOMOV et al., 2010).

Versões otimizadas destas operações são disponibilizadas através da *Basic Linear Algebra Subprograms* (BLAS), muitas vezes customizadas para determinadas configurações

de *hardware* bem específicas. A BLAS organiza as operações em três níveis de complexidade (NVIDIA Corporation, 2014a):

- Nível 1: englobas as operações sobre vetores, como normas vetoriais, produtos internos, entre outras;
- Nível 2: contém as operações envolvendo matrizes e vetores, como por exemplo, multiplicação matriz-vetor;
- Nível 3: contém as operações envolvendo apenas matrizes, como a multiplicação matriz-matriz;

Neste trabalho são empregadas operações dos três níveis, entretanto, somente as operações dos níveis 2 e 3 foram customizadas para operarem dentro dos conceitos de simetria e redução de espaço, conforme detalhado no Capítulo 5. As principais operações utilizadas pelos métodos de reconstrução no escopo deste trabalho usam as rotinas de multiplicação matriz-vetor e matriz-matriz.

Na multiplicação de matrizes, o primeiro algoritmo a vencer a barreira de complexidade assimptótica do laço triplo de $O(N^3)$ foi apresentado por Volker Strassen em 1969 com uma complexidade de $O(N^{2.81})$ (STRASSEN, 1969; LAI et al., 2013). A partir daí muitos estudos gradualmente reduziram este valor, como por exemplo, o algoritmo de Coppersmith-Winograd com uma complexidade de $O(N^{2.38})$. Apesar deste ganho, este algoritmo não é prático para aplicações, pois, sua vantagem só é sentida para matrizes extremamente grandes (LAI et al., 2013). Assim o algoritmo de Strassen continua sendo o mais utilizado para implementações de rotinas para multiplicação de matrizes. Neste trabalho as principais rotinas customizadas utilizam este algoritmo como base.

Este conjunto de rotinas, também conhecidas como Multiplicação Geral de Matrizes (GEMM), são computacionalmente intensivas e, por isso, podem se beneficiar ainda mais dos recursos de paralelismo à medida que o tamanho das matrizes aumenta. Para isso é necessário selecionar adequadamente os parâmetros de otimização que variam de acordo com as dimensões das matrizes envolvidas. Este procedimento é conhecido como "*auto-tune*", onde os parâmetros

são calculados e ajustados automaticamente (NATH et al., 2010). Estas rotinas são detalhadas no Apêndice E.

4.5 CONCLUSÃO DO CAPÍTULO

Neste capítulo foram apresentados e exemplificados os conceitos fundamentais do processamento paralelo, em especial, os relacionados com as arquiteturas das GPUs. Apresentou-se a arquitetura física das duas plataformas de GPUs a serem empregadas neste trabalho, assim como sua organização lógica. Os aspectos referentes à decomposição funcional e à dependência de dados foram discutidos e apresentados. Discutiu-se a velocidade das unidades de processamento e como isso influencia o desempenho. Discutiu-se a complexidade computacional e temporal e como elas podem colaborar no projeto do algoritmo paralelo. Foram abordados os princípios de programação utilizando CUDA e as principais operações de álgebra linear e seu paralelismo. No Capítulo 5 são apresentados os conceitos para utilização da simetria como forma de reduzir o tamanho dos dados necessários para a reconstrução de imagens de ultrassom.

5 MODELO DE AQUISIÇÃO DE ULTRASSOM COM PSF SIMÉTRICA

Neste capítulo apresentamos o modelo de aquisição proposto nesta tese. O modelo usa a simetria das funções de espalhamento de ponto com a finalidade de reduzir a memória requerida pelas matrizes do sistema. Desta forma, a proposta reduz a necessidade de memória da GPU, permitindo que imagens de maior dimensão possam ser reconstruídas na GPU.

5.1 PROPOSTA DE UTILIZAÇÃO DA SIMETRIA DA PSF

Mesmo considerando as GPUs mais atuais, as restrições de memória ainda exigem soluções econômicas em termos de consumo de memória [1]. A seguir, serão apresentados o novo modelo de construção da matriz do sistema e as rotinas customizadas para a multiplicação matriz-matriz e matriz-vetor paralelas que empregam a matriz $\mathbf{H}$ reduzida.

Considerando-se dois alvos específicos da ROI posicionados em $\mathbf{r}_{c,l} = (x, -y)$ e $\mathbf{r}_{c',l} = (x, y)$, isto é, localizados em posições equidistantes em relação ao eixo do transdutor e espelhados em relação a um eixo central, e supondo que a geometria do transdutor e o pulso disparado são lateralmente simétricos, então a PSF resultante será simetricamente espelhada. As Figuras 20 e 21, ilustram como a matriz simétrica pode ser construída.

Um transdutor com características simétricas pode ser definido como aquele em que os elementos individuais e equidistantes do centro do transdutor tem o mesmo formato, dimensões e respostas ao impulso. Desta forma, é razoável supor que eles geram PSFs idênticas. Um pulso simétrico é aquele que excita os elementos equidistantes do centro com a mesma onda, simultaneamente, gerando assim um pulso de ultrassom simétrico.

[1] O modelo mais atual da NVIDIA, a Tesla V100, possui no máximo 32 GB de memória

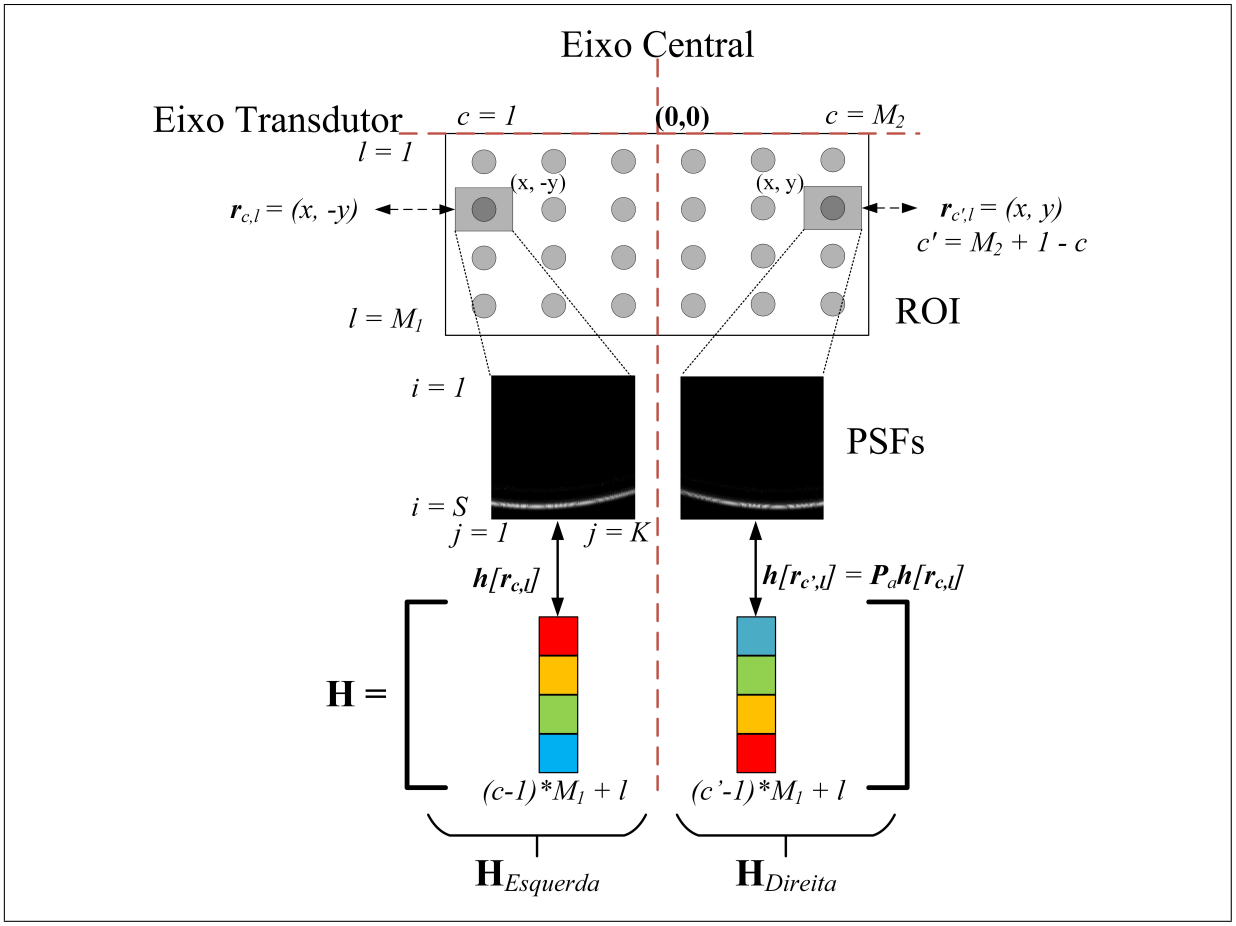


Figura 20: Formação da matriz **H** a partir de um modelo simétrico de PSF. Neste modelo se supõe que dois pontos da ROI equidistantes em relação à ambos os eixos têm uma PSF espelhada.
Fonte: Própria.

Cada coluna da matriz **H** é composta por blocos de sinais recebidos pelos K elementos do transdutor, de dimensões $S \times K$. Estes blocos são reordenados em formato de vetor e concatenados sequencialmente formando uma coluna da matriz, como ilustrado pelos blocos coloridos na Figura 20. Supondo uma PSF simétrica é possível calcular os valores correspondes a um determinado bloco de sinais aplicando um processo de espelhamento horizontal e vertical. Este espelhamento poderia ser feito como segue:

$$\begin{aligned} h_1(\mathbf{r}_{c,l},t) &= h_K(\mathbf{r}_{c',l},t) \\ h_2(\mathbf{r}_{c,l},t) &= h_{K-1}(\mathbf{r}_{c',l},t) \\ &\vdots \\ h_K(\mathbf{r}_{c,l},t) &= h_1(\mathbf{r}_{c',l},t) \end{aligned} \tag{44}$$

Através da Equação 44, seria possível a construção de matrizes de permutação $\mathbf{P}_a$ e $\mathbf{P}_b$ necessárias para realizar as permutações horizontais e verticais entre os blocos, para obter a submatriz simétrica correspondente.

Assim, é possível construir um modelo equivalente empregando matrizes de permutação e apenas uma das metades da matriz **H** original, como segue:

$$\begin{aligned} \mathbf{H}_{Direita} &= \mathbf{P}_a \mathbf{H}_{Esquerda} \mathbf{P}_b \\ \mathbf{H} &= \left[\mathbf{H}_{Esquerda} | \mathbf{H}_{Direita} \right] \end{aligned} \tag{45}$$

onde $\mathbf{H}_{Esquerda}$ é a metade esquerda da matriz original, de dimensões $N \times M/2$, como visto na Figura 20.

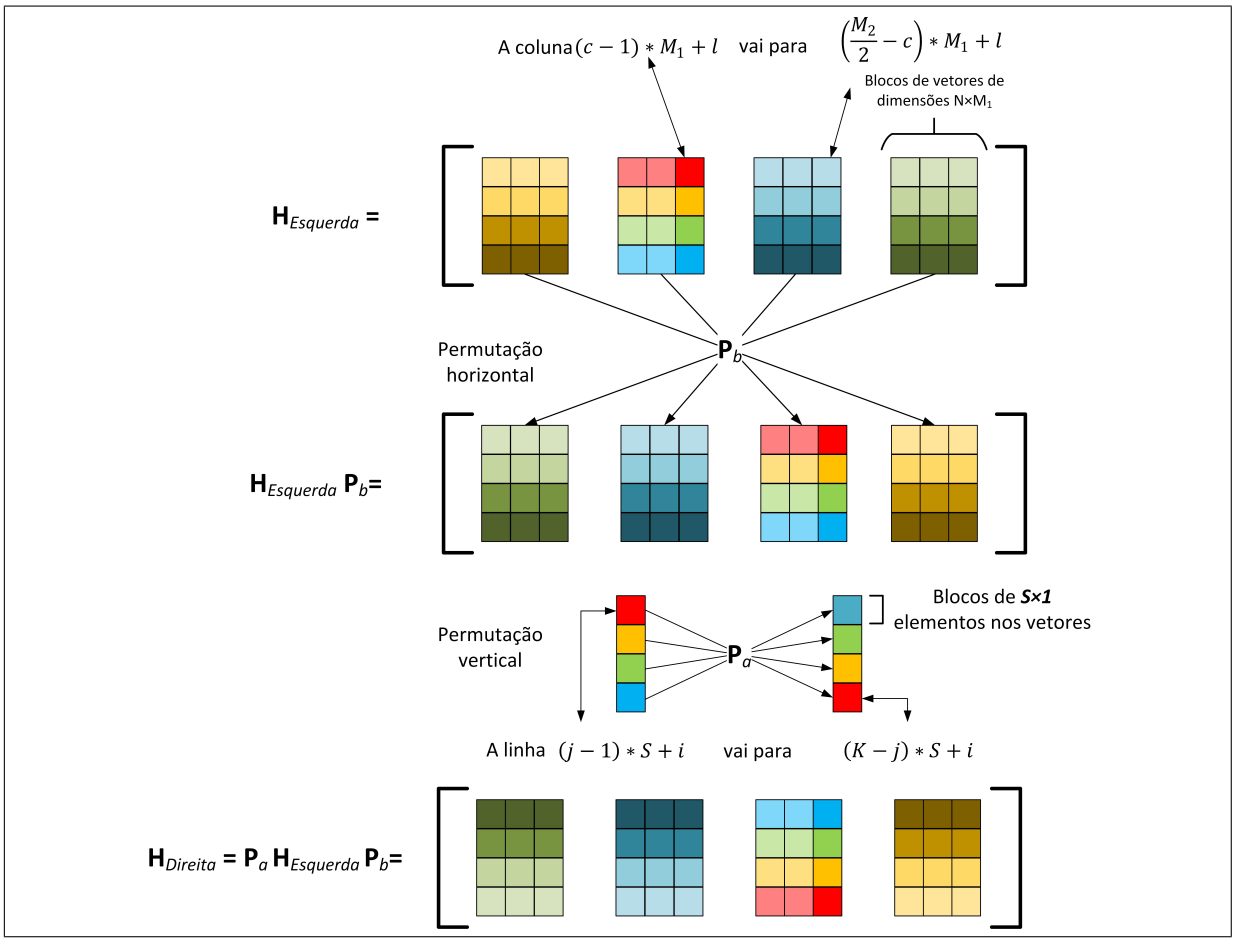


Figura 21: Processo de permutação de blocos para a composição do lado simétrico da matriz **H**. Este procedimento é composto de permutações horizontais e verticais que reposicionam os blocos de sinais recebidos de acordo com a posição do alvo e do elemento do transdutor.
Fonte: Própria.

A matriz $\mathbf{P}_b$ executa o equivalente a uma permutação esquerda-direita das PSFs

(blocos) relativas aos *pixels* da parte direita da imagem, obtendo a coluna da matriz associada ao ponto (c',l) a partir da coluna da matriz associada ao ponto (c,l). Esta é uma matriz de dimensões $M/2 \times M/2$ e é composta por apenas a metade dos elementos depois do reordenamento para vetor. $\mathbf{P}_b$ é composta basicamente de zeros exceto nas posições a serem permutadas, como segue:

$$\mathbf{P}_b(n,m) = \begin{cases} 1, & \text{para } n = \left(\frac{M_2}{2} - c\right) * M_1 + l \quad \text{e} \quad m = (c-1) * M_1 + l \\ 0, & \text{demais} \end{cases} \tag{46}$$

para $1 \leq c \leq M_2/2$ e $1 \leq l \leq M_1$.

Já a matriz $\mathbf{P}_a$ faz o equivalente a uma permutação direita-esquerda dos sinais capturados pelos diferentes elementos do transdutor. Esta permutação assume que os sinais, quando reordenados como uma matriz, contêm o indexamento temporal na direção vertical e a posição do elemento na horizontal, como ilustrado na Figura 21. $\mathbf{P}_a$ é uma matriz de dimensões $N \times N$ e é aplicada a todos os elementos da PSF após o reordenamento como vetor. De forma semelhante a $\mathbf{P}_b$, a matriz $\mathbf{P}_a$ também é composta por zeros, exceto para os elementos a serem permutados, de acordo com:

$$\mathbf{P}_a(n,m) = \begin{cases} 1, & \text{para } n = (K-j) * S + i \quad \text{e} \quad m = (j-1) * S + i \\ 0, & \text{demais} \end{cases} \tag{47}$$

para $1 \leq j \leq K$ e $1 \leq i \leq S$.

Embora o modelo utilizando as matrizes de permutação possa revelar os conceitos envolvidos na simetria e como ela pode ser alcançada a partir de apenas metade dos dados, ele não é prático para uma implementação em GPU. A criação dessas matrizes e seu armazenamento consumiriam tempo e memória. Desta forma, é necessário construir uma solução computacional para GPU que aplique este modelo sem a criação destas matrizes. Isto será abordado na próxima seção.

5.2 CUSTOMIZAÇÃO DAS OPERAÇÕES DE MULTIPLICAÇÃO PARALELA EM GPU

Nesta seção, são vistos os procedimentos para a customização das rotinas de multiplicação matriz-matriz e matriz-vetor para executarem aplicando os conceitos de simetria, discutidos na Seção 5.1. A customização permite a exploração da simetria sem a necessidade de construir as matrizes de permutação $\mathbf{P}_a$ e $\mathbf{P}_b$, isto contribui significativamente para o desempenho final da solução.

Observando os métodos de reconstrução, descritos nos Algoritmos 1 e 2, pode-se ver que as operações matriciais necessárias para a sua execução são os produtos matriz-vetor e vetor-vetor. Considerando que a multiplicação matriz-vetor pode ser vista como um sub-caso da multiplicação matriz-matriz, mesmo esta não sendo necessária para a implementação dos algoritmos, sua customização também foi realizada para fins de completude deste trabalho.

Considerando que o fluxo principal dos algoritmos seria executado através de *scripts* MATLAB (*MATrix LABoratory*), software produzido pela Mathworks Inc e voltado para o cálculo numérico (MATHWORKS, 2016), apenas o código referente a cada operação de multiplicação foi customizado.

A integração das rotinas com o ambiente MATLAB foi feita através dos recursos de compilação de funções C/C++ próprios do mesmo. Assim, foram construídas funções específicas para cada operação que são posteriormente chamadas pelos *scripts* com os algoritmos.

Conforme discutido no Capítulo 4, para o fluxo principal das rotinas foi tomado como base os algoritmos disponíveis na biblioteca BLAS "*Matrix Algebra on GPU and Multicore Architetures*"(MAGMA) (TOMOV et al., 2010). Considerando que esta é uma biblioteca de uso geral e muito mais amplo do que o escopo deste trabalho, as rotinas foram modificadas e seus códigos reunidos em arquivos únicos para cada tipo de operação. Assim, para operação de multiplicação simétrica matriz-matriz em precisão simples foi utilizada a função *symm_sgemm* e para o produto matriz-vetor a função *symm_sgemv*.

A Figura 22 apresenta o código C customizado que realiza o espelhamento simétrico

do índice de acordo com a posição do elemento na matriz **H**. Este código é parte do *kernel* principal da rotina de multiplicação e executa completamente em GPU.

```
//////////////////////////////////////////////////////////////////
/*
Calcula o índice de acordo com os parâmetros de simetria da matriz H.

   Argumentos Entrada:
      ind --> índice original do elemento da matriz H
      M  --> tamanho original da matriz H
      Ne    --> número de elementos do transdutor
      S  --> tamanho da amostra

   Argumento de saída:
      ind --> novo valor do índice do elemento pela simetria
*/
__device__ int GetElementIndex(
int ind, const int M, const int Ne, const int S)
{
   int cb = 0, cf = 0, cn = 0;
   if (ind < (M * Ne * S / 2))
      {
         return ind;
      }
   else
      {
         cb = ind / S + 1;
         cf = ind % S;
         cn = (Ne*M) - cb + 1;
         ind = (cn - 1)*S + cf;

         return ind;
      }
}
```

Figura 22: Código GPU customizado para recalcular os índices de acordo com a simetria. Fonte: Própria.

Este código é chamado a partir do fluxo principal das rotinas e recebe como parâmetro o endereço do elemento, o tamanho original da matriz, o número de elementos do transdutor e o tamanho da amostra. Este trecho de código verifica o endereço original do elemento e recalcula seu valor, caso seja necessário, realizando a operação de espelhamento simétrico *en passant*.

A Figura 23 apresenta o ponto de entrada para as rotinas customizadas, este modo de chamada permite a integração do código C/C++ escrito para GPU com os *scripts* do MATLAB. A resolução dos nomes das função é feita pelo MATLAB através dos nomes dos arquivos físicos criados.

```
//////////////////////////////////////////////////////////////////
/*
*  Função de integração MatLab C/C++/CUDA.
*
*  Esta função deve verificar os parâmetros de entrada e saída,
*  tratar os dados recebidos e preparar os dados para devolução ao
*  MatLab.
*
*  Tratamento direto de ponteiros alocados em GPU pelo script MatLab.
*
*  Argumentos Entrada:
*       argin[0] --> matriz H
*       argin[1] --> vetor g
*       argin[2] --> vetor f
*       argin[3] --> Total de linhas da matriz H (N)
*       argin[4] --> Total de colunas da matriz H (M)
*       argin[5] --> Nro elementos (Ne)
*       argin[6] --> Nro amostras (S)
*     argin[7] --> Calcular H transposta (0 - Normal, 1 - Transposta);
*/

void mexFunction(    int nargout,
                     mxArray *argout[],
                     int nargin,
                     mxArray *argin[] )
```

Figura 23: Definição da função de integração com o MATLAB.
Fonte: Própria.

