UNIVERSIDADE DE SÃO PAULO

FFCLRP - DEPARTAMENTO DE COMPUTAÇÃO E MATEMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

**Componente mediador para computação GPU de imagens
médicas no 3D SLICER**

Alexandre Freitas Duarte

Dissertação apresentada à Faculdade de Filosofia,
Ciências e Letras de Ribeirão Preto da USP, como
parte das exigências para a obtenção do título de
Mestre em Ciências, Área: Computação Aplicada.

Ribeirão Preto - SP
2018

**Componente mediador para computação GPU de imagens
médicas no 3D SLICER**

Alexandre Freitas Duarte

ORIENTADOR: PROF. DR. LUIZ OTAVIO MURTA JUNIOR

Ribeirão Preto - SP
2018

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

Duarte, Alexandre Freitas

Componente mediador para computação GPU de imagens médicas no 3D SLICER. Ribeirão Preto, 2018.

114 p. : il. ; 30 cm

Dissertação de Mestrado, apresentada à Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto/USP. Área de concentração: Computação Aplicada.

Orientador: Junior, Luiz Otavio Murta.

1. GPU. 2. Computação Paralela. 3. OPENCL. 4. 3D SLICER. 5. GPGPU. 6. Processamento de Imagens Médicas.

Mediator component for GPU computation of medical images in 3D SLICER

Alexandre Freitas Duarte

SUPERVISOR: PROF. DR. LUIZ OTAVIO MURTA JUNIOR

Ribeirão Preto - SP
2018

Alexandre Freitas Duarte

Componente mediador para computação GPU de imagens médicas no 3D SLICER

Dissertação apresentada à Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto da USP, como parte das exigências para a obtenção do título de Mestre em Ciências, Área: Computação Aplicada.

Aprovado em: 30 de julho de 2018

Banca Examinadora:

Prof. Dr. Luiz Otavio Murta Junior
Orientador

Prof. Dr. Joaquim Cezar Felipe

Prof. Dr. Paulo Mazzoncini de Azevedo Marques

Prof. Dr. Ricardo José Ferrari

Ribeirão Preto - SP
2018

AGRADECIMENTOS

Primeiramente, agradeço aos meus pais, Analice de Paula Mello Duarte e Nemésio Freitas Duarte, a toda a minha família (namorada, irmão, primos, tios e avós) o apoio que me deram ao longo destes três anos, com todas as oportunidades e incentivos para concluir este mestrado.

Agradeço imensamente ao orientador Luiz Otavio Murta Junior. Não foi simplesmente um orientador, mas, sim, um grande amigo. Confiou desde o início no meu trabalho, criticou e questionou quando precisou, mas ajudou, motivou e incentivou durante toda a sua orientação.

Agradeço aos amigos e colegas de mestrado por compartilharem experiências e pelos conselhos psicológicos em mesas de bares; grupos de bate-papo; e churrascos organizados.

Por fim, agradeço aos professores do Programa de Pós-Graduação de Computação Aplicada, do Departamento de Computação e Matemática da FFCLRP-USP, a ajuda, o apoio e motivações ao longo de todo meu mestrado.

“A persistência é o menor caminho do êxito.”

Charles Chaplin

RESUMO

O desenvolvimento e a utilização de ambientes computacionais como na área de processamento e gerenciamento de imagens, aliados à evolução das novas tecnologias, têm contribuído significativamente para o avanço da prática médica, e sendo cada vez mais frequentes e necessários na rotina de inúmeros procedimentos médicos que utilizam exames de imagem. Durante a última década, o interesse no uso de unidades de processamento gráfico (GPUs) para aplicações de propósito em geral (GPGPU) aumentou. Conjuntamente com o crescente uso de GPUs, ocorre o surgimento de novas tecnologias de desenvolvimento, que facilitam a computação paralela em GPU para os desenvolvedores. Mesmo assim, a capacidade de criação de aplicações GPGPU de alto desempenho, que utilizam a capacidade total do *hardware* GPU e seus benefícios, exige um domínio de *software* e *hardware* muito elevado. A fim de resolver a computação intensiva presente em inúmeros métodos de processamento de imagem médicas, neste contexto, o objetivo principal deste trabalho é propor e desenvolver um componente mediador para a computação GPU, este integrado ao *software* 3D SLICER. Com isso, espera-se que o componente proporcione aos pesquisadores ganhos em suas aplicações em relação ao tempo de execução, o projeto também será uma base e auxílio para as novas propostas de trabalhos que seguirem a mesma linha de pesquisa. Na validação deste trabalho um estudo experimental foi realizado, no qual alguns métodos de processamento de imagens foram selecionados e modificados para o uso da computação paralela em GPUs, seus tempos de execução foram avaliados e comparados. Por fim, os resultados demonstraram que o uso do componente mediador em um contexto experimental pode proporcionar ganhos altos no tempo de execução para alguns métodos alcançando tempos de até 40x mais rápido que o método sequencial, em contrapartida um dos métodos teve um ganho menos expressivo atingindo apenas 4x vezes desse aumento de tempo.

Palavras-chave: GPU, computação paralela, OPENCL, 3D SLICER, GPGPU, processamento de imagens médicas

ABSTRACT

The development and use of computational environments such as image processing and management, combined with the evolution of new technologies, have contributed significantly to the advancement of medical practice, and being increasingly frequent and necessary in the routine of numerous medical procedure that use imaging tests. Over the last decade, interest in the use of GPUs for general purpose applications (GPGPU) has increased. Along with the increasing use of GPUs, there are new development technologies that facilitate parallel computing in GPUs for developers. Even so, the ability to build high-performance GPGPU applications, which utilize the full capacity of GPU hardware and its benefits, requires a very high software and hardware domain. In order to solve the intensive computation present in numerous medical image processing methods, in this context, the main objective of this work is to propose and develop a mediator component for the GPU computation, this one integrated with the software 3D SLICER, with this, it is expected that the component provides the researchers with gains in their applications in relation to the execution time, the project will also be a base and aid for new projects that follow the same line of research. In the validation of this work was carried out an experimental study, where some selected techniques were applied and modified for the use of parallel computing in GPUs, their execution times were evaluated and compared. Finally, the results demonstrated that the use of the mediator component in an experimental context can provide high gains in execution time for some methods reaching times the of up to 40x times faster than the sequential method, in contrast one of the methods had a less expressive gain reaching only 4x times that time.

Keywords: GPU, paralel computing, OPENCL, 3D SLICER, GPGPU, medical image processing

LISTA DE FIGURAS

Figura 1. Comparação entre a arquitetura de uma CPU e uma GPU Fonte: NVIDIA CUDA (2013).....	28
Figura 2. Relação entre <i>host</i> e <i>devices</i> (<i>INTRODUCTION TO OPENCL,2010</i>).	30
Figura 3. Compilação do Kernel em tempo de execução (<i>INTRODUCTION TO OPENCL,2010</i>).	32
Figura 4. Mapeamento da tarefa em um dispositivo (<i>INTRODUCTION TO OPENCL,2010</i>).	33
Figura 5. Código sequencial em C++, para soma de dois vetores. (Fonte: elaborada pelo autor)	34
Figura 6. Código do <i>Kernel</i> baseado em C99, para a soma de dois vetores. (Fonte: elaborada pelo autor).....	34
Figura 7. Exemplo de espaço de índice (NDRANGE) unidimensional. (Fonte: elaborada pelo autor).....	35
Figura 8. Código sequencial em C++, para multiplicação de matrizes. (Fonte: elaborada pelo autor).....	35
Figura 9. Código do <i>Kernel</i> , para multiplicação de matrizes. (Fonte: elaborada pelo autor) ..	36
Figura 10. Exemplo de espaço de índice (NDRANGE) bidimensional. (Fonte: elaborada pelo autor).....	36
Figura 11. Exemplo de espaço de índice (NDRange) tridimensional. (Fonte: elaborada pelo autor).....	37
Figura 12. Modelo de gerenciamento de memória do OPENCL (<i>INTRODUCTION TO OPENCL,2010</i>).	38
Figura 13. Tela do 3D SLICER.	44
Figura 14. Planos Ortogonais Anatômicos.	45
Figura 15. Etapas da primeira fase. (Fonte: elaborada pelo autor)	56
Figura 16. Etapas da segunda fase. (Fonte: elaborada pelo autor)	58
Figura 17. Etapas da terceira fase. (Fonte: elaborada pelo autor)	59
Figura 18. Arquitetura das tecnologias.	63
Figura 19. Código C++ sequencial de multiplicação de matriz. (Fonte: elaborada pelo autor).	71
Figura 20. <i>Kernel</i> para multiplicação de matrizes. (Fonte: elaborada pelo autor)	71

Figura 21. Resultado do tempo de execução para multiplicação de matrizes. (Fonte: elaborada pelo autor).....	72
Figura 22. Módulo criado para o experimento de média móvel. (Fonte: elaborada pelo autor)	74
Figura 23. Elementos gráficos e seus parâmetros do filtro de média móvel. (Fonte: elaborada pelo autor).....	75
Figura 24. Comparação no tempo de execução do filtro de média móvel (1). (Fonte: elaborada pelo autor).....	76
Figura 25. Comparação no tempo de execução do filtro de média móvel (2). (Fonte: elaborada pelo autor).....	77
Figura 26. Elementos gráficos e seus parâmetros do filtro de dilatação e erosão. (Fonte: elaborada pelo autor)	79
Figura 27. Comparação no tempo de execução do filtro de dilatação (1). (Fonte: elaborada pelo autor).....	80
Figura 28. Comparação no tempo de execução do filtro de dilatação (2). (Fonte: elaborada pelo autor).....	81
Figura 29. Elementos gráficos e seus parâmetros do filtro de difusão anisotrópico. (Fonte: elaborada pelo autor)	83
Figura 30. Comparação no tempo de execução do filtro de difusão anisotrópico (1). (Fonte: elaborada pelo autor)	84
Figura 31. Envio do <i>Kernel</i> para GPU a) a aplicação <i>host</i> enviando os parâmetros de entrada para a GPU b) a aplicação <i>host</i> recuperando a imagem resultante. (Fonte: elaborada pelo autor)	85

LISTA DE TABELAS

Tabela 1. Tabela de equivalência entre as tecnologias OPENCL e CUDA.	53
Tabela 2. Dados numéricos do tempo de execução para multiplicação de matrizes. (Fonte: elaborada pelo autor)	73
Tabela 3. Dados numéricos do tempo de execução do filtro de média móvel (1). (Fonte: elaborada pelo autor)	76
Tabela 4. Dados numéricos do tempo de execução do filtro de média móvel (2). (Fonte: elaborada pelo autor)	77
Tabela 5. Dados numéricos do tempo de execução do filtro de dilatação (1). (Fonte: elaborada pelo autor)	80
Tabela 6. Dados numéricos do tempo de execução do filtro de dilatação (2). (Fonte: elaborada pelo autor)	81
Tabela 7. Dados numéricos do tempo de execução do filtro de difusão anisotrópico (2). (Fonte: elaborada pelo autor)	84

LISTA DE SIGLAS

Sigla	Descrição
CMAKE	<i>Cross Platform Make</i>
CP	Computação Paralela
CPU	<i>Central Processing Unit</i>
CT	<i>Computed Tomography</i>
CUDA	<i>Compute Unified Device Architecture</i>
DICOM	<i>Digital Imaging and Communication in Medicine</i>
FLOPS	<i>Floating-Point Operations Per Second</i>
FPGA	<i>Field Programmable Gate Array</i>
GPU	<i>Graphics Processing Unit</i>
GPGPU	<i>General Purpose Graphics Processing Unit</i>
GUI	<i>Graphical User Interface</i>
IRM	Imagem por Ressonância Magnética
ITK	<i>Insight Segmentation and Registration Toolkit</i>
NRRD	<i>Nearly Raw Raster Data</i>
OPENCL	<i>Open Computing Language</i>
OPENGL	<i>Open Graphics Library</i>
PET	<i>Positron Emission Tomography</i>
SDK	<i>Software Development Kit</i>
ULA	Unidade Lógica e Aritmética

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO	23
1.1 Considerações Iniciais	23
1.2 Contextualização	23
1.3 Objetivos	25
1.4 Organização do Trabalho	25
CAPÍTULO 2 - UNIDADE DE PROCESSAMENTO GRÁFICO	27
2.1 Considerações Iniciais	27
2.2 Unidade de Processamento Gráfico	27
2.3 O Modelo OPENCL	30
2.3.1 <i>Host, Kernel e Compute Device</i>	30
2.3.2 Esquema de Paralelização	32
2.3.3 Exemplos de NDRANGES	34
2.3.4 Gerenciamento de Memória	37
2.3.5 O <i>Framework</i> OPENCL.....	38
2.4 Considerações Finais	39
CAPÍTULO 3 - SOFTWARE 3D SLICER.....	41
3.1 Considerações Iniciais	41
3.2 Introdução.....	41
3.3 Visão Geral do 3D SLICER	43
3.4 Desenvolvimento de Módulos	46
3.5 Considerações Finais	47
CAPÍTULO 4 - TRABALHOS RELACIONADOS	49
4.1 Considerações Iniciais	49
4.2 Processamento de imagens utilizando o OPENCL.....	49
4.3 Componentes e Frameworks Baseados em OPENCL.....	50
4.4 Trabalhos Baseados em CUDA.....	52
4.5 Comparação CUDA e OPENCL	53
4.6 Considerações Finais	54
CAPÍTULO 5 - MATERIAL E MÉTODOS.....	55
5.1 Considerações Iniciais	55

5.2 Tipo de Pesquisa	55
5.3 Procedimentos Metodológicos	56
5.3.1 Concepção e Preparação da Pesquisa	56
5.3.2 Análise da Solução e Desenvolvimento	57
5.3.3 Validação.....	58
CAPÍTULO 6 - COMPONENTE MEDIADOR.....	61
6.1 Considerações Iniciais	61
6.2 Análise da solução	61
6.3 Estudo das Tecnologias Relacionadas	63
6.4 Visão Geral do Componente.....	65
6.5 Considerações Finais	67
CAPÍTULO 7 - RESULTADOS.....	69
7.1 Considerações Iniciais	69
7.2 Multiplicação de Matrizes (Experimento 1)	70
7.2.1 Resultados do Experimento 1	72
7.2.2 Conclusão do Experimento 1	73
7.3 Filtro de Média Móvel (Experimento 2).....	74
7.3.1 Resultados do Experimento 2.....	75
7.3.2 Conclusão do Experimento 2	77
7.4 Filtro Morfológico de Erosão/Dilatação (Experimento 3)	78
7.4.1 Resultados do Experimento 3.....	79
7.4.2 Conclusão do Experimento 3	82
7.5 Filtro de Difusão Anisotrópico (Experimento 4)	82
7.5.1 Resultados do Experimento 4.....	83
7.5.2 Conclusão do Experimento 4	86
7.6 Considerações Finais	86
CAPÍTULO 8 - CONCLUSÃO	89
8.1 Considerações Iniciais	89
8.2 Considerações Finais do Trabalho.....	89
8.3 Limitações do Trabalho	91
8.4 Trabalhos futuros	92
8.5 Publicação	93
REFERÊNCIAS	95
APÊNDICES	101

Capítulo 1

INTRODUÇÃO

1.1 Considerações Iniciais

Neste capítulo apresenta-se uma visão geral da proposta do trabalho, destacando o contexto no qual o projeto de pesquisa está inserido, bem como as principais motivações e justificativas para seu desenvolvimento.

1.2 Contextualização

Ao longo dos anos, tem-se acompanhado uma forte evolução tecnológica e computacional, e cada vez mais aplicações estão exigindo o desenvolvimento de processadores mais rápidos. Isso ocorre porque a maioria dessas aplicações requer um custo computacional elevado para processar grandes quantidades de dados (COHEN & GARLAND, 2009; SHI et. al., 2010).

A Computação Paralela (CP) surgiu como uma opção de resolver de forma eficiente os problemas da computação intensiva em diversas áreas do conhecimento humano. Neste cenário, novas tendências arquiteturais de *hardware* têm sido desenvolvidas. Soluções com vários processadores em uma mesma placa e processadores com vários núcleos de processamento são a nova tendência tecnológica na atualidade. A CP pode trazer muitos

benefícios devido a suas características que podem garantir uma aceleração na execução de métodos computacionais (SMITH, 2011).

Paralelamente à evolução do hardware, novas tecnologias estão surgindo, destacando-se duas delas: Compute *Unified Device Architecture*¹ (CUDA) e *Open Computing Language*² (OPENCL). Modelos de programação como CUDA e OPENCL permitem que usuários desenvolvedores possam usufruir de forma mais fácil do poder de uma GPU (PAULA et al., 2013b). Tanto CUDA quanto OPENCL permitem a implementação de aplicações eficientes, executadas de forma paralela em GPUs, mas mesmo assim estas ainda são mais difíceis de ser programadas comparadas às aplicações tradicionais, executadas de forma sequencial na CPU (PAULA et al., 2013a). Isso ocorre porque ainda exige um grande esforço de compreensão de algoritmos, adequação e modificação do algoritmo sequencial, a organização e paralelização dos dados em *cores* e *threads* geralmente devem ser gerenciados manualmente pelo programador (PAULA, 2014a).

A área de processamento de imagens médicas é essencial em muitos procedimentos de diagnósticos e terapêuticos realizados em estabelecimentos de saúde, e também está sempre contribuindo para o avanço da prática médica. Em imagens médicas existem as modalidades de aquisição de imagens, tais como raios X, tomografia computadorizada, ressonância magnética nuclear, cintilografia gama, angiografia e ultrassonografia. Em relação ao processamento da imagem, existe uma vasta gama de aplicações que podem ser desenvolvidas, envolvendo coregistro de imagens, extração de características, métodos de segmentação, recuperação baseada em conteúdo, métodos comparativos, métodos de compressão, arquivamento e comunicação de imagens médicas. Geralmente, no contexto das imagens tridimensionais, estas aplicações costumam demandar alto custo computacional em relação ao tempo de execução, exigindo algoritmos mais eficientes. Uma alternativa para tal problema da computação intensiva é a computação paralela (DHAWAN, 2011).

Observam-se na literatura ótimos resultados em relação ao ganho no tempo de execução em alguns métodos específicos, os estudos comparam métodos tradicionais executados de forma sequencial na CPU com métodos modificados executados de forma paralela em GPUs (ZHANG, 2010; KONG et al., 2010; TOKUDA, 2015).

Nessa perspectiva, diante dos desafios identificados para o uso eficiente de GPUs em sistemas gerais somados ao contexto de processamento de imagens médicas tridimensionais, o componente foi construído para proporcionar a prática da computação paralela, também com

¹ CUDA Zone - http://www.nvidia.com.br/object/cuda_home_new_br.html

² OPENCL Overview - <https://www.khronos.org/opencl/>

o objetivo de minimizar a complexidade para a criação de aplicações pelo desenvolvedor. Adicionalmente, para a validação do componente, um estudo experimental foi realizado, no qual métodos foram implementados para a prática da computação paralela usando GPU, seus tempos de execução foram avaliados e comparados. A seguir é formalizado o objetivo do projeto.

1.3 Objetivos

O presente projeto tem por objetivo principal a construção de um componente mediador de *software* que suporte o uso de GPUs dentro do 3D SLICER e que possa ser compatível com GPUs de diferentes fabricantes. Com este componente de *software*, espera-se possibilitar que processamentos complexos de imagens médicas tridimensionais tenham uma redução nos seus tempos de execução, dando mais agilidade a pesquisas e procedimentos médicos que utilizam exames de imagem. Para alcançar o objetivo principal da pesquisa, alguns objetivos específicos foram considerados ao longo deste trabalho, entre eles:

- **Análise da solução:** pesquisar e estudar os aspectos da solução que viabilize a integração entre as tecnologias envolvidas no trabalho;
- **Estudo experimental:** (1) implementar e avaliar o ganho no tempo de execução proveniente do uso da computação paralela (GPUs) em alguns métodos que foram selecionados para o estudo: filtro de média móvel, filtro morfológico de erosão/dilatação e filtro anisotrópico; e (2) fazer o mesmo também com GPUs de diferentes fabricantes e configurações de *hardware* diferentes;
- **Documentação:** elaboração de toda documentação necessária para o componente, com suas características e requisitos para a sua reutilização.

1.4 Organização do Trabalho

O presente plano de trabalho encontra-se subdividido em oito capítulos. O capítulo 2 aborda aspectos sobre os temas: unidade de processamento gráfico e modelo OPENCL, proporcionando uma fundamentação teórica necessária para a compreensão do projeto. O

capítulo 3 apresenta uma visão geral do software 3D SLICER utilizado na solução da proposta do trabalho. No capítulo 4 são abordados alguns trabalhos encontrados na literatura e relacionados ao tema. O capítulo 5 enfatiza a metodologia empregada no projeto, evidenciando o tipo de pesquisa e procedimentos metodológicos. O capítulo 6 apresenta o componente mediador. O capítulo 7 mostra os experimentos realizados, seus resultados e análises. Por fim, o capítulo 8 apresenta a conclusão do trabalho.

