OTIMIZAÇÃO DE RESTAURAÇÃO DE IMAGENS EMPREGANDO O FUNCIONAL DE TIKHONOV PARALELIZADO COM CUDA APLICADO À MICROSCOPIA DE FORÇA ATÔMICA

Klaus Natorf Quelhas

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Ricardo Cordeiro de Farias
Geraldo Antônio Guerrera Cidade

Rio de Janeiro
Março de 2015

OTIMIZAÇÃO DE RESTAURAÇÃO DE IMAGENS EMPREGANDO O FUNCIONAL DE TIKHONOV PARALELIZADO COM CUDA APLICADO À MICROSCOPIA DE FORÇA ATÔMICA

Klaus Natorf Quelhas

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

____

Prof. Ricardo Cordeiro de Farias, Ph.D.

____

Prof. Felipe Maia Galvão França, Ph.D.

____

Prof. Cristiana Barbosa Bentes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2015

Quelhas, Klaus Natorf

Otimização de Restauração de Imagens Empregando o Funcional de Tikhonov Paralelizado com Cuda Aplicado à Microscopia de Força Atômica/Klaus Natorf Quelhas. – Rio de Janeiro: UFRJ/COPPE, 2015.

XIV, 101 p.: il.; $29, 7$cm.

Orientadores: Ricardo Cordeiro de Farias
Geraldo Antônio Guerrera Cidade

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2015.

Referências Bibliográficas: p. 74 – 77.

1. image restoration. 2. Tkhonov's functional. 3. GPU. 4. CUDA. I. Farias, Ricardo Cordeiro de *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À meus pais, familiares e amigos, que acreditaram no sucesso desta jornada e me apoiaram incondicionalmente.

À Bárbara Helena Gomes, cujo amor, apoio e compreensão foram essenciais ao longo deste ciclo.

À memória de Geraldo Antônio Guerrera Cidade.

Agradecimentos

Agradeço a meus orientadores Ricardo Cordeiro de Farias e Geraldo Antônio Guerrera Cidade (in memoriam). Ao primeiro, pela oportunidade de realizar este curso de mestrado em uma instituição de grande porte que é a COPPE/UFRJ, bem como pelos ensinamentos e apoio ao longo deste ciclo, o que permitiu o desenvolvimento deste trabalho com sucesso. Ao segundo, por ter me apresentado os princípios de uma área completamente nova para mim, o que permitiu a enorme satisfação de desenvolver um projeto que integrasse, de maneira prática, áreas de conhecimento distintas. A ambos, pela compreensão, paciência e sabedoria compartilhada ao longo do desenvolvimento deste trabalho.

Aos professores da UERJ Cristiana Barbosa Bentes, por ter me apresentado esta oportunidade ainda durante a graduação, por ter me recomendado durante o processo seletivo e por tomar parte da banca examinadora deste trabalho, e João Araújo Ribeiro, também pela honrosa recomendação.

Aos membros da banca examinadora, por se disponibilizarem a avaliar a qualidade desta dissertação.

Agradeço também a Caio Bulgarelli, pela auxílio prestado durante os estágios iniciais da pesquisa e desenvolvimento, e a Augusto Garcia Almeida, por fornecer os códigos-fonte essenciais ao desenvolvimento do projeto.

Aos demais professores do Laboratório de Computação Gráfica (LCG), em especial a Ricardo Marroquim, Cláudio Esperança e Antônio Oliveira, cada um responsável por ensinar diferentes ferramentas que, em conjunto, permitiram o desenvolvimento deste trabalho. Aos meus colegas de curso, especialmente Luciano Viana de Paula, Bryan Hall e Leonardo Carvalho, pelo auxílio prestado em diferentes oportunidades durante o curso. Finalmente, a meus colegas da divisão de Metrologia Térmica do Inmetro, cujo apoio foi fundamental para que meus objetivos fossem alcançados.

Muito obrigado a todos.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

OTIMIZAÇÃO DE RESTAURAÇÃO DE IMAGENS EMPREGANDO O FUNCIONAL DE TIKHONOV PARALELIZADO COM CUDA APLICADO À MICROSCOPIA DE FORÇA ATÔMICA

Klaus Natorf Quelhas

Março/2015

Orientadores: Ricardo Cordeiro de Farias
Geraldo Antônio Guerrera Cidade

Programa: Engenharia de Sistemas e Computação

A técnica de Microscopia de Força Atômica permite a aquisição de imagens em escala nanométrica de praticamente qualquer superfície não-condutora ou biológica, ao contrário de outras técnicas pertencentes à família de Microscopia de Varredura por Sonda, que somente permitem a análise de amostras condutoras. Contudo, dependendo das dimensões da amostra a imagem obtida está sujeita a ser degradada por ruído externo (elétrico e mecânico), levando a baixas relações Sinal/Ruído, além de borramento devido à geometria da ponteira de medição, o que torna necessário o uso de técnicas de restauração de imagens para a correta visualização das amostras.

Esta dissertação apresenta uma proposta de otimização do algoritmo de restauração de imagens de Microscopia de Força Atômica baseado no funcional de regularização de Tikhonov, bem como uma proposta de paralelização da execução deste algoritmo através do uso de *Graphics Processing Units* (GPU) e da arquitetura *Compute Unified Device Architecture* (CUDA), comparando o desempenho destes com os obtidos usando abordagens anteriores. Neste trabalho também é apresentada uma ferramenta para visualização de imagens a partir dos dados de medições do microscópio, empregando abordagens serial e paralela.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

OPTIMIZING IMAGE RESTORATION USING THE TIKHONOV FUNCTIONAL PARALLELIZED IN CUDA APPLIED TO ATOMIC FORCE MICROSCOPY

Klaus Natorf Quelhas

March/2015

Advisors: Ricardo Cordeiro de Farias
Geraldo Antônio Guerrera Cidade

Department: Systems Engineering and Computer Science

The Atomic Force Microscopy technique allows the acquisition of nanometric scaled images of virtually any non-conductive or biological surface, in opposite to other techniques that belongs to the Scanning Probe Microscopy family, which allows only the analysis of conductive samples. However, depending on the dimensions of the sample, the resulting image maybe degraded due to external electronic or mechanical noises, leading to low Signal/Noise ratios. In addition there maybe blurring caused by the microscope's tip geometry, which makes necessary the use of image restoration techniques in order to correctly represent the sample.

This dissertation proposes an optimization of the Atomic Force Microscopy image restoration algorithm based on the Tikhonov's regularization functional, as well as a parallelization of this algorithm through the use of *Graphics Processing Units* (GPU) and *Compute Unified Device Architecture* (CUDA). We compare our performances to the ones obtained by previous approaches. In this work it is also presented a tool for display of images from the microscope's measurement data, using serial and parallel approaches.

Sumário

Lista de Figuras x

Lista de Tabelas xiii

Lista de Algoritmos xiv

1 Introdução 1

1.1 Trabalhos relacionados 2
1.2 Objetivos 3
1.3 Estrutura da dissertação 4

2 Microscopia de Força Atômica 5

2.1 Princípio de funcionamento 6
2.2 Modos de operação 7
2.2.1 Modo contato 8
2.2.2 Modo não-contato 8
2.2.3 Modo intermitente (*Tapping*) 8
2.3 Interação ponteira-amostra 9
2.4 Varredura 11
2.5 Aquisição de dados 14
2.6 Imagens obtidas 15

3 Computação paralela usando a arquitetura CUDA 17

3.1 Princípios básicos 19
3.2 Estruturas de blocos e *threads* 21
3.3 Hierarquia de memória e tráfego de dados 21
3.4 Otimizando o desempenho 23
3.4.1 Maximizando a utilização da GPU 23
3.4.2 Maximizando o acesso à memória 24
3.4.3 Maximizando o uso de instruções 24

4 Ferramenta para exibição de imagens de microscopia de força atômica 25
4.1 Não-sincronismo de dados . . . 26
4.2 Aquisição de dados e obtenção de imagens . . . 27
4.3 Visualização em 2D e 3D . . . 29
4.3.1 Visualização em escala real . . . 31
4.4 Paralelização utilizando a arquitetura CUDA . . . 32
4.5 Integração com o sistema de monitoramento e controle . . . 35

5 Restauração de imagens de microscopia de força atômica 37
5.1 Imagens digitais monocromáticas . . . 37
5.2 Modelo de degradação de imagens de microscopia de força atômica . . . 38
5.2.1 Convolução ponteira-amostra . . . 39
5.2.2 Convolução discreta . . . 40
5.2.3 Solução do problema inverso . . . 40
5.3 Restauração de imagens pelo método de regularização do funcional de Tikhonov . . . 43
5.4 Algoritmo de restauração . . . 45
5.4.1 Algoritmo serial . . . 47
5.4.2 Algoritmo paralelo . . . 49

6 Resultados e discussão 52
6.1 Ferramenta de exibição de imagens . . . 53
6.2 Algoritmos de restauração de imagens . . . 57
6.2.1 Desempenho . . . 58
6.2.2 Qualidade da restauração . . . 61
6.2.3 Efeitos do operador de borramento nas bordas de uma imagem . . . 63
6.2.4 Efeitos do dimensionamento do operador de borramento . . . 66

7 Conclusões 70
7.1 Trabalhos futuros . . . 72

Referências Bibliográficas 74

A Publicações 78

B Implementação dos algoritmos desenvolvidos 84
B.1 Restauração paralela . . . 84
B.2 Cabeçalho da classe *DBufferAFM* . . . 95
B.3 Cabeçalho da classe *GLAFMDisplay* . . . 100

Lista de Figuras

2.1 Técnicas de Microscopia de Varredura por Sonda. 5
2.2 Princípio de funcionamento de um Microscópio de Força Atômica. 6
2.3 Detalhes de um *cantilever* empregado em um Microscópio de Força Atômica. 7
2.4 Forças resultantes da interação entre a ponteira e a amostra. 8
2.5 Interações interatômicas entre a ponteira de um AFM e a amostra. 9
2.6 Efeitos da geometria da ponteira sobre as imagens obtidas. 10
2.7 Exemplo de borramento em uma imagem obtida de uma grade de teste através do uso de uma ponteira piramidal. 10
2.8 Exemplo do efeito de afastamento da ponteira (*parachuting*). 10
2.9 Exemplo de obtenção de duas imagens de 5x5 *pixels*, através do controle de varredura da plataforma piezoelétrica. 11
2.10 Exemplo de não-linearidade da cerâmica piezoelétrica e o efeito desta sobre imagens adquiridas antes e depois da correção. 12
2.11 Exemplo de forma de onda aplicada para correção da não-linearidade da plataforma piezoelétrica. 13
2.12 Composição de uma imagem a partir de dados de varreduras não-linearizadas em sentidos opostos. 14
2.13 Exemplo de conjuntos de aquisição de dados de imagens. 15

3.1 Evolução das taxas de processamento de CPUs e GPUs. 18
3.2 Execução de um programa CUDA em GPUs com diferentes números de SMs. 20
3.3 Distribuição de um *grid* em blocos e *threads*. 21
3.4 Hierarquia de memórias em CUDA. 22

4.1 Microscópio de Força Atômica em uso no IBCCF. 25
4.2 Funcionamento do *buffer* duplo. 27
4.3 Composição das imagens conforme diferentes modos. 28
4.4 Exemplo de visualização em 3D de uma imagem de AFM. 30
4.5 Exemplo de visualização em 2D de uma imagem de AFM. 30

5.1 Exemplo de imagem digital. 37
5.2 Modelo de degradação de imagens de microscopia de força atômica. . . . 38
5.3 Operador de borramento do tipo Gaussiano e sua representação no plano de imagem. 39
5.4 Representação matricial do operador de borramento. 40
5.5 Processo iterativo com o uso de uma referência fixa. 42
5.6 Processo iterativo com o uso da técnica de realimentação. 42
5.7 Área de atuação da matriz de borramento b centrada em um ponto (X) próximo às bordas de uma imagem hipotética. 44

6.1 Exemplo de visualização 3D de uma imagem com 256x256 *pixels* de uma grade de teste obtida com o uso de um AFM. 53
6.2 Visualização 3D de um conjunto de dados representando uma imagem artificialmente criada com 256x256 *pixels*. 54
6.3 Tempos de execução da ferramenta de exibição para imagens de 128x128 e 256x256 *pixels*. 55
6.4 Tempos de execução da ferramenta de exibição para imagens de 512x512 e 1024x1024 *pixels*. 55
6.5 Ganhos de desempenho (*speedups*) decorrentes da paralelização da ferramenta de exibição. 56
6.6 Razão percentual do tempo consumido com tráfego de dados nas abordagens paralelas. 56
6.7 Imagem de teste. 57
6.8 Imagens borradas utilizadas na avaliação dos algoritmos. 58
6.9 Resultado da restauração de uma imagem de 256x256 *pixels*. 59
6.10 Tempos de execução da restauração de imagens de 128x128 e 256x256 *pixels*. 60
6.11 Tempos de execução da restauração de imagens de 512x512 e 1024x1024 *pixels*. 60
6.12 Ganhos de desempenho (*speedups*) decorrentes da paralelização da restauração de imagens. 60
6.13 Imagens restauradas com o algoritmo paralelo implementado. 62
6.14 Resultado da restauração de um texto ilegível. 64
6.15 Efeitos de diferentes estratégias de normalização para o uso do operador de borramento. 65
6.16 Curva do tipo Gaussiana, com $\sigma = 5$, e representações triangulares de ponteiras com diferentes valores de largura L. 66
6.17 Diagrama descritivo de uma ponteira de AFM comercial. 67

6.18 Perfis de intensidade na direção horizontal (linha tracejada) de imagens restauradas com um matriz de borramento b com dimensões 21x21, para diferentes valores de variância. 68

Lista de Tabelas

6.1 Parâmetros de restauração empregados 62
6.2 Resultados da análise da qualidade das restaurações 63
6.3 Resultados da análise da qualidade das restaurações de um texto ilegível . 64
6.4 Resultados da análise da qualidade das restaurações da imagem D13v20 . 65

Lista de Algoritmos

4.1 Composição paralela das imagens de erro e topografia no modo *ida* 34
4.2 Composição paralela da imagem final 34
4.3 Construção paralela dos *arrays* de vértices e cores 35
5.1 Restauração serial de imagens por regularização de Tikhonov 48
5.2 Implementação simplificada do *kernel* `conv_dif` 49
5.3 Implementação simplificada do *kernel* `aplica_deltaX` 50
5.4 Restauração paralela de imagens por regularização de Tikhonov 51

Capítulo 1

Introdução

A técnica de Microscopia de Força Atômica (*Atomic Force Microscopy* - AFM) [1], também conhecida como Microscopia de Varredura por Força (*Scanning Force Microscopy* - SFM), permite a obtenção, em escala nanométrica, de superfícies não-condutoras ou biológicas. Esta característica proporciona ao AFM um amplo espectro de atuação em áreas estratégicas como a de materiais e biologia.

As imagens obtidas com o uso de AFM estão sujeitas a interferências provenientes de fontes externas, de origem elétrica (controle e instrumentação) e/ou mecânica (sistema de varredura, vibração, entre outros), o que pode resultar em pobres relações Sinal/Ruído, além dos efeitos degenerativos provenientes da interação da ponteira de medição e a amostra, que causam uma suavização, ou borramento, na imagem, seja através das influências das forças de interação envolvidas ou como um efeito de sua geometria, em função das dimensões da área de varredura. Tais efeitos somados promovem uma degradação considerável nas imagens adquiridas, o que, em escala nanométrica, poderá ocultar detalhes significativos da estrutura da amostra. Para minimizá-los, é recomendável o emprego de técnicas de restauração de imagens apropriadas.

Neste sentido, diversas abordagens foram desenvolvidas e avaliadas, de forma a encontrar aquela que proporciona a melhor restauração de maneira mais eficiente, ou ainda de forma a realçar determinadas características das imagens, como bordas ou detalhes. Como exemplo de técnicas estudadas, pode-se citar a transformada rápida de Fourier (FFT) e a morfologia matemática. Tais abordagens, contudo, frequentemente acabam por penalizar determinadas características das imagens, ao suavizá-la em demasia ou ao ressaltar excessivamente o ruído.

Desta forma, a técnica de restauração de imagens baseada na regularização do funcional de Tikhonov [2, 3], de natureza iterativa, foi proposta como uma abordagem de solução de problema inverso para buscar a otimização do compromisso entre a fidelidade aos dados e a suavidade da solução encontrada, para diferentes relações Sinal/Ruído. Para tal, é proposto um funcional de regularização no qual a informação necessária é extraída dos dados disponíveis em iterações anteriores, cujos resultados apresentados se mostra-

ram satisfatórios. Contudo, a restauração de imagens com o uso desta técnica demanda um grande esforço computacional, o que leva a tempos de execução extremamente elevados.

Considerada como uma técnica de quatro dimensões (plano de varredura xy, altura z e tempo t, a Microscópia de Força Atômica de Alta Velocidade (*High-Speed Atomic Force Microscopy* - HSAFM) [4] representa uma evolução do AFM, sendo capaz de produzir uma grande quantidade de imagens por segundo, dependendo das dimensões do campo de varredura, o que a torna atraente para a visualização dinâmica de processos biológicos *in vitro*. Tendo em vista a crescente evolução tecnológica, tanto as taxas de aquisição de dados como as resoluções das imagens tendem a crescer significativamente. Desta forma, faz-se necessário o uso de algoritmos e técnicas computacionais capazes de acompanhar estas tendências, que permitam a visualização de superfícies de amostras em tempo real, tendo minimizados os efeitos deletérios da instrumentação (ruído aditivo e borramento).

1.1 Trabalhos relacionados

Diversos trabalhos foram desenvolvidos no sentido de aprimorar a qualidade das imagens de microscopia de força atômica através do uso do funcional de regularização de Tikhonov. Dentre eles destaca-se o trabalho de Cidade [5–7], que ao descrever as etapas de desenvolvimento e operação do protótipo de um AFM em sua tese de doutorado, propôs a utilização do funcional de regularização de Tikhonov, apresentando resultados satisfatórios. O algoritmo utilizado para a restauração de imagens serviu como percursor para o desenvolvimento dos demais trabalhos. O equipamento encontra-se em uso no Instituto de Biofísica Carlos Chagas Filho (IBCCF).

Em sua dissertação de mestrado, Furtado [8] propõe o uso da técnica de realimentação na restauração de imagens, obtendo bons resultados ao acelerar a convergência da solução, restaurando imagens em um número reduzido de iterações.

Levando em consideração o grande esforço computacional necessário para a restauração de imagens com o funcional de regularização de Tikhonov, Stutz [9] propôs em sua dissertação de mestrado um algoritmo paralelo utilizando a biblioteca *MPI*, apresentando os ganhos de desempenho obtidos com esta técnica. Em sua tese de doutorado [10], realizou um extenso estudo dos ganhos computacionais obtidos através da execução em paralelo do algoritmo de restauração utilizando também a biblioteca *MPI*. Para tal, o algoritmo original de restauração, de natureza serial, sofreu alterações que o tornaram computacionalmente mais eficiente, sendo também mais atrativo a ser executado em paralelo, mantendo resultados equivalentes em termos de qualidade de restauração. Realizou também um estudo qualitativo da restauração das imagens, através do uso de diferentes métricas de avaliação, propondo inclusive uma nova métrica. Além disso, propôs o uso de uma técnica, chamada α *ótimo*, como uma estratégia para a estimativa dinâmica

de um dos parâmetros do funcional de regularização ao longo da execução do algoritmo de restauração, de forma a acelerar a convergência da solução e minimizar o número de iterações.

Como uma alternativa paralela ao uso da biblioteca *MPI*, Almeida [11] propôs o uso de unidades de processamento gráfico (*Graphics Processing Unit* - GPU), através do uso da linguagem *shader*, para a implementação do algoritmo de restauração proposto por Stutz, constatando a inviabilidade desta abordagem em função da natureza do problema e da arquitetura computacional envolvida. Finalmente, apresenta o uso da arquitetura CUDA (*Compute Unified Device Architecture*) para a restauração de imagens com o uso do funcional de regularização de Tikhonov em sua tese de doutorado [12], bem como os ganhos de desempenho obtidos com esta abordagem. Para tal, usou como base o algoritmo serial apresentado por Stutz. Isso foi necessário, uma vez que o modelo de paralelismo empregado para a elaboração do algoritmo paralelo utilizando a biblioteca *MPI* não é adequado para ser executado em GPU, dadas as diferentes arquiteturas envolvidas.

Finalmente, Quelhas [13] propõe uma restruturação do algoritmo serial desenvolvido por Stutz, apresentando em seguida uma implementação paralela em GPU deste algoritmo utilizando a arquitetura CUDA, obtendo ganhos de desempenho consideráveis mesmo com imagens de pequenas dimensões, em comparação com abordagens anteriores.

1.2 Objetivos

Este trabalho tem por objetivo principal avaliar os algoritmos seriais de restauração de imagens por funcional de regularização de Tikhonov propostos por Cidade [6] e Stutz [10], além do algoritmo paralelo proposto por Almeida [12], em busca de oportunidades de otimização em termos de tempo de execução, de forma a permitir o emprego desta técnica em restaurações de imagens de microscopia de força atômica em tempo real, tornando assim possível o acompanhamento de processos dinâmicos com qualidade de imagem aprimorada. Para tal, propõe um novo algoritmo serial de restauração, seguido de sua implementação em paralelo utilizando a arquitetura CUDA, a partir das expressões matemáticas apresentadas nos trabalhos anteriores.

Como objetivo secundário, buscou-se desenvolver uma ferramenta para exibição de imagens de microscopia de força atômica a partir dos dados brutos fornecidos pelos sistema de aquisição de dados e controle do AFM, a partir do uso de CUDA e da biblioteca OpenGL, de maneira a minimizar o uso da Unidade Central de Processamento (*Central Processing Unit* - CPU), durante a execução do programa desenvolvido para a operação do AFM descrito por Cidade, e em uso no IBCCF. Esta ferramenta deve permitir a incorporação, no futuro, do algoritmo paralelo de restauração de imagens desenvolvido para este trabalho, de forma a prover um sistema único de tratamento e exibição de imagens de microscopia de força atômica.

1.3 Estrutura da dissertação

Esta dissertação é dividida em sete capítulos. O capítulo 1 apresenta uma breve introdução aos tema central do trabalho, apresentando trabalhos relacionados e soluções apresentadas anteriormente. O capítulo 2 apresentas os princípios da microscopia de força atômica, técnicas de operação e obtenção de imagens em escala nanométrica. O capítulo 3 apresenta os princípios da computação paralela utilizando a arquitetura CUDA, seu histórico, estruturas básicas e hierarquia de memória, bem como diretrizes de otimização de aplicações paralelas utilizando esta plataforma.

O capítulo 4 apresenta os princípios e as etapas do desenvolvimento de uma ferramenta de exibição de imagens de AFM, bem como sua versão paralela utilizando a arquitetura CUDA, usando diferentes estratégias de alocação de espaço de memória e tráfego de dados, apresentando os algoritmos paralelos implementados.

Os princípios envolvidos no processo de restauração de imagens de microscopia de força atômica são apresentados no capítulo 5, assim como uma avaliação aprofundada das expressões matemáticas envolvidas, de forma a apresentar um algoritmo serial otimizado que foi usado como base para a implementação paralela utilizando a arquitetura CUDA, cujo algoritmo também é apresentado neste capítulo.

No capítulo 6 são apresentados os resultados das avaliações da ferramenta de exibição de imagens, bem como do algoritmo de restauração, no que diz respeito ao desempenho das aplicações e à qualidade das imagens obtidas. Os resultados obtidos com os processos de restauração implementados são comparados com duas implementações de trabalhos anteriores, de forma a atestar a eficiências destas. Por fim, neste capítulo é proposta uma metodologia de dimensionamento do operador de borramento a partir de informações da ponteira do AFM e da superfície em análise. Finalmente, o capítulo 7 apresenta as conclusões do trabalho desenvolvido, bem como oportunidades de trabalhos futuros.

Capítulo 2

Microscopia de Força Atômica

A Microscopia de Força Atômica (AFM), classificada como uma técnica de Microscopia de Varredura por Força (SFM), surgiu como uma variação da Microscopia de Varredura por Tunelamento (*Scanning Tunneling Microscopy* - STM) [14, 15], desenvolvida em 1981; publicada em 1982 como a primeira de uma série de técnicas capazes de capturar imagens em nível atômico, rendeu a seus inventores o prêmio Nobel de física de 1986 [16]. Todas estas técnicas enquadram-se no conceito de Microscopia de Varredura por Sonda (*Scanning Probe Microscopy* - SPM) (figura 2.1). A técnica de AFM diferencia-se das demais modalidades de microscopia conhecidas devido à capacidade de gerar imagens tridimensionais de amostras não-condutoras (o que inclui amostras biológicas vivas) em escala nanométrica, o que a coloca em grande vantagem estratégica.

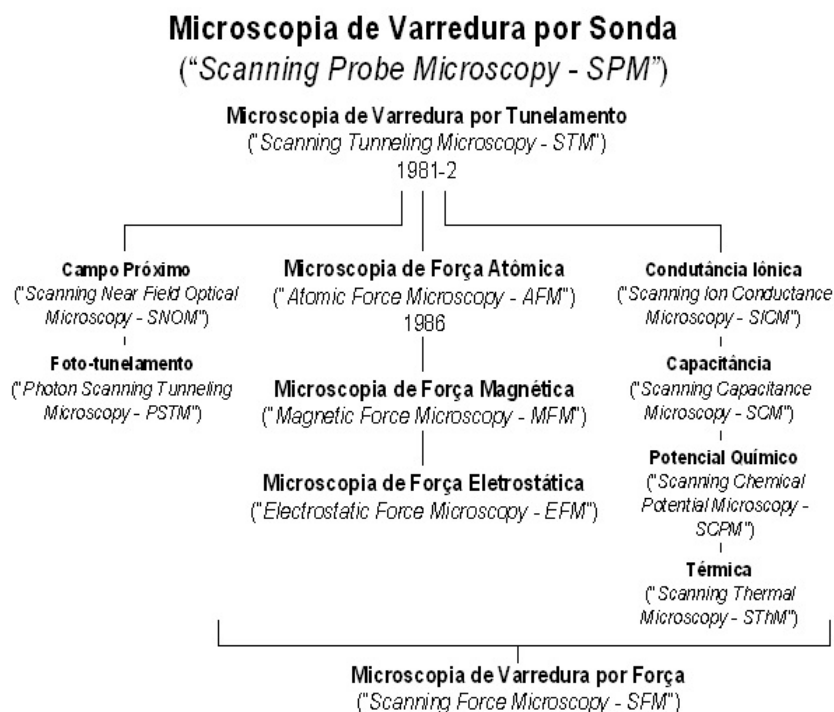

Figura 2.1: Técnicas de Microscopia de Varredura por Sonda. Cidade (2000) [6].

O que diferencia as técnicas de SPM é o método de sensoriamento empregado. Enquanto, por exemplo, o STM utiliza a captação de elétrons, por efeito de tunelamento, da superfície de uma amostra condutora sob a forma de uma corrente elétrica que decai exponencialmente em função da distância, a partir da aplicação de um campo elétrico, o AFM faz uso de um dispositivo sensor, que responde mecanicamente às variações da topografia da superfície da amostra em função das forças de interação (atração e/ou repulsão) resultantes, principalmente as de Van der Waals [17].

2.1 Princípio de funcionamento

A figura 2.2 descreve o princípio de funcionamento de um AFM, em que um feixe laser colimado incide sobre a extremidade de um braço de apoio, ou *cantilever*, com uma constante de mola específica k. Em sua extremidade encontra-se uma ponteira, ou *tip*, de formato piramidal com altura de alguns micrometros e uma extremidade com raio de algumas dezenas de nanometros (figura 2.3), que interage com a superfície da amostra, produzindo uma resposta mecânica que causa a sua deflexão. O feixe laser incidente é então refletido sobre um dispositivo fotodetetor, que responde eletricamente às variações topográficas da superfície da amostra em função da resposta mecânica do *cantilever*.

Para realizar a varredura da amostra, uma plataforma piezoelétrica (PP) construída a partir de atuadores piezoelétricos, cerâmicas especiais que atuam como um posicionadores de elevada sensibilidade e exatidão, move-se nas direções x, y e z. Na medida em que a amostra se movimenta, a resposta elétrica relativa à deflexão do *cantilever* sensibiliza um circuito de realimentação, que permite o reposicionamento da PP na direção z. Esta

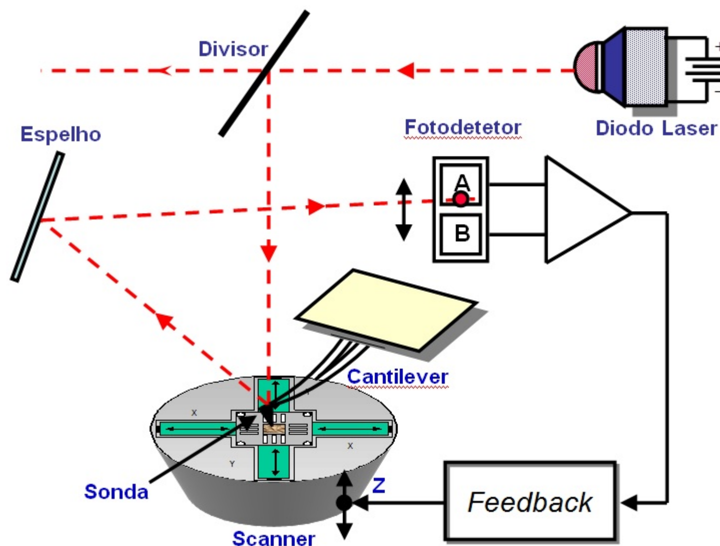

Figura 2.2: Princípio de funcionamento de um Microscópio de Força Atômica. Cidade (2007) [18].

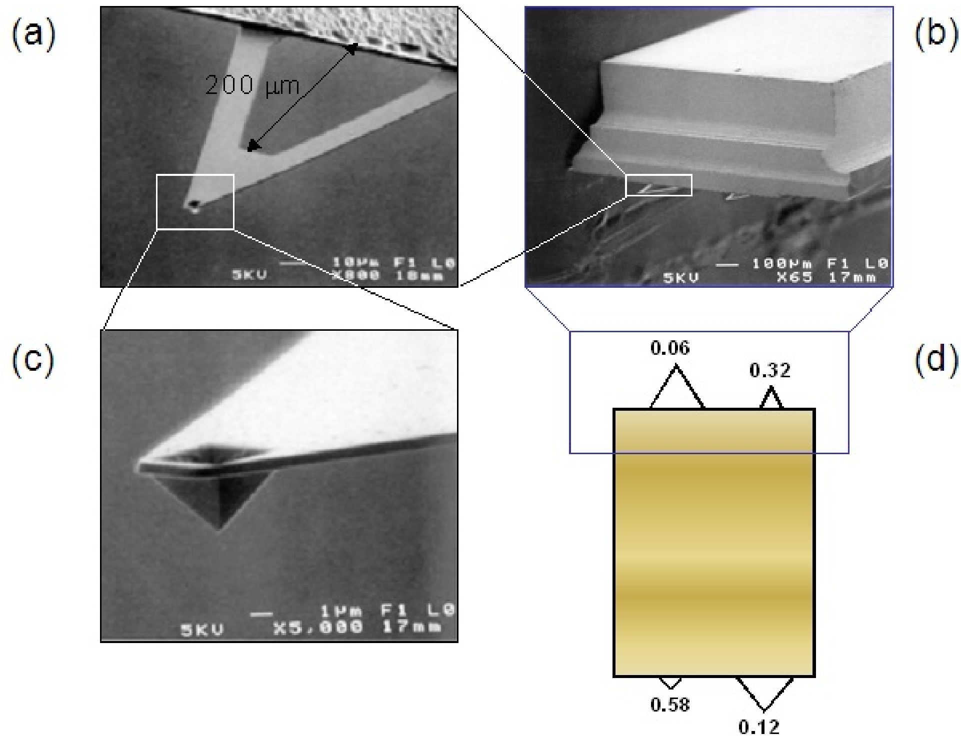

Figura 2.3: Detalhes de um *cantilever* empregado em um Microscópio de Força Atômica. Cidade (2000) [6].

configuração mantém o sistema em regime de *força constante*, que preserva a integridade de amostras frágeis contra eventuais danos. O sinal correspondente ao reposicionamento (ou correção) da plataforma piezoelétrica na direção z equivale ao relevo (ou topografia) da amostra. Em uma abordagem alternativa, a topografia da amostra pode ser representada somente pela resposta do fotodetetor, em um regime de *altura constante*, onde o correção da posição z da plataforma não é aplicada. Contudo, esta abordagem pode acarretar em danos à amostras mais frágeis, mas por outro lado é mais adequada para aplicações com altas taxas de aquisição de imagens (*fast scanning*), uma vez que a velocidade de varredura não é limitada pelo tempo de resposta do sistema de reposicionamento da plataforma no eixo z [19].

2.2 Modos de operação

A resposta do AFM depende da interação da ponteira com a superfície da amostra, gerando uma resposta mecânica que depende do relevo da superfície em estudo. A maneira como a ponteira interage com a amostra, ou seja, as forças envolvidas no processo de interação, depende do modo de operação do microscópio. A curva exibida na figura 2.4 descreve a força exercida sobre a ponteira em função da distância desta para a amostra, demarcando também as regiões de operação para cada modo.

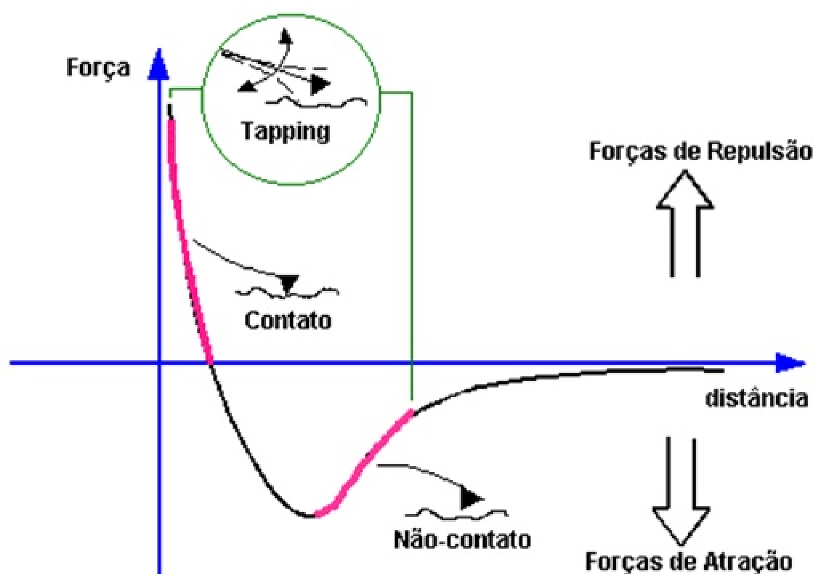

Figura 2.4: Forças resultantes da interação entre a ponteira e a amostra. Cidade (2000) [6].

2.2.1 Modo contato

Neste modo de operação a força resultante exercida sobre a ponteira é fortemente repulsiva, compreendendo predominantemente às forças de van der Waals. Cuidados devem ser tomados para evitar danos às amostras, com por exemplo a correção da posição z da PP, o que pode limitar as taxas de aquisição de imagens.

2.2.2 Modo não-contato

Neste modo de operação a ponteira é mantida a uma pequena distância da amostra, de forma que a força resultante é de natureza atrativa. Este modo é empregado de forma a preservar a superfície de amostras mais frágeis, contudo requer uma instrumentação com maior sensibilidade, em função das forças envolvidas apresentarem reduzidas intensidades.

2.2.3 Modo intermitente (*Tapping*)

Neste modo de operação, o *cantilever* oscila em uma de suas frequências de ressonância, reunindo as vantagens de ambos os modos anteriores ao interagir intermitentemente com a amostra. Como a interação ocorre por um breve período de tempo, possíveis danos a estruturas são minimizados, sem prejuízo da qualidade de imagem obtida através do modo contato. Além disso, distorções decorrentes das forças de adesão e torção da ponteira são evitadas.

2.3 Interação ponteira-amostra

As informações referentes ao relevo da amostra em análise são obtidas através da interpretação da interação entre a ponteira e a amostra. A deflexão do *cantilever* é resultante das forças atuantes entre a ponteira e amostra, que compreendem as forças de atração e/ou repulsão de van der Waals (figura 2.4), as forças de adesão de longo alcance resultantes da capilaridade induzida pela camada de hidratação em medições realizadas no ar, bem como as forças decorrentes da deflexão do *cantilever* [6].

Enquanto o STM possui resolução atômica, ou seja, é capaz de registrar a presença de um único átomo, o AFM não. Apesar de produzir imagens em escala atômica, ele o faz a partir da interação dos átomos que compõem a extremidade da ponteira com os átomos da superfície da amostra, de maneira que a imagem resultante é construída pela superposição das forças de vários átomos que interagem entre si (figura 2.5), o que leva a uma suavização, ou borramento, em escala atômica [19].