Na Figura 24 é possível ver um diagrama com as principais tarefas do processo de reconstrução e onde são executadas. Para cada configuração do protocolo de reconstrução é gerada uma versão específica da matriz **H** e armazenada com os parâmetros utilizados. Em seguida esta matriz é carregada na memória da CPU e é executado o processo de redução de acordo com o conceito de simetria. A matriz reduzida é então copiada para a memória da GPU juntamentos os dados do sinal coletado. O algoritmo de reconstrução é então executado, com parte executando em CPU e parte em GPU. A coordenação do fluxo principal de execução é realizado na CPU. Finalmente o vetor com a imagem reconstruída é transferido da memória da GPU para a memória da CPU.

A principal vantagem desta abordagem é a flexibilidade oferecida pela função de espelhamento simétrico dos índices; esta função poderia ser facilmente substituída por uma função geradora da PSF de acordo com os parâmetros de imageamento. Esta possibilidade não foi explorada no escopo deste trabalho, mas ficou posta como um possível trabalho futuro. Pelo

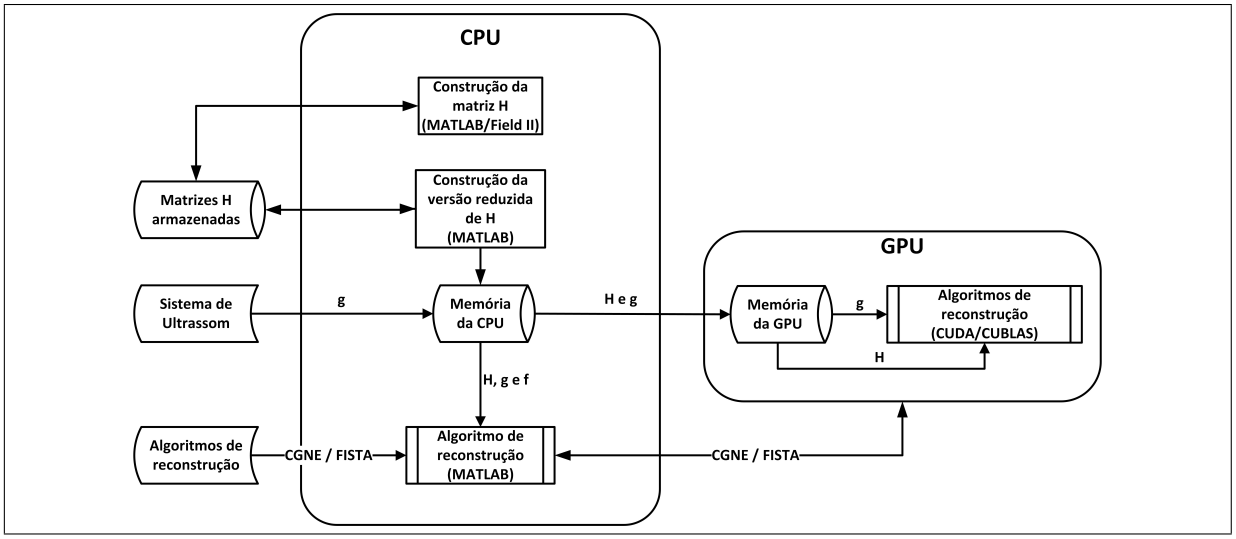


Figura 24: Diagrama de componentes do sistema de reconstrução de imagens. A matriz **H** é construída antecipadamente (*off-line*) e armazenada, uma matriz para cada configuração de imageamento. A matriz é então copiada para a memória da CPU e é executado o procedimento de redução conforme o conceito de simetria. Parte do algoritmo de reconstrução executa em CPU e parte em GPU.
Fonte: Própria.

fato deste procedimento ocorrer durante a operação de multiplicação matriz-matriz ou matriz-vetor, estas operações chamam esta função exatamente no momento em que o dado é necessário. Desta forma, ele poderia tanto realizar o espelhamento do índice quanto retornar o próprio valor a ser empregado na operação.

5.3 REDUÇÕES BÁSICAS

Uma maneira de se reduzir o volume de dados nas operações é a utilização de variáveis com precisão simples (32-bits) ao invés de precisão dupla (64-bits). Esta redução é, até certo ponto, trivial e intuitiva, pois permite reduzir o volume à metade sem praticamente nenhum esforço adicional de desenvolvimento.

Como muitas GPUs são otimizadas para processar dados de precisão simples, esta opção apresenta uma vantagem clara em termos de desempenho (NVIDIA Corporation, 2014bb, 2018a). Como é visto no Capítulo 6 os valores teóricos possíveis (em FLOPS), nas GPUs empregadas neste trabalho, apresentam uma diferença significativa entre a precisão simples e a dupla.

Considerando que os valores seriam truncados em uma determinada precisão foi necessário avaliar o quanto isto impactaria na qualidade da imagem final. Outro efeito indesejável poderia ser uma piora na convergência no caso dos métodos iterativos de reconstrução.

Outra opção para se diminuir a utilização de memória seria a operação no formato esparso. Neste caso os dados poderiam ser representados através de armazenamento de dados em matrizes esparsas. Neste trabalho foram projetados experimentos para se avaliar as várias possibilidades de emprego de matrizas esparsas, combinando dados em precisão simples e dupla.

5.4 CONCLUSÃO DO CAPÍTULO

Neste capítulo foram apresentados os conceitos envolvidos na contribuição deste trabalho para o imageamento por ultrassom utilizando problemas inversos. Foram discutidas as reduções básicas de tamanho possíveis para a economia de espaço de memória e suas implicações. Foram apresentados os conceitos envolvidos na redução de dados através da utilização das característica de espelhamento simétrico da PSF e as customizações realizadas para a sua implementação em um sistema de ultrassom experimental.

6 MATERIAIS E MÉTODOS - EXPERIMENTOS E RESULTADOS

Neste capítulo são apresentados os experimentos mais relevantes realizados neste trabalho e seus resultados. Foram executados experimentos com dados simulados e experimentais com o objetivo de validar e avaliar o desempenho da abordagem simétrica em relação as abordagens de reduções básicas discutidas no capítulo anterior.

6.1 AMBIENTE EXPERIMENTAL

Para a realização das simulações e ensaios foi preparado um equipamento com o hardware e software necessários para a execução das reconstruções. As rotinas de reconstruções e demais experimentos foram escritas em MATLAB. A Tabela 2 apresenta os detalhes da configuração.

Tabela 2: Configurações do computador utilizado.

Descrição	Configuração
CPU	Intel(R) Xeon(R) CPU E5-1607 v2 @ 3.00 GHz (36GB)
GPU	GeForce GTX 970 (4GB)
	GeForce GTX 780 (3GB)
Software	Microsoft Windows 10 Pro (Build 15063.608)
	MATLAB R2016a
	CUDA 8.0
	Field II v3.4

Fonte: Própria.

As GPUs empregadas nos ensaios foram a GeForce GTX 780 (NVIDIA Corporation, 2018a) e a GeForce GTX 970 (NVIDIA Corporation, 2014bb) cujas especificações estão detalhadas na Tabela 3.

Tabela 3: Especificações técnicas das GPUs.

Descrição	GeForce GTX 780	GeForce GTX 970
Processadores	12	13
Memória	3GB	4GB
Tamanho máximo do bloco de *threads*	[1024 1024 64]	[1024 1024 64]
Velocidade de escrita e gravação	224,536 GB/s	135,472 GB/s

Fonte: (NVIDIA Corporation, 2018a, 2014bb).

Para os experimentos que empregavam dados reais, os mesmos foram capturados utilizando o equipamento desenvolvido pela Verasonics, modelo Vantage 128, operando com o transdutor L11-4v (VERASONICS, 2015). Estes equipamentos se encontram disponíveis no laboratório de ensaios de ultrassom LUS/CPGEI/UTFPR. Neste trabalho foram utilizados apenas os 64 elementos centrais dos 128 elementos disponíveis no transdutor, isto se deu para minimizar a quantidade de dados a serem tratados. Os principais parâmetros utilizados estão descritos na Tabela 4.

Tabela 4: Parâmetros para o sistema de imageamento por ultrassom.

Descrição	Configuração
Número de elementos do transdutor	128
Número de elementos do transdutor usados	64
Frequência de ultrassom	6,25 MHz
Largura de banda do transdutor	3,84 MHz
Largura de banda fracionária	61,44%
Frequência de amostragem	25 MHz
Velocidade do som considerada	1.540 m/s
Comprimento de onda	0,2464 mm

Fonte: (VERASONICS, 2015) e (FLUKE-BIOMEDICAL, 2005).

Como dispositivo de referência (*phantom*) foi utilizado o modelo 84-317 ("*Multipurpose Tissue/Cyst Ultrasound Phantom*"), fabricado pela Fluke Biomedical, também disponível no laboratório LUS (FLUKE-BIOMEDICAL, 2005). Esse *phantom* contém grupos de alvos com características específicas para diversos tipos de testes de respostas ultrassônicas. Ele imita um meio com respostas semelhantes às fornecidas pelo parênquima hepático humano, com a mesma atenuação, velocidade de propagação e espalhamento. Este dispositivo está

ilustrado na Figura 25, assim como a definição da ROI para o escopo deste trabalho.

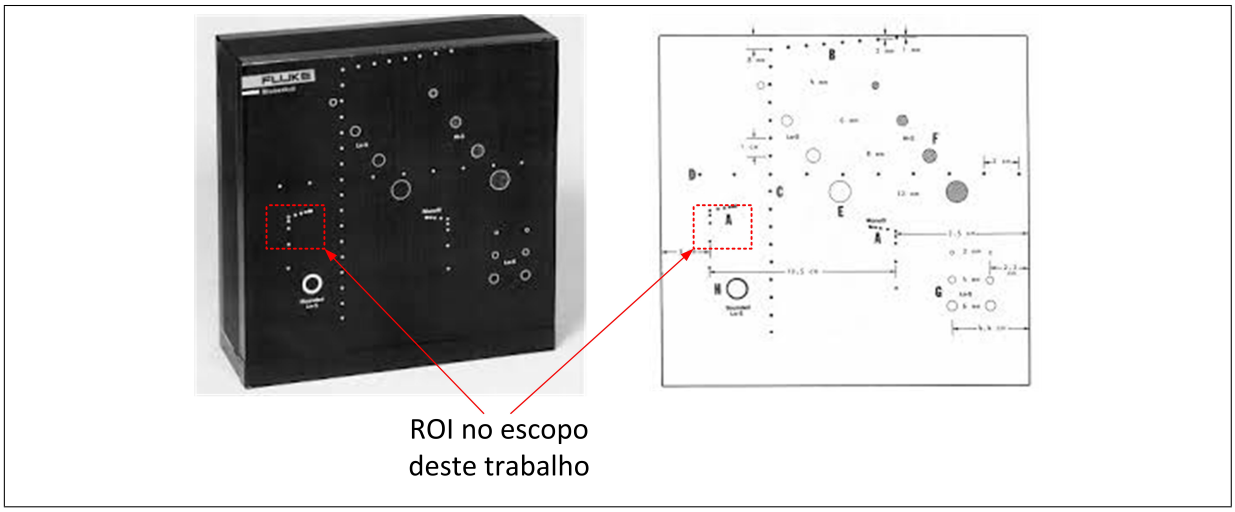


Figura 25: *Phantom* Fluke 84-317.
Fonte: Adaptado de (FLUKE-BIOMEDICAL, 2005).

Uma visão esquemática da ROI selecionada para este trabalho se encontra na Figura 26, onde as dimensões, tamanhos e distâncias representadas estão de acordo com as especificações descritas em FLUKE-BIOMEDICAL (2005). A área imageada é uma região de 19,96 mm $\times$ 19,96 mm que é reconstruída com diversas resoluções espaciais.

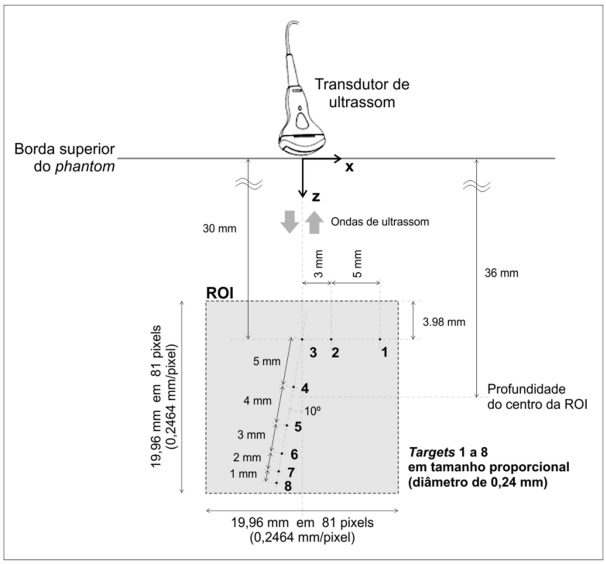
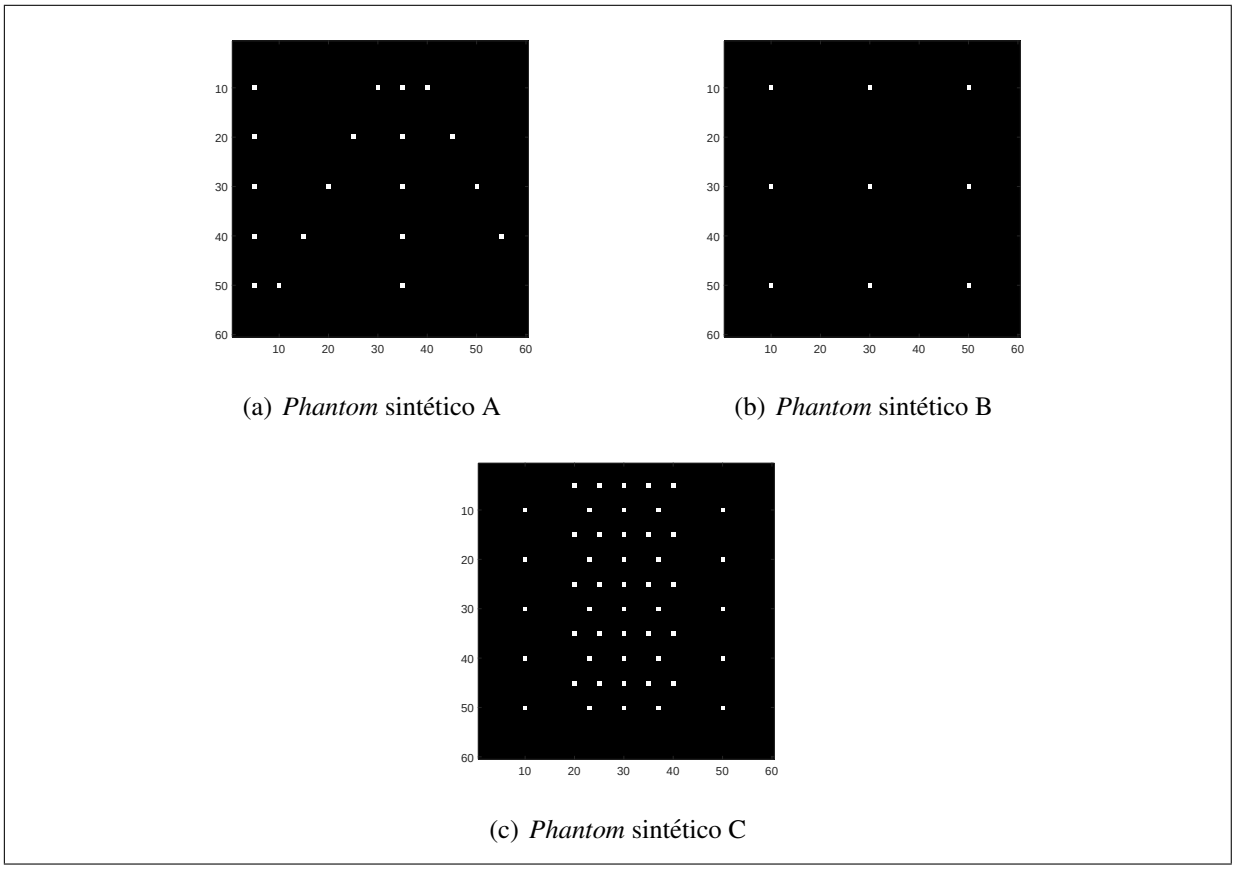


Figura 26: Vista esquemática dos alvos no interior do *phantom*. As dimensões, tamanhos e posições estão de acordo com as descritas em FLUKE-BIOMEDICAL (2005).
Fonte: (VALENTE, 2017).

Além desse *phantom*, diversas imagens bidimensionais de referência foram criadas, chamadas de “sintéticas”, com o objetivo de fornecer dados de controle para as reconstruções

empregando os métodos propostos neste trabalho, e possibilitando a validação numérica e de fidelidade dos resultados obtidos. A Figura 27 apresenta os diversos *phantoms* utilizados.



(a) *Phantom* sintético A

(b) *Phantom* sintético B

(c) *Phantom* sintético C

Figura 27: *Phantoms* sintéticos usados como imagens de controle.
Fonte: Própria.

Cada *phantom* corresponde a uma área de 19,8 mm $\times$ 19,8 mm e se empregada uma resolução espacial de 0,3333 mm/pixel $\times$ 0,3333 mm/pixel será visualizado como uma imagem de 60 pixels $\times$ 60 pixels. Na Figura 28 é possível ver uma ilustração esquemática do *phantom* C, principal imagem de controle utilizada. As linhas cruzando o *phantom* representam as distâncias de cada ponto até a lateral do mesmo.

A constante c de entrada para o método FISTA foi definida como $c \geq \|\mathbf{H}^T\mathbf{H}\|_2$, que de acordo com Beck e Teboulle (2009a) pode ser considerada uma escolha típica. O valor de λ foi definido como 10% do maior valor absoluto de $\mathbf{H}^T\mathbf{g}$.

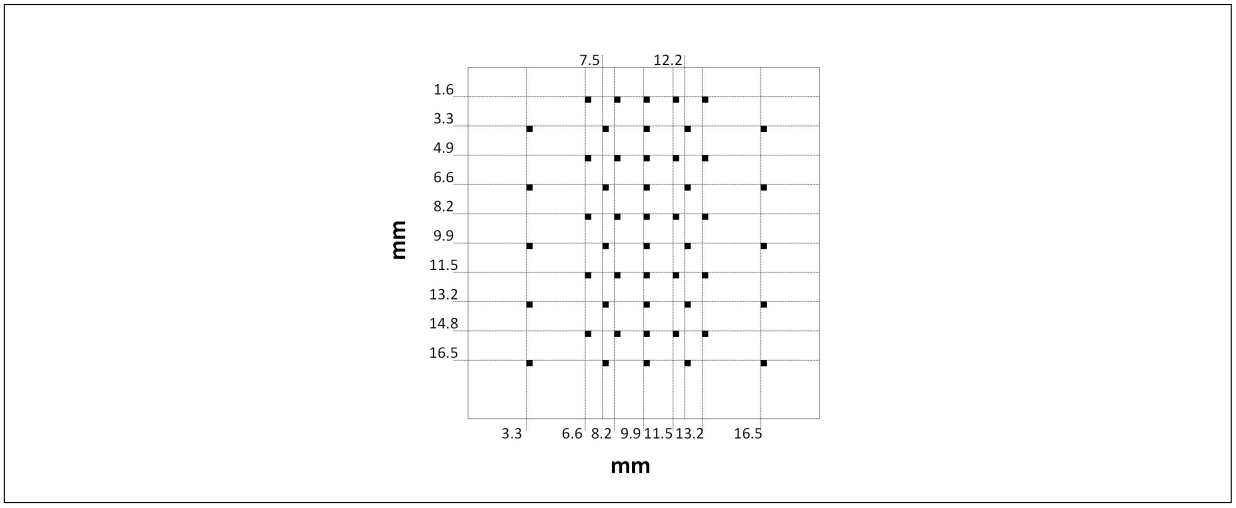


Figura 28: Vista esquemática do *phantom* C. Cada ponto tem uma área de 0,3333 mm $\times$ 0,3333 mm e corresponde a um pixel em uma imagem com 60 pixels $\times$ 60 pixels.
Fonte: Própria.

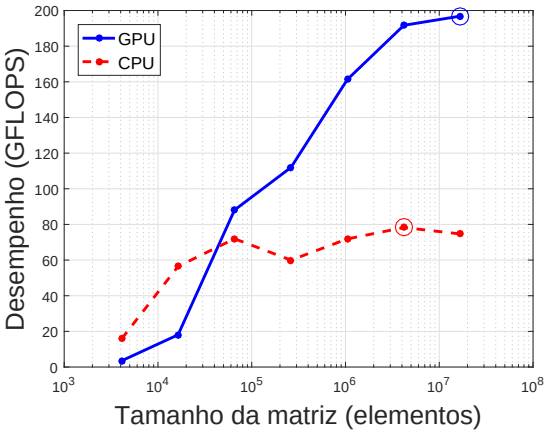
6.2 EXPERIMENTO 1: AVALIAÇÃO DE DESEMPENHO EM OPERAÇÕES BÁSICAS

Este primeiro experimento visa medir e avaliar o desempenho das GPUs utilizadas neste trabalho em relação as operações básicas (adição e multiplicação) com matrizes para dados em precisão simples e dupla com armazenamento denso (tradicional não esparso). Foram executadas operações com diversos tamanhos de matrizes, com valores iniciais definidos aleatoriamente e tempos coletados através de funcionalidades própria do MATLAB. Foram medidas a largura de banda para transferência de dados, velocidade de leitura seguida de gravação e a velocidade das operações de multiplicação matriz-matriz.