Capítulo 2

UNIDADE DE PROCESSAMENTO GRÁFICO

2.1 Considerações Iniciais

Neste capítulo apresenta-se uma fundamentação teórica sobre a utilização de GPUs para processamento de dados para propósito em geral, dando uma visão básica sobre modelo OPENCL, este sendo muito importante para que as aplicações desenvolvidas possam se comunicar com o dispositivo de *hardware*. Na Seção 2.2 é apresentada uma introdução sobre unidade de processamento gráfico. Na Seção 2.3 são apresentadas as terminologias e conceitos gerais relacionados ao OPENCL, destacando seu modelo de utilização, detalhes técnicos relevantes e *frameworks* para desenvolvimentos. Por fim, na Seção 2.4 são discutidas as considerações finais do capítulo.

2.2 Unidade de Processamento Gráfico

A *Graphics Processing Unit* (GPU) é um processador de propósito especial projetado para operações rápidas de processamento gráfico utilizado em placas de vídeos. Os primeiros esforços para explorar as Unidades de Processamento Gráficos (GPU) em aplicações não gráficas foram inicialmente introduzidos por algumas tecnologias, como o *DirectCompute*³ e

³ *DirectCompute* - <https://developer.nvidia.com/directcompute>

OPENGL⁴. Uma vez que as operações feitas para gráficos são uma forma de aritmética, as GPUs gradualmente foram sendo envolvidas em aplicações para computação de propósito em geral (EKLUND et. al.,2013; SMISTAD et. al.,2015), termo atualmente conhecido como *General Purpose Graphics Processing Unit* (GPGPU). A concepção geral da utilização de uma GPU é motivada pelo seu alto desempenho na execução de grandes quantidades de dados de forma paralela (EIJKHOUT, 2011).

Ao contrário da CPU (*Central Processing Unit*) que possui a capacidade de executar alguns *threads* de uma vez só, a GPU pode executar centenas ou milhares de *threads*. A Figura 1 mostra uma comparação entre a arquitetura de uma CPU com apenas quatro unidades lógica e aritmética (ULAs) e uma GPU com 128 ULAs.

Figura 1. Comparação entre a arquitetura de uma CPU e uma GPU Fonte: NVIDIA CUDA (2013).

Atualmente, com os grandes volumes de placas gráficas implantadas em computadores pessoais ou estações de trabalho devido a sua popularização, as unidades de processamento gráfico (GPUs) tornaram-se uma plataforma de computação paralela acessível a uma vasta gama de usuários. Dadas as fortes características de alta largura de banda de memória e capacidade de computação (FLOPS), há um forte incentivo para usar GPUs para computação de propósito geral (GPGPU) e tem evidenciado muitos relatos de sucesso na literatura sobre esses esforços (KONG et al., 2010).

A computação GPU é útil apenas para cargas de trabalho em que os dados são extremamente passíveis de paralelização, e os cálculos semelhantes são executados em uma grande quantidade de dados que estão dispostos em uma forma de grade regular, por isso é uma solução ideal quando se trata do manuseio de imagens tridimensionais. Os resultados mostram que os algoritmos paralelos podem alcançar notável aceleração, em comparação com os métodos sequenciais executados na CPU (ZHANG, 2010).

⁴ OPENGL - <https://www.opengl.org/>

A NVIDIA⁵ desenvolveu a linguagem CUDA como um veículo para a programação em suas GPUs. A linguagem é basicamente uma extensão de C e tornou-se muito popular. Mais recentemente, a tecnologia OPENCL foi desenvolvida pela APPLE, AMD, INTEL e também NVIDIA. O OPENCL também é uma ligeira extensão de C e tem como objetivo fornecer uma interface uniforme que trabalha com dispositivos multicore, além de GPUs (SCARPINO, 2012).

O OPENCL é o primeiro padrão aberto para multiplataforma, mantido pelo *Khronos Group*⁶ para computação paralela de processadores modernos encontrados em computadores pessoais, servidores e dispositivos portáteis/incorporado. OPENCL (*Open Computing Language*) melhora muito a velocidade e capacidade de resposta para um amplo espectro de aplicações em inúmeras categorias do mercado de jogos e entretenimento, para software científico e médico (OPENCL, 2009).

Rotinas em OPENCL podem ser executadas em GPUs e CPUs de grandes fabricantes como AMD, NVIDIA e INTEL, sem a necessidade de se preocupar com qual empresa projetou o processador ou quantos núcleos que ele contém. A aplicação OPENCL irá compilar e executar nos mais recentes processadores *Fusion* da AMD, nos processadores da INTEL, processadores *Fermi* da NVIDIA e *Cell Broadband Engine* da IBM (SCARPINO, 2012).

Neste contexto, devido à popularização da GPU, suas características motivadoras e o surgimento de tecnologias como visto anteriormente que auxiliam os usuários na criação de aplicações, observa-se um crescimento de iniciativas computacionais que vêm sendo realizadas a fim de resolver problemáticas da computação intensiva de algoritmos de processamento de imagens médicas (EKLUND et. al.,2013; SMISTAD et. al.,2015), com base nas vantagens em operações paralelas fornecidas pela GPU. A seguir iremos abordar um pouco sobre as iniciativas da tecnologia OPENCL, que é o foco principal da proposta do trabalho.

⁵ Home page NVIDIA - <http://www.nvidia.com.br/page/home.html>

⁶ Conformant Companies in OPENCL - <https://www.khronos.org/conformance/adopters/conformant-companies#opencl>

2.3 O Modelo OPENCL

Como dito anteriormente, o OPENCL é o primeiro padrão aberto para computação paralela e seu principal objetivo é permitir o desenvolvimento de aplicações para a prática da computação paralela, permitindo o envio de rotinas a serem executadas em diversos tipos de dispositivos e também de diferentes fabricantes, basta que estes sejam compatíveis com a especificação do OPENCL. O *Central Processing Units* (CPUs), *Graphic Processing Units* (GPUs) e *Field Programmable Gate Array* (FPGA) são apenas alguns dos exemplos mais populares de dispositivos de computação que podem ser explorados através do OPENCL. A seguir serão detalhados alguns aspectos importantes sobre o modelo e arquitetura do OPENCL - grande parte do conteúdo foi retirado de dois livros (SCARPINO, 2012; MUNSHI et al., 2011).

2.3.1 Host, Kernel e Compute Device

Em uma aplicação OPENCL, inicialmente é necessário identificar e entender três entidades principais: (1) *Host*, (2) *Compute device*, (3) *Kernel*.

Como mostrado na Figura 2, o *host* é a aplicação responsável por fazer a ligação entre o ambiente externo (por exemplo, usuário) e com os dispositivos de computação compatíveis ao OPENCL indicados como *Compute Devices*, ou seja, é a parte do programa que cria as estruturas de dados que gerenciam a comunicação *Host-Devices*. Cada dispositivo de computação (por exemplo: GPU, CPU e FPGA) é composto por uma ou várias unidades de computação (por exemplo: *cores* e *multiprocessor*) e por fim essas são compostas por um ou mais elementos de processamentos.

Figura 2. Relação entre *host* e *devices* (INTRODUCTION TO OPENCL, 2010).

Por meio desta ligação é possível enviar rotinas a serem executadas nesses dispositivos de *hardware*. Essas rotinas ou funções são chamadas de *Kernels*, estes são responsáveis por conter todo o trabalho lógico a ser executado no dispositivo de forma paralela. Para que os desenvolvedores construam aplicações OPENCL, terão que lidar com dois tipos de codificações: a aplicação a qual chamados de *host* que será executada na CPU e as funções a serem executadas no dispositivo GPU, estas chamadas de *kernel*. Como os códigos do *host* e do *kernel* são executados em dois espaços de memória diferentes (memória principal da CPU e memória da GPU), será necessário realizar transferências de dados explícitas entre o *host* e as GPUs, exigindo encapsulamento de dados em *buffers* específicos do OPENCL. O programa *Host*, dependendo da linguagem escolhida, poderá ser escrito em: C, C++, Java dentre outras, em contrapartida o *Kernel* é implementado em uma linguagem OPENCL C (baseada no C99). Outras entidades importantes são:

- **Platform:** a estrutura de dados que permite no programa *Host* gerenciar as diferentes implementações (plataformas) de diferentes fabricantes de *hardware* instaladas no computador. O OPENCL permite múltiplas plataformas em um único *Host*;
- **Context:** um contexto identifica um conjunto de dispositivos, mas apenas aqueles selecionados para trabalhar em conjunto (de uma mesma plataforma). Ou seja, você não pode criar um contexto que contém dispositivos, ambos AMD e NVIDIA. Você tem que criar um contexto diferente para cada plataforma. O *Host* pode gerenciar dispositivos usando mais de um contexto e pode até mesmo criar múltiplos contextos de dispositivos em uma única plataforma. Os contextos tornam possível a criação de *Command Queues*, as estruturas que permitem que um *Host* envie *Kernels* para serem executados nos dispositivos;
- **Command Queue:** uma fila é usada pelo *Host* para dizer ao dispositivo a forma com que o *Kernel* deverá ser executado;
- **Program:** a estrutura de dados responsável por guardar um ou mais *Kernels* e essencial para compilar em tempo de execução os códigos dos *Kernels* implementados. Dependendo do tipo de dispositivo escolhido, o compilador da plataforma irá gerar um código a ser executado em uma GPU ou CPU, por exemplo. Veja Figura 3;

Figura 3. Compilação do Kernel em tempo de execução (INTRODUCTION TO OPENCL,2010).

Uma das características importantes do modelo OPENCL é a compilação do *Kernel* em tempo de execução, isto permite ao OPENCL garantir a portabilidade de código entre diferentes tipos de *hardware*, no caso ilustrado acima uma CPU e GPU.

2.3.2 Esquema de Paralelização

O paralelismo dos dados está presente em dois níveis em uma GPU: (1) unidades de computação (núcleos) rodando simultaneamente na GPU; e (2) inúmeros elementos de processamento rodando simultaneamente dentro de um único núcleo.

O esquema de paralelização (mapeamento) de uma tarefa em uma GPU é um processo importante conhecido como NDRANGE (ou *gridification* em CUDA), consiste, por conseguinte, em separar o problema dado sobre estes dois níveis de paralelismo citados acima.

Quando os *Kernels* OPENCL são submetidos para execução em um dispositivo OPENCL, executam dentro do conceito de ciência da computação de um espaço de índice. Um exemplo de um espaço de índice que é fácil de entender é um *loop* comum em C++. No loop definido pela instrução "*for (int i = 0; i < 10; i ++)*", qualquer declaração dentro deste loop será executada dez vezes, com $i = 0, 1, 2 \dots, 9$. Neste caso, o espaço de índice do ciclo é $[0, 1, 2 \dots, 9]$. Em OPENCL, espaços de índice são chamados NDRANGE podem ter 1, 2 ou 3 dimensões.

As funções do *Kernel* são executadas exatamente uma vez para cada ponto no espaço de índice NDRANGE. Esta unidade de trabalho para cada ponto no NDRANGE é chamada de *work-item*. Ao contrário de *loops* em C++, no qual iterações do *loop* são executadas sequencialmente e em ordem, em tempo de execução do OPENCL o dispositivo está livre

para executar *work-items* em paralelo e em qualquer ordem. É esta característica de modelo de execução OPENCL que permite ao programador aproveitar os recursos de computação paralela e permita obter uma considerável aceleração no tempo de execução quando comparado com algoritmo sequencial.

Um único *work-item* será uma tarefa designada a ser executada por um único elemento de processamento, assim, no mesmo instante, um núcleo no dispositivo com N elementos de processamento só pode executar N *work-items* por vez. Outro ponto importante é que os *work-items* não são programados para execução individualmente, em vez disso, os *work-items* são agrupados em *work-groups*, estes sim poderão ser programados e distribuídos da forma que o desenvolvedor desejar dentre os núcleos disponíveis no dispositivo. Tanto a quantidade de núcleos quanto a quantidade de elementos de processamento dentro do mesmo dependerá do tipo de *hardware* disponível na máquina. Veja o esquema apresentado na Figura 4.

Figura 4. Mapeamento da tarefa em um dispositivo (INTRODUCTION TO OPENCL,2010).

Resumindo, quando um usuário envia um *Kernel* para a execução, ele precisa fornecer um NDRANGE, o mesmo é composto por um *offset* e *global_size*, que definem, respectivamente, onde irá começar e terminar o espaço de índice. O usuário também pode definir o tamanho de um *work-group*, utilizando o parâmetro chamado *local_size*.

O usuário pode deixar que o próprio OPENCL defina em tempo de execução com base nas propriedades do *Kernel* e do dispositivo selecionado. Uma vez que o *local_size* tenha sido determinado, o *global_size* é dividido automaticamente em *work-groups*, e estes serão agendados para execução no dispositivo. O OPENCL suporta um NDRANGE unidimensional, bidimensional e tridimensional.

2.3.3 Exemplos de NDRANGES

Como explicado anteriormente, para enviar um *Kernel* a ser executado é necessário definir um espaço de índice, que em OPENCL é chamado de NDRANGE. A seguir serão detalhados alguns exemplos de tarefas a serem mapeadas para a computação paralela segundo o modelo OPENCL.

- a) **NDRANGE unidimensional:** vamos supor um exemplo simples de uma adição de dois vetores de tamanho iguais (N=2048). O código sequencial escrito em C++ para esta operação está na Figura 5, o comprimento do vetor é 2048, e a função faz uma iteração usando um *loop* explícito.

```
void vector_add(int* a, int* b, int* c){
    int i;
    for(i=0; i<2048; i++){
        c[i] = a[i] + b[i];
    }
}
```

Figura 5. Código sequencial em C++, para soma de dois vetores. (Fonte: elaborada pelo autor)

Em OPENCL, o *Kernel* para realizar a mesma tarefa é mostrado na Figura 6. Observe que a chamada “*get_global_id(0)*” fornece a localização atual no NDRANGE e é análoga ao índice de um *loop* explícito.

```
__kernel void vector_add(__global int* a, __global int* b, __global int* c){
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Figura 6. Código do *Kernel* baseado em C99, para a soma de dois vetores. (Fonte: elaborada pelo autor)

Como dito antes, para ser enviada a tarefa que será executada de forma paralela no dispositivo, deve-se definir um NDRANGE (por exemplo: *offset=0*, *global_size=2048* e *local_size=256*). Com a definição do NDRANGE, isto permite que a tarefa de adição dos vetores seja dividida em oito *work-groups*, cada uma com 256 de *work-items*. Veja a seguir na Figura 7 a ilustração do NDRANGE específico para esse exemplo.

Figura 7. Exemplo de espaço de índice (NDRANGE) unidimensional. (Fonte: elaborada pelo autor)

- b) **NDRANGE bidimensional:** O exemplo anterior pode ser estendido a um NDRANGE bidimensional. Este tipo de NDRANGE funciona bem com dados bidimensionais, tais como matrizes. Veja na Figura 8 um trecho de código sequencial que faz multiplicação de duas matrizes de tamanhos iguais.

```
for(int i=0; i<size; ++i)
for(int j=0; j<size; ++j)
for(int k=0; k<size; ++k)
{
    result[i][j] += matrixA[i][k] * matrixB[k][j];
}
```

Figura 8. Código sequencial em C++, para multiplicação de matrizes. (Fonte: elaborada pelo autor)

No OPENCL, a linguagem do *Kernel* baseada em C99 não permite trabalhar com estrutura de dados do tipo matrizes. Para isso deve-se trabalhar com vetores unidimensionais. A seguir o *Kernel* para realizar a mesma tarefa é representado na Figura 9.

```

__kernel void matvec_mult( __global float* matrixA, __global float* matrixB,
__global float* result, int size) {

    int i = (int)get_global_id(0);
    int j = (int)get_global_id(1);

    for(int k = 0; k<size; k++){

        result[i*size+j] += matrixA[i*size +k] * matrixB[k*size+j];