Bem acima da escala atômica, as imagens obtidas com o uso do AFM estão sujeitas a um segundo efeito de borramento, desta vez decorrente da geometria da ponteira. Se as dimensões desta forem comparáveis às do relevo em estudo (detalhes da estrutura), a imagem obtida apresentará um suavização resultante de uma convolução entre a ponteira e a amostra. A figura 2.6 ilustra o efeito de borramento devido às dimensões da ponteira, enquanto a figura 2.7 exibe uma imagem de uma grade de testes obtida com um AFM através do emprego de uma ponteira piramidal.

Em regime de *fast scanning*, as imagens adquiridas também estão sujeitas a um efeito de afastamento da ponteira, chamado *parachuting* [20], em razão de uma variação brusca no relevo, como um degrau ou inclinação acentuada, fazendo com que a ponteira se afaste demasiadamente da amostra, entrando novamente em contato com a amostra após um determinado intervalo de tempo. A principal indicação deste efeito são rastros encontrados na imagem (figura 2.8).

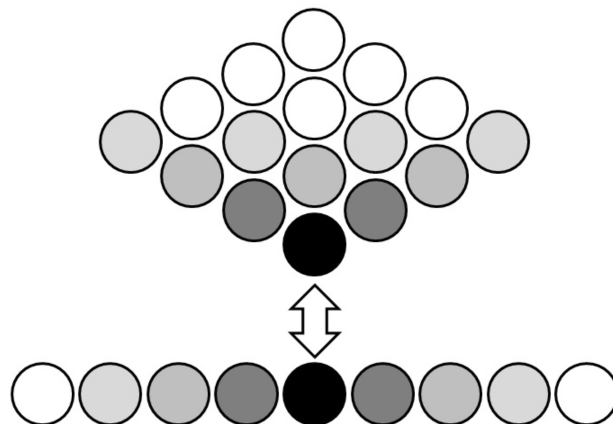

Figura 2.5: Interações interatômicas entre a ponteira de um AFM e a amostra. Thermomicroscopes (2000) [19].

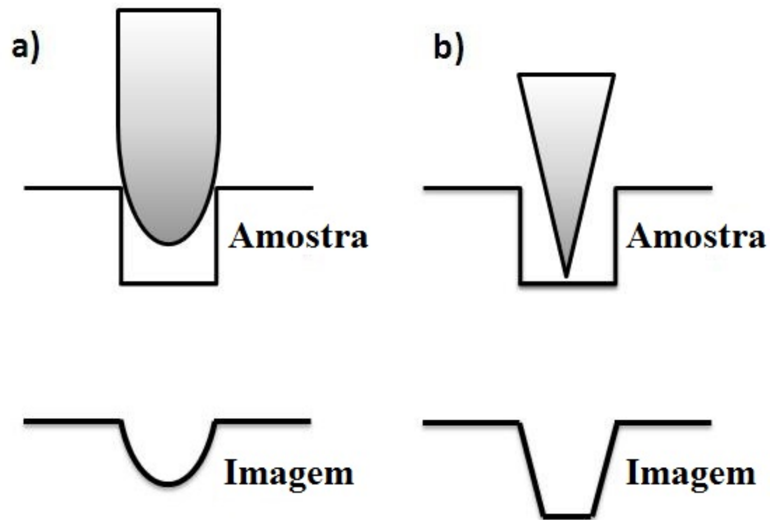

Figura 2.6: Efeitos da geometria da ponteira sobre as imagens obtidas.

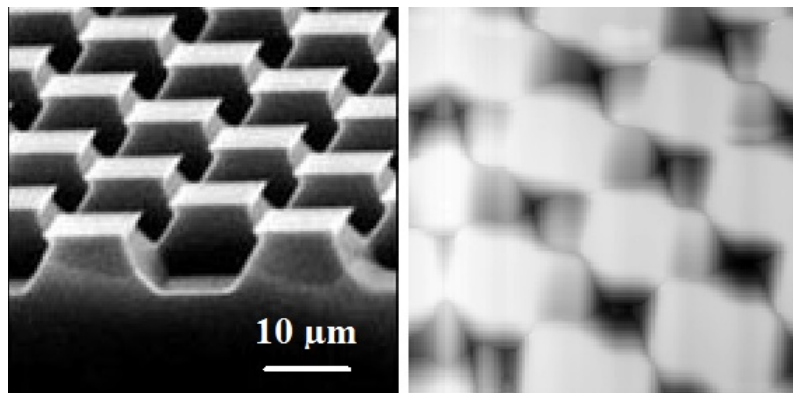

Figura 2.7: Exemplo de borramento (imagem à direita) em uma imagem obtida de uma grade de teste (imagem à esquerda) através do uso de uma ponteira piramidal.

Figura 2.8: Exemplo do efeito de afastamento da ponteira (*parachuting*).

2.4 Varredura

A obtenção de imagens de um AFM requer uma operação sincronizada dos sistemas de posicionamento x e y da PP e do sistema de amostragem dos sinais de resposta do fotodetetor e/ou correção da plataforma piezoelétrica na direção z, de maneira a obter uma imagem correspondente à área da superfície da amostra, descrita por uma matriz cujos elementos (*pixels*) representam o o sinal amostrado.

Para tal, um circuito aplica duas formas de onda distintas, na forma de tensões elétricas, que excitam e deslocam a PP nas direções x e y. Na medida em que a varredura é executada, os dados da imagem são amostrados em intervalos regulares, definidos como uma fração do período base de varredura. Em regime de *força constante*, os dados adquiridos são os relativos às posições z da PP, enquanto que em regime de *altura constante* referem-se às deflexões do *cantilever*, expressas em termos de resposta elétrica do fotodetetor.

A figura 2.9 exibe um exemplo aproximado de obtenção de duas imagens de um AFM com resolução de 5x5 *pixels* de através do controle de varredura da PP por meio da aplicação de tensões elétricas periódicas. É possível observar que a amostragem é realizada de maneira regular à medida que a varredura é executada continuamente no eixo x, enquanto que o deslocamento no eixo y ocorre somente durante a metade do período. Isso permite obter um conjunto de dados correspondentes a linhas e colunas da imagem desejada, que compreende metade dos dados amostrados. Isso ocorre pelo fato de que durante metade do período de varredura em x, a plataforma está sendo deslocada em na direção y, sem que a aquisição de medição de dados seja interrompida, o que pode levar a distorções nas imagens obtidas.

É importante observar que as técnicas de varredura e amostragem disponíveis não se

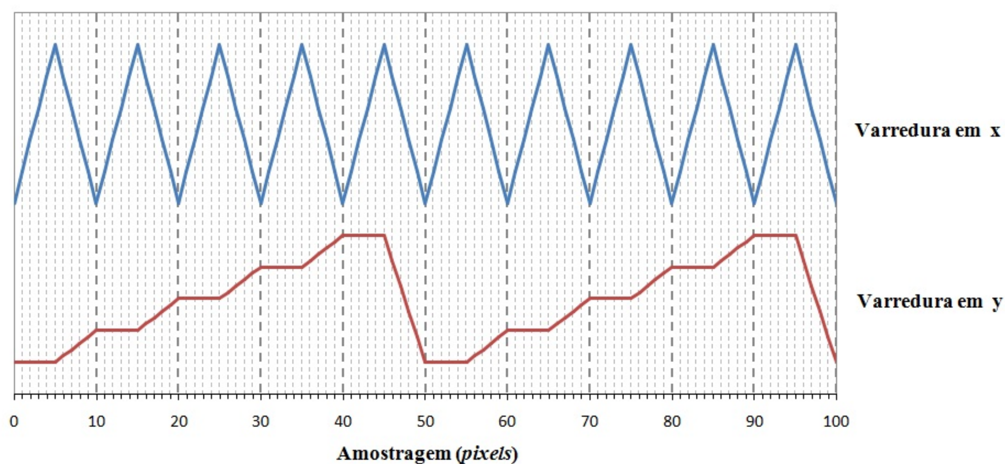

Figura 2.9: Exemplo de obtenção de duas imagens de 5x5 *pixels*, através do controle de varredura da plataforma piezoelétrica.

limitam ao exemplo descrito na figura 2.9, que tem por objetivo apenas apresentar uma descrição simplificada do processo de obtenção de imagens de um AFM. Aplicações reais devem levar em consideração diversos fatores de natureza elétrica, mecânica, bem como inerentes ao hardware e software de controle, tais como não-linearidade e histerese da cerâmica piezoelétrica da qual é composta a plataforma de varredura, bem como tempos de resposta do sistema de controle e efeitos mecânicos inerentes à interação da ponteira com a amostra, principalmente em regime de *fast scanning*, que acarretariam na necessidade do emprego de formas de onda mais complexas que a apresentada no exemplo. As figuras 2.10 e 2.11 exibem, respectivamente, um exemplo de não-linearidade da cerâmica piezoelétrica, sua correção e o efeito de ambas as curvas na aquisição de uma imagem de teste, bem como um exemplo de forma de onda aplicada à plataforma piezoelétrica de forma a corrigir o deslocamento não-linear.

Apesar de a aquisição de dados ser realizada de maneira regular e contínua durante a varredura, a forma de onda aplicada à PP pode vir a favorecer a utilização de dados amostrados em um dos sentidos de varredura (ida ou volta), ou ambos, de maneira a obter imagens mais estáveis e/ou fiéis ao relevo real da amostra. No exemplo apresentado na figura 2.9, é recomendável que os dados a serem empregados na construção das imagens sejam aqueles obtidos durante o semiciclo crescente da forma de onda aplicada ao eixo x, visto que neste intervalo a posição da plataforma é constante na direção y. Uma

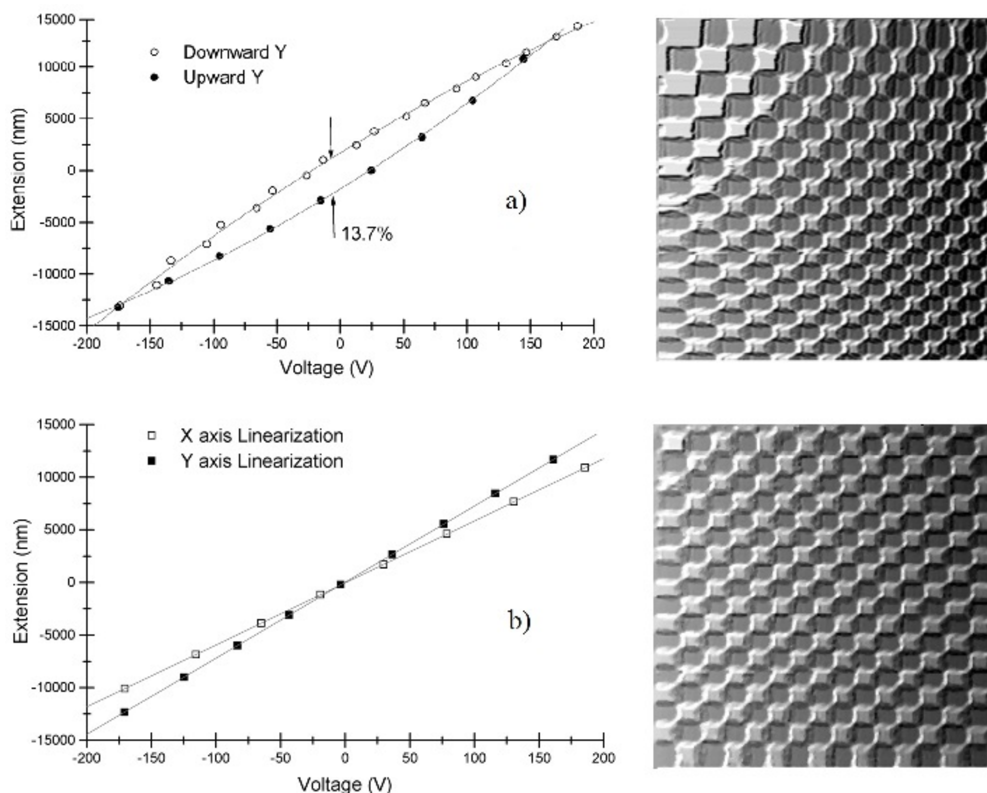

Figura 2.10: Exemplo de não-linearidade da cerâmica piezoelétrica e o efeito desta sobre imagens adquiridas a) antes e b) depois da correção. Cidade (2000) [6].

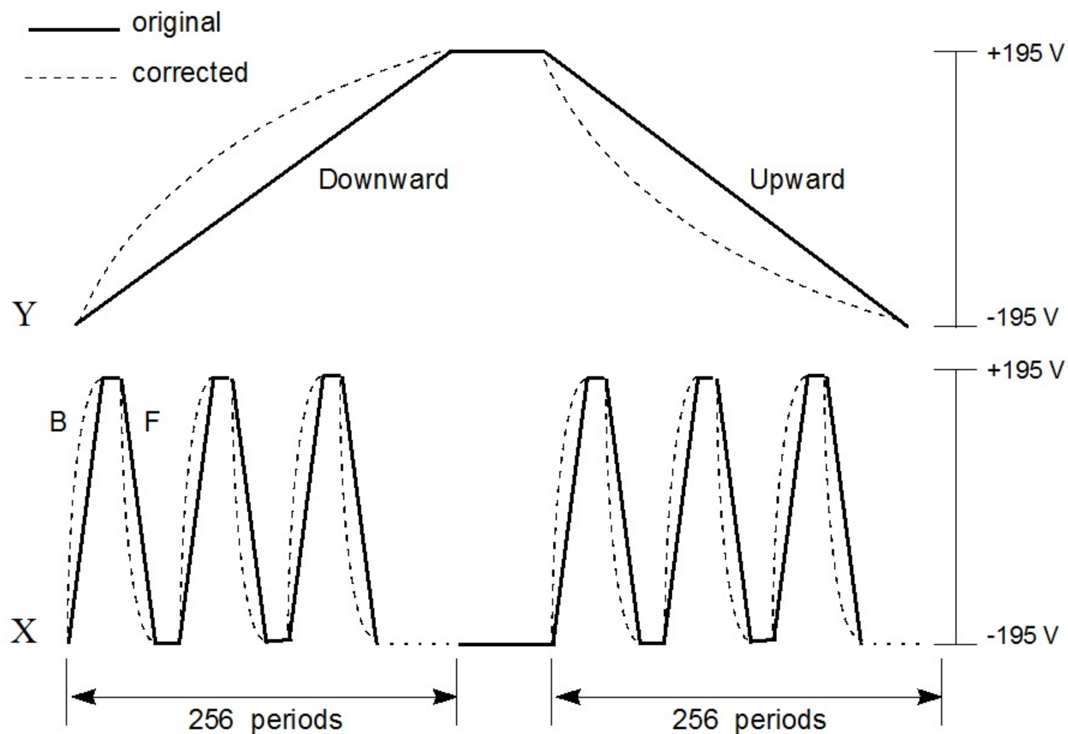

Figura 2.11: Exemplo de forma de onda aplicada para correção da não-linearidade da plataforma piezoelétrica. Cidade (2000) [6].

outra hipótese seria a aplicação da correção da não-linearidade em apenas um dos semiciclos da forma de onda, o que geraria uma imagem linearizada em um dos sentidos e uma distorcida em outro. Finalmente, como medida de redução dos ruídos de origem mecânica(vibrações) decorrentes de varreduras em altas velocidades, que prejudicam a qualidade da imagem amostrada, é possível empregar uma forma de onda tal que provoque um deslocamento amortecido (uma onda senoidal, por exemplo) da PP em um dos semiciclos de varredura. Estes fatores devem ser levados em consideração ao selecionar os dados úteis à construção de imagens. A figura 2.12 exibe imagens adquiridas em diferentes sentidos de varredura, de forma a ilustrar as possíveis diferenças entre elas.

Em outras aplicações, pode ser conveniente utilizar os dados de ambos os sentidos de varredura (ida e volta) de forma a eliminar efeitos decorrentes do deslocamento da ponteira sobre a amostra, ou vice-versa, como torção da ponteira ou *parachuting*, desde que a não-linearidade do deslocamento da PP seja corrigida em ambos os sentidos. Neste caso, a imagem resultante é uma combinação das imagens obtidas com os dados amostrados em ambos os sentidos de varredura. Caso as formas de onda aplicada em ambos os sentidos não sejam corrigidas quanto à não-linearidade, a combinação de ambas resultará em uma imagem distorcida, em um efeito chamado *efeito persiana*, decorrente da sobreposição de imagens desalinhadas entre si. A figura 2.12 exibe o resultado da combinação de imagens desalinhadas.

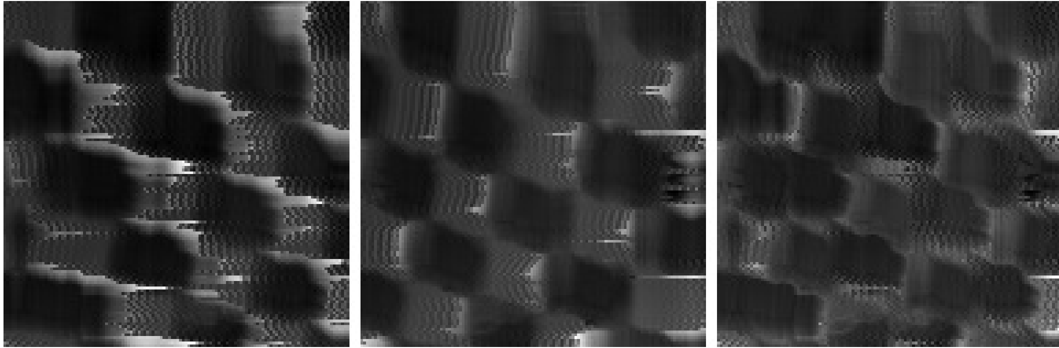

Figura 2.12: Composição de uma imagem (à direita) a partir de dados de varreduras não-linearizadas em sentidos opostos (à esquerda e ao centro).

2.5 Aquisição de dados

O AFM em uso no IBCCF emprega um sistema de aquisição de dados implementado para permitir a aquisição de imagens a altas taxas, consistindo em uma placa de aquisição de dados de quatro canais, instalada no barramento PCI Express, e um módulo baseado em um microcontrolador, que realiza a varredura e aquisição de dados. Através do uso de acesso direto à memória (*direct memory access* - DMA), é possível desonerar a CPU do custo de gerenciar a transferência de dados a taxas elevadas e permitir o emprego da mesma na construção e exibição das imagens em tempo real, cujo custo computacional é elevado.

O conjunto de dados transferidos corresponde aos sentidos de varredura em x e y da PP, bem como o sinal do fotodetetor decorrente da deflexão do *cantilever* e a correção da plataforma na direção z. Os sinais correspondentes aos sentidos de varredura em x e y são medidos em termos de tensão elétrica contínua representando os estados lógicos 0 ou *falso* (0 V) e 1 ou *verdadeiro* (5 V), e permitem a identificação dos inícios de cada linha e de uma nova imagem, respectivamente. Os sinais do fotodetetor e da correção da PP na direção z são também descritos em termos de tensão elétrica contínua e permitem a construção da imagem correspondente ao relevo da amostra, uma vez conhecidas as sensibilidades do fotodetetor e da plataforma piezoelétrica à variações nas tensões medidas. A varredura e amostragem são executadas de maneira contínua, e regularmente conjuntos de dados são transferidos para a memória.

O sincronismo da varredura e amostragem com a aquisição de dados assegura a obtenção, a cada intervalo definido de tempo, de um conjunto de dados com quantidade correspondente a uma varredura completa contendo o mesmo número de linhas, onde cada linha possui um número igual e inteiro de elementos. O sincronismo não assegura, contudo, que cada conjunto de dados possua exatamente uma imagem completa, mas sim que o conjunto de dados transferido possui sempre um número de elementos equivalente a uma varredura completa. Isso acarreta em conjuntos de dados contendo imagens fracionadas, conforme é ilustrado na figura 2.13, o que requer a busca do início da cada nova

imagem, bem como o uso de dados provenientes de mais de um conjunto de dados. Finalmente, apesar do sincronismo e da regularidade no que tange à quantidade de dados transferidos, a varredura é de natureza analógica, de forma que eventualmente há excesso ou falta de elementos em uma ou mais linhas, o que requer a busca do início de cada uma das linhas.

2.6 Imagens obtidas

As imagens obtidas pela técnica de AFM são, em geral, de baixa resolução, podendo, em casos especiais, apresentar matrizes de 1024x1024 *pixels*. Em aplicações com elevadas taxas de varredura (*fast scanning*), as resoluções empregadas usualmente são da ordem de 128x128 ou 256x256 *pixels*, dependendo da área de varredura e da taxa de aquisição de dados. As dimensões usualmente são as mesmas em ambos os eixos (x e y), porém em casos especiais a taxa de aquisição de dados pode ser aumentada sem que a taxa de varredura seja modificada, acarretando em imagens com resolução ampliada no eixo de alta velocidade de varredura, ao longo das linhas.

As imagens são representadas por matrizes contendo valores em ponto flutuante representando a altura do relevo da amostra num dado ponto $\{x, y\}$. No plano bidimensional (imagens planas), são usualmente descritas por variações de tons de cinza através da conversão dos valores medidos para uma escala monocromática discreta de 8 *bits*, obtendo assim 256 gradações distintas.

Em razão dos diversos efeitos degenerativos atuantes (borramento e ruído aditivo),

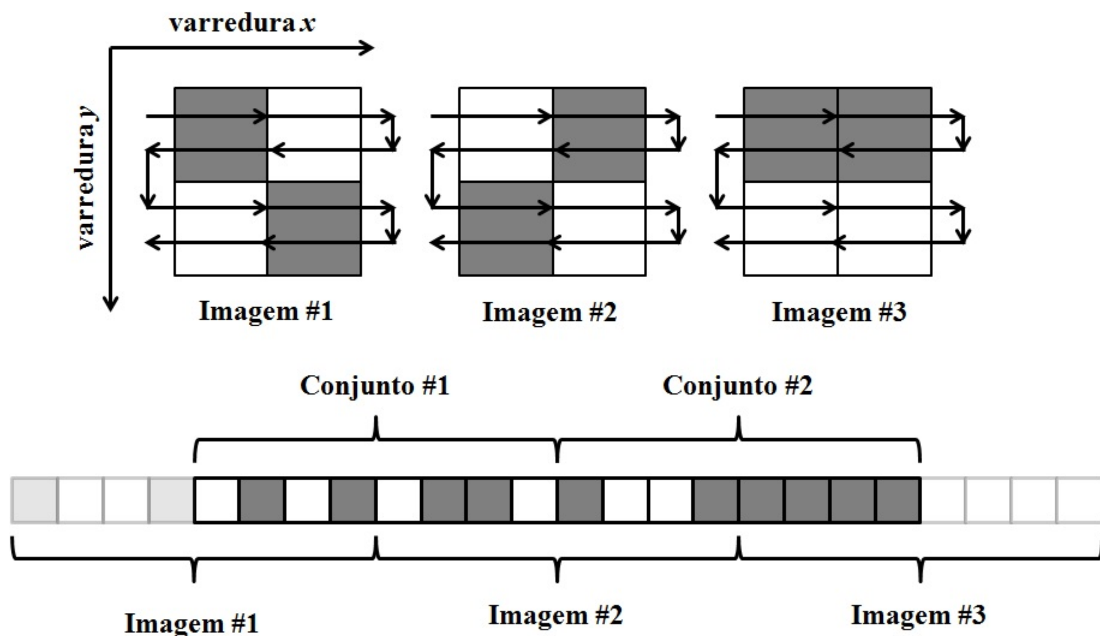

Figura 2.13: Exemplo de conjuntos de aquisição de dados de imagens.

devido às características de interação da ponteira com a amostra, da varredura, bem como da instrumentação e montagem empregados, as imagens obtidas apresentam uma relação sinal/ruído pobre, o que leva à necessidade do emprego de técnicas de restauração de imagens para a correta visualização das superfícies em análise.

Capítulo 3

Computação paralela usando a arquitetura CUDA

Os avanços tecnológicos observados nas últimas décadas decorrem principalmente do aumento da capacidade de processamento experimentado pelos computadores. Desde os primeiros computadores de uso pessoal, no início da década de 1980, este aumento devia-se principalmente, dentre outros fatores, ao aumento da frequência de operação, ou *clock*, dos processadores. Desde então, as taxas de *clock* evoluíram de cerca de 1 MHz para cerca de 4 GHz, o que corresponde a uma aceleração de 4000 vezes, aproximadamente. Contudo, no início dos anos 2000, a velocidade com que o aumento do *clock* dos processadores era observada caiu drasticamente, em razão das limitações físicas alcançadas no desenvolvimento dos circuitos integrados que compõem a unidade central de processamento (*Central Processing Unit* - CPU) dos computadores: Tamanho dos transistores, superaquecimento e potência de operação [21]. Paralelamente, a demanda por parte de desenvolvedores e usuários comuns por computadores com maior poder de processamento continuou aumentando a taxas cada vez maiores, de forma que outras alternativas de ganho de poder computacional passaram a ser exploradas.

Neste período surgiram os primeiros processadores com duas ou mais unidades de processamento, o que alavancou o desenvolvimento de aplicações empregando processamento paralelo. Até então a grande maioria das aplicações desenvolvidas eram escritas como programas sequenciais, e as aplicações paralelas se restringiam basicamente a programas de grande porte executados em supercomputadores, ou *clusters*, e processamentos gráficos executados em placas de vídeo, ou Unidades de Processamento Gráfico (*Graphical Processing Units* - GPUs).

Estas últimas desempenharam um papel importante no desenvolvimento de técnicas de computação paralela, devido a crescente demanda de aplicações com qualidade gráfica superior a partir do início dos anos 90, que exigiam a execução massiva de operações aritméticas. Neste período, o paralelismo empregado nas GPUs era baseado em *pipelines*, cuja operação era configurável, mas não programável [22]. A operação das GPUs con-

sistia em, a partir de dados previamente informados, obter valores numéricos correspondentes a cor de um *pixel* a ser plotado na tela, sendo as operações aritméticas necessárias realizadas pelo *pipeline*, sem intervenção dos programadores.

Em 2001, atendendo à crescente demanda do mercado de desenvolvimento de software, e atendendo aos requisitos do recém-lançado padrão DirectX 8.0, a NVIDIA lançou a GeForce 3 como a primeira GPU com *pipeline* programável. Não tardou para que os desenvolvedores percebessem que, uma vez que a aritmética realizada na GPU poderia ser controlada, os dados fornecidos não necessariamente deveriam ser referentes à imagens, mas sim dados quaisquer, o que incentivou o desenvolvimento de aplicações que fossem executadas na GPU em busca de menores tempos de execução. Entretanto, as restrições no modelo de programação, as limitações dos recursos disponíveis, bem como a necessidade de se conhecer as bibliotecas gráficas, como OpenGL ou DirectX (as únicas maneiras de interação com as GPUs), bem como as linguagens de *shading*, se tornaram um obstáculo para a aceitação geral deste modelo de programação.

Finalmente, em 2006 a NVIDIA lançou a GeForce 8800 GTX, a primeira GPU implementada com a arquitetura CUDA (*Compute Unified Device Architecture*), voltada tanto para aplicações gráficas quanto para computação numérica em geral, razão pela qual recebeu a denominação de Unidade de Processamento Gráfico de uso Geral (*General-Purpose Graphics Processing Unit* - GPGPU). Logo em seguida lançou uma linguagem de programação baseada na mundialmente consagrada linguagem C acrescida

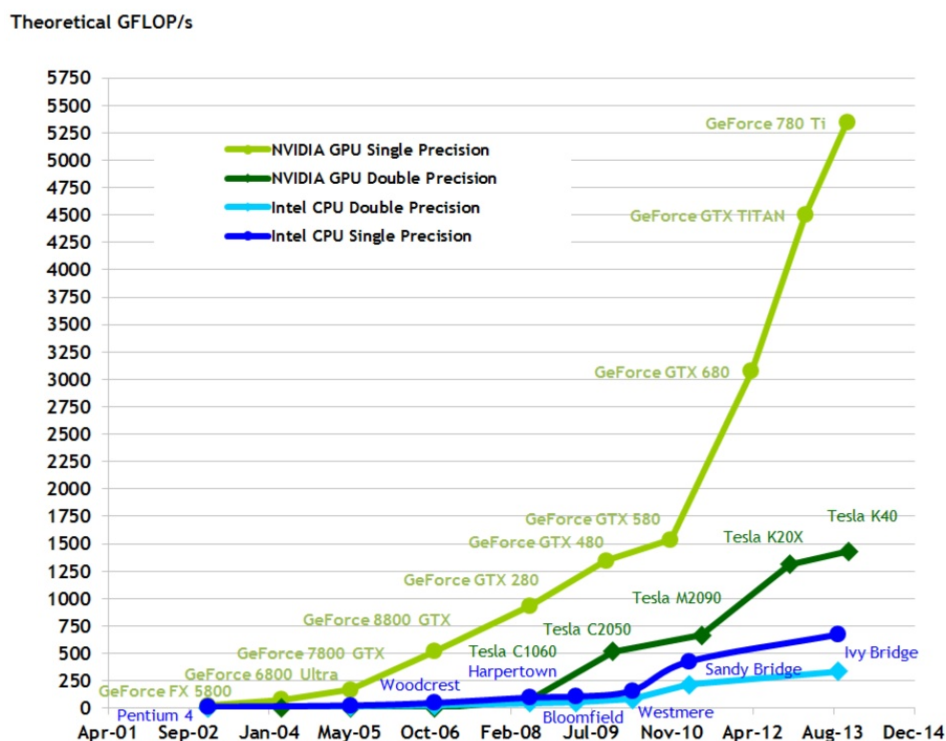

Figura 3.1: Evolução das taxas de processamento de CPUs e GPUs. NVIDIA (2012) [23].

de palavras-chave específicas (linguagem CUDA C), acompanhada de um compilador próprio (nvcc) e um *driver* que explorava as funcionalidades da arquitetura CUDA. Desta forma, o conhecimento de bibliotecas gráficas ou linguagens de *shading* para o uso da arquitetura paralela das GPUs não era mais necessário, e aplicações empregando as funcionalidades da arquitetura passaram a ser largamente exploradas. Atualmente, há suporte para a utilização da plataforma CUDA com diversas outras linguagens de programação, como C++, Fortran, Java e Python, e ambientes de desenvolvimento.

Enquanto que a velocidade de aumento da capacidade de processamento das CPUs caiu significativamente, a capacidade de processamento das GPUs continuou aumentando a taxas surpreendentes. A figura 3.1 exibe a evolução das taxas de processamento das CPUs e GPUs em GFLOPS/s, o que equivale ao número máximo de operações de ponto flutuante por segundo que o dispositivo é capaz de executar. A enorme discrepância observada se deve ao fato de as GPUs serem projetadas para realizarem operações aritméticas massivamente e em paralelo, como são as operações gráficas, enquanto que CPUs são projetadas levando em consideração fatores como controle de fluxo e *cache* de dados [23]. De uma maneira geral, GPUs dedicam mais transistores para o processamento de dados, razão pela qual é mais adequada para soluções que podem ser executada massivamente em paralelo, com poucos desvios de fluxo.

3.1 Princípios básicos

A arquitetura CUDA é baseada no conceito de *Streaming Multiprocessors* (SMs), que consiste em processadores capazes de executar centenas de *threads* simultaneamente. O conceito é baseado na arquitetura *Single Instruction Multiple Thread* (SIMT), que por sua vez se assemelha ao modelo SIMD (*Single Instruction Multiple Data*), que define arquiteturas vetoriais [23]. Cada SM cria, gerencia, oganiza e executa *threads* em grupos de 32 *threads* paralelas chamados *warps*. As *threads* de um warp executam o mesmo programa, porém cada uma possui o próprio contador de instruções e registradores de estado, de forma que podem executar o mesmo conjunto de instruções de maneira independente. As instruções das *threads* dentro de um *warp* são executadas uma de cada vez de maneira simultânea, de maneira que o desempenho máximo é alcançado quando todas as *threads* executam as mesmas instruções. Em caso de divergências durante a execução (desvios condicionais), o *warp* executará todos os caminhos possíveis serialmente, desativando as *threads* que não seguirem cada caminho, até que os caminhos de execução convirjam novamente. Divergências ocorrem apenas dentro de um *warp* e prejudicam de maneira significativa o desempenho da aplicação, de forma que desvios condicionais devem ser aplicados apenas quando for estritamente necessário.

Um programa CUDA é uma função, chamada *kernel*, que quando invocada pelo programa principal é executada por N *threads* distintas em paralelo. As *threads* são organi-

Figura 3.2: Execução de um programa CUDA em GPUs com diferentes números de SMs. NVIDIA (2012)[23].

zadas em *warps*, que por sua vez são organizados em blocos. Uma vez que há recursos dedicados para cada *thread*, há um limite de *threads* que podem ser alocadas por bloco. Cada SM executa um bloco por vez, alternando a execução dos *warps* conforme a conveniência. Isso garante a escalabilidade da arquitetura CUDA, ao permitir que GPUs com diferentes números de SMs sejam capazes de executar um mesmo kernel, conforme ilustra a figura 3.2. Uma GPU com um número maior de SMs executará o mesmo programa em menos tempo. Por exemplo, a GPU GeForce GT 555M, utilizada neste trabalho, possui 3 SMs com 48 núcleos cada, sendo capaz de alocar até 1024 *threads* por bloco. Um conjunto de blocos é chamado de *grid*.

Uma aplicação CUDA pode conter um ou mais *kernels* que são invocados durante a execução do programa principal, além da codificação serial tradicional. Além disso, um *kernel* pode realizar chamadas a funções que serão executadas diretamente na GPU. Este modelo heterogêneo de programação é adotado em razão de haverem tarefas que são executadas de maneira mais eficiente em CPU do que em GPU. Tanto o programa serial como o paralelo possuem os próprios espaços de memória, de forma que são executados de forma independente. Uma vez invocado um *kernel*, o programa principal imediatamente continua a ser executado na CPU, de forma que a sincronização entre CPU e GPU se dá somente através de chamas a *kernels*, transferências de dados entre as memórias da CPU e GPU, bem como através de solicitações específicas de sincronização.

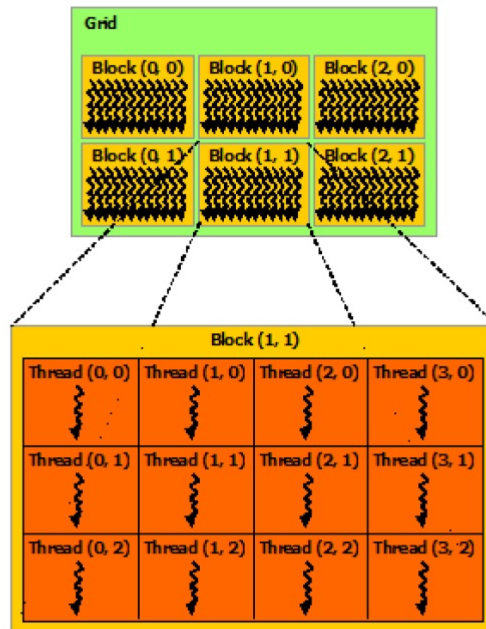

Figura 3.3: Distribuição de um *grid* em blocos e *threads*. NVIDIA (2012)[23].

3.2 Estruturas de blocos e *threads*

No momento da chamada ao *kernel*, o número de blocos e *threads* por bloco são especificados, de forma que o número total de *threads* é igual ao número de *threads* por bloco vezes o número de blocos. As *threads* que compõem um bloco são organizadas uni, bi ou tridimensionalmente e possuem uma identificação única tridimensional (`threadIdx`) correspondente à sua posição no bloco. Os blocos que compõem um *grid* são organizados da mesma forma, possuindo também uma identificação única tridimensional (`blockIdx`) correspondente à sua posição no *grid*. Isso permite estruturar de uma maneira mais pragmática problemas envolvendo vetores, matrizes ou volumes, e através dos identificadores `blockIdx` e `threadIdx` é possível identificar cada *thread* distinta durante a execução do programa (vide figura 3.3).

Os blocos são executados de maneira independente uns dos outros, em qualquer ordem, seja em série ou em paralelo, o que permite que sejam distribuídos em qualquer número de SMs, conforme a figura 3.2. Sincronização e compartilhamento de dados ocorrem somente entre as *threads* de um bloco específico, não podendo comunicar-se com *threads* de outros blocos.