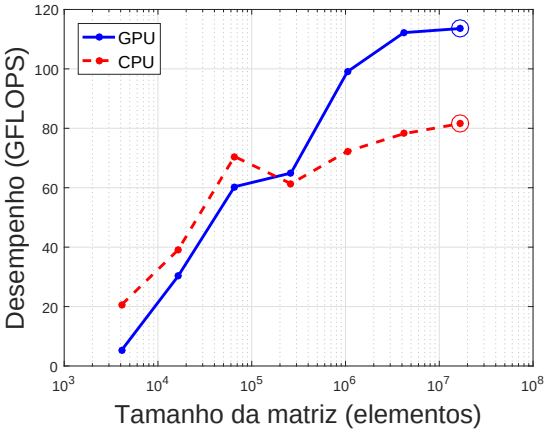
Os resultados para o teste de multiplicação matriz-matriz com dados em precisão dupla são apresentados na Figura 29.

A Figura 30 apresenta os resultados para o teste de multiplicação matriz-matriz com dados em precisão simples e a Figura 31 apresenta o comparativo para a multiplicação matriz-vetor com dados em precisão simples e dupla executados na GTX970.

Os resultados observados nas Figuras 29 e 30 corroboram a Lei de Gustafson-Barsis, que afirma que, em processamento paralelo, o desempenho melhora com o aumento da quantidade de dados a serem processados (KARBOWSKI, 2008). Os demais resultados encontram-se no Apêndice A.



(a) GTX780



(b) GTX970

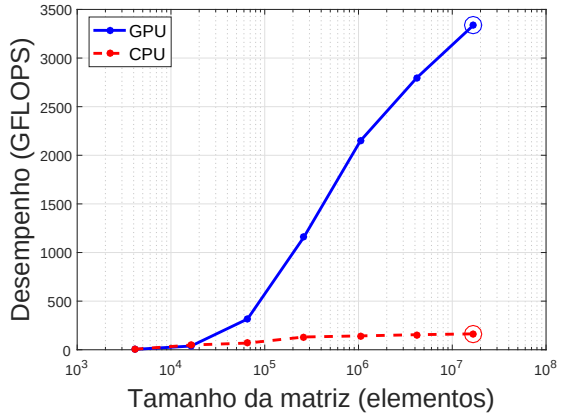
Figura 29: Desempenho na multiplicação matriz-matriz em precisão dupla.
Fonte: Própria.

Estes resultados serão usados como parâmetro de comparação e balizamento para as avaliações de desempenho das rotinas customizadas propostas neste trabalho.

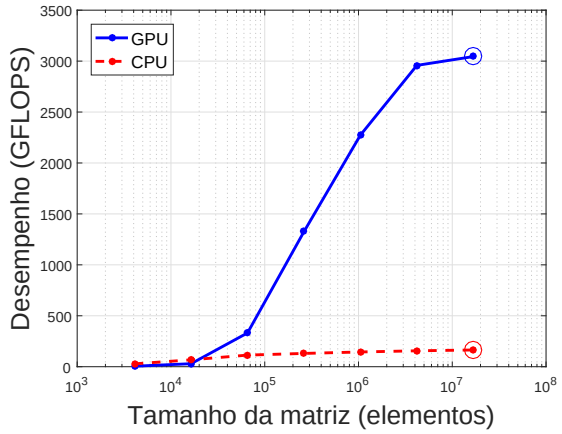
6.3 EXPERIMENTO 2: AVALIAÇÃO DE DESEMPENHO COM MATRIZES ESPARSAS

Neste experimento procurou-se avaliar e comparar os resultados para a execução das operações de multiplicação matriz-vetor utilizando armazenamento esparso e denso em precisão dupla. Como matriz de referência foi utilizado o tamanho de 5.000×5.000 elementos e geradas versões com esparsidade variando entre 10% e 90%. Os resultados são apresentados na Figura 32.

Mesmo representando uma opção para a redução do consumo de memória, o



(a) GTX780



(b) GTX970

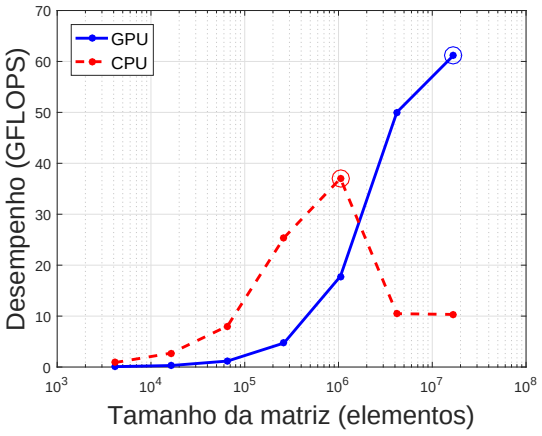
Figura 30: Desempenho na multiplicação matriz-matriz em precisão simples.
Fonte: Própria.

armazenamento esparso apresenta diferenças significativas de desempenho na execução das operações o que pode comprometer os tempos finais de reconstrução, inviabilizando o imageamento em tempo real.

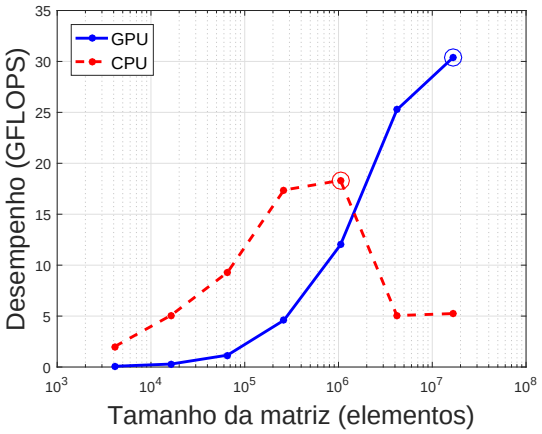
Estes resultados reforçam a necessidade de alternativas para contornar o problema da quantidade de dados necessários para a reconstrução de imagens de ultrassom por problemas inversos. Detalhes adicionais se encontram no Apêndice B.

6.4 EXPERIMENTO 3: AVALIAÇÃO DO MODELO SIMÉTRICO

Foram realizados testes com dados sintéticos e reais para validar o modelo e medir o desempenho do processamento em GPU. A coleta dos tempos foi realizada através de



(a) Precisão Simples



(b) Precisão Dupla

Figura 31: Desempenho na multiplicação matriz-vetor em precisão simples e dupla na GTX970.
Fonte: Própria.

funções próprias do MATLAB e também através do software NVIDIA *Visual Profile* (NVIDIA Corporation, 2016). Para cada teste foram executadas 5 rodadas de processamento, cada rodada compreendendo 100 repetições do algoritmo sendo avaliado.

Como referência para avaliação do erro nas reconstruções foi empregado o erro quadrado médio normalizado (NRMSE - *Normalized Root Mean Square Error*). A imagem de referência foi reconstruída utilizando a matriz tradicional (sem simetria) em CPU. O NRMSE foi calculado conforme a equação:

$$NRMSE = \left(\frac{\|\mathbf{f}_p - \mathbf{f}_r\|_2}{\|\mathbf{f}_r\|_2} \right) \tag{48}$$

onde $\mathbf{f}_p$ representa o resultado da reconstrução empregando o método proposto neste trabalho

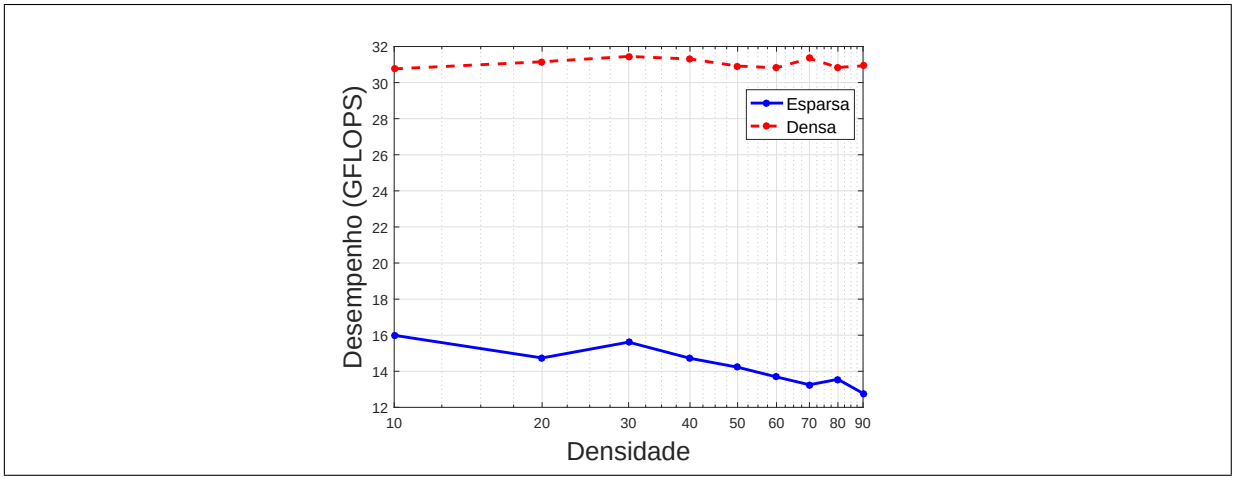


Figura 32: Desempenho na multiplicação matriz-vetor com armazenamento esparso em precisão dupla (GTX970).
Fonte: Própria.

e $\mathbf{f}_r$ é a imagem reconstruída em CPU utilizando a matriz tradicional. São sempre comparadas as imagens obtidas em CPU vs.GPU na mesma iteração ($CPU_1 \leftrightarrow GPU_1, CPU_2 \leftrightarrow GPU_2, \ldots, CPU_n \leftrightarrow GPU_n$).

6.4.1 TESTES COM DADOS SINTÉTICOS

Neste conjunto de testes foram realizadas reconstruções utilizando os *phantoms* sintéticos apresentados anteriormente. Os testes foram realizados utilizando variáveis de precisão simples. Os erros de reconstrução são apresentados na Figura 33 com o número de iterações variando entre 1 e 20.

Nas Figuras 34 e 35 são apresentadas as imagens reconstruídas para os *phantoms* sintéticos. Nestas reconstruções se procurou identificar artefatos ou anomalias nos resultados obtidos através do modelo simétrico. Através de inspeção visual não foi possível detectar qualquer tipo de artefato nas imagens obtidas. Desta forma, estes resultados contribuem para a validação do modelo proposto, já que ele fornece imagens com qualidade e fidelidade aceitáveis.

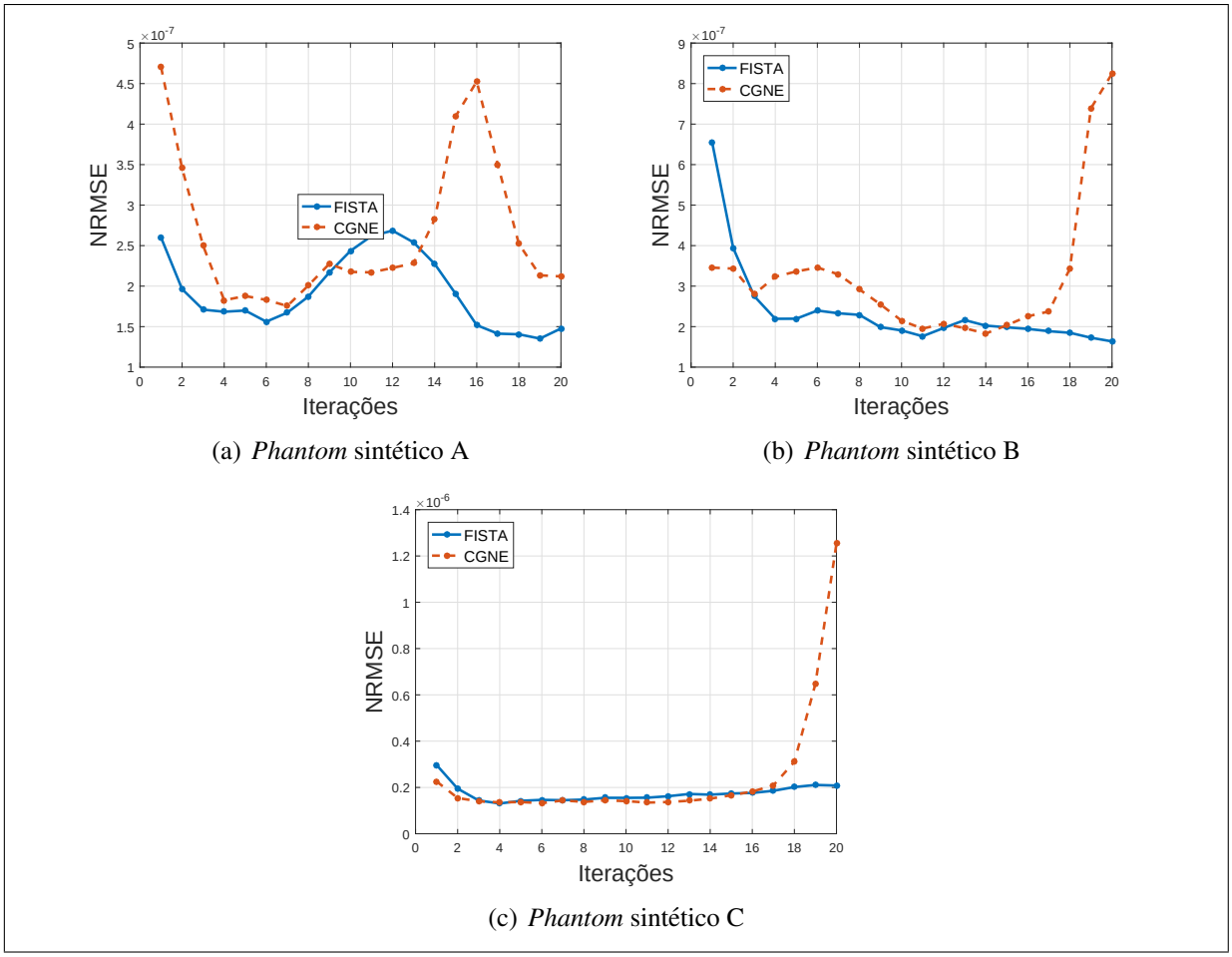


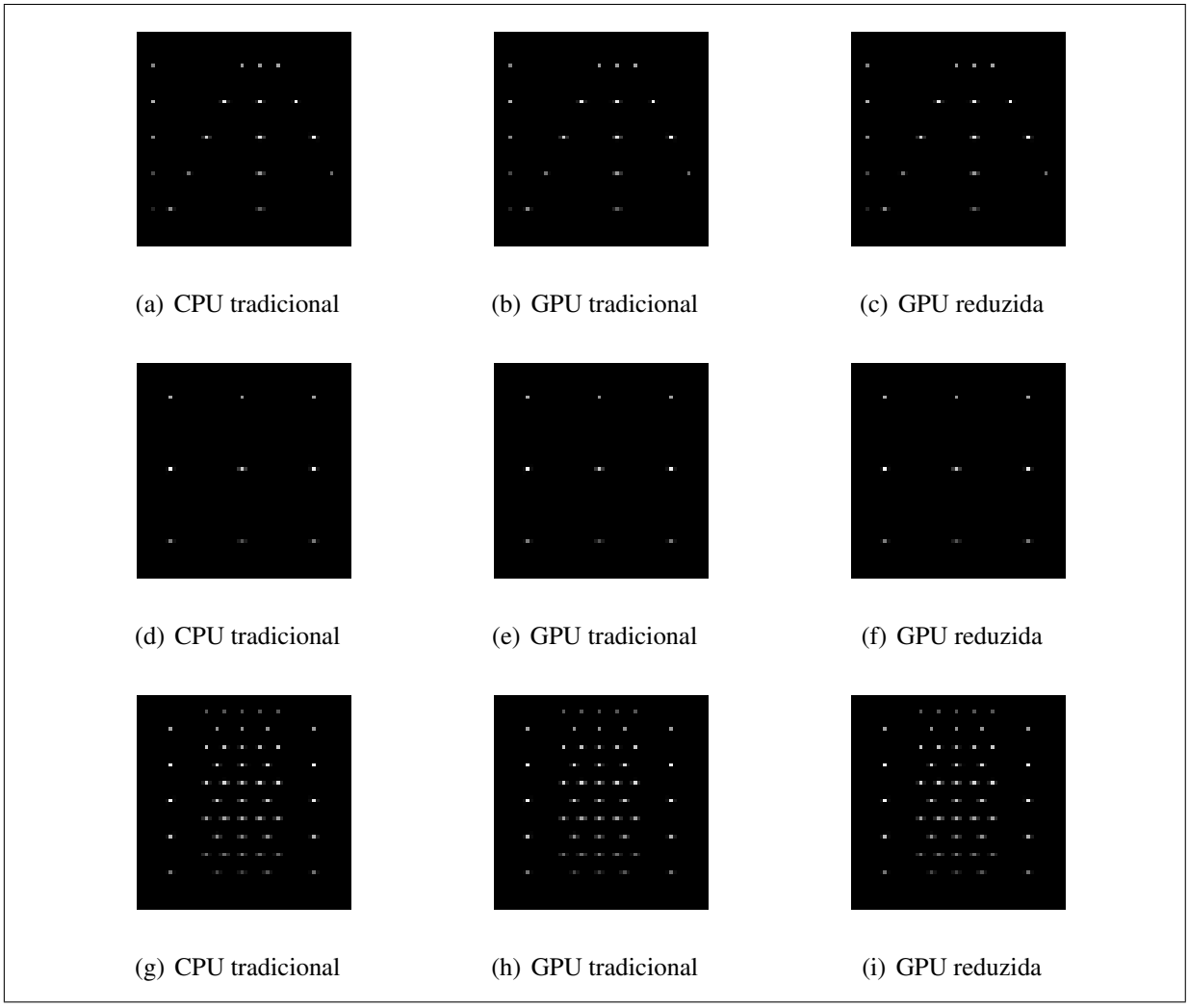
Figura 33: Erros de reconstrução para os *Phantoms* sintéticos (GTX 970).
Fonte: Própria.

6.4.2 TESTES COM DADOS REAIS

Este conjunto de testes visa avaliar o desempenho e o comportamento do modelo simétrico quando aplicado a um conjunto de dados reais. Diferentemente dos dados sintéticos anteriores, nesta situação diversos fatores podem interferir nos resultados, dentre outros, os erros de amostragem e ruídos inerentes ao processo de amostragem com transdutores reais.

É preciso considerar que apesar de o *phantom* empregado ser um dispositivo de qualidade e estar dentro dos padrões para este tipo de aplicação, ele pode apresentar variações dentro de uma determinada faixa. Podemos citar, por exemplo, a velocidade real do som que pode variar em até ± 6 m/s internamente ao *phantom* (FLUKE-BIOMEDICAL, 2005).