    }

}

```

Figura 9. Código do *Kernel*, para multiplicação de matrizes. (Fonte: elaborada pelo autor)

Do mesmo modo que o exemplo anterior, um NDRANGE é definido só que agora em duas dimensões. Suponhamos a operação de uma multiplicação de duas matrizes de tamanho 32x32. Uma opção de NDRANGE seria: o *offset* = (0,0), o *global_size* = (32,32) e o *local_size* = (2,2). A tarefa será executada em 256 *work-groups* contendo cada um 4 *work-items* (2x2), de um total de 1024 *work-items* (32x32). Veja a ilustração do NDRANGE específico para esse exemplo na Figura 10.

Figura 10. Exemplo de espaço de índice (NDRANGE) bidimensional. (Fonte: elaborada pelo autor)

- c) **NDRANGE tridimensional:** O presente projeto irá trabalhar com NDRANGE tridimensionais pelo fato de trabalhar com imagens médicas 3D, qualquer operação de processamento nessas imagens é necessário um espaço de índice também tridimensional. Vejamos um exemplo de uma imagem 3D fictícia de tamanho 16x16x16 na Figura 11.

Figura 11. Exemplo de espaço de índice (NDRange) tridimensional. (Fonte: elaborada pelo autor)

2.3.4 Gerenciamento de Memória

A memória é outro aspecto importante da computação GPU que merece descrição detalhada. Existem quatro tipos de memória da GPU em OPENCL: (1) Memória global, que é acessível a todos os *work-items* GPU. O OPENCL define um objeto de memória apontando para a memória global. Um objeto de memória pode ser ou um *buffer* para armazenar elementos unidimensionais ou um objeto de imagem para armazenar imagens 2D ou 3D. (2) A memória da constante, que é uma região de memória global, serve apenas de leitura e mantém-se constante durante a execução de uma função de *Kernel*. (3) De memória local, que é uma região de memória associada a um *work-group* e só é acessível aos *work-items* contidos no mesmo. (4) A memória privada, que é privada para um *work-item* e não visível para quaisquer outros.

Por esta razão, na aplicação *host* ao implementar a transferência de dados *Host-Device* é importante saber como configurar os argumentos dos *Kernels*, levando em consideração os quatro tipos de qualificadores de espaço de endereço citados acima, ou seja, todo o gerenciamento de memória no dispositivo é feito de modo explícito pelo desenvolvedor. A Figura 12 resume tudo o que foi explicado.

Figura 12. Modelo de gerenciamento de memória do OPENCL (INTRODUCTION TO OPENCL,2010).

2.3.5 O Framework OPENCL

O *Khronos Group*, que mantém o OPENCL, não fornece nenhum *framework* para uso, em vez disso, cada fabricante que é responsável em aderir ao OPENCL e implementar sua própria plataforma compatível seguindo as especificações do OPENCL. Tais implementações são disponibilizadas para os desenvolvedores por meio de SDKs (*Software Development Kits*), exemplos:

- Intel GPUs: Intel SDK for OPENCL;
- AMD/ATI GPUS: AMD APP SDK;
- NVIDIA GPUS: CUDA Toolkit;
- Altera FPGAs: Altera SDK for OPENCL;

Alguns pontos importantes à serem seguidos para a instalação:

- Saber qual o fabricante e modelo do *hardware* que se pretende programar, e se ele suporta OPENCL;

- Procurar o SDK respectivo ao sistema operacional desejado e realizar o processo de instalação;
- Ter uma compreensão básica dos arquivos presentes no SDK e saber quais arquivos necessários a serem importados no projeto.

Com isso, ao construir uma aplicação OPENCL, o mesmo irá garantir a portabilidade do código, para que outros usuários usem a aplicação e garanta a execução, independentemente do sistema operacional e do *hardware* (basta que este seja OPENCL compatível).

2.4 Considerações Finais

Neste capítulo foram discutidos dois temas importantes e diretamente relacionados com a prática da computação paralela em GPUS, apresentando uma fundamentação teórica sobre a unidade de processamento gráfico, este sendo o dispositivo de *hardware* de interesse para processar dados de forma paralela. Outro tema foi sobre a tecnologia OPENCL, que é um *software* responsável por criar a aplicação (*host*) e pela comunicação com o *hardware* (GPU), assim, possibilitando o envio de rotinas a serem executadas de forma paralela na GPU.

Recentemente, o OPENCL foi introduzido no campo de computação paralela de alto desempenho. Uma ferramenta poderosa, que simplifica e fornece uma estrutura de dados para que os desenvolvedores escrevam programas que serão executados em diferentes plataformas, incluindo CPUs convencionais, GPUs de diferentes fabricantes e outros processadores. O desenvolvimento de aplicações que funcionem em diferentes plataformas potencialmente fornece uma solução para o problema de portabilidade e elimina a preocupação do desenvolvedor em desperdiçar esforços na aprendizagem de linguagens e desenvolvimento específico para diferentes plataformas de GPU.

O capítulo apresentado serviu para mostrar a tecnologia OPENCL e evidenciar que embora a programação GPU possa ser significativamente simplificada com essas novas tecnologias que estão surgindo, tanto o OPENCL como CUDA, ainda sim é exigida por parte do desenvolvedor uma profunda compreensão de algoritmos (o algoritmo da aplicação, o algoritmo desejado de paralelização, *Kernels*) e também conceitos básicos de *hardware* (arquiteturas de GPU e CPU), tornando às vezes indescritível para muitos usuários finais. A

proposta do componente é proporcionar a prática da computação paralela minimizando a complexidade para a criação de aplicações pelo desenvolvedor.

O próximo capítulo apresenta uma visão geral do *software* 3D SLICER, evidenciando aspectos de suas funcionalidades, características e dos modelos de desenvolvimento de módulos. Estes são importantes para a incorporação do componente mediador dentro do 3D SLICER, que é exigido quando se trata de processamento de imagens tridimensionais, parte da proposta deste trabalho.

Capítulo 3

SOFTWARE 3D SLICER

3.1 Considerações Iniciais

Este capítulo apresenta uma visão e conceitos gerais sobre o 3D SLICER, *software* envolvido na proposta do projeto. Um *software* robusto para análise e visualização de imagens médicas tridimensionais. Inicialmente na Seção 3.2 é apresentada uma introdução sobre o *software*. Na Seção 3.3 é apresentada uma visão geral do *software*, abordando suas principais características e funcionalidades. A Seção 3.4 descreve especificamente sobre a forma de integração e incorporação de novas funcionalidades dentro do *software*. Por fim, na Seção 3.5 são apresentadas as considerações finais deste capítulo.

3.2 Introdução

Junto com a forte evolução tecnológica e computacional, o uso do computador para o processamento e análise de imagens médicas está se desenvolvendo rapidamente. A área de processamento de imagens médicas fornece cada vez mais informações quantitativas claras, precisas e úteis para diagnóstico e tratamento de certas doenças (por exemplo: câncer, doenças cardiovasculares e distúrbios neurológicos). Hoje em dia, inúmeras plataformas de processamento de imagens médicas têm surgido, a maioria não é de código aberto e não é facilmente extensível e redistribuído em comparação com 3D SLICER (FANG,2013).

Aplicações para computação em imagens médicas são peças complexas de *software* que requerem um conjunto comum de funcionalidades base, bem como a capacidade de

personalização para aplicações clínicas específicas como por exemplo de diferentes especialidades. O ambiente de pesquisa é muitas vezes necessário para criar protótipos que permitem a exploração e refinamento de novos *softwares* que ao longo do tempo possam tornar-se completos para a sua utilização por usuários finais. As pesquisas sobre o 3D SLICER não só promovem a disseminação da tecnologia de processamento de imagem médica, mas também ajudam a projetar e melhorar a própria plataforma (Pieper et al. 2004).

O *software* 3D SLICER, ou simplesmente *Slicer*, é um pacote gratuito e *open source*, desenvolvido para a visualização e análise de imagens de qualquer natureza (Pieper et al. 2004; Pieper et al 2006; Fedorov et al. 2012a). É originalmente concebido para estar disponível em múltiplas plataformas, incluindo *Windows*, *Linux* e *Mac OS X*. As últimas versões possuem capacidades robustas de manipulação de imagens DICOM, segmentação interativa, renderização volumétrica, coregistro rígido e não rígido de volumes, funcionalidade de captura de tela *sceneSnapshot*, visualização de imagens 4D, interface gráfica com layouts e visualizadores de fatias flexíveis e gerenciador de extensão para a exploração e instalação de plug-ins. O 3D SLICER é uma plataforma criada com a finalidade de análise e visualização de imagens para finalidades específicas:

- Suporta imagens de multimodalidade, incluindo, IRM, CT, ultrassonografia, medicina nuclear e microscopia;
- Suporte a vários órgãos da cabeça aos pés com funcionalidades específicas;
- Interface bidirecional para dispositivos diagnósticos e terapêuticos que podem ser guiados por imagens é um importante característica do *software*, uma vez que permite uma fácil interface com variados tipos de *hardware* de aquisição de imagens e de equipamentos responsáveis por tratamentos guiados por imagens;
- Fornece interfaces capazes de se comunicarem com outros *softwares* de pesquisa biomédica e também possui uma arquitetura *plug-in-play* que permite aos desenvolvedores a criação e integração de módulos específicos à plataforma.

O 3D SLICER foi iniciado como um projeto no Laboratório de Planejamento Cirúrgico no Hospital *Brigham and Women* e do Laboratório de Inteligência Artificial do MIT em 1998. Uma variedade de publicações foi permitida pelo software 3D SLICER (Pieper et al. 2004; Pieper, et al. 2006; Egger et al. 2013; Kikinis & Pieper 2011; Fedorov et al. 2012a; Egger et al. 2012). Atualmente o *software* faz parte do trabalho desenvolvido pela *National*

Alliance for Medical Image Computing (NA-MIC), financiada pelos *National Institute of Health*⁷.

Com protocolos clínicos devidamente criados e geridos, o 3D SLICER tem sido utilizado em pesquisa clínica. Na investigação sobre a terapia guiada por imagem, o 3D SLICER é frequentemente usado para construir e visualizar conjuntos de dados de IRM que estão disponíveis pré e intraoperatório para permitir a aquisição de coordenadas espaciais para rastreamento do instrumento. Ele fornece uma interface gráfica para interagir com os dados. Além da segmentação manual e a criação de modelos de superfície 3D a partir de imagens de ressonância magnética convencional, o 3D SLICER tem sido utilizado para o coregistro e para incorporar modelos da ramificação vascular cerebral baseada em IRM permitindo a condução guiada de cateter para intervenções⁸. Também já desempenhou um papel crucial em várias modalidades de terapias guiada por imagem, e a comunidade continua crescendo e desenvolvendo novos métodos a serem incorporados dentro do 3D SLICER.

3.3 Visão Geral do 3D SLICER

Nesta seção será apresentada uma visão geral do *software* 3D SLICER, evidenciando suas principais características e funcionalidades, a Figura 13 ilustra a tela principal do *software*, ela disponibiliza acesso aos recursos mais utilizados, e organiza-os em agrupamentos lógicos. Estes grupos de recursos ou funcionalidades são apresentados em certos painéis ou menus. A interface é projetada para ser fácil de aprender e lembrar, facilitando a navegação em uma grande quantidade de funcionalidades disponíveis e esconder quando você não tem interesse de vê-las. Uma breve descrição de alguns recursos úteis ilustrados na Figura 13:

- **Load & Save:** carrega imagens de diversos formatos ou cenas, baixa conjunto de dados via *internet*. Uma opção de salvar imagens e cenas também é fornecido neste *menu*;
- **Module Selection & Navigation:** fornece opções para a busca de nomes de módulos já implementados;

⁷ NAMIC - <http://www.na-mic.org/>

⁸ Slicer Wiki - <http://wiki.slicer.org/slicerWiki/index.php/Documentation/4.5/Training>

- **Menu File:** carrega/Salva imagens de diversos formatos ou cenas, baixa conjunto de dados via *internet*;
- **3D Viewer:** janela que mostra os volumes e modelos, em uma superfície 3D.
- **Slices Viewers:** visualizadores que mostram as fatias 2D de cada plano ortogonal.
- **Modulo Panel:** qualquer módulo selecionado no **Slicer** irá exibir sua interface de usuário no *Painel Module* (localizado canto esquerdo);
- **Interactive Data Probe Panel:** traz informações dos valores de dados em *voxels* indexados pela posição atual do mouse em todas as camadas.

Figura 13. Tela do 3D SLICER⁹.

Os 3 planos ortogonais anatômicos (Figura 14), o Axial, Frontal e Sagital são planos perpendiculares entre si e exibidos em cada *Slice Viewer* apresentado anteriormente:

- **Plano Sagital:** é um plano paralelo ao plano mediano, vertical, que atravessa o eixo mais longo do corpo, dos pés à cabeça, separando o corpo em lado esquerdo e direito;
- **Plano Coronal:** também é um plano vertical, que passa pelo eixo maior (dos pés à cabeça), porém perpendicular ao plano sagital, separando o corpo em lado da frente e de trás;

⁹ Slicer Wiki - <https://www.slicer.org/wiki/Documentation/4.1/SlicerApplication/MainApplicationGUI>

- **Plano Axial:** conhecido também por plano horizontal ou transversal, (transverso) atravessa o eixo menor do corpo, isto é, da posição posterior para a anterior, separando o corpo em porções superior ou inferior.

Figura 14. Planos Ortogonais Anatômicos¹⁰.

O desenvolvimento de aplicações com interfaces gráficas de usuário (GUI) leva tempo e pode ser um trabalho duro, ainda mais para visualização de imagens médicas 2D e 3D. Tornar que essas aplicações funcionem em diferentes sistemas operacionais pode ser ainda mais complexo. Atualmente, muitas aplicações são desenvolvidas para uma única plataforma, depois para levar essa aplicação a ser executada em outra plataforma é necessário a reescrita

¹⁰ Plano Transversal - https://pt.wikipedia.org/wiki/Plano_Transversal

de uma grande quantidade de linhas de códigos. As versões atuais do 3D SLICER as interfaces gráficas já são desenvolvidas utilizando o QT¹¹.

O kit de ferramentas GUI QT permite uma fácil portabilidade de aplicações entre plataformas. QT é um *framework* multiplataforma para desenvolvimento de interfaces gráficas em C++ criado pela empresa norueguesa *Trolltech*. Com ele é possível desenvolver aplicativos e bibliotecas uma única vez e compilá-los para diversas plataformas sem que seja necessário alterar o código fonte. Alguns exemplos de plataformas populares suportadas: *Linux*, *OS X*, *Windows*, *Android*, *iOS*, *BlackBerry*, dentre outras.

A plataforma 3D SLICER fornece uma poderosa coleção de bibliotecas, aplicações e ferramentas para o processamento de imagens médicas, visualização e análise, que também são de código abertos, por exemplo: *Visualization Toolkit* (VTK), o *Insight Toolkit* (ITK) e as bibliotecas OPENCV.

3.4 Desenvolvimento de Módulos

O 3D SLICER suporta o desenvolvimento de módulos integrados, possui três tipos de módulos: *Command Line Interface* (CLI), *Loadable Modules* e o *Scripted Modules*. O desenvolvedor deve escolher um dos três tipos para implementação do seu módulo, independente da escolha o usuário final não vai notar a diferença. A escolha de um determinado tipo de módulo é geralmente baseada no tipo de entradas/parâmetros para o mesmo. Seguem os tipos:

- ***Command Line Interface (CLI)***: são módulos executáveis com um número limitado de parâmetros de entrada e saída. Nesse tipo de modulo, o *front-end* (GUI) é automaticamente gerado por meio de um arquivo XML, e na construção desse XML há como opção uma série de elementos gráficos prédefinidos pela plataforma. O *front-end* é quem fará a interação com o usuário. É por meio desses elementos que serão enviados os parâmetros de entrada para o *back-end* implementado em C++ (por exemplo: imagem de entrada, imagem de saída, e um parâmetro necessário para processamento). Recebendo os parâmetros no *back-end*, normalmente apenas será necessário utilizar as classes do ITK para o

¹¹ Qt | Cross-platform software development for embedded & desktop - <https://www.qt.io/>

processamento desejado juntamente com os parâmetros de entrada. A imagem de saída por padrão será renderizada no visualizador do 3D SLICER;

- **Loadable Modules:** são módulos também em C++, porem diferentemente do anterior CLI, o desenvolvedor possui total liberdade para modificar o *front-end* (GUI) e também um controle maior sobre a mesma para sua aplicação, fazendo o uso do QT. A escolha de qual tipo de modulo CLI ou *Loadable* dependerá do tipo de necessidade exigida pela aplicação;
- **Scripted Modules:** são módulos escritos em Python, recomendados para uma rápida prototipação e um desenvolvimento de fluxo de trabalho personalizado.

3.5 Considerações Finais

Atualmente o uso de imagens médicas são muito comuns para diagnóstico e investigação científica. Muitas imagens são convencionais, isto é bidimensional, mas imagens obtidas por ressonância magnética (IRM), *Positron Emission Tomography* (PET) e tomografia computadorizada (CT) são tridimensionais e muitas vezes ocupam dezenas de *megabytes*. As imagens tridimensionais são normalmente armazenadas em duas maneiras diferentes: como pilhas de imagens convencionais, isto é, um conjunto de imagens bidimensionais com nomes de arquivo sequencialmente numerados; ou como arquivos que suportam representação da imagem tridimensional, como o formato de arquivo NRRD e DICOM (TORIWAKI, 2009).

Cada uma dessas modalidades apresenta características e aplicabilidades diferentes, desempenhando papel decisivo em procedimentos cruciais de diagnóstico e terapia. Assim, a demanda por métodos computacionais complexos que extraem e processam informações intrínsecas e funcionais destes tipos de imagens tridimensionais tem crescido e são relevantes.

De modo geral, o 3D SLICER tem como o objetivo fornecer uma plataforma para a prática clínica em imagens médicas 3D, possibilitando a construção de módulos e fornecendo uma interface que facilite a visualização e interação com usuário. É também uma alternativa a diversos *softwares* caros existentes no mercado, utilizados para mesmos fins.

Entretanto, por se tratar de processamento de imagens tridimensionais (grande quantidade de dados), e métodos que implementam cálculos altamente complexos, isto pode demandar um alto custo computacional (de tempo), sendo executado de forma sequencial em

uma CPU podem levar horas para apresentar o resultado, inviabilizando devido ao tempo de espera. Com isso, tais aplicações exigem algoritmos mais eficientes, e uma possível solução é a adoção da computação paralela em GPU.

Pensando no contexto de processamento de imagens tridimensionais, a incorporação do componente mediador dentro do 3D SLICER por meio da criação de um módulo é uma parte da solução muito importante, pois além de proporcionar a prática da computação paralela, permitirá uma alta abstração da complexidade, em relação ao processamento, gerenciamento e renderização dessas imagens, visto que será possível acessar e reutilizar todas as funcionalidades existentes no 3D SLICER. Seria inviável para a pesquisa caso fosse necessário criar ou implementar do zero todas as funcionalidades e estrutura de dados necessário para uma adequada manipulação dessas imagens tridimensionais.

O próximo capítulo apresenta um conjunto de trabalhos relacionados, fornecendo informações pertinentes para o atual projeto.

Capítulo 4

TRABALHOS RELACIONADOS

4.1 Considerações Iniciais

Neste capítulo serão apresentados alguns trabalhos encontrados na literatura relacionados à aplicação de métodos ou técnicas de processamento de imagens médicas, utilizando abordagem de computação paralela em GPUs. É importante salientar que não foram encontrados na literatura trabalhos diretamente relacionados ao *software* 3D SLICER no contexto da computação paralela. A seguir será apresentada uma revisão dos trabalhos, com pontos importantes e relevantes ao presente projeto.

4.2 Processamento de imagens utilizando o OPENCL

Na literatura, são comumente encontrados trabalhos que descrevem aplicações em OPENCL no contexto da área de processamento de imagens. Métodos de processamento de imagens normalmente executam operações semelhantes em uma grande quantidade de dados, sendo extremamente passíveis de paralelização. Com isso é possível usufruir da computação paralela de uma GPU para superar as implementações sequenciais executadas na CPU.

O trabalho (BERNABÉ et. al. 2012) é relacionado ao método da transformada rápida de *Wavelet* 3D, esta técnica é particularmente importante em processamento e análise de imagens, porque no domínio transformado algumas características relevantes da imagem ficam mais evidentes. Os autores apresentaram implementações da transformada rápida de

Wavelet 3D em CUDA e OPENCL, tendo como resultados ganhos relevantes na execução da tarefa comparado ao método sequencial na CPU.

Uma versão paralela da técnica de afinamento tridimensional é proposta no trabalho (JIMÉNEZ et. al. 2012). Esta é uma das técnicas para o cálculo da curva de esqueleto 3D de um volume, tendo uma grande variedade de usos e aplicações como: navegação virtual, endoscopia virtual, animação, segmentação de estruturas anatômicas, registo de imagem médica, correspondência de objetos 3D, reconstrução da superfície, e malha de reparo, entre outros. Um problema importante é que o cálculo do esqueleto é um processo muito demorado.

Outros trabalhos utilizaram bibliotecas já existentes com suporte ao OPENCL e bem robustas no contexto de processamento de imagens, como o ITK. Descrevendo novas implementações de métodos usando a computação paralela proporcionada pelo OPENCL. O autor do trabalho (BARDOSI, 2015) forneceu implementações GPU para duas operações morfológicas binárias: erosão e dilatação, aplicáveis a imagens 1D, 2D e 3D. Semelhante ao anterior, os autores (VIDAL-MIGALLÓN, 2013) propõem uma implementação GPU para o filtro de borramento gaussiano recursivo.

Já no trabalho (OLIVEIRA et. al., 2015) os autores propõem uma nova biblioteca para o uso da tecnologia OPENCL, suportando trabalhar com imagens 2D e 3D. O trabalho também realiza testes e comparação de performance em várias outras bibliotecas existentes que suportam o OPENCL. Para a análise do desempenho utilizam-se algumas operações básicas de processamento de imagens como: convolução, erosão e borramento.

4.3 Componentes e Frameworks Baseados em OPENCL

O estudo realizado (CHEN, 2018) propõe uma arquitetura nomeada de FLINKCL, uma arquitetura de computação de memória em clusters de CPU-GPU heterogêneos baseados no OPENCL, que permite ao *Apache Flink*¹² utilizar a capacidade massiva de processamento paralelo da GPU. Semelhante a proposta do presente trabalho, os autores do trabalho validaram com um conjunto de cargas de trabalho representativas para mostrar sua eficácia, e tendo resultados de ganhos do tempo de execução de até 11x para alguns algoritmos computacionalmente pesados e mantendo-se pequenas melhorias de desempenho para outros

¹² *Apache Flink*® - *Stateful Computations over Data Streams* - <https://flink.apache.org/>

algoritmos. Em outro estudo (YUAN, 2016) os autores propõem a utilização de uma tecnologia chamada de *Apache Spark*¹³.

Na literatura, alguns estudos estão utilizando a computação paralela em métodos de aprendizado de máquina. No trabalho proposto por (WANG, 2017), os autores exploram a computação paralela do OPENCL em dispositivos FPGA. No contexto de redes neurais convolucionais, é proposto um componente chamado de PIPECNN, um acelerador FPGA eficiente que pode ser implementado em uma variedade de plataformas FPGA (*Field Programmable Gate Array*) com desempenho e custo reconfiguráveis. O projeto PIPECNN é abertamente acessível e, portanto, pode ser usado tanto por pesquisadores como uma estrutura genérica para explorar novas arquiteturas de hardware ou por professores como um exemplo de design para qualquer curso acadêmico relacionado a FPGAs.

O trabalho (LIAO, 2018) utiliza o OPENCL aplicado em métodos de redes neurais profundas, por meio da computação paralela visando o ganho no tempo de execução do processo de treinamento dessas redes neurais. Também é proposto um framework baseado em OPENCL chamado *UHCL-Darknet*.

Para trabalhos relacionados ao processamento de imagens multidimensionais, foi encontrado o (Simmross-Wattenberg, 2018), este descreve um *framework*, chamado OPENCLIPER e escrito em C++, baseado no OPENCL que é capaz de manipular dispositivos de computação dedicados de forma integrada e que permite ao desenvolvedor se concentrar em tarefas de processamento de imagem, bem similar à proposta atual do trabalho. A estrutura lida automaticamente com a descoberta e inicialização de dispositivos, transferências de dados de e para o dispositivo e o sistema de arquivos e carregamento e compilação do *kernel*. Estruturas de dados precisam ser definidas apenas uma vez, independentemente do dispositivo de computação; o código é único, consequentemente, para cada dispositivo, incluindo a CPU do *host*. O mapeamento de memória / buffer fixo é usado para obter o máximo desempenho nas transferências de dados. Os fragmentos de código incluídos no artigo mostram como o dispositivo de computação é quase imediatamente e sem esforço disponível para os algoritmos dos usuários, para que eles possam se concentrar no trabalho produtivo. O código necessário para a seleção e inicialização do dispositivo, carregamento de dados e *streaming* e compilação do *kernel* é mínimo e sistemático. O trabalho é validado em um método de reconstrução de imagem de ressonância magnética.

¹³ *Apache Spark*™ - Unified Analytics Engine for Big Data - <https://spark.apache.org/>

4.4 Trabalhos Baseados em CUDA

Relacionado ao processamento de imagens multidimensionais utilizando CUDA, temos o trabalho (UECKER, 2015; Tamir, 2016), ambos os artigos se referem à criação do *Berkeley Advanced Reconstruction Toolbox*¹⁴ (BART), é uma biblioteca de programação, bem como uma coleção de ferramentas de linha de comando para realizar a reconstrução de imagens de ressonância magnética. No artigo, os autores referem-se a essa caixa de ferramentas como um *framework* para a reconstrução de imagem iterativa, de modo que os termos de biblioteca e *framework* são usados de forma intercambiável, embora haja diferenças entre ambos. Essa caixa de ferramentas tem uma orientação dupla, ou seja, serve para prototipagem e testes rápidos de algoritmos e também facilita a integração desses algoritmos no pipeline de aquisição e reconstrução de dados. A biblioteca também fornece suporte para computação paralela usando várias CPUs ou para alguns dos algoritmos disponíveis em GPUs. A tecnologia CUDA é, no entanto, a única API suportada, de modo que o processamento dedicado é limitado a dispositivos do tipo GPU da NVIDIA.