3.3 Hierarquia de memória e tráfego de dados

Uma *thread* acessa diferentes espaços de memória durante sua execução, com diferentes níveis de compartilhamento e tempos de existência, conforme é ilustrado na figura 3.4. A nível de *thread*, cada uma possui um conjunto de registradores, o que inclui os identifi-

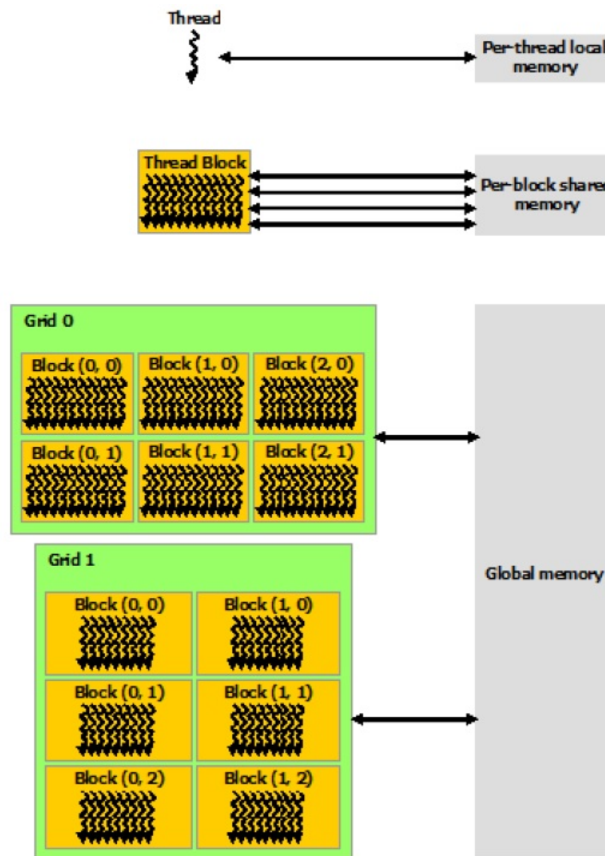

Figura 3.4: Hierarquia de memórias em CUDA. NVIDIA (2012)[23].

cadores `blockIdx` e `threadIdx`, além de variáveis declaradas como privadas à *thread*. Este espaço de memória somente é acessível pela própria *thread*, e uma vez extinta, ao fim da execução do *kernel*, os dados são perdidos. Registradores possuem elevada velocidade de acesso (leitura e escrita), sendo que cada SM possui um pequeno número destes, que são distribuídos entre as *threads*, o que é um limitador ao número de *threads* que podem ser alocadas por bloco.

A nível de bloco, cada SM possui uma memória compartilhada, que pode se acessada por todas as *threads* tanto para escrira como para leitura, com elevada velocidade. Ela se assemelha a uma memória *cache* que pode ser gerenciada pelo desenvolvedor, e de fato, para GPUs mais recentes, o espaço de memória dedicado ao *cache* L1 e a memória compartilhada é o mesmo, cabendo ao programando definir quanto deste espaço cabe a cada tipo de memória. A memória compartilhada, além de ser uma memória de acesso eficiente, permite o trabalho cooperativo entre *threads* de um mesmo bloco. Uma vez encerrado o *kernel*, o conteúdo da memória compartilhada é perdido. Apesar de comportar mais dados que registradores, o espaço da memória compartilhada também é limitado e deve ser usado com sabedoria, restringindo assim o número de *threads* por bloco.

Finalmente, a GPU possui uma memória global, acessível por todas as *threads* de todos os blocos tanto para leitura quanto escrita, cujo conteúdo persiste durante toda a

aplicação, inclusive entre *kernels* diferentes, o que favorece imensamente o trabalho cooperativo entre *threads*. O custo desta flexibilidade é uma velocidade de acesso consideravelmente inferior, quando comparada à memória compartilhada e registradores. Essa memória é acessível também pelo programa principal em execução na CPU (*host*), de forma que as transferências de dados entre a CPU e a GPU se dão através da memória global. Para GPUs mais recentes, acessos à memória global para a SM são feitos através do *cache*, de forma que explorar a localidade de dados é um importante aspecto a ser observado para compensar os altos tempos de acesso. Além da memória global, a CPU possui outros dois tipos de memória usadas somente para leitura por todas as *threads*, sendo estes a memória de constante e de textura, que também podem ser acessadas pelo *host* e, da mesma forma que a memória global, mantém os dados durante toda a aplicação.

O acesso às memórias global, de constante e textura realizadas pelo *host*, tanto para escrita quanto para leitura, se dá com velocidade ainda menores, de maneira que trocas de dados entre a CPU e a GPU devem ser reduzidos ao mínimo necessário, sob o risco de se comprometer gravemente o ganho de desempenho da aplicação, que é o objetivo principal quando se desenvolve programas utilizando CUDA. Algumas estratégias, contudo, podem minimizar o tempo de acesso à memória, como por exemplo o uso de memória não-paginável (*page-locked memory*) no *host*. Uma vez que elas não são paginadas pelo sistema operacional, o tempo de acesso é reduzido consideravelmente, além do fato de que transferências envolvendo memórias não-pagináveis podem ser realizadas de maneira concorrente [23]. Além disso, o uso de memória não-paginável permite que um endereço de memória no *host* seja mapeado diretamente pela GPU, tornando as cópias de dados desnecessárias. Aplicações deste tipo são ditas como sendo *zero-copy*, onde o acesso à memória do *host* pela GPU é realizado cada vez que o *kernel* acessa a memória mapeada, o que em casos específicos resulta em ganhos de desempenho ainda maiores.

3.4 Otimizando o desempenho

Considerando a arquitetura CUDA, bem como aspectos de programação paralela em geral, a otimização de aplicações pode ser alcançada através de algumas medidas [23].

3.4.1 Maximizando a utilização da GPU

Para que uma aplicação tenha sua utilização maximizada, ela deve ser estruturada para explorar ao máximo o paralelismo disponível, utilizando todos os SMs, de maneira que todos os processadores estejam ocupados o máximo possível, evitando setores ociosos.

As *threads* de um *warp* ficam ociosas enquanto esperam por recursos, como acessos à memória, por exemplo, ou quando realizam uma instrução. Este período de espera (ou *latência*) depende da operação realizada, podendo variar de alguns ciclos de *clock* para

algumas centenas. Assim sendo, o gerenciador de *warps* da SM substitui o *warp* ocioso por um que esteja pronto para executar sua próxima instrução, a cada ciclo de *clock*. Desta forma, é necessário estruturar a aplicação paralela com um número de *warps* suficiente para ocultar esta latência.

3.4.2 Maximizando o acesso à memória

Deve-se priorizar o acesso às memórias com menores tempo de acesso, como as memórias compartilhadas e registradores, uma vez que o acesso à memória global é significativamente mais lento. Programas que fazem uso destas memórias geralmente apresentam desempenhos superiores.

Por outro lado, a estruturação do algoritmo para fazer o uso adequado da memória compartilhada pode aumentar excessivamente a complexidade do código a ser implementado. Além disso, o uso da memória compartilhada é um fator limitador ao número de *threads* que podem ser executadas em um bloco, o que pode impactar negativamente no desempenho da aplicação.

Em acessos à memória global, a localidade de dados deve ser levada em consideração, de maneira a tirar vantagem do uso do *cache* dos SMs. Acessos à memória que levam em consideração a localidade de dados, nos quais as *threads* de um bloco acessam um trecho contíguo da memória global, são chamados de acessos *coalescentes*, e podem ocultar os altos tempos de espera por dados.

As trocas de dados entre CPU e GPU devem ser reduzidas ao mínimo necessário, uma vez que os tempos envolvidos são extremamente elevados. O uso de memórias não-pagináveis no *host*, bem como o mapeamento de memória diretamente pela GPU (*zero-copy*) podem reduzir de maneira significativa os tempos de acesso.

3.4.3 Maximizando o uso de instruções

Deve-se dar preferência a funções intrínsecas, que são versões menos precisas porém de execução mais rápida de funções regulares, desde que esta medida não afete o resultado final. É recomendável também evitar as conversões de dados, visto que consomem, às vezes desnecessariamente, muitos ciclos de *clock*, além usar variáveis de ponto flutuante com precisão simples ao invés de precisão dupla, e evitar sincronizações desnecessárias.

É extremamente recomendável reduzir ao mínimo necessário os desvios condicionais, que acabam por manter *threads* ociosas, prejudicando o desempenho da aplicação. Em muitos casos operações lógicas substituem adequadamente os desvios condicionais, otimizando a execução, uma vez que todas as instruções serão executadas para todas as *threads*.

Capítulo 4

Ferramenta para exibição de imagens de microscopia de força atômica

Considerando o princípio de funcionamento do AFM descrito no capítulo 2, no que tange às particularidades da varredura, aquisição de dados e composição das imagens obtidas, foi desenvolvida uma ferramenta para exibição de imagens de microscopia de força atômica, na forma de uma biblioteca de funções.

Esta ferramenta foi desenvolvida utilizando a linguagem C/C++ [24, 25] utilizando a metodologia orientada a objetos, para ser utilizada em conjunto com o software de controle e monitoramento do AFM desenvolvido por Cidade [6], construído em uma parceria entre o Instituto Nacional de Metrologia, Qualidade e Tecnologia (Inmetro) e o IBC-CF/UFRJ, atualmente em uso no IBCCF (figura 4.1). O foco deste trabalho, apesar de es-

Figura 4.1: Microscópio de Força Atômica em uso no IBCCF.

pecífico, não excluiu a possibilidade de utilização desta ferramenta em outras aplicações de microscopia de força atômica, e os princípios que nortearam o desenvolvimento da aplicação são apresentados nas sessões seguintes.

A biblioteca implementa a classe *DBufferAFM*, sendo todas as funcionalidades apresentadas neste capítulo acessadas através dos métodos públicos desta, o que permite a composição e exibição de imagens de microscopia de força atômica de maneira totalmente transparente ao usuário da ferramenta. Uma segunda classe foi implementada (*GLAFMDisplay*) para gerenciar a exibição das imagens resultantes em 2D e 3D, sendo o acesso a esta classe restrito aos métodos implementados na classe *DBufferAFM*.

4.1 Não-sincronismo de dados

Conforme foi descrito na seção 2.5 e ilustrado na figura 2.13, o sincronismo da varredura e amostragem do AFM com a aquisição de dados assegura apenas que cada conjunto de dados obtido contém sempre o mesmo número de dados, que correspondem a uma varredura completa em termos de números de linhas e colunas, não assegurando contudo que o conjunto de dados possua exatamente uma imagem inteira, mas sim frações de imagens distintas e amostradas em sequência. Desta forma, para obter uma imagem completa é necessário aguardar o próximo conjunto de dados.

Como forma de contornar esse problema, a classe implementa um *buffer* duplo para armazenamento dos dados do AFM, com tamanho correspondente a exatamente dois conjuntos de dados, o que assegura que, a partir do segundo conjunto de dados obtido, exista uma imagem completa no *buffer*. Este modelo é adotado para os quatro subconjuntos de dados adquiridos, que correspondem aos sentidos de varredura em x e y, o sinal do fotodetetor e a correção da PP.

A figura 4.2 ilustra o funcionamento do *buffer* duplo implementado na biblioteca, para cada instante de tempo. A cada ciclo de aquisição, o conjunto de dados armazenado na segunda metade do *buffer* é movido para a primeira metade, e o novo conjunto é armazenado na segunda metade. Isso assegura a contiguidade dos dados, permitindo a obtenção de imagens completas.

Os quatro subconjuntos de dados adquiridos são vetores de números de ponto flutuante que representam, a cada amostragem, o sinal retornado pelo fotodetetor referente ao erro, correspondente ao sinal do fotodetetor (`photo`), o sinal de correção da plataforma piezoelétrica (`z`), e sinais de tensão elétrica representando os estados lógicos que definem os sentidos de varredura nas direções x (`dir_x`) e y (`dir_y`), onde o estado lógico 1 ou *verdadeiro* (5 V) representa o sentido positivo, ou *ida*, e o estado lógico 0 ou *falso* representa o sentido negativo, ou *volta*. As transições dos estados lógicos de `dir_y` e `dir_x` são utilizadas para identificar o início de cada imagem e o início e/ou o fim de cada linha dentro de uma imagem, respectivamente. O tamanho dos vetores sempre corresponde a

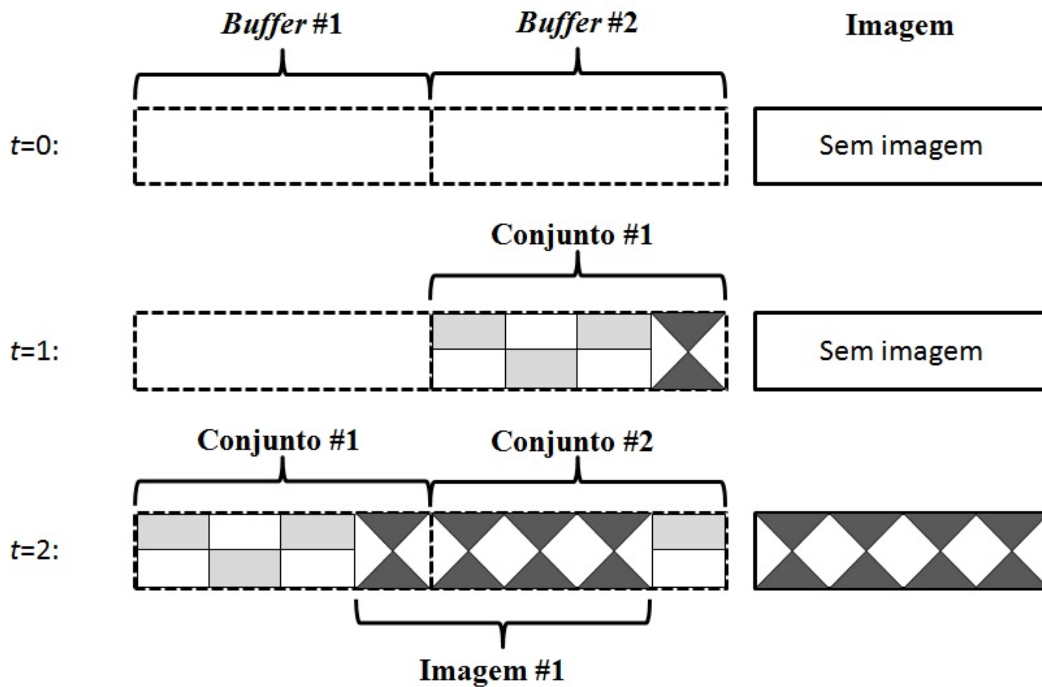

Figura 4.2: Funcionamento do *buffer* duplo.

um conjunto completo de dados, equivalente a uma varredura completa. A alocação do *buffer* duplo é realizada em função da quantidade de dados amostrados durante uma varredura, o que deve ser informado pelo usuário, podendo ser realocado durante a execução em razão de mudanças nestes parâmetros. É importante observar que as dimensões das imagens (resolução) não necessariamente possuem relação de proporcionalidade com as dimensões da área amostrada, conforme será mostrado adiante.

4.2 Aquisição de dados e obtenção de imagens

Conforme foi descrito em 2.4, os dados a serem utilizados para a obtenção de imagens devem ser selecionados de acordo com as formas de onda aplicadas à PP para a realização da varredura. Diante das possibilidades apresentadas, a ferramenta disponibiliza três modos de composição de imagens. Os modos definidos como *ida* e *volta* utilizam, respectivamente, os dados amostrados apenas nos semiciclos crescente e decrescente da varredura horizontal, enquanto o modo *ida e volta* utiliza os dados amostrados em ambos os sentidos de varredura. Isso permite que a ferramenta seja aplicável a diferentes estratégias de varredura e amostragem.

A identificação de uma nova imagem se dá através da identificação da transição dos valores de `dir_y` do nível lógico 0 para 1. Enquanto o estado lógico de `dir_y` permanecer igual a 1, as linhas de varredura serão identificadas através da identificação da transição do estado lógico de `dir_x` de 0 para 1. Esta transição, a cada linha de varredura,

identifica tanto o início de uma linha de varredura no modo *ida* como o fim da uma linha varredura no modo *volta*, servindo para ambos os propósitos no modo *ida e volta*. Uma vez que, apesar de o conjunto de dados possuir sempre o mesmo número de elementos, as linhas não necessariamente possuem o mesmo número de elementos, conforme descrito em 2.5, o que requer a busca e o armazenamento das posições do início de cada linha em vetores que contém as posições de início de cada ida e/ou o fim de cada volta. Finalmente, a regularidade dos dados permite que o ponto de início da primeira imagem seja armazenado para ser utilizado nas próximas imagens, dentro de uma pequena margem de segurança de alguns elementos, o que reduz significativamente o número de iterações nas buscas pelo início das próximas imagens.

A composição da imagem com os dados dos vetores é ilustrada na figura 4.3. No modo *ida*, uma vez identificadas as transições em `dir_x`, as linhas são preenchidas com os dados que se seguem às transições. No modo *volta*, a primeira transição é ignorada, e a partir da segunda transição as linhas são preenchidas com os dados que antecedem cada transição, em ordem inversa. No modo *ida e volta*, ambas as estratégias são empregadas, e o valor do elemento de cada linha é assumido como sendo o menor dos valores correspondentes encontrados na ida e na volta. Isso é assumido em razão do efeito *parachuting* descrito em 2.3, no qual ocorre o afastamento da ponteira em relação a amostra em razão de degraus ou inclinações acentuadas, resultando em medidas superiores às esperadas.

A obtenção da imagem final se dá através da composição de duas imagens distintas, representadas por matrizes de números de ponto flutuante com $W \times H$ elementos, correspondentes às imagens de erro, relativa ao sinal do fotodetetor, e a imagem relativa à correção em z da plataforma piezoelétrica. Isto se faz necessário uma vez que os sinais representam variações no relevo da amostra de maneira diferenciada, de forma que pre-

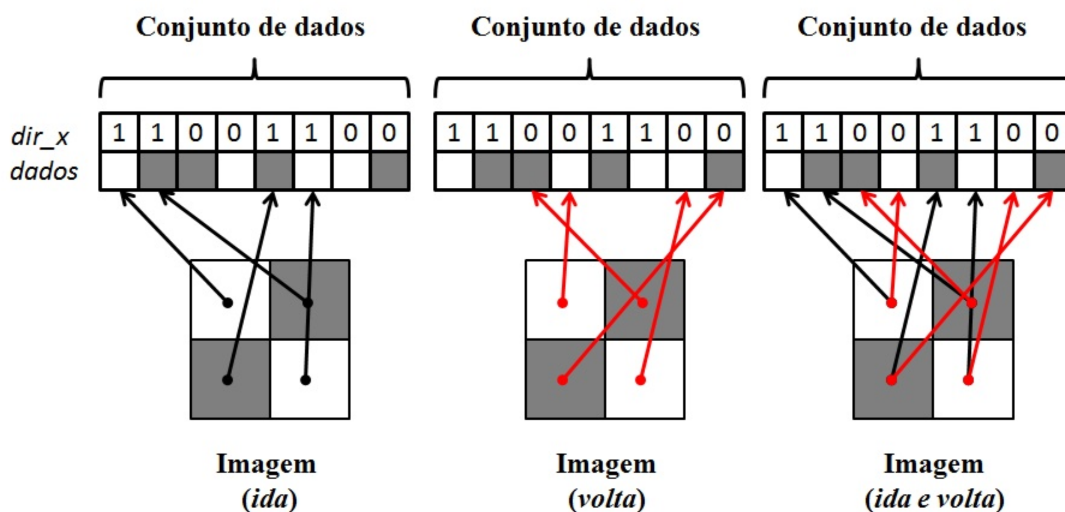

Figura 4.3: Composição das imagens conforme conforme diferentes modos. O vetor *dados* refere-se a qualquer um dos subconjuntos de dados.

cisam ser ajustados conforme a sensibilidade de cada sistema de medição, que deve ser previamente informada, para que possam representar adequadamente a mesma grandeza medida (comprimento), devendo também ser corrigidas pelo seu desvio (*offset*). Em regime de *altura constante* a imagem final será a imagem de erro, e em regime de *força constante* será o inverso da imagem de correção da PP, uma vez que esta correção aplicada tende a remover o desnível registrado pelo fotodetetor. É possível também compor a imagem como a soma dos dados de ambas as imagens, em uma abordagem híbrida. A imagem final é então descrita por uma matriz de ponto flutuante, onde cada elemento $\{i, j\}$, ou $\{x, y\}$, representa a altura (posição z) da superfície àquela posição.

4.3 Visualização em 2D e 3D

A ferramenta permite a visualização das imagens obtidas através do uso da biblioteca gráfica OpenGL [26] e da interface *freeglut* [27, 28]. A figura 4.4 exibe uma imagem obtida a partir de dados obtidos com o AFM do IBCCF, e corresponde a uma região de uma grade de teste com 9x9 micrometros de área e uma varredura de 128 linhas (eixo y) por 128 elementos (eixo x). É possível rotacionar, transladar, aproximar e/ou afastar a imagem livremente, de maneira a permitir a visualização de detalhes das imagens obtidas, ou ainda visualizá-la em tela cheia. Além disso, como forma de permitir a visualização das imagens em um formato mais tradicional à técnica de AFM, a ferramenta permite alterar o modo de projeção, alternando entre perspectiva e paralela, obtendo assim uma visualização 2D (figura 4.5), que também permite as transformações espaciais já citadas (rotação, translação e zoom).

Tendo em vista as particularidades da varredura e aquisição de dados, principalmente em regime de *fast scanning*, e considerando ainda que as imagens obtidas poderão vir a ser submetidas a um custoso processo de restauração, que será descrito em detalhes no capítulo 5, a visualização foi desenvolvida com o intuito de ser computacionalmente eficiente, de maneira a não retardar a execução do software de controle, bem como a aquisição e composição das imagens implementadas na ferramenta, de acordo com as recomendações de otimização descritas em [26].

Dentre as medidas adotadas, destaca-se o uso de *vertex arrays* previamente indexados, ao invés do lançamento individual de vértices, o que é extremamente mais custoso. Uma vez que a estrutura horizontal das imagens (posições x e y de cada vértice) é rigorosamente a mesma ao longo da execução, salvo eventuais redimensionamentos, previstos na implementação, o vetor contendo os índices que orientam a construção da superfície 3D (conjunto de triângulos adjacentes), a partir dos *arrays* de vértices e de cores, é inicializado no construtor da classe que gerencia a exibição das imagens, enquanto que os demais vetores são preenchidos em função dos dados adquiridos.

Apesar de as imagens obtidas com a técnica de AFM serem naturalmente mono-

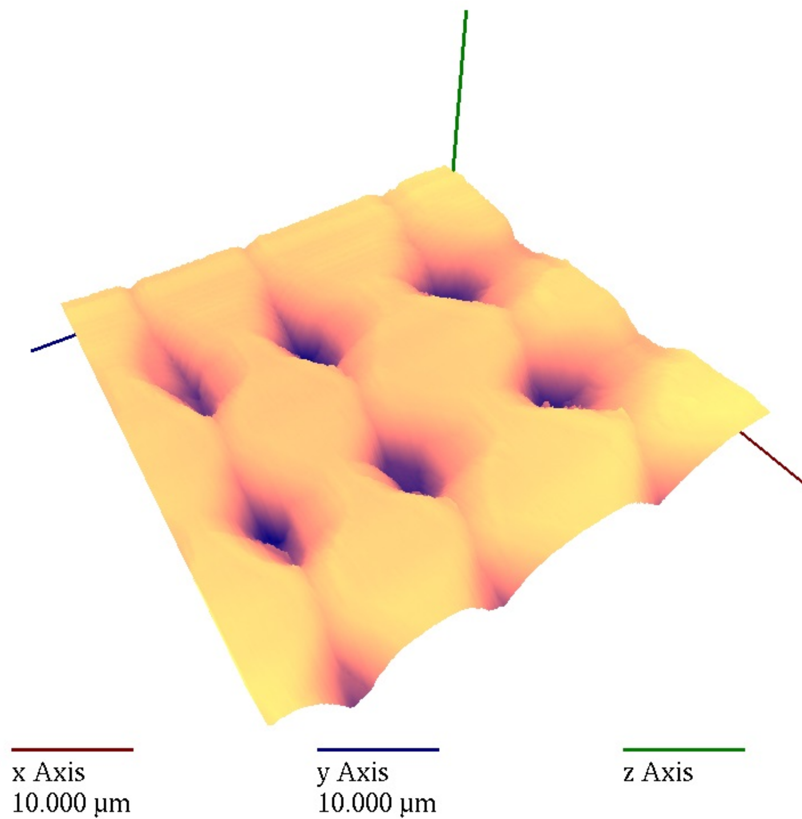

Figura 4.4: Exemplo de visualização em 3D de uma imagem de AFM, obtida a partir de uma varredura de 256x256 elementos cobrindo uma área de 9x9 micrometros.

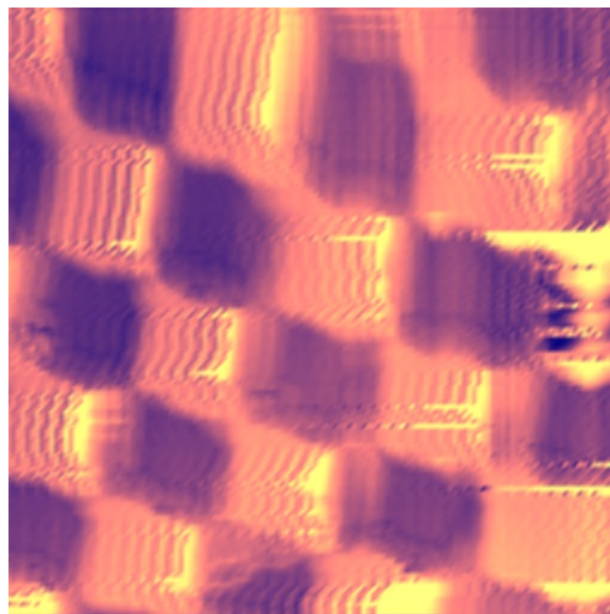

Figura 4.5: Exemplo de visualização em 2D de uma imagem de AFM.

cromáticas, por serem representadas por apenas um valor de intensidade por elemento, uma escala de cores simplificada foi implementada, de maneira a permitir a visualização dos detalhes do relevo da superfície exibida, conforme exibido na figura 4.4. Esta abor-

dagem foi uma alternativa mais eficiente ao uso de técnicas de iluminação, que requerem cálculos de vetores normais para cada vértice, além de adicionarem tarefas extras ao *pipeline* gráfico da GPU.

Finalmente, como forma de evitar que o processo de renderização das imagens obstruísse a execução das demais tarefas, a janela de exibição é inicializada em uma *thread* separada do processo principal, de forma que a execução da mesma é realizada de maneira independente. A manipulação dos *arrays* de vértices e cores é realizada pelo programa principal, e o acesso concorrente é evitado através do uso de barreiras. O conjunto de medidas apresentado foi responsável por uma redução de mais de 90% no tempo de renderização, quando comparando com uma abordagem menos eficiente, que envolvia cálculo de normais, iluminação e vértices sendo lançados individualmente.

Uma alternativa ao uso da biblioteca OpenGL disponibilizada pela ferramenta é o uso da biblioteca CImg [29], utilizada para exibição de imagens 2D monocromáticas. Para tal, as posições z das imagens obtidas devem ser convertidas para uma escala discreta positiva de 8 bits, obtendo assim 256 gradações de tons de cinza distintas. As figuras 2.8 e 2.12 exibem exemplos de imagens obtidas através da biblioteca CImg.

4.3.1 Visualização em escala real

Para que seja possível a visualização 3D da superfície de uma amostra em escala real, ou seja, que sejam mantidas as proporções entre comprimento, largura e altura da área amostrada, é necessário que sejam informadas as dimensões da área de varredura, de forma que a posição $\{x, y, z\}$ de cada vértice (elemento) seja ajustada conforme estas dimensões, considerando que os valores dos elementos já estejam ajustados conforme as sensibilidades do fotodetetor e/ou da PP, representando assim valores em unidades de comprimento. Para tal, são calculados valores de sensibilidade nas direções x e y, definidos como a razão entre a dimensão da área de varredura e a dimensão da imagem, expressos em $\mu m/pixel$.

Esta abordagem é particularmente útil nos casos onde as dimensões da área de varredura e da imagem não seguem uma relação de proporcionalidade. Conforme descrito em 2.6, em casos particulares a taxa de amostragem de dados do AFM pode ser aumentada de forma a obter imagens com resolução ampliada na direção x de varredura, mesmo que as dimensões métricas da área de varredura sejam idênticas. Caso as imagens não fossem ajustadas conforme a área de varredura, ela seria exibida distorcida na direção x. Considerando as inúmeras combinações de dimensões de imagens (pixels) e área de varredura (m^2) possíveis, optou-se por manter a fidelidade ao relevo através do ajuste das imagens conforme as sensibilidades do fotodetetor e da PP, bem como conforme as dimensões da imagem e da área de varredura, mesmo que isso resulte em um aumento no tempo de processamento. Como exemplo, a visualização 3D exibida na figura 4.4, bem

como sua projeção paralela foram ajustadas conforme estes parâmetros. Mesmo que as dimensões da imagem e da área de varredura sejam proporcionais, esta abordagem ainda assim assegura a proporcionalidade no eixo z, em relação aos demais.

As sensibilidades do fotodetetor e da plataforma piezoelétrica, no caso específico do AFM para o qual esta ferramenta foi desenvolvida, foram determinadas através de um experimento empregando um sensor fotônico, capaz de detectar deslocamentos da ordem de nanometros. O sensor é posicionado sobre PP, e nela é aplicada uma determinada tensão que provoca um deslocamento desta no eixo z, que é detectado pelo sensor, permitindo assim determinar a sensibilidade da cerâmica piezoelétrica, e por consequência do sinal de reposicionamento da PP empregado no modo *força constante*. De posse desta informação, os deslocamentos da PP no eixo z, agora de comprimentos conhecidos, são registrados a partir do fotodetetor, o que permite a determinação da sensibilidade deste para a obtenção de imagens no modo *altura constante*. O experimento resultou em valores de sensibilidade de $0,32\ \mu m/V$ e $6,57\ \mu m/V$ para os sinais de correção da PP e do fotodetetor, respectivamente. O cálculo correto das sensibilidades aliado à exibição das imagens em escala real pode permitir a realização de medições dimensionais das amostras estudadas.

4.4 Paralelização utilizando a arquitetura CUDA

Levando em conta o esforço computacional necessário à composição das imagens de microscopia de força atômica a partir dos dados das varreduras, e considerando as altas taxas de aquisição de dados a serem empregadas no modo *fast scanning*, o que requer o gerenciamento eficiente das informações enviadas pelo sistema de medição, além da necessidade de permitir o controle da varredura em tempo de execução (alterar parâmetros de varredura e aquisição de dados), uma estratégia de composição de imagens empregando a arquitetura CUDA foi desenvolvida de forma a desonerar a CPU, ao menos em parte, deste custo.

Para tal, o processo de obtenção de imagens foi dividido em etapas, de maneira a identificar aquelas que seriam candidatas a serem executadas em GPU. As etapas envolvidas são descritas a seguir:

1. *Swap* do *buffer* duplo e carregamento de novo conjunto de dados;
2. Busca pelo início da imagem e o início de cada linha;
3. Composição da imagem:
    (a) Composição das imagens de erro (fotodetetor) e correção em z;
    (b) Determinação do *offset* de ambas as imagens;

(c) Composição da imagem final;

4. Renderização da imagem:

    (a) Cálculo de vértices e cores (*vertex arrays*);

    (b) Chamada à função de *display* via OpenGL;

Dentre as etapas apresentadas, aquelas que se mostram mais apropriadas a serem executadas em paralelo são as etapas 3(a), 3(c) e 4(a), uma vez que envolvem operações a serem realizadas em cada elemento da matriz (imagem) individualmente, a partir somente da leitura de setores específicos do conjunto de dados, ou seja, não existe relação de dependência entre os dados a serem calculados. As etapas 2 e 3(b) envolvem a busca de valores ao longo dos dados analisados (busca do início de imagens e linhas, e busca do mínimo (*offset*) da imagen de erro e máximo da imagem de correção em z, respectivamente), tarefas que são executadas sequencialmente, uma vez que há dependência entre os dados obtidos em cada iteração. A etapa 1 é realizada simplesmente pela chamada à função *memcpy*, conforme padrão ANSI C [24], enquanto que a chamada à função de *display* descrita em 4(b) se dá apenas pela indicação de que há uma nova imagem a ser renderizada, uma vez que o gerenciamento da janela de exibição é realizada por uma *thread* independente.

A estrutura de blocos e *threads* empregada na abordagem paralela é tal que cada bloco é responsável pelo processamento de uma linha, e cada *thread* é responsável pelo cálculo de um elemento (*pixel*) de cada imagem. Uma vez que a localidade dos dados é levada em consideração, visto que os dados consistem em vetores cujos valores são armazenados sequencialmente, o acesso a estes é otimizado pelo uso do *cache* (acesso *coalescente*), razão pela qual os dados relativos às imagens são armazenados na memória global da GPU. Dado que o início da imagem já foi identificado na etapa 2, assim como o início de cada linha, apenas a porção correspondente a um conjunto de dados, a partir do ponto de início identificado, é enviada à memória global, ao invés de todo o *buffer* duplo, o que minimiza o tráfego no barramento de dados. Junto com os conjuntos de dados são enviados dois vetores, contendo as posições dentro do conjunto enviado, para cada linha da imagem, do início de cada ida e do fim de cada volta. Em razão disso, o envio dos vetores `dir_x` e `dir_y` não é necessário, o que reduz ainda mais o tráfego de dados. A partir da identificação da *thread* (elemento) dentro de um dado bloco (linha), é possível, com o uso dos vetores descritores, identificar de imediato a posição de cada dado dentro do conjunto. O algoritmo 4.1 ilustra a estrutura do *kernel* implementado para a obtenção das imagens de erro e topografia no modo *ida*.

Uma vez obtidas as imagens de erro e correção em z, elas são transferidas de volta para a CPU para que o *offset* de ambas (mínimo do erro e máximo da correção) seja determinado sequencialmente. Em seguida a a imagem final é obtida em paralelo a partir

Algoritmo 4.1: Composição paralela das imagens de erro e topografia no modo *ida*

Entrada: ida[], dados_erro[], dados_corr_z[], w
Saída: img_erro[], img_corr_z[]

Define as posições nos vetores:
1: elemento_img ← (Id_bloco * w) + Id_thread;
2: elemento_ida ← ida [Id_bloco] + Id_thread;
Preenche as imagens:
3: img_erro[elemento_img] ← dados_erro[elemento_ida];
4: img_corr_z[elemento_img] ← dados_corr_z[elemento_ida];

dos valores de *offset*, das sensibilidades do fotodetetor e da plataforma piezoelétrica, e das imagens de erro e correção em z (que ainda se encontram na memória global da GPU), conforme exibido no algoritmo 4.2.

Desta forma, a imagem resultante é constituída de valores positivos que representam a altura da amostra, para cada posição da matriz, em escala de micrometros, dado que as sensibilidades são expressas em $\mu m/V$. Nestas condições, a imagem está apta a ser renderizada adequadamente. Em seguida, o cálculo dos elementos dos *arrays* de vértices e cores (proporcionais à profundidade da superfície) para cada elemento é executado em paralelo, conforme o algoritmo 4.3, onde a variável *norma* é definida como sendo a maior das dimensões laterais, x ou y, de forma a assegurar o ajuste adequado da imagem na tela.

Para realçar a profundidade de diferentes regiões da área amostrada, uma vez que o uso de iluminação foi evitado por razões de desempenho, uma escala de cores simplificada foi adotada, variando do azul (mínimo) ao amarelo (máximo), de acordo com o valor da posição z do vértice, conforme pode ser visto no algoritmo 4.3. Uma vez preenchidos os *arrays*, eles são finalmente renderizados. Como o gerenciamento da janela de exibição em OpenGL é realizado em uma *thread* distinta, é necessário o uso de barreiras para evitar o acesso simultâneo e a consequente exibição de imagens incompletas. A comunicação entre o programa principal e a *thread* de gerenciamento da exibição se dá através do compartilhamento dos *arrays* de vértices e cores, do controle do acesso concorrente e do uso de *flags* indicando a existência de novas imagens. É importante observar que existem outros métodos que provavelmente integram de maneira mais eficiente as bibliotecas CUDA

Algoritmo 4.2: Composição paralela da imagem final

Entrada: min_erro, max_corr_z, sens_foto, sens_PP, img_erro[], img_corr_z[], w
Saída: img[]

Define a posição na imagem:
1: elemento ← (Id_bloco * w) + Id_thread;
Preenche a imagem final:
2: img [elemento] ← (img_erro [elemento] - min_erro) × sens_foto
- (img_corr_z [elemento] - max_corr_z) × sens_PP;

Algoritmo 4.3: Construção paralela dos *arrays* de vértices e cores

Entrada: img[], w, h, sens_x, sens_y, norma

Saída: vertices[], cores[]

Define as posições na imagem e nos arrays:
1: elemento_img ← (Id_bloco * w) + Id_thread;
2: elemento_array ← 3 × elemento_img;
Preenche o array de vértices:
3: vertices [elemento_array] ← (Id_thread - (w / 2)) × sens_x / norma;
4: vertices [elemento_array + 1] ← ((w / 2) - Id_bloco) × sens_y / norma;
5: vertices [elemento_array + 2] ← img [elemento_img] / norma;
Preenche o array de cores:
6: cores [elemento_array] ← vertices [elemento_array + 2] × 10;
7: cores [elemento_array + 1] ← vertices [elemento_array + 2] × 5;
8: cores [elemento_array + 2] ← 0.5;

e OpenGL. Entretanto, o tempo de processamento da construção dos *arrays* de vértices e cores para a renderização da imagem final representam uma parcela pouco significativa do tempo total de processamento, razão pela qual o estudo destes métodos foi deixado para trabalhos futuros. Tanto na abordagem serial quanto na paralela, as chamadas à biblioteca OpenGL são realizadas da mesma forma.