É necessário considerar, também, que as condições de coleta de dados dependem da habilidade e precisão do operador. Pelo fato de ser empregado um gel acoplador a distância,



(a) CPU tradicional (b) GPU tradicional (c) GPU reduzida

(d) CPU tradicional (e) GPU tradicional (f) GPU reduzida

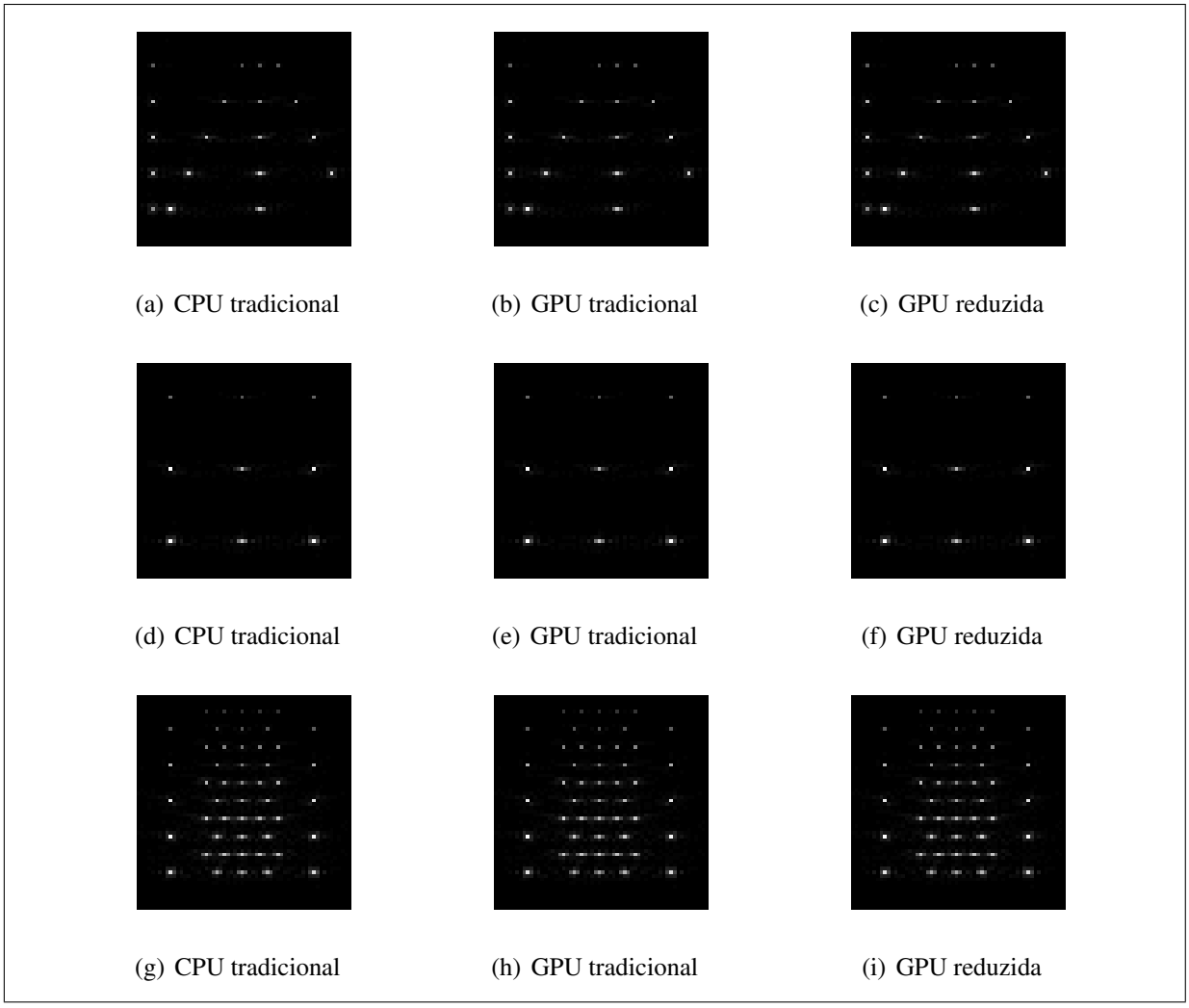
(g) CPU tradicional (h) GPU tradicional (i) GPU reduzida

Figura 34: Imagens reconstruídas para os *Phantoms* sintéticos utilizando FISTA. Resultado da reconstrução com parada em 5 iterações.
Fonte: Própria.

inclinação e translação do transdutor podem variar em relação aos alvos definidos na Figura 26.

Foram preparados quatro conjuntos de dados de testes descritos na Tabela 5; nestes conjuntos se variou a dimensão da imagem tanto no número de *pixels* quanto na área a ser escaneada. É possível observar o crescimento do volume de dados a medida que a área aumenta de tamanho. Para imagens de tamanhos práticos para aplicações reais, como a do último conjunto, a questão da quantidade de dados se torna um fator a ser considerado.

O crescimento da matriz **H** está diretamente ligado a três fatores:(i) número de *pixels* da imagem; (ii) número de elementos do transdutor empregados nas capturas e (iii) tamanho da ROI. Para os conjuntos de dados empregados neste experimento esse valores são apresentados



(a) CPU tradicional (b) GPU tradicional (c) GPU reduzida

(d) CPU tradicional (e) GPU tradicional (f) GPU reduzida

(g) CPU tradicional (h) GPU tradicional (i) GPU reduzida

Figura 35: Imagens reconstruídas para os *Phantoms* sintéticos utilizando CGNE. Resultado da reconstrução com parada em 5 iterações.
Fonte: Própria.

na Tabela 6. O tamanho da ROI influencia o número de amostras a serem coletadas, quanto maior a ROI e mais distante do transdutor maior o tempo de amostragem, como vista na Tabela 6.

Na Tabela 7 temos os tempos medidos para a execução das operações de multiplicação matriz-vetor, tanto direta quanto transposta. Através destes tempos é possível calcular o ganho de desempenho alcançado com o modelo simétrico empregando a matriz reduzida. No caso das matrizes reduzidas ainda não foi possível obter valores de *speed ups* muito próximos dos valores alcançados pela rotinas convencionais que utilizam matrizes tradicionais.

A Tabela 8 apresenta os resultados dos tempos de execução para o algoritmo CGNE

Tabela 5: Conjunto de dados reais utilizados.

Conjunto	Tamanhos		Memória		Dimensões da matriz (linhas × colunas)	Crescimento em relação ao A
	(pixels × pixels)	(mm × mm)	Tradicional (GB)	Reduzida (GB)		
A	140 × 140	10 × 10	2,18	1,09	20.160 × 19.600	1,0x
B	140 × 140	15 × 15	3,12	1,56	39.552 × 19.600	1,4x
C	140 × 140	20 × 20	4,05	2,03	51.328 × 19.600	1,9x
D	180 × 180	20 × 20	**6,66**	**3,33**	51.456 × 32.400	**3,1x**

Fonte: Própria.

Tabela 6: Características dos conjuntos de dados.

Conjunto	Número de *pixels*	Número de elementos	Tamanho da ROI (mm × mm)	Número de amostras
A	19.600	64	10 × 10	440
B	19.600	64	15 × 15	618
C	19.600	64	20 × 20	802
D	32.400	64	20 × 20	**804**

Fonte: Própria.

por iteração. Neste caso foi possível se atingir valores de *speed up* de até 3,8x em relação às operações com CPU. Observando os tempos, em segundos, é possível notar que o crescimento no caso da CPU é maior do que no caso da GPU; quanto maior o conjunto de dados melhor, a relação de tempos para a GPU. Comparando-se o resultado do conjunto A com o conjunto D vemos que o aumento no caso da CPU foi de **3,0x** enquanto na GPU foi de **2,8x**. Os conjuntos C e D não puderam ser testados com a matriz tradicional por falta de espaço de memória na GPU, indicado por NMA na tabela.

Os resultados para uma reconstrução completa (com parada em 5 iterações) podem ser vistos na Tabela 9. Neste caso o ganho de desempenho favoreceu ainda mais a GPU, reforçando assim sua vocação para processamento massivo de dados em paralelo. Foi possível alcançar valores de *speed up* de até **5,7x**. Na comparação entre os diversos conjuntos de dados, vemos

Tabela 7: Tempos das operações, em segundos.

Conjunto	Tradicional				*Speed up* médio (CPU/GPU)	Reduzida		*Speed up* médio (CPU / GPU)
	Hf		$\mathbf{H}^T\mathbf{f}$			**Hf**	$\mathbf{H}^T\mathbf{f}$	
	CPU	GPU	CPU	GPU		GPU	GPU	
A	0,093	0,019	0,088	0,016	5,2x	0,029	0,024	3,4x
B	0,140	0,021	0,129	0,023	**6,1x**	0,034	0,033	4,0x
C	0,179	NMA	0,163	NMA	-	0,039	0,045	4,1x
D	0,334	NMA	0,323	NMA	-	0,065	0,090	**4,2x**

Fonte: Própria.

Tabela 8: Tempos medianos para o CGNE por iteração, em segundos.

Conjunto	Aumento de tamanho em relação ao A	CPU tradicional		GPU reduzida		*Speed up* médio (CPU/GPU)
		Tempo	Inc.	Tempo	Inc.	
A	1,0x	0,178	1,0x	0,056	1,0x	3,2x
B	1,4x	0,257	1,4x	0,070	1,2x	3,7x
C	1,9x	0,333	1,9x	0,087	1,5x	**3,8x**
D	3,1x	0,540	**3,0x**	0,159	**2,8x**	3,4x

Fonte: Própria.

que a relação de aumento de tempos de processamento é ainda mais favorável à GPU com um incremento de **3,0x** para a CPU e apenas **2,5x** para a GPU, entre os conjuntos A e D.

Tabela 9: Tempos medianos para o CGNE com 5 iterações, em segundos.

Conjunto	Aumento de tamanho em relação ao A	CPU tradicional		GPU reduzida		*Speed up* médio (CPU/GPU)
		Tempo	Inc.	Tempo	Inc.	
A	1,0x	0,898	1,0x	0,208	1,0x	4,3x
B	1,4x	1,293	1,4x	0,249	1,2x	5,2x
C	1,9x	1,704	1,9x	0,299	1,4x	**5,7x**
D	3,1x	2,706	**3,0x**	0,512	**2,5x**	5,3x

Fonte: Própria.

Nas Tabelas 10 e 11 são apresentados os resultados equivalentes para o algoritmo FISTA. Neste caso os resultados são ainda mais animadores, pois no caso do tempo por iteração o ganho máximo foi de até **4,3x** contra **3,8x** do CGNE. Observando o crescimento do tempo entre os conjuntos A e D observa-se que neste caso a relação piorou na CPU com **3,6x** e se manteve a mesma na GPU com **2,8x**. Para o caso da reconstrução completa o *speed up* máximo obtido foi de **7,4x** e relação entre os conjuntos A e D para o crescimento do tempo de processamento foi de **3,7x** para a CPU e **2,8x** para a GPU.

Tabela 10: Tempos medianos para o FISTA por iteração, em segundos.

Conjunto	Aumento de tamanho em relação ao A	CPU tradicional		GPU reduzida		*Speed up* médio (CPU/GPU)
		Tempo	Inc.	Tempo	Inc.	
A	1,0x	0,189	1,0x	0,057	1,0x	3,3x
B	1,4x	0,268	1,4x	0,071	1,2x	3,8x
C	1,9x	0,350	1,9x	0,087	1,5x	4,0x
D	3,1x	0,688	**3,6x**	0,158	**2,8x**	**4,3x**

Fonte: Própria.

Tabela 11: Tempos medianos para o FISTA com 5 iterações, em segundos.

Conjunto	Aumento de tamanho em relação ao A	CPU tradicional		GPU reduzida		*Speed up* médio (CPU/GPU)
		Tempo	Inc.	Tempo	Inc.	
A	1,0x	0,955	1,0x	0,169	1,0x	5,7x
B	1,4x	1,366	1,4x	0,209	1,2x	6,5x
C	1,9x	1,775	1,9x	0,257	1,5x	6,9x
D	3,1x	3,493	**3,7x**	0,472	**2,8x**	**7,4x**

Fonte: Própria.

Os resultados quanto ao erro de reconstrução em relação à imagem reconstruída em CPU utilizando matriz densa e tradicional são apresentados na Figura 36. Observando os gráficos é possível verificar que o algoritmo FISTA teve um melhor comportamento neste ponto, mantendo-se mais estável enquanto o número de iterações aumentava. O erro está dentro dos padrões esperados e está de acordo com o padrão definido pelo *Institute of Electrical and Electronics Engineers* (IEEE), o padrão IEEE-754-2008, e suas implementações em plataformas paralelas (STANDARDS; SOCIETY, 2008). Mais detalhes sobre a precisão numérica podem ser vistos no trabalho de Revol e Theveny (2014), que realizaram um estudo detalhado dos efeitos do paralelismo massivo sobre a acurácia e precisão das operações aritméticas.

As imagens resultantes para os algoritmos avaliados são apresentadas na Figura 37. Ambos os algoritmos foram finalizados na quinta iteração e os resultados são apresentados em escala de cinza com valores absolutos. Para fins de comparação é apresentado também o resultado da reconstrução empregando o método *beamforming*, que serve como referência para comparação e busca por artefatos ou anomalias.

Por inspeção visual não foi possível detectar anomalias nas imagens que pudessem estar relacionadas com o modelo simétrico. É possível observar algumas diferenças de resultados entre os métodos, o que é aceitável, uma vez que são métodos com propostas distintas de regularização.

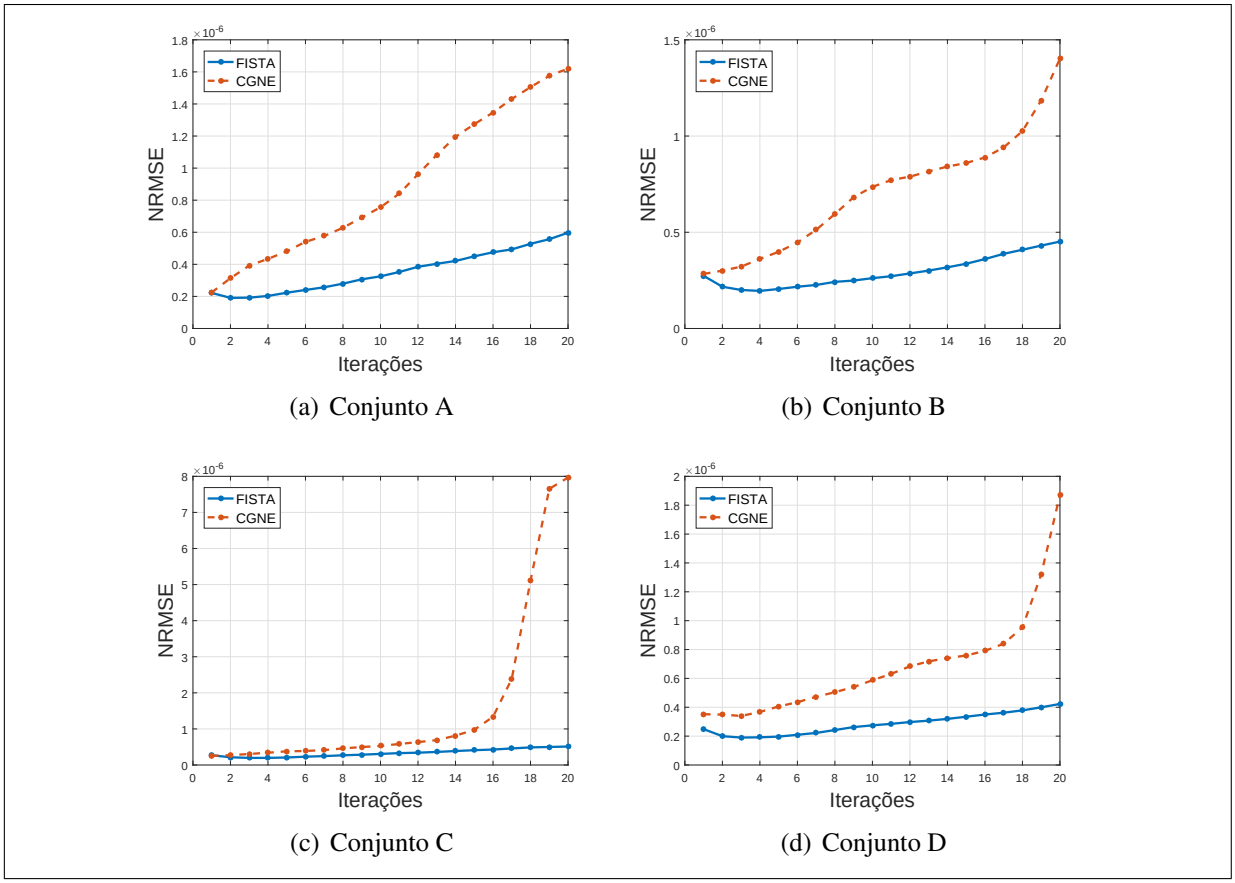
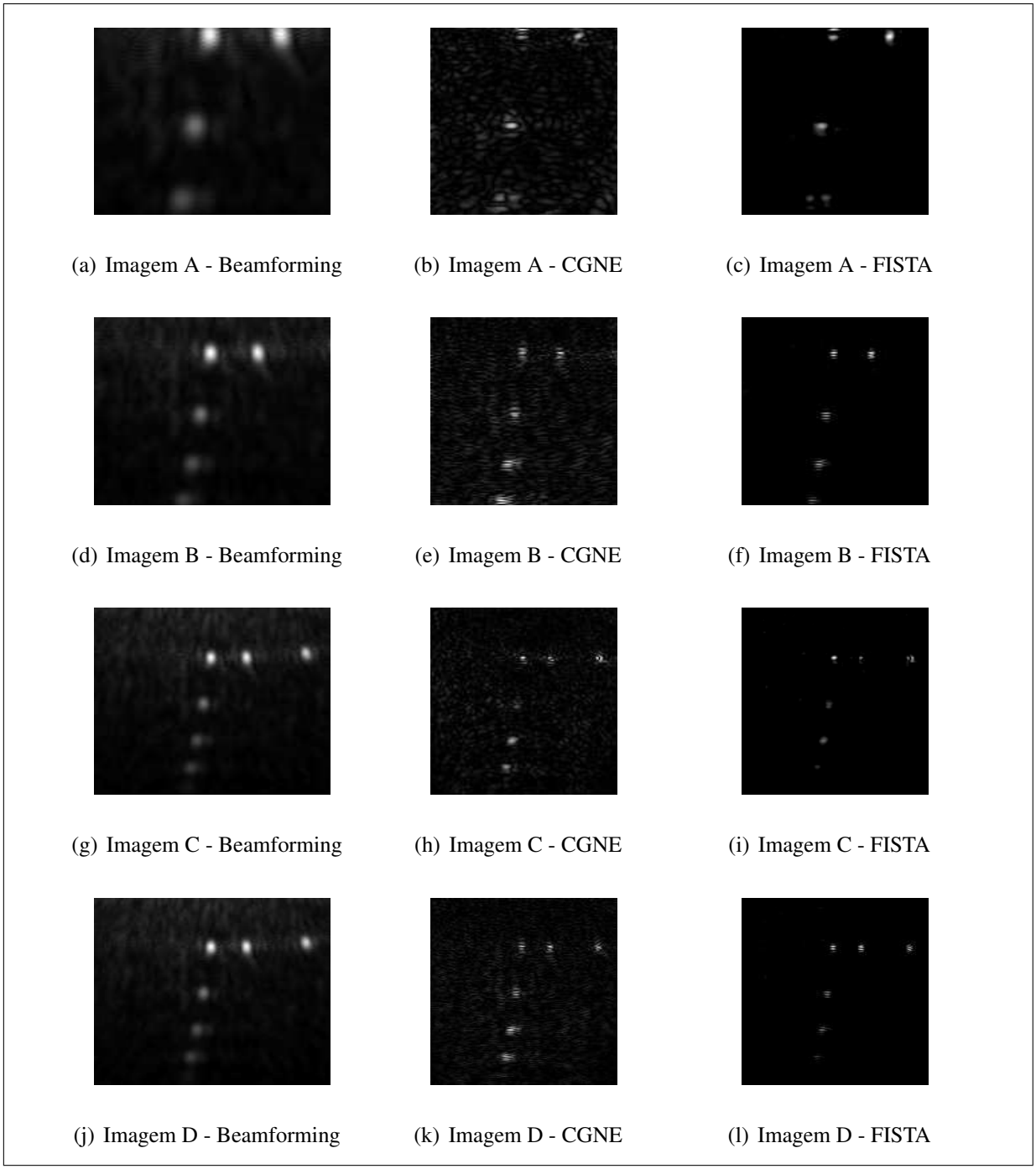


Figura 36: Erros de reconstrução para os conjuntos de dados reais. NRMSE calculado com relação à imagem de referência reconstruída em CPU com precisão simples e armazenamento tradicional.
Fonte: Própria.

6.5 CONCLUSÃO DO CAPÍTULO

O ambiente experimental se mostrou satisfatório para a realização dos experimentos necessários para a verificação do modelo simétrico bem como da medição dos tempos de reconstrução. Uma vez estabilizadas as rotinas de reconstrução e as de multiplicação matriz-matriz e matriz-vetor simétricas foi possível a execução de todo o conjunto de experimentos e testes.

A realização dos testes do experimento 1 permitiu a avaliação da capacidade computacional das GPUs empregadas neste trabalho e a geração de dados para uma análise comparativa futura. Estes resultados embasaram a decisão de utilizar dados em precisão simples na execução dos testes com os *phantoms* sintéticos, pois estes proporcionam um desempenho computacional bastante superior além de reduzir o volume de dados necessários.



(a) Imagem A - Beamforming (b) Imagem A - CGNE (c) Imagem A - FISTA

(d) Imagem B - Beamforming (e) Imagem B - CGNE (f) Imagem B - FISTA

(g) Imagem C - Beamforming (h) Imagem C - CGNE (i) Imagem C - FISTA

(j) Imagem D - Beamforming (k) Imagem D - CGNE (l) Imagem D - FISTA

Figura 37: Resultados das reconstruções com dados reais empregando o modelo simétrico. As imagens são apresentadas em escala de intensidade com valores absolutos e processos finalizados em 5 iterações.
Fonte: Própria.

Os testes com *phantoms* sintéticos mostraram que as rotinas customizadas de multiplicação matriz-matriz e matriz-vetor funcionaram adequadamente, uma vez que as imagens não apresentavam anomalias ou artefatos. De acordo com o que havíamos suposto, o modelo simétrico de reconstrução de imagens de ultrassom apresentou bons resultados práticos,

verificados no experimento 3. Foi possível avaliar a qualidade final das imagens, bem como, a existência ou não de artefatos ou anomalias nos resultados com dados reais. É importante ressaltar que a qualidade e a fidelidade das imagens resultantes depende diretamente da boa escolha do parâmetros de reconstrução.

Com a utilização das rotinas customizadas de multiplicação matriz-matriz e matriz-vetor foi possível alcançar ganhos de desempenho bastante significativos. A redução do volume total de dados manipulados também foi considerável, uma vez que neste modelo necessita-se apenas da metade dos dados armazenados na matriz **H**, que é a maior estrutura de dados utilizada.

7 CONSIDERAÇÕES FINAIS E CONCLUSÕES

A principal contribuição deste trabalho é um modelo de reconstrução de imagens de ultrassom que reduz em 50% a necessidade de memória e executa a reconstrução paralelamente em GPU.