Similarmente ao anterior temos o trabalho (HANSEN, 2013; XUE, 2015). Os autores propõem um *framework* de código aberto para reconstrução de imagens médicas chamado *Gadgetron*¹⁵. A estrutura implementa um sistema flexível para a criação de pipelines de processamento de dados em fluxo contínuo em que os dados passam por uma série de módulos ou *gadgets* de dados brutos para imagens reconstruídas. O pipeline de processamento de dados é configurado dinamicamente em tempo de execução com base em uma descrição de configuração em arquivos XML. O *framework* promove a reutilização e o compartilhamento de módulos de reconstrução e novos *gadgets* podem ser adicionados à estrutura do *Gadgetron* por meio de uma arquitetura semelhante a um plugin, sem recompilar todo o *framework*. Geralmente, os *gadgets* são implementados em C / C ++, porém é permitido ao usuário implementar novos módulos na linguagem de script *Python* para prototipagem rápida. Ao fim do trabalho os autores discutem que o *framework* é limitado às GPUs da NVIDIA e como trabalhos futuros desejam estender dando suporte também ao OPENCL, tornando o *framework* independente de fornecedores de *hardwares* de GPU.

¹⁴ BART Toolbox - <https://mrirecon.github.io/bart/>

¹⁵ Gadgetron - <http://gadgetron.github.io/>

4.5 Comparação CUDA e OPENCL

Acredita-se que ambas as tecnologias, tanto CUDA como OPENCL, irão desempenhar um papel importante para a proliferação da prática GPGPU. No contexto de computação paralela em GPUs, existem diferenças significativas que muitas vezes atrapalham e podem confundir os desenvolvedores. Ambas as tecnologias possuem muitas semelhanças conceituais, mas eles divergem em algumas terminologias (KARIMI, 2010; FANG, 2011; DU, 2012; PAULA, 2014). A Tabela 1 mostra alguns exemplos de termos equivalentes.

Tabela 1. Tabela de equivalência entre as tecnologias OPENCL e CUDA.

OPENCL	CUDA
<i>Host</i>	<i>Host CPU</i>
<i>Compute unit</i>	<i>Streaming multiprocessor</i>
<i>Processing element</i>	<i>Scalar score</i>
<i>Work-item</i>	<i>Thread</i>
<i>Work-group</i>	<i>Thread-block</i>
<i>NDRange</i>	<i>Grid</i>
<i>Global memory</i>	<i>Global or device memory</i>
<i>Local memory</i>	<i>Shared (per-block) memory</i>
<i>Private memory</i>	<i>Registers</i>

Os autores do trabalho (WEBER et. al., 2010) compararam implementações em CUDA e OPENCL de uma aplicação do método de Monte Carlo em diversos tipos de *hardwares*. O trabalho mostrou que o OPENCL garante portabilidade de código, porém não proporciona necessariamente portabilidade de desempenho. Além disso, os autores mostraram que as linguagens específicas da plataforma CUDA muitas vezes, mas não sempre, superaram a do OPENCL.

Outros trabalhos também identificam a linguagem de programação CUDA claramente muito mais fácil de programar a GPU para cálculos de uso geral e obtendo uma vantagem de desempenho significativa sobre o OPENCL, entretanto tendo como uma principal desvantagem do CUDA o suporte apenas em GPUs da NVIDIA (KONG et. al., 2010; KARIMI et. al., 2010). Mas segundo (FANG et. al., 2011) isso está diminuindo com o aperfeiçoamento dos compiladores OPENCL e atualmente situam-se em não mais do que 20-30% menos desempenho comparado ao CUDA.

4.6 Considerações Finais

O capítulo apresentado foi muito importante e serviu como um guia inicial de aprendizado, obtendo uma noção do que já existia na literatura pertinente aos temas: computação paralela em GPUs e processamento de imagens tridimensionais.

Na literatura encontram-se vários trabalhos relacionados, utilizando tanto a tecnologia OPENCL como o CUDA, propondo a construção de um componente para algum propósito específico, mesmo que diferente do contexto de processamento de imagens tridimensionais. Os trabalhos que apresentavam uma comparação entre o CUDA e OPENCL foram de extrema importância para análise da solução do componente e também para a fase do projeto em que foram definidas as tecnologias a serem utilizadas na construção do componente mediador. O próximo capítulo apresenta a metodologia que foi seguida no trabalho.

Capítulo 5

MATERIAL E MÉTODOS

5.1 Considerações Iniciais

Este capítulo descreve e classifica o tipo de pesquisa que foi realizado, além de apresentar o detalhamento de seu processo metodológico, que vai desde a concepção e revisão bibliográfica, até procedimentos relacionados à execução e validação referentes a utilização do componente.

5.2 Tipo de Pesquisa

Na literatura, seguindo as fundamentações de WAZLAWICK (2014) e GERHARDT & SILVEIRA (2009), uma pesquisa científica pode ser classificada em quatro dimensões: (1) quanto à sua abordagem; (2) quanto a sua natureza; (3) quanto aos objetivos e (4) quanto aos procedimentos. Em relação à abordagem pode ser qualitativa ou quantitativa. Quanto à natureza, o trabalho pode ser classificado como uma pesquisa básica (pura) ou aplicada. Quanto a seus objetivos, a pesquisa pode ser exploratória, descritiva ou explicativa. Em relação aos procedimentos, a pesquisa pode ser experimental, bibliográfica, documental, um estudo de caso dentre outros.

Após as fundamentações, a pesquisa então foi classificada como quantitativa, não havendo nenhuma análise/estudo de caráter subjetivo na implementação da proposta do trabalho, e considerada de natureza aplicada, pois o estudo visa combinar conhecimentos,

tanto de *software* como de *hardware* para a aplicação prática, a qual pretende-se propor um componente mediador para a computação GPU de imagens médicas no *software* 3D SLICER.

Quanto aos objetivos de caráter exploratório, visa à exploração de novas tecnologias e análises de soluções em tecnologias que são necessárias para a implementação de aplicações que praticam a computação paralela em GPUs. O trabalho utilizará procedimentos experimentais, e o componente será implementado em um ambiente artificial previamente configurado. Isto permite estudar a arquitetura proposta, definir seus limites e melhorias, facilitando sua validação.

Todo o trabalho foi fundamentado em revisões bibliográficas e pesquisas documentais, visando à análise de livros de diversos autores, periódicos, revistas e demais publicações em relação ao tema proposto.

5.3 Procedimentos Metodológicos

As atividades que foram executadas no presente trabalho, de modo geral, estão definidas em três fases descritas a seguir.

5.3.1 Concepção e Preparação da Pesquisa

A primeira fase diz respeito à concepção e aspectos iniciais para a realização da pesquisa em questão. A Figura 15 exemplifica cada uma de suas etapas.

Figura 15. Etapas da primeira fase. (Fonte: elaborada pelo autor)

- **Definição do escopo do problema:** uma das fases iniciais, e muito importante, de um trabalho científico é a definição da problemática em questão, a ser estudada. A definição surgiu a partir da observação de uma alta quantidade e processamento de dados demandados por inúmeros métodos computacionais na área de processamento de imagens médicas 3D. Tais métodos podem manusear imagens de tamanho grande das mais variadas modalidades. E neste contexto, em relação à computação paralela GPU, apresenta-se uma ausência de ferramentas ou componentes que proporcionem o uso desta prática dentro do *software* 3D SLICER, tendo então como benefícios: aplicações mais rápidas devido ao uso da computação paralela;
- **Revisão bibliográfica sobre o tema GPU:** esta etapa se refere à coleta e estudo de materiais bibliográficos, revisões literárias sobre os seguintes temas: computação paralela, computação heterogênea, *hardware* GPU e CPU, OPENCL, CUDA, GPGPU;
- **Revisão bibliográfica sobre o tema 3D SLICER:** esta etapa se conduz à coleta e estudos de materiais bibliográficos e revisões literárias sobre os seguintes temas: 3D SLICER, algoritmos e métodos de processamento de imagens médicas tridimensionais, ITK, VTK;
- **Revisão de trabalhos relacionados:** esta etapa refere-se a revisões de trabalhos atuais que estejam mais relacionados com a proposta do projeto. Essas revisões têm como objetivo ajudar no estudo e desenvolvimento e também evidenciar tanto os principais resultados já obtidos por outros pesquisadores como os problemas que ainda persistem.

5.3.2 Análise da Solução e Desenvolvimento

Nesta fase diz respeito à definição, análise da solução e estudo das tecnologias relacionadas, como também desenvolvimento e documentação do componente. A Figura 16 ilustra cada uma de suas etapas.

Figura 16. Etapas da segunda fase. (Fonte: elaborada pelo autor)

- **Definição e estudo das tecnologias para o componente:** esta etapa teve como objetivo uma cuidadosa definição das tecnologias a serem trabalhadas, assim como uma análise da solução adequada para construção do componente;
- **Levantamento dos requisitos funcionais e não funcionais:** tanto os requisitos funcionais como os não funcionais necessários a serem incluídos no componente serão levantados a partir de consultas em trabalhos relacionados. Com os requisitos, teremos as informações necessárias para a proposta do componente, levá-lo ao desenvolvimento, documentá-lo e por fim sua reutilização;
- **Implementação dos requisitos:** fase de construção que corresponde à implementação, atendendo aos requisitos (funcionais e não funcionais) e arquitetura definidos nas fases anteriores;
- **Documentação de *software*:** criação de documentação do componente de *software* proposto para que posteriormente o mesmo possa ser reutilizado ou adaptado por desenvolvedores - alguns modelos gráficos também foram gerados.

5.3.3 Validação

A última fase proporcionou resultados finais por meio de um estudo experimental. A Figura 17 ilustra cada uma de suas etapas.

Figura 17. Etapas da terceira fase. (Fonte: elaborada pelo autor)

- **Seleção e estudo dos métodos computacionais:** para a escolha dos métodos a serem utilizados na validação do componente, alguns aspectos foram considerados: (1) importância do método na área de processamento de imagens médicas; (2) o custo computacional exigido pelo método; (3) se o método é passível de paralelização. Após a escolha, um profundo estudo das mesmas foi realizado, juntamente com seu algoritmo, e suas operações matemáticas;
- **Conversão dos métodos:** os métodos de processamento de imagens que foram escolhidos são bem conhecidos na literatura e já possuem seus algoritmos implementados da forma sequencial (tradicional), tais foram convertidos e modificados para o modelo OPENCL, para serem executados de forma paralela na GPU;
- **Análise e validação:** para a validação, aplicações (módulos do 3D SLICER utilizando o componente) foram implementadas para cada método escolhido. Os resultados de tempos de execução entre o algoritmo sequencial e o algoritmo modificado (execução na GPU) foram analisados, comparados e sumarizados.

Capítulo 6

COMPONENTE MEDIADOR

6.1 Considerações Iniciais

Neste capítulo serão apresentados aspectos relacionados à construção do componente, desde a análise da solução, tecnologias relacionadas, até uma visão geral do componente implementado.

6.2 Análise da solução

Antes de iniciar a codificação do componente proposto, um profundo estudo e análise sobre a solução do componente foram realizados, para fins de definição de quais tecnologias seriam adotadas, assim como um levantamento dos requisitos funcionais e não funcionais a serem contemplados no componente de *software*.

Em relação á prática de computação paralela em GPUs, foi adotada a tecnologia OPENCL para a construção do componente em vez do CUDA. Como visto anteriormente este possui algumas vantagens relevantes, por ser um padrão aberto mantido pelo *Khronos Group*, permitir portabilidade de código e suporte a diversos dispositivos de diferentes fabricantes. Porém dois motivos foram predominantes: (1) pelo fato do OPENCL fornecer bibliotecas na

linguagem C++, sendo então possível, e de fácil integração com o 3D SLICER; (2) adotando o OPENCL (de código aberto), permitirá em trabalhos futuros expandir o componente, melhorando a arquitetura do mesmo.

A escolha do *software* 3D SLICER foi muito importante pois permitiu uma alta abstração da complexidade de manipulação da interface gráfica com o usuário no processamento e gerenciamento de imagens tridimensionais médicas. Em relação a estes aspectos, seria inviável para a pesquisa, caso fosse necessário criar ou implementar do zero.

Para a incorporação do componente ao 3D SLICER, das três formas possíveis de desenvolvimento de módulos explicadas anteriormente no capítulo 3, será utilizado o padrão *CLI Modules*. Este modelo simplifica consideravelmente a complexidade de desenvolvimento a ser lidada pelo lado do 3D SLICER. Nesse tipo de módulo, o *front-end* (GUI) é gerado automaticamente por meio de um arquivo XML, e na construção desse XML há como opção uma série de elementos gráficos predefinidos pela plataforma. Para a codificação do módulo é utilizado a linguagem C++.

Neste contexto, visto que tanto o OPENCL como o modelo CLI utilizam a mesma linguagem C++, esta linguagem foi escolhida como base para a codificação integral do componente, e concomitantemente também foi adotado o paradigma de programação genérica na implementação dos códigos, para fins de reutilização do componente por parte de outros desenvolvedores. Para o uso do componente, este deverá ser importado ao projeto do desenvolvedor, trazendo consigo todas as bibliotecas necessárias, tanto para o uso da prática da computação paralela em GPU (OPENCL) como para o funcionamento e construção de módulos no *software* 3D SLICER. Além disso, todo um conjunto de implementações (classes e métodos) estará disponível para uso, abstraindo tanto aspectos técnicos da tecnologia OPENCL, como também aspectos relacionados ao processamento e gerenciamento de imagens 3D, para que o desenvolvedor possa desfrutar da computação paralela de uma forma simplificada em aplicações específicas criadas por ele mesmo.

O CMAKE¹⁶ terá o papel de facilitar o gerenciamento de todas essas dependências exigidas pelo componente. É um *software* livre multiplataforma e de código aberto para a gestão do processo de construção de *software* independente do compilador. Ele é projetado para facilitar aplicações que dependem de várias bibliotecas, no caso do presente trabalho, bibliotecas do OPENCL e 3D SLICER, e outras que consequentemente serão necessárias. Ele é usado em conjunto com ambientes de compilação nativa, como o *Unix*, *Xcode* da *Apple* e

¹⁶ <https://cmake.org/overview/>

Microsoft Visual Studio. Tem dependências mínimas, necessitando apenas de um compilador C++ em seu próprio sistema de compilação.

6.3 Estudo das Tecnologias Relacionadas

De modo geral, para o desenvolvimento completo do projeto, foram necessários o domínio e a compreensão de algumas tecnologias relacionadas. A Figura 18 ilustra as principais tecnologias que foram estudadas no presente projeto:

Figura 18. Arquitetura das tecnologias¹⁷.

- **VTK:** o *Visualization Toolkit* é um pacote *open-source*, *software* livre e disponível para computação gráfica 3D, processamento de imagem e visualização. É composto por uma biblioteca em C++ e várias camadas de interface. VTK suporta uma ampla variedade de algoritmos de visualização, incluindo escalar, vetor, tensor, textura e métodos volumétricos, bem como técnicas de modelagem avançadas, tais como modelagem implícita, redução de polígono, suavização de malha, corte, contorno, etc. Possui uma extensa estrutura de visualização de

¹⁷ Engineering - <https://www.na-mic.org/pages/Engineering>

informação e um conjunto de widgets de interação 3D. VTK também é multiplataforma e roda em plataformas *Linux*, *Windows*, *Mac* e *Unix*¹⁸;

- **ITK:** Insight Segmentation and Registration Toolkit é um sistema multiplataforma de código aberto que fornece aos desenvolvedores um extenso conjunto de ferramentas de software para análise de imagem. Desenvolvido através de metodologias de programação extremas, ITK emprega algoritmos de ponta para registrar e segmentação de dados multidimensionais¹⁹;
- **QT:** é um *framework* de aplicações multiplataforma que é amplamente utilizado para o desenvolvimento de interfaces gráficas em aplicações que podem ser executados em várias plataformas de *software* e *hardware* com pouca ou nenhuma mudança na base de código subjacente, enquanto continuam a ser um aplicativo nativo com as capacidades e velocidade do mesmo. Ele vem com um editor de código (IDE) e é integrado com ferramentas para projetar, codificar, testar, implantar e manter seu *software* durante todo o ciclo de vida do produto²⁰;
- **CMAKE:** é um *open-source*, multiplataforma da família de ferramentas projetadas para construir, testar e empacotar *software*. CMAKE é usado para controlar o processo de compilação de *software* utilizando a plataforma simples e arquivos de configuração do compilador independente e gerar arquivos *makefiles* e espaços de trabalho nativas que podem ser usados no ambiente de compilador de sua escolha. O conjunto de ferramentas foi criado pela necessidade de um poderoso ambiente de compilação multiplataforma para projetos de código aberto, como o ITK e VTK²¹;
- **OPENCL:** umas das tecnologias principais do projeto, detalhada nas seções anteriores, é um *framework* para a computação paralela²²;
- **NRRD:** é um formato de biblioteca e de arquivo projetado para suportar visualização científica e processamento de imagem envolvendo dados N-dimensionais. NRRD significa *nearly raw raster data*. Muitas imagens tridimensionais utilizadas no 3D SLICER utilizam esse formato²³;

¹⁸ The Visualization Toolkit - <http://www.vtk.org/>

¹⁹ Segmentation & Registration Toolkit - <https://itk.org/>

²⁰ Qt framework - <https://www.qt.io/>

²¹ Qt framework - <https://www.qt.io/>

²² OPENCL Overview - <https://www.khronos.org/opencl/>

²³ Teem:nrrd - <http://teem.sourceforge.net/nrrd/>

- **MRML:** *Medical Reality Modeling Language* (MRML) é uma linguagem implementada como um tipo de documento no formato XML, com novas *tags* definidas para manipular tipos de dados de imagens médicas, como volumes, modelos ou transformações de coordenadas. Para um aprofundamento sobre o software 3D SLICER, é interessante ao desenvolvedor expandir seus conhecimentos para esta linguagem²⁴;
- **PYTHON:** é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. No 3D SLICER o padrão Scripted Modules são módulos escritos em Python. Conforme descritos anteriormente no capítulo 3, são recomendados para uma rápida prototipação e um desenvolvimento de fluxo de trabalho personalizado;
- **UBUNTU:** um sistema operacional construído a partir do núcleo *Linux*. O qual oferece rapidez e simplicidade, na criação dos ambientes de desenvolvimento e instalação de bibliotecas necessárias para o projeto.

6.4 Visão Geral do Componente

Diante dos desafios que identificamos para o uso eficiente de GPUs em sistemas gerais e do contexto de processamento de imagens médicas tridimensionais, o componente foi construído para proporcionar a prática da computação paralela. Além disso, com o objetivo de minimizar a complexidade para a criação de aplicações pelo desenvolvedor. Em resumo, o componente consegue trazer para os usuários as seguintes contribuições:

- **Desempenho:** permite ao usuário criar aplicações de alto desempenho no contexto de processamento de imagens tridimensionais, tendo como benefício um ganho no tempo de execução do método modificado. Esse ganho alcançado vai depender da capacidade de computação paralela do dispositivo disponível;
- **Comunicação com *hardwares*:** o componente proporciona um fácil acesso e comunicação com os dispositivos compatíveis e disponíveis na máquina do usuário, abstraindo uma boa parte da complexidade da comunicação e distribuição de cargas dos dados entre os dois níveis de paralelismo do dispositivo (unidades e

²⁴ MRML Slicer Wiki - https://www.slicer.org/wiki/Adding_MRML

elementos de computação). Algumas operações de baixo nível são mais propensas a erros e aumentam a complexidade de codificação;

- **Reutilização:** o componente foi construído centrado na ideia de reutilização de software, uma perspectiva interessante, o qual adquire benefícios como:
 - 1) **Autocontido:** após a implementação do componente, este para ser reutilizado possui dependência apenas com o OPENCL, podendo ser importado em projetos como do 3D SLICER, porém com outras tecnologias: VTK, ITK e qualquer outra tecnologia na linguagem C++;
 - 2) **Identificação:** o componente deve ser claramente identificável, em um pacote contendo todos os arquivos necessários, em vez de distribuídos e misturados com outras tecnologias de *software*;
 - 3) **Simplificação:** por meio do componente, basta que os usuários utilizem as funcionalidades fornecidas. Será abstraído de complexidade e detalhes que não são necessários ao reuso provenientes do OPENCL e 3D SLICER, o qual simplificará a codificação do usuário desenvolvedor e também permitirá implementação de novas necessidades/melhorias quando necessário;
 - 4) **Documentação:** a documentação do componente deve ser escrita de forma clara e detalhada, indispensável para a compreensão e reutilização do componente por futuros desenvolvedores.
- **Científico:** além do aspecto de reutilização descrito anteriormente, muito importante para usuários pesquisadores que seguirem a mesma linha de pesquisa, o componente também facilitará confrontar o método modificado para a computação paralela com o método sequencial, de uma forma simples. Pois o usuário terá controle pela interface gráfica do 3D SLICER, podendo escolher de qual maneira irá executar o método, de forma paralela ou sequencial, enviados para dispositivos CPU ou GPU, e também validar se ambos métodos convergem para a mesma imagem resultante resultado. Para cada execução disparada pelo usuário, o componente registra o tempo de execução e exibe de uma forma resumida, facilitando posteriormente a criação de gráficos de comparação entre os tempos de execução.

Durante todo o trabalho seguiram-se boas práticas de engenharia de *software*, gerando uma documentação para todo o processo de desenvolvimento do componente, desde a fase posterior a codificação que envolveu as atividades de definição do escopo do problema,

análise e definição das tecnologias, levantamento de requisitos funcionais e não funcionais, definição do público alvo, até construção de diagramas. Após a codificação do componente, foi somado à documentação um manual de suporte, contendo informações técnicas relevantes para a utilização do mesmo, abrangendo uma descrição geral técnica, requisitos mínimos, manual de instalação e funções implementadas. O documento elaborado seguiu um modelo da IEEE²⁵ (para maiores detalhes o documento se encontra no Apêndice A). Todo o código do componente, módulos do *3D SLICER*, assim como outras aplicações e testes realizados durante o projeto, estarão hospedados e disponíveis no repositório GITHUB do Laboratório de Computação em Sinais e Imagens Médicas (CSIM), acessível em <https://github.com/CSIM-Toolkits/opencv>

6.5 Considerações Finais

Diante da proposta desafiadora neste trabalho, a análise da solução visou escolher a melhor estratégia em termos de tecnologia para a construção do componente. A escolha de todas as tecnologias anteriores também reflete os objetivos pretendidos do trabalho, que não possui teor econômico, mas sim científico, com isso foi adotado o OPENCL para a prática da computação paralela em vez do CUDA, o *software* 3D SLICER para o processamento de imagens tridimensionais e como linguagem principal o C++, todas as tecnologias de código aberto.

Após a implementação do componente, podem-se identificar algumas características e contribuições proporcionadas por ele aos usuários: desempenho, comunicação com *hardwares*, reutilização e científico. Durante todo o trabalho, as etapas foram sendo documentadas e ao fim um documento foi elaborado e anexado. Após a criação do componente foi necessário realizar uma validação, por isso o próximo capítulo apresenta os experimentos que foram realizados, e seus resultados analisados e discutidos, a fim de comprovar seu funcionamento proporcionando a prática da computação paralela em GPU, e sua capacidade de ganho no tempo de execução do método modificado.

²⁵ IEEE Standard Association - <https://standards.ieee.org/findstds/standard/830-1993.html>

Capítulo 7

RESULTADOS

7.1 Considerações Iniciais

Para evidenciar o ganho no tempo de execução da computação paralela em relação à execução sequencial, alguns experimentos foram realizados fazendo o uso do componente mediador. Primeiramente um experimento introdutório foi realizado utilizando o método de multiplicação de matrizes de ordem elevadas, um método simples que permitiu simular uma alta carga de processamento. Tal escolha foi feita para adquirir um aprendizado inicial e para verificar a capacidade do componente em proporcionar ganhos relevantes (de tempo) do método de multiplicação de matrizes executado na GPU.

Após o experimento introdutório, foram realizados mais três experimentos, em que foram escolhidos três métodos de processamento de imagens: filtro de média móvel, filtro morfológico de erosão/dilatação e filtro anisotrópico. Optou-se por métodos bem difundidos na literatura e que fosse de fácil acesso aos seus algoritmos sequenciais. Para estes experimentos foi necessária a implementação de módulos no 3D SLICER. Os módulos foram criados no padrão CLI (*Command Line Interface*), e os algoritmos, tanto sequencial como o modificado (OPENCL), foram incorporados. Interfaces personalizadas também foram implementadas de acordo com os parâmetros de entrada de cada experimento.

Em cada experimento os métodos tiveram seus algoritmos adaptados para o uso do OPENCL, e também em cada experimento foi confrontado o tempo de execução do método sequencial com o método modificado. Para isso, testes de execução foram realizados variando

diferentes parâmetros de entrada. Ao fim de cada execução foi registrado o tempo para uma análise posterior. Em todos os experimentos foram utilizados os seguintes dispositivos:

- **CPU Intel i5:** para a execução sequencial na CPU;
- **GPU AMD Radeon R7 M260:** para a execução de forma paralela utilizando o OPENCL, a GPU possui 6 *cores*, cada um contendo 256 *threads*.

É muito importante salientar que a medição do tempo de execução, feita nos métodos modificados, é realizado no *host* (aplicação C++) desde o momento da conexão com o dispositivo GPU, envio do *kernel* para o dispositivo (transferência de memória entre CPU para GPU) e até a obtenção do retorno da imagem resultante (transferência de memória entre GPU para CPU).

7.2 Multiplicação de Matrizes (Experimento 1)

A multiplicação de matrizes de ordens elevadas é um exemplo interessante e óbvio para simular uma tarefa de alta demanda computacional. Uma das primeiras tentativas de cálculos não gráficos em uma GPU foi uma multiplicação de matrizes (LARSEN, 2001). Na literatura já existem iniciativas de trabalhos como (CZAJKOWSKI, 2012; MATSUMOTO et al., 2012) que utilizam métodos de multiplicação matrizes aplicadas para a prática da computação em GPUS utilizando o OPENCL.

Para fins de estudo e aprendizagem da tecnologia OPENCL, um experimento introdutório foi realizado, para isso um algoritmo de multiplicação de matrizes foi implementado para ser executado na CPU de forma sequencial, e outro para ser executado na GPU de forma paralela. Os tempos de execução foram registrados e analisados.

O método sequencial de multiplicação de matrizes bidimensional é simples de implementação, de complexidade $O(n^3)$, consiste de um laço triplo, os dois laços mais exteriores para fazer a iteração sobre o índice da linha e a coluna, enquanto o laço mais interno para executar o produto escalar dos vetores da linha e coluna. Definiu-se que as matrizes de entrada na operação de multiplicação possuem a mesma ordem. Segue o trecho do código sequencial em C++ na Figura 19.


