UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Gerador automático de código Python para processamento heterogêneo de imagens em OpenCL por meio da biblioteca VisionGL

Trabalho de Conclusão de Curso

Artur Santos Nascimento

Departamento de Computação/UFS

São Cristóvão – Sergipe

2019

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Artur Santos Nascimento

Gerador automático de código Python para processamento heterogêneo de imagens em OpenCL por meio da biblioteca VisionGL

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Daniel Oliveira Dantas

São Cristóvão – Sergipe

2019

Este trabalho é dedicado à minha família.
Pois mesmo nas adversidades e desafios que a vida me proporcionou,
foi ela quem sempre esteve comigo e me possibilitou alcançar este objetivo.

Agradecimentos

Embora seja eu quem receba o diploma, as responsabilidades por este feito não são exclusivamente minhas. Nessa caminhada, muitas pessoas me ajudaram, me acompanharam, me passaram conhecimento e me permitiram chegar até este trabalho.

Primeiramente, minha família. Mesmo em tantas diversidades, sempre estiveram comigo, me incentivaram a continuar estudando, se esforçaram para que eu conseguisse suplantar cada um dos obstáculos e me permitiram chegar até onde estou hoje. A minha mãe, Edna; e meu pai, Claudio; que com muito esforço conseguiram permitir que eu tivesse educação e saúde por toda minha vida. A meu irmão, Samuel, pela paciência e conversas. A minha avó Maria Aglária, que sempre me incentivou a continuar estudando e minha avó Francisca, que não está mais conosco mas me inspirou em minha iniciação científica. Meu muito obrigado a todos vocês.

De todos os amigos que acumulei desde que me mudei para Aracaju, hoje alguns moram longe e outros encontro com muito pouca frequência. Muito embora, vocês continuam comigo, me dão força, me apoiam e me ajudam mesmo nos momentos mais críticos. Sarah, Gabriel e Gaby, vocês são os melhores amigos que eu poderia sonhar em ter. Muito obrigado por tudo.

Aos amigos que fiz na Universidade, agradeço por cada conversa, aos apoios, caronas, cafés, estudos em grupo. Aos colegas do PRODAP, agradeço pelos bons momentos e pelas aventuras de desenvolver um sistema do zero. Foram ótimas experiências e meu amadurecimento no desenvolvimento de *softwares* certamente alcançou outro patamar graças a vocês. A Jusley, Alana, Marina e Elton meu muito obrigado pelo companheirismo de sempre. Vocês são pessoas incríveis e eu sou muito grato por ter vocês como amigas e amigo meus. E a você, Alessandra, muito obrigado pela sua paciência comigo, sua cordialidade e por estar ali sempre.

Por fim, a todos os professores que tive na UFS. Em especial ao professor Bruno, que nos orientou no PRODAP; ao professor Gilton "Mal" Ferreira, que me orientou na iniciação científica e me mostrou a felicidade que é descobrir coisas novas e ter um artigo publicado; à professora Beatriz Trinchão, por ser uma das melhores professoras que já tive, mostrar com empolgação o que é trabalhar com o que gosta e aceitar o convite, disponibilizando seu tempo para ler e assistir a minha defesa; e ao professor Daniel Dantas, que me orientou neste trabalho e me mostrou o quão interessante é fazer atividades serem completadas mais rápido utilizando abordagens diferenciadas. Muito obrigado a todos vocês.

Cada um de vocês, à sua forma, foi importante para a minha formação acadêmica e de vida. As conversas, debates, discussões e brincadeiras que tivemos me agregaram de diversas formas e me ajudaram na minha caminhada até hoje. Muito obrigado por tudo.

“A felicidade pode ser encontrada
inclusive nos momentos mais escuros;
só é preciso lembrar-se de acender a luz.”
(Alvo Dumbledore)

Resumo

A computação heterogênea pode ser definida como um conjunto de dispositivos computacionais que processam informações de formas distintas e trabalham em conjunto. Algumas aplicações, como os jogos eletrônicos, utilizam-se da computação heterogênea (que neste caso, consiste na utilização da CPU em conjunto com o GPU) para renderizar cenas mais complexas e/ou realistas. O processamento de imagens também pode tirar proveito da grande quantidade de núcleos que as GPUs oferecem, executando esses processos de forma mais rápida. Para utilização desse poder de processamento, existem alguns *frameworks* como o *OpenCL*, que ajudam a usar computação paralela, encapsulando algumas complexidades do desenvolvedor. Este trabalho descreve o desenvolvimento do *wrapper* Python para a biblioteca *VisionGL*, de forma que se integre à biblioteca. Foi desenvolvido também um gerador de código automático para gerar o *wrapper Python* sempre que necessário. Além disso, É feito um teste comparativo de desempenho entre a versão Python criada e a versão C++ da VisionGL, que mostrou vantagem significante para versão Python na maioria dos testes executados.

Palavras-chave: Computação Heterogênea, *Wrapper*, *OpenCL*, Python, *PyOpenCL*.

Abstract

Heterogeneous computing can be defined as a set of computing devices that process information in different ways to work together. Some applications, like games, use heterogeneous computing (which in this case, consists on using CPU and GPU) to render more complex or realistic scenes. Image processing applications can also take advantage using the amount of cores that GPU offers to execute the processes more quickly. To use this processing power, there are some frameworks like *OpenCL*, that helps to use parallel computing, hiding some complexities from the developer. This work describes the development of a Python wrapper in a way it integrates the VisionGL library. An automatic code generator is developed to automatically generate the Python wrapper whenever needed and a comparative performance test between the C++ and Python versions of VisionGL is performed, which showed significant advantage to the Python version that was executed.

Keywords: Heterogeneous Computing, Wrapper, OpenCL, Python, PyOpenCL.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

GPU	*Graphic Processing Unit* (Unidade de Processamento Gráfico)
CPU	*Central Processing Unit* (Unidade de Processamento Central)
AP	*Automata Processor* (Processador de Autômatos)
DSP	*Digital Signal Processor* (Processador Digital de Sinais)
ASIC	*Application-Specific Integrated Circuit* (Circuito Integrado para Aplicação Específica)
API	*Application Programming Interface* (Interface de Programação de Aplicação)
OCL	*Open Computing Language* ou OpenCL
CUDA	*Compute Unified Device Architecture*
VGL	Biblioteca VisionGL
DVFS	*Dynamic Voltage and Frequency Scaling*
SIMD	*Single Instruction, Multiple Data*
MIMD	*Multiple Instruction, Multiple Data*
HPC	*High Performance Computing*
GPGPU	*General-Purpose GPU*
CD	*Computing Devices*
RTCG	*Real-Time Code Generation*

Sumário

1 Introdução

A computação heterogênea é uma área da computação que estuda e desenvolve formas de fazer dispositivos computacionais (*computing devices* ou CDs) que processam informações de formas diferentes trabalharem em conjunto. Um programa preparado para executar em um sistema heterogêneo pode escolher o dispositivo mais apropriado para executar uma determinada parte do código, de forma a otimizar o tempo total necessário para que a tarefa seja concluída (Zahran, 2017).

Com a tendência para o uso da computação heterogênea, se fez necessário criar uma linguagem ou ferramenta que a suporte em dispositivos de fabricantes e aplicações diversas. Utilizando-se de ferramentas como o *OpenMP*, *CUDA* e o *OpenCL*, os desenvolvedores adaptaram aplicações que possuam algoritmos paralelizáveis para executar em processadores *multicore* e GPUs, aproveitando-se da característica paralela destes dispositivos para alcançar um desempenho melhor em tarefas paralelizáveis (Stone; Gohara; Shi, 2010; Klöckner et al., 2012). Um exemplo de aplicação que tira vantagem da paralelização é o processamento de imagens.

O processamento de imagens tem como objetivo extrair informações de uma ou mais imagens de entrada. No decorrer do processo, muitos cálculos independentes entre si são feitos com base nos pixels das imagens de entrada. Dessa forma, os cálculos que não possuem interdependência podem ser feitos paralelamente (Saxena; Sharma; Sharma, 2016; Nugteren; Corporaal; Mesman, 2011), o que acabou levando à utilização de GPUs e outros aceleradores para atuar na execução dessas aplicações (Zahran, 2017; Stone; Gohara; Shi, 2010).

O *OpenCL* é um padrão aberto para programação paralela de propósito geral para CPUs, GPUs e outros dispositivos. Ele consiste em uma linguagem de programação *C-like* para desenvolvimento dos *kernels* – as rotinas paralelizáveis –, uma biblioteca C e um sistema em tempo de execução (Munshi et al., 2011). A linguagem de programação Python possui a biblioteca *PyOpenCL*, que permite a integração entre o Python e o *OpenCL* (Klöckner et al., 2012)

A VisionGL (VGL) é uma biblioteca de processamento de imagens e vídeos escrita em C++ que tem por objetivo prover uma forma facilitada de prototipar funções de processamento de imagens e vídeos. A VGL utiliza-se do *OpenCL*, CUDA e GLSL para acelerar a execução das tarefas. A VGL possui também geradores automáticos de código que, partir dos *kernels OpenCL* cria os *wrappers* responsáveis por encapsular as complexidades do código *OpenCL*, provendo uma interface facilitada para o programador escrever o software desejado (Dantas; Leal; Sousa, 2015; Dantas; Leal; Sousa, 2016).

Os *wrappers* são classes que adaptam uma interface para outra, fazendo com que duas interfaces incompatíveis se comuniquem. A VGL dispõe de *wrappers* para *OpenCL, CUDA* e *GLSL* a fim de adaptar os tipos de dados e preparar a execução dos processos na biblioteca. Além disso, geradores automáticos de código geram os *wrappers* com base nos *kernels* da biblioteca a fim de facilitar a produção dos *wrappers* caso novas funções sejam adicionadas ou funções antigas sejam modificadas (Dantas; Barrera, 2011).

1.1 Objetivos

Este trabalho tem como objetivo produzir um gerador automático de *wrappers* para a VGL na linguagem Python, de forma a oferecer uma abordagem com maior potencial produtivo, visto que Python é uma linguagem mais expressiva e que permite executar um mesmo procedimento em menos linhas se comparado com o C++.