Conforme pode ser observado nos algoritmos, em cada um deles a quantidade de operações realizadas por elemento é muito pequena, razão pela qual a paralelização pode vir a não ser justificável, especialmente para imagens pequenas, em virtude do custo das chamadas à API CUDA para a execução de cada *kernel*, bem como que em razão do tempo de transferência de dados entre CPU e GPU, sendo este último usualmente o maior responsável pelo comprometimento do desempenho de uma aplicação CUDA. Levando em consideração este aspecto, diferentes estratégias de alocação e tráfego de dados foram experimentadas, de maneira a identificar aquela que proporciona o melhor desempenho. As estratégias empregadas foram a alocação padrão da linguagem C (*malloc*), a alocação de memória não-paginável (*page-locked memory*), e o mapeamento da memória diretamente pela GPU. Os resultados da avaliação dos algoritmos, em comparação com uma abordagem serial, serão mostrados no capítulo 6. É importante também observar que uma vez que a imagem obtida já se encontra na GPU, o custo do tráfego dos dados desta é evitado para uma futura restauração paralela, cujo processo será descrito no capítulo 5.

4.5 Integração com o sistema de monitoramento e controle

O sistema empregado para o controle e monitoramento do AFM em uso no IBCCF foi desenvolvido em ambiente Windows com o uso da linguagem C#, e permite a

configuração dos parâmetros relativos à varredura (dimensões da área de estudo, velocidade do deslocamento da PP) e à amostragem (taxa de aquisição de dados, número de linhas e de elementos por linha), bem como diversos outros parâmetros inerentes à correta operação do AFM. Através deste, o usuário realiza a comunicação com o microcontrolador que gerencia, de maneira independente, a operação do microscópio, bem como obtém os dados provenientes da placa de aquisição.

De maneira a permitir a comunicação entre o sistema de controle e a ferramenta de exibição de imagens, esta última foi compilada como uma biblioteca dinâmica (*dynamic-link library* - dll) do Windows, de forma que o acesso aos métodos implementados na classe *DBufferAFM* se dá mediante chamadas a funções que são disponibilizadas por esta.

O uso da ferramenta requer a prévia configuração das dimensões da área de varredura (em micrometros), das dimensões da imagem a ser obtida (em *pixels*), do tamanho dos vetores de dados, do modo de composição de imagens (*ida*, *volta* ou *ida e volta*), bem como das sensibilidades do fotodetetor e da correção da plataforma piezoelétrica, o que permite a correta alocação do espaço de memória necessário. É permitida a alteração destes parâmetros em tempo de execução, sendo apenas necessário alocar novamente os dados, o que é realizado de forma automática pela ferramenta. Esta operação, contudo, é transparente ao usuário, sem prejuízo ao processo de controle e aquisição de dados. Uma vez configurados os parâmetros, os dados obtidos pela placa de aquisição são enviados para a ferramenta, que realiza a composição das imagens automaticamente. A exibição das imagens é realizada através de chamadas à funções, de acordo com a conveniência, sendo possível configurar a ferramenta para realizar a exibição de forma automática, tanto 2D com 3D, após o envio de cada conjunto de dados. Finalmente, como forma de auxiliar a utilização da ferramenta, alguns códigos de erro foram especificados, sendo acessíveis em tempo de execução.

Capítulo 5

Restauração de imagens de microscopia de força atômica

5.1 Imagens digitais monocromáticas

Uma imagem digital é representada por uma função bidimensional $f(i, j)$ de intensidades discretas de energia (luz) em coordenadas espaciais discretas, na qual o valor de f para um dado par de coordenadas (i, j) corresponde à intensidade de energia de uma imagem real, representada por uma função de natureza contínua, naquela posição. Desta forma, uma imagem digital é uma representação discreta de uma função contínua, obtida através da amostragem da imagem original nas direções i e j, sendo descrita por uma matriz de $W \times H$ elementos, ou *pixels*.

Apesar de o conceito de imagens digitais se aplicar predominantemente para representar intensidades luminosas como forma de descrever imagens visíveis (fotografias), ele é largamente aplicado para representar diversas outras grandezas que podem ser des-

Figura 5.1: Exemplo de imagem digital.

critas por funções 2D contínuas, com por exemplo a temperatura ao longo uma superfície (imagens térmicas), o grau de absorção ou transmissão de raios-X (radiografias), dentre inúmeros outros exemplos possíveis, cujos valores amostrados ao longo do espaço são representados por intensidades luminosas em um plano bidimensional, para serem visualizados adequadamente.

No caso particular deste trabalho, o conceito de imagens digitais monocromáticas é aplicado para a visualização do relevo de uma área amostrada com uso de um AFM, no qual o valor ou intensidade de $f(i, j)$ representa a altura da amostra para uma dada posição (i, j), e corresponde à representação discreta de uma função contínua $z(x, y)$ que descreve o relevo em estudo.

5.2 Modelo de degradação de imagens de microscopia de força atômica

As imagens obtidas com o uso da técnica de AFM estão sujeitas aos efeitos degenerativos provenientes da instrumentação empregada durante a etapa de amostragem, de origem elétrica (sistemas eletrônicos de controle e medição) e mecânica (sistema de varredura, vibrações, entre outros), resultando em pobres relações Sinal/Ruído, bem como os efeitos de borramento provenientes da interação entre a ponteira e a superfície da amostra, seja devido às interações interatômicas como devido à geometria da ponteira, quando as dimensões desta são comparáveis às dimensões do relevo em estudo, conforme descrito na seção 2.3. Desta forma, o modelo de degradação das imagens de microscopia de força atômica pode ser descrito pelo diagrama ilustrado na figura 5.2, onde x representa a imagem real, y representa a imagem amostrada, e B e η representam, respectivamente, o operador de borramento (interação ponteira-amostra) e o ruído aditivo (instrumentação) [2].

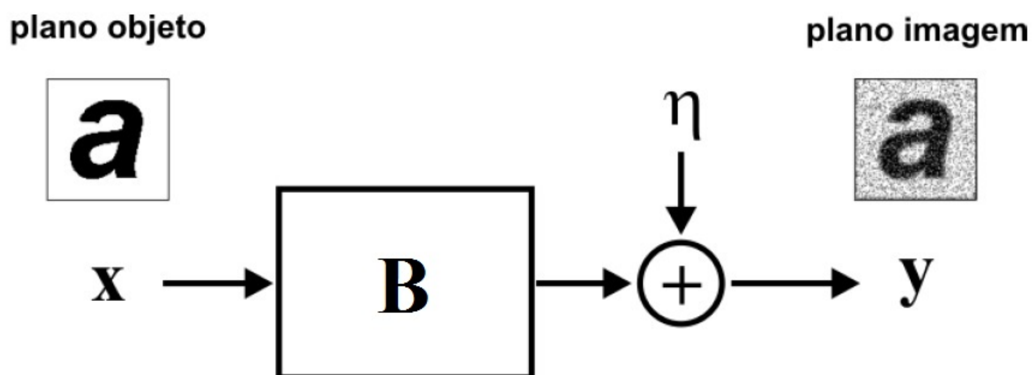

Figura 5.2: Modelo de degradação de imagens de microscopia de força atômica. Adaptado de Stutz (2009) [10]

O modelo apresentado é representado pela equação abaixo:

$$y = B * x + \eta \tag{5.1}$$

5.2.1 Convolução ponteira-amostra

O termo $B * x$ na equação 5.1 descreve uma convolução entre o operador de borramento e a imagem real, representando a ação degenerativa da interação entre a ponteira e o relevo da amostra. Para que o processo de restauração seja realizado adequadamente, o operador de borramento deve ser previamente conhecido e modelado matematicamente, o que em geral é inviável. Se por um lado a geometria da ponteira de medição é facilmente representável, este não é o caso do campo de atuação das forças de interação envolvidas, tampouco é conhecido a *priori* a porção da ponteira que de fato interage com a amostra, o que depende fortemente do relevo da mesma. Isto leva à necessidade da adoção de um modelo aproximado.

Dentre os modelos aplicáveis à técnica de AFM destacam-se as distribuições Gaussiana, exponencial, de Poisson e Rayleigh, bem como o uso de um paraboloide de revolução. Este trabalho se baseia no estudo desenvolvido por Cidade [6], no qual o modelo de superfície adotado assume uma distribuição do tipo Gaussiana, conforme a equação 5.2 abaixo, onde d é a distância euclidiana entre os pontos da superfície em relação ao centro da mesma, e σ^2 é a variância, empregada de forma a simular as relações de aspecto (altura/largura) de diferentes tipos de ponteiras. A figura 5.3 ilustra a superfície descrita pelo operador de borramento.

$$B = e^{-\frac{d^2}{\sigma^2}} \tag{5.2}$$

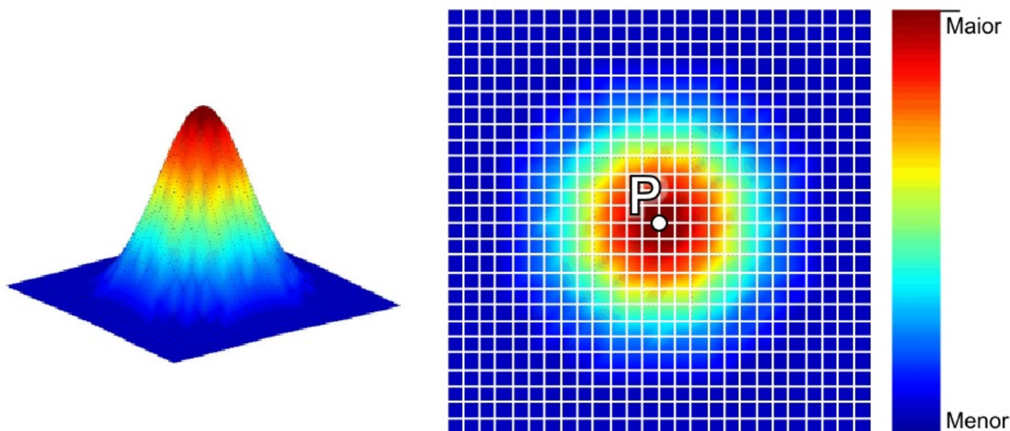

Figura 5.3: Operador de borramento do tipo Gaussiano e sua representação no plano de imagem. Adaptado de Stutz (2009) [10]

5.2.2 Convolução discreta

Uma vez que as imagens obtidas pelo AFM são digitais, sendo representadas em coordenadas discretas, o modelo de degradação das imagens pode ser melhor descrito pela equação abaixo:

$$y_{i,j} = \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot x_{i+k,j+l} + \eta_{i,j} \quad , \tag{5.3}$$

onde o termo $\eta_{i,j}$ representa a porção do ruído aditivo aplicado sobre aquele elemento, de natureza aleatória. O operador de borramento B é representado por uma matriz quadrada b, de dimensões $(2N+1)$, sendo aplicado sobre uma imagem de $M \times M$ elementos. As dimensões da matriz de borramento são equivalentes à porção da ponteira que efetivamente entra em contato com a amostra, não necessariamente representando toda a sua extensão. Uma vez que o processo de convolução não deve adicionar nem absorver energia, a matriz de borramento deve ser normalizada, de forma que o somatório de seus elementos sejam iguais à unidade. Isso é obtido através da divisão de todos os elementos da matriz pelo somatório dos elementos da mesma. A figura 5.4 ilustra uma matriz de borramento normalizada de dimensão $5x5$, com uma variância igual a 1. Os valores de cada elemento representam os pesos empregados no somatório, que decrescem à medida que se afastam do centro de aplicação do operador.

$$B = \begin{bmatrix} 2,969 \times 10^{-3} & 1,331 \times 10^{-2} & 2,194 \times 10^{-2} & 1,331 \times 10^{-2} & 2,969 \times 10^{-3} \\ 1,331 \times 10^{-2} & 5,963 \times 10^{-2} & 9,832 \times 10^{-2} & 5,963 \times 10^{-2} & 1,331 \times 10^{-2} \\ 2,194 \times 10^{-2} & 9,832 \times 10^{-2} & 1,621 \times 10^{-1} & 9,832 \times 10^{-2} & 2,194 \times 10^{-2} \\ 1,331 \times 10^{-2} & 5,963 \times 10^{-2} & 9,832 \times 10^{-2} & 5,963 \times 10^{-2} & 1,331 \times 10^{-2} \\ 2,969 \times 10^{-3} & 1,331 \times 10^{-2} & 2,194 \times 10^{-2} & 1,331 \times 10^{-2} & 2,969 \times 10^{-3} \end{bmatrix}$$

Figura 5.4: Representação matricial do operador de borramento

5.2.3 Solução do problema inverso

Desconsiderando o ruído aditivo na equação 5.3, uma solução para a obtenção da imagem original a partir da imagem degradada se daria pela determinação e aplicação da inversa da matriz B. Esta solução, contudo, se torna inviável pelo fato de a matriz de borramento ser singular, não possuindo inversa, o que torna o problema mal-condicionado [3]. O processo de obtenção numérica de uma estimativa para B^{-1} por sua vez é muito sensível a pequenas variações em y, o que se traduz em uma grande instabilidade na solução do problema [10, 30].

Uma outra solução seria a estimativa da imagem original através do método dos mínimos quadrados, considerando-se B fixo:

$$L(x) = ||y - Bx||^2 \ , \tag{5.4}$$

onde a solução se daria pela minimização do funcional $L(x)$. Contudo, ao considerar que o operador B também possa estar sujeito a perturbações, o problema é novamente considerado mal-condicionado [3]. Estas perturbações associadas ao operador de borramento podem estar relacionadas aos erros numéricos provenientes do processo iterativo de determinação da solução [31]. De fato, ainda que a imagem esteja livre de ruído aditivo, sendo degradada apenas em função do operador de borramento, os erros numéricos decorrentes do processo iterativo são responsáveis pela introdução de ruído à solução [6].

Assim sendo, uma alternativa para o problema é proposta em [6], onde um termo de regularização é empregado para o tratamento da porção ruidosa da imagem, e a solução é encontrada através da minimização do funcional de regularização de Tikhonov:

$$Q(\hat{x}) = \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left[y_{i,j} - \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot \hat{x}_{i+k,j+l} \right]^2 + \alpha S \ , \tag{5.5}$$

onde αS é o termo de regularização, sendo α o parâmetro de regularização, cujo valor está diretamente ligado à variabilidade do ruído aditivo, determinando a estabilidade do funcional Q. O parâmetro S é um funcional de regularização empregado como critério de estabilidade do processo de restauração, baseado no funcional q-discrepância [32] e na distância de Bregman [33]:

$$S = \frac{1}{1+q} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left[\hat{x}_{i,j} \left(\frac{\hat{x}_{i,j}^q - \overline{x}_{i,j}^q}{q} \right) - \overline{x}_{i,j}^q (\hat{x}_{i,j} - \overline{x}_{i,j}) \right] \ , \tag{5.6}$$

onde q é um parâmetro ajustável, a partir do qual outros dois funcionais pode ser obtidos:

1. Mínima energia ($q = 1$):

$$S = \frac{1}{2} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} (\hat{x}_{i,j} - \overline{x}_{i,j})^2 \ ; \tag{5.7}$$

2. Entropia-cruzada ($q \to 0$):

$$S = - \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left[\hat{x}_{i,j} - \overline{x}_{i,j} - \hat{x}_{i,j} \ln \frac{\hat{x}_{i,j}}{\overline{x}_{i,j}} \right] \ , \tag{5.8}$$

onde $\hat{x}_{i,j}$ representa a estimativa da solução, e $\overline{x}_{i,j}$ representa uma imagem de referência qualquer, empregada para tratar o ruído aditivo, que pode ser a própria imagem experi-

mental, um valor constante qualquer (zero, por exemplo), ou a média das intensidades dos *pixels* da imagem experimental. Em [8] foi mostrado que o uso de imagem obtida na estimativa imediatamente anterior como referência (técnica de realimentação) proporciona uma redução significativa no número de iterações necessários para que a convergência seja alcançada. As figuras 5.5 e 5.6 ilustram o processo de restauração utilizando-se uma referência fixa e a técnica de realimentação, respectivamente.

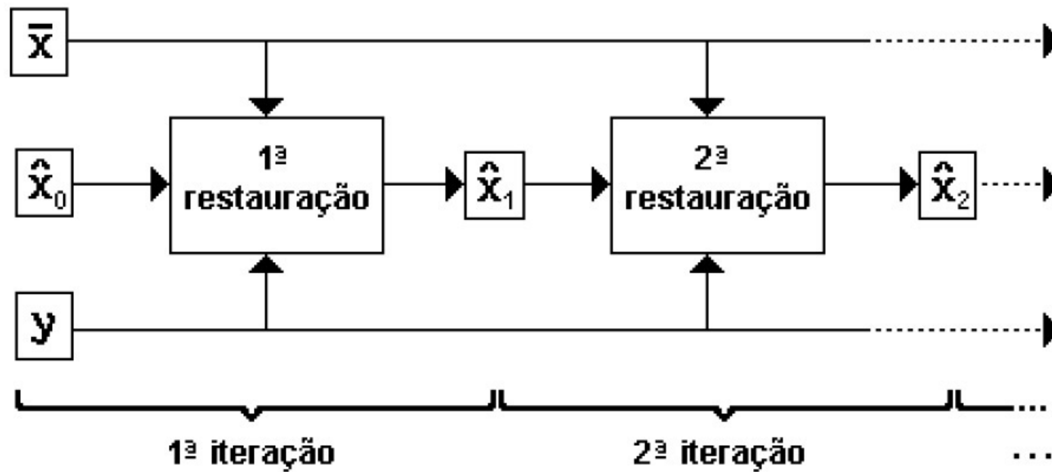

Figura 5.5: Processo iterativo com o uso de uma referência fixa. Furtado (2002) [8] e Stutz (2009) [10]

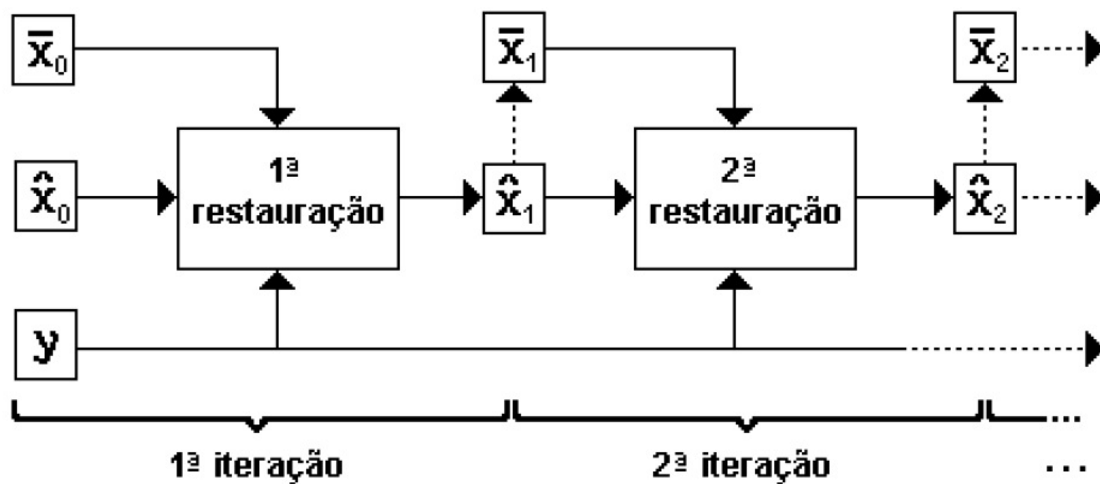

Figura 5.6: Processo iterativo com o uso da técnica de realimentação. Furtado (2002) [8] e Stutz (2009) [10]

5.3 Restauração de imagens pelo método de regularização do funcional de Tikhonov

Considerando uma matriz $M \times M$ y, representando uma imagem borrada e ruidosa, obtida experimentalmente a partir de uma imagem real x, onde o borramento aplicado a ela pode ser representado por uma matriz $N \times N$ normalizada b, que descreve a distorção decorrente da geometria da ponteira, a imagem restaurada $\hat{x}$ é assumida como sendo a melhor aproximação de x quando o funcional

$$
\begin{aligned}
Q(\hat{x}) &= \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left[y_{i,j} - \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot \hat{x}_{i+k,j+l} \right]^2 \\
&+ \alpha \frac{1}{1+q} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left[\hat{x}_{i,j} \left(\frac{\hat{x}_{i,j}^q - \overline{x}_{i,j}^q}{q} \right) - \overline{x}_{i,j}^q (\hat{x}_{i,j} - \overline{x}_{i,j}) \right] \qquad (5.9)
\end{aligned}
$$

alcança seu mínimo, onde $\alpha > 0$ é um fator ajustável de suavização, atuando como um filtro passa-baixa, de forma a assegurar a convergência do funcional. O termo S expresso ao lado de α representa uma função de contraste, atuando como um filtro passa-alta, conforme equações 5.6, 5.7 e 5.8, no qual $q > 0$ é um parâmetro ajustável. Assume-se que as imagens são quadradas apenas para simplificação do desenvolvimento, podendo este método ser aplicado a imagens de resoluções quaisquer.

O processo de restauração proposto em [6] emprega o método iterativo de Gauss-Seidel para resolver um sistema linear de $M \times M$ equações, e então obter a melhor estimativa quando o funcional $Q(\hat{x})$ alcançar seu mínimo. Apesar de eficaz, o método de Gauss-Seidel requer um grande esforço computacional para ser executado serialmente, de forma que o emprego do método iterativo de Jacobi foi proposto em [10], seguido por um simplificação deste método que tornou mais eficiente o processo de restauração. Ainda neste trabalho, o autor apresenta uma abordagem paralela utilizando a biblioteca *MPI*, aprimorando ainda mais o desempenho da aplicação, especialmente com imagens de elevadas resoluções. Baseado no método de Jacobi simplificado, uma outra abordagem paralela, utilizando a arquitetura CUDA, foi proposta em [12], resultando em ganhos de desempenho ainda maiores, mesmo com imagens de resoluções mais baixas.

De acordo com o método de Jacobi simplificado, os valores dos elementos (*pixels*) da próxima estimativa da imagem são dados por:

$$\hat{x}\big|^{t+1} = \hat{x}\big|^{t} + \gamma \cdot \Delta\hat{x}\big|^{t+1} \qquad (5.10)$$

$$\Delta\hat{x}_{r,s}\big|^{t+1} = \frac{F_{r,s}\big|^{t}}{F_{m,n}\big|^{t}_{m=r;n=s}}\,, \qquad (5.11)$$

onde t é o contador de iterações, γ é um fator arbitrário de atenuação da correção, e

$$F_{r,s} = \frac{\partial Q(\hat{x})}{\partial \hat{x}_{r,s}} = -2 \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left(y_{i,j} - \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot \hat{x}_{i+k,j+l} \right) b_{r-i,s-j} + \frac{\alpha}{q} (\hat{x}^q_{r,s} - \overline{x}^q_{r,s}) \tag{5.12}$$

e

$$F_{n,m} = \frac{\partial F_{r,s}}{\partial \hat{x}_{m,n}} = 2 \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot b_{r-m+k,s-n+l} + \alpha \hat{x}^{q-1}_{r,s} \cdot \delta(r,m) \cdot \delta(s,n) \tag{5.13}$$

são, respectivamente, a primeira e segunda derivadas de $Q(\hat{x})$, onde δ é o delta de Kronecker. É possível identificar que três convoluções envolvendo a matriz de borramento b são realizadas nas equações 5.12 e 5.13. Após cada convolução, os elementos resultantes devem ser normalizados, através da divisão destes pela soma dos elementos da matriz b efetivamente empregados na operação. Isso é necessário pois, apesar de b já ser normalizada, conforme definido anteriormente, em pontos próximos às bordas das imagens a matriz pode ultrapassar os limites da mesma, de forma que a normalização deve ser realizada novamente, conforme ilustra a figura 5.7, de forma a manter a energia inalterada mesmo utilizando-se menos elementos da matriz de borramento. O valor de normalização para cada elemento da imagem pode ser calculado conforme abaixo:

$$normaB_{r,s} = \sum_{i=max(-N,-r)}^{min(N,M-r-1)} \sum_{j=max(-N,-s)}^{min(N,M-s-1)} b_{i,j} \tag{5.14}$$

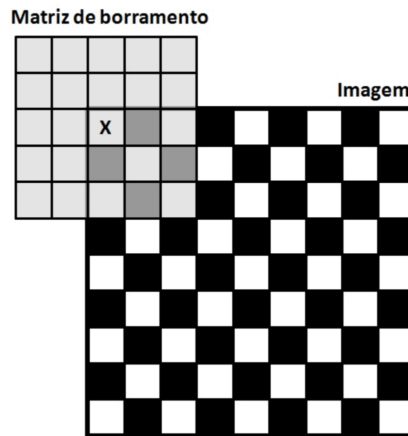

Figura 5.7: Área de atuação da matriz de borramento b centrada em um ponto (X) próximo às bordas de uma imagem hipotética.

5.4 Algoritmo de restauração

O objetivo principal deste trabalho é a obtenção de um algoritmo otimizado de restauração paralela executada em GPGPU usando utilizando a arquitetura CUDA, levando em consideração as abordagens empregadas em [6], [10] e [12]. A primeira providência neste sentido foi identificar as variáveis e/ou matrizes que podiam ser definidas (calculadas) apenas uma vez, na inicialização da aplicação. Este é o caso da matriz de borramento b e os valores de normalização a serem aplicados a cada elemento da imagem após cada convolução. Estes são armazenados em uma matriz $M \times M$ $normaB$, construída conforme a equação 5.14.

A equação 5.13, que define a segunda derivada de $Q(\hat{x})$, apresenta em seu interior uma convolução da matriz b sobre ela mesma, o que também pode ser calculado e armazenado apenas uma vez. No entanto, a equação 5.11 restringe a aplicação da $F_{n,m}$ para os casos nos quais $(m = r)$ e $(n = s)$. Tais restrições simplificam a equação 5.13 para:

$$F_{n,m} = 2 \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot b_{k,l} + \alpha \hat{x}_{r,s}^{q-1} \ , \tag{5.15}$$

onde os somatórios não mais representam uma convolução de b sobre ela mesma, mas sim um somatório dos quadrados de seus termos, o que também pode ser calculado na inicialização, sendo os resultados armazenados em outra matriz $M \times M$, uma vez que depende apenas da matriz de borramento e a posição do elemento na imagem, assim como $normaB$. É importante observar que a normalização deve ser considerada também nesta etapa, de forma a manter a energia inalterada. Empregando a matriz:

$$S2BB_{r,s} = 2 \sum_{k=max(-N,-r)}^{min(N,M-r-1)} \sum_{l=max(-N,-s)}^{min(N,M-s-1)} \left(\frac{b_{k,l}}{normaB_{r,s}} \right)^2 \ , \tag{5.16}$$

a equação 5.15 pode ser reescrita como:

$$F_{n,m} = S2BB_{r,s} + \alpha \hat{x}_{r,s}^{q-1} \quad (r = m, s = n) \tag{5.17}$$

Pode ser observado que, assim como ocorre com a matriz $normaB$, os elementos da matriz $S2BB$ mais afastados das bordas, a uma distância maior que a metade da matriz de borramento, apresentam os mesmos valores. Estes elementos correspondem, na maioria dos casos, a maior parte da imagem, e isso ocorre porque as dimensões da matriz de borramento são geralmente muito menores que as dimensões da imagem. Os demais elementos sofrem os efeitos da área de atuação da matriz de borramento em regiões próximas às bordas, conforme ilustrado na figura 5.7. Uma abordagem empregada nos algoritmos implementados em [10] e [12] para aprimorar o desempenho da aplicação foi o uso de vetores, cujas posições correspondem à soma das distâncias nas direções horizontal e ver-

tical que ultrapassam os limites da imagem, sendo os valores em cada posição iguais aos valores de normalização correspondentes a estas somas de distâncias. Para $S2BB$, por sua vez, os autores empregaram um único valor para todos os elementos. Esta abordagem, contudo, resultou em artefatos nas regiões próximas às bordas das imagens restauradas quando as dimensões da matriz de borramento aumentam. Em razão disso, optou-se por utilizar matrizes neste trabalho, uma vez que o desempenho não é prejudicado, pois as matrizes são calculadas apenas uma vez, e os resultados são mais precisos por calcular os valores para cada pixel individualmente, mesmo que isso resulte em um maior uso do espaço de memória. A diferença resultante desta abordagem, em comparação com o uso dos vetores, será apresentada no capítulo 6.

O termo dentro do primeiro somatório na equação 5.9 representa a diferença entre a imagem experimental e a convolução da matriz de borramento sobre a estimativa da imagem real. Uma vez que a mesma expressão é empregada na equação 5.12, o resultado desta operação foi armazenado em uma matriz de forma a otimizar o desempenho da aplicação. Desta forma, considerando também a normalização da convolução de b sobre $\hat{x}$, a matriz YBX é definida como:

$$YBX_{i,j} = y_{i,j} - \frac{\sum_{k=max(-N,-i)}^{min(N,M-i-1)} \sum_{l=max(-N,-j)}^{min(N,M-j-1)} b_{k,l} \cdot \hat{x}_{i+k,j+l}}{normaB_{r,s}} , \tag{5.18}$$

de forma que a equação 5.9 pode ser reescrita como:

$$Q(\hat{x}) = \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} YBX_{i,j}{}^2 + \alpha S , \tag{5.19}$$

onde S é dado pela equação 5.6 e seus casos particulares. A partir da equação 5.12, pode ser inferido que o somatório somente resultará em valores diferentes de zero quando as diferenças $(r-i)$ e $(s-j)$ estiverem dentro dos limites da matriz de borramento, o que significa que os limites dos somatórios podem ser reduzidos para $[-N \dots N]$. Desta forma, a expressão resultante para $F_{r,s}$ é dada por:

$$F_{r,s} = -2 \frac{\sum_{k=max(-N,-r)}^{min(N,M-r-1)} \sum_{l=max(-N,-s)}^{min(N,M-s-1)} b_{k,l} \cdot YBX_{r+k,s+l}}{normaB_{r,s}} + \frac{\alpha}{q} (\hat{x}^q_{r,s} - \overline{x}^q_{r,s}) \tag{5.20}$$

O termo dentro do somatório do funcional S dado em 5.6, e empregado em 5.9, pode ser armazenado em outra matriz, denominada MS, dada por:

$$MS_{i,j} = \hat{x}_{i,j} \left(\frac{\hat{x}^q_{i,j} - \overline{x}^q_{i,j}}{q} \right) - \overline{x}^q_{i,j} (\hat{x}_{i,j} - \overline{x}_{i,j}) , \tag{5.21}$$

a qual, para os casos particulares ($q = 1$ e $q \to 0$), pode ser reescrita como

$$MS_{i,j} = (\hat{x}_{i,j} - \overline{x}_{i,j})^2 \,, \qquad (q = 1) \tag{5.22}$$

$$MS_{i,j} = -\left[\hat{x}_{i,j} - \overline{x}_{i,j} - \hat{x}_{i,j} \ln \frac{\hat{x}_{i,j}}{\overline{x}_{i,j}}\right] \,, \qquad (q \to 0) \,, \tag{5.23}$$

de forma que a equação 5.6 pode finalmente ser reescrita como:

$$S = \frac{1}{q+1} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} MS_{i,j} \tag{5.24}$$

Os somatórios dos elementos de MS e dos quadrados dos elementos de YBX são utilizados para calcular S e então $Q(\hat{x})$. Uma vez identificadas as operações que podem ser realizadas apenas uma vez e aquelas cujos resultados podem ser reaproveitados, o passo seguinte foi identificar as operações que podem ser realizadas em conjunto, de forma a tirar proveito dos ciclos de iterações. Neste sentido, dois loops foram identificados. O primeiro realiza a convolução da imagem estimada com a matriz de borramento, calcula as diferenças do resultado da convolução com y, armazenando em YBX para a próxima iteração, os termos de MS, e finalmente calcula os somatórios dos elementos de ambas as matrizes para os cálculos de S e $Q(\hat{x})$. O segundo loop, por sua vez, realiza o cálculo das derivadas primeira e segunda de $Q(\hat{x})$, $F_{r,s}$ e $F_{m,n}$, o que permite a obtenção da nova estimativa da imagem restaurada.

5.4.1 Algoritmo serial

O resultado desta análise foi uma otimização do algoritmo serial de restauração empregando o método de Jacobi simplificado descrito em [10], apresentado no algoritmo 5.1. Neste, em adição à verificação de convergência dos valores de $Q(\hat{x})$, as diferenças absolutas ou relativas entre os valores obtidos podem ser avaliadas e comparadas com um valor de tolerância, uma vez que a convergência em alguns casos pode ser muito lenta, requerendo muitos ciclos de iteração. Usualmente, um número fixo de iterações é fixado (algo entre 50 e 100), e o valores de α, q e γ são ajustados de forma a acelerar a convergência e/ou produzir resultados melhores.

É possível observar no algoritmo que inicialmente tanto a imagem de referência $\overline{x}$ como a estimativa de imagem restaurada $\hat{x}$ são assumidos como sendo iguais à imagem experimental y, passando a ser definidos como a imagem recém-estimada nas demais iterações, conforme o método de realimentação. Além disso, a dependência de dados decorrente da necessidade de se realizar previamente a convolução de b sobre $\hat{x}$ para o cálculo de YBX para então realizar o cálculo de $\hat{x}$, bem como a dependência decorrente da necessidade de se calcular $\hat{x}$ para então se poder calcular YBX impedem que todas as

Algoritmo 5.1: Restauração serial de imagens por regularização de Tikhonov

Entrada: $y[]$, dim_b, σ^2, α, q, γ, N

Saída: $\hat{x}[]$, $Q(\hat{x})$

Inicialização:

1: Calcula b (eq. 5.2)
2: Calcula $normaB$ (eq. 5.14)
3: Calcula $S2BB$ (eq. 5.16)
4: $\hat{x} \leftarrow y$
5: $\overline{x} \leftarrow y$

Cálculo inicial de $Q(\hat{x})$:

6: **para** $i = 0 \rightarrow (M-1); j = 0 \rightarrow (M-1)$
7: Calcula $YBX_{i,j}$ (eq. 5.18)
8: Calcula $MS_{i,j}$ (eq. 5.21)
9: $somaYBX2 \leftarrow somaYBX2 + {YBX_{i,j}}^2$
10: $somaMS \leftarrow somaMS + MS_{i,j}$
11: **fim para**
12: Calcula S (eq. 5.24)
13: Calcula $Q_0(\hat{x})$ (eq. 5.19)

Loop principal:

14: **para** $t = 0 \rightarrow (N-1)$

Calcula nova estimativa:

15: **para** $r = 0 \rightarrow (M-1); s = 0 \rightarrow (M-1)$
16: Calcula $F_{r,s}$ (eq. 5.20)
17: Calcula $F_{n,m}$ (eq. 5.17)
18: Calcula $\hat{x}_{r,s}$ (eq. 5.10)
19: **fim para**

Calcula o novo valor de $Q(\hat{x})$:

20: **para** $i = 0 \rightarrow (M-1); j = 0 \rightarrow (M-1)$
21: Calcula $YBX_{i,j}$ (eq. 5.18)
22: Calcula $MS_{i,j}$ (eq. 5.21)
23: $somaYBX2 \leftarrow somaYBX2 + {YBX_{i,j}}^2$
24: $somaMS \leftarrow somaMS + MS_{i,j}$
25: **fim para**
26: Calcula S (eq. 5.24)
27: Calcula $Q_1(\hat{x})$ (eq. 5.19)

Verifica convergência:

28: **se** $Q_1(\hat{x}) > Q_0(\hat{x})$ **então interrompa**

Atribui valores para a próxima iteração:

29: $\overline{x} \leftarrow \hat{x}$
30: $Q_0(\hat{x}) \leftarrow Q_1(\hat{x})$
31: **fim para**
32: **retorna**

---

operações sejam realizadas dentro do mesmo loop.

5.4.2 Algoritmo paralelo

Ao analisar atentamente o algoritmo 5.1, foi possível observar que após as inicializações, praticamente todas as etapas podem ser executadas em paralelo, à exceção daquelas descritas entre as linhas 9 e 13, bem como entre as linhas 23 e 27, onde os somatórios dos quadrados dos elementos de YBX e dos elementos de MS são realizados para o cálculo de S e $Q(\hat{x})$. Estas etapas devem ser executadas serialmente, em função das variáveis $somaYBX2$ e $somaMS$, uma vez que o acesso concorrente a estas prejudicaria significativamente o desempenho da abordagem paralela implementada neste trabalho. As demais etapas, por outro lado, correspondem a operações realizadas sobre *pixels* individualmente, o que as tornam excelentes candidatas à paralelização, sendo cada *thread* responsável pela computação sobre um *pixel*.

O algoritmo paralelo proposto foi desenvolvido para mover toda a informação necessária à execução para a memória global da GPU na inicialização, minimizando assim o tráfego de dados entre a CPU e a GPU durante a execução de seu loop principal, o que poderia comprometer criticamente o desempenho da aplicação. Em razão da dependência de dados descrita em 5.4.1, a execução paralela é realizada por dois *kernels*.