```
for(int i=0; i<size; ++i)
for(int j=0; j<size; ++j)
for(int k=0; k<size; ++k)
{
    result[i][j] += matrixA[i][k] * matrixB[k][j];
}
```

Figura 19. Código C++ sequencial de multiplicação de matriz. (Fonte: elaborada pelo autor)

Para o método modificado, uma aplicação (*host*) OPENCL foi implementada para dar fácil acesso à computação paralela na GPU, um *kernel* foi criado para implementar toda a lógica de multiplicação de matriz que será executado na GPU. Veja na Figura 20. Recapitulando o que foi explicado nos capítulos de fundamentações, note que no *Kernel* abaixo todos os objetos de entrada serão executados usando a memória global, segundo o modelo de memória OPENCL, e no código veja também que os dois laços mais externos foram retirados, pois cada instância do *kernel* será executada de forma independente, por meio do espaço de índice (NDRANGE). Além do *Kernel*, o programa *host* também tem um papel fundamental a desempenhar como: localização dos dispositivos compatíveis OPENCL, carregamento do *Kernel* e para a definição das dimensões NDRANGE.

```
__kernel void matvec_mult( __global float* matrixA, __global float* matrixB,
                          __global float* matrixResult, __global int size) {
    int i = (int) get_global_id(0);
    int j = (int) get_global_id(1);