1.1.1 Objetivos específicos

Os objetivos específicos deste trabalho são descritos a seguir:

- Elaborar um script que crie os *wrappers* automaticamente, usando como base os *kernels OpenCL* da biblioteca, provendo assim um meio automatizado de gerar os *wrappers* caso sejam adicionados novos *kernels* à VGL. Esse gerador de código para linguagem Python será escrito em Perl, utilizando como base os geradores já existentes na VGL;
- Os *wrappers* em devem receber os objetos Python `pyopencl.Image`, `pyopencl.Buffer`, `numpy.float32`, `numpy.int32` e `numpy.uint8` e transformá-los para que os *kernels OpenCL* possam ler esses dados como objetos *OpenCL* `image2d_t` e `image3d_t`; variáveis *OpenCL* `float`, `int` e `unsigned char`; *arrays OpenCL* `float*`, `unsigned char*` e `char*`; e as estruturas da VGL `vglClStrEl*` e `vglClShape*` pelo *kernel*;
- Desenvolver um teste de desempenho com objetivo de comparar os *wrappers* C++ e Python da VGL no que se refere a velocidade de execução das funções dos *wrappers*;

1.2 Estrutura do Documento

Este trabalho está estruturado nos seguintes capítulos:

- Capítulo 1 – Introdução: uma visão geral sobre o a área e descrição dos resultados visados por este trabalho;
- Capítulo 2 – Conceitos: definições principais para a compreensão to tema.;
- Capítulo 3 – Trabalhos Relacionados: trabalhos que serviram de referência para o desenvolvimento do trabalho;
- Capítulo 4 – Metodologia: descrição do desenvolvimento dos *wrappers*;
- Capítulo 5: Configuração do Sistema de Execução – descreve o sistema em que os testes foram executados e como foi feita a configuração e instalação da VGL;
- Capítulo 6 – Resultados: apresentação dos resultados obtidos;
- Capítulo 7 – Conclusão: considerações finais do trabalho e sugestão de trabalhos futuros.

2

Conceitos Básicos

Neste capítulo serão apresentadas as definições sobre o que são a computação heterogênea (seção 2.1); o *OpenCL* (seção 2.2); o *wrapper* (seção 2.3); a VGL (seção 2.4), o que são as *Threads* (seção 2.5); o conceito de nuvem (seção 2.6); e outras definições (seção 2.7).

2.1 Computação Heterogênea

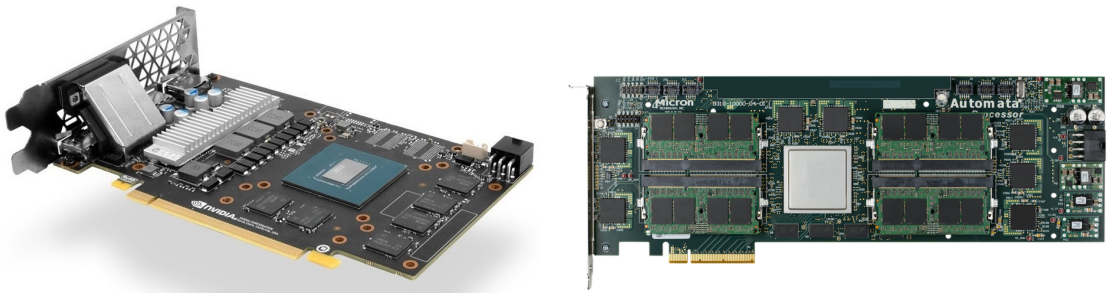
Quando os sistemas paralelos começaram a ser desenvolvidos, eles eram baseados em processadores cujos núcleos de processamento (*cores*) eram homogêneos entre si. Ou seja, que os CDs eram compostos pelo mesmo conjunto de instruções; capacidade de executar mais de uma *thread* por ciclo de *clock*; e demais capacidades de processamento (Zahran, 2017).

Com a evolução das tecnologias, os processadores passaram a ter mais *cores* e cada *core* seu próprio escalonamento dinâmico de frequência e voltagem (DVFS) (Silberschatz, 2008; Zahran, 2017). Isso fez com que, mesmo *cores* com arquitetura e capacidades similares fossem heterogêneos entre si, visto que um *core* mais quente trabalharia em menor frequência – e consequentemente, mais lento – que os demais *cores*. Esse é o primeiro tipo de heterogeneidade.

A segunda forma de heterogeneidade envolve *cores* com arquiteturas diferentes, como é o caso dos processadores *big.LITTLE*: essa arquitetura utiliza dois conjuntos de *cores*: o conjunto "*big*", constituído de *cores* mais simples e que consomem menos energia para quando o sistema requer pouco poder de processamento; e o conjunto "*LITTLE*", constituído *cores* mais poderosos que só são utilizados quando mais poder de processamento é necessário. Essa abordagem melhora o consumo de energia dos dispositivos que a utilizam (Padoin et al., 2014; Zahran, 2017; Khokhar et al., 1993).

Nessas duas formas de heterogeneidade, o desenvolvedor tem a visão de que esses *cores* fazem o processamento de forma sequencial, mesmo que haja paralelismo na execução das instruções – como é o caso das instruções do tipo SIMD e MIMD –. Neste sistema, é possível

escrever código paralelo, mas as *threads*(ou processos) são executadas de forma sequencial pelo *core* (Zahran, 2017; Khokhar et al., 1993).



(a) Fonte – Material de propaganda da NVIDIA. (b) Fonte – Reprodução (Wang et al., 2016).

Figura 1 – (a) GPU NVIDIA GTX 1060 com dissipadores removidos para visualização do PCB; (b) Placa de testes do processador de autômatos desenvolvido pela Micron.

O terceiro tipo de heterogeneidade se difere dos outros dois por envolver um processador que siga o modelo de execução não-sequencial, como é o caso das GPUs. A GPU é um dispositivo que possui uma grande quantidade de *cores* e uma alta velocidade de memória, possibilitando a computação paralela. Uma placa de referência NVIDIA GeForce GTX 1060 – mostrada na Figura 1a – oferece 3GB ou 6GB de memória com *bandwidth* de 192 GB/s; e 1280 núcleos funcionando a 1.7GHz. Tomando como exemplo o cálculo da multiplicação de uma matriz por uma constante, enquanto uma CPU precisa multiplicar cada elemento da matriz por vez, a GPU tem a capacidade de multiplicar a matriz inteira de uma vez só – ou em poucas iterações, dependendo da quantidade de *cores* da GPU e do tamanho da matriz em questão (Khokhar et al., 1993).

Outro exemplo de processador paralelo é o Processador de Autômatos (AP). O AP é um acelerador de expressões regulares que executa em paralelo (a placa de desenvolvimento da AP é mostrada na Figura 1b). Se um problema puder ser formulado nos moldes de uma expressão regular, um AP entregará um desempenho superior a uma GPU. Esse tipo de processador é muito eficiente no processamento de grafos, análise de padrão e estatística (Dlugosch et al., 2014).

Nesse contexto, é possível definir a computação heterogênea como sendo o uso de um conjunto de CDs a fim de suprir diferentes necessidades computacionais, onde um CD pode constituir-se desde um núcleo de processador, empregado apenas em um tipo específico de processamento, a até um nó em uma nuvem computacional (Khokhar et al., 1993).

2.2 *OpenCL*

A necessidade crescente por desempenho computacional na ciência e engenharia levou ao uso de sistemas heterogêneos com GPUs e outros aceleradores – como o Intel Xeon Phi – atuando como coprocessadores em cargas de trabalho que possam ser paralelizáveis (Stone; Gohara; Shi, 2010; Munshi et al., 2011).

Em contrapartida, desenvolver aplicações para plataformas paralelas pode se tornar desafiador, tendo em vista que os dispositivos podem manipular dados em memória de formas diferentes – a depender do fabricante e da finalidade para a qual o dispositivo foi criado – dificultando ao desenvolvedor tirar proveito da heterogeneidade dos dispositivos a partir de um único código (Munshi et al., 2011).

Nesse contexto, o *Open Computing Language* (*OpenCL*) foi desenvolvido com objetivo de se tornar um padrão *Open Source* para programação de coleções de CPUs, GPUs e outros dispositivos paralelos. A especificação do *OpenCL* envolveu empresas desenvolvedoras de software e hardware com objetivo de oferecer ao desenvolvedor uma forma de fazer o mesmo código funcionar em diversos dispositivos (Stone; Gohara; Shi, 2010; Munshi et al., 2011).

O *OpenCL* é um *Framework* que consiste em uma linguagem de programação; uma API; bibliotecas; e um sistema em tempo de execução para dar suporte ao desenvolvimento, fazendo com que não seja necessário ao desenvolvedor, por exemplo, mapear seu programa de propósito geral para uma API de gráficos tridimensionais, como é o caso do *OpenGL* e *DirectX*.

Além disso, o *OpenCL* dá suporte aos modelos de programação baseados em tarefas e dados; utiliza do subconjunto ISO C99 com extensões para paralelismo [1]; cumpre as definições numéricas com base na IEEE 754 [2]; define perfis de configuração para dispositivos portáteis e embarcados; e oferece interoperabilidade com *OpenGL*, *OpenGL ES* e outras *APIs* gráficas.

O *OpenCL* também define como os *kernels* devem ser executados (subseção 2.2.1); de que forma os contextos devem ser definidos (subseção 2.2.2); a maneira com a qual a interação entre o *host program* (programa que é executado no hospedeiro) e os CDs se comunicam (subseção 2.2.3); e como os objetos de memória devem ser definidos (subseção 2.2.4).

2.2.1 Modelo de Execução

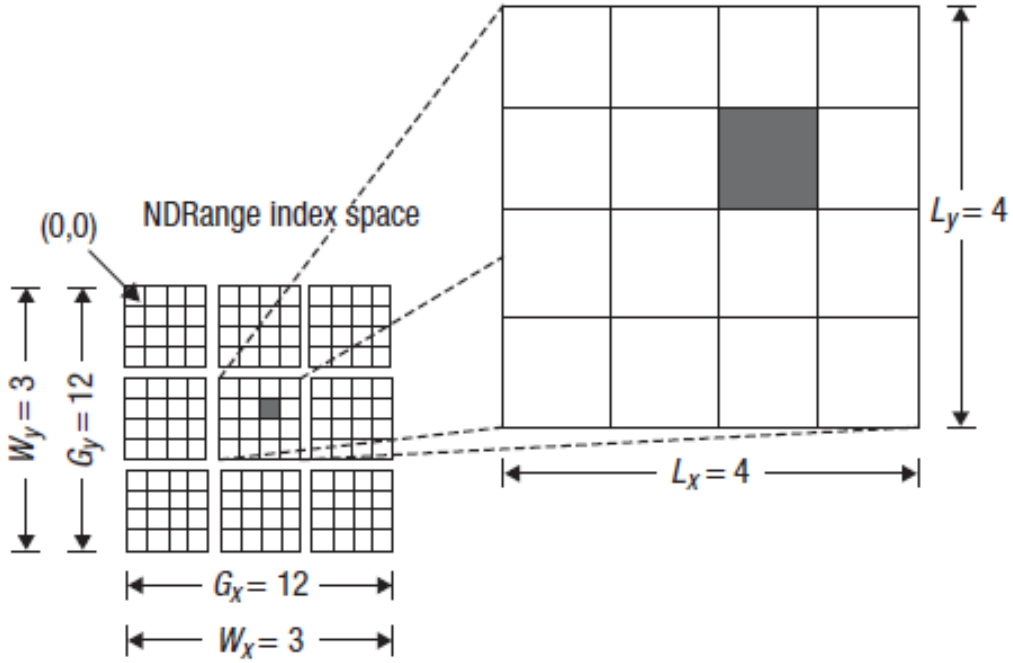
A execução de um programa *OpenCL* consiste em duas partes distintas: o *host program* e uma coleção de um ou mais *kernels*. O *host program* define o contexto em que os *kernels* serão executados e gerencia a suas execuções, faz o pré-processamento das estruturas de dados para execução no *kernel* e recepciona o resultado depois do kernel tê-las processado (Munshi et al., 2011, p. 11).