Nos Capítulos 2, 3 e 4 foram vistos os conceitos fundamentais e os métodos utilizados. O Capítulo 5 apresentou o modelo simétrico proposto neste trabalho e o Capítulo 6 descreveu o ambiente experimental utilizados e os experimentos realizados.

Alguns trabalhos similares foram avaliados e comparados, entre outros citamos:

- Szasz et al. (2016) propõe uma nova implementação do *beamforming* com ganhos sobre a resolução espacial para GPUs. Neste estudo, eles empregam diversas excitações para formar um frente de ondas e posteriormente coletar os dados dos ecos. No nosso caso, empregamos apenas uma única excitação;

- Lin et al. (2015), propõe dividir a matriz e processá-la parte a parte. A nossa proposta apresentou melhores resultados em relação aos tempos de processamento.

- Birk et al. (2014) e Goncharsky et al. (2014) discutem uma implementação para GPU de um método para reconstrução de imagens de ultrassom 3D. Apesar dos ganhos obtidos, ainda apresentam tempos de reconstrução elevados mesmo para imagens de baixa resolução, tempos próximos de meio segundo. Os resultados obtidos pelo nosso trabalho obteve tempos próximos de 0.1 segundos para imagens em alta resolução. Os resultados obtidos por Birk et al. (2014) e Goncharsky et al. (2014) ainda não permitem uma reconstrução próximo de tempo real;

- Choe et al. (2013) implementaram um versão do *beamforming* que utiliza um transdutor em forma de anel e reconstrução em tempo real. Neste caso, a diferença principal com a nossa proposta é a melhor qualidade de imagem que podemos obter com a reconstrução baseada em problemas inversos.

Os resultados iniciais de desempenho mostram a vantagem das GPUs nas operações básicas com ganhos de até **6,1**$\times$ em relação a CPU, quando empregada a precisão simples e a matriz tradicional, e de até **4,2**$\times$ para o caso da matriz reduzida. Estes resultados demonstram que o ambiente experimental se comportou adequadamente e, assim, é possível considerar o primeiro objetivo específico atingido.

Com o uso do modelo simétrico foi possível aumentar consideravelmente o tamanho das imagens construídas, tanto em *pixels* como em relação ao tamanho da ROI. As reconstruções que utilizaram a GTX970 (4 GB de memória) estavam limitadas a imagens de tamanho **140 *pixels*** $\times$ **140 *pixels* para uma ROI de 15 mm** $\times$ **15 mm**. Quando se aplicou o modelo simétrico foi possível reconstruir imagens de até **180 *pixels*** $\times$ **180 *pixels* e ROI de 20 mm** $\times$ **20 mm**. Estas observações nos permitem concluir que o segundo objetivo específico também foi alcançado.

Os resultados obtidos em termos de desempenho computacional são considerados bons. As rotinas paralelas de reconstrução permitiram ganhos de *speed up* de até **6**$\times$ para o modelo tradicional e de até $\mathbf{4,2}\times$ quanto empregado o modelo simétrico. Estes resultados se referem às operações básicas de multiplicação matriz-vetor. Quando observados os resultados das rotinas simétricas na reconstrução completa de uma imagem, os ganhos de *speed up* foram de até $\mathbf{4,3}\times$ para o algoritmo CGNE e de até $\mathbf{7,4}\times$ para o FISTA. Neste caso, verificou-se uma clara vantagem do algoritmo FISTA. Vale ressaltar que estes resultados foram obtidos utilizando-se a GPU GTX970, um modelo relativamente modesto se comparado com os últimos modelos disponíveis [1]. Desta forma, é possível concluir que o terceiro objetivo específico foi igualmente alcançado.

[1] Alguns especificações para GPUs disponíveis podem ser encontrados em: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-970/specifications e https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications.

7.1 TRABALHOS FUTUROS

Como trabalhos futuros, sugerimos a avaliação de outros algoritmos paralelos para a multiplicação matriz-matriz e matriz-vetor que levem em consideração a forma da matriz. Nos experimentos realizados, quando aplicado o modelo simétrico, as matrizes tenderam a ficar altas e magras (muito mais linhas do que colunas pela redução do número de colunas à metade). Assim, algoritmos paralelos que explorem melhor essa característica poderiam apresentar ganhos de desempenho melhores.

Sugerimos, também, a avaliação de outros algoritmos de reconstrução por problemas inversos, como por exemplo, o algoritmo proposto por Valente (2017), o ADMM acelerado, que apresenta bons resultados com um número reduzido de iterações, o que permitiria reduzir o tempo total necessário para a obtenção da imagem.

Quanto ao modelamento do problema, uma possível linha de trabalho seria o de se substituir o uso de matrizes fixas pré-calculadas por funções matemáticas implementas através de algoritmos paralelos. Atualmente a matriz é construída em separado e antecipadamente, desta forma, qualquer alteração nos parâmetros do modelo exigem a geração de uma nova matriz. Com a implementação de uma função geradora da PSF os dados poderiam ser calculados durante a reconstrução de acordo com os parâmetros utilizados no momento. Isto poderia agregar uma maior flexibilidade ao sistema como um todo.

REFERÊNCIAS

ABDELFATTAH, A.; KEYES, D.; LTAIEF, H. KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators. **Proceedings of the 14th International Conference on Artificial Intelligence and Statistics**, v. 15, n. 212, p. 434–442, 2014. ISSN 15324435.

AZHARI, H. **Basics of Biomedical Ultrasound for Engineers**. [S.l.]: Wiley, 2010. 377 p. ISBN 9780470465479.

BARRETT, H. H.; MYERS, K. J.; RATHEE, S. **Foundations of Image Science**. [S.l.: s.n.], 2004. 953 p. ISSN 00942405. ISBN 978-0-471-15300-9.

BECK, A.; TEBOULLE, M. A fast Iterative Shrinkage-Thresholding Algorithm with application to wavelet-based image deblurring. In: **2009 IEEE International Conference on Acoustics, Speech and Signal Processing**. [S.l.]: IEEE, 2009b. p. 693–696. ISBN 978-1-4244-2353-8.

BECK, R.; TEBOULLE, M. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. **SIAM Journal on Imaging Sciences**, v. 2, n. 1, p. 183–202, 2009a. ISSN 1936-4954.

BERTERO, M.; BOCCACCI, P. **Introduction to Inverse Problems in Imaging**. London: Institute of Physics Publishing, 2008. 350 p. ISBN 0750304359.

BIRK, M. et al. A comprehensive comparison of GPU- and FPGA-based acceleration of reflection image reconstruction for 3D ultrasound computer tomography. **Journal of Real-Time Image Processing**, v. 9, n. 1, p. 159–170, 2014. ISSN 18618200.

BOVIK, A. C. **Handbook of Image and Video Processing**. 1. ed. San Diego: Academic Press, 2000. ISBN 0-12-119790-5.

BUSHBERG, J. T. et al. **The essential physics of medical imaging**. Second. [S.l.]: Lippincott Williams & Wilkins, 2002. 935 p. ISSN 1098-6596. ISBN 9788578110796.

CHOE, J. W. et al. GPU-based real-time imaging software suite for medical ultrasound. **IEEE International Ultrasonics Symposium, IUS**, v. 768, p. 2057–2060, 2013. ISSN 19485719.

CHRISTENSEN, D. **Ultrasound Bioinstrumentation**. New York: John Wiley & Sons, Inc., 1998. 235 p.

DAI, Y. et al. Real-time visualized freehand 3D ultrasound reconstruction based on GPU. **IEEE Transactions on Information Technology in Biomedicine**, v. 14, n. 6, p. 1338–1345, 2010. ISSN 10897771.

DEMIRLI, R.; SANIIE, J. Model-based estimation of ultrasonic echoes part II: Nondestructive evaluation applications. **IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control**, v. 48, n. 3, p. 803–811, 2001. ISSN 08853010.

EIDHEIM, O. C.; SKJERMO, J.; AURDAL, L. Real-time analysis of ultrasound images using GPU. **International Congress Series**, v. 1281, p. 284–289, 2005. ISSN 05315131.

EIJKHOUT, V.; CHOW, E.; GEIJN, R. van de. **Introduction to High Performance Scientific Computing**. 2nd editio. ed. [S.l.: s.n.], 2014. 446 p. ISBN 978-1-257-99254-6.

ELAD, M. **Sparse and redundant representations: From theory to applications in signal and image processing**. New York, NY: Springer, 2010. 1–376 p. ISSN 144197010X. ISBN 9781441970107.

ELLIS, M. A.; VIOLA, F.; WALKER, W. F. Super-resolution image reconstruction using diffuse source models. **Ultrasound in Medicine and Biology**, v. 36, n. 6, p. 967–977, 2010. ISSN 03015629.

FLUKE-BIOMEDICAL. **Nuclear Associates: 84-317 and 84-317-700 Multipurpose Tissue/Cyst Ultrasound Phantoms - Users Manual**. [S.l.], 2005.

FONTES, P. X. de et al. Real time ultrasound image denoising. **Journal of Real-Time Image Processing**, v. 6, n. 1, p. 15–22, 2011. ISSN 18618200.

GARLAND, M. et al. Parallel computing experiences with CUDA. **IEEE Micro**, v. 28, n. 4, p. 13–27, 2008. ISSN 02721732.

GONCHARSKY, A. V.; ROMANOV, S. Y.; SERYOZHNIKOV, S. Y. Inverse problems of 3D ultrasonic tomography with complete and incomplete range data. **Wave Motion**, Elsevier B.V., v. 51, n. 3, p. 389–404, 2014. ISSN 01652125. Disponível em: <http://dx.doi.org/10.1016/j.wavemoti.2013.10.001>.

GRAMA, A. et al. **Introduction to Parallel Computing**. [S.l.]: Addison-Wesley, 2003. 856 p. ISBN 0201648652.

GROOVER, M. P. **Automation, Production Systems and Computer Integrated Manufacturing**. [S.l.: s.n.], 1987. 867 p. ISBN 9780130895462.

GUARNERI, G. A. **IDENTIFICAÇÃO DE DESCONTINUIDADES EM PEÇAS METÁLICAS UTILIZANDO SINAIS ULTRASSÔNICOS E TÉCNICAS DE PROBLEMAS INVERSOS**. Tese (Doutorado) — Universidade Tecnológica Federal do Paraná, 2015.

GUARNERI, G. A. et al. A sparse reconstruction algorithm for ultrasonic images in nondestructive testing. **Sensors (Switzerland)**, v. 15, n. 4, p. 9324–9343, 2015. ISSN 14248220.

HANSEN, C. **Rank-Deficient and Discrete Ill-Posed Problems - Numerical Aspects of Linear Inversion**. [S.l.]: SIAM, 1998. 264 p. ISBN 0898714036.

HEDRICK, W. R.; HYKES, D. L.; STARCHMAN, D. E. **Ultrasound Physics and Instrumentation**. 3rd editio. ed. [S.l.]: Mosby, 1995. ISBN 0815142463.

JANG, D. K. B.; DO, H. P. S. Multi GPU Implementation of Iterative Tomographic Reconstruction Algorithms. p. 185–188, 2009.

JENSEN, J. A. A Model for the Propagation and Scattering of Ultrasound in Tissue. **Journal of Acoustical Society of America**, n. 6, p. 182–190, 1991.

JENSEN, J. A. Calculation of Pressure Fields from Arbitrarily Shaped, Apodized, and Excited Ultrasound Transducers. **IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control**, v. 39, n. 2, p. 262–267, 1992a.

JENSEN, J. A. Deconvolution of Ultrasound Images. **Ultrasound Imaging**, v. 14, p. 1–15, 1992b.

JENSEN, J. A. Ultrasound imaging and its modeling. In: **Imaging of Complex Media with Acoustic and Seismic Waves**. Springer, 2002. v. 84, n. August, p. 1–39. ISBN 978-3-540-41667-8. Disponível em: <http://www.springerlink.com/content/whq1x71p6xcf7ynx/>.

JENSEN, J. A. Simulation of advanced ultrasound systems using Field II. **2004 2nd IEEE International Symposium on Biomedical Imaging: Nano to Macro (IEEE Cat No. 04EX821)**, p. 636–639, 2004.

JENSEN, J. A. **Linear description of ultrasound imaging systems, Notes for the International Summer School on Advanced Ultrasound Imaging**. [S.l.]: Center for fast Ultrasound Imaging, Dept. of Elec. Eng., Technical University of Denmark, 2015. 81 p.

KARBOWSKI, A. Amdahl's and Gustafson-Barsis laws revisited. **Distributed, Parallel, and Cluster Computing (cs.DC)**, 2008.

KARL, W. C. Image Restoration and Reconstruction. In: BOVIK, A. C. (Ed.). **Handbook of Image and Video Processing**. [S.l.]: Academic Press, 2000. cap. 3.6, p. 141–160. ISBN 0-12-119790-5.

KIRK, D.; HWU, W.-M. W. **Programming Massively Parallel Processors: A Hands-on Approach**. 2d edition. ed. Elsevier, 2010. 280 p. ISSN 978-0123814722. ISBN 0123814723. Disponível em: <http://www.amazon.de/dp/0123814723>.

LAI, P. W. et al. Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs. **20th Annual International Conference on High Performance Computing, HiPC 2013**, p. 139–148, 2013.

LIN, J. et al. CS-based fast ultrasound imaging with improved FISTA algorithm. In: **Proc. of SPIE**. [S.l.: s.n.], 2015. v. 9622, p. 96220A. ISBN 9781628418033. ISSN 1996756X.

LINGVALL, F.; OLOFSSON, T. On time-domain model-based ultrasonic array imaging. **IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control**, v. 54, n. 8, p. 1623–1633, 2007. ISSN 08853010.

LU, J. Y.; ZOU, H.; GREENLEAF, J. F. Biomedical ultrasound beam forming. **Ultrasound in Medicine and Biology**, v. 20, n. 5, p. 403–428, 1994. ISSN 03015629.

LU, Y.; DEMIRLI, R.; SANIIE, J. NDE applications of compressed sensing, signal decomposition and echo estimation. **IEEE International Ultrasonics Symposium, IUS**, n. 2, p. 1928–1931, 2014. ISSN 19485727.

MATHWORKS. **MATLAB**. 2016. Disponível em: <https://www.mathworks.com/products/matlab.html>.

NATH, R.; TOMOV, S.; DONGARRA, J. An improved MAGMA GEMM for Fermi graphics processing units. **International Journal of High Performance Computing Applications**, v. 24, n. 4, p. 511–515, 2010. ISSN 10943420.

NATH, R.; TOMOV, S.; DONGARRA, J. Accelerating GPU kernels for dense linear algebra. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 6449 LNCS, p. 83–92, 2011. ISSN 03029743.

NAVAUX, P. O. A. Fundamentos das Arquiteturas para Processamento Paralelo e Distribuído. In: **Escola Regional de Alto Desempenho - ERAD**. [S.l.]: Sociedade Brasileira de Computação, 2011. p. 22–59. ISBN 2177-0085.

NVIDIA Corporation. **NVIDIA's Netx Generation CUDA Compute Architecture: Kepler GK110**. [S.l.]: NVIDIA Corporation, 2012. 1–24 p.

NVIDIA Corporation. **CUBLAS Library**. 2014a.

NVIDIA Corporation. **Cuda c programming guide**. [S.l.]: NVIDIA Corporation, 2014b. 227 p.

NVIDIA Corporation. **GTX 970 Specifications**. 2014b. Disponível em: <https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-970/specifications>.

NVIDIA Corporation. NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made. p. 1–32, 2014c.

NVIDIA Corporation. **Profiler User ' S Guide**. [S.l.]: NVIDIA Corporation, 2016.

NVIDIA Corporation. **GTX 780 Specifications**. 2018a. Disponível em: <https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications>.

PACHECO, P. S. **An Introduction to Parallel Programming**. Burlington, USA: Elsevier, 2011. 391 p. ISBN 978-0-12-374260-5.

PASSARIN, T. A. R. et al. Reconstrução em ultrassom usando mínimos quadrados Iterativamente reponderados para regularização L1. **XXIII Congresso Brasileiro em Engenharia Biomédica**, p. 1–5, 2012.

RAMM, A. G. **Inverse Problems - Mathematical and Analytical Techiques with Application to Engineering**. [S.l.]: Springer, 2005. 442 p. ISBN 0387231951.

REGINSKA, T. A Regularization Parameter in Discrete Ill-Posed Problems. **Journal on Scientific Computing**, v. 17, n. 3, p. 740–749, 2012.

REVOL, N.; THEVENY, P. Numerical reproducibility and parallel computations: Issues for interval algorithms. **IEEE Transactions on Computers**, v. 63, n. 8, p. 1915–1924, 2014. ISSN 00189340.

SAAD, Y. **Iterative Methods for Sparse Linear Systems**. 2d edition. ed. [S.l.]: SIAM, 2003. 535 p. ISBN 978-1-57735-281-5.

SCHIWIETZ, T. et al. MR Image Reconstruction Using the GPU. **SPIE Medical Imaging**, p. 1279–1290, 2006b. ISSN 0277786X.

SCHIWIETZ, T. et al. MR image reconstruction using the GPU. **Medical Imaging**, p. 61423T–61423T, 2006b.

STANDARDS, M.; SOCIETY, C. **IEEE Std 754-2008 , IEEE Standard for Floating-Point Arithmetic (Revision of IEEE Std 754-1985)**. [S.l.: s.n.], 2008. 1–58 p. ISSN 01419331. ISBN 9780738157528.

STEPANISHEN, P. R. The Time-Dependent Force and Radiation Impedance on a Piston in a Rigid Infinite Planar Baffle. **The Journal of the Acoustical Society of America**, v. 49, n. 3B, p. 841–849, 2005. ISSN 0001-4966.

STERGIOPOULOS, S. **Advanced Signal Processing**. 2nd ed.. ed. [S.l.]: Taylor & Francis Group, 2009. 723 p. ISBN 9781420062380.

STONE, S. S. et al. Accelerating advanced MRI reconstructions on GPUs. **Journal of Parallel and Distributed Computing**, v. 68, n. 10, p. 1307–1318, 2008. ISSN 07437315.

STRASSEN, V. Gaussian elimination is not optimal. **Numerische Mathematik**, v. 13, n. 4, p. 354–356, 1969. ISSN 0029599X.

SYNNEVAG, J. F.; AUSTENG, A.; HOLM, S. Adaptive beamforming applied to medical ultrasound imaging. **Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on**, v. 54, n. 8, p. 1606–1613, 2007. ISSN 08853010.

SZASZ, T.; BASARAB, A.; KOUAME, D. Beamforming Through Regularized Inverse Problems in Ultrasound Medical Imaging. **IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control**, v. 63, n. 12, p. 2031–2044, 2016. ISSN 08853010.

TOMOV, S. et al. Dense linear algebra solvers for multicore with GPU accelerators. **Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010**, 2010. ISSN 1878-3449.

VALENTE, S. A. **Reconstrução de Imagens de Ultrassom usando Esparsidade - Metódos Iterativos Rápidos**. 118 p. Tese (Doutorado) — Universidade Tecnológica Federal do Paraná, 2017. Disponível em: <http://repositorio.utfpr.edu.br/jspui/handle/1/2583>.

VALENTE, S. A. et al. An assessment of iterative reconstruction methods for sparse ultrasound imaging. **Sensors (Switzerland)**, v. 17, n. 3, 2017. ISSN 14248220.

VERASONICS. **The Vantage 256, Vantage 128 & Vantage 64 LE Systems Verasonics**. [S.l.]: Verasonics Inc., 2015. 4 p.

VIOLA, F.; ELLIS, M. A.; WALKER, W. F. Time-domain optimized near-field estimator for ultrasound imaging: Initial development and results. **IEEE Transactions on Medical Imaging**, v. 27, n. 1, p. 99–110, 2008. ISSN 02780062.

VOGEL, C. R. **Computational Methods for Inverse Problems**. [s.n.], 2002. ISBN 978-0-89871-550-7. Disponível em: <http://epubs.siam.org/doi/book/10.1137/1.9780898717570>.

VOTANO, J.; PARHAM, M.; HALL, L. **Introduction to Parallel Processing Algorithms and Architectures**. [s.n.], 2004. 1–27 p. ISBN 0306469642. Disponível em: <http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstract>.

WEBB, A. **Introduction To Biomedical Imaging**. [S.l.]: Wiley, 2003. 272 p. ISBN 0471237663.

XU, W. et al. Auto-tuning GEMV on many-core GPU. **Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS**, p. 30–36, 2012. ISSN 15219097.

ZANIN, L. G. D. S. **Reconstrução de Imagens de Ultrassom Baseada em Problemas Inversos (Dissertação de Mestrado)**. 2011. 109 p.

ZANIN, L. G. S.; ZIBETTI, M. V. W.; SCHNEIDER, F. K. Conjugate gradient and regularized inverse problem-based solutions applied to ultrasound image reconstruction. **IEEE International Ultrasonics Symposium, IUS**, n. 1, p. 377–380, 2011. ISSN 19485719.

ZEMP, R. J.; ABBEY, C. K.; INSANA, M. F. Linear system models for ultrasonic imaging: Application to signal statistics. **IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control**, v. 50, n. 6, p. 642–654, 2003. ISSN 08853010.