    for(int k = 0; k < size; k++){
        matrixResult[i*size+j] += matrixA[i*size +k] * matrixB[k*size+j];
    }
}
```

Figura 20. *Kernel* para multiplicação de matrizes. (Fonte: elaborada pelo autor)

7.2.1 Resultados do Experimento 1

Foram realizados dois testes para analisar o tempo de execução do método de multiplicação de matrizes, em dois tipos de dispositivos (CPU e GPU), conforme descritos anteriormente. As matrizes passadas como parâmetro de entrada eram de mesma ordem e seus valores internos gerados aleatoriamente entre 0 e 10. As ordens das matrizes a serem executadas foram variadas, aumentando de 1000 em 1000 o tamanho, de modo que o método aumente o custo computacional progressivamente. Os testes foram:

1. Método executado de forma sequencial na CPU;
2. Método executado de forma paralela na GPU, usando OPENCL.

Em cada teste, os valores do tempo de execução em segundos foram registrados para cada tamanho de matriz (1000 até 6000), e depois gráficos foram gerados para a comparação da performance.

Como resultado na Figura 21, foram representados graficamente os resultados do tempo de execução do método tradicional executado de forma sequencial na CPU (linha azul) e do método OPENCL executado de forma paralela na GPU (linha laranja).

Figura 21. Resultado do tempo de execução para multiplicação de matrizes. (Fonte: elaborada pelo autor)

Veja que o tempo de execução do método executado de forma sequencial na CPU tende a aumentar de forma exponencial para este conjunto de dados, enquanto no método modificado a ser executado na GPU o tempo de execução cresce ligeiramente. Na Tabela 2 é apresentado os dados e números da análise, e a aceleração do tempo ganho pelo método mais rápido foi calculado. Segundo os resultados, em uma operação de multiplicação entre duas matrizes de tamanho 7000x7000, a tarefa é concluída em aproximadamente 305 minutos na CPU enquanto usando a computação paralela na GPU a conclusão leva um pouco mais de 30 minutos. Podemos dizer, para este conjunto de dados, que o método modificado foi em média 10 vezes mais rápido do que o método tradicional e sequencial.

Tabela 2. Dados numéricos do tempo de execução para multiplicação de matrizes. (Fonte: elaborada pelo autor)

Tamanho da matriz	1000	2000	3000	4000	5000	6000	7000
CPU	12.13	108.46	385.49	916.62	1853.92	2949.17	4107.77
GPU	1.19	10.40	38.21	93.37	196.48	315.30	407.05
Aceleração	~11x	~11x	~10x	~9x	~9x	~9x	~10x

7.2.2 Conclusão do Experimento 1

A partir da análise do gráfico e dos números apresentados é fácil identificar um notável aumento de velocidade obtida pelo método modificado que foi executado de forma paralela nos núcleos da GPU.

O principal objetivo deste experimento realizado, além de mostrar a potencial aceleração no tempo de execução que se pode alcançar na computação paralela em comparação com o método sequencial, o experimento foi essencial para o aprendizado da tecnologia OPENCL e da linguagem C++, permitindo então o avanço e progresso no projeto.

Dando continuidade, os próximos experimentos irão trabalhar no contexto de processamento de imagens tridimensionais, utilizando o *framework* OPENCL e o *software* 3D SLICER.

7.3 Filtro de Média Móvel (Experimento 2)

O experimento consiste na implementação de um filtro de média móvel para imagens tridimensionais que serão carregadas pelo usuário por meio do módulo do 3D SLICER. O filtro foi implementado de duas maneiras uma de forma sequencial e outra utilizando o OPENCL para ser executado de forma paralela.

Primeiramente um módulo do tipo CLI (*Command Line Interface*) foi criado com o nome *MeanFilter*. Como mostra na Figura 22 o módulo carregado está localizado no canto esquerdo da tela do 3D SLICER

Figura 22. Módulo criado para o experimento de média móvel. (Fonte: elaborada pelo autor)

O método sequencial do filtro de média móvel é bem simples, trata-se de um laço triplo que percorre toda a matriz tridimensional *voxel* a *voxel* e outro laço triplo interno responsável por fazer a operação de convolução para cada *voxel*. Já o método do *kernel* que será enviado para ser executado na GPU é semelhante ao sequencial, porém o laço triplo mais externo é substituído pelos identificadores dos *work-items* de um espaço de índice NDRANGE contendo três dimensões. Os trechos do código de ambos os métodos estão disponibilizados no Apêndice B.

A interface gráfica do módulo interage com o usuário, como mostrado na Figura 23, e permite que a mesma escolha o formato da imagem 3D (por exemplo: NRRD ou DICOM) a ser carregada e renderizada no visualizador. O usuário também escolhe qual dos dois métodos deseja utilizar no processamento da imagem, o que levará a tarefa a ser executada na CPU de

forma sequencial ou o método modificado usando OPENCL usando a computação paralela para executar a tarefa em uma GPU disponível. O último parâmetro que o usuário pode escolher é o tamanho da máscara de vizinhança que será aplicada na imagem, tendo algumas opções de tamanho. Como se trata de uma imagem 3D, a máscara também é volumétrica.

A interface de configuração do filtro de média móvel apresenta quatro controles principais: 1. 'Input Volume' com uma caixa de seleção contendo 'Input3DImage'. 2. 'Output Volume' com uma caixa de seleção contendo 'Select a Volume'. 3. 'Method Type' com dois botões de opção: 'GPU' (selecionado) e 'CPU'. 4. 'Size of Mask' com uma série de botões de opção numerados de 2 a 8, onde o botão '2' está selecionado.

Figura 23. Elementos gráficos e seus parâmetros do filtro de média móvel. (Fonte: elaborada pelo autor)

Após definir os parâmetros de entrada (imagem, tipo de método e tamanho da máscara) o processamento é iniciado e ao término é lançada a imagem resultante e o tempo de execução em segundos que levou para obtê-la.

7.3.1 Resultados do Experimento 2

Para simular uma alta carga de processamento de dados, foi utilizada uma imagem tridimensional proveniente do próprio banco de imagens do 3D SLICER. Uma imagem de dimensões 441x321x215 foi utilizada no experimento, variando apenas o tamanho N da máscara de vizinhança, que também é tridimensional e com valores iguais em suas dimensões (x,y,z).

Como resultado, na Figura 24, foram representados graficamente os resultados do tempo de execução para a conclusão do filtro de média móvel, do método tradicional executado de forma sequencial na CPU (linha azul) e do método OPENCL executado de forma paralela na GPU (linha laranja). O eixo X do gráfico representa o tamanho da máscara aplicada no filtro, que interfere no tempo de execução. O custo computacional aumenta proporcionalmente ao tamanho da máscara.

Figura 24. Comparação no tempo de execução do filtro de média móvel (1). (Fonte: elaborada pelo autor)

Aqui vemos que a execução sequencial (linha azul) tem um crescimento progressivo alto, diferentemente da execução paralela (linha laranja) que cresce ligeiramente apenas. É possível verificar, para esse conjunto de amostras, que o algoritmo modificado é em média 10x mais rápido quando comparado com o sequencial. Na Tabela 3 é apresentados os dados brutos do experimento, juntamente com o ganho entre os algoritmos

Tabela 3. Dados numéricos do tempo de execução do filtro de média móvel (1). (Fonte: elaborada pelo autor)

Tamanho da máscara	9	12	15	18	21	25	28
CPU	15.84	43.87	63.38	103.08	125.33	179.77	204.87
GPU	1.91	4.64	6.34	9.56	11.89	16.38	21.01
Aceleração	~8x	~9x	~10x	~11x	~10x	~11x	~10x

Adicionalmente, o experimento foi realizado novamente, porém, agora, fixando o tamanho da máscara de vizinhança em 28x28x28 e variando o tamanho das imagens de entrada (eixo Y). O gráfico é ilustrado na Figura 25.

Figura 25. Comparação no tempo de execução do filtro de média móvel (2). (Fonte: elaborada pelo autor)

Como resultado, algo interessante é identificado, pois similarmente ao experimento anterior o algoritmo modificado para este conjunto de amostras também é em média 10x mais rápido quando comparado com o sequencial. Na Tabela 4 são apresentados os dados brutos do experimento, juntamente com o ganho entre os algoritmos.

Tabela 4. Dados numéricos do tempo de execução do filtro de média móvel (2). (Fonte: elaborada pelo autor)

Tamanho da imagem	256x256x51	512x512x51	441x321x215	800x800x50
CPU	22.27	53.14	204.87	481.23
GPU	2.9	5.7	21.01	47.21
Aceleração	~8x	~10x	~10x	~10x

7.3.2 Conclusão do Experimento 2

Semelhante ao experimento anterior, o objetivo também é verificar e evidenciar o ganho em tempo de execução que houve, comparando o método OPENCL com o método sequencial.

A partir da análise dos resultados apresentados, novamente foi evidenciado um relevante ganho na aceleração de execução comparando o método sequencial com o método OPENCL. Pelos dados numéricos também é evidenciado que, quanto maior a carga ou custo computacional da tarefa, maior é o ganho alcançado no tempo de execução do método que usou a computação paralela.

Um filtro de média móvel é relativamente simples não exigindo um custo computacional alto. Porém por se tratar de uma imagem volumétrica, a quantidade de dados aumenta consideravelmente exigindo uma carga de trabalho alta.

Quando se trata de imagens médicas, diversas modalidades exigem altas cargas de trabalho (por exemplo: grande quantidade de dados e cálculos complexos), acarretando algoritmos computacionalmente caros, não satisfazendo muitas vezes a exigência de tempo no processamento em ambientes reais, como, por exemplo, em serviços de saúde para a prática clínica.

7.4 Filtro Morfológico de Erosão/Dilatação (Experimento 3)

Para este experimento, foram utilizadas as operações morfológicas de erosão e dilatação, as quais são fundamentais para a manipulação morfológica de imagens. O efeito da dilatação aumenta ou engrossa, enquanto a erosão encolhe ou afina objetos de uma imagem (DHAWAN,2011).

Foi criado um único módulo no 3D SLICER para o experimento, permitindo ao usuário escolher se deseja aplicar a operação de erosão ou dilatação na imagem tridimensional. Semelhante ao experimento anterior, uma interface gráfica personalizada foi criada para interagir com o usuário, como parâmetros de entrada para temos: o *foreground* que é o valor de intensidade do *voxel* que será aplicado à erosão ou dilatação e o tamanho da vizinhança da operação (*radius*). O parâmetro chamado *background* é utilizado apenas na erosão, este determina o valor de intensidade do *voxel* após o mesmo sofrer a erosão, a Figura 26 temos a ilustração do módulo construído.

The image shows a software interface for the 'DilateErode' filter. At the top, there is a dropdown menu labeled 'DilateErode'. Below it, a 'Parameter set:' dropdown is set to 'DilateErode'. Underneath, there is a section titled 'IO' with several parameters: 'Input Volume' is set to 'Input3DImage'; 'Output Volume' is set to 'Select a Volume'; 'ForeGround' is set to '0'; 'BackGround' is set to '0'; 'Radius' is set to '0'; 'Boundary To Foreground' is an unchecked checkbox; 'Method Type' has radio buttons for 'GPU' (selected) and 'CPU'; and 'Filter Type' has radio buttons for 'Dilate' (selected) and 'Erode'.

Figura 26. Elementos gráficos e seus parâmetros do filtro de dilatação e erosão. (Fonte: elaborada pelo autor)

Para mais detalhes sobre a codificação dos algoritmos sequencial e modificado, das operações de dilatação e erosão, veja respectivamente no Apêndice C e D.

7.4.1 Resultados do Experimento 3

Para análise de performance, foi utilizada apenas a operação de dilatação, visto que são operações bem similares (dilatação e erosão), tendo a mesma complexidade praticamente. Foi utilizada uma imagem tridimensional proveniente do próprio banco de imagens do 3D SLICER, uma imagem de dimensões 441x321x215 foi utilizada no experimento, variando apenas o tamanho N da máscara de vizinhança, que também é tridimensional e com valores iguais em suas dimensões (x,y,z).

Como resultado, na Figura 27, foram representados graficamente os resultados do tempo de execução para a conclusão do filtro de dilatação, do método tradicional executado de forma sequencial na CPU (linha azul) e do método OPENCL executado de forma paralela na GPU (linha laranja). O eixo X do gráfico representa o tamanho da máscara aplicada no filtro, que interfere no tempo de execução, o custo computacional aumenta proporcionalmente ao tamanho da máscara. Esse resultado teve um ganho bem mais expressivo quando comparado ao filtro de média móvel.

Figura 27. Comparação no tempo de execução do filtro de dilatação (1). (Fonte: elaborada pelo autor)

Pode-se identificar que, para esse conjunto de dados, o algoritmo modificado é em média 37x mais rápido quando comparado com o sequencial. Na Tabela 5 são apresentados os dados brutos do experimento, juntamente com o ganho entre os algoritmos.

Tabela 5. Dados numéricos do tempo de execução do filtro de dilatação (1). (Fonte: elaborada pelo autor)

Tamanho da máscara	3	4	5	6	7	8	9
CPU	75.09	168.35	311.94	501.15	736.49	986.86	1355.03
GPU	2.89	4.7	7.73	12.78	19.75	25.55	34.26
Aceleração	~26x	~36x	~40x	~39x	~37x	~39x	~40x

Adicionalmente, o experimento foi realizado novamente, porém, agora, fixando o tamanho da máscara de vizinhança em 9x9x9 e variando o tamanho das imagens de entrada (eixo Y). O gráfico é ilustrado na Figura 28.

Figura 28. Comparação no tempo de execução do filtro de dilatação (2). (Fonte: elaborada pelo autor)

Como resultado, algo interessante é identificado, pois, similarmente ao experimento anterior, o algoritmo modificado para este conjunto de amostras também é em média 10x mais rápido quando comparado com o sequencial. Na Tabela 6 são apresentados os dados brutos do experimento, juntamente com o ganho entre os algoritmos.

Tabela 6. Dados numéricos do tempo de execução do filtro de dilatação (2). (Fonte: elaborada pelo autor)

Tamanho da imagem	256x256x51	512x512x51	441x321x215	800x800x50
CPU	149.125	321.183	1455.03	2155.75
GPU	3.9	7.88	34.26	45.36
Aceleração	~38x	~41x	~42x	~48x

7.4.2 Conclusão do Experimento 3

O experimento consistiu na aplicação de duas operações morfológicas: dilatação e erosão. Ambas as operações tiveram seus algoritmos adaptados para a prática da computação paralela, e testes foram realizados com imagens tridimensionais.

O módulo do 3D SLICER criado para este experimento poderá facilmente ser expandido, visto que a dilatação e erosão são bases para outras operações morfológicas como: abertura e fechamento (DHAWAN,2011).

Percebemos um ganho mais expressivo no tempo de execução do algoritmo de dilatação em contrapartida ao filtro de média móvel. Fazendo uma comparação em nível de complexidade de algoritmo, vemos que o filtro de dilatação possui uma maior quantidade de instruções e condições do que o filtro de média móvel.

7.5 Filtro de Difusão Anisotrópico (Experimento 4)

Como último experimento, foi utilizado um filtro de difusão anisotrópico clássico. Este é um método de suavização iterativa, que ao contrário dos filtros lineares tradicionais, os quais desfocam as bordas, visa reduzir o ruído da imagem sem remover partes significativas do conteúdo da imagem, tipicamente arestas, linhas ou outros detalhes importantes para a interpretação da imagem (PERONA,1990). O módulo no 3D SLICER foi criado com os seguintes parâmetros de entrada. Veja a Figura 29.

▼ AnisotropicClassic	
Parameter set:	AnisotropicClassic
▼ IO	
Input Volume	Input3DImage
Output Volume	Input3DImage
Method Type	☒ GPU ☐ CPU
Num. Iteration	1
Coef. Kappa	15.0
Coef. Delta	0.125

Figura 29. Elementos gráficos e seus parâmetros do filtro de difusão anisotrópico. (Fonte: elaborada pelo autor)

Como parâmetros de entrada, o método de difusão anisotrópico possui:

- **Número de iterações:** se refere basicamente à quantidade de vezes que a operação de difusão anisotrópica será realizada, até a imagem resultante final;
- **Coefficiente *Kappa* e *Delta*:** respectivamente definidos por *condutance* e *time step*, ambos os parâmetros têm uma relação intrínseca com a intensidade de filtragem anisotrópica aplicada à imagem. O trabalho (PERONA,1990) descreve em detalhes a definição e importância desses coeficientes.

No contexto de processamento de imagens, além de o método ser mais complexo do que os anteriores, o motivo principal da escolha do mesmo para o experimento final é a presença do parâmetro de número de iterações, que ao serem definidos valores altos, isso aumenta bastante a complexidade e tempo de execução, visto que em N iterações o método será executado N vezes até a obtenção final da imagem resultante.

De maneira geral para a implementação do método de difusão anisotrópico, trata-se de um laço triplo que percorre toda a matriz tridimensional *voxel* a *voxel*, cuja nova intensidade do *voxel* será definida a partir de cálculos entre a intensidade de difusão anisotrópica e os gradientes nas diferentes direções: norte, sul, leste, oeste, nordeste, noroeste, sudoeste, sudeste. A implementação do *kernel* é semelhante ao sequencial, porém o laço triplo mais externo é substituído pelos identificadores dos *work-items* de um espaço de índice NDRANGE contendo três dimensões. Os trechos do código de ambos os métodos estão disponibilizados no Apêndice E.

7.5.1 Resultados do Experimento 4

Para análise de performance foi utilizada uma imagem tridimensional proveniente do próprio banco de imagens do 3D SLICER, uma imagem de dimensões 441x321x215 foi utilizada no experimento, variando apenas o número de iterações do filtro de difusão anisotrópico.

Como resultado, na Figura 30, foram representados graficamente os resultados do tempo de execução para a conclusão do filtro de difusão anisotrópico, do método tradicional executado de forma sequencial na CPU (linha azul) e do método OPENCL executado de forma paralela na GPU (linha laranja). O eixo X do gráfico representa o número de iterações,

esperava-se um ganho alto quando confrontado os tempos de execução na abordagem sequencial e paralela devido ao aumento relevante no número de iterações realizados pelo algoritmo no experimento.

Figura 30. Comparação no tempo de execução do filtro de difusão anisotrópico (1). (Fonte: elaborada pelo autor)

Analisando os tempos de execução e calculando o ganho alcançado pela abordagem de computação paralela na GPU, foi evidenciado um resultado relativamente ruim para o algoritmo modificado. Em média teve um ganho de apenas 4x vezes em comparação ao método sequencial, na Tabela 7 são apresentados os dados brutos do experimento.

Tabela 7. Dados numéricos do tempo de execução do filtro de difusão anisotrópico (2). (Fonte: elaborada pelo autor)

Número de iterações	50	100	150	200	250	300
CPU	214.25	430.00	648.60	865.30	1094.85	1316.80
GPU	55.50	113.85	172.25	230.05	290.5	347.8
Aceleração	~4x	~4x	~4x	~4x	~4x	~4x

Para poder explicar o ganho baixo alcançado pelo algoritmo modificado, foi analisado atentamente toda a implementação do módulo criado exatamente para este experimento, e

com isso identificada a razão do ocorrido. O filtro de difusão anisotrópico trata-se de um problema de dependência de dados, a partir de sua segunda iteração, ela depende do resultado produzido pela iteração anterior.

No modelo OPENCL, tanto no filtro de média móvel quanto o de dilatação/erosão, é necessário que o *kernel* seja submetido pela aplicação *host* para ser executado na GPU uma única vez. Após a conclusão da execução na GPU, a mesma retorna ao *host* a imagem resultante. Como o filtro de difusão anisotrópico possui N iterações de sua execução, e cada uma das N iterações depende do resultado da anterior, ou seja, serão N *kernels* a serem submetidos para a GPU, um após o outro.

Nesta operação de envio do *kernel* da aplicação *host* para a GPU ocorre uma transferência de dados entre a memória principal da CPU, na qual se encontra a aplicação *host*, para a memória da GPU em que será executado o método. E também, após a conclusão da execução na GPU, outra transferência de memória ocorre para retornar a imagem resultante ao *host*. Esse tempo de transferência de memória foi analisado e leva-se em média aproximadamente 0.5 segundos. Isto consegue explicar o motivo para um ganho pouco expressivo do método.

Na execução do filtro de difusão anisotrópico, caso, por exemplo, seja definido como $N=200$ iterações, ao fim da execução completa do método apenas com transferências de memória serão gastos aproximadamente $0.5 \times 200 \times 2 = 200$ segundos (cada iteração realiza um envio e um retorno). Para uma melhor compreensão do que foi explicado, uma ilustração foi elaborada na Figura 31.

Figura 31. Envio do *Kernel* para GPU a) a aplicação *host* enviando os parâmetros de entrada para a GPU b) a aplicação *host* recuperando a imagem resultante. (Fonte: elaborada pelo autor)

7.5.2 Conclusão do Experimento 4

O experimento consistiu na aplicação de um método mais complexo comparado aos demais experimentos, para a simulação de um alto processamento de dados. Esperava-se uma média no ganho de tempo de execução do método modificado bem maior do que nos experimentos anteriores. Isto não se observou, evidenciando um ganho relativamente baixo de apenas 4x. Para isso uma análise foi realizada na implementação do módulo do 3D SLICER e foi descoberto que o filtro de difusão anisotrópica se tratava de um problema de caráter de dependência de dados, levando a aplicação *host* para realizar N iterações

No trabalho (REGAN,2017) é estudada a prática da computação paralela em GPU em processamento de imagens provenientes da técnica *Laser Speckle Imaging*, uma técnica não invasiva para estudar o fluxo sanguíneo (isto é, glóbulos vermelhos) com uma resolução espacial e temporal elevada. No trabalho os autores explicitam os tempos necessários para fazer a transferência de dados entre a memória principal da CPU com a GPU e vice-versa.

7.6 Considerações Finais

Este capítulo apresentou a validação que foi realizada no componente mediador. Para isso um estudo experimental foi realizado, no qual métodos foram implementados para a prática da computação paralela usando GPU, nos experimentos foram utilizados os seguintes métodos: multiplicação de matrizes de ordens elevadas, filtro de média móvel, filtro morfológico de dilatação/erosão e filtro de difusão anisotrópico.

Em cada experimento os métodos tiveram seus algoritmos adaptados para o uso do OPENCL, e também em cada experimento foi confrontado o tempo de execução do método sequencial com o método modificado. Para isso, testes de execução foram realizados variando diferentes parâmetros de entrada. Ao fim de cada execução foi registrado o tempo. Os seus resultados foram analisados e discutidos.

O experimento 1 (introdutório) foi realizado com objetivo de adquirir conhecimento e mostrar a capacidade do componente mediador. Mesmo sendo um método simples, evidenciou-se uma média de 10x de ganho de tempo. No experimento 2 utilizou-se o método de média móvel, um filtro tradicional da área de processamento de imagens médicas. Como resultado o método modificado evidenciou uma média de ganho de 10x mais rápido do que o

método sequencial. Esta média foi observada nos dois testes realizados dentro do experimento.

No experimento 3, o filtro morfológico de dilatação obteve o maior valor de ganho dentre os métodos escolhidos, o primeiro teste houve um ganho de até 40x, e no segundo teste chegando até 50x comparado ao método sequencial. E por fim, no experimento 4, o filtro de difusão anisotrópico, um método mais robusto e complexo do que os demais, esperava-se um ganho bem mais expressivo do que os experimentos anteriores, porém teve um ganho de apenas 4x. Isto foi analisado e discutido, identificando o método como um problema de dependência de dados.

Foi de grande relevância a realização dos experimentos, não apenas para comprovar o funcionamento do componente para a prática da computação paralela em GPUs, porém, o mais importante foi demonstrar o ganho que pode ser alcançado em métodos adaptados para a computação paralela. Observa-se que, com exceção do experimento introdutório utilizando a multiplicação de matrizes, na literatura não foi encontrado nenhum trabalho que utilizou quaisquer tecnologias de computação paralela para os métodos adotados neste trabalho.

Capítulo 8

CONCLUSÃO

8.1 Considerações Iniciais

Neste capítulo é apresentada uma síntese dos passos percorridos no decorrer deste trabalho conjuntamente com as principais conclusões a que o estudo chegou. Por fim, são discutidas as limitações do estudo e as sugestões para trabalhos futuros.

8.2 Considerações Finais do Trabalho

Temos acompanhado uma forte evolução tecnológica e computacional; cada vez mais aplicações e métodos vêm exigindo uma alta demanda de processamento de dados. Neste contexto, a computação paralela surgiu como uma opção de resolver de forma consciente os problemas da computação intensiva em diversas áreas do conhecimento humano. Na área de processamento de imagens médicas tridimensionais, diversos métodos exigem altas cargas de trabalho (grande quantidade de dados e cálculos complexos), acarretando algoritmos computacionalmente caros em relação ao tempo de execução, muitas vezes não satisfazendo a exigência de tempo em ambientes reais, como, por exemplo, em trabalhos de pesquisa e até em serviços de saúde para a prática clínica.

Com o propósito de ampliar o conhecimento sobre a problemática da computação intensiva presente em inúmeros métodos de processamento de imagens tridimensionais, foi estudada e analisada, como uma possível solução, a utilização da tecnologia OPENCL, aplicando a computação paralela em dispositivos GPUs, adicionalmente ao OPENCL foi adotado do *software* 3D SLICER para auxiliar no contexto de imagens tridimensionais. Para a construção do componente mediador, ambas as tecnologias abstraem uma grande parte da complexidade de implementação, tanto em relação à computação paralela e *hardwares*, como toda a parte de processamento, gerenciamento e visualização de imagens tridimensionais. Em relação a estes aspectos seria inviável para a pesquisa, caso fosse necessário criar tudo do zero.

Foi discutido que componente mediador, embora a programação GPU tenha sido significativamente simplificada com o surgimento de novas tecnologias, como o CUDA e OPENCL, ainda sim é exigida uma profunda compreensão (tanto de *software* como de *hardware*) para a prática da computação paralela em GPU, tornando as vezes indescritível para muitos usuários finais. Por isso o objetivo da construção do componente é minimizar essa complexidade para a criação de aplicações pelo desenvolvedor. Também foram apresentados alguns aspectos gerais de contribuição da utilização do componente: (1)Desempenho, (2)Comunicação com *hardwares*, (3)Reutilização e (4)Científico.

Para a validação do trabalho foram escolhidos e implementados alguns métodos de processamento de imagens com objetivo de confrontar a implementação do algoritmo modificado com o sequencial. O experimento 1 (introdutório) foi realizado com objetivo de adquirir conhecimento e mostrar a capacidade do componente mediador. Mesmo sendo um método simples, evidenciou-se uma média de 10x de ganho de tempo. O experimento 2 foi com o filtro de média móvel, que é relativamente simples e não exige um custo computacional muito alto, porem, por se tratar de imagens tridimensionais, a quantidade de dados aumenta consideravelmente exigindo uma alta carga de processamento. Como resultado o método modificado evidenciou uma média de ganho de 10x mais rápido do que o método sequencial. Esta média foi observada nos dois testes realizados dentro do experimento. No experimento 3, o filtro morfológico de dilatação/erosão dilatação obteve o maior valor de ganho dentre os métodos escolhidos. No primeiro teste houve um ganho de até 40x, e no segundo teste chegando até 50x comparado ao método sequencial. No último experimento, foi escolhido o método com filtro de difusão anisotrópico, um algoritmo mais robusto e complexo do que os demais. Esperava-se um ganho bem mais expressivo do que os

experimentos anteriores, porém teve um ganho de apenas 4x. Isto foi analisado e discutido, identificando o método como um problema de dependência de dados.

Uma das contribuições deste estudo consistiu em mostrar a capacidade de computação paralela em um dispositivo GPU quando comparado a uma execução sequencial em uma CPU. É perceptível, principalmente na área de pesquisa da computação, que trabalhos cada vez mais estão demandando uma alta quantidade de processamento para atingir certos resultados e/ou objetivos. Com o processamento sequencial na CPU, as vezes o algoritmo pode se tornar inviável pela demora do seu tempo de execução, e mais ainda, às vezes é necessário executá-lo diversas inúmeras vezes. Como solução é necessário investir em infraestruturas como supercomputadores e clusters, os quais são caros e normalmente laboratórios ou instituições não possuem recursos para aquisição desses equipamentos.