Já os *kernels* são funções que executam no dispositivo-alvo e têm a finalidade de transformar os objetos de entrada em objetos de saída. O *OpenCL* não define os detalhes de como os programas funcionam, pois o ele é apenas uma especificação. Entretanto, a forma com que os objetos interagem entre o CD e o *host* é definida pelo padrão *OpenCL* (Munshi et al., 2011, p. 12).

Ao submeter o comando que envia um *kernel* para ser executado, o *OpenCL* aloca um

[1] Especificação da linguagem C, revisão de 1999 (ISO, 1999).
[2] Especificação da IEEE sobre tipos e precisão numéricos de ponto flutuante (IEEE, 1985).

Figura 2 – Exemplo da organização do espaço de indexação e de sua subdivisão em *work-groups* e *work-items*.



Fonte – Imagem retirada do livro (Munshi et al., 2011, p. 16).

espaço de indexação, chamado *NDRange*. Até a versão 2.1, esse espaço de indexação suporta uma, duas ou três dimensões, onde para cada ponto nesse espaço de indexação uma instância do *kernel* é executada. Uma instância é chamada de "*work-item*", e é identificada por suas coordenadas no espaço de indexação, que por sua vez são chamadas de *global ID* (Munshi et al., 2011, p. 13).

Conjuntos de *work-items* são organizados em *work-groups* (vide Figura 2). Os *work-groups* dividem o espaço de indexação de forma que a soma das extensões dos *work-groups* é igual a extensão do espaço de indexação. Cada *work-group* possui um identificador único com a mesma dimensionalidade do espaço de indexação e os *work-items* possuem um identificador local dentro do *work-group*, de forma que um dado *work-item* possa ser encontrado tanto pelo seu *global ID* quanto pela combinação do identificador do *work-group* e de seu identificador local (Munshi et al., 2011, p. 14).

Os *work-items* de um *work-group* são executados de forma concorrente no processamento de uma unidade de computação. Apesar disso, uma implementação do *OpenCL* pode sequencializar a execução de *kernels*, podendo até sequencializar a execução de *work-groups*. Portanto, o padrão garante apenas que os *work-items* de um *work-group* são executados concorrentemente. Já os *work-groups* podem ou não ser executados de forma concorrente entre si (Munshi et al., 2011, p. 14).

2.2.2 Contexto *OpenCL*

O *host program* é uma parte essencial do *OpenCL*. É nele que se definem os *kernels*; o contexto *OpenCL*; o espaço de indexação *NDRange*; as filas de comando (subseção 2.2.3); e é a partir dele que os comandos são enviados ao CD. Primeiro, o *host program* precisa definir o contexto *OpenCL*. O contexto define o ambiente no qual os *kernels* são definidos e executados (Munshi et al., 2011, p. 17). Esse ambiente é constituído pelos:

- ***Devices*** - Coleção de CDs que podem ser usados pelo *host program*;
- ***Kernels*** - As funções que são executadas nos CDs;
- ***Memory Objects*** - O conjunto de objetos que são visíveis aos CDs, contendo valores que podem ser operados por instâncias do *kernel*;
- ***Program Objects*** - Objeto que manipula os códigos dos *kernels*, compilando-os, atribuindo os argumentos. O *program object* é compilado em tempo de execução pelo *host program*.

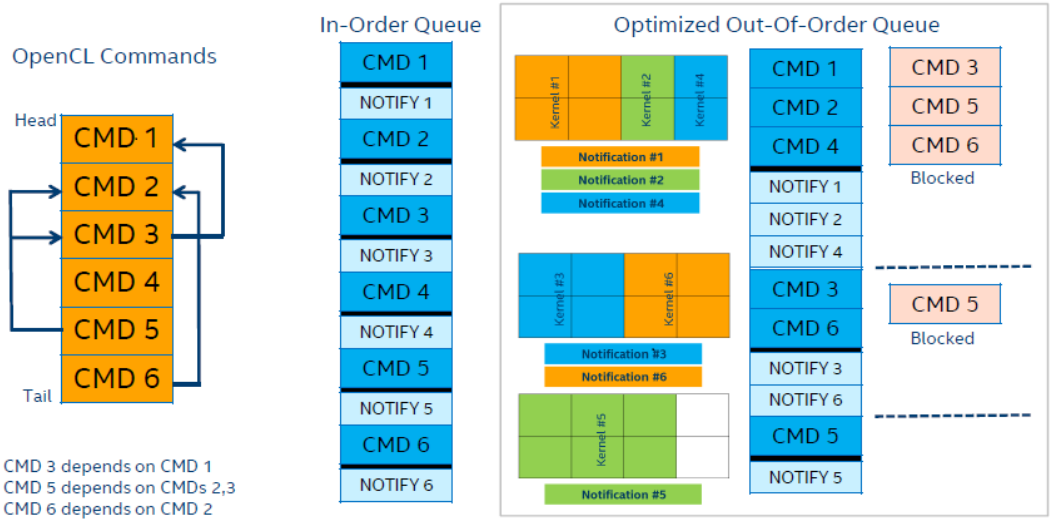
O contexto é criado e manipulado por meio de funções definidas na API do *OpenCL*, de forma que o *host program* solicita à API que procure pelos recursos computacionais no sistema em que ele está sendo executado. A partir de então, o *host program* pode decidir quais dispositivos serão utilizados na aplicação *OpenCL*, que a depender do problema e dos *kernels*, um ou mais tipos de dispositivos podem ser escolhidos, definindo assim o contexto (Munshi et al., 2011, p. 17).

2.2.3 Filas de comando

A comunicação entre *host program* e os CDs ocorre por meio de comandos enviados pelo *host program*. Esses comandos são adicionados a uma fila de comandos (*command-queue*), onde ficam até serem executados no CD. Uma fila é associada a um CD depois de o contexto ter sido definido (Munshi et al., 2011, p. 18).

O *OpenCL* define comandos de execução do kernel; transferência de dados entre o *host* e objetos de memória, alocação e desalocação de memória; e comandos de sincronização. A execução dos comandos pode ser feita de duas formas: em ordem – quando os comandos são executados na ordem em que foram enfileirados –, ou fora de ordem – quando o desenvolvedor explicita que certo comando pode ser executado logo após outro comando, ou apenas depois da finalização de um comando específico – (Figura 3). A execução fora de ordem é um recurso opcional de implementação e pode não ser suportada em alguns dispositivos (Munshi et al., 2011, p. 19).

Figura 3 – Exemplo de uma fila de comandos em ordem e fora de ordem.



Fonte – Reprodução (Bi; Sardella, 2017)

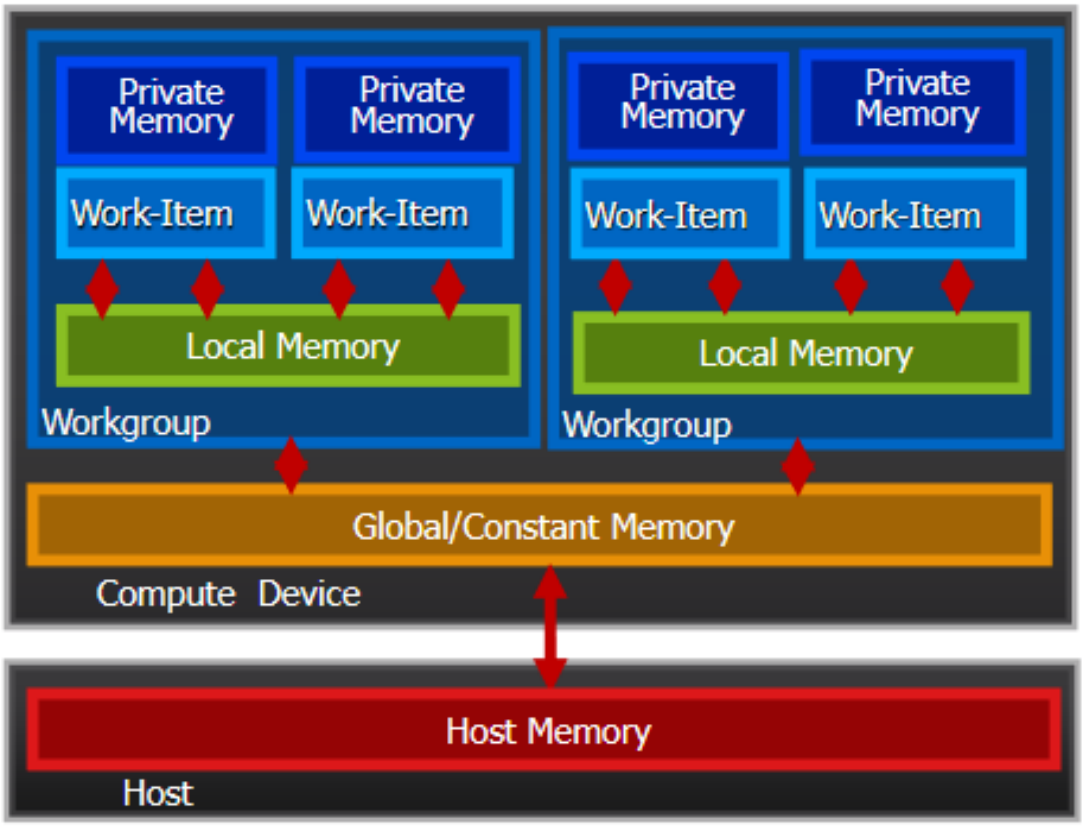
2.2.4 Modelo de Memória

Em sistemas heterogêneos, se faz necessária a gerência de espaços de endereçamento de memória diferentes, pois tanto o *host* quanto os CDs possuem memórias próprias e o programador precisa manipular os dados entre esses diferentes espaços de endereçamento. Para lidar com isso, o *OpenCL* possui o recurso dos objetos de memória. Eles são definidos pelo *host program* e transferidos entre ele e os CDs (Munshi et al., 2011, p. 21).

Até *OpenCL* 2.1, foram definidos dois tipos de objetos de memória: *buffer objects* e *image objects*. Um *buffer object* consiste em um espaço contínuo de memória que fica disponível ao *kernel*. O desenvolvedor pode utilizar esse objeto para mapear estruturas de dados para que sejam passadas do *host* ao *kernel*, por exemplo. Os *image objects* são objetos usados para representar um *buffer* que armazena uma textura ou imagem. Nos *kernels*, o objeto *Image* é acessível por um vetor onde cada entrada possui quatro componentes (que podem ser os canais vermelho, verde, azul e transparência, por exemplo). O objeto *Image* também oferece funções para escrever e ler uma imagem, transformando os dados desse objeto para o formato apropriado para imagem e vice-versa. A partir da especificação *OpenCL* 1.1, subregiões de objetos de memória podem ser especificadas como um objeto de memória distinto (Munshi et al., 2011, p. 21). O esquema de divisão das memórias no *OpenCL* segue o exemplo ilustrado pela Figura 4, onde:

- A ***Host Memory***, em vermelho na Figura 4, é a memória RAM do computador. Essa região de memória é visível apenas ao *host*. O *OpenCL* define apenas como a *host memory* interage com seus objetos e construtores.
- A ***Global Memory***, presente na parte laranja da Figura 4, é localizada no CD. Essa memória

Figura 4 – Exemplo do esquema da organização das memórias no *OpenCL*.