O primeiro *kernel* calcula a convolução da matriz de borramento b sobre $\hat{x}$ e os elementos das matrizes YBX e MS correspondentes à *thread* em execução, de forma a permitir o cálculo de S e $Q(\hat{x})$, e corresponde aos trechos do algoritmo 5.1 entre as linhas 6 e 8, e entre as linhas 20 e 22. A estrutura do *kernel* paralelo `conv_dif` é apresentada de maneira simplificada no algoritmo abaixo:

Algoritmo 5.2: Implementação simplificada do *kernel* `conv_dif`

Entrada: y[], $\hat{x}$[], $\overline{x}$[], b[], $normaB$[], α, q
Saída: YBX[], MS[]

Inicialização:
1: i ← Id_thread
2: j ← Id_bloco
Efetua cálculos:
3: Calcula $YBX_{i,j}$ (eq. 5.18)
4: Calcula $MS_{i,j}$ (eq. 5.21)
5: **retorna**

O segundo *kernel* realiza os cálculos dos elementos correspondentes à *thread* em execução das derivadas primeira e segunda, $F_{r,s}$ e $F_{n,m}$, da correção $\Delta\hat{x}$, aplicando-a sobre $\hat{x}$ para obter a nova estimativa de imagem restaurada, e corresponde aos trechos do algoritmo 5.1 entre as linhas 15 e 19. A estrutura do *kernel* `aplica_deltaX` é também apresentada de maneira simplificada no algoritmo que segue.

Uma definidos ambos os *kernels* `conv_dif` e `aplica_deltaX`, o processo de restauração paralela proposto é apresentado no algoritmo 5.4. A principal característica

Algoritmo 5.3: Implementação simplificada do *kernel* `aplica_deltaX`

Entrada: $\hat{x}^t[]$, $\overline{x}[]$, $b[]$, $normaB[]$, $S2BB[]$, α, q, γ

Saída: $\hat{x}^{t+1}[]$

Inicialização:

1: r $\leftarrow$ Id_thread

2: s $\leftarrow$ Id_bloco

Efetua cálculos:

3: Calcula $F_{r,s}$ (eq. 5.20)

4: Calcula $F_{n,m}$ (eq. 5.17)

5: Calcula $\hat{x}_{r,s}$ (eq. 5.10)

6: **retorna**

deste algoritmo é transferir, em sua inicialização, todas as informações necessárias à sua execução para a memória global da GPU, de forma a evitar transferências de dados excessivas, o que pode comprometer drasticamente o desempenho da execução. Os passos sublinhados correspondem às transferências entre CPU e GPU e, dentro do loop principal, se limitam às buscas das matrizes YBX e MS usadas para o cálculo serial de $Q(\hat{x})$. Estas matrizes são alocadas na memória principal como espaço não-paginável (*page-locked memory*), de forma a reduzir o tempo de transferência, conforme descrito em 3.3.

A estrutura de blocos e *threads* de ambos os *kernels* foi estabelecida para computar os resultados de uma linha por bloco e um elemento (*pixel*) por *thread*. Ainda que os dados estejam alocados na memória global, esta estrutura favorece o acesso usando o *cache* da GPU, o que oculta os elevados tempos de acesso em comparação com níveis mais baixos de memória. Isso é possível em razão de as *threads* acessarem a memória global em posições sequenciais, respeitando a localidade dos dados e permitindo o acesso *coalescente*, conforme descrito em 3.4.2.

A complexidade do algoritmo paralelo proposto é $O(n^3)$, em razão da substituição dos dois loops secundários pelos *kernels*. De fato, considerando-se o loop principal ($O(n)$), ambos os *kernels* ($O(n^2)$), os somatórios ($O(n^2)$) e as demais operações ($O(1)$), tem-se $O(n \times (n^2 + n^2 + n^2 + 1 + 1 + 1 + 1 + 1)) = O(n^3)$. Entretanto, deve-se levar em consideração que apesar de os *kernels* possuírem complexidade $O(n^2)$, esta é devida aos loops de convoluções utilizando a matriz de borramento, cujas dimensões são em geral consideravelmente menores que as dimensões das imagens, o que torna o custo computacional destas operações pequeno, apesar da complexidade. Além disso, dada a natureza e quantidade das operações matemáticas executadas nos trechos paralelizados, pode-se se dizer que a maioria das operações executadas no algoritmo serial foram paralelizadas com sucesso, restando apenas somatórios e verificações para serem executados serialmente.

Finalmente, apesar de a verificação de convergência ser uma etapa necessária para garantir a correição dos resultados, ela pode ser evitada em aplicações onde são realizadas restaurações em tempo real, com um número fixo e reduzido de iterações, um dos

Algoritmo 5.4: Restauração paralela de imagens por regularização de Tikhonov

Entrada: $y[]$, dim_b, σ^2, α, q, γ, N

Saída: $\hat{x}[]$, $Q(\hat{x})$

Inicialização:

1: Calcula b (eq. 5.2)

2: Calcula $normaB$ (eq. 5.14)

3: Calcula $S2BB$ (eq. 5.16)

4: $\hat{x} \leftarrow y$

5: $\overline{x} \leftarrow y$

6: Envia y, $\hat{x}$, $\overline{x}$, b, $normaB$ e $S2BB$ para a GPU

Cálculo inicial de $Q(\hat{x})$:

7: Executa `conv_dif` (alg. 5.2)

8: Busca YBX e MS da GPU

9: **para** $i = 0 \rightarrow (M-1); j = 0 \rightarrow (M-1)$

10: $somaYBX2 \leftarrow somaYBX2 + {YBX_{i,j}}^2$

11: $somaMS \leftarrow somaMS + MS_{i,j}$

12: **fim para**

13: Calcula S (eq. 5.24)

14: Calcula $Q_0(\hat{x})$ (eq. 5.19)

Loop principal:

15: **para** $t = 0 \rightarrow (N-1)$

Calcula nova estimativa:

16: Executa `aplica_deltaX` (alg. 5.3)

Calcula o novo valor de $Q(\hat{x})$:

17: Executa `conv_dif` (alg. 5.2)

18: Busca YBX e MS da GPU

19: **para** $i = 0 \rightarrow (M-1); j = 0 \rightarrow (M-1)$

20: $somaYBX2 \leftarrow somaYBX2 + {YBX_{i,j}}^2$

21: $somaMS \leftarrow somaMS + MS_{i,j}$

22: **fim para**

23: Calcula S (eq. 5.24)

24: Calcula $Q_1(\hat{x})$ (eq. 5.19)

Verifica convergência:

25: **se** $Q_1(\hat{x}) > Q_0(\hat{x})$ **então interrompa**

Atribui valores para a próxima iteração:

26: $\overline{x} \leftarrow \hat{x}$

27: $Q_0(\hat{x}) \leftarrow Q_1(\hat{x})$

28: **fim para**

29: Busca $\hat{x}$ da GPU

30: **retorna**

---

focos principais deste trabalho. Nestes casos, o resultado da restauração é exibido imediatamente, e o usuário define em tempo de execução os parâmetros de restauração que resultam nos melhores resultados. O capítulo 6 apresenta os resultados da avaliação do desempenho dos algoritmos nos casos em que a verificação de convergência é aplicada ou não.

Capítulo 6

Resultados e discussão

Os objetivos deste trabalho foram fornecer ferramentas para composição, exibição e restauração de imagens obtidas por microscopia de força atômica com o melhor desempenho possível, de forma a permitir o emprego destas em aplicações de exibição de resultados em tempo real, sem deixar de lado, contudo, a qualidade da imagens apresentadas. Desta forma foram avaliados tanto o desempenho como a qualidade das aplicações desenvolvidas.

O desenvolvimento do trabalho se deu através do desenvolvimento em paralelo de duas ferramentas, que em seu estado atual ainda não operam em conjunto. Isso se deve ao fato de que enquanto que o programa de restauração é facilmente executável em qualquer computador que possua uma placa gráfica compatível com a tecnologia CUDA, o que é relativamente simples de se encontrar no mercado, a ferramenta de exibição de imagens foi desenvolvida como um objetivo secundário, para uma aplicação muito mais específica, não necessariamente sendo aplicável a todas as tecnologias de microscopia de força atômica disponíveis. Não obstante, a integração das ferramentas é perfeitamente viável, e é apresentada como uma das propostas de trabalhos futuros, dentre outras.

Todos os testes de execução foram realizados em notebook dotado de um processador Intel i7 de segunda geração com 2,20 GHz de *clock*, 6 GB de memória RAM DDR3, e uma GPU GeForce GT555M com 2 GB de memória e 144 núcleos com 1,05 GHz de *clock*, operando com o *driver* CUDA versão 5.0. Todos os programas foram escritos para serem executados em Windows 7, em linguagem C/C++, no ambiente de desenvolvimento Visual Studio 2012 (ou adaptados para este ambiente, no caso das versões desenvolvidas em trabalhos anteriores), sendo a dll responsável pela integração da ferramenta com o programa de controle do AFM descrito no capítulo 4 escrita em linguagem C# no mesmo ambiente de desenvolvimento.

Os programas executados serialmente foram escritos usando variáveis de ponto flutuante de precisão dupla (*double*), uma vez que foi observado que eles são executados mais rapidamente nestas condições em arquiteturas de 64 bits, e utilizando a opção -O2 como parâmetro de compilação. Os algoritmos paralelos, por sua vez, foram escritos utilizando

variáveis de ponto flutuante de precisão simples (*float*).

Todos os tempos de execução apresentados nas avaliações de desempenho correspondem a médias aritméticas de 10 execuções alternadas, como medida de compensar eventuais flutuações no desempenho global do sistema durante os testes. Por exemplo, ao avaliar dois algoritmos, executa-se o primeiro e o segundo em sequência, repetindo esse processo 10 vezes para computar as médias.

6.1 Ferramenta de exibição de imagens

Uma vez que a configuração do microscópio de força atômica em uso no IBCCF atualmente produz apenas imagens de 128x128 ou 256x256 *pixels*, como o exemplo da figura 6.1, foi necessário o uso de diferentes conjuntos de dados representando diferentes imagens de teste criadas especificamente para o propósito de avaliar adequadamente o desempenho da ferramenta de exibição de imagens para dimensões maiores que as disponíveis. A figura 6.2 exibe a visualização de uma imagem artificial de 256x256 *pixels*.

A avaliação da ferramenta consistiu em comparar os tempos de execução da mesma quando executada serialmente e em paralelo, através de CUDA. A aplicação paralela foi

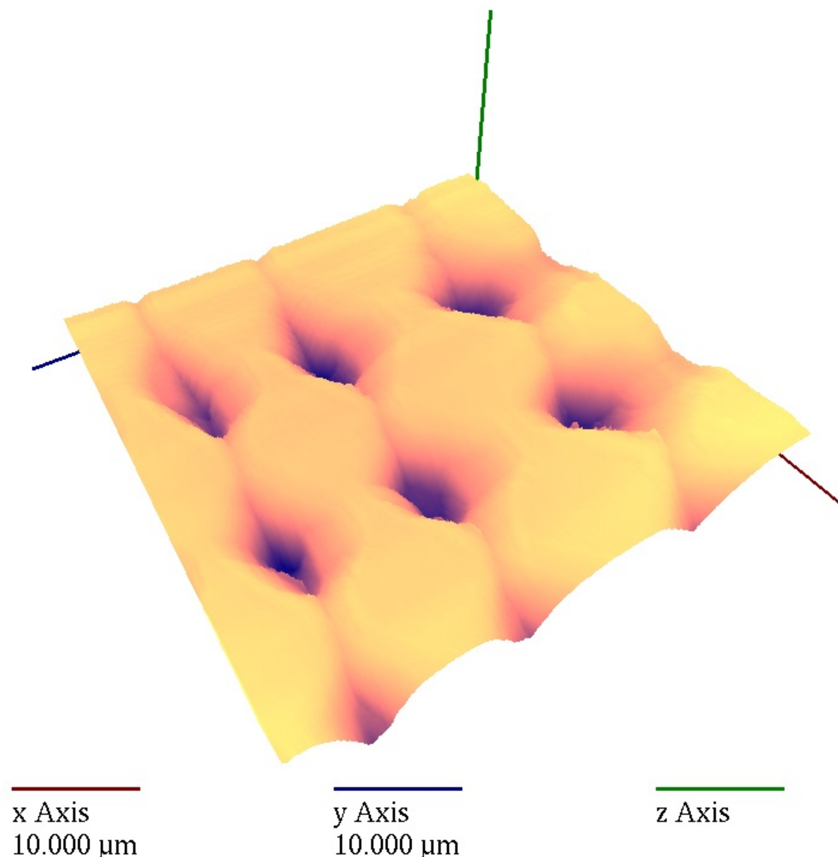

Figura 6.1: Exemplo de visualização 3D de uma imagem com 256x256 *pixels* de uma grade de teste obtida com o uso de um AFM.

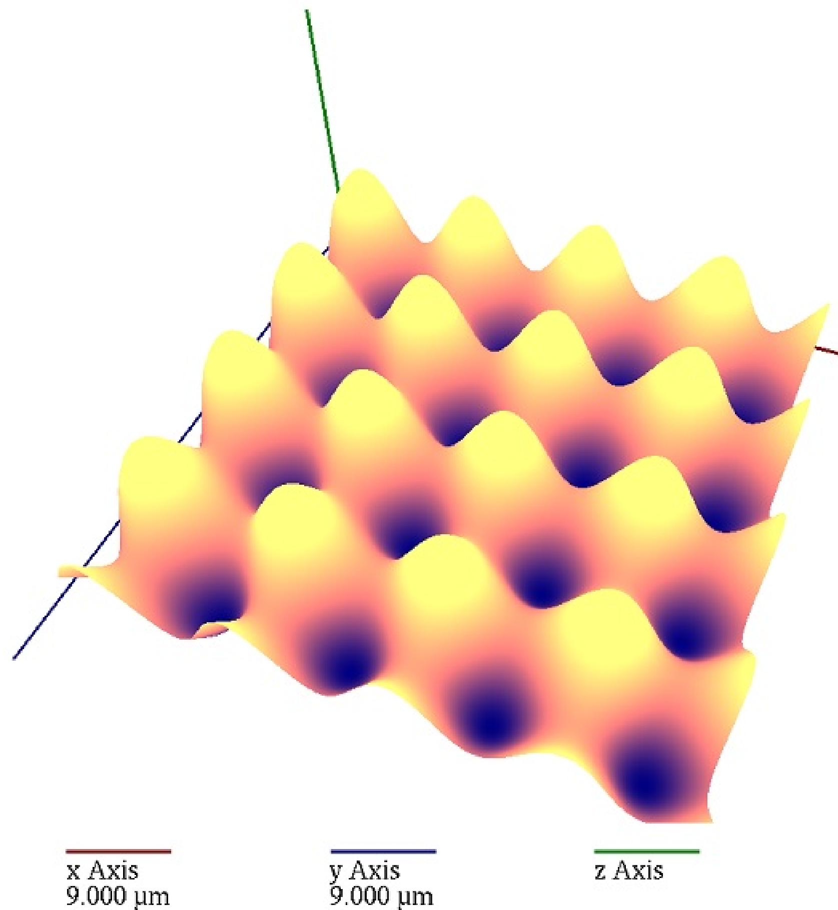

Figura 6.2: Visualização 3D de um conjunto de dados representando uma imagem artificialmente criada com 256x256 *pixels*.

avaliada em diferentes versões, onde cada uma delas apresenta uma metodologia distinta de alocação e acesso à memória principal do sistema, a saber, a alocação tradicional da biblioteca C (*malloc*), a alocação de memória não-paginável (*page-locked memory*), e o mapeamento da memória não-paginável diretamente pela GPU (*zero-copy*). Os gráficos das figuras 6.3 e 6.4 exibem os tempos de processamento de 500 conjuntos de dados, o que corresponde à exibição de 500 imagens, para cada uma das abordagens empregadas, enquanto o gráfico da figura 6.5 exibe os ganhos, ou acelerações (*speedups*), de cada abordagem paralela, em comparação com a abordagem serial. O ganho é definido como:

$$g = \frac{t_r}{t_a}, \tag{6.1}$$

onde t_r e t_a são respectivamente os tempos de execução observados em um programa usado como referência e os tempos do programa sob análise. Neste trabalho, os ganhos serão sempre apresentados tomando-se como referência o programa executado serialmente em CPU.

É possível observar nas imagens que, para imagens de 128x128 *pixels*, o desempenho

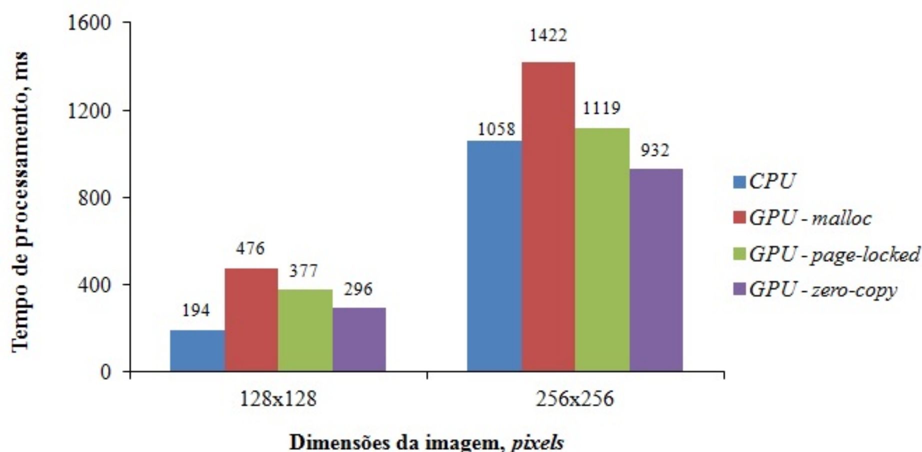

Figura 6.3: Tempos de execução da ferramenta de exibição para imagens de 128x128 e 256x256 *pixels*.

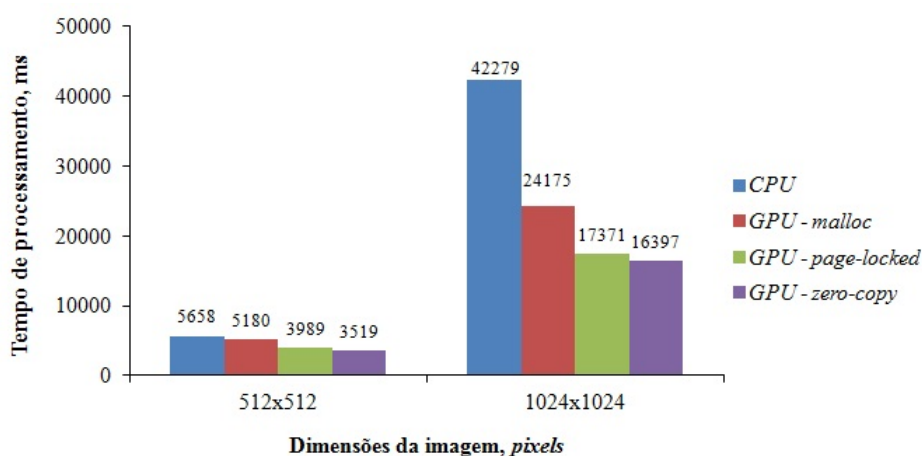

Figura 6.4: Tempos de execução da ferramenta de exibição para imagens de 512x512 e 1024x1024 *pixels*.

da abordagem serial é superior ao das demais abordagens, em virtude do elevado custo das chamadas à *API* CUDA para transferências de pequenos conjuntos de dados, bem como para a execução dos três *kernels*, em comparação com as computação executada. O gráfico da figura 6.6 ilustra o impacto do tráfego de dados CPU↔GPU no tempo total de processamento. Além disso, para imagens pequenas, o desempenho da CPU é favorecido pelo uso do *cache* da mesma. Conforme análise feita no capítulo 4, já era esperado que a pequena quantidade de operações realizadas em paralelo por cada *thread* dos *kernels* poderia ser um obstáculo à paralelização da aplicação, especialmente para imagens de dimensões reduzidas.

À medida que as dimensões das imagens aumentam, por outro lado, o desempenho das abordagens paralelas vão, uma a uma, se mostrando superiores ao desempenho da abordagem serial. Ao analisar o gráfico da figura 6.6, pode-se inferir que à medida que o tráfego de dados deixa de impactar significativamente no tempo total de processamento

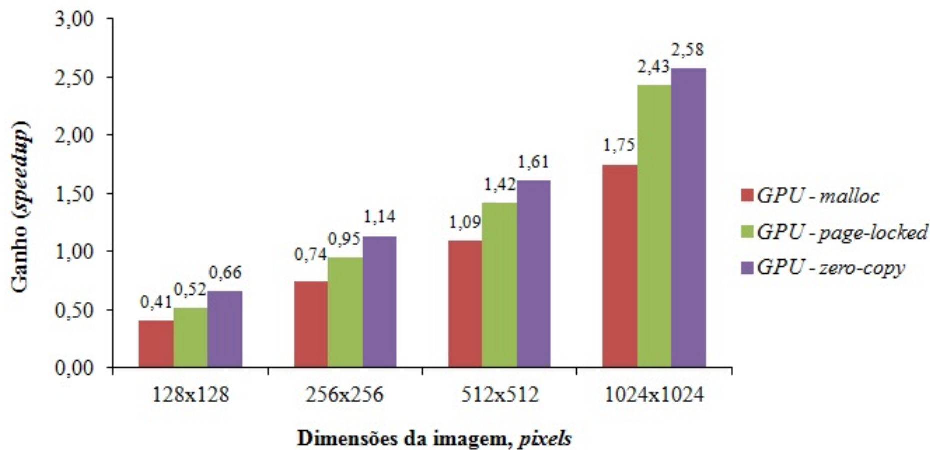

Figura 6.5: Ganhos de desempenho (*speedups*) decorrentes da paralelização da ferramenta de exibição.

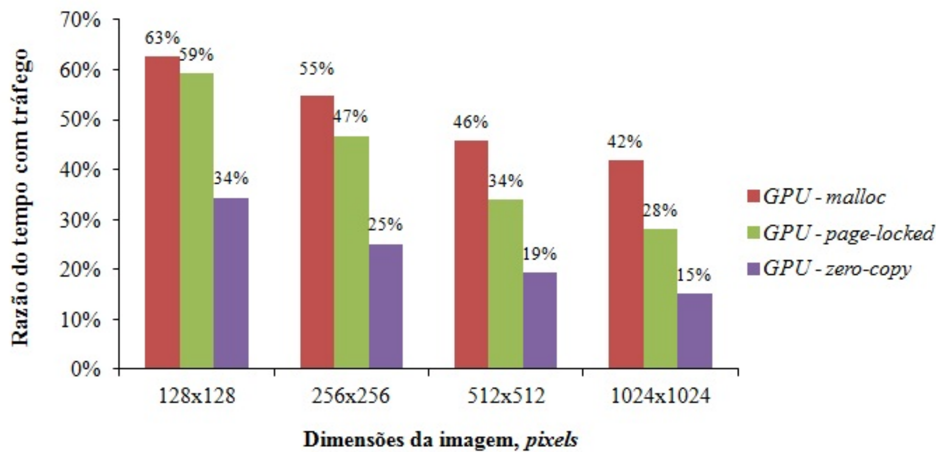

Figura 6.6: Razão percentual do tempo consumido com tráfego de dados nas abordagens paralelas.

da aplicação CUDA o desempenho da mesma aumenta.

Apesar de, dentre as abordagens paralelas empregadas, a que apresentou o melhor desempenho foi aquela que emprega o mapeamento de memória diretamente pela GPU, esta prática deve ser adotada com cautela. A técnica de *zero-copy* é empregada com sucesso em casos particulares, onde as leituras/escritas são realizadas com pouca frequência, em comparação com a computação efetuada, devido ao custo elevado de se realizar estas operações. Além disso ela é mais indicada para sistemas onde a GPU é integrada, usualmente compartilhando a memória com a CPU [21], o que é o caso do sistema empregado nesta avaliação. Nos demais casos, a abordagem usando memória não paginável poderia a mais indicada. De fato, apenas os vetores contendo os dados do fotodetetor e da correção em z, bem como suas respectivas imagens parciais, foram alocados através da técnica de *zero-copy*, sendo os demais dados alocados como memória não-paginável, uma vez que esta abordagem foi a que proporcionou os melhores resultados.

6.2 Algoritmos de restauração de imagens

De maneira a demonstrar os ganhos de desempenho obtidos com o uso dos algoritmos desenvolvidos, estes foram comparados com outros dois implementados anteriormente. O primeiro foi empregado em [10], e trata-se de um algoritmo serial empregado Jacobi simplificado, e foi utilizado para a avaliação da paralelização implementada utilizando a biblioteca *MPI*, sendo aqui denominado *serial #1*. O segundo, desenvolvido em [12], usou o mesmo algoritmo serial como parâmetro de avaliação de ganhos, implementando-o como uma versão paralela utilizando CUDA, sendo aqui denominado *paralelo #1*. Cabe observar que o algoritmo paralelo de Stutz, que emprega a biblioteca *MPI*, não foi avaliado neste trabalho.

O algoritmos avaliados são a versão serial otimizada do algoritmo de restauração utilizando o método de Jacobi simplificado, conforme apresentado em 5.4.1, sendo denominado *serial #2*, sua paralelização em GPU utilizando CUDA, conforme apresentado em 5.4.2, denominado *paralelo #2*, bem como este último algoritmo com as verificações de convergência suprimidas, sendo este denominado *paralelo #2 - sem verificação*. Optou-se por esta denominação para este último algoritmo pelo fato de os *kernels* implementados serem rigorosamente idênticos.

Da maneira similar à avaliação da biblioteca de exibição, foi necessário o uso de imagens de teste artificiais para permitir a avaliação dos algoritmos para uma faixa razoável de resoluções de imagens. A imagem de teste é apresentada na figura 6.7. Sobre estas imagens foram aplicadas simulações dos efeitos degenerativos, sob a forma de um operador de borramento de dimensões D e variância v, simulando o efeito da interação da ponteira de medição, além da de ruído aditivo simulando os efeitos decorrentes da instrumentação empregada, obtido através de um gerador de valores pseudo-aleatórios, de forma que o valor a ser adicionado respeitasse uma relação Sinal/Ruído s expressa em decibéis (dB), onde assume-se não haver correlação entre o borramento e o ruído aditivo.

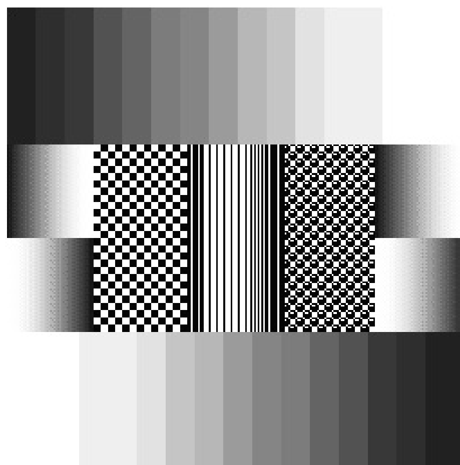

Figura 6.7: Imagem de teste.

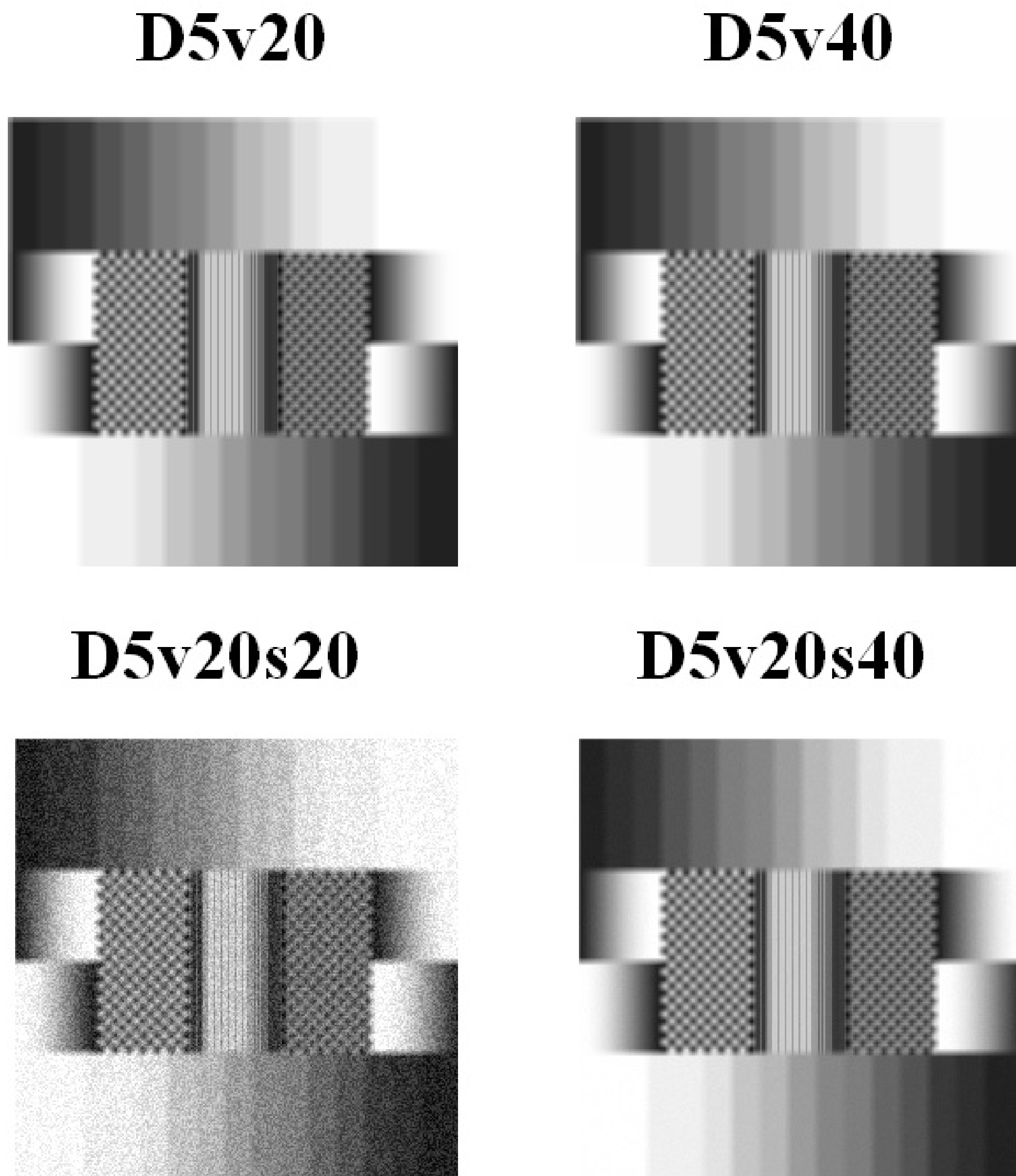

Figura 6.8: Imagens borradas utilizadas na avaliação dos algoritmos.

Como forma de identificar adequadamente as imagens modificadas, uma nomenclatura foi proposta em [8], de forma que cada imagem degradada será nomeada conforme a sintaxe $D[val_D]v[val_v]s[val_s]$, onde $[val_x]$ representa os valores adotados para cada parâmetro. A imagens modificadas são apresentadas na figura 6.8. Para a avaliação dos algoritmos foram utilizadas imagens com e sem a presença de ruído aditivo.

6.2.1 Desempenho

Para a avaliação do desempenho dos algoritmos, as imagens de teste originais de 128x128, 256x256, 512x512 e 1024x1024 *pixels* foram borradas empregando-se um operador b de dimensões 5x5 e variância igual a 20, sem a presença de ruído aditivo, para

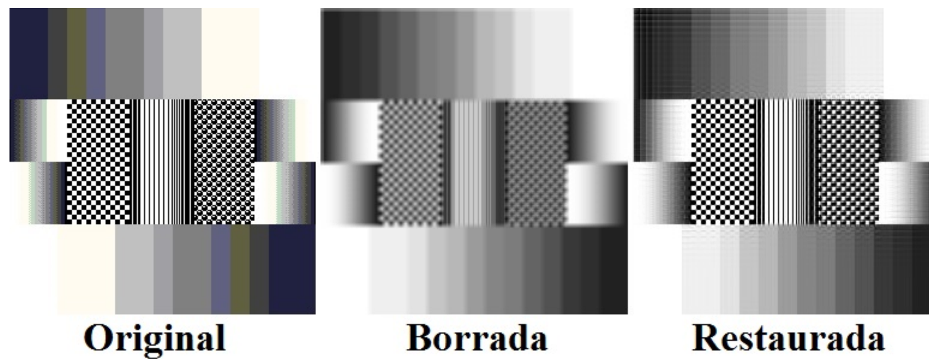

Figura 6.9: Resultado da restauração de uma imagem de 256x256 *pixels*.

então serem restauradas por cada um dos algoritmos com um número fixo de iterações, utilizando-se os mesmos parâmetros, sendo computados os tempos de processamento de cada um. De fato, a natureza do processo iterativo de restauração é tal que o desempenho deste independe das características da imagem a ser restaurada, à exceção do número de iterações necessários para a solução convergir, se houver convergência, o que depende das imagens e dos parâmetros empregados na restauração (matriz de borramento b, α, q e γ). A figura 6.9 ilustra o resultado de uma restauração utilizando o algoritmo paralelo proposto neste trabalho.

Para todas as restaurações, a matriz de borramento empregada foi a mesma, com dimensões 5x5, variância 20, e os parâmetros α, q e γ foram ajustados para 1 em todos os casos. Além disso, todos os algoritmos empregaram a técnica de realimentação. Os gráficos apresentados nas figuras 6.10 e 6.11 exibem os tempos de processamento de restaurações de imagens para um número de iterações igual a 100.

Pode-se observar claramente nos gráficos a considerável redução no tempo de processamento resultante da otimização do algoritmo serial proposta, sendo inclusive mais rápido do que o algoritmo paralelo anterior. Isso ocorre porque, na implementação paralela anterior, os dados utilizados se encontravam todos armazenados na memória principal do sistema, sendo transferidos para a memória de GPU a cada iteração, comprometendo o desempenho da aplicação, devido ao custo do constante tráfego CPU$\leftrightarrow$GPU. Por outro lado, no algoritmo serial apresentado neste trabalho, todos os cálculos que poderiam ser executados apenas uma vez o foram, e todas as operações que poderiam ser executadas em conjunto foram agrupadas nos mesmos loops, resultando nas reduções de tempos observadas. Além disso, o reaproveitamento de dados previamente calculados contribui significativamente para os resultados obtidos.

Para os algoritmos paralelos propostos, ainda que o uso de memória mapeada pela GPU tenha se mostrado mais eficiente, conforme os resultados obtidos com a avaliação da ferramenta de exibição em 6.1, para os algoritmos de restauração a metodologia empregada foi o uso de memória não paginável, uma vez que a técnica de *zero-copy* é dependente da plataforma onde o programa será executado. Ainda assim, os resultados

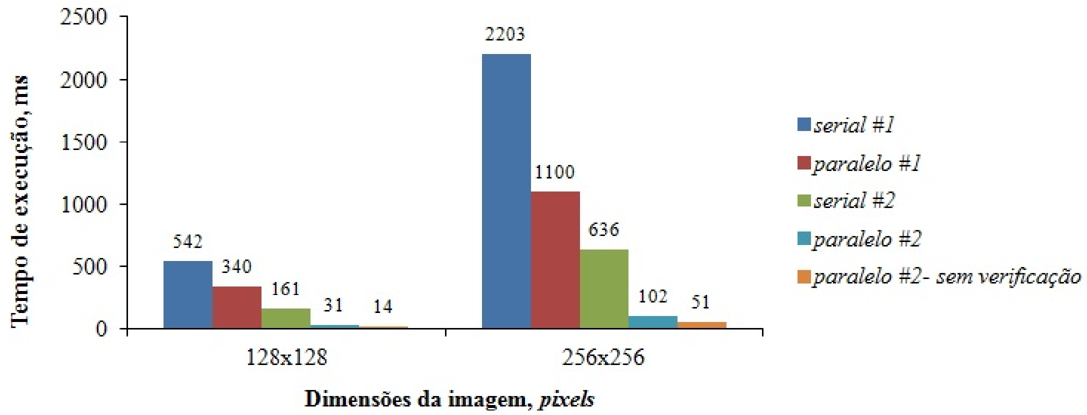

Figura 6.10: Tempos de execução da restauração de imagens de 128x128 e 256x256 *pixels*.

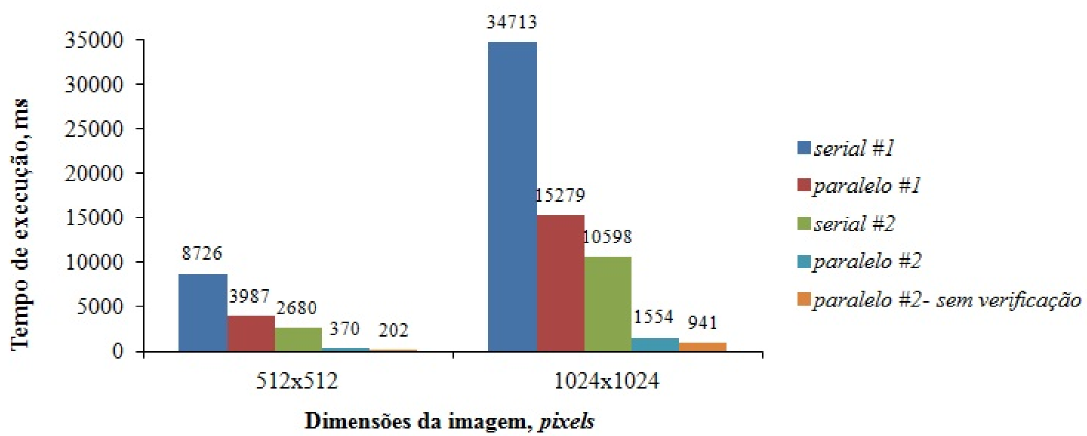

Figura 6.11: Tempos de execução da restauração de imagens de 512x512 e 1024x1024 *pixels*.