A computação paralela em GPUs surge como uma solução de baixo custo financeiro, visto que, hoje em dia, as unidades de processamento gráfico (GPUs) tornaram-se uma plataforma de computação paralela acessível a uma vasta gama de usuários. Dada as fortes características de alta largura de banda de memória e capacidade de computação (FLOPS), há um forte incentivo para usar GPUs para computação de propósito geral (GPGPU) e existem muitos relatos de sucesso na literatura sobre esses esforços (Kong et al.; 2010).

8.3 Limitações do Trabalho

Toda a análise da solução do componente, a escolha das tecnologias e codificação do mesmo foram feitas pensando em uma solução multiplataformas. Porém no presente trabalho não houve testes de funcionamento e comportamentais do componente, nos seguintes contextos multiplataformas:

- **Sistema Operacional:** o componente foi exclusivamente utilizado em ambiente *Linux*, não sendo utilizado por exemplo em ambientes *Windows* e *MacOS*;
- **Hardware:** o componente foi exclusivamente utilizado com dois tipos de dispositivos CPU e GPU, respectivamente dos fabricantes INTEL e AMD, cujas configurações e capacidade de processamento foram detalhadas no capítulo 7. Não houve a utilização de outros dispositivos como por exemplo FPGA (*Field Programmable Gate Array*), ou de diferentes fabricantes como por exemplo NIVIDIA, ou com poder de processamento mais elevados. No estudo

experimental, devido à limitação de acesso a esses recursos de dispositivos de *hardware* com maiores capacidades de computação (*cores* e *threads*), neste contexto, os métodos modificados para a computação paralela poderiam ter evidenciado ganhos muito mais expressivos em relação aos métodos sequenciais;

- **Software:** a ideia inicial era a proposta de construção do componente utilizando o padrão *Loadable* do 3D SLICER, devido à liberdade adquirida na aplicação (*host*). Porém por falta de documentação, quase inexistente, se dificultou a aprendizagem e assim decidiu-se pelo padrão CLI do 3D SLICER.

Considerando os desafios da proposta do projeto e as limitações discutidos anteriormente, a construção do componente mediador para a computação GPU foi alcançada e aplicada no contexto de imagens tridimensionais com sucesso, visto que existem poucos trabalhos na literatura para este contexto. E por fim a validação também conseguiu convergir para resultados muito bons, reais e positivos.

8.4 Trabalhos futuros

Como trabalhos futuros, pretende-se colocar em uso prático em um ambiente real, no Laboratório de Computação em Sinais e Imagens Médicas da USP Ribeirão Preto. Ainda se pretende estudar e modificar algoritmos mais complexos na área de processamento de imagens, assim como investigar e trabalhar com configurações diferentes de *hardware*.

O trabalho serviu de base para o entendimento e domínio do processo de desenvolvimento de módulos no 3D SLICER junto com o OPENCL, tendo como ponto de partida inicial a proposta de construção de um componente mediador que facilite a usuários pesquisadores implementarem seus próprios métodos de processamento de imagens tridimensionais modificados para a prática da computação GPU.

8.5 Publicação

Um artigo foi elaborado e submetido para a Revista Brasileira de Computação Aplicada. O texto sintetiza e analisa um dos experimentos realizados utilizando o componente mediador: Duarte, A. F.; Junior, L. O. M. Computação paralela de imagens médicas no 3D SLICER.

REFERÊNCIAS

- BARDOSI, Z. **OPENCL accelerated GPU binary morphology image filters for ITK**. p. 3–5, 2015.
- BERNABÉ, G.; GUERRERO, G. D.; FERNÁNDEZ, J. **CUDA and OPENCL implementations of 3D Fast Wavelet Transform**. 2012 IEEE 3rd Latin American Symposium on Circuits and Systems, LASCAS 2012 - Conference Proceedings, 2012.
- COHEN, J.; GARLAND, M. **Solving computational problems with GPU computing**. Computing in Science and Engineering 2009;11(5):58–63.
- CHEN, C., Li, K., OUYANG, A., Li, K. (2018). **FlinkCL: An OpenCL-based In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data**. IEEE Transactions on Computers.
- CZAJKOWSKI, T. S. et al. **From opengl to high-performance hardware on FPGAs** - 22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, 2012, pp. 531-534.
- DHAWAN, P. **Medical Image Analysis**. Second edition. New Jersey: Clearance Center, 2011. p. 2-22.
- DU, P. (2012). **From CUDA to OPENCL: Towards a performance-portable solution for multi-platform GPU programming**. Parallel Computing. 38 (8), p391-407.
- EGGER, J. et al. **GBM volumetry using the 3D SLICER medical image computing platform**. Sci Rep, 3, 1364, 2013.
- EGGER, J. et al. **Pituitary adenoma volumetry with 3D SLICER**. PloS one, 7(12), e51788, 2012.
- EIJKHOUT, V. **Introduction to High Performance Scientific Computing**. [S.l.: s.n.] p. 446, 2011.
- EKLUND, A. et al. **Medical image processing on the GPU – Past, present and future**. Medical Image Analysis, v. 17, n. 8, p. 1073–1094, 2013.
- FANG, L. et al. **Research on internationalization of 3D slicer**. Proceedings - 2013 International Conference on Computer Sciences and Applications, CSA 2013, p. 469–473, 2013

- FANG, J. (2011). **A comprehensive performance comparison of CUDA and OPENCL**. IEEE International Conference on. Parallel Processing (ICPP). P216-225.
- FEDOROV, A. et al. **3D SLICER as an image computing platform for the Quantitative Imaging Network**. Magn Reson Imaging, 30(9), 1323-41, 2012a.
- GERHARDT, T.E., SILVEIRA, D.T. **Métodos de Pesquisa**. Porto Alegre: Editora UFRGS, 2009.
- HANSEN, M. S., & SØRENSEN, T. S. (2013). **Gadgetron: an open source framework for medical image reconstruction**. Magnetic resonance in medicine, 69(6), 1768-1776.
- INTRODUCTION TO OPENCL**, presentation from OPENCL workshop at OzViz 2010 held in Brisbane, Australia. Disponível em <https://pt.slideshare.net/TomaszBednarz1/introduction-to-opencl-2010>
- JIMÉNEZ, J., MIRAS, J.R.D., 2012. **Three-dimensional thinning algorithms on graphics processing units and multicore CPUs**. Concurr. Comput.: Practive Experience 24, 1551–1571. <http://dx.doi.org/10.1002/cpe>.
- KARIMI, K. (2010). **A performance comparison of CUDA and OPENCL**. arXiv preprint arXiv:1005.2581.
- KIKINIS, R. PIEPER, S. **3D SLICER as a tool for interactive brain tumor segmentation**. Conf Proc IEEE Eng Med Biol Soc, 2011, 6982-4, 2011.
- KONG, J. et al. **Accelerating MATLAB Image Processing Toolbox functions on GPUs**. GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, p. 75–85, 2010.
- LARSEN, E. S.; MCALLISTER, D. **Fast Matrix Multiplies using Graphics Hardware**. SC '01 Proceedings of the 2001 ACM/IEEE conference on Supercomputing, p. 55, 2001.
- LIAO, L., LI, K., LI, K., YANG, C., TIAN, Q. (2018, August). **UHCL-Darknet: An OpenCL-based Deep Neural Network Framework for Heterogeneous Multi-/Many-core Clusters**. In Proceedings of the 47th International Conference on Parallel Processing (p. 44). ACM.
- MATSUMOTO, K.; NAKASATO, N.; SEDUKHIN, S. G. **Performance Tuning of Matrix Multiplication in OpenCL on Different GPUs and CPUs**. 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, 2012, pp. 396-405.
- MUNSHI, A., GASTER, B. et al. **OPENCL Programming Guide**, 1st ed. Addison-Wesley Professional, 2011.
- NVIDIA CUDA C Programming Guide**. NVIDIA Corporation, ed. 5.0.: NVIDIA Corporation, 2013.
- OLIVEIRA D. D., DANILO H. P. L., (2015). **Fast 2D and 3D image processing with OPENCL**. In IEEE International Conference on Image Processing (ICIP) (pp. 4858–4862).

OPENCL. **The open standard for parallel programming of heterogeneous systems.** Disponível em: <<http://www.khronos.org/OPENCL/2009>>.

PAULA, L. C. M. et al. **Aplicação de Processamento Paralelo em método iterativo para solução de sistemas lineares.** In: Encontro Anual de Computação, X., 2013a, Catalão. Anais...Catalão: UFG. p. 129-136, 2013a.

PAULA, L. C. M. et al. **Partial Parallelization of the Successive Projections Algorithm using Compute Unified Device Architecture.** In: International Conference on Parallel and Distributed Processing Techniques and Applications. 19th. 2013b, [S.l.]. Proceedings... [S.l.: s.n.]. p. 737-741, 2013b.

PAULA, L. C. M. **Paralelização e comparação de métodos iterativos na solução de sistemas lineares grandes e esparsos.** ForScience: revista científica do IFMG, v. 1, n. 1, p. 01-12, 2013c.

PAULA, L. C. M. **Programação Paralela de um Método Iterativo para Solução de Grandes Sistemas de Equações Lineares usando a Integração CUDA-Matlab.** Revista de Sistemas e Computação (Aceito para publicação), v. 4, n. 1, 2014a.

PAULA, L. C. M. et al. **Parallelization of a Modified Firefly Algorithm using GPU for Variable Selection in a Multivariate Calibration Problem.** International Journal of Natural Computing Research, v. 4, n. 1, p. 31-42, 2014b.

PAULA, L. C. M. de. (2014). **CUDA vs. OPENCL: uma comparação teórica e tecnológica.** ForScience, 2(1), 31-46. Retrieved from <http://formiga.ifmg.edu.br/forscience/index.php/forscience/article/view/53>

PERONA, P., MALIK, J. (1990). **Scale-space and edge detection using anisotropic diffusion.** IEEE Transactions on pattern analysis and machine intelligence, 12(7), 629-639.

PIEPER, S., Lorensen, B., Schroeder, .W. et al. **The NA-MIC Kit: ITK, VTK, pipelines, grids and 3D slicer as an open platform for the Medical Image Computing community.** 2006 3rd Ieee International Symposium On Biomedical Imaging: Macro To Nano, Vols 1-3 Book Series: IEEE International Symposium On Biomedical Imaging, 698-701, 2006.

PIEPER, S., Halle, M. Kikinis, R. **3D SLICER. Biomedical Imaging: Nano to Macro,** 2004. IEEE International Symposium on, 632, 2004.

PRESSMAN, ROGER S. Engenharia de Software. 5ª Ed. Rio de Janeiro: McGraw- Hill, 2002.

REGAN, C., HAYAKAWA, C., CHOI, B. (2017). **Momentum transfer Monte Carlo for the simulation of laser speckle imaging and its application in the skin.** Biomedical optics express, 8(12), 5708-5723.

SAMETINGER, Johannes. **Software Engineering with Reusable Components.** New York: Springer, 1997. 271p.

SCARPINO, M. **OPENCL in Action: How to Accelerate Graphics and Computation.** Manning Publications Co. NY. 2012.

- SHI, G.; KINDRATENKO, V.; PRATAS, F.; TRANCOSO, P.; GSCHWIND, M. **Application acceleration with the Cell broadband engine**. Computing in Science and Engineering 2010;12(1):76–81.
- SIMMROSS-WATTENBERG, F., RODRÍGUEZ-CAYETANO, M., ROYUELA-DEL-VAL, J., MARTÍN-GONZÁLEZ, E., MOYA-SÁEZ, E., MARTÍN-FERNÁNDEZ, M., & ALBEROLA-LÓPEZ, C. (2018). **OpenCLIPER: an OpenCL-based C++ Framework for Overhead-Reduced Medical Image Processing and Reconstruction on Heterogeneous Devices**. arXiv preprint arXiv:1807.11830.
- SMISTAD, E. et al. **Medical image segmentation on GPUs – A comprehensive review**. Medical Image Analysis, v. 20, n. 1, p. 1–18, 2015.
- SMITH, S. M. **The GPU Computing Revolution**. [S.l.: s.n.] 2011.
- TAMIR, J. I., ONG, F., CHENG, J. Y., UECKER, M., LUSTIG, M. (2016, May). **Generalized magnetic resonance image reconstruction using the Berkeley Advanced Reconstruction Toolbox**. In ISMRM Workshop on Data Sampling & Image Reconstruction, Sedona, AZ.
- TOKUDA, J. et al. **Graphics Processing Unit–Accelerated Nonrigid Registration of MR Images to CT Images During CT-Guided Percutaneous Liver Tumor Ablations**. Academic Radiology, v. 22, n. 6, p. 722–733, 2015.
- TORIWAKI J., Yoshida H. **Fundamentals of Three-dimensional Digital Image Processing**. Springer Science & Business Media, 2009.
- UECKER, M., ONG, F., TAMIR, J. I., BAHRI, D., VIRTUE, P., CHENG, J. Y., LUSTIG, M. (2015). **Berkeley advanced reconstruction toolbox**. In Proc. Intl. Soc. Mag. Reson. Med (Vol. 23, p. 2486).
- VIDAL-MIGALLÓN, I. et al. **GPU & CPU implementation of Young - Van Vliet's Recursive Gaussian Smoothing Filter**. Insight Journal (ITK), p. 16, 2013.
- WANG, D., XU, K., JIANG, D. (2017, December). **PipeCNN: an OpenCL-based open-source FPGA accelerator for convolution neural networks**. In Field Programmable Technology (ICFPT), 2017 International Conference on (pp. 279-282). IEEE.
- WAZLAWICK, R. S.: **Metodologia de Pesquisa para Ciência da Computação**. Elsevier, 2014.
- WEBER R., GOTHANDARAMAN A., HINDE R.J., PETERSON G.D., **Comparing hardware accelerators in scientific applications: a case study**. IEEE Transactions on Parallel and Distributed Systems 99 (RapidPosts). <<http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.125>>.
- XUE, H., INATI, S., SØRENSEN, T. S., KELLMAN, P., & HANSEN, M. S. (2015). **Distributed MRI reconstruction using gadgetron-based cloud computing**. Magnetic resonance in medicine, 73(3), 1015-1025.

YUAN, Y., SALMI, M. F., HUAI, Y., WANG, K., LEE, R., ZHANG, X. (2016, December). **Spark-GPU: An accelerated in-memory data processing engine on clusters. In Big Data (Big Data)**, 2016 IEEE International Conference on (pp. 273-283). IEEE.

ZHANG, N.; WANG, J. L.; CHEN, Y. S. **Image parallel processing based on GPU Proceedings** - 2nd IEEE International Conference on Advanced Computer Control, ICACC 2010. Anais...Ieee, 2010.Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5486836>

APÊNDICES

Apêndice A

A seguir é apresentado toda a documentação de caráter mais técnico elaborada para o componente mediador proposto pelo projeto.

Especificação dos Requisitos do Software

Componente mediador para computação *GPU* de imagens médicas no *3D SLICER 1.0*

Autor: Alexandre Freitas Duarte

INTRODUÇÃO

Objetivos do documento

Este documento tem como objetivo, a especificação dos requisitos contemplados pelo componente mediador para computação GPU de imagens médicas no *3D SLICER*. Com intuito de fornecer todas as informações necessárias para o projeto, a codificação, testes e validação. Os pesquisadores/desenvolvedores do *3D SLICER* são o principal público alvo deste documento.

Escopo do produto

Missão do *software*

Dada a necessidade da computação intensiva de muitos algoritmos de processamento de imagens médicas tridimensionais, cujo estes demandam um alto custo computacional, o componente tem como missão principal proporcionar o uso da computação paralela em GPUs dentro do *3D SLICER* e que seja compatível com diferentes plataformas (*hardwares*). Com este componente de software, espera-se possibilitar que processamentos complexos em imagens médicas tridimensionais ganhem um notável ganho nos seus tempos de execução.

Benefícios do *software*

1. Ganho no tempo de execução em processamentos complexos de imagens médicas tridimensionais (Essencial)
2. Permitir a computação paralela em diferentes plataformas (Essencial)
3. Permitir por meio de classes e métodos, trazer um auxílio e facilidade para a computação paralela (Desejável)
4. Permitir diferentes tipos de execuções, e seus respectivos tempos de execução (Essencial)

Materiais de referência

IEEE Std. 830 – 1993. IEEE Recommended Practice for Software Requirements Specifications.

Definições e siglas

Descreve-se aqui a definição de todas as siglas, abreviações e termos usados no documento.

CPU	<i>Central Processing Unit</i>
GPU	<i>Graphics Processing Unit</i>

ITK	<i>Insight Segmentation and Registration Toolkit</i>
OPENCL	<i>Open Computing Language</i>
VTK	<i>Visualization Toolkit</i>

Visão geral deste documento

De acordo com o Padrão para Especificação de Requisitos IEE, o documento está organizado, como se segue:

- Parte 2: Descrição geral do produto
- Parte 3: Requisitos específicos
- Parte 4: Informações de suporte

DESCRIÇÃO GERAL DO PRODUTO

Perspectiva do produto

Diagrama de atividade

A seguir é ilustrado um diagrama de atividades do componente, o qual tem como finalidade demonstrar graficamente o comportamento e atividades do componente.

Diagrama de classe de domínio

A seguir é ilustrado um diagrama de classe de domínio, o qual tem como finalidade a representação da estrutura e relações das classes que servem de modelo para objetos que o componente possuirá.

Público alvo

O principal usuário do componente serão os pesquisadores/desenvolvedores do *3D SLICER*, que trabalham com diversas técnicas inovadoras de processamento de imagens médicas, e por se tratar de imagens tridimensionais, muitas vezes tais aplicações demandam um alto custo computacional (em relação ao tempo de execução). Neste contexto eles poderiam usar o componente para fins de obtenção de resultados melhores no ganho do tempo de execução, e podendo realizar análises, comparando métodos tradicionais cujo o processamento é realizado de forma sequencial na CPU com métodos os modificados à serem executados de forma paralela nos núcleos disponíveis de uma GPU.

REQUISITOS ESPECÍFICOS

Requisitos funcionais

RF00. No modulo de interface gráfica, antes de tudo deve-se verificar se há alguma plataforma do *OPENCL* instalada no computador do usuário, caso necessite mensagem de aviso será lançada.

RF01. No modulo de interface gráfica, deve ser possível do usuário abrir uma imagem (2D ou 3D) armazenada no seu computador, esta imagem será carregada e visualizada.

- Por meio de um botão “Abrir Imagem”, o usuário procurará o caminho da imagem desejada a ser carregada e visualizada, sendo esse o passo inicial da utilização do componente.
- Verifica-se se a imagem está em conformidade com os formatos aceitos para se realizar a computação paralela no *OPENCL*. Caso necessite mensagem de aviso será lançada.

RF02. No modulo de interface gráfica, deve ser possível do usuário escolher com qual dispositivo (GPU ou CPU) ele deseja trabalhar.

- Por meio de uma lista do tipo ‘*Select Box*’, automaticamente deve estar listado os nomes dos dispositivos *OPENCL* compatíveis encontrados no computador do usuário, assim o usuário pode escolher em qual queira trabalhar, sendo este o segundo passo.
- Ao selecionar o dispositivo, é interessante mostrar logo abaixo, informações relevantes sobre o mesmo, como nome completo, fabricante, número de ‘*OpenCl cores*’, que serão os núcleos disponíveis para executar as tarefas, e quantidade máxima de *work-items* que pode ser executada por vez por um único ‘*OpenCl core*’ ou núcleo.

RF03. No modulo de interface gráfica, deve ser possível do usuário escolher com qual método de processamento de imagem ele deseja trabalhar.

- Por meio de uma lista do tipo ‘*Select Box*’, automaticamente deve estar listado os nomes dos ‘*kernels*’ já implementados encontrados no componente, assim o usuário pode escolher em qual método de processamento de imagens quer trabalhar e aplicar em sua imagem já carregada, sendo este o terceiro passo.

RF04. No modulo de interface gráfica, deve ser possível do usuário entrar com parâmetros de configuração relativo tanto com o método escolhido, tal como parâmetro de configuração do *OPENCL*.

- Normalmente métodos computacionais de processamento de imagens requerem parâmetros de entrada, quando o usuário selecionar a método desejado (RF03) caso esta tenha parâmetros de entrada, para cada parâmetro deve-se criar um campo (elemento gráfico) para que o usuário possa fazer a entrada e configure para obter resultado desejado na imagem de saída.
- Alguns parâmetros *OPENCL* também deve-se estar disponível para o usuário modificar. Um campo para a quantidade de ‘*work item*’ que será executada por vez por um único núcleo, este campo depende do dispositivo, então alguns

dispositivos permitirão apenas 256, 512 ou 1024, por exemplo. Caso o dispositivo permita apenas 256 ‘*work-items*’ por vez de execução em 1 único núcleo, o usuário não será possível de inserir como entrada um número maior que este.

RF04. No modulo de interface gráfica, na interação com o usuário, este ao preencher todos os campos, opções e a validação estiver concluída, um botão de “Executar” aparecerá disponível, para que dê prosseguimento as próximas etapas que será o envio dos parâmetros escolhidos junto com a imagem carregada.

- Deve ser possível o envio adequado das informações (parâmetro e imagem) para o ‘host’.
- Deve ser possível o envio da imagem, encapsulada em uma estrutura de dados adequada para o manuseio no ‘host’.

RF04. No modulo de interface gráfica, ao completar todo o processo de execução no ‘host’ e no dispositivo, o resultado (a imagem) deve-se voltar para a interface gráfica para ser mostrada ao usuário como resultado.

- O resultado da imagem deve ser enviado pelo ‘host’ a interface gráfica no mesmo formato que foi enviado, para que seja convertido em imagem e visualizado pelo usuário.

Diagrama de caso de uso

Para um melhor entendimento dos atores e suas relações com as funcionalidades necessárias ao produto, casos de uso foram ilustrados a seguir

Modo de execução - sequencial

Modo de execução - paralela

Requisitos não-funcionais

RF01. A interface de interação do usuário, deve ser implementada como um modulo do software *3D SLICER*, que são plugins em C++. Uma *GUI* personalizada e especifica será definida.

RF02. Para flexibilizar a comunicação com o usuário, algum mecanismo de mensagens de avisos deve ser implementado, em decorrência de erros e avisos. Caso seja necessário, mensagens pré-definidas de avisos serão lançadas para os usuários, avisando-os de ações que contrariam as políticas do componente.

MANUAL DE SUPORTE

Propósito do documento

O objetivo deste documento, é apresentar algumas recomendações, instalação e requisitos obrigatórios para o funcionamento do componente mediador para computação *GPU* de imagens médicas no *3D SLICER*. Fica de responsabilidade do usuário (pesquisador/desenvolvedor) dispor da integralidade dos mesmos, que serão listados a seguir.

Especificações técnicas do software

Todo o componente mediador foi codificado, utilizando a linguagem C++. Para a prática da computação paralela, foi adotado a tecnologia *OPENCL* e utilizando o SDK disponibilizado pela AMD, escrito em C++. Para a incorporação do componente em módulos do *3D SLICER*, foi utilizado o padrão *CLI Modules* para desenvolvimento de módulos. Para auxílio no gerenciamento e importação das dependências ao projeto, foi utilizado o *CMAKE*.

Requisitos mínimos de hardware

Como requisitos de hardware, é necessário ter disponível um ou mais dispositivos de hardware (e.g. *CPU*, *GPU*, *FPGA*), este podendo ser de quaisquer fabricantes, contudo, que sejam compatíveis ao *OPENCL*. A tecnologia *OPENCL* é mantida por um grupo chamado *Khronos Group* e a lista de todos as companhias participantes pode ser encontrada no link <https://www.khronos.org/conformance/adopters/conformant-companies>. Para cada hardware é necessário a instalação e configuração dos drivers corretos para os dispositivos de hardwares de acordo com cada fabricante.

Requisitos mínimos de software

Como requisitos de software serão necessários:

1. Ambiente de desenvolvimento para aplicações C++, assim como compilador da mesma. Todas as tecnologias usadas no projeto são desenvolvidas em C++, seguem as mais importantes: *3D SLICER*, *Insight Segmentation and Registration Toolkit* (ITK), *The Visualization ToolKit* (VTK) e QT;
2. A proposta do componente mediador, é que este seja implementado como uma extensão do software *3D SLICER*, logo então é exigido a instalação e configuração do mesmo (versão para desenvolvedores) no computador. Para maiores informações sobre a instalação, pode-se encontrar em https://www.slicer.org/wiki/Slicer_3.0_User_Information;
3. Por fim para desfrutar de toda abstração proporcionada pelo *OPENCL* para a prática da computação paralela em dispositivos de hardware, ainda é exigido a instalação de um SDK (*Software Development Kit*). Implementações do *OPENCL* são fornecidos por qualquer fabricante de hardware compatível. Basta a instalação de um SDK para permitir a comunicação e envio de tarefas à serem executadas de forma paralela em dispositivos de hardware.

Limitações técnicas do software

1. O componente não automatizará o processo totalmente, sendo ainda necessário a intervenção do pesquisador/desenvolvedor, para adequar seu ambiente de acordo com os requisitos de hardware e software apresentados anteriormente.
2. Não houve testes de funcionamento do componente em sistemas operacionais *Windows* e *MacOS*, apenas para *Linux*.
3. Não houve testes de funcionamento do componente com dispositivos diferentes de CPU e GPU, respectivamente dos fabricantes Intel e AMD
4. Não houve testes de funcionamento do componente em sistemas operacionais *Windows* e *MacOS*, apenas para *Linux*.
5. Para a sua utilização será necessário um conhecimento médio-avançado pelo pesquisador/desenvolver, das tecnologias: *3D SLICER*, *OPENCL*, C++, CMAKE, VTK, ITK, QT.

Recomendações

Algumas recomendações são sugeridas, tais como:

1. Como sistema operacional, utilizar o *Linux*.
2. Como IDE de desenvolvimento, instalar o framework *Qt Creator*
3. Utilização do SDK disponibilizado pela AMD para codificação de aplicações *OPENCL*, disponível no link <https://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/> ;
4. Apesar do framework *OPENCL* proporcionar uma abstração de desenvolvimento para a prática da computação paralela em dispositivos, possibilitando o desenvolvimento na linguagem C/C++. Ainda assim é exigido do usuário, no caso o desenvolvedor, um conhecimento nível Avançado de *OPENCL*, para que ele consiga desfrutar e utilizar do componente mediador, como referências de livros temos o [1] e [2].

5. Para a sua utilização será necessário um conhecimento médio-avançado pelo pesquisador/desenvolver, das tecnologias: *3D SLICER*, *OPENCL*, C++, *CMAKE*, *VTK*, *ITK*, *QT*.

Descritivo técnico do componente

Pode-se dizer que o componente construído é dividido em três partes:

1. Configuração: toda aplicação em C++ precisa ser compilada (*build*), e quando existe um grande número de bibliotecas a serem importadas no projeto (dependências), o comando de compilação começa a ficar complexo e desorganizado. Como no caso da utilização da biblioteca *OPENCL* para o componente mediador incorporado no *3D SLICER*, este último, também possui muitas dependências (*VTK*, *ITK*, *QT* entre outras). Para isso foi utilizado o *CMAKE*, um software multiplataforma, open-source, para gerenciar todas as dependências existentes nas aplicações e módulos criados. Com isso foi necessário a codificação de um arquivo de configuração, na linguagem *CMAKE*, contendo toda a inteligência e gerenciamento de bibliotecas e até tratamentos para diferentes tipos de sistema operacionais (*Linux*, *MacOS* e *Windows*), pois para cada sistema operacional o formato e localização das bibliotecas são diferente.
2. Aplicação Host: aplicação escrita em C++, parte principal do componente que realiza algumas funções essenciais como: integração e comunicação com todas as bibliotecas necessárias, responsável pela comunicação com o dispositivo de hardware, configuração e compilação do kernel. Foi codificado um pacote de funções específicas para abstração da complexidade proveniente da computação paralela usando o *OPENCL* e do gerenciamento de imagens tridimensionais proveniente do *3D SLICER*, e da integração entre ambas.
3. Kernel: aplicação escrita em uma linguagem baseada em C99, o *kernel* é a rotina/método a ser executado de forma paralela no dispositivo. Algumas funções auxiliares forem criadas para auxiliar a manipulação da imagem dentro do *kernel*, visto que não existe estrutura de dados do tipo matrizes, apenas vetores unidimensionais.

Versionamento de código

Todo o código do componente, módulos do *3D SLICER*, assim como outras aplicações e testes realizados durante o projeto, estarão hospedados e disponíveis no repositório

GITHUB do Laboratório de Computação em Sinais e Imagens Médicas (CSIM), acessível em <https://github.com/CSIM-Toolkits/opencl>

Referências

- [1] MUNSHI, A., GASTER, B. et al. OPENCL Programming Guide, 1st ed. Addison-Wesley Professional, 2011.
- [2] SCARPINO, M. Opencl in Action: How to Accelerate Graphics and Computation. Manning Publications Co. NY. 2012.

Apêndice B

Na Figura B.1 e B.2 estão representados os trechos do código que implementam o filtro de média móvel para imagens tridimensionais. Na Figura B.1 o código do filtro de média móvel que será executado de forma sequencial na CPU, e na Figura B.2 o *Kernel* com o código que será executado de forma paralela em uma GPU.