Fonte – Reprodução (Munshi et al., 2011)

permite leitura e escrita por todos os *work-groups* e *work-items*.

- A ***Constant Memory***, também presente na área laranja da Figura 4, é uma área de memória que permanece sem alterações durante toda execução do *Kernel*. O *host program* aloca e inicializa os objetos postos na *constant memory* e os *work-items* têm apenas poder de leitura nessa área de memória.

- A ***Local Memory***, na parte verde da Figura 4, é uma área de memória local ao ponto de vista dos *work-groups*. Nessa área de memória podem ser alocadas as variáveis compartilhadas pelos *work-items* no *work-group* em questão. A *local memory* pode ser mapeada em seções da *global memory*.

- A ***Private Memory***, na parte azul-claro da Figura 4, é uma área de memória que apenas o *work-item* pode acessar. Um dado *work-item* não pode acessar variáveis que outro *work-item* definiu em sua *private memory*.

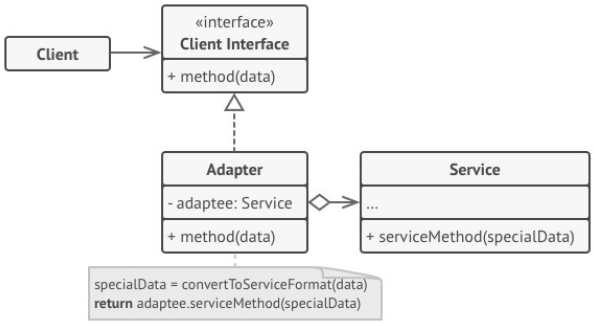
O *host* e os CDs são independentes um do outro em relação a suas memórias. Entretanto, elas podem interagir entre si copiando dados, mapeando e "desmapeando" regiões de um *memory object*. A cópia de dados se dá pelo enfileiramento de comandos de cópia, que podem ser blocantes ou não-blocantes. Quando blocantes, a função retorna apenas quando a cópia é

finalizada; e quando não-blocante, a função retorna assim que enfileirar todos os comandos para que a cópia seja realizada. Já o mapeamento de memória permite ao *host* mapear uma região de um *memory object OpenCL* no seu espaço de memória. Esse comando também pode ser blocante ou não-blocante (Munshi et al., 2011, p. 22).

2.3 O *Wrapper*

O *wrapper* – também conhecido como *Adapter* – é um padrão estrutural de projeto que converte a interface de uma funcionalidade de forma apropriada para outra funcionalidade, permitindo que interfaces incompatíveis entre si possam se comunicar (Gamma, 2009, p. 140). Na Figura 5, é mostrado um exemplo no qual o *wrapper* faz a comunicação entre uma classe origem e a classe destino.

Figura 5 – Exemplo do funcionamento de um *Wrapper*.



Fonte – Reprodução (Gamma, 2009).

No que se refere a este trabalho, um *wrapper* Python recebe como argumentos objetos Python e os processa de forma que o *OpenCL* possa ler esses argumentos de forma apropriada. O processamento desses dados é detalhado na seção 4.2.

Além disso, é no *wrapper* que esses dados são validados; os *kernels* são compilados (ou carregados, caso já tenham sido compilados); a memória é alocada no CD; a cópia dos dados que forem necessários entre o *host* e o CD é efetuada; os objetos *OpenCL* que devem ser usados como argumento na execução de um *kernel* são referenciados; e o *kernel* é encaminhado para execução no CD. Esses procedimentos são detalhados na seção 4.3.

2.4 *VisionGL*

A *VisionGL* (VGL) é uma biblioteca de código aberto que oferece um conjunto de funções para processamento de imagens bidimensionais, tridimensionais, vídeos e uma abstração para imagens n-dimensionais. Esses processamentos podem ser feitos utilizando *CUDA*, *GLSL*, *OpenCL* e o *OpenCV* (Dantas; Barrera, 2011).

As funções de processamento da biblioteca estão dispostas na forma de *kernels*, em sua hierarquia de pastas. Os *wrappers* da biblioteca encapsulam as complexidades das chamadas de API suportadas pela biblioteca, facilitando o uso (Dantas; Barrera, 2011).

A VGL também é dotada de um gerador de código automático escrito em linguagem *Perl* que analisa os *kernels* e, de acordo com os argumentos recebidos pelos *kernels*, monta uma *wrapper* C++ que encapsula, no caso do *OpenCL*, todas as chamadas de fila de comando, preparação das imagens e das estruturas de dados. Dessa forma, para adicionar novas funções à biblioteca, se faz necessário criar o *kernel OpenCL* e executar o *script* que gera os *wrappers* (Dantas; Leal; Sousa, 2015; Dantas; Leal; Sousa, 2016; Dantas; Barrera, 2011).

2.5 *Threads*

Uma *thread* é uma unidade básica de utilização da CPU. Ela é compreendida por uma ID de *thread*, um contador de programa, um conjunto de registradores e uma pilha; compartilhando seu código com outras *threads* originadas do mesmo processo. Um processo tem pelo menos uma *thread* (Silberschatz, 2008).

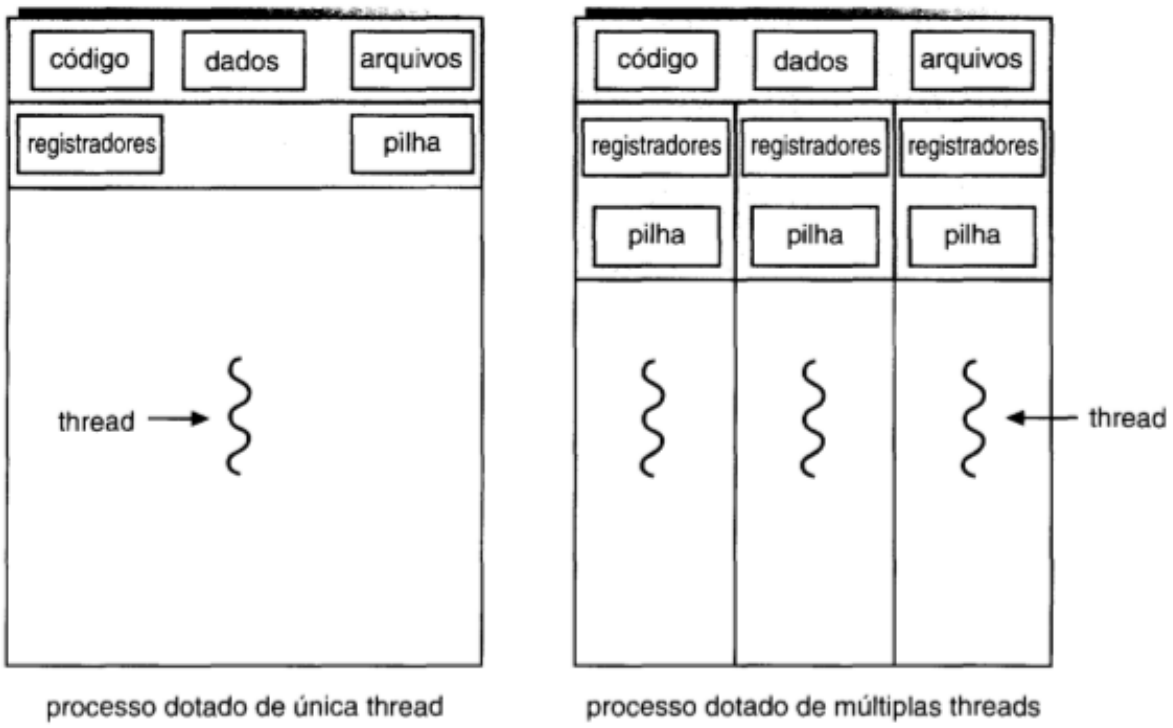
Se um processo tiver mais de uma *thread*, ele pode fazer mais de uma coisa ao mesmo tempo, mas o suporte a *threads* deve ser oferecido pelo sistema operacional (Silberschatz, 2008). A Figura 6 mostra o esquema de um processo com uma *thread* e um processo com várias *threads*. Entre os maiores benefício no uso de *threads* estão a responsividade, já que um programa pode continuar executando mesmo que uma parte dele esteja bloqueada; o compartilhamento de recursos, com várias *threads* trabalhando em coisas diferentes dentro de um mesmo espaço de endereços; economia com alocação de memória e recursos na criação de novos processos; e a utilização de arquiteturas multiprocessadas, como uma GPU (Silberschatz, 2008).

2.6 Nuvem

Nuvem, também conhecida como *cloud computing*, é um modelo de computação que provê acesso ubíquo, conveniente e sob demanda a recursos computacionais. Uma aplicação de computação em nuvem pode oferecer serviços de armazenamento, banco de dados, redes, software e inteligência artificial de forma transparente ao usuário: mesmo que os computadores que ofereçam esses serviços estejam em lados opostos da Terra, o usuário pode evocá-los como se estivessem sob um único computador (Mell; Grance, 2011).

Empresas como a Microsoft (<https://azure.microsoft.com>), a IBM (<https://www.ibm.com/cloud/>), a Amazon (<https://aws.amazon.com/pt/?nc2=h_lg>) e diversas outras ofrecem serviços do tipo. Entre os benefícios do emprego da computação em nuvem estão a economia com servidores, a utilização sob demanda dos serviços oferecidos pela plataforma e a economia com manutenção de *hardware* e *software*.

Figura 6 – Processo com uma *thread* e processo com múltiplas *threads*.



Fonte – Reprodução (Silberschatz, 2008, p 104)

2.7 Outras Definições

Nesta seção, são explicados o que é o processador Intel Xeon Phi (subseção 2.7.1) e o OpenGL (subseção 2.7.2).

2.7.1 Processador Xeon Phi

O Xeon Phi é um processador criado pela Intel para prover uma plataforma paralela própria. Segundo a Intel, a plataforma Xeon Phi oferece mais desempenho por unidade de energia gasto comparando o Xeon Phi 7250 com um sistema com 2 soquetes de processadores Xeon E5-2697 v4 em um sistema *Red Hat Linux* (Intel, 2019).

2.7.2 OpenGL

O OpenGL é um padrão para gráficos de alto desempenho. Ele foi o primeiro padrão a oferecer gráficos 2D e 3D interativos, em 1992. Entre as vantagens de utilizar o OpenGL estão a portabilidade (pois ele é suportado por diversos sistemas operacionais, linguagens e *hardware*), escalabilidade, estabilidade e a documentação abrangente. Ainda segundo seus criadores, o OpenGL é a API mais utilizada no mundo para gráficos 2D e 3D (Khronos Group, 2019a).