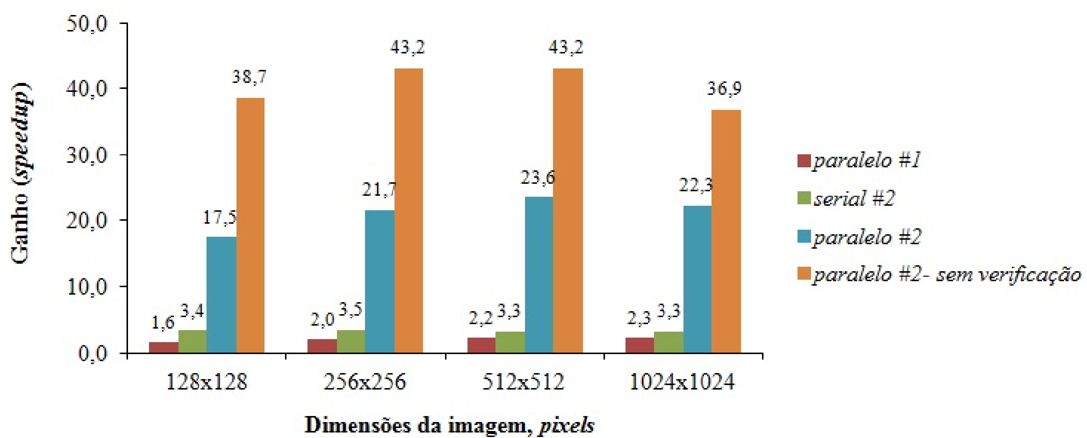

Figura 6.12: Ganhos de desempenho (*speedups*) decorrentes da paralelização da restauração de imagens.

mostram que os tempos de execução das abordagens paralelas são significativamente inferiores ao tempo de processamento das abordagens anteriores. Uma das razões, além do uso de memória não paginável (o que reduz o custo do tráfego de dados), é o fato de todos os dados empregados na restauração estarem armazenados na memória de GPU. Os únicos dados transferidos durante a execução do loop principal são aqueles utilizados para o cálculo de $Q(\hat{x})$, para fins de verificação de convergência. Uma vez removida esta verificação, os tempos de execução foram reduzidos consideravelmente, devido à economia do custo de tráfego de dados a cada iteração. O gráfico da figura 6.12 exibe os ganhos obtidos com a otimização do algoritmo serial, bem como com a paralelização do algoritmo, com e sem verificações de convergência.

6.2.2 Qualidade da restauração

Para a avaliação da qualidade das restaurações realizadas com os algoritmos propostos, as imagens degradadas apresentadas na figura 6.8, resultantes da aplicação de borramento e/ou ruído à imagem de teste (figura 6.7) foram restauradas utilizando-se os dois algoritmos anteriores, bem como com os dois algoritmos, serial e paralelo, propostos neste trabalho. O algoritmo paralelo sem verificações de convergência não foi objeto de avaliação de qualidade por tratar-se de um algoritmo idêntico ao algoritmo *paralelo #2* no que tange ao processo de restauração.

As métricas utilizadas na avaliação são o valor final do funcional de regularização $Q(\hat{x})$, o resíduo dado pelo somatório dos elementos da matriz YBX, que define as diferenças entre a imagem original e a restaurada convoluída por b, e o erro médio quadrático (*Mean Square Error* - MSE) absoluto e relativo. O MSE é dado pela expressão:

$$MSE = \frac{\sum_{i=0}^{M-1} \sum_{j=0}^{M-1} [y_{i,j} - \hat{x}_{i,j}]^2}{N_{elm}} , \qquad (6.2)$$

sendo o MSE relativo (%MSE) dado pela razão percentual entre o MSE da imagem restaurada pelo MSE da imagem degradada inicial. Para as restaurações com todos os algoritmos a técnica de realimentação foi empregada, as imagens possuíam 256x256 *pixels* de resolução, o número de iterações foi definido como 100, e os parâmetros empregados para cada imagem são apresentados na tabela 6.1. A figura 6.13 exibe os resultados das restaurações das imagens degradadas com o uso da aplicação paralela proposta.

A tabela 6.2 apresenta os resultados da avaliação da qualidade das restaurações realizadas. É possível observar que as diferenças entre os valores obtidos utilizando todos os algoritmos é muito pequena, o que evidencia que o uso dos algoritmos propostos produz resultados com qualidade similar aos encontrados em abordagens anteriores, de maneira mais eficiente.

Tabela 6.1: Parâmetros de restauração empregados

Imagem	b	σ^2	α	q	γ
D5v20	5x5	20	0,03	1	0,1
D5v40	5x5	20	0,03	1	0,1
D5v20s20	5x5	20	0,07749	1	0,1
D5v20s40	5x5	20	0,03	1	0,1

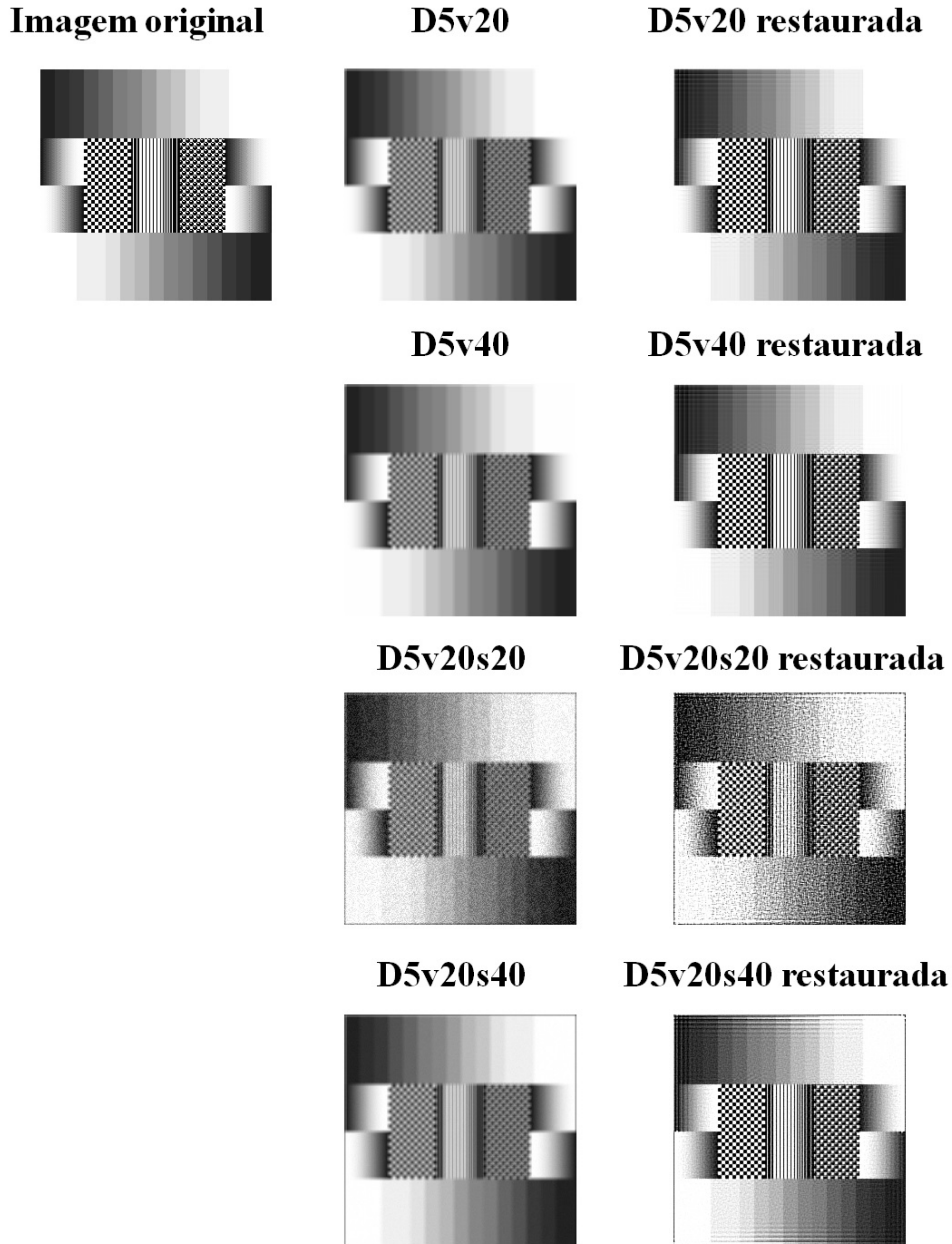

Figura 6.13: Imagens restauradas com o algoritmo paralelo implementado.

Tabela 6.2: Resultados da análise da qualidade das restaurações

D5v20

Métrica	Imagem borrada	*serial #1*	*paralelo #1*	*serial #2*	*paralelo #2*
$Q(\hat{x})$	97,5724	0,7520	0,7519	0,7485	0,7484
resíduo	7,7921	3,2211	3,2212	3,1582	3,1582
MSE	3193,4504	554,4484	554,4476	563,2589	563,2589
%MSE	-	17,36	17,36	17,64	17,64

D5v40

Métrica	Imagem borrada	*serial #1*	*paralelo #1*	*serial #2*	*paralelo #2*
$Q(\hat{x})$	100,4325	0,7322	0,7322	0,7319	0,7319
resíduo	7,9036	2,7358	2,7357	2,6954	2,6954
MSE	3228,2549	522,4752	522,4751	531,1597	531,1600
%MSE	-	16,18	16,18	16,45	16,45

D5v20s20

Métrica	Imagem borrada	*serial #1*	*paralelo #1*	*serial #2*	*paralelo #2*
$Q(\hat{x})$	359,1738	190,0514	190,0506	190,2767	190,2761
resíduo	-46,5214	27,2521	27,2520	27,5563	27,5564
MSE	3607,3142	2815,6194	2815,6206	2780,2302	2780,2312
%MSE	-	78,05	78,05	77,07	77,07

D5v20s40

Métrica	Imagem borrada	*serial #1*	*paralelo #1*	*serial #2*	*paralelo #2*
$Q(\hat{x})$	124,8848	17,8055	17,8052	17,8443	17,8440
resíduo	-48,4056	5,4664	5,4663	5,3255	5,3256
MSE	3345,5012	1048,9967	1048,9967	1044,2080	1044,2080
%MSE	-	31,36	31,36	31,21	31,21

Uma segunda etapa da avaliação da qualidade das restaurações consistiu em aplicar os algoritmos de restauração sobre a imagem de um texto previamente borrada com um operador de borramento b com dimensões 5x5 e variância 20, o que tornou a mesma ilegível. A figura 6.14 exibe a imagem original, a mesma imagem degradada, e o resultado da restauração. Os resultados da tabela 6.3 descrevem a análise da qualidade das restaurações com os quatro algoritmos. Em todos os casos, os parâmetros de restauração foram b(5x5), com $\sigma^2 = 20$, 100 iterações, e α, q e γ iguais a 1. Mais uma vez é possível observar que todos os algoritmos apresentam resultados semelhantes, o que valida o uso das abordagens propostas neste trabalho.

6.2.3 Efeitos do operador de borramento nas bordas de uma imagem

Conforme havia sido descrito em 5.4, o algoritmo proposto emprega matrizes para armazenar os valores de normalização das convoluções utilizando o operador de borra-

Original Borrada Restaurada

Figura 6.14: Resultado da restauração de um texto ilegível.

mento b, bem como os valores dos resultados da convolução $b * b$, que se reduziu a um somatório dos quadrados dos termos envolvidos na convolução, enquanto que em ambas as abordagens anteriores foram utilizados um vetor e um valor fixo, respectivamente.

O uso de matrizes foi adotado para que os resultados das convoluções em regiões próximas às bordas das imagens fossem mais exatos, sem prejuízo ao desempenho da aplicação, conforme apontaram os tempos de execução apresentados. Isso se torna crítico à medida que as dimensões da matriz de borramento aumentam e se tornam comparáveis às dimensões da imagem. Um exemplo de caso onde estas condições seriam satisfeitas é a varredura de uma região muito pequena da superfície amostrada, de forma que as dimensões da ponteira do AFM seriam comparáveis ao relevo da amostra. Nestes casos, a área de atuação da matriz de borramento que representa a ponteira seria comparável à área da imagem.

A figura 6.15 apresenta os resultados de restaurações de uma imagem de teste de 128x128 *pixels* degradadas por um operador de borramento de dimensões 13x13, com uma variância igual a 20, nomeada D13v20, através do uso dos algoritmos *serial #1* e *paralelo #2*. As dimensões do operador representam, portanto, cerca de 10% da dimensão lateral da imagem. A imagem obtida com o algoritmo *serial #1* apresenta artefatos nas regiões próximas às bordas, o que não ocorre com a imagem restaurada com o algoritmo proposto. Isso se deve ao uso das matrizes com os valores exatos para cada elemento da imagem, ainda que o custo desta abordagem seja um uso maior do espaço de memória da GPU.

A tabela 6.4 apresenta os resultados das restaurações da imagem D13v20. Em ambas

Tabela 6.3: Resultados da análise da qualidade das restaurações de um texto ilegível

Métrica	Imagem borrada	*serial #1*	*paralelo #1*	*serial #2*	*paralelo #2*
$Q(\hat{x})$	69,4673	0,5387	0,5387	0,5172	0,5172
resíduo	0,0905	16,5453	16,5451	16,3795	16,3797
MSE	2025,2158	322,8092	322,8101	325,2868	325,2868
%MSE	-	15,94	15,94	16,06	16,06

Tabela 6.4: Resultados da análise da qualidade das restaurações da imagem D13v20

Métrica	Imagem borrada	*serial #1*	*paralelo #2*
$Q(\hat{x})$	16,1563	0,5179	0,0173
resíduo	3,2744	1,3230	0,0847
MSE	3630,7639	2421,2844	2390,0632
%MSE	-	66,69	65,83

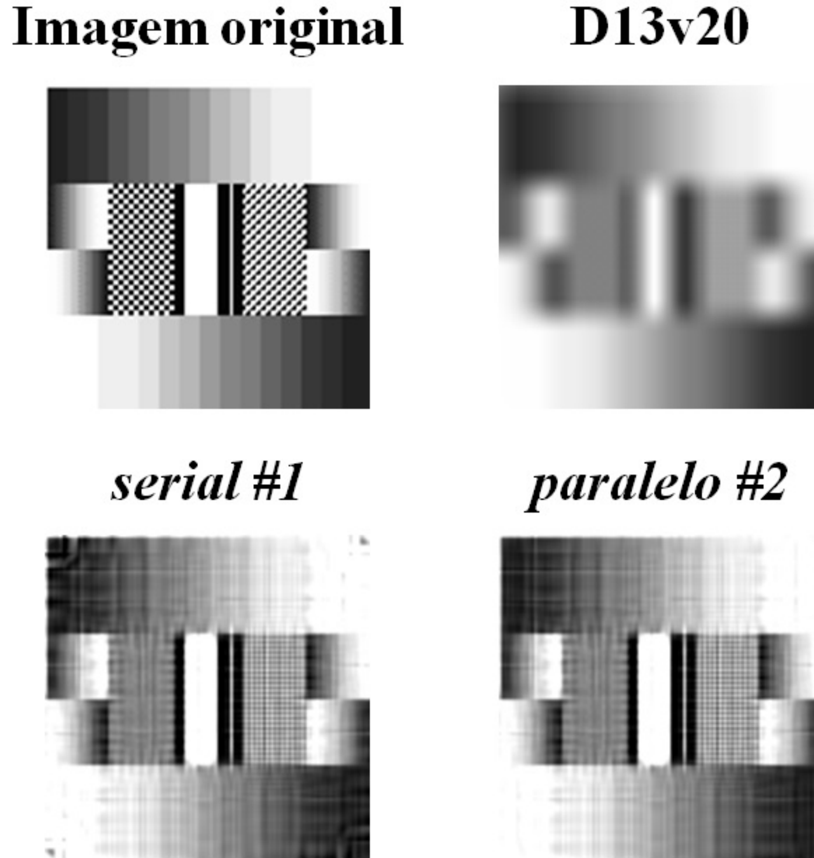

Figura 6.15: Efeitos de diferentes estratégias de normalização para o uso do operador de borramento.

as execuções, os parâmetros empregados foram b(13x13), com $\sigma^2 = 20$, 1000 iterações, e α, q e γ iguais a 1. Apesar das pequenas diferenças entre os valores encontrados com o uso do MSE, os valores de $Q(\hat{x})$ e resíduo obtidos com a abordagem proposta são consideravelmente menores que os obtidos com a abordagem anterior, o que reflete numericamente as diferenças observadas na figura 6.15, bem como o que havia sido previsto durante o desenvolvimento dos algoritmos. É importante observar que este efeito se intensifica à medida que o número de iterações aumenta, conforme descrito em [9].

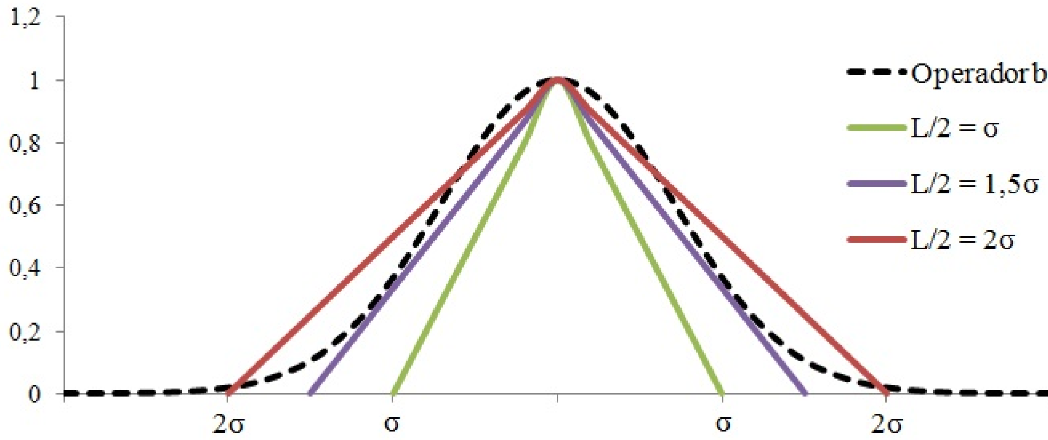

Figura 6.16: Curva do tipo Gaussiana, com $\sigma = 5$, e representações triangulares de ponteiras com diferentes valores de largura L. Valores adimensionais.

6.2.4 Efeitos do dimensionamento do operador de borramento

Um fator crítico na restauração de imagens de microscopia de força atômica é o correto dimensionamento do operador de borramento, que representa o campo de atuação da ponteira de medição, usualmente piramidal, sobre a superfície da amostra. O dimensionamento consiste na determinação das dimensões da matriz de borramento, bem como a variância da mesma, de maneira a representar adequadamente a geometria da ponteira em dimensões de imagem (*pixels*).

O gráfico da figura 6.16 exibe uma curva do tipo Gaussiana, conforme descrito na equação 5.2 com desvio-padrão (σ), bem como três representações triangulares de ponteiras, com larguras L cujos valores de $L/2$ são iguais a σ, 1,5σ e 2σ, de maneira a determinar a relação existente entre a largura da ponteira e a variância do operador de borramento. Claramente, a largura que melhor se adapta ao formato da curva é aquela na qual $L/2$ é igual a 1,5σ.

As dimensões da matriz de borramento, por outro lado, dependem das dimensões físicas da ponteira (altura e largura), e da porção da ponteira que de fato interage com a superfície da amostra, conforme discutido em 5.2.1, o que requer certo conhecimento prévio do relevo a ser estudado, bem como o conhecimento das dimensões da imagem. A figura 6.17 descreve o formato de uma ponteira comercialmente disponível, onde h é a altura da ponteira, TSB é a largura da sua porção frontal, e FA, BA e SA são, respectivamente os ângulos frontal, traseiro e lateral da mesma [34]. Estes ângulos não necessariamente são idênticos, porém para efeitos práticos podem ser considerados como tal, de forma que $TSB = L/2$, e

$$TSB = \frac{L}{2} = h \cdot tg(FA) \rightarrow L = 2 \cdot h \cdot tg(FA) \tag{6.3}$$

Uma vez determinada a largura da ponteira, as dimensões da matriz de borramento

devem ser calculadas. Para tal, é necessário conhecer a sensibilidade da imagem em relação às dimensões físicas da área de amostragem, de maneira idêntica à apresentada em 4.3.1. Considerando imagens quadradas, assim como a área de varredura, a sensibilidade da imagem é expressa em $\mu m/pixel$, e dada por:

$$sens_img = \frac{L_a}{L_img} \, , \tag{6.4}$$

onde L_a é a largura da área de varredura e L_img é a largura da imagem, de forma que a dimensão lateral da matriz de borramento é dada por:

$$dim_b = \frac{L}{sens_img} \tag{6.5}$$

O valor obtido deve ser arredondado para o valor ímpar mais próximo, de forma que a matriz possua um elemento central. Finalmente, considerando a análise da figura 6.16, o desvio-padrão σ e por consequência a variância σ^2 são dados por:

$$1,5 \cdot \sigma = \frac{dim_b - 1}{2} \rightarrow \sigma = \frac{dim_b - 1}{3} \tag{6.6}$$

$$\sigma^2 = \left(\frac{dim_b - 1}{3} \right)^2 \tag{6.7}$$

Uma vez definidas expressões para o dimensionamento da matriz de borramento, elas foram verificadas através da restauração de uma imagem obtida, com o uso de uma ponteira de dimensões conhecidas, de uma superfície de uma grade de teste, já apresentada na

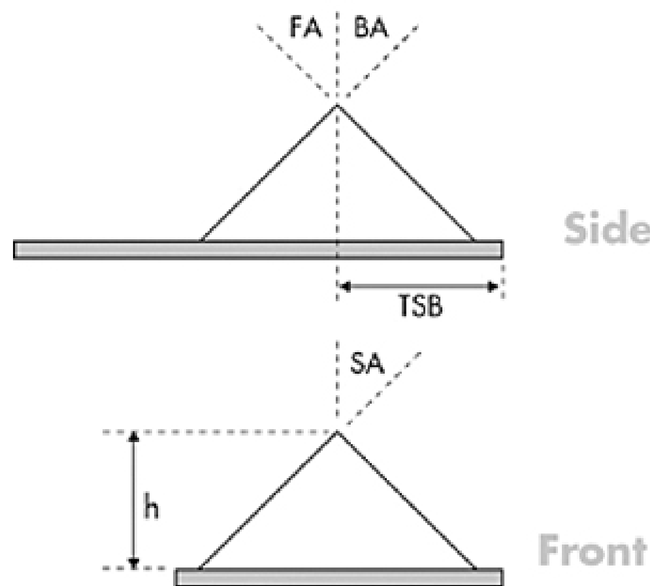

Figura 6.17: Diagrama descritivo de uma ponteira de AFM comercial. BRUKER AFM Probes [34]

figura 2.7, cujas dimensões são conhecidas. O ângulo FA da ponteira, de acordo com as especificações do fabricante, é de 35°, e a altura da mesma é de 3 μm. Entretanto, a altura máxima do relevo da grade de teste é especificada como sendo cerca de 1 μm, de forma que este será o valor atribuído a h para o cálculo de L, de forma a obter a área da ponteira que efetivamente interage com a amostra. As dimensões da imagem são 128x128 *pixels* e as dimensões da área de varredura são 9x9 μm.

Desta forma, foram encontrados os valores de L e $sens_img$, iguais a 1,4 μm e 0,07 $\mu m/pixel$, respectivamente, o que levou a um valor de dim_b igual a 20, sendo arredondado para 21, valor ímpar imediatamente superior. Uma vez definida a dimensão lateral da matriz, a variância foi calculada como sendo igual a 44.

Para verificar a exatidão dos cálculos, a imagem amostrada da grade de teste foi restaurada utilizando-se diferentes valores de variância. Como na imagem obtida a grade aparecia inclinada, as imagens resultantes, assim como a imagem original foram rotacionadas, de forma que fosse possível analisar os perfis de intensidade de linhas horizontais que seguem os padrões da grade, conforme apresentado na figura 6.18. Todas

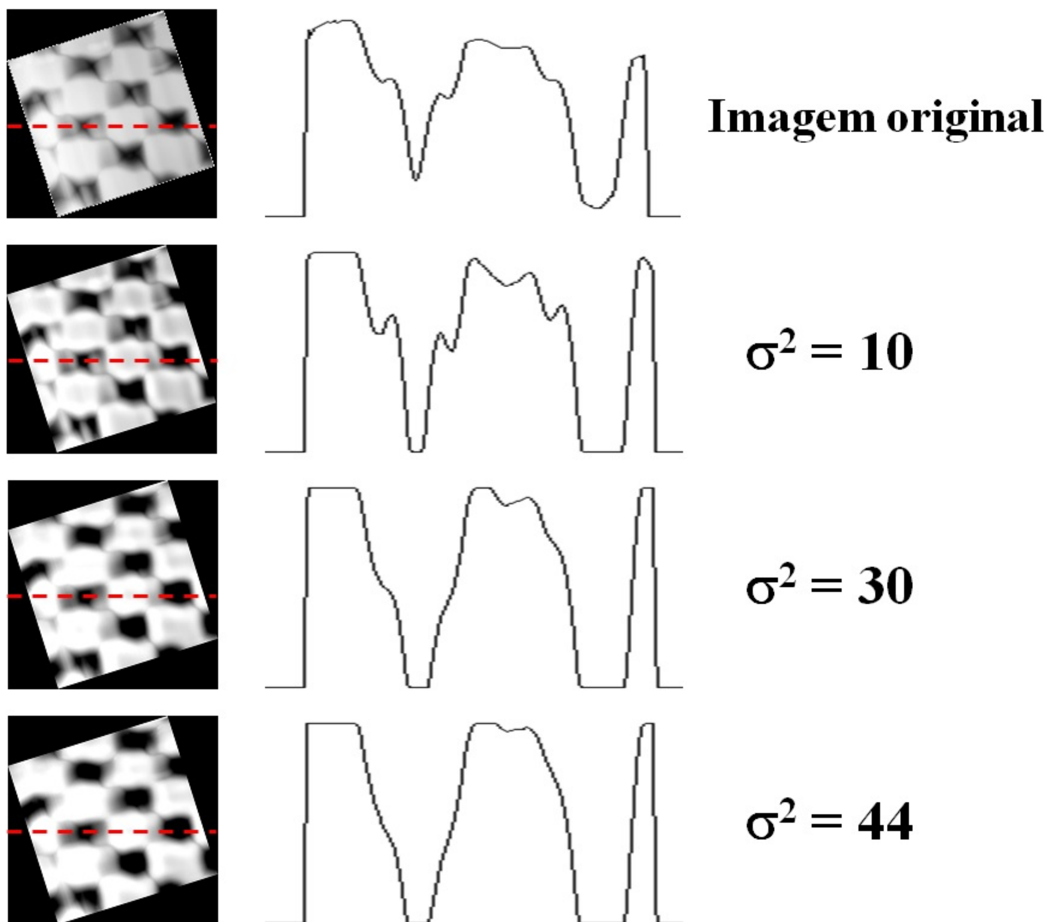

Figura 6.18: Perfis de intensidade na direção horizontal (linha tracejada) de imagens restauradas com um matriz de borramento b com dimensões 21x21, para diferentes valores de variância.

as restaurações foram executadas com uma matriz de borramento de dimensões 21x21, empregando como parâmetros $\alpha = 0,03$, $q = 1$, $\gamma = 0,1$ e realimentação.

É possível observar tanto nas imagens restauradas como nos perfis de cada uma delas que, à medida que a variância aumenta, o perfil quadriculado da grade gradativamente é recuperado, evidenciando assim a qualidade da restauração e validando a metodologia proposta nesta seção. Um aspecto muito interessante deste experimento foi o fato de ter sido necessária somente uma iteração para que a solução convergisse, o que pode, a princípio, ser atribuído às elevadas dimensões da matriz de borramento, visto que a influência desta na degradação da imagem visivelmente é superior à influência do ruído aditivo. As causas deste efeito, contudo, devem ser melhor investigadas.

É perfeitamente viável a adoção das dimensões da ponteira, bem como uma estimativa da altura do relevo da amostra, como parâmetros para o algoritmo de restauração. Além disso, uma vez que a ferramenta de exibição de imagens já determina as sensibilidades das imagens para exibição destas em escala real, a adoção destes parâmetros em uma eventual integração do algoritmo paralelo de restauração à ferramenta proporcionará a obtenção de imagens tridimensionais de microscopia de força atômica restauradas com qualidade aprimorada e em tempo real.

Finalmente, é de extrema importância observar que a adoção da metodologia proposta permite o emprego de ponteiras de medição mais largas e robustas, cujo custo de aquisição é consideravelmente inferior ao custo de ponteiras mais afiadas e frágeis, que proporcionam imagens mais nítidas que as primeiras. A possibilidade de remoção do borramento causado pelas ponteiras mais largas, aliada ao custo reduzido e a maior durabilidade destas, pode torna a técnica de microscopia de força atômica mais acessível a institutos acadêmicos e de pesquisa.

Capítulo 7

Conclusões

Este trabalho apresentou como proposta principal um novo algoritmo paralelo de restauração de imagens de microscopia de força atômica, empregando o funcional de regularização de Tikhonov, através do uso da arquitetura CUDA. Para tal, um novo algoritmo serial otimizado foi elaborado, através de uma análise aprofundada das operações envolvidas no processo de restauração, que por si só já acrescentou ganhos de desempenho quando comparado com abordagens anteriores.

O resultado desta análise foi a implementação de uma aplicação paralela capaz de restaurar imagens de 128x128 *pixels* em até 14 ms e imagens de 256x256 *pixels* em até 51 ms, o que representa velocidades de processamento até 43 vezes mais rápidas que as registradas com a abordagem serial anterior, usada como referência, apresentando ganhos consideráveis mesmo com imagens de dimensões reduzidas. Esta aceleração é devida a adoção de uma estratégia na qual o tráfego de dados entre CPU e GPU foi reduzido ao mínimo necessário, sendo inclusive totalmente suprimido no caso da eliminação das verificações de convergência. Isso foi alcançado através do armazenamento de todos os dados na memória da GPU. Além disso, o adequado agrupamento das operações que não possuíam dependências de dados entre si no mesmo *kernel* também contribuiu significativamente para que os objetivos deste trabalho fossem alcançados.

Cabe salientar que os resultados apresentados são relativos a uma GPU mediana (GeForce GT555M), em comparação com uma CPU de primeira linha (Intel i7), o que sugere ganhos ainda maiores através do uso de GPUs mais modernas. Como comparação, em sua tese de doutorado, Cidade [6] relatou tempos de processamento de aproximadamente 15 minutos para um total de 50 iterações, durante restaurações de imagens de 256x256 *pixels*, o que equivale a cerca de 30 minutos para um total de 100 iterações. Isso representa um tempo de processamento aproximadamente 35000 vezes maior que o apresentado neste trabalho, para imagens de mesmas dimensões. Guardadas as significativas diferenças entre os recursos computacionais empregados nos dois trabalhos (à época, o algoritmo foi avaliado com um computador dotado de um processador Pentium III, com 550 MHz de *clock*), ainda assim é possível evidenciar os ganhos de desempenho decorrentes da es-

tratégia adotada, bem como do uso de tecnologias mais recentes.

Além dos ganhos de desempenho observados, a qualidades das imagens restauradas foi aprimorada através do uso das matrizes de normalização dos cálculos envolvendo o operador de borramento b, o que resultou em restaurações satisfatórias mesmo em regiões próximas às bordas das imagens.

Como objetivo secundário, uma biblioteca dinâmica (dll) de exibição de imagens de AFM em 3D foi desenvolvida para o microscópio em uso no IBCCF, baseado nas características de operação e aquisição de dados do mesmo, permitindo a visualização em tempo real de imagens tridimensionais em escala real das superfícies amostradas, através do uso da biblioteca OpenGL. Uma implementação paralela desta ferramenta utilizando CUDA foi apresentada, com diferentes metodologias de alocação de acesso à memória, resultando em ganhos de desempenho com imagens de dimensões a partir de 256x256 *pixels*. Ainda que os ganhos sejam discretos, os tempos de execução apresentados correspondem a 500 conjuntos de dados, o que significa dizer, por exemplo, que o tempo de composição de uma única imagem de 1024x1024 *pixels* e sua representação tridimensional é de menos de 38 ms, enquanto que para uma imagem de 128x128 *pixels* este tempo se reduz a 0,6 ms, sendo estes tempos desprezíveis quando comparados com os tempos de restauração. É possível compor, restaurar e exibir uma imagem de 128x128 *pixels* em menos de 15 ms, e em menos de 1 segundo para imagens de 1024x1024 *pixels*. A principal vantagem desta abordagem foi desonerar a CPU do custo de executar tais operações, permitindo que a mesma se concentre no controle e aquisição de dados do AFM, realizado por um programa específico, que se comunica dinamicamente com a ferramenta, de forma totalmente transparente. Além disso, a paralelização da ferramenta assegura que a imagem resultante da varredura já esteja armazenada na memória da GPU para uma futura restauração, o que será possível a partir da integração das duas soluções, a ser implementada em um futuro próximo, que irá permitir a visualização tridimensional de imagens restauradas em tempo real.

Um fruto adicional deste trabalho foi a elaboração da metodologia de dimensionamento do operador de borramento b a partir de informações prévias da ponteira de medição do AFM, bem como da superfície analisada, de forma a se determinar a porção da ponteira que de fato interage com a amostra. Com base nestas informações foi possível determinar a dimensão da matriz de borramento e sua variância, resultando em melhores restaurações. A avaliação de imagens restauradas com diferentes valores de variância ratificou o que havia sido previsto durante a elaboração da metodologia, que também será integrada às duas outras soluções, resultando em uma ferramenta mais completa para restaurações de imagens de microscopia de força atômica. A adoção desta metodologia também possibilita a adoção de ponteiras mais robustas, sem prejuízo das imagens restauradas, o que pode representar uma considerável redução nos custos de operação do AFM, tornando esta técnica de microscopia mais acessível.

7.1 Trabalhos futuros

Ainda que os resultados apresentados apontem que os objetivos deste trabalho foram plenamente alcançados, a própria pesquisa para o desenvolvimento deste abriu portas para implementações futuras, onde o desempenho das aplicações seja aprimorado ainda mais, a qualidade dos resultados apresentados seja ainda melhor, e onde novas funcionalidades possam ser adicionadas às soluções apresentadas.

A primeira, dentre todas, certamente é a integração do algoritmo de restauração à ferramenta de exibição de imagens, o que irá permitir a visualização de imagens restauradas em tempos real. Através de um controle dinâmico dos parâmetros de restauração (α, q e γ), o usuário será capaz de ajustar a imagem restaurada à sua necessidade, dado que cada pessoa possui uma percepção visual diferenciada, de forma que os parâmetros ótimos para uma pessoa não necessariamente serão adequado para outra. Outros filtros, inclusive, podem ser adicionados ao sistema de maneira a obter efeitos extras (detecção de bordas, reconhecimento de padrões, dentre outros), de maneira a permitir a implementação de mecanismos de medição das dimensões das superfícies estudadas.

Além disso, a adoção da metodologia de dimensionamento da matriz de borramento a este conjunto irá simplificar consideravelmente a operação do sistema. De fato, as próprias imagens obtidas pela ferramenta de exibição podem fornecer informações do relevo úteis para o dimensionamento da matriz a ser empregada nas próprias restaurações das mesmas. A adoção desta metologia será objeto de estudos mais aprofundados para prever, inclusive, matrizes de borramento retangulares, para os casos nos quais as resoluções de amostragem horizontal e vertical são diferentes, conforme apresentado em 2.6.

Dado que a ponteira pode interagir de maneira diferenciada em diferentes regiões da superfície, uma abordagem alternativa seria mapear seções da imagem onde diferentes operadores poderiam ser empregados. Isso, contudo, pode acarretar em aumentos consideráveis no custo computacional do processo de restauração, e deve ser avaliado com cautela.

Com relação ao desempenho do processo de restauração, ainda que os resultados apresentados sejam expressivos, algumas medidas podem (ou não) resultar em ganhos ainda maiores. Um exemplo seria a adoção da técnica de *zero-copy* para arquiteturas que favoreçam tal medida. Um outro exemplo seria o uso das memórias compartilhada e/ou de constante, ainda que o uso da memória global tenha sido otimizado através do acesso *coalescente*. Ainda mais, novas estratégias de blocos e *threads* podem ser investigadas, inclusive com o uso de diferentes níveis de memória da GPU, pouco explorados neste trabalho.