```
__kernel void MeanFilter(const __global float* in,__global float* out,int radiusx,
int radiusy, int radiusz,int width, int height, int depth)
{
    int gix = (int)get_global_id(0);
    int giy = (int)get_global_id(1);
    int giz = (int)get_global_id(2);

    unsigned int gidx = width*(giz*height + giy) + gix;

    float sum = 0;
    unsigned int num = 0;

    bool isValid = true;
    if(gix >= width) isValid = false;
    if(giy >= height) isValid = false;
    if(giz >= depth) isValid = false;

    if( isValid )
    {
        for(int z = max(0, (int)(giz-radiusz)); z <= min((int)(depth-1), (int)(giz+radiusz)); z++)
        {
            for(int y = max((int)0, (int)(giy-radiussy)); y <= min((int)(height-1), (int)(giy+radiusy)); y++)
            {
                for(int x = max((int)0, (int)(gix-radiusx)); x <= min((int)(width-1), (int)(gix+radiusx)); x++)
                {
                    unsigned int cidx = width*(z*height + y) + x;

                    sum += (float)in[cidx];
                    num++;
                }
            }
        }

        out[gidx] = (float)(sum/(float)num);
    }
}
```

Figura B.1 – Código (*Kernel*) de filtro média móvel. (Fonte: elaborada pelo autor)

```

for(int gix=0; gix<width; gix++){
    for(int giy=0; giy<height; giy++){
        for(int giz=0; giz<deep; giz++){

            unsigned int gidx = width*(giz*height + giy) + gix;

            float sum = 0;
            unsigned int num = 0;

            bool isValid = true;
            if(gix >= width) isValid = false;
            if(giy >= height) isValid = false;
            if(giz >= deep) isValid = false;

            if( isValid )
            {
                num = (2*radiusx + 1)*(2*radiusy + 1)*(2*radiusz + 1);
                for(int z = giz-radiusz; z <= giz+radiusz; z++)
                {
                    unsigned int zid = (unsigned int) (min(max(0, z), (int) (deep-1)));
                    for(int y = giy-radiusy; y <= giy+radiusy; y++)
                    {
                        unsigned int yid = (unsigned int) (min(max(0, y), (int) (height-1)));
                        for(int x = gix-radiusx; x <= gix+radiusx; x++)
                        {
                            unsigned int cidx = width*(zid*height + yid) +
                                (unsigned int) (min(max(0, x), (int) (width-1)));

                            sum += (float)pixelData[cidx];
                        }
                    }
                }

                pixelDataOut[gidx] = (float) (sum/(float)num);
            }
        }
    }
}

```

Figura B.2 – Código (Sequencial) de filtro média móvel. (Fonte: elaborada pelo autor)

Apêndice C

Na Figura C.1 e C.2 estão representados os trechos do código que implementam o filtro de dilatação para imagens tridimensionais. Na Figura C.1 o código do filtro de dilatação que será executado de forma sequencial na CPU, e na Figura C.2 o *Kernel* com o código que será executado de forma paralela em uma GPU.

```
__kernel void BinaryDilateFilter(const __global float* in, __global float* out, __constant float* op,
const int radius, int sesum, const int fgValue, const int bgValue, const int borderFg)
{
    int gix = get_global_id(0);
    int giy = get_global_id(1);
    int giz = get_global_id(2);

    unsigned int gidx = width*(giz*height + giy) + gix;
    unsigned int opIdx = (2*radius + 1)*(2*radius + 1)*(2*radius + 1) - 1;

    bool isValid = true;
    if(gix < 0 || gix >= width) isValid = false;
    if(giy < 0 || giy >= height) isValid = false;
    if(giz < 0 || giz >= deep) isValid = false;

    float result = in[gidx];
    bool finished = false;

    if( isValid )
    {
        for(int z = -radius; z <= radius; z++)
        {
            for(int y = -radius; y <= radius; y++)
            {
                for(int x = -radius; x <= radius; x++)
                {
                    if(op[opIdx] == 1){
                        int tx = gix-x;
                        if(tx < 0 || tx >= width){
                            if(borderFg){ result = fgValue; finished = true; break; }
                            --opIdx; continue; }

                        int ty = giy-y;
                        if(ty < 0 || ty >= height){
                            if(borderFg){ result = fgValue; finished = true; break; }
                            --opIdx; continue; }

                        int tz = giz-z;
                        if(tz < 0 || tz >= deep){
                            if(borderFg){ result = fgValue; finished = true; break; }
                            --opIdx; continue; }

                        unsigned int tidx = width*height*tz + width*ty + tx;
                        if(in[tidx] == fgValue){
                            result = fgValue;
                            finished = true;
                            break;
                        }
                    }
                    --opIdx;
                }
                if(finished) break;
            }
            if(finished) break;
        }

        out[gidx] = result;
    }
}
```

Figura C.1 – Código (*Kernel*) de filtro de dilatação. (Fonte: elaborada pelo autor)

```

for(int giz=0; giz<deep; giz++){
    for(int giy=0; giy<height; giy++){
        for(int gix=0; gix<width; gix++){

            unsigned int gidx = width*(giz*height + giy) + gix;
            unsigned int opIdx = (2*radius + 1)*(2*radius + 1)*(2*radius + 1) - 1;

            bool isValid = true;
            if(gix < 0 || gix >= width) isValid = false;
            if(giy < 0 || giy >= height) isValid = false;
            if(giz < 0 || giz >= deep) isValid = false;

            float result = pixelData[gidx];
            bool finished = false;

            if( isValid )
            {
                for(int z = -radius; z <= radius; z++)
                {
                    for(int y = -radius; y <= radius; y++)
                    {
                        for(int x = -radius; x <= radius; x++)
                        {
                            if(pixelDataKernel[opIdx] == 1){
                                int tx = gix-x;
                                if(tx < 0 || tx >= width){
                                    if(borderFg){ result = fgValue; finished = true; break; }
                                    --opIdx; continue; }

                                int ty = giy-y;
                                if(ty < 0 || ty >= height){
                                    if(borderFg){ result = fgValue; finished = true; break; }
                                    --opIdx; continue; }

                                int tz = giz-z;
                                if(tz < 0 || tz >= deep){
                                    if(borderFg){ result = fgValue; finished = true; break; }
                                    --opIdx; continue; }

                                unsigned int tidx = width*height*tz + width*ty + tx;
                                if(pixelData[tidx] == fgValue){
                                    result = fgValue;
                                    finished = true;
                                    break;
                                }
                            }
                            --opIdx;
                        }
                        if(finished) break;
                    }
                    if(finished) break;
                }
                pixelDataOut[gidx] = result;
            }
        }
    }
}

```

Figura C.2 – Código (Sequencial) de filtro de dilatação. (Fonte: elaborada pelo autor)

Apêndice D

Na Figura D.1 e D.2 estão representados os trechos do código que implementam o filtro de erosão para imagens tridimensionais. Na Figura D.1 o código do filtro de erosão que será executado de forma sequencial na CPU, e na Figura D.2 o *Kernel* com o código que será executado de forma paralela em uma GPU.

```
__kernel void BinaryErodeFilter(const __global float* in, __global float* out, __constant float* op,
const int radius, const int sesum, const int fgValue, const int bgValue, const int borderFg)
{
    int gix = get_global_id(0);
    int giy = get_global_id(1);
    int giz = get_global_id(2);

    unsigned int gidx = width*(giz*height + giy) + gix;
    unsigned int sum = 0;
    unsigned int opIdx = 0;

    bool isValid = true;
    if(gix < 0 || gix >= width) isValid = false;
    if(giy < 0 || giy >= height) isValid = false;
    if(giz < 0 || giz >= deep) isValid = false;

    float result = in[gidx];
    bool finished = false;

    if( isValid )
    {
        for(int z = -radius; z <= radius; ++z)
        {
            for(int y = -radius; y <= radius; ++y)
            {
                for(int x = -radius; x <= radius; ++x)
                {
                    if(op[opIdx] == 1)
                    {
                        int tx = gix-x;
                        if(tx < 0 || tx >= width){
                            if(!borderFg){ finished = true; break; }
                            ++opIdx; ++sum; continue;
                        }

                        int ty = giy-y;
                        if(ty < 0 || ty >= height){
                            if(!borderFg){ finished = true; break; }
                            ++opIdx; ++sum; continue;
                        }

                        int tz = giz-z;
                        if(tz < 0 || tz >= deep){
                            if(!borderFg){ finished = true; break; }
                            ++opIdx; ++sum; continue;
                        }

                        unsigned int tidx = width*height*tz + width*ty + tx;

                        if(in[tidx] == fgValue){
                            ++sum;
                        }
                    }
                    opIdx++;
                }
                if(finished) break;
            }
            if(finished) break;
        }

        if(in[gidx] == fgValue && sum < sesum)
            result = bgValue;

        out[gidx] = result;
    }
}
```

Figura D.1 – Código (*Kernel*) de filtro de erosão. (Fonte: elaborada pelo autor)

```

for(int giz=0; giz<deep; giz++){
    for(int giy=0; giy<height; giy++){
        for(int gix=0; gix<width; gix++){

            unsigned int gidx = width*(giz*height + giy) + gix;
            unsigned int sum = 0;
            unsigned int opIdx = 0;

            bool isValid = true;
            if(gix < 0 || gix >= width) isValid = false;
            if(giy < 0 || giy >= height) isValid = false;
            if(giz < 0 || giz >= deep) isValid = false;

            float result = pixelData[gidx];
            bool finished = false;

            if( isValid )
            {
                for(int z = -radius; z <= radius; z++)
                {
                    for(int y = -radius; y <= radius; y++)
                    {
                        for(int x = -radius; x <= radius; x++)
                        {
                            if(pixelDataKernel[opIdx] == 1){
                                int tx = gix-x;
                                if(tx < 0 || tx >= width){
                                    if(!borderFg){ finished = true; break; }
                                    ++opIdx; ++sum; continue;
                                }

                                int ty = giy-y;
                                if(ty < 0 || ty >= height){
                                    unsigned int tidx = width*height*tz + width*ty +

                                    if(pixelData[tidx] == fgValue){
                                        ++sum;
                                    }
                                }
                                opIdx++;
                            }
                            if(finished) break;
                        }
                        if(finished) break;
                    }
                }

                if(pixelData[gidx] == fgValue && sum < m_seCount)
                    result = bgValue;

                pixelDataOut[gidx] = result;
            }
        }
    }
}

```

Figura D.2 – Código (Sequencial) de filtro de erosão. (Fonte: elaborada pelo autor)

Apêndice E

Na Figura E.1 e E.2 estão representados os trechos do código que implementam o filtro de difusão anisotrópico para imagens tridimensionais. Na Figura E.1 o código do filtro de difusão anisotrópico que será executado de forma sequencial na CPU, e na Figura E.2 o *Kernel* com o código que será executado de forma paralela em uma GPU.

```
for(int interactions=0; interactions < numIterations;interactions++){
    for(int giz=0; giz<deep; giz++){
        for(int giy=0; giy<height; giy++){
            for(int gix=0; gix<width; gix++){
                int index = width*(giz*height + giy) + gix;
                bool isValid = true;

                if(gix < 0 || gix >= width) isValid = false;
                if(giy < 0 || giy >= height) isValid = false;
                if(giz < 0 || giz >= deep) isValid = false;

                if( isValid )
                {
                    //do gradient north
                    if((index-(int)width)<=0){
                        GradNData[index] = 0.0;
                    }
                    else{
                        int auxInt = index - (int)width;
                        GradNData[index] = pixelData[auxInt] - pixelData[index];
                    }

                    //do gradient south
                    if((index+(int)width)> ((int)width*(int)height)){
                        GradSData[index] = 0;
                    }
                    else{
                        int auxInt = index + (int)width;
                        GradSData[index] = pixelData[auxInt] - pixelData[index];
                    }

                    //do gradient east and west
                    GradLData[index] = pixelData[index + 1] - pixelData[index];
                    GradOData[index] = pixelData[index - 1] - pixelData[index];

                    //do gradient northeast
                    if((index - (int)width + 1)<0){
                        GradNLData[index] = 0;
                    }
                    else{
                        int auxInt = index - (int)width + 1;
                        GradNLData[index] = pixelData[auxInt] - pixelData[index];
                    }

                    //do gradient northwest
                    if((index - (int)width - 1)<0){
                        GradNOData[index] = 0;
                    }
                    else{
                        int auxInt = index - (int)width - 1;
                        GradNOData[index] = pixelData[auxInt] - pixelData[index];
                    }

                    //do gradient southeast
                    if((index + (int)width + 1)> ((int)width*(int)height)){
                        GradSLData[index] = 0;
                    }
                    else{
                        int auxInt = index + (int)width + 1;
                        GradSLData[index] = pixelData[auxInt] - pixelData[index];
                    }

                    //do gradient south-west
                    if((index + (int)width - 1)> ((int)width*(int)height)){
                        GradSOData[index] = 0;
                    }
                    else{
                        int auxInt = index + (int)width - 1;
                        GradSOData[index] = pixelData[auxInt] - pixelData[index];
                    }

                    fPerona0Data[index] = exp(-pow((float) (GradNData[index]/kappa),2));
                    fPerona1Data[index] = exp(-pow((float) (GradNLData[index]/kappa),2));
                    fPerona2Data[index] = exp(-pow((float) (GradLData[index]/kappa),2));
                    fPerona3Data[index] = exp(-pow((float) (GradSLData[index]/kappa),2));
                    fPerona4Data[index] = exp(-pow((float) (GradSData[index]/kappa),2));
                    fPerona5Data[index] = exp(-pow((float) (GradSOData[index]/kappa),2));
                    fPerona6Data[index] = exp(-pow((float) (GradOData[index]/kappa),2));
                    fPerona7Data[index] = exp(-pow((float) (GradNOData[index]/kappa),2));

                    float diffImg[8];
                    diffImg[0] = auxEven*fPerona0Data[index]*GradNData[index];
                    diffImg[1] = auxOdd*fPerona1Data[index]*GradNLData[index];
                    diffImg[2] = auxEven*fPerona2Data[index]*GradLData[index];
                    diffImg[3] = auxOdd*fPerona3Data[index]*GradSLData[index];
                    diffImg[4] = auxEven*fPerona4Data[index]*GradSData[index];
                    diffImg[5] = auxOdd*fPerona5Data[index]*GradSOData[index];
                    diffImg[6] = auxEven*fPerona6Data[index]*GradOData[index];
                    diffImg[7] = auxOdd*fPerona7Data[index]*GradNOData[index];

                    float sum=0.0;

                    for (int k = 0; k < 8; k++) {
                        sum += diffImg[k];
                    }

                    pixelDataOut[index] = (float) (pixelData[index] + sum);
                }
            }
        }
    }
}
```

Figura E.1 – Código (Sequencial) do filtro de difusão anisotrópico. (Fonte: elaborada pelo autor)


```

__kernel void AnisotropicClassic2(const __global float* in, __global float* out,
const float kappa, const float even, const float odd)
{
    float diffImg[8];
    float fPerona[8];
    float gradN, gradS, gradL, gradO, gradNL, gradNO, gradSL, gradSO;
    int index = getCurrentIndex();

    //do gradient north
    if((index-width)<0){
        gradN = 0;
    }else{
        gradN = in[index - width] - getCurrentPixel(in);
    }

    //do gradient south
    if((index+width)> (width*height)){
        gradS = 0;
    }else{
        gradS = in[index + width] - getCurrentPixel(in);
    }

    //do gradient east and west
    gradL = in[index + 1] - getCurrentPixel(in);
    gradO = in[index - 1] - getCurrentPixel(in);

    //do gradient northeast
    if((index - width + 1)<0){
        gradNL = 0;
    }else{
        gradNL = in[index - width + 1] - getCurrentPixel(in);
    }

    //do gradient northwest
    if((index - width - 1)<0){
        gradNO = 0;
    }else{
        gradNO = in[index - width - 1] - getCurrentPixel(in);
    }

    //do gradient southeast
    if((index + width + 1)> (width*height)){
        gradSL = 0;
    }else{
        gradSL = in[index + width + 1] - getCurrentPixel(in);
    }

    //do gradient south-west
    if((index + width - 1)> (width*height)){
        gradSO = 0;
    }else{
        gradSO = in[index + width - 1] - getCurrentPixel(in);
    }

    fPerona[0] = exp(-pow((float) (gradN/kappa),2));
    fPerona[1] = exp(-pow((float) (gradNL/kappa),2));
    fPerona[2] = exp(-pow((float) (gradL/kappa),2));
    fPerona[3] = exp(-pow((float) (gradSL/kappa),2));
    fPerona[4] = exp(-pow((float) (gradS/kappa),2));
    fPerona[5] = exp(-pow((float) (gradSO/kappa),2));
    fPerona[6] = exp(-pow((float) (gradO/kappa),2));
    fPerona[7] = exp(-pow((float) (gradNO/kappa),2));

    diffImg[0] = even*fPerona[0]*gradN;
    diffImg[1] = odd*fPerona[1]*gradNL;
    diffImg[2] = even*fPerona[2]*gradL;
    diffImg[3] = odd*fPerona[3]*gradSL;
    diffImg[4] = even*fPerona[4]*gradS;
    diffImg[5] = odd*fPerona[5]*gradSO;
    diffImg[6] = even*fPerona[6]*gradO;
    diffImg[7] = odd*fPerona[7]*gradNO;

    float sum=0.0;

    for (int k = 0; k < 8; k++) {
        sum += diffImg[k];
    }

    out[index] = (float)(in[index] + sum);
}

```

Figura E.2 – Código (*Kernel*) do filtro de difusão anisotrópico. (Fonte: elaborada pelo autor)