O OpenGL ES é uma versão do OpenGL para execução em sistemas embarcados, como

consoles, telefones e automóveis. Essa versão é otimizada para consumir pouca energia (Khronos Group, 2019b).

2.7.3 Processos de Segmentação e Registro

O processo de segmentação é que identifica e classifica informações obtidas em uma amostra representada digitalmente. No ambiente médico essa representação amostrada geralmente pode ser uma imagem obtida por instrumentos médicos como equipamentos de ultrassom, tomografia ou ressonância magnética (Insight Toolkit, 2019).

O processo de registro é aquele em que dados obtidos de formas diferentes são alinhados para que seja feita correspondência entre eles. Como exemplo, é possível alinhar a imagem obtida por um tomógrafo e uma ressonância magnética para que as informações obtidas nessas duas imagens sejam combinadas (Insight Toolkit, 2019).

3

Trabalhos Relacionados

Os critérios de escolha da linguagem para desenvolvimento deste trabalho estão descritos na seção 3.1. Em seguida, a subseção seção 3.2 mostra algumas bibliotecas para processamento de imagens e a seção 3.3 mostra algumas APIs para processamento paralelo e de GPUs.

3.1 Escolha pela linguagem Python

Algumas linguagens de alto nível possuem bibliotecas que possibilitam o uso do *OpenCL* nessas linguagens. Dentre as bibliotecas encontradas, destacam-se a *QtOpenCL* (Qt), *PyOpenCL* (Python), *ScalaCL* (Scala) e *JavaCL* (Java) (Passerat-Palmbach; Hill, 2014).

O objetivo deste trabalho é a utilização de uma linguagem multiplataforma como *host program*. Das APIs destacadas que possibilitam a utilização do *OpenCL*, as multi-plataforma são o Java, Scala e Python.

A linguagem Scala compila *bytecodes* Java, sendo compatível com a JVM 1.4 (Odersky; Spoon; Venners, 2008). O *ScalaCL* é um plugin que suporta um conjunto restrito de construtos da linguagem, que são mapeados em tempo de compilação para o equivalente em *OpenCL*. A partir de então, o código gerado é compilado para o *OpenCL* e, apenas em tempo de execução, é enviado ao CD (Passerat-Palmbach; Hill, 2014). O modo de operação do *ScalaCL* é incompatível com o que é desejado por este trabalho e portanto a utilização do *ScalaCL* foi descartada.

O *JavaCL*, é uma biblioteca que permite ao desenvolvedor acessar todos os recursos do *OpenCL*, incorporando inclusive os erros *OpenCL* em exceções (Passerat-Palmbach; Hill, 2014). Nenhuma documentação oficial foi encontrada sobre o *JavaCL* e as últimas atualizações no repositório [1] datam de 2015, o que são indícios de que o projeto tenha sido descontinuado. Por esses motivos, o *JavaCL* também foi descartado.

[1] Repositório *JavaCL*: <https://github.com/nativelibs4java/JavaCL>, acessado em 15/04/2019.

O *PyOpenCL* – assim como o *JavaCL* – permite ao desenvolvedor acesso aos recursos oferecidos pelo *OpenCL*, contando também com mapeamento de erros para exceções. O pacote *PyOpenCL* conta com uma documentação completa e o repositório [2] conta com atualizações frequentes. Além disso, a abordagem utilizada para construir as bibliotecas – *Real Time Code Generation (RTGC)* – garantiu uma melhora significativa no desempenho do código (Klöckner et al., 2012).

Com essas informações em mente, concluiu-se que a linguagem Python, com a biblioteca *PyOpenCL* seria a melhor escolha para a implementação deste trabalho.

3.1.1 Documentação Python

O desenvolvimento dos programas deste trabalho teve como referência os construtos, paradigmas e boas práticas explicados em sua documentação primária, com ajuda de demonstrações fornecidas por ela (van Rossum, 2007).

3.1.2 *PyOpenCL*

O *PyOpenCL* é uma biblioteca de código aberto que permite acesso à API *OpenCL* a partir da linguagem Python. O *PyOpenCL* funciona como uma API, permitindo acesso completo aos recursos que o *OpenCL* oferece, inclusive traduzindo os erros do *OpenCL* em exceções do Python, permitindo que o *script* Python funcione de forma equivalente ao programa "*host*" C++ do *OpenCL* (Klöckner et al., 2012; Passerat-Palmbach; Hill, 2014).

Em adição, o *PyOpenCL* também funciona como uma espécie de *wrapper* Python para utilização do *OpenCL*, visto que o *PyOpenCL* oferece uma série de encapsulamentos, exigindo que menos código seja escrito pelo desenvolvedor em comparação com os programas Python de mesma funcionalidade. Além disso o *PyOpenCL* também conta com sua funcionalidade de RTCG, que analisa o código de *kernel* enviado para execução e faz otimizações nele (Klöckner et al., 2012).

3.2 Bibliotecas de processamento de imagens

Nesta subseção são apresentadas algumas bibliotecas de processamento de imagens. Na subseção 3.2.1 o OpenCV e seus módulos são apresentados; na subseção 3.2.2 a biblioteca VIGRA; na subseção 3.2.3 a biblioteca ITK; na subseção 3.2.4 a biblioteca IPP; na subseção 3.2.5 a biblioteca NPP; na subseção 3.2.6 a biblioteca OpenCLIPP; e na subseção 3.2.7 sobre a biblioteca Python *scikit-image*.

[2] Repositório *PyOpenCL*: <https://github.com/inducer/pyopencl>, acessado em 15/04/2019.

3.2.1 OpenCV

A *Open Source Computer Vision Library* (OpenCV) é uma biblioteca de código aberto para visão computacional e *machine learning*, com intuito de prover uma infraestrutura comum para visão computacional e percepção de máquina, possuindo interfaces com C++, Java, Python e MATLAB. Foi iniciada como um projeto de pesquisa da *Intel* em 1998 e disponível desde 2000. O OpenCV possui diversas funções para processamento de imagens, com funções de suavização, correspondência, morfologia, redimensionamento, convolução, gradiente e diversas outras (Bradski; Kaehler, 2008).

Em 2010, foi adicionado ao OpenCV um módulo que provê aceleração por GPU chamado "*GPU module*". Esse módulo cobre boa parte das funcionalidades da biblioteca, sendo implementado em CUDA, possibilitando a utilização de GPUs NVIDIA para paralelização de tarefas sem que o programador seja treinado em programar para GPUs. Esse módulo é consistente com a versão de CPU do OpenCV, o que facilita a adoção pelo desenvolvedor. Em testes de *benchmarking*, o *GPU module* apresentou uma aceleração entre 6 e 30 vezes nas funções usadas para comparação com a versão de CPU do OpenCV (Pulli et al., 2012).

O OpenCV também possui o módulo OpenCL (*OCL module*), que é um módulo da biblioteca que contém um conjunto de classes e funções que implementam e aceleram funcionalidades do OpenCV usando o dispositivos compatíveis com OpenCL. O *OCL module* foi desenvolvido para ser fácil de usar e não requer conhecimento sobre o OpenCL (assim como a versão GPU não requer conhecimento em CUDA). A principal vantagem do *OCL module* é habilitar aceleração em dispositivos que não sejam da NVIDIA (OpenCV, 2019).

3.2.2 VIGRA

A *Vision with Generic Algorithms* (VIGRA) é uma biblioteca de processamento e análise de imagens com ênfase em customização de algoritmos, estruturas de dados e o uso do paradigma de programação genérica. Essa biblioteca está disponível para as linguagens C++ e Python (2.7 e 3.5), mas não possui aceleração por GPU. Segundo seus autores, a biblioteca é flexível o suficiente para que, com uso de *wrappers*, seja possível adaptar os algoritmos da VIGRA para funcionar com outras estruturas de dados já existentes, assim como adaptar o uso das estruturas de dados da VIGRA para outros algoritmos ou aplicações (VIGRA, 2019).

3.2.3 ITK

Em 1999, a *US National Library of Medicine of the National Institutes of Health* recebeu um contrato para o desenvolver uma ferramenta de código aberto para registro e segmentação, que posteriormente veio a se tornar a *Insight Toolkit* (ITK). A ITK está disponível em C++ e possui *wrappers* que são gerados automaticamente, oferecendo interfaces para Java e Python (Insight Toolkit, 2019).

A ITKv4 é uma versão em estágio de desenvolvimento da biblioteca ITK. O objetivo dessa nova versão da biblioteca é revisar, refatorar, simplificar, acelerar e aumentar a quantidade de imagens com suporte nativo pela biblioteca (Insight Toolkit, 2019). Uma das abordagens utilizadas para acelerar os processamentos da biblioteca é o uso da aceleração via GPU por meio do OpenCL (Liu et al., 2014; Valero et al., 2011).

3.2.4 Intel IPP

A *Intel Integrated Performance Primitives* (IPP) é uma biblioteca de processamento de imagens e sinais produzida pela Intel. É uma biblioteca comercial (o código não é aberto) mas que pode ser baixada gratuitamente, possuindo a vantagem de ter funções otimizadas para as instruções suportadas pelos processadores Intel, que são a MMX, SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVX2, AVX-512 e AES-NI, provendo também funções de compressão de dados e criptografia. A biblioteca está disponível na suíte de aplicativos para paralelização da Intel ou de forma independente (Landré, 2004).

3.2.5 NVIDIA NPP

A *NVIDIA Integrated Performance Primitives* (NPP) é uma biblioteca que provê processamento de sinais, imagens e vídeos acelerados por GPU por meio do CUDA. Assim como a IPP, é uma biblioteca comercial mas que pode ser baixada gratuitamente apenas com o ambiente de computação paralela da NVIDIA. Segundo seus autores, a NPP oferece um desempenho melhor se comparado com a IPP para tarefas paralelas (NVIDIA, 2011). Entretanto, é importante observar que para as comparações citadas, a NPP foi usada em uma GPU e a IPP foi usada em uma CPU, dando vantagem à NPP para tarefas paralelizáveis.

3.2.6 OpenCLIPP

A *OpenCL Integrated Performance Primitives* (OpenCLIPP) é uma biblioteca de código aberto que oferece processamento de imagens e visão computacional. É uma biblioteca similar a IPP e NPP, implementada em OpenCL e C++, funcionando com qualquer dispositivo compatível com OpenCL. Ainda segundo seus autores, o OpenCLIPP é mais veloz que a IPP e é comparável com a NPP, sendo em certos casos, até 2x mais rápida usando a mesma GPU (Moulay; Antoine, 2014).

3.2.7 *scikit-image*

A *scikit* é uma biblioteca para manipulação e processamento de imagens do Python. Essa biblioteca faz parte do ecossistema de bibliotecas científicas do Python e permite que imagens sejam manipuladas por meio de objeto *ndarray* do *NumPy*. Os algoritmos de processamento de imagens não são acelerados por GPU. De acordo com seus autores, a biblioteca possui todas as

suas funcionalidades documentadas e permite que *plugins* específicos sejam escolhidos para a manipulação de imagens (van der Walt et al., 2014).