Uma outra alternativa de otimização do desempenho da aplicação, porém mais voltada à estrutura do algoritmo, seria a paralelização dos somatórios das matrizes empre-

gadas para o cálculo de $Q(\hat{x})$, através do trabalho cooperativo de *threads*, o que evitaria os envios de matrizes que comprometem o desempenho do programa. Além disso, uma alternativa ao uso das matrizes para normalização das convoluções com o operador de borramento está sendo elaborada, porém não pôde ser implementada neste trabalho. Considerando que, na grande maioria dos casos, para a maior parte dos elementos das imagens os valores de $norma_b$ e $S2BB$ são idênticos, o uso das matrizes, apesar de correto, pode ser evitado através do uso de matrizes menores, com dimensões iguais às da matriz de borramento, o que reduziria consideravelmente o espaço de memória necessário para armazená-las, permitindo inclusive que estas sejam armazenadas na memória compartilhada da GPU.

Apesar de não terem sido exploradas neste trabalho, existem maneiras de se integrar as bibliotecas CUDA e OpenGL, de maneira a evitar o tráfego de dados relativos aos *arrays* de vértices e cores, o que resultaria em ganhos no desempenho da aplicação paralela. Esta abordagem, entretanto, pode requerer mudanças estruturais profundas no algoritmo implementado, de maneira que deve ser estudada com atenção.

Finalmente, a técnica de α ótimo apresentada em [10] poderia ser integrada ao processo de restauração, simplificando o controle dos demais parâmetros e reduzindo o número de iterações necessário para a convergência da solução. Uma versão paralela, usando a arquitetura CUDA, pode ser apresentada como forma de se obter ganhos ainda maiores no desempenho da restauração. Neste mesmo trabalho, outras métricas de avaliação de imagens devem são apresentadas, que podem ser aplicadas às imagens restauradas com os algoritmos propostos, obtendo assim mais parâmetros para comparação da qualidade das imagens resultantes.

Referências Bibliográficas

[1] BINNIG, G., QUATE, C. F., GERBER, C. H. "Atomic Force Microscope", *Phys. Rev. Lett.*, v. 56, n. 9, pp. 930–933, 1986.

[2] KANG, M. G., KATSAGGELLOS, A. K. "General Choice of the Regularization Functional in Regularized Image Restoration", *IEEE Transactions on Image Processing*, v. 4, n. 5, pp. 594–602, 1995.

[3] TIKHONOV, A. N., ARSENIN, V. Y. "Solutions of Ill-Posed Problems", *Mathematics of Computation*, v. 32, n. 144, pp. 1320–1322, 1977.

[4] FANTNER, G. E., HEGARTY, P., KINDT, J. H., et al. "Data acquisition system for high speed atomic force microscopy", *Review of Scientific Instruments*, v. 76, n. 2, pp. 026118, 2005. doi: 10.1063/1.1850651.

[5] CIDADE, G. A. G., WEISSMÜLLER, G., BISCH, P. M. "A microcontroller-based system for peizoscanner nonlinearity correction: Atomic force microscope", *Rev. Sci. Instrum.*, v. 69, n. 10, pp. 136–140, 1998.

[6] CIDADE, G. A. G. *Desenvolvimento de metodologias de aquisição e processamento de imagens biológicas em microscopia de força atômica (AFM).* Tese de D.Sc., Instituto de Biofísica Carlos Chagas Filho, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil, 2000.

[7] CIDADE, G. A. G., SILVA NETO, A. J., ROBERTY, N. C. "Restauração Imagens com Aplicações em Biologia e Engenharia - Problemas Inversos em Nanociência e Nanotecnologia", *Notes in Applied Mathematics*, v. 1, 2003.

[8] FURTADO, V. *Avaliação do uso do funcional de Tikhonov com uma família de funções de regularização a um parâmetro para a restauração de imagens biológicas*. Dissertação de M.Sc., Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brasil, 2002.

[9] STUTZ, D. *Restauração de imagens em escala nanométrica com funcional de regularização de Tikhonov e computação paralela*. Dissertação de M.Sc., Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brasil, 2004.

[10] STUTZ, D. *Estratégias de Computação Paralela para a Restauração de Imagens com o Funcional de Regularização de Tikhonov*. Tese de D.Sc., Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brasil, 2009.

[11] ALMEIDA, A. G., STUTZ, D., CIDADE, G. A. G., et al. "Uso de graphics processing unit (gpu) na restauração de imagens de microscopia de força atômica com regularização de tikhonov". In: *Congresso Nacional de Matemática Aplicada e Computacional*, p. 5, Brasil, 2009. Sociedade Brasileira de Matemática Aplicada e Computacional.

[12] ALMEIDA, A. G. *Restauração de imagens de microscopia de força atômica com uso da regularização de tikhonov via processamennto em GPU*. Tese de D.Sc., Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brasil, 2013.

[13] QUELHAS, K. N., CIDADE, G. A. G., FARIAS, R. "High-Speed Restoration of Atomic Force Microscopy Images Using Tikhonov Regularization in GPGPU". In: *IEEE 26th International Symposium on Computer Architecture and High Performance Computing Workshops - SBAC-PADW 2014*, pp. 7–11, 2014. doi: 10.1109/SBAC-PADW.2014.22.

[14] BINNIG, G., ROHRER, H., GERBER, C. H., et al. "Tunneling through a controllable vacuum gap", *Appl. Phys. Lett.*, v. 40, n. 2, pp. 178–180, 1982. doi: 10.1063/1.92999.

[15] BINNIG, G., ROHRER, H., GERBER, C. H., et al. "Surface Studies by Scanning Tunneling Microscope", *Appl. Phys. Lett.*, v. 49, n. 1, pp. 57–60, 1982.

[16] MORITA, S. "Atom world based on nano-forces: 25 years of atomic force microscopy", *Journal of Electron Microscopy*, , n. 60(Supplement 1), pp. S199–S211, 2011. doi: 10.1093/jmicro/dfr047.

[17] HARTMANN, U. "van der Waals interactions between sharp probes and flat sample surfaces", *Physical Rev. B*, v. 3, n. 43, pp. 2404–2407, 1991.

[18] WEISSMÜLLER, G., CIDADE, G. A. G., BISCH, P. M. "Microscopia de Força Atômica – Nanoscopia, Nanomanipulação e nanocaracterização de Biomateriais." In: *Terapias Avançadas – Células-tronco, Livro: Terapia Gênica e Nanotecnologia Aplicada à Saúde*, pp. 307–321. Editora Atheneu, 2007. ISBN: 978-85-7379-929-3.

[19] HOWLAND, R., BENATAR, L. *A pratical guide to scanning probe microscopy*. ThermoMicroscopes, 2000.

[20] ANDO, T., UCHIHASHI, T., FUKUMA, T. "High-speed atomic force microscopy for nano-visualization of dynamic biomolecular processes", *Progress in Surface Science 83 (2008) 337–437*, , n. 83, pp. 337–437, 2008.

[21] SANDERS, J., KANDROT, E. *CUDA by example : an introduction to general-purpose GPU programming*. USA, Addison-Wesley, 2011. ISBN: 978-0-13-138768-3.

[22] KIRK, D. B., MEI HWU, W. *Programming Massively Parallel Processors: A Hands-on Approach*. USA, Elsevier, 2010. ISBN: 978-0-12-381472-2.

[23] NVIDIA. *CUDA C PROGRAMMING GUIDE v5.0*. www.nvidia.com, NVIDIA, 2012. ISBN: 978-0-12-381472-2.

[24] KERNIGHAN, B. W., RITCHIE, D. M. *The C Programming Language*. Prentice Hall Software Series. 2nd ed. USA, Prentice Hall, 1989. ISBN: 0-13-110370-9.

[25] DEITEL, H. M. *C++ How to program*. 5th ed. USA, Prentice Hall, 2005. ISBN: 0-13-185757-6.

[26] SHREINER, D. *OpenGL Programming Guide*. 7th ed. USA, Addison-Wesley, 2009. ISBN: 978-0-321-55262-4.

[27] FREEGLUT PROGRAMMING CONSORTIUM, T. "The Open-Source OpenGL Utility Toolkit (freeglut 3.0.0) Application Programming Interface Version 4.0". Disponível em: <http://freeglut.sourceforge.net/docs/api.php>. Acesso em: 26 de Outubro de 2013.

[28] KILGARD, M. J. "The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3". Disponível em: <https://www.opengl.org/resources/libraries/glut/spec3/spec3.html>. Acesso em: 26 de Outubro de 2013.

[29] TSCHUMPERLÉ, D. "The CImg Library". Disponível em: <http://cimg.sourceforge.net/reference/index.html>. Acesso em: 10 de Junho de 2013.

[30] KATSAGGELLOS, A. K. "Iterative Image Restoration Algorithm", *Optical Engineering Special Issue on Visual Comunications and Image Processing*, v. 28, n. 7, pp. 735–748, 1989. ISSN: 0091-3286.

[31] CIDADE, G. A. G., ROBERTY, N. C., SILVA NETO, A. J. "Reconstrução de Imagens com Regularização de Tikhonov e Determinação do Parâmetro

de Regularização Ótimo". In: *II Simpósio Interdisciplinar em Tecnologia e Saúde*, Rio de Janeiro, Brasil, 1998. Coppe/UFRJ.

[32] CARITA MONTERO, R., ROBERTY, N., SILVA NETO, A. "Natural Base Construction for Absorption Coefficient Estimation in Heterogeneous Participating Media with Divergent Beams". In: *International Conference on Inverse Problems in Engineering: Theory and pratice*, Port Ludlow, Washington, USA, 1999.

[33] BREGMAN, L. "The Relaxation Method of Finding the Common Point of Convex Sets and its Application to the Solution of Problems in Convex Programming", *Zh. vy̆chisl. Mat. mat. Fiz.*, v. 7, n. 3, pp. 620–631, 1967.

[34] BRUKER. "DNP-10 Tip Schematic". Disponível em: <http://www.brukerafmprobes.com/showTipDetails.aspx?imagename=/images/Product/tip/fullsize/DNP.jpg>. Acesso em: 05 de Janeiro de 2015.

Apêndice A

Publicações

Neste apêndice encontra-se o artigo submetido ao *5th Workshop on Applications for Multi-Core Architectures* (WAMCA 2014), realizado em conjunto com o *26th International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD 2014), publicado pelo *Institute of Electrical and Electronics Engineers* - IEEE, em outubro de 2014.

High-Speed Restoration of Atomic Force Microscopy Images using Tikhonov Regularization in GPGPU

Klaus N. Quelhas
COPPE/System Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, RJ, Brazil
klaus@cos.ufrj.br

Geraldo A. G. Cidade
Biophysics Institute Carlos Chagas Filho
Federal University of Rio de Janeiro
Rio de Janeiro, RJ, Brazil
gcidade@biof.ufrj.br

Ricardo Farias
COPPE/System Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, RJ, Brazil
rfarias@cos.ufrj.br

***Abstract*—The Atomic Force Microscopy (AFM) is a scanning probe technique widely used to produce nanometric scaled images of virtually any kind of non-conductive or biological surface. Depending on the scanning dimensions an expected AFM image structure is subjected to be modified by great amounts of external noise (low Signal/Noise ratios) - electrical or mechanical - and/or blurring due to the geometry of the measuring probe. In order to minimize such effects, image restoration techniques can be employed. The one based on the minimization of the Tikhonov's regularization functional is described, taking into account the characteristics of the measuring probe and the S/N ratio. This work proposes optimizations on both serial and parallel restoration algorithms, using CUDA library on a General Purpose Graphics Processing Unit, GPGPU, in terms of time performance and quality of restoration in regime of high speed imaging, one frame/sec or more. The results obtained are so far very promising, reaching speedups up to 43x over previous implementations.**

Keywords*—Atomic Force Microscopy, Image restoration, Tikhonovs Regularization Method, Parallel computing, GPGPU and CUDA.

I. Introduction

The Atomic Force Microscopy (AFM) technique [1] consists on the topographical imaging of non-conductive surfaces in nanoscale. The AFM makes use of a laser beam focused on the extremity of a cantilever device (with a specific spring constant), that interacts with the sample by means of a small tip sensor. The beam is then reflected over a photodetector surface producing a voltage signal proportional to the topography of the sample (Z direction); when conjugated with the scanning movement of the sample (X and Y directions) the AFM is capable to produce tridimensional images. The high speed AFM version [2] refers to the fast scanning movement of the sample, allowing the time to be the 4th imaging dimension, which is very useful for the acquisition of biological events at rates of images/second (conventional AFMs produce one image along several minutes). During the acquisition process the image signal can be affected by different sources: i) aspect ratio (height/width) of the tip, ii) mechanical vibration and electrical noise, leading, respectively, to blurring distortions and low signal/noise ratios. In nanometric scale, the final effect of the mentioned sources can hide valuable information, such as edges and structural details. Fig. 1 exhibits the effects of different tip geometries on images obtained by AFM. As can be seen, blurring depends strongly on the shape and the dimensions of the tip, when comparable to the dimensions of the scanned topography.

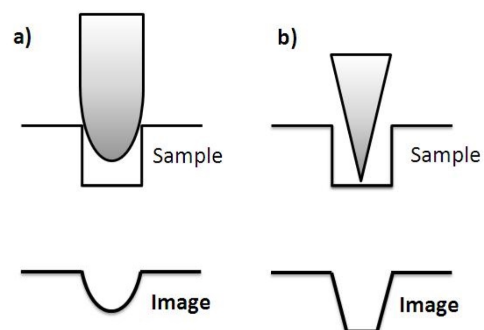

Fig. 1. Effects of tip geometry on resulting images obtained by a) a rounded tip, and b) a sharp tip, scanning the same sample area

One way to recover the original aspect of the image representation of the sample is the use of restoration techniques. The restoration proposed in this work is based on the Tikhonov's regularization method [3], where a 2nd order functional is minimized. This work proposes an optimization of a serial algorithm for image restoration, focused on both quality and time performance, also proposing a CUDA-based parallel implementation on a General Purpose Graphics Processing Unit, GPGPU. The restoration principles, the proposed serial and parallel algorithms, as well as the improvements obtained are described in the next sections.

II. Image restoration and Tikhonov's method

The image restoration proposed by [3] employs a Gauss-Seidel iterative method to solve a system with MxM linear equations and, therefore, obtain the best image representation when the Tikhonov's functional (1) reaches its minimum. Despite of its effectiveness, the Gauss-Seidel method demands high computational effort to run serially. Thus, a simplified Jacobi iterative approach was employed by [4], followed by a parallel approach using *MPI* library, with an increase in performance specially with high-resolution images. Based on the serial method, a CUDA-based parallel approach was also proposed [5], showing increases in performance, even with low-resolution images. For instance, images provided by AFM

usually present low resolution, usually less than 1024x1024 pixels.

Considering a MxM matrix y representing a real noisy and blurred AFM image x, where the blurring can be described by a $[-N \dots N]$ normalized matrix b, which represents the tip geometry, the restored image $\hat{x}$ is assumed to be the best representative when the Tikhonov's regularization functional

$$Q(\hat{x}) = \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left[y_{i,j} - \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot \hat{x}_{i+k,j+l} \right]^2 + \alpha S \quad (1)$$

reaches its minimum, where α is an adjustable smoothing factor (low-pass filtering) to guarantee functional convergence along the iterative process and S represents a general contrast function (q-discrepancy) acting as a high-pass filter, given by

$$S = \frac{1}{1+q} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left[\hat{x}_{i,j} \left(\frac{\hat{x}_{i,j}^q - \overline{x}_{i,j}^q}{q} \right) - \overline{x}_{i,j}^q (\hat{x}_{i,j} - \overline{x}_{i,j}) \right] \quad (2)$$

where $q \geq 0$, and usually $1 \leq q \leq 2$. When $q \to 0$ the entropy functional is configured, and when $q = 1$ equation (2) can be simplified to

$$S = \frac{1}{2} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} (\hat{x}_{i,j} - \overline{x}_{i,j})^2 \quad (q = 1) \quad (3)$$

The $\overline{x}$ term represents a reference image used to smooth the additive noise, which can be a constant value or, even, the previous restored image. It was shown by [6] that the use of the previous estimation as a reference reduces significantly the number of iterations to reach the minimum of $\hat{x}$. According to [4], the step values for the next estimated image is given by

$$\Delta \hat{x}_{r,s}|^{t+1} = \frac{F_{r,s}|^t}{F_{m,n}|^t_{m=r;n=s}} \quad (4)$$

where t is the iteration counter, and

$$\begin{aligned} F_{r,s} &= \frac{\partial Q(\hat{x})}{\partial \hat{x}_{r,s}} \\ &= -2 \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left(y_{i,j} - \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot \hat{x}_{i+k,j+l} \right) b_{r-i,s-j} \\ &+ \frac{\alpha}{q} (\hat{x}_{i,j}^q - \overline{x}_{i,j}^q) \end{aligned} \quad (5)$$

$$\begin{aligned} F_{n,m} = \frac{\partial F_{r,s}}{\partial \hat{x}_{m,n}} &= 2 \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot b_{r-m+k,s-n+l} \\ &+ \alpha \hat{x}_{r,s}^{q-1} \cdot \delta(r,m) \cdot \delta(s,n) \end{aligned} \quad (6)$$

are, respectively, the first and second derivatives of $Q(\hat{x})$, where δ is the Kronecker's delta. It can be observed that three convolutions are performed in equations (5) and (6) involving the blurring matrix b. In order to keep the energy constant after each convolution, specially in the regions next to the edges, the data must be normalized by dividing each pixel element by the sum of matrix b terms; this approach is necessary considering the reduced number of matrix b elements covering the borders. In all other cases the sum of elements is equal to one, as the blurring matrix is already normalized. Once the new estimate is obtained, the new value of the functional is calculated and compared to the previous.

III. ALGORITHM

The main goal of our work is the optimization of a parallel restoration algorithm, executed in GPGPU using CUDA library, based on an optimized serial version, also proposed in this work. The first step is the identification of variables and/or matrices that can be calculated at initialization, as is the case of the matrices b and $normB$. The last is a MxM one, and contains the normalization terms for the convolutions using b for each image element, and is given by:

$$normB_{r,s} = \sum_{i=max(-N,-r)}^{min(N,M-r-1)} \sum_{j=max(-N,-s)}^{min(N,M-s-1)} b_{i,j} \quad (7)$$

Equation (6) defines the second derivative of $Q(\hat{x})$, that involves a convolution of b over itself, and can also be calculated as pre-processing. However, equation (4) defines the second derivative restrictions, which are given by the conditions $(m = r)$ and $(n = s)$. Such conditions simplifies equation (6) to the following form:

$$F_{n,m} = 2 \sum_{k=-N}^{N} \sum_{l=-N}^{N} b_{k,l} \cdot b_{k,l} + \alpha \hat{x}_{r,s}^{q-1} \quad (8)$$

where the sums do not represent a convolution of b over itself, but only a sum of its square terms, and can be calculated just once, being stored in another MxM matrix, since it depends only on the blurring matrix and the position of the image pixel, to which the matrix will be applied. It is important to point out that the normalization must be considered for each image pixel in order to keep energy next the edges. Making

$$S2BB_{r,s} = 2 \sum_{k=max(-N,-r)}^{min(N,M-r-1)} \sum_{l=max(-N,-s)}^{min(N,M-s-1)} \left(\frac{b_{k,l}}{normB_{r,s}} \right)^2 \quad (9)$$

equation (8) can be rewritten as:

$$F_{n,m} = S2BB_{r,s} + \alpha \hat{x}_{r,s}^{q-1} \quad r = m, s = n \quad (10)$$

In the implementations of the algorithms described in [4] and [5], a vector and a single value were used instead of the matrices $normB$ and $S2BB$ respectively, in order to increase the restoration performance. However, this approach leads to

errors next the edges, so the matrices were used to keep the quality of results without loss in performance, as will be shown.

Considering equation (1), the term inside the sums represents the difference between the real image and the current estimate convolved by b. As this expression is also used in (5), its results must be kept as a matrix in order to increase performance. In such case, the matrix YBX is defined as

$$YBX_{i,j} = y_{i,j} - \frac{\sum_{k=max(-N,-i)}^{min(N,M-i-1)} \sum_{l=max(-N,-j)}^{min(N,M-j-1)} b_{k,l} \cdot \hat{x}_{i+k,j+l}}{normB_{r,s}} \tag{11}$$

From equation (5), it can be inferred that the sum will only return non-zero values when the differences $(r-i)$ and $(s-j)$ are within the dimensions of the blurring matrix, which means that the limits of the sum can be reduced to $[-N \ldots N]$. The resulting expression for $F_{r,s}$ is

$$F_{r,s} = -2\frac{\sum_{k=max(-N,-i)}^{min(N,M-i-1)} \sum_{l=max(-N,-j)}^{min(N,M-j-1)} b_{k,l} YBX_{r+k,s+l}}{normB_{r,s}} + \frac{\alpha}{q}(\hat{x}^q_{i,j} - \overline{x}^q_{i,j}) \tag{12}$$

Finally, the term inside the sums in (2) can be kept in another matrix, called MS:

$$MS_{i,j} = \hat{x}_{i,j}\left(\frac{\hat{x}^q_{i,j} - \overline{x}^q_{i,j}}{q}\right) - \overline{x}^q_{i,j}(\hat{x}_{i,j} - \overline{x}_{i,j}) \tag{13}$$

The sums of the elements of MS and the square of the elements of YBX are used to compute S and then $Q(\hat{x})$. The second step for the algorithm optimization is the identification of the operations that can be performed together in the same loop. Two secondary loops were identified as possible candidates. One computes the first and second derivatives, $F_{r,s}$ and $F_{m,n}$, obtaining the new estimate for the restored image. The other loop performs the convolution of the estimated image with b, calculates the differences relative to y (storing the results as YBX for the next iteration), the terms of MS and finally the sum of the elements of both matrices in order to obtain the value of S and $Q(\hat{x})$. The proposed optimization is shown in Algorithm 1.

On the proposed algorithm, in addition to the convergence verification of $Q(\hat{x})$, the absolute or relative difference may be assessed and compared to a tolerance value, since convergence can take a long time. Usually, a small number of iterations is fixed between 50 and 100, and the values of α and q are changed in order to speed up convergence and produce better results. It can be observed that almost all steps of the algorithm can be executed in parallel. The only exceptions are on lines 9 to 12 and 24 to 27, where the sums of the terms of YBX and MS are performed to obtain the values of S and $Q(\hat{x})$, which were executed serially. Thus, the proposed parallel algorithm is

Algorithm 1 Tikhonov's serial image restoration

```
1:  initialization:
2:  Calculates b
3:  Calculates normB
4:  Calculates S2BB
5:  initial calculation of Q(x̂):
6:  for i = 0 → (M − 1); j = 0 → (M − 1) do
7:      Calculates YBX_{i,j}
8:      Calculates MS_{i,j}
9:      sumYBX2 ← YBX^2_{i,j}
10:     sumMS ← MS_{i,j}
11: Calculates S
12: Calculates Q_0(x̂)
13: Main loop:
14: for t = 0 → (N − 1) do
15:     Calculates new estimate:
16:     for r = 0 → (M − 1); s = 0 → (M − 1) do
17:         Calculates F_{r,s}
18:         Calculates F_{n,m}
19:         Calculates x̂_{r,s}
20:     Calculates new value of Q(x̂):
21:     for i = 0 → (M − 1); j = 0 → (M − 1) do
22:         Calculates YBX_{i,j}
23:         Calculates MS_{i,j}
24:         sumYBX2 ← YBX^2_{i,j}
25:         sumMS ← MS_{i,j}
26:     Calculates S
27:     Calculates Q_1(x̂)
28:     Verifying convergence:
29:     if Q_1(x̂) < Q_0(x̂) then return
30:     Assign values for next iteration:
31:     x̄ ← x̂
32:     Q_0(x̂) ← Q_1(x̂)
33: return
```

designed to keep all needed information in the GPU memory, minimizing data exchange between CPU and GPU, which would strongly compromise execution performance. In this first parallel algorithm, all data is stored in GPU global memory, and the execution is performed by two kernels. The first calculates the new estimate for $\hat{x}$, corresponding to lines 16 to 19, and the second calculates the elements of the matrices YBX and MS, corresponding to lines 6 to 8 and 21 to 23. These two matrices are then moved to the CPU memory, and will be used to serially calculate the sums of their terms and finally $Q(\hat{x})$. The convergence check is also performed by the CPU, and when reached, the resulting image is retrieved back to the CPU memory. The obtained results, using both algorithms, are shown in the next section.

IV. Results

Aiming to demonstrate the performance improvements obtained by the proposed serial and parallel algorithms, in terms of speedups, we executed both previous implementations, as well as the proposed ones, over a test image created for this purpose, with different resolutions. The comparison between the execution times and performance gains with respect to the first serial approach are shown in table I and Fig. 3, where *1st serial* and *1st parallel* refer to the approaches proposed by

[4] and [5], respectively, and *2nd serial* and *2nd parallel* are the ones we propose in this work. The test image (Fig. 2a) was blurred using a $5x5$ Gaussian matrix (Fig. 2b), simulating the interaction of the AFM tip with the sample surface, and then recovered (Fig. 2c) using the parameters α and q set to 1. As the performance of the restoration does not depend on the characteristics of the image, but only on the image dimensions and the parameters employed (which could speed up or slow down convergence), the results obtained reflect the performance of the algorithms on restoring any kind of image with the same dimensions and using the same parameters. It is up to the user to set the parameters in order to balance performance and quality of restoration.

The tests were performed on a 2.2 GHz Intel i7 processor with 6 GB DDR3 of RAM memory, and a GeForce GT 555M GPU board with 2 GB of global memory, running under a CUDA [7] driver version 5.0. Both serial algorithms were built using double-precision floating point variables, as its execution is faster on 64-bit architectures, using compiling optimization parameters -O2. All four algorithms were compiled with Microsoft 2012 C compiler under Windows 7. The execution of the parallel approach proposed by [4] using *MPI* library was not evaluated in this work, since our main goal is to optimize parallel execution in GPGPU.

The kernel block size of both parallel algorithms was set to allow the computation of one line of the image by each block, and each pixel by one thread. On both approaches the image data processed by the GPU was stored in the global memory. The resulting times reported correspond to the average of 10 executions with 100 iterations for each value, and the standard deviation of each mean was less than 5%, in the worst case.

Table I shows that the proposed serial algorithm is faster than its previous parallel version. This is because in the previous parallel approach, all data was stored in CPU memory, and for every iteration, it was moved back and forth to the GPU memory, for every single iteration, resulting in a great amount of overhead for the GPU execution time, strongly compromising the performance. On the other hand, in the proposed serial algorithm, all operations that can be executed together were grouped in the same loop, taking advantage of the same iteration counters.

The structure of the serial algorithm has made easy for us to implement the parallel version. Its main feature is to keep every needed information stored in GPU global memory, and only exchange the matrices used to compute S and $Q(\hat{x})$. Furthermore, these matrices were allocated on CPU memory as *pinned memory*, reducing the transfer time. Comparing the proposed parallel approach to the optimized serial implementation, we achieved speedups up to 7.2x. When compared to the previous implementations, described in [4] and [5], we achieved speedups up to 23.6x (Fig. 3).

Although convergence check is a necessary step to ensure the correctness of the calculations, it may be avoided when performing real-time restorations, with a fixed and small number of iterations. In this situation, the user would set the parameters to obtain better convergence, and make a visual inspection of the restoration quality, on the screen. This can be a useful approach to restore images acquired from high-speed AFM at the rate of one image per second. Table II exhibits the execution times and gains of performance of *2nd parallel* approach without convergence checks. The parameters α and q, and the test image used to evaluate the execution were the same used on the previous evaluation. As can be seen, the gains obtained with this approach increased about 100%, reaching execution performances up to 43.2 faster than the *1st serial* approach. This came from the reduction on the data transfer between CPU and GPU necessary to calculate $Q(\hat{x})$ and to verify convergence.

The restoration quality of the four approaches was also evaluated by analysing the values of the functional $Q(\hat{x})$, and the results are shown in table III. The differences observed are due to the use of the matrices for determining the values of $normB$ and $S2BB$ for each image pixel, instead of using a vector and a single value. The effect of these changes are noted only near the edges of the images, where the relative differences are bigger on smaller images, 3.8% lower on 128×128 images against 1.6% lower on 1024×1024, as the blurring matrix employed was the same for all images, and these differences tend to increase with higher dimension ones. Fig. 4 exhibits the effects of both approaches on the borders of the images. Although *2nd serial* and *2nd parallel* approaches use matrices to calculate $normB$ and $S2BB$, small differences

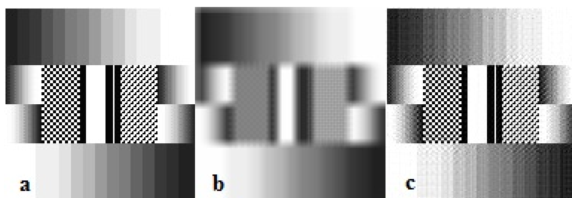

Fig. 2. Results of restoring a 128x128 pixels sample image, where a) is the original image, b) is the original image after blurring, and c) is the restored image

TABLE I. MEAN EXECUTION TIMES IN ms

Image dimensions	*1st serial*	*2nd serial*	*1st parallel*	*2nd parallel*
128x128	542	161	340	31
256x256	2203	636	1100	102
512x512	8276	2680	3987	370
1024x1024	34713	10598	15279	1554

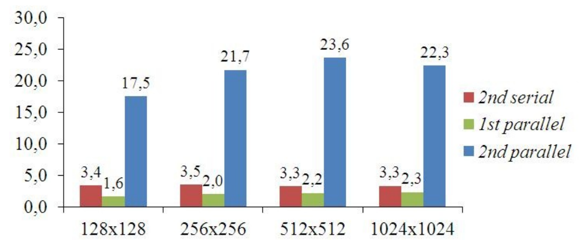

Fig. 3. Speedups for different image dimensions

TABLE II. EXECUTION TIMES IN ms AND SPEEDUPS OF *2nd parallel* WITHOUT CONVERGENCE CHECKS

Image dimensions	*execution time*	*performance gain*
128x128	14	38,7
256x256	51	43,2
512x512	202	43,2
1024x1024	941	36,9

TABLE III. VALUES OF THE FUNCTIONAL $Q(\hat{x})$

Image dimensions	*1st serial*	*2nd serial*	*1st parallel*	*2nd parallel*
128x128	0.33322	0.32076	0.33322	0.32076
256x256	0.74086	0.72881	0.74082	0.72878
512x512	1.22613	1.21240	1.22559	1.21187
1024x1024	1.25004	1.23240	1.24883	1.23070

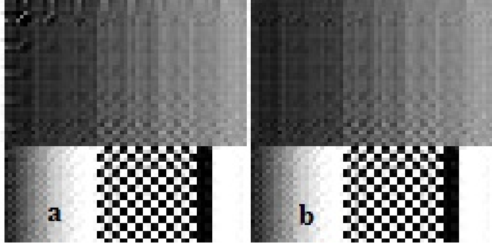

Fig. 4. Detail of the upper left corners of two restored 128x128 pixels images, where a) is the restored image using the algorithm employed in [5], and b) is the restored image using the proposed algorithm.

are observed between the values of $Q(\hat{x})$ obtained. These are due to the use of different floating-point variables. While the serial algorithm employs double-precision floating-point variables, the parallel one employs single-precision variables.

The quality of the restoration of real images is demonstrated in Fig. 5, where a biological sample image (Fig. 5a), obtained with an AFM, is restored using the proposed parallel algorithm (Fig. 5b). As the restoration parameters are changed, details of the image are revealed, and the user is able to set the ones that better fit his visual perception and make convergence faster. In the case of the sample image, restoration has taken less than 4 milliseconds, due to the use of suitable parameters. With the use of the parallel algorithm, the parameters can be defined dynamically, during image acquisition and display.

V. CONCLUSION

The proposed algorithm based on Tikhonov's regularization functional has shown to be suitable to restore images very fast, producing better results, in comparison to the previous approaches, when parallelized in GPGPU using CUDA library. The parallel algorithm has shown to be faster than the optimized serial implementation, and to be suitable for real-time image restorations, applicable to a high-speed AFM [2], specially after removing convergence verifications. For future work, we intend to simplify the use of the matrices $normB$ and $S2BB$, and to use shared and constant memories of the GPU to improve execution performance, in addition to the automatic determine of the blurring matrix based on the AFM tip geometry.

ACKNOWLEDGMENT

The authors would like to thank the National Institute of Metrology, Quality and Technology (Inmetro) for motivating the student Klaus Natorf Quelhas to pursue his master degree, and also Augusto Almeida Garcia for providing the codes of the two previous implementations.

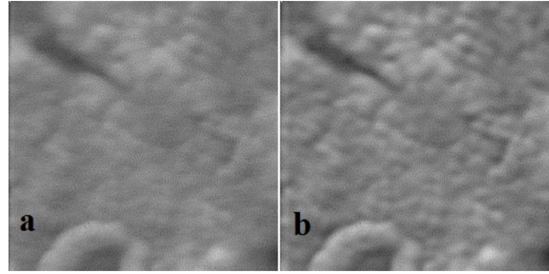

Fig. 5. Restoration of a biological AFM 256x256 pixels image using the proposed algorithm, where a) is the sample image, and b) is the restored one. Courtesy of Cidade [3].

REFERENCES

[1] Binnig G, Quate CF and Gerber CH, Atomic Force Microscope, Phys. Rev. Lett. 56(9), pp. 930-933, 1986.

[2] Fantner GE, Hegarty P, Kindt JH, Schitter G, Cidade GAG and Hansma PK, Data acquisition system for high speed atomic force microscopy. Review of Scientific Instruments, v. 76, n.2, p. 026118, 2005.

[3] Cidade, G. A. G., Silva Neto, A. J., Roberty, N. C. (2003), Restauração de Imagens com Aplicações em Biologia e Engenharia Problemas Inversos em Nanociência e Nanotecnologia, Notes in Applied Mathematics, vol. 1, So Carlos, SP.

[4] Stutz D, Silva Neto AJ and Cidade, GAG, Parallel Computation Approach for the Restoration of AFM Images based on the Tikhonov Regularization Method. Microscopy and Microanalysis, v. 11, p. 22-25, 2005.

[5] A. G. Almeida, Restauração de imagens de microscopia de força atômica com uso de regularização de Tikhonov via processamento em GPU, Universidade do Estado do Rio de Janeiro, PhD Thesis, 2012.

[6] V. Furtado, Avaliação do uso do funcional de Tikhonov com uma família de funções de regularização a um parâmetro para a restauração de imagens biológicas, Universidade do Estado do Rio de Janeiro, 2002.