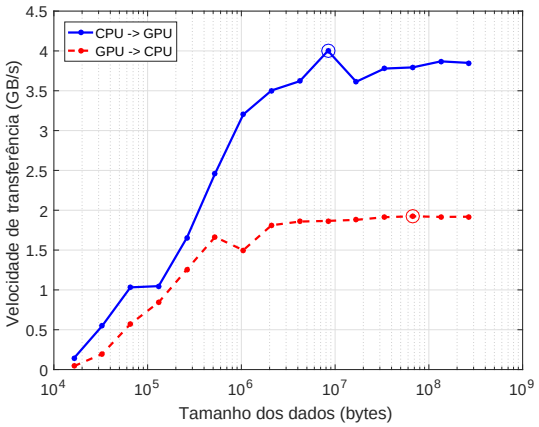
ZIBETTI, M. V.; BAZÁN, F. S.; MAYER, J. Determining the regularization parameters for super-resolution problems. **Signal Processing**, v. 88, n. 12, p. 2890–2901, 2008. ISSN 01651684.

ZIBULEVSKY, M.; ELAD, M. L1-L2 optimization in signal and image processing. **IEEE Signal Processing Magazine**, v. 27, n. 3, p. 76–88, 2010. ISSN 10535888.

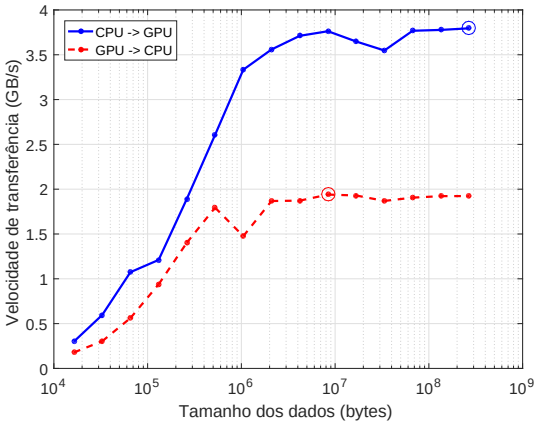
APÊNDICE A – DESEMPENHO EM OPERAÇÕES BÁSICAS

Resultados complementares para o teste de desempenho das GPUs empregadas neste trabalho.

A Figura 38 mostra os resultados para a largura de banda na transferência de dados entre a CPU e a GPU e vice-versa.



(a) GTX780

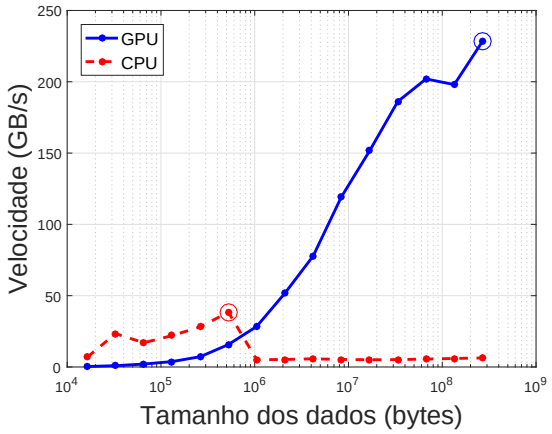


(b) GTX970

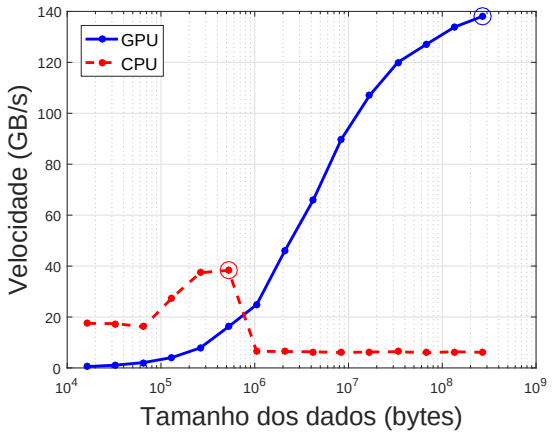
Figura 38: Larguras de banda para transferência de dados.
Fonte: Própria.

Neste caso é importante observar que a velocidade de transferência de dados da CPU para a GPU é bem maior do que a velocidade no sentido contrário. Isto é um fator relevante para os objetivos deste trabalho, pois, a maior quantidade de dados a serem transferidos será sempre no sentido da GPU, a matriz **H** é a principal estrutura a ser transferida e a de maior custo também. Esta característica é uma grande vantagem no emprego de GPUs para a reconstrução de imagens de ultrassom por problemas inversos.

Na Figura 39 são apresentados os resultados para o teste de velocidade de leitura e gravação de dados para ambas as GPUs. Este teste consiste em uma leitura de dados seguida de uma soma de um valor escalar aleatório que resulta em uma alteração do valor armazenado numa determinada posição de um vetor.



(a) GTX780



(b) GTX970

Figura 39: Velocidade de leitura e gravação.
Fonte: Própria.

APÊNDICE B – DESEMPENHO COM MATRIZES ESPARSAS

Resultados complementares para os testes com matrizes esparsas.

Na Figura 40 é possível ver um comparativo de desempenho para a operação de multiplicação matriz-vetor com armazenamento esparso entre CPU e GPU executado na GTX970.

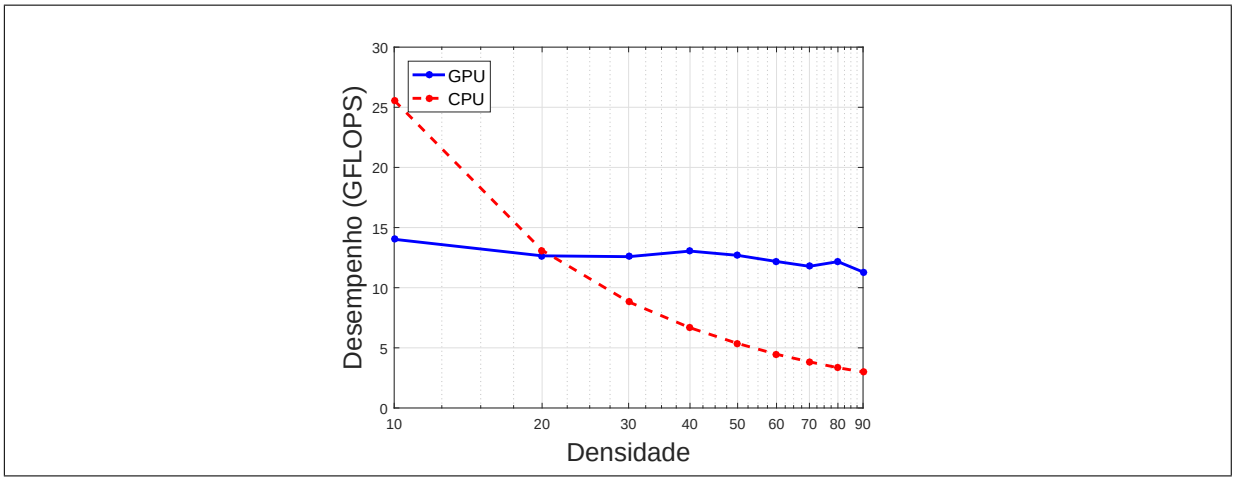


Figura 40: Comparativo de desempenho na multiplicação matriz-vetor com armazenamento esparso em precisão dupla executado na GTX970.
Fonte: Própria.

APÊNDICE C – ARQUITETURAS DAS GPUS

C.1 ARQUITETURA KEPLER

O modelo GTX 780 é baseado nesta arquitetura que foi lançada pela Nvidia Corporation em 2012 e possui em sua versão completa 15 multi-processadores de fluxo Kepler (SMX) e 6 controladores de memória de 64-bits. Seu diagrama de blocos completo pode ser visto na Figura 41 (NVIDIA Corporation, 2012).

Figura 41: Diagrama de blocos completo da arquitetura Kepler.
Fonte: NVIDIA Corporation (2012).

Cada unidade SMX contém 192 núcleos de precisão simples da *Compute Unified*

Device Architecture (CUDA), que é uma biblioteca disponibilizada pela Nvidia Corporation para o desenvolvimento de programas para as suas diversas GPUs (NVIDIA Corporation, 2014ba). Cada núcleo CUDA está equipado com unidades lógico-aritméticas completas para ponto flutuante e inteiros.

As unidades SMX agendam as *threads* em grupos de 32 *threads* simultâneas que são chamados *warps*. Cada SMX possui 4 escalonadores de *warps* possibilitando que até 4 *warps* sejam executados concorrentemente. Os escalonadores possuem duas unidades de disparo de instruções cada, sendo possível a execução de duas instruções simultâneas. A Figura 42 ilustra o esquema de funcionamento de um *warp* (NVIDIA Corporation, 2012).

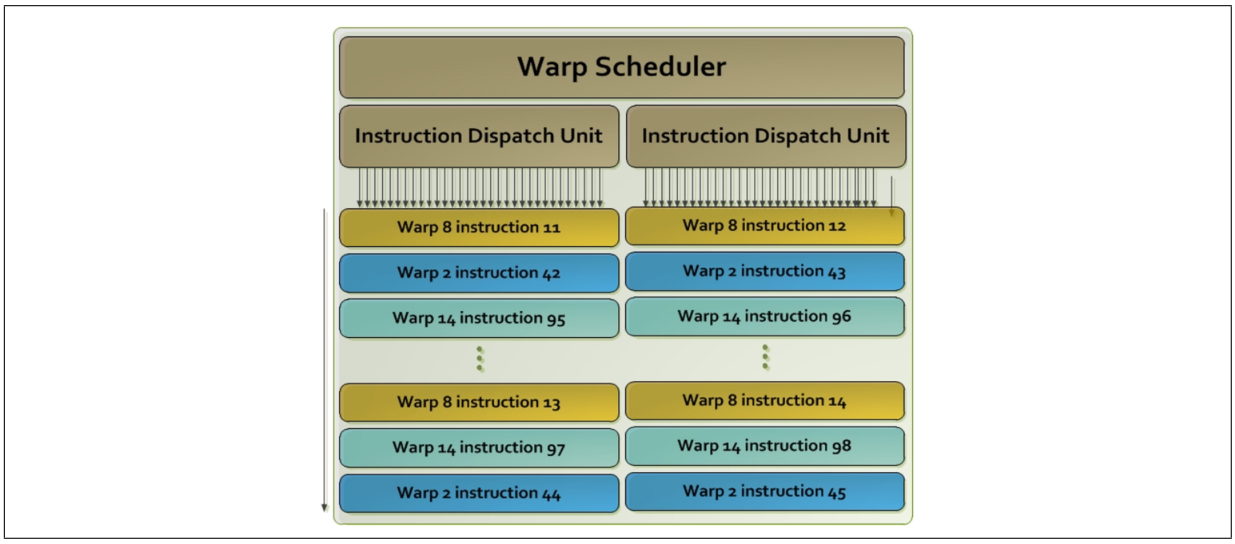


Figura 42: Esquema de funcionamento de um *warp* Kepler.
Fonte: NVIDIA Corporation (2012).

A memória na arquitetura Kepler está organizada em memória principal, *cache* L1 e L2, memória apenas de leitura e memória compartilhada. Cada SMX possui 64 KB disponíveis para a memória compartilhada e o *cache* L1, que pode ser divida em 48 KB para memória compartilhada e 16 KB para o *cache* L1, ou 16 KB por 48 KB ou ainda 32 KB / 32 KB. Estão disponíveis, também, outros 48 KB de memória *cache* apenas de leitura. O tamanho de memória *cache* L2 é de 1,536 KB (NVIDIA Corporation, 2012).

C.2 ARQUITETURA MAXWELL

Lançada em 2014, esta arquitetura é composta por 4 conjuntos de processadores gráficos (GPC) com 16 multi-processadores de fluxo Maxwell (SMM) e 4 controladores de memória. A SMM possui 128 núcleos CUDA divididos em 4 blocos distintos de 32 núcleos cada. Nesta arquitetura se procurou alinhar os blocos de núcleos CUDA com o tamanho do *warp*, isto favorece uma utilização mais eficiente dos recursos (NVIDIA Corporation, 2014c). A Figura 43 apresenta o diagrama de blocos desta arquitetura (NVIDIA Corporation, 2014c).

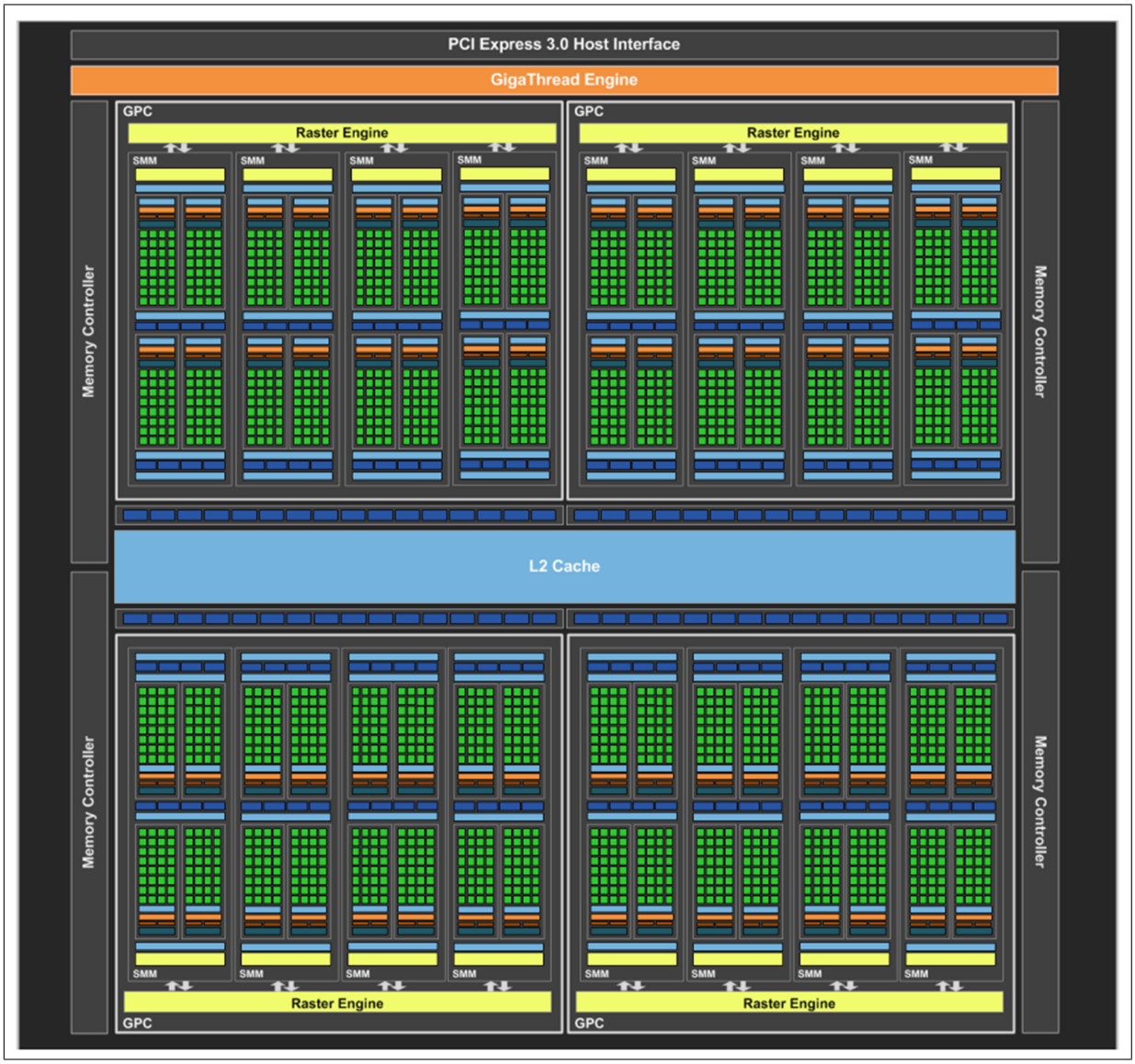


Figura 43: Diagrama de blocos completo da arquitetura Maxwell.
Fonte: NVIDIA Corporation (2014c).

Da mesma forma que na arquitetura Kepler, aqui cada SMM possui 4 escalonadores

de *warps* com capacidade de executar duas instruções simultaneamente. Neste caso foram introduzidas melhorias nos processos de escalonamento visando melhorar o desempenho. Outra diferença é na organização da memória, ao invés da memória compartilhada dividir espaço com a memória *cache* L1 ela passou a ser uma unidade de 96 KB exclusiva (NVIDIA Corporation, 2014c). Esta arquitetura equipa o modelo GTX 970 utilizado neste trabalho.

APÊNDICE D – CODIFICAÇÃO EM CUDA

A CUDA permite ao programador organizar os blocos de *threads* em estruturas lógicas multidimensionais chamadas *grid*, que podem ser uni, bi ou tridimensional. Esta estrutura permite a identificação única de cada *thread* sendo executada, isto é feito através de coordenadas de acordo como o número de dimensões do *grid*, um identificador para o unidimensional, um par de coordenadas (x,y) para um *grid* bidimensional e uma túpla (x,y,z) para os tridimensionais. Através destes identificadores é possível se customizar o código para manipular porções específicas dos dados ou executar tarefas diferenciadas (NVIDIA Corporation, 2014ba).

A Figura 44 apresenta duas porções de código para a implementação de uma função presente nas diversas versões da biblioteca BLAS, a SAXPY. Esta função executa a operação matricial $\mathbf{y} = a * \mathbf{x} + \mathbf{y}$, a implementação serial é um laço simples que calcula um elemento de **y** a cada iteração. Na versão paralela o *kernel* executa cada uma das operações de forma independente e simultaneamente, atribuindo para cada *thread* o cálculo de um elemento de **y**. A palavra reserva __**global**__ indica a definição de um *kernel* CUDA e tem uma extensão sintática para a chamada da função em questão: $\langle\langle\langle \mathbf{B}, \mathbf{T} \rangle\rangle\rangle$, onde **B** indica o número de blocos a serem criados e **T** o número de *threads* para cada bloco. Cada *thread* utiliza suas coordenadas para calcular sobre qual elemento de **y** deve atuar (GARLAND et al., 2008).

Este exemplo mostra uma forma bastante comum de realizar o paralelismo de um laço sequencial no qual as operações são independentes umas das outras. Ele é, também, um exemplo de paralelismo de dados onde o trabalho pode ser dividido para operar sobre porções independentes de dados.