3.3 APIs para Processamento Paralelo

Nesta seção são apresentadas as APIs CUDA (subseção 3.3.2), *OpenCL* (subseção 3.3.1) e OpenMP (subseção 3.3.3), voltadas para programação paralela.

3.3.1 OpenCL

O *Open Computing Language* (OpenCL) é uma *framework* criada para o desenvolvimento de programas que funcionem em sistemas heterogêneos, permitindo programação em GPUs. Várias empresas de desenvolvimento de *software* e *hardware* se juntaram para desenvolver suas especificações e compatibilizaram seus dispositivos e softwares com essa *framework*, tornando-a o padrão de interface para sistemas heterogêneos e paralelos (Munshi et al., 2011). O OpenCL é detalhado no Capítulo 2, seção 2.2.

3.3.2 CUDA

O *Compute Unified Device Architechture* (CUDA) é uma API para computação paralela e heterogênea proprietária da NVIDIA. Portanto, é uma API proprietária e funciona apenas com os dispositivos desenvolvidos pela empresa. Ela está disponível de forma gratuita e inclui o conjunto de instruções do CUDA para desenvolvimento dos programas de forma semelhante ao OpenCL. O CUDA pode ser usado nativamente em C++ e Fortran (Sanders; Kandrot, 2010).

3.3.3 OpenMP

O *Open Multi-Processing* (OpenMP) é um padrão que consiste num conjunto de diretivas para compiladores e rotinas executadas em tempo de execução que promovem paralelismo de memória compartilhada primariamente ao Fortran (e separadamente ao C e C++) mas podendo ser implementado em qualquer linguagem. O OpenMP foi desenvolvido e é supervisionado por diversos fabricantes e desenvolvedores de *hardware* e *software*, aprovando e desenvolvendo novas versões do software e da especificação do padrão (Dagum; Menon, 1998; OPENMP, 2019).

4

Metodologia

A VGL exige a instalação de dependências e configurações preliminares iniciais. O processo de instalação e os *software* instalados estão descritos na Capítulo 5. Em seguida, com a finalidade de compreender melhor o funcionamento da biblioteca *PyOpenCL* e os *kernels* da VGL, foram desenvolvidos programas Python com propósito de executar alguns *kernels* da VGL de forma isolada (seção 4.2).

Com os exemplos isolados funcionando, os *wrappers* foram implementados (seção 4.3) e finalmente a geração automática dos *wrappers* foi desenvolvida (seção 4.4). Por fim, um teste comparativo de desempenho entre a versão C++ da biblioteca VGL e a versão escrita em Python foi desenvolvida, e descrita na subseção 6.2.1.

4.1 *VisionGL*

Os estudos da biblioteca VGL se dividiram entre a sua instalação (descrita no Capítulo 5), geração automática dos *wrappers*, funcionamento da biblioteca – seus objetos, como eles se comunicam e funcionam – e como os *wrappers* encapsulam os argumentos de entrada para que os *kernels OpenCL* os utilizem de forma apropriada.

Além de reuniões semanais com um dos autores da biblioteca, os trabalhos referenciados nas próximas subseções ajudaram a compreensão desses tópicos, sobretudo na geração automática de código (subseção 4.1.1) e no funcionamento da biblioteca (subseção 4.1.2).

4.1.1 Geração automática de código

Assim como descrito em (Dantas; Barrera, 2011), a construção da biblioteca VGL, lançando as bases do que ela se tornou posteriormente. Nele, os autores demonstraram como foi desenvolvido o gerador automático de código para os *wrappers* GLSL e CUDA. O trabalho também faz um comparativo entre a biblioteca criada com o OpenCV e GPUCV, constatando

um melhor desempenho da VGL em comparação a essas outras bibliotecas nos processamentos testados. A única exceção se deu pela transferência dos dados da memória da GPU para a memória RAM, onde A VGL foi mais lenta que o GPUCV.

Além disso, outros trabalhos descrevem o desenvolvimento de novas funções para a VGL e a geração automática de código para os *kernels* que efetuam o processo dessas funções (Dantas; Leal; Sousa, 2016; Dantas; Leal; Sousa, 2015).

4.1.2 Funcionamento e objetos da biblioteca VisionGL

A VGL é dotada de *kernels* que precisam de objetos específicos da biblioteca para funcionar corretamente. Dessa forma, foram desenvolvidos *kernels OpenCL* para processamento de imagens n-dimensionais, utilizando-se dos objetos de memória *buffer* do *OpenCL* e desenvolvendo os objetos `VglClStrEl` e `VglClShape`, que armazenam as janelas de convolução e o formato da imagem a ser processada, respectivamente. Esses objetos indicam ao *kernel OpenCL* a estrutura da imagem e como os pixels são indexados (Dantas; Leal; Sousa, 2016).

Por outro lado, outro trabalho demonstrou a adaptação do VGL para o processamento de *shaders OpenCL*, com imagens bidimensionais e tridimensionais usando os objetos `image2d` e `image3d` do *OpenCL* (Dantas; Leal; Sousa, 2015).

4.2 Desenvolvimento dos *scripts* Python preliminares

Os qualificadores de variáveis dos *kernels OpenCL* definem a posição de memória no CD em que os dados recebidos serão serão armazenados, bem como o comportamento desses dados. No exemplo mostrado no Código 1, na linha 1, o *kernel* recebe dois argumentos do tipo *image object* do *OpenCL*, um apenas leitura (`__read_only`) e outro escrita(`__write_only`). Esses qualificadores são definidos porque *Image objects* não podem ser lidos e gravados ao mesmo tempo e, se não for explicitado, assume-se que o objeto é apenas leitura (Munshi et al., 2011).

Código 1 – `Kernel vglClCopy.cl`.

```
__kernel void vglClCopy(__read_only image2d_t img_input, __write_only image2d_t
 ↪  img_output)
{
        int2 coords = (int2)(get_global_id(0), get_global_id(1));
        const sampler_t smp = CLK_NORMALIZED_COORDS_FALSE
                            | CLK_ADDRESS_CLAMP
                            | CLK_FILTER_NEAREST;
        float4 p = read_imagef(img_input, smp, coords);
        write_imagef(img_output,coords,p);
}
```

Fonte – VisionGL: <https://github.com/ddantas/visiongl/blob/master/src/CL/vglClCopy.cl>

A Tabela 1 relaciona os qualificadores, suas descrições e a localização na hierarquia de memória do `OpenCL`. Os qualificadores "`__read_only`" e "`__write_only`" são usados quando `Image objects` do *OpenCL* são enviados ao *kenel*, enquanto os qualificadores "`__global`" e "`__constant`" são empregados quando `buffers OpenCL` são utilizados.

Tabela 1 – Tabela de qualificadores de memória do OpenCL usados na VGL.

Qualificador	Descrição	Localização
`__read_only`	Objeto exclusivamente de leitura	*Global memory*
`__write_only`	Objeto exclusivamente de escrita	*Global memory*
`__global`	Objeto de leitura e escrita	*Global memory*
`__constant`	Objeto constante	*Constant memory*

Fonte – Tabela produzida pelo autor com dados de (Stone; Gohara; Shi, 2010).

De forma semelhante, a Tabela 2 relaciona os tipos recebidos pelos *kernels OpenCL* com sua descrição e seu tipo equivalente em Python. Para cada tipo de objeto recebido pelo *kernel*, o Python precisa enviar um tipo que seja equivalente.

Os tipos `image2d_t` e `image3d_t` são encapsulados pelo *PyOpenCL*, de forma que o seu equivalente no Python é o objeto `pyopencl.Image`. O `float`, `int` e `unsigned char` são empregados quando o *kernel* precisa ler algum dado unitário e seus equivalentes são, o `numpy.float32`, `numpy.int32` e `numpy.uint8` respectivamente.

Os tipos `float*`, `unsigned char*` e `char*` são empregados quando o kernel precisa ler arrays e o equivalente a eles é o objeto `pyopencl.Buffer` criado a partir de objetos `numpy.float32`, `numpy.uint8` e `numpy.int8` respectivamente. Os tipos `vglClStrEl` e

`vglClShape` se referem a estruturas do VGL e necessitam de um tratamento específico, que será explicado na subseção 4.2.1.

Tabela 2 – Tabela de tipos do OpenCL usados na VGL.

Tipo	Descrição	Equivalente em Python
`image2d_t`	*Image Object* OpenCL de duas dimensões	`pyopencl.Image`
`image3d_t`	*Image Object* OpenCL de três dimensões	`pyopencl.Image`
`float`	Variável de ponto flutuante de 32 bits	`numpy.float32`
`float*`	Array de ponto flutuante de 32 bits	`numpy.float32`
`int`	Variável Inteira de 32 bits	`numpy.int32`
`unsigned char`	Variável Inteira sem sinal de 8 bits	`numpy.uint8`
`unsigned char*`	Array de variável Inteira sem sinal de 8 bits	`numpy.uint8`
`char*`	Array de variável Inteira de 8 bits	`numpy.uint8`
`vglClStrEl*`	Objeto da VGL para elementos estruturantes	`pyopencl.buffer`
`vglClShape*`	Objeto da VGL para *shape* de imagens	`pyopencl.buffer`

Fonte – Tabela produzida pelo autor.

4.2.1 Tratamento do `vglClStrEl` e `vglClShape`

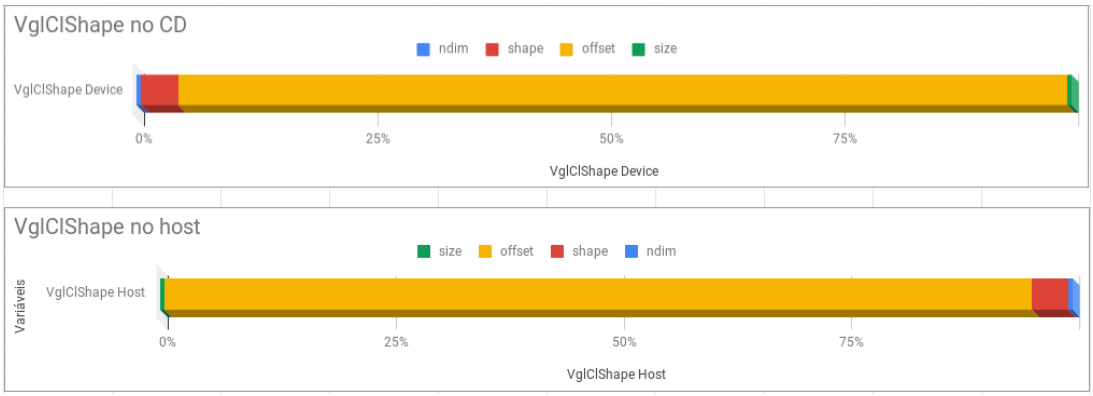
Os tipos `vglClStrEl` e `vglClShape` são, respectivamente, estruturas originadas dos objetos `vglStrEl` e `vglShape` definidas na VGL. Esses objetos são responsáveis por guardar informações sobre o formato e conteúdo do elemento estruturante para uma convolução e o formato (*shape*) de uma imagem. O Código 3, entre as linhas 1 e 13, mostra como essas estruturas são formadas.