[7] NVIDIA Corporation, CUDA C Programming Guide, 2012

[8] D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors. A Hands-on Approach, Elsevier Inc., 2010

[9] J. Sanders, E. Kandrot, Cuda By Example. An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2011

[10] W. W. Hwu, GPU Computing Gems. Emerald Edition, Elsevier Inc., 2011

Apêndice B

Implementação dos algoritmos desenvolvidos

B.1 Restauração paralela

Código B.1: restauracaoCUDA.cu

```
//Programa realiza a restauração de imagens por minimização do funcional de
//regularização de Tikhonov através de paralelização em GPU
//Baseado nos trabalhos de Almeida (2013), Stutz (2009) e Cidade (2000)
//Klaus Natorf Quelhas - Junho/2014. Atualizado: Setembro/2014

#include<stdio.h>
#include<stdlib.h>
#include <math.h>
#include "CImg.h"
#include "kTimer.h"
#include <cuda.h>
#include<cuda_runtime.h>
#pragma comment(lib, "cudart")

using namespace std;
using namespace cimg_library;

//Definição de macros
#define maximo(a,b) (a > b) ? a : b
#define minimo(a,b) (a < b) ? a : b
#define dZERO 1e-270
#define isZERO(x) fabs(x)<dZERO
#define isUM(x) fabs(x-1.0)<dZERO

//Função monta a matriz de borramento B
//@ B: ponteiro para a matriz B
//@ dim_B: dimensão da matriz B
//@ lim_B: limites de iteração da matriz B ( dim(B) = 2*lim_B+1 )
//@ v: variância
void monta_B(float *B, int dim_B, int lim_B, float v)
{
    //contadores de iterações
  int i,j,k,l;
  //somatório dos elementos de B
```

```
35    float sB = 0;

37    //monta matriz de borramento
38    for(j=0, l=-lim_B; l<=lim_B; j++, l++)
39      for(i=0, k=-lim_B; k<=lim_B; i++, k++)
40        sB += ( B[ j*dim_B + i] = exp( -( (k*k+l*l)/(2*v) ) ) );
41    //normaliza a matriz de borramento
42    for(j=0; j<dim_B; j++)
43      for(i=0; i< dim_B; i++)
44        B[ j*dim_B + i] /= sB;
45  }

47  //Função monta as matrizes normaB e S2BB
48  //@ B: matriz de borramento
49  //@ normaB: matriz contendo os valores de normalização de convoluções com B
50  //@ S2BB: Matriz de itens de 2*B(k,l)*B(r-m+k, s-m+l) = 2*B(k,l)^2
51  //@ dim_B: dimensão da matriz B
52  //@ lim_B: limites de iteração da matriz B ( dim(B) = 2*lim_B+1 )
53  //@ w: largura da matriz (imagem)
54  //@ h: altura da matriz (imagem)
55  void monta_normaB_S2BB(float *B, float *normaB, float *S2BB, int dim_B, int lim_B,
56      int w, int h)
57  {
58      //contadores de iterações
59    int i, j, k, l;
60    //limites de iterações
61    int ik, il, fk, fl;

63    for(j=0; j<h; j++)
64      for(i=0; i<w; i++)
65      {
66        //inicializa valores
67        normaB[j*w + i] = 0;
68        S2BB[j*w + i] = 0;
69        //determina limites de iteração de B
70        ik = maximo(-lim_B, -i);
71        fk = minimo(lim_B, (w-i-1) );
72        il = maximo(-lim_B, -j);
73        fl = minimo(lim_B, (h-j-1) );
74        //calcula normaB(i,j)
75        for(l=il; l<=fl; l++)
76          for(k=ik; k<=fk; k++)
77            normaB[j*w + i] += B[(l+lim_B)*dim_B + (k+lim_B) ];
78        //calcula S2BB(i,j)
79        for(l=il; l<=fl; l++)
80          for(k=ik; k<=fk; k++)
81            S2BB[j*w + i] += ( B[(l+lim_B)*dim_B + (k+lim_B) ] * B[(l+lim_B)*dim_B
82                + (k+lim_B) ] ) / ( normaB[j*w + i] * normaB[j*w + i] );
83        S2BB[j*w + i] *= 2;
84      }
85  }

87  //Função calcula a convolução BX, as diferenças y-BX, e os termos da matriz M_S para
88  //cálculo do termo de regularização S e o funcional Q(x) em GPU.
89  //1 thread por elemento (w threads e h blocos)
90  //@ Y: Imagem amostral
91  //@ X: Imagem estimada
92  //@ Xr: Imagem de referência
93  //@ w: largura das imagens
```

```
//@ h: altura das imagens
//@ B: matriz de borramento
//@ dim_B: dimensão da matriz B
//@ lim_B: limites de iteração da matriz B ( dim(B) = 2*lim_B+1 )
//@ normaB: matriz de normalização da convolução BX
//@ alfa: parâmetro de regularização
//@ q: parâmetro do termo de regularização S
//(return) M_S: matriz de elementos do somatório para cálculo de S (dependente de q)
//(return) YBX: Matriz contendo  as diferenças Y-BX
__global__ void conv_diff_GPU(float *Y, float *X, float *Xr, int w, int h, float *B,
int dim_B, int lim_B, float *normaB, float alfa, float q, float *M_S, float *YBX)
{
    //contadores de iterações
  int i, j, k, l;
  //limites de iterações
  int ik, il, fk, fl;
  //elemento da convolução de B sobre X
  float BX;

  //ponteiros para iterações:
  //elemento corrente da matriz de borramento
  float *pB;
  //elemento corrente de X durante convolução com B(k,l)
  float *pXkl;

  //determina posições dos ponteiros
  i = threadIdx.x;
  j = blockIdx.x;
  int elem = j * w + i;

  //determina limites de iteração de B em l
  il = maximo(-lim_B, -j);
  fl = minimo(lim_B, (h-j-1) );

  //inicializa valores
  BX=0;

  //determina limites de iteração de B em k
  ik = maximo(-lim_B, -i);
  fk = minimo(lim_B, (w-i-1) );

  //determina ponteiros para convolução
  pB = B + (il + lim_B) * dim_B + (ik + lim_B);
  pXkl = X + elem + il * w + ik;

  //Calcula convolução BX em (i,j)
  for( l = il; l <= fl; l++, pB += (ik - fk + dim_B - 1), pXkl += (ik - fk + w - 1) )
  {
    for( k = ik; k <= fk; k++, pB++, pXkl++)
    {
      BX += (*pB) * (*pXkl);
    }
  }
  BX /= normaB[elem];

  //Calcula YBX(i,j)
  YBX[elem] = Y[elem] - BX;

  //Calcula M_S(i,j)
```

```
  if( isZERO(q) )
    M_S[elem] = -( X[elem] - Xr[elem] - Xr[elem] * log( X[elem] / Xr[elem] ) );
  else if( isUM (q) )
    M_S[elem] = ( X[elem] - Xr[elem] ) * ( X[elem] - Xr[elem] );
  else
    M_S[elem] = X[elem] * ( ( pow( X[elem], q ) - pow( Xr[elem], q ) ) / q )
        - Xr[elem] * ( X[elem] - Xr[elem] );
}


//Função  calcula as derivadas Frs e Fnm, e determina a nova estimativa para a imagem X
//em GPU. 1 thread por elemento (w threads e h blocos)
//@ Xt0: Imagem estimada atual
//@ Xr: Imagem de referência
//@ w: largura das imagens
//@ h: altura das imagens
//@ B: matriz de borramento
//@ dim_B: dimensão da matriz B
//@ lim_B: limites de iteração da matriz B ( dim(B) = 2*lim_B+1 )
//@ normaB: matriz de normalização da convolução BX
//@ S2BBB: Matriz de itens de 2*B(k,l)*B(r-m+k, s-m+l) = 2*B(k,l)^2
//@ YBX: Matriz contendo  as diferenças Y-BX
//@ alfa: parâmetro de regularização
//@ q: parâmetro do termo de regularização S
//@ gama: fator de atenuação da corrreção
//(return) Xt1: Imagem estimada corrigida
__global__ void aplica_DeltaX_GPU(float *Xt0, float *Xr, int w, int h, float *B,
    int dim_B, int lim_B, float *normaB, float *S2BB, float *YBX, float alfa, float q,
    float gama, float *Xt1)
{
    //contadores de iterações
  int r, s, k, l;
  //limites de iterações
  int ik, il, fk, fl;
  //elemento da convolução de B sobre YBX
  float YBXB;
  //derivada primeira de Q(x)
  float Frs;
  //derivada segunda de Q(x)
  float Fnm;

  //ponteiros para iterações
  //elemento corrente da matriz de borramento
  float *pB;
  //elemento corrente de Y durante convolução com B(k,l)
  float *pYBXkl;

  //determina posições dos ponteiros
  r = threadIdx.x;
  s = blockIdx.x;
  int elem = s * w + r;

  //determina limites de iteração de B em l
  il = maximo(-lim_B, -s);
  fl = minimo(lim_B, (h-s-1) );

  //inicializa valores
  YBXB=0;
```

```
212    //determina limites de iteração de B em k
213    ik = maximo(-lim_B, -r);
214    fk = minimo(lim_B, (w-r-1) );

216    //determina ponteiros para convolução
217    pB = B + (il + lim_B) * dim_B + (ik + lim_B);
218    pYBXkl = YBX + elem + il * w + ik;

220    //Calcula convolução YBXB em (i,j)
221    for( l = il; l <= fl; l++, pB += (ik - fk + dim_B - 1), pYBXkl += (ik - fk + w - 1) )
222    {
223      for( k = ik; k <= fk; k++, pB++, pYBXkl++)
224      {
225        YBXB += (*pB) * (*pYBXkl);
226      }
227    }
228    YBXB /= normaB[elem];

230    //Calcula Frs
231    if( isZERO(q) )
232      Frs = -2 * YBXB + alfa * log( Xt0[elem] / Xr[elem] );
233    else if( isUM(q) )
234      Frs = -2 * YBXB + alfa * ( Xt0[elem] - Xr[elem] );
235    else
236      Frs = -2 * YBXB + (alfa/q)*( pow( ( Xt0[elem] ), q ) - pow( ( Xr[elem] ), q ) ) ;

238    //Calcula Fnm
239    if( isZERO(q) )
240      Fnm = S2BB[elem] + alfa / Xr[elem];
241    else if( isUM(q) )
242      Fnm = S2BB[elem] + alfa;
243    else
244      Fnm = S2BB[elem] + alfa * pow( ( Xt0[elem] /*+ 1.0E-10*/ ), (q-1) );

246    //Calcula nova estimativa de X
247    Xt1[elem] = Xt0[elem] - gama*( Frs/Fnm );

249    //ajusta valores para evitar overflow em imagens de 8 bits. Remover em caso de
250    //imagens em ponto flutuante.
251    if( Xt1[elem] < 0 )
252      Xt1[elem]=0;
253    if( Xt1[elem] > 1 )
254      Xt1[elem]=1;
255  }

257  //Programa realiza a restauração de imagens de microscopia de força atômica empregando
258  //o método de Jacobi simplificado e realimentação
259  //sintaxe ( restauracaoCUDA [imagem] [D] [v] [N] [TOL] [alfa] [q] [gama]
260  // restauracaoCUDA: Arquivo executável
261  // imagem:           Arquivo contendo a imagem
262  // D:                Dimensão da matriz de borramento (n impar maior que 1)
263  // v:                Variância da matriz de borramento (positivo maior que zero)
264  // N:                Número máximo de iterações do laço principal
265  // TOL:              Tolerância relativa do processo de restauração
266  // alfa:             Parâmetro de regularização (maior que zero)
267  // q:                Parâmetro do termo de regularização S (maior ou igual a zero)
268  // gama:             Fator de atenuação (positivo maior que zero)
269  int main(int argc, char *argv[])
270  {
```

```
271    //*******************************************************************
272    // Atribuição de parâmetros por linha de comando
273    //*******************************************************************

275    if(argc < 9)
276    {
277      printf("\nERRO! Numero de parametros insuficiente\n");
278      return 1;
279    }

281      //arquivo contendo a imagem
282    char *filename = argv[1];
283    //dimensão da matriz de borramento (ímpar maior que 1)
284    int dim_B = atoi(argv[2]);
285    //limites da matriz de borramento a partir do centro (-dim_B ... dim_B)
286    int lim_B = (dim_B - 1) / 2;
287    //variância da matriz de borramento (positivo maior que zero)
288    float v = atof(argv[3]);
289    //número máximo de iterações do laço principal
290    int Nt = atoi(argv[4]);
291    //tolerância do processo de restauração
292    float tol = atof(argv[5]);
293    //parâmetro de regularização (positivo maior que zero)
294    float alfa = atof(argv[6]);
295    //parâmetro do termo de regularização S (positivo maior ou igual a zero)
296    float q = atof(argv[7]);
297    //fator de atenuação (positivo maior que zero)
298    float gama = atof(argv[8]);

300    if( !(dim_B % 2) || (dim_B < 3) )
301    {
302      printf("Matriz de borramento deve ter dimensão positiva, ímpar e maior que
303          1! (%d)\n", dim_B);
304      return 1;
305    }

307    //*******************************************************************
308    // Declaração de varáveis de controle e processamento
309    //*******************************************************************

311    //*************** dados da imagem a ser restaurada ***************
312    //imagem a ser restaurada
313    CImg<unsigned char> img;
314    //largura da imagem
315    int width;
316    //altura da imagem
317    int height;
318    //ponteiro para os dados da imagem a ser processada
319    unsigned char *img_data;

321    //********** dados independentes das iterações (CPU e GPU) **********
322    //matriz de borramento
323    float *B, *d_B;
324    //elementos de normalização da matriz de borramento (para efeito de bordas)
325    float *normaB, *d_normaB;
326    //Matriz de itens de 2*B(k,l)*B(r-m+k, s-m+l) = 2*B(k,l)^2
327    float *S2BB, *d_S2BB;

329    //********** dados de imagens (CPU e GPU) **********
```

```
330    //imagem amostral
331    float *Y, *d_Y;
332    //imagem de referência
333    float *Xr, *d_Xr;
334    //estimativa da imagem atual
335    float *Xt0, *d_Xt0;
336    //próxima estimativa da imagem
337    float *Xt1, *d_Xt1;

339    //***** dados de convoluções e diferenças (CPU e GPU) *****
340    //matriz contendo (Y-B*X)
341    float *YBX, *d_YBX;
342    //matriz de elementos do somatório para cálculo de S (dependente de q)
343    float *M_S, *d_M_S;
344    //fator constante do termo de regularização (dependente de q)
345    float f_S;
346    //resíduos de Y-BX
347    float r0, r1;
348    //somatório dos elementos de M_S
349    float soma_S;
350    //somatório dos quadrados dos elementos de YBX
351    float soma_YBX2;
352    //termo de regularização S
353    float S;
354    //valor atual do funcional de regularização
355    float Qt0;
356    //valor novo do funcional de regularização
357    float Qt1;
358    //Erro na restauração
359    float erro;
360    //ponteiro auxiliar
361    float *paux;

363      //********** outros **********
364      //contador de iterações do laço principal
365    int t;
366    contador de iterações
367    int i;
368    //contador de tempo (kTimer.h)
369    CHRTimer timer;
370    //verificador de erro CUDA
371    cudaError err;

373    //*****************************************************************
374    // Alocação de memória e atribuição de variáveis (CPU e GPU)
375    //*****************************************************************

377    //OBS: O CImg organiza os dados dos canais rgb separadamente dentro de um mesmo vetor.
378    //Os primeiros w*h elementos correspondem ao canal vermelho, os próximos correspondem
379    //aos canais verde e azul, sequencialmente
380    img.assign(filename);
381    width = img.width();
382    height = img.height();
383    img_data = img.data();
384    img.display("Imagem de entrada");

386    B = (float *) malloc( dim_B * dim_B * sizeof(float) );
387    err = cudaMalloc( &d_B, dim_B * dim_B * sizeof(float) );
388    if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}
```

```
390   normaB = (float *) malloc( width * height * sizeof(float) );
391   err = cudaMalloc( &d_normaB,  width * height * sizeof(float) );
392   if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}

394   S2BB = (float *) malloc( width * height * sizeof(float) );
395   err = cudaMalloc( &d_S2BB,  width * height * sizeof(float) );
396   if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}

398   Y = (float *) malloc( width * height * sizeof(float) );
399   err = cudaMalloc( &d_Y,  width * height * sizeof(float) );
400   if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}

402   Xr = (float *) malloc( width * height * sizeof(float) );
403   err = cudaMalloc( &d_Xr,  width * height * sizeof(float) );
404   if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}

406   Xt0 = (float *) malloc( width * height * sizeof(float) );
407   err = cudaMalloc( &d_Xt0,  width * height * sizeof(float) );
408   if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}

410   Xt1 = (float *) malloc( width * height * sizeof(float) );
411   err = cudaMalloc( &d_Xt1,  width * height * sizeof(float) );
412   if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}

414   err = cudaHostAlloc( (void**)&YBX, width*height * sizeof(float), cudaHostAllocMapped);
415   if(err) {printf("\nErro na alocação de dados! (cudaHostAlloc)\n"); return;}
416   err = cudaMalloc( &d_YBX,  width * height * sizeof(float) );
417   if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}

419   err = cudaHostAlloc((void**)&M_S, width*height * sizeof(float), cudaHostAllocMapped);
420   if(err) {printf("\nErro na alocação de dados! (cudaHostAlloc)\n"); return;}
421   err = cudaMalloc( &d_M_S,  width * height * sizeof(float) );
422   if(err) {printf("\nErro na alocação de dados! (cudaMalloc)\n"); return;}

424   //Verifica alocação de memória
425   if( !B || !normaB || !S2BB || !Y || !Xr || !Xt0 || !Xt1 || !YBX || !M_S )
426   {
427     printf("Erro de alocação de memória!\n");
428     return 1;
429   }

431   //******************************************************************
432   //IMPLEMENTANDO FUNÇÕES SEPARADAS PARA MONTAGEM DAS MATRIZES INDEPENDENTES
433   //COMO ELAS SERÃO EXECTUADAS APENAS UMA VEZ, ELAS PODERÃO SER EXECUTADAS
434   //NO SETUP E ALOCADAS NA GPU
435   //******************************************************************

437   // Inicializações

439   //Atribui valores iniciais às imagens. Imagem de referência será a imagem de entrada,
          uma vez que o método de realimentação será empregado
440   for(i = 0; i < (width * height); i++)
441   {
442     Y[i] = Xt0[i] = (float) img_data[i] / 255.0;
443   }
444   //Atribui ponteiro, em função da técnica de realimentação:
445   Xr = Xt0;
```

```
447     //Transfere imagens para a GPU
448     err = cudaMemcpy( d_Y, Y, width * height * sizeof(float), cudaMemcpyHostToDevice);
449     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

451     err = cudaMemcpy( d_Xt0, Xt0, width * height * sizeof(float), cudaMemcpyHostToDevice);
452     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

454     //Atribui ponteiro, em GPU, em função da técnica de realimentação:
455     d_Xr = d_Xt0;

457     //Monta matriz de borramento
458     monta_B(B, dim_B, lim_B, v);

460     //Monta matrizes de normalização e S2BB ( B(k,l)^2 )
461     monta_normaB_S2BB(B,normaB, S2BB, dim_B, lim_B, width, height);

463     //Transfere matrizes de borramento, norma e S2BB para a GPU
464     err = cudaMemcpy( d_B, B, dim_B * dim_B * sizeof(float), cudaMemcpyHostToDevice);
465     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

467     err = cudaMemcpy( d_normaB, normaB, width * height * sizeof(float),
            cudaMemcpyHostToDevice);
468     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

470     err = cudaMemcpy( d_S2BB, S2BB, width*height * sizeof(float), cudaMemcpyHostToDevice);
471     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

473     //calcula o termo constante de S:
474     if( isZERO(q) )
475       f_S = -1;
476     else if( isUM(q) )
477       f_S = 0.5;
478     else
479       f_S = 1/(1+q);

481     //CRIA IMAGEM MONOCROMATICA (CImg)
482     CImg <unsigned char> img_output(width,height, 1, 1);
483     unsigned char *img_output_data = img_output.data();

485     //**********************************************************************
486     // Inicia processo iterativo
487     //**********************************************************************
488     int nblocks = height;
489     int nthreads = width;
490     t=0;

492     //Cálculo das convoluções e diferenças
493     conv_diff_GPU <<<nblocks, nthreads>>> (d_Y, d_Xt0, d_Xr, width, height, d_B, dim_B,
            lim_B, d_normaB, alfa, q, d_M_S, d_YBX);

495     //Transfere matrizes YBX e M_S de volta para a CPU
496     err = cudaMemcpy( YBX, d_YBX, width * height * sizeof(float), cudaMemcpyDeviceToHost);
497     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

499     err = cudaMemcpy( M_S, d_M_S, width * height * sizeof(float), cudaMemcpyDeviceToHost);
500     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

502     //calcula somatórios, r0 e Qt0:
503     soma_YBX2 = soma_S = r0 = 0;
```

```
504   for( i = 0; i < width * height; i++)
505   {
506     soma_S += M_S[i];
507     soma_YBX2 += YBX[i] * YBX[i];
508     r0 += YBX[i];
509   }

511   //Calcula S
512   S = f_S * soma_S;

514   //Calcula Q(x) e valores de retorno
515   Qt0 = soma_YBX2 + alfa * S;

517   //exibe resultado inicial
518   printf("\nIteração= %d\nQ(x)= %f\nr= %f\n",t, Qt0, r0);

520   //*****************************************************************
521   // Loop principal
522   //*****************************************************************
523   timer.Start();
524   for(t=0; t<Nt; t++)
525   {
526     //Aplica correção e calcula nova estimativa de X
527     aplica_DeltaX_GPU<<<nblocks,nthreads>>>(d_Xt0, d_Xr, width, height, d_B, dim_B,
          lim_B, d_normaB, d_S2BB, d_YBX, alfa, q, gama, d_Xt1);

529     //Calcula novas convoluções e diferenças
530     conv_diff_GPU <<<nblocks, nthreads>>> (d_Y, d_Xt1, d_Xr, width, height, d_B, dim_B,
          lim_B, d_normaB, alfa, q, d_M_S, d_YBX);

532     //Transfere matrizes YBX e M_S de volta para a CPU
533     err = cudaMemcpy( YBX, d_YBX, width*height * sizeof(float), cudaMemcpyDeviceToHost);
534     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}
535     err = cudaMemcpy( M_S, d_M_S, width*height * sizeof(float), cudaMemcpyDeviceToHost);
536     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

538     //Calcula somatórios e Qt1:
539     soma_YBX2 = soma_S = r1 = 0;
540     for( i = 0; i < width * height; i++)
541     {
542       soma_S += M_S[i];
543       soma_YBX2 += YBX[i] * YBX[i];
544       r1 += YBX[i];
545     }

547     //Calcula S
548     S = f_S * soma_S;
549     //Calcula Q(x) e valores de retorno
550     Qt1 = soma_YBX2 + alfa * S;

552     //Calcula o erro da restauração
553     erro = (isZERO(Qt1)) ? fabs(Qt1-Qt0)     // Erro absoluto
554       : fabs((Qt1-Qt0)/Qt1);             // Erro relativo

556     //Verifica resultados parciais
557     if(Qt1>Qt0)
558     {
559       //Se solução diverge, logo o resultado provém da iteração anterior
560       printf("\nMinimo alcancado!\n");
```

```
561         Qt1 = Qt0;
562         r1 = r0;
563         //troca ponteiros de Xt0 e Xt1
564         paux = d_Xt0;
565         d_Xt0 = d_Xt1;
566         d_Xt1 = paux;
567         break;
568       }
569       if( erro < tol )              //verifica se tolerância foi alcançada
570       {
571         printf("\nTolerancia alcancada!\n");
572         break;
573       }

575       //Associa valores para a próxima iteração
576       Qt0 = Qt1;
577       r0 = r1;
578       //Faz da imagem recém restaurada referência (realimentação)
579       d_Xr=d_Xt1;
580       //Atribui Xt1 para Xt0, e vice-versa
581       paux = d_Xt0;
582       d_Xt0 = d_Xt1;
583       d_Xt1 = paux;
584     }
585     timer.Stop();

587     //********************************************************************
588     // Exibição de resultados e imagem restaurada
589     //********************************************************************
590     printf("
        ********************************************************************************\n"
        );
591     printf("Resultaods:\nIteração: %d\t Q(X)=%.5f\t Residuo=%.5f\n", t, Qt1, r1);
592     printf("Tempo decorrido: %.3f\n",timer.GetElapsedAsMiliseconds() );
593     printf("
        ********************************************************************************\n"
        );

595     //transfere a imagem restaurada de volta para a CPU
596     err = cudaMemcpy( Xt1, d_Xt1, width * height * sizeof(float), cudaMemcpyDeviceToHost);
597     if(err) {printf("\nErro na transferencia de dados! (cudaMemcpy)\n"); return;}

599     //atribui valores para imagem de saída
600     for(i=0; i<width*height; i++)
601       img_output_data[i] = (unsigned char) ceil( Xt1[i] * 255 );
602     img_output.display("imagem recuparada");
603     img_output.save_bmp("rec.bmp");

605     //********************************************************************
606     // Liberação de memória
607     //********************************************************************

609     free(B);
610     free(normaB);
611     free(S2BB);
612     free(Y);
613     free(Xt0);
614     free(Xt1);
615     cudaFreeHost(YBX);
```

```
616     cudaFreeHost(M_S);
617     cudaFree(d_B);
618     cudaFree(d_normaB);
619     cudaFree(d_S2BB);
620     cudaFree(d_Y);
621     cudaFree(d_Xt0);
622     cudaFree(d_Xt1);
623     cudaFree(d_YBX);
624     cudaFree(d_M_S);

626     system("pause");
627     return 0;
628 }
```

B.2 Cabeçalho da classe *DBufferAFM*

Código B.2: DBufferAFM.h

```
1  //Cabeçalho define a classe DBufferAFM, que contem o buffer duplo de dados de medição do
       AFM
2  //Classe deve ser instanciada e manipulada através de uma interface C# -> C++
3  //Klaus Natorf Quelhas: Set/2013. Atualizado: Nov/2014
4  #ifndef DBUFFERAFM_H
5  #define DBUFFERAFM_H

7  #include"GLAFMDisplay.h"
8  #include "kTimer.h"
9  #include "CImg.h"

11 using namespace cimg_library;

13 //Tipos de varredura
14 enum varredura {IDA, VOLTA, IDA_E_VOLTA};
15 //Tipos de mostragem
16 enum amostragem {CONTACT, NON_CONTACT, TAPPING};
17 //Códigos de erro
18 // 1 - erro de alocação
19 // 2 - ausência de memória alocada
20 // 3 - erro na transferência para a GPU (cudaMemcpy)
21 // 4 - ausencia de imagem processada
22 // 5 - imagem incompleta no buffer duplo
23 // 6 - Ausencia de instância OpenGL

25 class DBufferAFM
26 {
27 //********** Declaração de atributos privados **********
28 private:
29   //********** dados em CPU **********
30   //ponteiros para os buffers duplos para tratamento na GPU
31   float *db_photo, *db_z, *db_dx, *db_dy;
32   //ponteiros que irão apontar para a metade dos buffers
33   float *h_db_photo, *h_db_z, *h_db_dx, *h_db_dy;
34   //ponteiros para vetores de informações de posições de ida e volta nas imagens
35   int *ida, *volta;
36   //posição inicial da imagem no buffer
37   int ini;
```

```
38   //imagens processadas (topografia, erro e combinada)
39   float *img_photo, *img_z, *img;
40   //informações de imagens enviadas (largura, altura e número de dados enviados por
         vetor, em números de elementos)
41   int dim_x, dim_y, data_size;
42   //define uma margem de segurança para eventuais inconsistencias de dados, em numero de
         elementos
43   int safety_margin;
44   //dimensões das imagens a serem processadas em numero de elementos
45   int w, h;
46   //dimensões da amostra em x e y, em micrometros
47   float size_x, size_y;
48   //sensibilidade do fotodetector, em micrometros por volt
49   float photo_sens;
50   //sensibilidade da plataforma piezoelétrica no eixo z, em micrometros por volt
51   float z_sens;
52   //tipo de varredura para tratamento dos dados (somente ida, somente volta ou ida e
         volta)
53   varredura v_type;
54   //tipo de amostragem (contato, não-contato ou tapping. Implementação futura)
55   amostragem a_type;
56   //variável contendo código de erro
57   int erro;
58   //numero de conjuntos de dados no buffer (0, 1 ou 2), para critérios de processamento
59   int nbuffer;

61   //********** dados em GPU **********
62   //ponteiros para vetores de informações de posições de ida e volta nas imagens
63   int *d_ida, *d_volta;
64   //ponteiros para os vetores enviados à GPU
65   float *d_photo, *d_z;
66   //imagens processadas na GPU (topografia, erro e combinada)
67   float  *d_img_photo, *d_img_z, *d_img;

69   //********** atributos de controle de execução **********
70   //define se o processamento será em GPU ou CPU
71   bool cudaEnabled;
72   //define se há área de dados alocada (ou está tudo alocado ou nada está)
73   bool alocado;
74   //define se a renderização pode ser realizada automaticamente após o envio de dados
75   bool autorender;
76   //define se a normalização pode ser realizada automaticamente após o envio de dados
77   bool autonorm;
78   //define se a exibição da imagem pode ser realizada automaticamente após o envio de
         dados (CImg)
79   bool autodisp;

81   //********** imagem (CImg) **********
82   //imagem combinada
83   CImg<unsigned char> *img_bmp;
84   //display da imagem combinada
85   CImgDisplay img_disp;

87   //******* tempos de execução, em milisegundos *******
88   //Contadores de tempo
89   CHRTimer timer, timer_total;
90   //tempo de realização de swap de memória e de carregamento de dados
91   float t_send_and_swap;
92   //tempo de preenchimento dos vetores descritores
```

```
93	  float t_find;
94	  //tempo de carregamento de dados na GPU
95	  float t_load;
96	  //tempo de execução do kernel em GPU
97	  float t_kernel;
98	  //tempo de descarregamento de dados para a CPU
99	  float t_unload;
100	  //tempo de renderização da imagem
101	  float t_render;
102	  //tempo total de execução
103	  float t_total;

105	  //********** display opengl **********
106	  //Objeto para processamento OpenGL
107	  GLAFMDisplay *gldisplay;
108	  //array de posições
109	  float *vertexArray, *d_vertexArray;
110	  //array de cores
111	  float *colorArray, *d_colorArray;

113	public:
114	  //********** construtores e destrutor **********

116	  //Construtor padrão. Apenas inicializa atributos, sem alocar memória
117	  DBufferAFM();

119	  //Construtor da Classe. Aloca dados conforme informações enviadas
120	    //dx - largura das imagens a serem enviadas
121	    //dy - altura das imagens a serem enviadas
122	    //dsize - numero de elementos enviados por vetor
123	    //sx - tamanho real da amostra em x, em micrometros
124	    //sy - tamanho real da amostra em y, em micrometros
125	    //sphoto - sensbilidade do fotodetector, em micrometros por volt
126	    //sz - sensibilidade da plataforma piezoelétrica no eixo z, em micrometros por volt
127	  DBufferAFM(int dx, int dy, int dsize, float sx, float sy, float sphoto, float sz);

129	  //Construtor da Classe. Aloca dados conforme informações enviadas
130	    //dx - largura das imagens a serem enviadas
131	    //dy - altura das imagens a serem enviadas
132	    //dsize - numero de elementos enviados por vetor
133	    //dsize - numero de elementos enviados por vetor
134	    //sx - tamanho real da amostra em x, em micrometros
135	    //sy - tamanho real da amostra em y, em micrometros
136	    //sphoto - sensbilidade do fotodetector, em micrometros por volt
137	    //sz - sensibilidade da plataforma piezoelétrica no eixo z, em micrometros por volt
138	    //v - tipo de varredura
139	    //a - tipo de amostragem
140	  DBufferAFM(int dx, int dy, int dsize, float sx, float sy, float sphoto, float sz,
	      varredura v, amostragem a);

142	  //Destrutor. Desaloca dados
143	  ~DBufferAFM();

145	    //********** Métodos públicos **********

147	    //Método inicializa variáveis e aloca memória
148	    //ATENÇÃO: AO REALIZAR O SETUP, OS DADOS PRÉ ALOCADOS SERÃO PERDIDOS
149	    //dx - largura das imagens a serem enviadas
150	    //dy - altura das imagens a serem enviadas
```

```
151         //dsize − numero de elementos enviados por vetor
152         //sx − tamanho real da amostra em x, em micrometros
153         //sy − tamanho real da amostra em y, em micrometros
154         //sphoto − sensbilidade do fotodetector, em micrometros por volt
155         //sz − sensibilidade da plataforma piezoelétrica no eixo z, em micrometros por volt
156     void setup(int dx, int dy, int dsize, float sx, float sy, float sphoto, float sz);

158     //Método inicializa variáveis e aloca memória
159         //ATENÇÃO: AO REALIZAR O SETUP, OS DADOS PRÉ ALOCADOS SERÃO PERDIDOS
160         //dx − largura das imagens a serem enviadas
161         //dy − altura das imagens a serem enviadas
162         //dsize − numero de elementos enviados por vetor
163         //dsize − numero de elementos enviados por vetor
164         //sx − tamanho real da amostra em x, em micrometros
165         //sy − tamanho real da amostra em y, em micrometros
166         //sphoto − sensbilidade do fotodetector, em micrometros por volt
167         //sz − sensibilidade da plataforma piezoelétrica no eixo z, em micrometros por volt
168         //v − tipo de varredura
169         //a − tipo de amostragem
170     void setup(int dx, int dy, int dsize, float sx, float sy, float sphoto, float sz,
            varredura v, amostragem a);

172     //Método ajusta a margem de segurança das imagens para o valor desejado
173     void set_safetymargin(int m);

175     //Método altera o método de varredura para fins de pre−processamento
176     void set_scanning_type(varredura v);

178     //Método habilita o uso da biblioteca CUDA
179     void enableCUDA();

181     //Método desabilita o uso da biblioteca CUDA
182     void disableCUDA();

184     //Método habilita a renderização em OpenGL automática das imagens após o envio de
            dados
185     void enableAutoRender();

187     //Método desabilita a renderização em OpenGL automática das imagens após o envio de
            dados
188     void disableAutoRender();

190     //Método habilita a normalização automática das imagens após o envio de dados
191     void enableAutoNorm();

193     //Método desabilita a normalização automática das imagens após o envio de dados
194     void disableAutoNorm();

196     //Método habilita a exibição das imagens após o envio de dados (CImg)
197     void enableAutoDispCImg();

199     //Método desabilita a exibição das imagens após o envio de dados (CImg)
200     void disableAutoDispCImg();

202     //Método retorna o código de erro ativo. Caso novas operações sejam feitas, o código
            será perdido
203     int getErrCode();

205     //Função envia conjunto de dados para o buffer duplo
```

```
206     //p - vetor contendo dados do fotodetetor
207     //z - vetor contendo dados da correção da plataforma piezoelétrica
208     //dx - vetor contendo dados da direção de varredura em x
209     //dy - vetor contendo dados da direção de varredura em y
210     int send_data(float *p, float *z, float *dx, float *dy);

212     //Método retorna o ponteiro para a imagem
213     //imagem consiste em um array W x H de float representando a posição em z no espaço de
            cada ponto
214     //retorna 0 (ponteiro nulo) caso não haja imagem disponível (nbuffer < 2), ou em caso
            de erro
215     //esta pode não estar normalizada, depende da chamada do método normaliza()
216     float *img_afm();

218     //Método chama a função de normalização da imagem
219       //retorna 0 em caso de sucesso e um código de erro, caso contrário
220     int normaliza_img();

222     //********** imagens CImg **********

224     //Método chama a função de exibição de imagem
225       //retorna 0 em caso de sucesso e um código de erro, caso contrário
226     int display_CImg();

228     //********** imagens OpenGL **********

230     //Método renderiza a imagem em OpenGL
231     //retorna 0 em caso de sucesso e um código de erro, caso contrário
232     int display_GL();

234     //********** tempos de execução (public) **********

236     //Método reinicia todos os contadores de tempo
237     void reset_timers();

239     //Método retorna o tempo de swap e carregamento de dados
240     float send_and_swap_time();

242     //Método retorna o tempo de preenchimento dos vetores descritores
243     float find_time();

245     //Método retorna o tempo de carregamento de dados na GPU
246     float load_time();

248     //Método retorna o tempo de execução do kernel
249     float kernel_time();

251     //Método retorna o tempo de descarregamento de dados para a CPU
252     float unload_time();

254     //Método retorna o tempo de renderização
255     float render_time();

257     //Método retorna o tempo total de execução
258     float total_time();



261       //********** Métodos privados **********
262   private:
```

```
263     //********** operações em memória **********

265     //Método aloca buffers e variaveis em CPU e em GPU
266     //retorna 0 em caso de sucesso e 1 em caso de erro)
267     int alocaMem();

269     //Método desaloca áreas de dados
270     void desalocaMem();

272     //Método realiza o swap de dados no buffer duplo
273     //retorna 0 em caso de sucesso e 2 em caso de erro
274     int swap_data();

276     //********** processamento da imagem **********

278     //Método define os parâmetros e processa a imagem na CPU ou na GPU
279     //retorna 0 em caso de sucesso
280     int pre_process();

282     //Método busca os pontos de inicio, para cada linha da varredura, para a ida e a volta
283     //retorna codigo de erro 5 caso a imagem esteja incompleta (caso o inicio da mesma
            esteja apos a metade do buffer , o que significa que provavelmente a imagem estará
             incompleta
284     int find_ini();

286     //********** tratamento e exibição de imagens **********

288     //Método monta os vetores de vertices, cores e indices para construção das imagens em
            OpenGL
289     int monta_arrays();

291     //Método realiza a chamada para a renderização da imagem obtida em OpenGL
292     int renderGL();

294     //Método realiza a normalização dos dados da imagem para exibição em CImg ou em outro
            meio 2D
295     void normaliza();

297     //Método exibe a imagem obtida (CImg)
298     void display();

300 };
301 #endif
```

B.3 Cabeçalho da classe *GLAFMDisplay*

Código B.3: GLAFMDisplay.h

```
1 #ifndef GLAFMDISPLAY_H
2 #define GLAFMDISPLAY_H

4 #include<Windows.h>
5 #include<stdio.h>
6 #include<stdlib.h>
7 #include<math.h>
8 #include<GL\freeglut.h>
```

```
9  #include<thread>
10 #include<mutex>

12 using namespace std;

14 class GLAFMDisplay
15 {
16 private:

18     //thread responsavel por manter o loop de visualização
19   thread *glthread;

21   //lança a thread responsavel pela inicialização do display
22   void setupGLAFMDisplay();

24 public:

26     //construtor da classe GLAFMDisplay. Aloca vetor contendo a imagem
27     //a ser exibida e inicializa a janela de display 3D
28     //w: largura da imagem, em pixels
29     //h: altura da imagem, em pixels
30     //psize_x: largura da imagem, em micrometros
31     //psize_y: altura da imagem, em micrometros
32     //vArray: array de vértices
33     //cArray: array de cores
34   GLAFMDisplay(int w, int h, float psize_x, float psize_y, float *vArray, float *cArray)
       ;

36   //Destrutor. desaloca imagem e finaliza thread
37   ~GLAFMDisplay();

39     //ativa barreira para evitar o acesso concorrente aos arrays de vértices e cores
40   void lock();

42   //desativa barreira
43   void unlock();

45   //chama a função de display
46   void render();
47 };

49 #endif
```