Os recursos da CUDA permitem que o programador escreva o código necessário com poucas adaptações em relação a linguagem C usual. Desta forma, ele pode se concentrar em

```
void saxpy(int n, float a, float *x, float *y)
{
for (int i = 0; i <n; ++i)
y[i] = a*x[i] + y[i];
}

void main()
{
...
// Executa a função SAXPY
saxpy(n, 2.0, x, y);
...
}
```

: (a) Sequencial

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i<n) y[i] =  a*x[i] + y[i];
}

void main()
{
...
// Executa o kernel SAXPY em paralelo
// com (n/256) blocos de 256 threads cada
saxpy <<ceil(n/256), 256>> (n, 2.0, x, y);
...
}
```

: (b) Paralelo

Figura 44: Exemplos de códigos para a função SAXPY ($\mathbf{y} = a * \mathbf{x} + \mathbf{y}$) em implementações sequencial e paralela.
Fonte: Adaptado de Garland et al. (2008).

resolver o problema em questão ao invés de programar detalhes internos da GPU. Cada *thread* será mapeada pela CUDA para uma *thread* física residente na GPU e cada bloco de *threads* será alocado fisicamente em uma SM e suas coordenadas serão acessíveis através das variáveis internas pré-definidas **blockIdx.***, **blockDim.*** e **threadIdx.***.

APÊNDICE E – ALGORITMOS PARALELOS PARA ÁLGEBRA LINEAR

E.1 ALGORITMO DE STRASSEN-WINOGRAD

Este algortimo é uma variação do algoritmo original de Strassen e é utilizado na biblioteca CUDA BLAS (CUBLAS) da NVIDIA (NVIDIA Corporation, 2014a). Sendo $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, onde $\mathbf{A}$ é $m \times k$, $\mathbf{B}$ é $k \times n$ e $\mathbf{C}$ é $m \times n$. A Figura 45 ilustra a decomposição da multiplicação matricial em blocos onde $\mathbf{A}_{11}$ tem dimensões $\frac{m}{2} \times \frac{k}{2}$, $\mathbf{B}_{11}$ é $\frac{k}{2} \times \frac{n}{2}$ e $\mathbf{C}_{11}$ é $\frac{m}{2} \times \frac{n}{2}$. O algoritmo de multiplicação matricial comum requer mnk multiplicações e $mnk - mn$ adições, assim temos um total de $2mnk - mn$ operações ou $2N^3 - N^2(O(N^3))$, se considerarmos matrizes quadradas com $m = n = k = N$.

$$\begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \times \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}$$

Figura 45: Decomposição da multiplicação de matrizes em blocos.
Fonte: Adaptado de Lai et al. (2013).

Strassen propôs um algoritmo capaz de resolver a multiplicação das 2×2 submatrizes da Figura 45 através de 7 multiplicações e 18 adições, ao invés das 8 multiplicações e 4 adições dos algoritmos convencionais. A variante de Winograd conseguiu reduzir número de adições para 15, como visto na Figura 46, o que torna esta opção melhor para aplicações práticas (LAI et al., 2013).

Essas submatrizes são chamadas de *thread blocks*, segundo a terminologia CUDA. O tamanho dessas submatrizes, ou dos *thread blocks*, deve ser configurada de forma fixa via codificação e é crucial para o bom desempenho (NVIDIA Corporation, 2014a; NATH et al., 2010).

$\mathbf{S}_1 = \mathbf{A}_{21} + \mathbf{A}_{22}$	$\mathbf{M}_1 = \mathbf{S}_2 \times \mathbf{S}_6$	$\mathbf{V}_1 = \mathbf{M}_1 + \mathbf{M}_2$
$\mathbf{S}_2 = \mathbf{S}_1 - \mathbf{A}_{11}$	$\mathbf{M}_2 = \mathbf{A}_{11} \times \mathbf{B}_{11}$	$\mathbf{V}_2 = \mathbf{V}_1 + \mathbf{M}_4$
$\mathbf{S}_3 = \mathbf{A}_{11} - \mathbf{A}_{21}$	$\mathbf{M}_3 = \mathbf{A}_{12} \times \mathbf{B}_{21}$	$\mathbf{V}_3 = \mathbf{M}_5 + \mathbf{M}_6$
$\mathbf{S}_4 = \mathbf{A}_{12} - \mathbf{S}_2$	$\mathbf{M}_4 = \mathbf{S}_3 \times \mathbf{S}_7$	$\mathbf{C}_{11} = \mathbf{M}_2 + \mathbf{M}_3$
$\mathbf{S}_5 = \mathbf{B}_{12} - \mathbf{B}_{11}$	$\mathbf{M}_5 = \mathbf{S}_1 \times \mathbf{S}_5$	$\mathbf{C}_{12} = \mathbf{V}_1 + \mathbf{V}_3$
$\mathbf{S}_6 = \mathbf{B}_{22} - \mathbf{S}_5$	$\mathbf{M}_6 = \mathbf{S}_4 \times \mathbf{B}_2$	$\mathbf{C}_{21} = \mathbf{V}_2 - \mathbf{M}_7$
$\mathbf{S}_7 = \mathbf{B}_{22} - \mathbf{B}_{12}$	$\mathbf{M}_7 = \mathbf{A}_{22} \times \mathbf{S}_8$	$\mathbf{C}_{22} = \mathbf{V}_2 + \mathbf{M}_5$
$\mathbf{S}_8 = \mathbf{S}_6 - \mathbf{B}_{21}$		

Figura 46: Algoritmo de Strassen-Winograd para multiplicação de matrizes.
Fonte: Adaptado de Lai et al. (2013).

As versões de algoritmos GEMMs empregadas neste trabalho se baseiam nas rotinas disponíveis na biblioteca BLAS "*Matrix Algebra on GPU and Multicore Architetures*"(MAGMA) (TOMOV et al., 2010). As rotinas usadas executam a operação $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$, onde α e β são escalares. Este algoritmos foram projetados para tirar o máximo de proveito das características físicas das GPUs atuais, tais como: a memória compartilhada, o número de registradores, o número de núcleos CUDA, dentre outras (NATH et al., 2010).

As operações são organizadas através de um *grid* bidimensional de blocos de *threads* de dimensões $TB_x \times TB_y$. Cada bloco é associado a $TH = TH_x \times TH_y$ *threads*. As submatrizes de **A** e de **B** são carregadas em memória compartilhada. Como exemplo de uma possível parametrização temos: $TB_x = TB_y = 64$ e $TH_x = TH_y = 16$, desta forma cada bloco de 16×16 *threads* calcula 64×64 elementos da matriz **C**, ou seja, cada *thread* calcula 16 elementos. A Figura 47 ilustra este procedimento (NATH et al., 2010, 2011).

Os blocos de 64×64 da matriz **C** são divididos em 16 sub-blocos de dimensões $SB_x \times SB_y$, no exemplo $SB_x = SB_y = 16$. Cada um desdes 16×16 sub-blocos são operados por 16×16 *threads*, isto é, um elemento é calculado em uma *thread*. Mais especificamente, um elemento (x, y) de um sub-bloco será calculado pela *thread* (x, y) do bloco de *threads* (NATH et al., 2010).

A cada iteração todas as *threads* de um bloco de *threads* carregam para a memória compartilhada 64×16 elementos de **A** e 16×64 elementos de **B** de maneira coesa, promovendo o máximo de localização espacial e temporal para os dados[1]. Esta divisão permite executar as

[1]**Localização espacial** diz respeito a forma como os dados são armazenados em memória, onde o acesso a

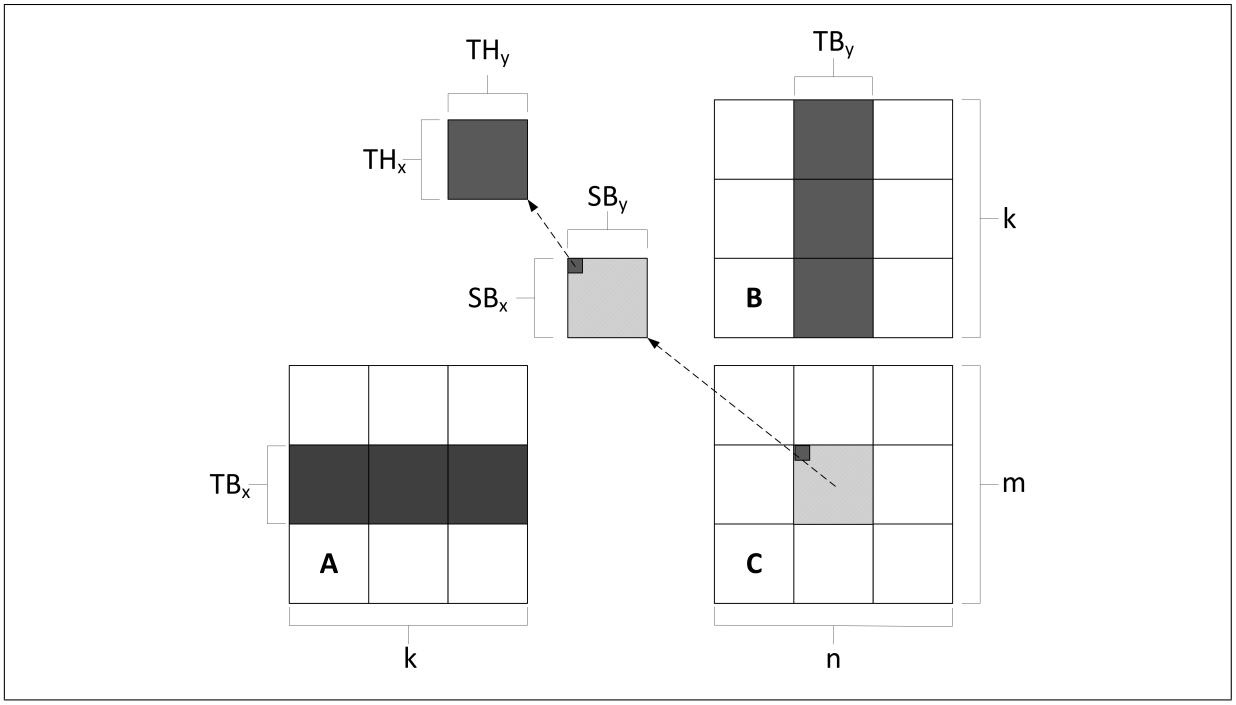


Figura 47: Diagrama ilustrativo do processo de particionamento das matrizes para multiplicação paralela.
Fonte: Adaptado de Nath et al. (2010).

transferências dos resultados finais dos registradores para a matriz **C** armazenada em memória global de maneira coesa (NATH et al., 2010).

Se os dados forem de precisão simples, um processo semelhante pode ser aplicado. Para este exemplo uma melhor divisão seria de blocos de 96×96 com 16×16 *threads*, onde cada *thread* opera uma matriz de 6×6 elementos (NATH et al., 2010).

De acordo com Nath et al. (2010), esta abordagem favorece o processo de "*auto-tune*"baseado nos tamanhos das matrizes envolvidas, permitindo uma definição mais apropriada para cada situação alcançando um melhor desempenho.

uma posição de memória é seguido por um próximo acesso a uma posição de memória próxima. **Localização temporal** é o conceito onde os dados acessados num futuro próximo (por exemplo, uma instrução seguinte) estão localizados espacialmente próximos. Estes conceitos são fundamentais para o bom aproveitamento das memórias do tipo *cache* (PACHECO, 2011).

APÊNDICE F – MULTIPLICAÇÃO MATRIZ-VETOR PARALELA E SIMÉTRICA

```
1   /*
2    *  Modelo de estrutura de programa para utilização de recursos CUDA
3    *  através de funções MatLab.
4    *
5    *  Este modelo trata os parâmetros de entrada e saída do MatLab para o
6    *  código CUDA e vice-versa.
7    *
8    *  Release notes:
9    *          - Implementação do uso de const para a matriz H e o vetor g;
10   *          - Pré-alocação de memória para o vetor f;
11   */
12
13  /*
14   *  Bibliotecas de uso geral do C/C++
15   *
16   */
17
18  #include <stdio.h>
19  #include <math.h>
20
21  /*
22   *  Bibliotecas de uso específico do CUDA
23   *
24   */
25
26  #include <cuda.h>
27  #include <cuda_profiler_api.h>
28  #include <cublas_v2.h>
29  #include <cuComplex.h>
30  #include <math_functions.h>
31
32  //#include <helper_cuda.h>
33
34  /*
35   *  Biblioteca de integração MatLab C/C++ e CUDA
36   *
37   */
38
39  #include "mex.h"
40  #include "gpu/mxGPUArray.h"
41  #include "matrix.h"
42
43  /*
44   *  Definição do tamanho do grid para operação não transposta.
45   *
46   */
47
48  #define     nDIM_X          512
49  #define     nDIM_Y          1
50  #define     nTILE_SIZE      512
51
52  /*
53   *  Definição do tamanho do grid para operação transposta.
54   *
55   */
56  // #define     tDIM_X          128
57  // #define     tDIM_Y          1
58  // #define     tTILE_SIZE      1
59
60  #define     tDIM_X          512
61  #define     tDIM_Y          1
62  #define     tTILE_SIZE      512
63
64  //#define usefixedcondition
65
66  // ----------------------------------------
67  /// Does sum reduction of n-element array x, leaving total in x[0].
68  /// Contents of x are destroyed in the process.
69  /// With k threads, can reduce array up to 2*k in size.
70  /// Assumes number of threads <= 1024 (which is max number of threads up to CUDA
    capability 3.0)
71  /// Having n as template parameter allows compiler to evaluate some conditions at
    compile time.
```

```
72  /// Calls __syncthreads before & after reduction.
73  __device__ void magma_sum_reduce(
74      int n, int i, float *x)
75  {
76      __syncthreads();
77      if (n > 1024) { if (i < 1024 && i + 1024 < n) { x[i] += x[i + 1024]; }
        __syncthreads(); }
78      if (n > 512) { if (i < 512 && i + 512 < n) { x[i] += x[i + 512]; }
        __syncthreads(); }
79      if (n > 256) { if (i < 256 && i + 256 < n) { x[i] += x[i + 256]; }
        __syncthreads(); }
80      if (n > 128) { if (i < 128 && i + 128 < n) { x[i] += x[i + 128]; }
        __syncthreads(); }
81      if (n > 64) { if (i < 64 && i + 64 < n) { x[i] += x[i + 64]; }
        __syncthreads(); }
82      if (n > 32) { if (i < 32 && i + 32 < n) { x[i] += x[i + 32]; }
        __syncthreads(); }
83      // probably don't need __syncthreads for < 16 threads
84      // because of implicit warp level synchronization.
85      if (n > 16) { if (i < 16 && i + 16 < n) { x[i] += x[i + 16]; }  __syncthreads(); }
86      if (n > 8) { if (i < 8 && i + 8 < n) { x[i] += x[i + 8]; }  __syncthreads(); }
87      if (n > 4) { if (i < 4 && i + 4 < n) { x[i] += x[i + 4]; }  __syncthreads(); }
88      if (n > 2) { if (i < 2 && i + 2 < n) { x[i] += x[i + 2]; }  __syncthreads(); }
89      if (n > 1) { if (i < 1 && i + 1 < n) { x[i] += x[i + 1]; }  __syncthreads(); }
90  }
91  // end sum_reduce
92
93  ////////////////////////////////////////////////////////////////////////////////
    ////
94  /*
95      Calcula o índice de acordo com os parâmetros de simetria da matriz H.
96
97      Argumentos Entrada:
98          ind --> índice original do elemento da matriz H
99          M   --> tamanho original da matriz H
100         Ne  --> número de elementos do transdutor
101         S   --> tamanho da amostra
102
103     Argumento de saída:
104         ind --> novo valor do índice do elemento pela simetria
105 */
106 __device__ int GetElementIndex(
107     int ind, const int M, const int Ne, const int S)
108 {
109     int cb = 0, cf = 0, cn = 0;
110     if (ind < (M * Ne * S / 2))
111     {
112         return ind;
113     }
114     else
115     {
116         cb = ind / S + 1;
117         cf = ind % S;
118         cn = (Ne*M) - cb + 1;
119         ind = (cn - 1)*S + cf;
120
121         return ind;
122     }
123 }
124
125 ////////////////////////////////////////////////////////////////////////////////
    ////
126 /*
127     Operação H x g (não transposta).
128 */
129 __global__ void Symmetric_v2_2(
130     int m, int n, int Ne, int S, float alpha,
131     const float *__restrict__ dH, int lda,
132     const float *__restrict__ dg, int incg, float beta,
133     float       *__restrict__ df, int incf)
134 {
135     if (m <= 0 || n <= 0) return;
136
```

```
137       int num_threads = blockDim.x * blockDim.y * blockDim.z;
138
139       if (nDIM_X * nDIM_Y != num_threads) return; // need to launch exactly the same
          number of threads as template parameters indicate
140
141       int thread_id = threadIdx.x + threadIdx.y * blockDim.x;
142
143       // threads are all configurated locally
144       int tx = thread_id % nDIM_X;
145       int ty = thread_id / nDIM_X;
146
147       int ind = blockIdx.x*nTILE_SIZE + tx;
148       int indH = 0;
149       int OffSetH =0;
150
151       __shared__ float sdata[nDIM_X * nDIM_Y];
152
153       int st = blockIdx.x * nTILE_SIZE;
154
155       int ed = min(st + nTILE_SIZE, ((m + nDIM_X - 1) / nDIM_X) * nDIM_X);
156
157       int iters = (ed - st) / nDIM_X;
158
159       for (int i = 0; i < iters; i++)
160       {
161           if (ind < m) OffSetH = ind;
162
163           float res = 0.0f;
164
165           if (ind < m)
166           {
167               for (int col = ty; col < n; col += nDIM_Y)
168               {
169                   indH = GetElementIndex(OffSetH + col * lda, n, Ne, S);
170                   res += dH[indH] * dg[col*incg];
171               }
172           }
173
174           if (nDIM_X >= num_threads) // indicated 1D threads configuration. Shared
              memory is not needed, reduction is done naturally
175           {
176               if (ty == 0 && ind < m)
177               {
178                   df[ind*incf] = alpha*res + beta*df[ind*incf];
179               }
180           }
181           else
182           {
183               sdata[ty + tx * nDIM_Y] = res;
184
185               __syncthreads();
186
187               if (nDIM_Y > 16)
188               {
189                   magma_sum_reduce(nDIM_Y, ty, sdata + tx * nDIM_Y);
190               }
191               else
192               {
193                   if (ty == 0 && ind < m)
194                   {
195                       for (int i = 1; i < nDIM_Y; i++)
196                       {
197                           sdata[tx * nDIM_Y] += sdata[i + tx * nDIM_Y];
198                       }
199                   }
200               }
201
202               if (ty == 0 && ind < m)
203               {
204                   df[ind*incf] = alpha*sdata[tx * nDIM_Y] + beta*df[ind*incf];
205               }
206
207               __syncthreads();
```

```
208         }
209
210         if (ind < m) OffSetH = 0;
211
212         ind += nDIM_X;
213     }
214 }
215
216 ////////////////////////////////////////////////////////////////////////////////
    ////
217 /*
218     Operação H' x g (transposta).
219 */
220 __global__ void Symmetric_Trans_v2_2(
221     int m, int n, int Ne, int S, float alpha,
222     const float *__restrict__ dH, int lda,
223     const float *__restrict__ dg, int incg, float beta,
224     float       *__restrict__ df, int incf)
225 {
226     if (m <= 0 || n <= 0) return;
227
228     int num_threads = blockDim.x * blockDim.y * blockDim.z;
229
230     // need to launch exactly the same number of threads as template parameters
        indicate
231     if (tDIM_X * tDIM_Y != num_threads) return;
232
233     int thread_id = threadIdx.x + threadIdx.y * blockDim.x;
234
235     // threads are all configurated locally
236     int tx = thread_id % tDIM_X;
237     int ty = thread_id / tDIM_X;
238
239     __shared__ float sdata[tDIM_X * tDIM_Y];
240
241     float res;
242     int mfull = (m / tDIM_X) * tDIM_X;
243
244     int start = blockIdx.x * tTILE_SIZE + ty;
245     int iters;
246
247     int offsetH = 0;
248
249     //#define usefixedcondition
250 #ifdef usefixedcondition
251     /*fixed condition*/
252     iters = tTILE_SIZE / tDIM_Y;
253 #else
254     /* flexible condition based on global n (has drops when size is roughly bigger
        than nTILE_SIZE)*/
255     //int iters = magma_ceildiv(min(n,nTILE_SIZE), nDIM_Y);
256
257     /* flexible condition based on my local nloc=ed-st*/
258     int st = blockIdx.x * tTILE_SIZE;
259     //int ed = magma_ceildiv( min(n, st + nTILE_SIZE), nDIM_Y ) * nDIM_Y;
260     int ed = min(st + tTILE_SIZE, ((n + tDIM_Y - 1) / tDIM_Y) * tDIM_Y);
261     iters = (ed - st) / tDIM_Y;
262 #endif
263
264     if (tx < m) offsetH += tx;
265
266     for (int i = 0; i < iters; i++)// at 2Gflops/ overhead
267                                    //for (int col=start; col <
                                       (blockIdx.x+1)*nTILE_SIZE; col += nDIM_Y)// at
                                       least 3Gflop/s overhead
268     {
269         int indH = 0;
270         int col = start + i * tDIM_Y;
271
272         if (col < n) offsetH += col*lda;
273
274         res = 0.0f;
275
```

```
276           // partial sums
277           if (col < n)
278           {
279               for (int i = 0; i < mfull; i += nDIM_X) {
280                   indH = GetElementIndex(i + offsetH, n, Ne, S);
281                   res += dH[indH] * dg[(tx + i)*incg];
282               }
283               if (tx + mfull < m) {
284                   //res += dH[GetElementIndex(mfull + offsetH, m, Ne, S)] * dg[(tx +
                      mfull)*incg];
285
286                   indH = GetElementIndex(mfull + offsetH, n, Ne, S);
287
288                   res += dH[indH] * dg[(tx + mfull)*incg];
289               }
290           }
291           sdata[tx + ty * tDIM_X] = res;
292
293           // tree reduction of partial sums,
294           // from nDIM_X sums to ... 128 to 64 to 32 ... to 1 sum in sdata[0]
295           if (tDIM_X > 16)
296           {
297               magma_sum_reduce(tDIM_X, tx, sdata + ty * tDIM_X);
298           }
299           else
300           {
301               __syncthreads();
302
303               if (tx == 0 && col < n)
304               {
305                   for (int i = 1; i < m && i < tDIM_X; i++)
306                   {
307                       sdata[0 + ty * tDIM_X] += sdata[i + ty * tDIM_X];
308                   }
309               }
310               __syncthreads();
311           }
312
313           if (tx == 0 && col < n) {
314               df[col*incf] = alpha*sdata[0 + ty * tDIM_X] + beta*df[col*incf];
315           }
316
317           __syncthreads();
318
319           if (col < n)  offsetH -= col * lda;
320       }
321   }
322
323   ////////////////////////////////////////////////////////////////////////////////
      ////
324   /*
325    *  Função de integração MatLab C/C++/CUDA.
326    *
327    *  Esta função deve verificar os parâmetros de entrada e saída,
328    *  tratar os dados recebidos e preparar os dados para devolução ao
329    *  MatLab.
330    *
331    *  Tratamento direto de ponteiros alocados em GPU pelo script MatLab.
332    *
333    *  Argumentos Entrada:
334    *           argin[0] --> matriz H
335    *           argin[1] --> vetor g
336    *           argin[2] --> vetor f
337    *           argin[3] --> Total de linhas da matriz H (N)
338    *           argin[4] --> Total de colunas da matriz H (M)
339    *           argin[5] --> Nro elementos (Ne)
340    *           argin[6] --> Nro amostras (S)
341    *           argin[7] --> Calcular H transposta (0 - Normnal, 1 - Transposta);
342    */
343
344   void mexFunction(int nargout, mxArray *argout[], int nargin, mxArray *argin[])
345   {
346       mxGPUArray const *H;
```