O padrão *OpenCL* garante a acessibilidade à memória dos *work-items* e *work-groups* nos moldes do que foi mostrado na subseção 2.2.4. No entanto, quem implementa o *driver OpenCL* do CD tem liberdade para fazê-lo da forma que desejar, sendo impossível garantir que a organização dos *bytes* no *host* será a mesma do *device*.

Em outras palavras, a organização interna dos objetos pode ser completamente oposta, como ilustra a Figura 7, fazendo com que os dados enviados pelo *host* sejam lidos de forma incorreta pelo CD. Para solucionar esse problema, um *kernel* de teste (Código 3) é executado. Esse *kernel* cria uma instância do `vglClStrEl` e do `vglClShape`, obtém a organização das variáveis atribuindo-as a um *array* que é lido pelo host após a execução do *kernel* e armazenado no *host*. A partir de então, quando um *kernel* que precisa dessas estruturas é executado, um *buffer object OpenCL* é criado com os dados organizados de acordo com o que foi obtido pelo *array* de testes anteriormente.

No Código 2, é possível observar o método Python que obtém o *array* com a organização dos *bytes* no kernel (linha 2); o objeto python com os dados da estrutura (linha 3); a declaração

Figura 7 – Exemplo de como a memória pode ser organizada internamente no *host* e no CD.



Fonte – Imagem criada pelo autor.

do *ndarray* Python (linha 5); a cópia dos dados para o *ndarray* na organização correta para o CD (linha 7 a 11); e a transformação do *ndarray* em um *buffer object OpenCL* na linha 13.

Código 2 – Trecho de código Python que faz a cópia dos *bytes* do objeto `VglClStrEl` para um *buffer* com a organização apropriada para o *kernel*.

```
1  def get_asVglClStrEl_buffer(self):
2          struct_sizes = vgl_lib.get_struct_sizes()
3          image_cl_strel = self.asVglClStrEl()
4
5          strel_obj = np.zeros(struct_sizes[0], np.uint8)
6
7          self.copy_into_byte_array(image_cl_strel.data,  strel_obj, struct_sizes[1])
8          self.copy_into_byte_array(image_cl_strel.shape, strel_obj, struct_sizes[2])
9          self.copy_into_byte_array(image_cl_strel.offset,strel_obj, struct_sizes[3])
10         self.copy_into_byte_array(image_cl_strel.ndim,  strel_obj, struct_sizes[4])
11         self.copy_into_byte_array(image_cl_strel.size,  strel_obj, struct_sizes[5])
12
13         return vgl_lib.get_vglstrel_opencl_buffer(strel_obj)
14
15 def copy_into_byte_array(self, value, byte_array, offset):
16         for iterator, byte in enumerate( value.tobytes() ):
17         byte_array[iterator+offset] = byte
```

Fonte – Repositório: <https://github.com/arturxz/TCC/blob/master/vgl_lib/vglStrEl.py>

O Código 3 mostra o *kernel* de teste. Nele, é possível observar o *array* que retorna a organização dos bytes no *kernel* recebido como argumento na linha 14; as declarações das estruturas nas linhas 17 e 18; e as atribuições ao *array* entre as linhas 22 e 34.

Código 3 – Definição das estruturas `vglClStrEl` e `vglClShape`.

```
typedef struct VglClStrEl{
    float data[VGL_ARR_CLSTREL_SIZE];
    int ndim;
    int shape[VGL_ARR_SHAPE_SIZE];
    int offset[VGL_ARR_SHAPE_SIZE];
    int size;
} VglClStrEl;
typedef struct VglClShape{
    int ndim;
    int shape[VGL_ARR_SHAPE_SIZE];
    int offset[VGL_ARR_SHAPE_SIZE];
    int size;
} VglClShape;
__kernel void get_struct_sizes( __global uint *struct_sizes )
{
    const uint global_id = get_global_id(0u)+get_global_id(1u)*get_global_size(0u);
    VglClStrEl strel;
    VglClShape shape;
    uint base;

    if (global_id == 0){
        base = (uint) &strel;
        struct_sizes[0] = (uint) sizeof(strel);
        struct_sizes[1] = (uint) (&strel.data)-base;
        struct_sizes[2] = (uint) (&strel.shape)-base;
        struct_sizes[3] = (uint) (&strel.offset)-base;
        struct_sizes[4] = (uint) (&strel.ndim)-base;
        struct_sizes[5] = (uint) (&strel.size)-base;
        base = (uint) &shape;
        struct_sizes[6] = (uint) sizeof(shape);
        struct_sizes[7] = (uint) (&shape.ndim)-base;
        struct_sizes[8] = (uint) (&shape.shape)-base;
        struct_sizes[9] = (uint) (&shape.offset)-base;
        struct_sizes[10] = (uint) (&shape.size)-base;
    }
    return;
}
```

Fonte – Repositório: <https://github.com/arturxz/TCC/blob/master/vgl_lib/get_struct_sizes.cl>

4.3 Desenvolvimento dos *wrappers* Python

Antes de iniciar o desenvolvimento dos *wrappers*, foi necessário antes finalizar o desenvolvimento da versão Python da biblioteca VGL. Para facilitar a manutenção da biblioteca,

procurou-se escrevê-la na forma mais próxima possível da versão original em C++, mas as diferenças entre as linguagens como a falta de *headers* e os tratamentos especiais da versão python (como o descrito na subseção 4.2.1) fez com que algumas alterações fossem necessárias.

Durante a fase descrita na seção 4.2, os objetos `vglStrEl` e `vglShape` foram criados e estavam muito próximos de seus equivalentes C++, mas o restante ainda não tinha sido feito. Portanto, a primeira coisa feita nessa fase foi o desenvolvimento da biblioteca VGL em sua versão Python (subseção 4.3.1). Em seguida, foram desenvolvidos *wrappers* da forma mais semelhante possível com seus equivalentes em C++ (subseção 4.3.2), com exceção do *wrapper* de imagens binárias (subseção 4.3.3).

4.3.1 Construção da `vgl_lib`

Para suprir a falta dos *headers*, se fez necessária construção de um pacote Python. A VGL oferece uma série de objetos e métodos em arquivos e classes diferentes. De acordo com a documentação, a melhor escolha para esses casos é criar um `namespace package` (van Rossum, 2007).

Neste tipo de pacote, um diretório é criado com o nome do `namespace`; dentro do diretório são adicionados todos os arquivos referentes ao pacote; e cria-se um arquivo chamado `__init__.py`. Esse arquivo é executado implicitamente todas as vezes que o pacote é importado. Ele é responsável por carregar todos os métodos que devem ficar visíveis sob o `namespace`. O `__init__.py` pode importar tanto classes e métodos dos arquivos na mesma pasta que ele quanto pode importar outros pacotes e os deixar visíveis sob a `namespace` (van Rossum, 2007).

O nome do pacote criado para este trabalho foi chamado de "`vgl_lib`", acrônimo para *VisionGL Library*. Com a versão Python do VGL pronta, os *scripts* feitos no passo descrito na seção 4.2 foram alterados para funcionar com a biblioteca, importando-a e utilizando os métodos e objetos dela.

4.3.2 Os *wrappers*

Com a biblioteca implementada e os *scripts* preliminares servindo de base, os *wrappers* podem ser desenvolvidos. Basicamente, a tarefa que o *wrapper* desempenha é receber alguns parâmetros, validá-los, prepará-los para utilização do *kernel*, preparar a execução do *kernel*, executar o *kernel*, recepcionar e retornar o resultado.

Como modelo, foram utilizados os *wrappers* em C++ já existentes, fazendo os métodos de forma equivalente no Python. Apenas os erros não puderam ser tratados da mesma forma que na versão C++, pois quando o *PyOpenCL* detecta algum erro no *OpenCL*, uma exceção é lançada e o fluxo de execução é parado. Mesmo capturando a exceção, o fluxo de execução é finalizado e não é possível tratá-lo de forma semelhante ao C++.

O trecho de código mostrado no Código 4 destaca a validação no *host* de um tipo inteiro para um kernel que recebe o tipo inteiro do *OpenCL*. Se a variável recebida pelo *wrapper* não for do tipo necessário (linha 1), é feita uma tentativa de conversão (linha 4)– se a conversão for bem sucedida, segue o fluxo de execução. Se não, o programa é parado (linha 8) –. As validações para `float`, `int` e `unsigned char` do *OpenCL* são feitas de forma similar.

Código 4 – Exemplo de validação de um tipo numérico.

```
1 if( not isinstance(window_size_x, np.uint32) ):
2     print("vglClConvolution: Warning: window_size_x not np.uint32! Trying to
      ↪  convert...")
3     try:
4         window_size_x = np.uint32(window_size_x)
5     except Exception as e:
6         print("vglClConvolution: Error!! Impossible to convert window_size_x as a
          ↪  np.uint32 object.")
7         print(str(e))
8         exit()
```

Fonte – Repositório: <https://github.com/arturxz/TCC/blob/master/cl2py_shaders.py>

O Código 5 mostra a conversão de um objeto *ndarray* do Python para um *buffer OpenCL*. O processo de conversão para os *arrays* recebidos pelos *kernels OpenCL* são semelhantes ao demonstrado e o processo é independente do tipo de dados que o array armazena.

Código 5 – Exemplo de validação de um array a ser passado ao kernel.

```
1 try:
2     mobj_convolution_window = pyopencl.Buffer(vgl_lib.get_ocl().context,
      ↪  pyopencl.mem_flags.READ_ONLY, convolution_window.nbytes)
3     pyopencl.enqueue_copy(vgl_lib.get_ocl().commandQueue, mobj_convolution_window,
      ↪  convolution_window.tobytes(), is_blocking=True)
4     convolution_window = mobj_convolution_window
5 except Exception as e:
6     print("vglClConvolution: Error!! Impossible to convert convolution_window to
      ↪  pyopencl.Buffer object.")
7     print(str(e))
8     exit()
```

Fonte – Repositório: <https://github.com/arturxz/TCC/blob/master/cl2py_shaders.py>

O *PyOpenCL* encapsula os tipos `image2d_t` e `image3d_t` num único objeto chamado `pyopencl.Image`. A movimentação da imagem entre o CD e o *host* é feita por meio de métodos implementados na `vgl_lib`. Para copiar os dados do *host* ao CD, utiliza-se o método

`vgl_lib.vglClUpload(img)`. Quando os dados contidos na memória são copiados do CD para a memória do *host*, é utilizado o método `vgl_lib.vglClDownload(img)`. Em ambos os casos, `img` é um objeto `vglImage`.