```
347       mxGPUArray const *g;
348       mxGPUArray const *f;
349       cudaError_t e;
350
351       float const *d_H;
352       float const *d_g;
353       float *d_f;
354
355       int N, M, Ne, S, Tr;
356
357       if (nargin != 8)
358           mexErrMsgTxt("Invaid number of input arguments");
359
360       if (nargout != 1)
361           mexErrMsgTxt("Invalid number of outputs");
362
363       /* Initialize the MathWorks GPU API. */
364       mxInitGPU();
365
366       H = mxGPUCreateFromMxArray(argin[0]);
367       e = cudaGetLastError();
368       if (e != cudaSuccess) {
369           mexErrMsgTxt("Erro: Alocação matriz H.");
370       }
371       if (mxGPUGetClassID(H) != mxSINGLE_CLASS)
372           mexErrMsgTxt("The matrix H  must be single (float)");
373
374       g = mxGPUCreateFromMxArray(argin[1]);
375       e = cudaGetLastError();
376       if (e != cudaSuccess) {
377           mexErrMsgTxt("Erro: Alocação vertor g.");
378       }
379       if (mxGPUGetClassID(g) != mxSINGLE_CLASS)
380           mexErrMsgTxt("The vector g must be single (float)");
381
382       f = mxGPUCreateFromMxArray(argin[2]);
383       e = cudaGetLastError();
384       if (e != cudaSuccess) {
385           mexErrMsgTxt("Erro: Alocação vertor f.");
386       }
387       if (mxGPUGetClassID(f) != mxSINGLE_CLASS)
388           mexErrMsgTxt("The vector f must be single (float)");
389
390       d_H = (float const *)(mxGPUGetDataReadOnly(H));
391       e = cudaGetLastError();
392       if (e != cudaSuccess) {
393           mexErrMsgTxt("Erro: Alocação matriz d_H");
394       }
395
396       d_g = (float const *)(mxGPUGetDataReadOnly(g));
397       e = cudaGetLastError();
398       if (e != cudaSuccess) {
399           mexErrMsgTxt("Erro: Alocação vetor d_g");
400       }
401
402       d_f = (float *)(mxGPUGetDataReadOnly(f));
403       e = cudaGetLastError();
404       if (e != cudaSuccess) {
405           mexErrMsgTxt("Erro: Alocação vetor d_f");
406       }
407
408       N = mxGetScalar(argin[3]);
409       M = mxGetScalar(argin[4]);
410       Ne = mxGetScalar(argin[5]);
411       S  = mxGetScalar(argin[6]);
412       Tr = mxGetScalar(argin[7]);
413
414
415
416       if (Tr == 0)
417       {
418           dim3 dim_grid( ceil((float)N / nTILE_SIZE) );
419           dim3 dim_block(nDIM_X, nDIM_Y);
```

```
420
421         Symmetric_v2_2 << < dim_grid, dim_block >> >(N, M, Ne, S, 1, d_H, N, d_g, 1,
            0, d_f, 1);
422     }
423     else
424     {
425         dim3 dim_grid( ceil((float)M / tTILE_SIZE), 1);
426         dim3 dim_block(tDIM_X, tDIM_Y);
427
428         Symmetric_Trans_v2_2 << < dim_grid, dim_block >> >(N, M, Ne, S, 1, d_H, N,
            d_g, 1, 0, d_f, 1);
429     }
430
431     e = cudaGetLastError();
432     if (e != cudaSuccess) {
433         mexErrMsgTxt("Erro: Chamada Symmetric_MxV");
434     }
435
436     e = cudaThreadSynchronize();
437     if (e != cudaSuccess) {
438         // mexErrMsgTxt("Erro: cudaThreadSynchronize");
439          mexErrMsgTxt(cudaGetErrorString(e));
440     }
441
442     argout[0] = mxGPUCreateMxArrayOnGPU(f);
443     e = cudaGetLastError();
444     if (e != cudaSuccess) {
445         mexErrMsgTxt("Erro: mxGPUCreateMxArrayOnGPU(f)");
446     }
447
448     mxGPUDestroyGPUArray(H);
449     e = cudaGetLastError();
450     if (e != cudaSuccess) {
451         mexErrMsgTxt("Erro: mxGPUDestroyGPUArray(H)");
452     }
453
454     mxGPUDestroyGPUArray(g);
455     e = cudaGetLastError();
456     if (e != cudaSuccess) {
457         mexErrMsgTxt("Erro: mxGPUDestroyGPUArray(g)");
458     }
459
460     mxGPUDestroyGPUArray(f);
461     e = cudaGetLastError();
462     if (e != cudaSuccess) {
463         mexErrMsgTxt("Erro: mxGPUDestroyGPUArray(f)");
464     }
465 }
466
467
```

APÊNDICE G – MULTIPLICAÇÃO MATRIZ-MATRIZ PARALELA E SIMÉTRICA

```
1   /*
2    *  Modelo de estrutura de programa para utilização de recursos CUDA
3    *  através de funções MatLab.
4    *
5    *  Este modelo trata os parâmetros de entrada e saída do MatLab para o
6    *  código CUDA e vice-versa.
7    *
8    *  Release notes:
9    *          - Implementação do uso de const para a matriz H e o vetor g;
10   *          - Pré-alocação de memória para o vetor f;
11   */
12
13  /*
14   *  Bibliotecas de uso geral do C/C++
15   *
16   */
17
18  #include <stdio.h>
19  #include <math.h>
20
21  /*
22   *  Bibliotecas de uso específico do CUDA
23   *
24   */
25
26  #include <cuda.h>
27  #include <cuda_profiler_api.h>
28  #include <cublas_v2.h>
29  #include <cuComplex.h>
30  #include <math_functions.h>
31
32  //#include <helper_cuda.h>
33
34  /*
35   *  Biblioteca de integração MatLab C/C++ e CUDA
36   *
37   */
38
39  #include "mex.h"
40  #include "gpu/mxGPUArray.h"
41  #include "matrix.h"
42
43  /*
44   *  Definição do tamanho do grid para operação não transposta.
45   *
46   */
47
48  #define     nDIM_X          512
49  #define     nDIM_Y          1
50  #define     nTILE_SIZE      512
51
52  /*
53   *  Definição do tamanho do grid para operação transposta.
54   *
55   */
56  // #define     tDIM_X          128
57  // #define     tDIM_Y          1
58  // #define     tTILE_SIZE      1
59
60  #define     tDIM_X          512
61  #define     tDIM_Y          1
62  #define     tTILE_SIZE      512
63
64  //#define usefixedcondition
65
66  // ----------------------------------------
67  /// Does sum reduction of n-element array x, leaving total in x[0].
68  /// Contents of x are destroyed in the process.
69  /// With k threads, can reduce array up to 2*k in size.
70  /// Assumes number of threads <= 1024 (which is max number of threads up to CUDA
    capability 3.0)
71  /// Having n as template parameter allows compiler to evaluate some conditions at
    compile time.
```

```
72  /// Calls __syncthreads before & after reduction.
73  __device__ void magma_sum_reduce(
74      int n, int i, float *x)
75  {
76      __syncthreads();
77      if (n > 1024) { if (i < 1024 && i + 1024 < n) { x[i] += x[i + 1024]; }
        __syncthreads(); }
78      if (n > 512) { if (i < 512 && i + 512 < n) { x[i] += x[i + 512]; }
        __syncthreads(); }
79      if (n > 256) { if (i < 256 && i + 256 < n) { x[i] += x[i + 256]; }
        __syncthreads(); }
80      if (n > 128) { if (i < 128 && i + 128 < n) { x[i] += x[i + 128]; }
        __syncthreads(); }
81      if (n > 64) { if (i < 64 && i + 64 < n) { x[i] += x[i + 64]; }
        __syncthreads(); }
82      if (n > 32) { if (i < 32 && i + 32 < n) { x[i] += x[i + 32]; }
        __syncthreads(); }
83      // probably don't need __syncthreads for < 16 threads
84      // because of implicit warp level synchronization.
85      if (n > 16) { if (i < 16 && i + 16 < n) { x[i] += x[i + 16]; }  __syncthreads(); }
86      if (n > 8) { if (i < 8 && i + 8 < n) { x[i] += x[i + 8]; }  __syncthreads(); }
87      if (n > 4) { if (i < 4 && i + 4 < n) { x[i] += x[i + 4]; }  __syncthreads(); }
88      if (n > 2) { if (i < 2 && i + 2 < n) { x[i] += x[i + 2]; }  __syncthreads(); }
89      if (n > 1) { if (i < 1 && i + 1 < n) { x[i] += x[i + 1]; }  __syncthreads(); }
90  }
91  // end sum_reduce
92
93  ////////////////////////////////////////////////////////////////////////////////
    ////
94  /*
95      Calcula o índice de acordo com os parâmetros de simetria da matriz H.
96
97      Argumentos Entrada:
98          ind --> índice original do elemento da matriz H
99          M   --> tamanho original da matriz H
100         Ne  --> número de elementos do transdutor
101         S   --> tamanho da amostra
102
103     Argumento de saída:
104         ind --> novo valor do índice do elemento pela simetria
105 */
106 __device__ int GetElementIndex(
107     int ind, const int M, const int Ne, const int S)
108 {
109     int cb = 0, cf = 0, cn = 0;
110     if (ind < (M * Ne * S / 2))
111     {
112         return ind;
113     }
114     else
115     {
116         cb = ind / S + 1;
117         cf = ind % S;
118         cn = (Ne*M) - cb + 1;
119         ind = (cn - 1)*S + cf;
120
121         return ind;
122     }
123 }
124
125 ////////////////////////////////////////////////////////////////////////////////
    ////
126 /*
127     Operação H x g (não transposta).
128 */
129 __global__ void Symmetric_v2_2(
130     int m, int n, int Ne, int S, float alpha,
131     const float *__restrict__ dH, int lda,
132     const float *__restrict__ dg, int incg, float beta,
133     float       *__restrict__ df, int incf)
134 {
135     if (m <= 0 || n <= 0) return;
136
```

```
137       int num_threads = blockDim.x * blockDim.y * blockDim.z;
138
139       if (nDIM_X * nDIM_Y != num_threads) return; // need to launch exactly the same
          number of threads as template parameters indicate
140
141       int thread_id = threadIdx.x + threadIdx.y * blockDim.x;
142
143       // threads are all configurated locally
144       int tx = thread_id % nDIM_X;
145       int ty = thread_id / nDIM_X;
146
147       int ind = blockIdx.x*nTILE_SIZE + tx;
148       int indH = 0;
149       int OffSetH =0;
150
151       __shared__ float sdata[nDIM_X * nDIM_Y];
152
153       int st = blockIdx.x * nTILE_SIZE;
154
155       int ed = min(st + nTILE_SIZE, ((m + nDIM_X - 1) / nDIM_X) * nDIM_X);
156
157       int iters = (ed - st) / nDIM_X;
158
159       for (int i = 0; i < iters; i++)
160       {
161           if (ind < m) OffSetH = ind;
162
163           float res = 0.0f;
164
165           if (ind < m)
166           {
167               for (int col = ty; col < n; col += nDIM_Y)
168               {
169                   indH = GetElementIndex(OffSetH + col * lda, n, Ne, S);
170                   res += dH[indH] * dg[col*incg];
171               }
172           }
173
174           if (nDIM_X >= num_threads) // indicated 1D threads configuration. Shared
              memory is not needed, reduction is done naturally
175           {
176               if (ty == 0 && ind < m)
177               {
178                   df[ind*incf] = alpha*res + beta*df[ind*incf];
179               }
180           }
181           else
182           {
183               sdata[ty + tx * nDIM_Y] = res;
184
185               __syncthreads();
186
187               if (nDIM_Y > 16)
188               {
189                   magma_sum_reduce(nDIM_Y, ty, sdata + tx * nDIM_Y);
190               }
191               else
192               {
193                   if (ty == 0 && ind < m)
194                   {
195                       for (int i = 1; i < nDIM_Y; i++)
196                       {
197                           sdata[tx * nDIM_Y] += sdata[i + tx * nDIM_Y];
198                       }
199                   }
200               }
201
202               if (ty == 0 && ind < m)
203               {
204                   df[ind*incf] = alpha*sdata[tx * nDIM_Y] + beta*df[ind*incf];
205               }
206
207               __syncthreads();
```

```
208         }
209
210         if (ind < m) OffSetH = 0;
211
212         ind += nDIM_X;
213     }
214 }
215
216 ////////////////////////////////////////////////////////////////////////////////
    ////
217 /*
218     Operação H' x g (transposta).
219 */
220 __global__ void Symmetric_Trans_v2_2(
221     int m, int n, int Ne, int S, float alpha,
222     const float *__restrict__ dH, int lda,
223     const float *__restrict__ dg, int incg, float beta,
224     float       *__restrict__ df, int incf)
225 {
226     if (m <= 0 || n <= 0) return;
227
228     int num_threads = blockDim.x * blockDim.y * blockDim.z;
229
230     // need to launch exactly the same number of threads as template parameters
        indicate
231     if (tDIM_X * tDIM_Y != num_threads) return;
232
233     int thread_id = threadIdx.x + threadIdx.y * blockDim.x;
234
235     // threads are all configurated locally
236     int tx = thread_id % tDIM_X;
237     int ty = thread_id / tDIM_X;
238
239     __shared__ float sdata[tDIM_X * tDIM_Y];
240
241     float res;
242     int mfull = (m / tDIM_X) * tDIM_X;
243
244     int start = blockIdx.x * tTILE_SIZE + ty;
245     int iters;
246
247     int offsetH = 0;
248
249     //#define usefixedcondition
250 #ifdef usefixedcondition
251     /*fixed condition*/
252     iters = tTILE_SIZE / tDIM_Y;
253 #else
254     /* flexible condition based on global n (has drops when size is roughly bigger
        than nTILE_SIZE)*/
255     //int iters = magma_ceildiv(min(n,nTILE_SIZE), nDIM_Y);
256
257     /* flexible condition based on my local nloc=ed-st*/
258     int st = blockIdx.x * tTILE_SIZE;
259     //int ed = magma_ceildiv( min(n, st + nTILE_SIZE), nDIM_Y ) * nDIM_Y;
260     int ed = min(st + tTILE_SIZE, ((n + tDIM_Y - 1) / tDIM_Y) * tDIM_Y);
261     iters = (ed - st) / tDIM_Y;
262 #endif
263
264     if (tx < m) offsetH += tx;
265
266     for (int i = 0; i < iters; i++)// at 2Gflops/ overhead
267                                    //for (int col=start; col <
                                       (blockIdx.x+1)*nTILE_SIZE; col += nDIM_Y)// at
                                       least 3Gflop/s overhead
268     {
269         int indH = 0;
270         int col = start + i * tDIM_Y;
271
272         if (col < n) offsetH += col*lda;
273
274         res = 0.0f;
275
```

```
276           // partial sums
277           if (col < n)
278           {
279               for (int i = 0; i < mfull; i += nDIM_X) {
280                   indH = GetElementIndex(i + offsetH, n, Ne, S);
281                   res += dH[indH] * dg[(tx + i)*incg];
282               }
283               if (tx + mfull < m) {
284                   //res += dH[GetElementIndex(mfull + offsetH, m, Ne, S)] * dg[(tx +
                      mfull)*incg];
285
286                   indH = GetElementIndex(mfull + offsetH, n, Ne, S);
287
288                   res += dH[indH] * dg[(tx + mfull)*incg];
289               }
290           }
291           sdata[tx + ty * tDIM_X] = res;
292
293           // tree reduction of partial sums,
294           // from nDIM_X sums to ... 128 to 64 to 32 ... to 1 sum in sdata[0]
295           if (tDIM_X > 16)
296           {
297               magma_sum_reduce(tDIM_X, tx, sdata + ty * tDIM_X);
298           }
299           else
300           {
301               __syncthreads();
302
303               if (tx == 0 && col < n)
304               {
305                   for (int i = 1; i < m && i < tDIM_X; i++)
306                   {
307                       sdata[0 + ty * tDIM_X] += sdata[i + ty * tDIM_X];
308                   }
309               }
310               __syncthreads();
311           }
312
313           if (tx == 0 && col < n) {
314               df[col*incf] = alpha*sdata[0 + ty * tDIM_X] + beta*df[col*incf];
315           }
316
317           __syncthreads();
318
319           if (col < n)  offsetH -= col * lda;
320       }
321   }
322
323   ////////////////////////////////////////////////////////////////////////////////
      ////
324   /*
325    *  Função de integração MatLab C/C++/CUDA.
326    *
327    *  Esta função deve verificar os parâmetros de entrada e saída,
328    *  tratar os dados recebidos e preparar os dados para devolução ao
329    *  MatLab.
330    *
331    *  Tratamento direto de ponteiros alocados em GPU pelo script MatLab.
332    *
333    *  Argumentos Entrada:
334    *           argin[0] --> matriz H
335    *           argin[1] --> vetor g
336    *           argin[2] --> vetor f
337    *           argin[3] --> Total de linhas da matriz H (N)
338    *           argin[4] --> Total de colunas da matriz H (M)
339    *           argin[5] --> Nro elementos (Ne)
340    *           argin[6] --> Nro amostras (S)
341    *           argin[7] --> Calcular H transposta (0 - Normnal, 1 - Transposta);
342    */
343
344   void mexFunction(int nargout, mxArray *argout[], int nargin, mxArray *argin[])
345   {
346       mxGPUArray const *H;
```

```
347       mxGPUArray const *g;
348       mxGPUArray const *f;
349       cudaError_t e;
350
351       float const *d_H;
352       float const *d_g;
353       float *d_f;
354
355       int N, M, Ne, S, Tr;
356
357       if (nargin != 8)
358           mexErrMsgTxt("Invaid number of input arguments");
359
360       if (nargout != 1)
361           mexErrMsgTxt("Invalid number of outputs");
362
363       /* Initialize the MathWorks GPU API. */
364       mxInitGPU();
365
366       H = mxGPUCreateFromMxArray(argin[0]);
367       e = cudaGetLastError();
368       if (e != cudaSuccess) {
369           mexErrMsgTxt("Erro: Alocação matriz H.");
370       }
371       if (mxGPUGetClassID(H) != mxSINGLE_CLASS)
372           mexErrMsgTxt("The matrix H  must be single (float)");
373
374       g = mxGPUCreateFromMxArray(argin[1]);
375       e = cudaGetLastError();
376       if (e != cudaSuccess) {
377           mexErrMsgTxt("Erro: Alocação vertor g.");
378       }
379       if (mxGPUGetClassID(g) != mxSINGLE_CLASS)
380           mexErrMsgTxt("The vector g must be single (float)");
381
382       f = mxGPUCreateFromMxArray(argin[2]);
383       e = cudaGetLastError();
384       if (e != cudaSuccess) {
385           mexErrMsgTxt("Erro: Alocação vertor f.");
386       }
387       if (mxGPUGetClassID(f) != mxSINGLE_CLASS)
388           mexErrMsgTxt("The vector f must be single (float)");
389
390       d_H = (float const *)(mxGPUGetDataReadOnly(H));
391       e = cudaGetLastError();
392       if (e != cudaSuccess) {
393           mexErrMsgTxt("Erro: Alocação matriz d_H");
394       }
395
396       d_g = (float const *)(mxGPUGetDataReadOnly(g));
397       e = cudaGetLastError();
398       if (e != cudaSuccess) {
399           mexErrMsgTxt("Erro: Alocação vetor d_g");
400       }
401
402       d_f = (float *)(mxGPUGetDataReadOnly(f));
403       e = cudaGetLastError();
404       if (e != cudaSuccess) {
405           mexErrMsgTxt("Erro: Alocação vetor d_f");
406       }
407
408       N = mxGetScalar(argin[3]);
409       M = mxGetScalar(argin[4]);
410       Ne = mxGetScalar(argin[5]);
411       S  = mxGetScalar(argin[6]);
412       Tr = mxGetScalar(argin[7]);
413
414
415
416       if (Tr == 0)
417       {
418           dim3 dim_grid( ceil((float)N / nTILE_SIZE) );
419           dim3 dim_block(nDIM_X, nDIM_Y);
```

```
420
421         Symmetric_v2_2 << < dim_grid, dim_block >> >(N, M, Ne, S, 1, d_H, N, d_g, 1,
            0, d_f, 1);
422     }
423     else
424     {
425         dim3 dim_grid( ceil((float)M / tTILE_SIZE), 1);
426         dim3 dim_block(tDIM_X, tDIM_Y);
427
428         Symmetric_Trans_v2_2 << < dim_grid, dim_block >> >(N, M, Ne, S, 1, d_H, N,
            d_g, 1, 0, d_f, 1);
429     }
430
431     e = cudaGetLastError();
432     if (e != cudaSuccess) {
433         mexErrMsgTxt("Erro: Chamada Symmetric_MxV");
434     }
435
436     e = cudaThreadSynchronize();
437     if (e != cudaSuccess) {
438         // mexErrMsgTxt("Erro: cudaThreadSynchronize");
439          mexErrMsgTxt(cudaGetErrorString(e));
440     }
441
442     argout[0] = mxGPUCreateMxArrayOnGPU(f);
443     e = cudaGetLastError();
444     if (e != cudaSuccess) {
445         mexErrMsgTxt("Erro: mxGPUCreateMxArrayOnGPU(f)");
446     }
447
448     mxGPUDestroyGPUArray(H);
449     e = cudaGetLastError();
450     if (e != cudaSuccess) {
451         mexErrMsgTxt("Erro: mxGPUDestroyGPUArray(H)");
452     }
453
454     mxGPUDestroyGPUArray(g);
455     e = cudaGetLastError();
456     if (e != cudaSuccess) {
457         mexErrMsgTxt("Erro: mxGPUDestroyGPUArray(g)");
458     }
459
460     mxGPUDestroyGPUArray(f);
461     e = cudaGetLastError();
462     if (e != cudaSuccess) {
463         mexErrMsgTxt("Erro: mxGPUDestroyGPUArray(f)");
464     }
465 }
466
467
```