Contudo, o *wrapper* não chama esses métodos diretamente. A `vgl_lib` possui métodos que gerenciam a localização das imagens, que constituem a `vglContext`. Na VGL, existem cinco possíveis localizações (contextos) para as imagens: RAM, *OpenGL*, CUDA, *OpenCL* ou nulo. Como a atualmente a versão Python só trabalha com *OpenCL*, a análise dos contextos se resumiu apenas a três contextos: *OpenCL*, RAM e nulo. O contexto `vglContext` não tem relação com contexto do *OpenCL*. A Tabela 3 mostra os métodos `vglContext` e a descrição do que cada um faz. Portanto, quando um *wrapper* recebe uma imagem do tipo `vglImage`, ele se certifica de que a imagem esteja no contexto do CD, como mostra o exemplo do Código 6, nas linhas 2 e 3.

Tabela 3 – Métodos `vglContext` e suas descrições.

Nome da Função	**Descrição**
`vglIsContextValid(ctx)`	Verifica se o contexto indicado por ctx é único.
`vglIsContextUnique(ctx)`	Verifica se ctx indica apenas um contexto. Retorna 1 se for único e 0 se for múltiplo.
`vglIsInContext(img, ctx)`	Verifica se a imagem está no contexto indicado por ctx. Retorna 1 se estiver e 0 se não estiver.
`vglAddContext(img, ctx)`	Adiciona o contexto indicado por ctx ao contexto da imagem.
`vglSetContext(img, ctx)`	Coloca a imagem no contexto indicado por ctx.
`vglCheckContext(img, ctx)`	Verifica se a imagem está no contexto indicado por ctx. Se não estiver, copia para o contexto indicado.
`vglCheckContextForOutput(img)`	Acusa se a imagem é nula. Retorna 0 se nula e 1 caso contrário.

Fonte – Tabela criada pelo autor.

Ainda, na linha 5 do Código 6, é possível observar o processo de obtenção do *program object OpenCL*. O método `get_compiled_kernel()` está localizado na biblioteca e é responsável por ler o arquivo do *kernel*, verificar se esse kernel já foi compilado e compilá-lo caso não tenha sido. Ele também armazena na memória os *object programs* criados.

Na linha 5, é obtido o *object program* e na linha 6 o *kernel* compilado. Então, nas linhas 8 e 9, os ponteiros para os *Image objects OpenCL* são atribuídos à execução do *kernel* e na linha 11, o *kernel* é enfileirado para execução no CD.

Finalmente, na linha 12, o contexto da imagem é alterado para o *OpenCL*, para explicitar que a última alteração foi feita no CD.

Código 6 – Método `vglCl3dBlurSq3`, que recebe dois objetos do tipo `pyopencl.Image`.

```
1  def vglCl3dBlurSq3(img_input, img_output):
2      vgl_lib.vglCheckContext(img_input, vgl_lib.VGL_CL_CONTEXT())
3      vgl_lib.vglCheckContext(img_output, vgl_lib.VGL_CL_CONTEXT())
4
5      _program = vgl_lib.get_ocl_context().get_compiled_kernel("CL/vglCl3dBlurSq3.cl",
       ↪  "vglCl3dBlurSq3")
6      _kernel = _program.vglCl3dBlurSq3
7
8      _kernel.set_arg(0, img_input.get_oclPtr())
9      _kernel.set_arg(1, img_output.get_oclPtr())
10
11     cl.enqueue_nd_range_kernel(vgl_lib.get_ocl().commandQueue, _kernel,
       ↪  img_input.get_oclPtr().shape, None)
12     vgl_lib.vglSetContext(img_output, vgl_lib.VGL_CL_CONTEXT())
```

Fonte – Repositório: <https://github.com/arturxz/TCC/blob/master/cl2py_shaders.py>

4.3.3 O *wrapper* de imagens binárias

Ainda na referência da linha 11 do Código 6, a função de enfileiramento do *kernel* requer como dados de entrada a fila de comando, o *kernel* a ser executado e o `global_work_size`, que é uma tupla de três valores que define a quantidade total de *work-items* que executarão o *kernel*. Na maioria das imagens, a tupla colocada como argumento é a forma da imagem em largura, altura e profundidade. Se a imagem for bidimensional, a profundidade é 1; e se for unidimensional, a largura é 1.

Imagens binárias digitais – ou imagens binárias – são aquelas onde cada *pixel* é representado por um bit, dando aos *pixels* a possibilidade de assumir duas cores (geralmente preto ou branco) (Kim; Kim; Kim, 2005). Usualmente, cada pixel da imagem é representado por um inteiro de um ou mais *bytes*; entretanto, como a imagem binária possui apenas os valores 1 e 0, é possível comprimir a representação da imagem usando os *bits* para representar os *pixels* da imagem. Dessa forma, uma imagem bidimensional de 16 por 16 *pixels* pode ser armazenada em apenas 32 *bytes* ao invés dos 256 *bytes* que seriam necessários. A VGL utiliza essa abordagem.

Por conta disso, se faz necessário calcular o `global_work_size` de forma diferente do que é feito nos outros *wrappers*. No Código 7, observamos um trecho de código em C++ que carrega os valores apropriados para que a imagem binária seja processada com *kernels* da VGL.

Nesse trecho, caso a imagem a ser processada for uma imagem binária (ou seja, se a cláusula *if* retornar verdadeiro), um cálculo diferente para o `worksize[]` é feito, levando em consideração a altura, largura e profundidade original da imagem, calcula-se o tamanho comprimido em da imagem binária. Essa compressão leva em consideração também o tamanho

de palavra em que se deseja comprimir essa imagem.

Código 7 – Trecho de código do *wrapper* C++ da VGL onde o `global_work_size` é adaptado caso a imagem seja sbinária.

```
1 size_t _worksize_0 = img_input->getWidthIn();
2 if (img_input->depth == IPL_DEPTH_1U) {
3   _worksize_0 = img_input->getWidthStepWords();
4 }
5 if (img_output->depth == IPL_DEPTH_1U) {
6  _worksize_0 = img_output->getWidthStepWords();
7 }
8 size_t worksize[] = { _worksize_0, img_input->getHeightIn(),  img_input->getNFrames()
   ↪  };
9 clEnqueueNDRangeKernel(cl.commandQueue,_kernel,_ndim,NULL,worksize,0,0,0,0);
```

Fonte – Repositório: <https://github.com/ddantas/visiongl/blob/master/src/cl2cpp_BIN.cpp>

4.4 Geração automática dos *wrappers* Python

Na subseção 4.1.1 são citados os geradores de código automático da VGL. Um deles gera o código dos *wrappers* C++, que será utilizado como base para escrita do gerador Python, pois os *wrappers* Python foram feitos baseados nos *wrappers* C++, fazendo com que os *scripts* sejam semelhantes.

A indentação é parte essencial de um programa Python, pois é ela quem define os blocos de execução do programa. Dessa forma, como sugerido por (van Rossum, 2007), a geração automática de código seguirá o padrão de indentação de 4 espaços (4 espaços equivalem a um nível de indentação).

O gerador automático busca pelos *kernels* contidos na VGL e os lê. Com base nos nomes das funções dos *kernels* e nos argumentos recebidos por eles, o gerador automático escreve as funções em Python que recebem as informações necessárias para executar o *kernel*.

O Código 8 mostra três exemplos de cabeçalho de *kernels OpenCL*, enquanto no Código 9 mostra os cabeçalhos das funções *wrapper* escritas pelo gerador automático de código para esses *kernels*. O *wrapper* Python não recebe explicitamente o objeto `vglShape` pois ele está incluso no objeto `vglImage`, que é a imagem de entrada.

Código 8 – Exemplos de cabeçalho dos *kernels* OpenCL.

```
1  // CABEÇALHO DO KERNEL COM IMAGEM ndarray, vglClShape e vglStrEl
2  __kernel void vglClNdConvolution(__global unsigned char* img_input,
3                                   __global unsigned char* img_output,
4                                   __constant VglClShape* img_shape,
5                                   __constant VglClStrEl* window) {
6
7  // CABEÇALHO DO KERNEL COM unsigned char e imagem ndarray
8  __kernel void vglClNdThreshold( __global char* img_input,
9                                  __global char* img_output,
10                                 unsigned char thresh,
11                                 unsigned char top /*= 255*/) {
12
13 // CABEÇALHO DE KERNEL COM OBJETO IMAGEM, ARRAY DE FLOAT E INTEIROS
14 __kernel void vglClConvolution(__read_only image2d_t img_input,
15                                __write_only image2d_t img_output,
16                                __constant float* convolution_window,
17                                int window_size_x,
18                                int window_size_y)
```

Fonte – Repositório: <https://github.com/ddantas/visiongl/>

Código 9 – Exemplos de cabeçalho das funções *wrapper* em Python.

```
1  # CABEÇALHO DA FUNÇÃO WRAPPER PARA KERNEL COM NDARRAY, vglClShape e vglClStrEl
2  def vglClNdConvolution(img_input, img_output, window):
3
4  # CABEÇALHO DA FUNÇÃO WRAPPER PARA KERNEL COM unsigned char*
5  def vglClNdThreshold(img_input, img_output, thresh, top = 255):
6
7  # CABEÇALHO DA FUNÇÃO WRAPPER PARA KERNEL COM OBJETO IMAGEM, ARRAY DE FLOAT E
   ↪  INTEIROS
8  def vglClConvolution(img_input, img_output, convolution_window, window_size_x,
   ↪  window_size_y):
```

Fonte – Repositório: <https://github.com/arturxz/TCC/>

Com os argumentos necessários sendo recepcionados pelos *wrappers*, é necessário fazer a verificação dos tipos recebidos e, se possível, converter os argumentos enviados com tipo errado. O Código 10 demonstra a verificação dos tipos, verificando se a imagem está no formato `ndarray` (linha 2); se é um objeto `vglStrEl` (linha 7); se é um `unsigned char` (linha 12); e se é um `int` (linha 22).

Nos casos dos tipos `unsigned char` e `int` é tentada conversão caso os tipos recebidos pelo *wrapper* não sejam apropriadas. Em todos os casos, se houver algum problema na recepção dos dados, o programa é finalizado.

Código 10 – Exemplos de validação dos dados recebidos no *wrapper* Python.

```
1  # IMAGENS ndarray
2  if( not img_input.clForceAsBuf == vl.IMAGE_ND_ARRAY() ):
3      print("vglClNdCopy: Error: this function supports only OpenCL data as buffer and
       ↪  img_input isn't.")
4      exit(1)
5
6  # OBJETO vglStrEl
7  if( not isinstance(window, vl.VglStrEl) ):
8      print("vglClNdConvolution: Error: window is not a VglClStrEl object. aborting
       ↪  execution.")
9      exit()
10
11 # UNSIGNED CHAR
12 if( not isinstance(top, np.uint8) ):
13     print("vglClConvolution: Warning: top not np.uint8! Trying to convert...")
14     try:
15         top = np.uint8(top)
16     except Exception as e:
17         print("vglClConvolution: Error!! Impossible to convert top as a np.uint8
           ↪  object.")
18         print(str(e))
19         exit()
20
21 # INTEIRO
22 if( not isinstance(window_size_x, np.uint32) ):
23     print("vglClConvolution: Warning: window_size_x not np.uint32! Trying to
       ↪  convert...")
24     try:
25         window_size_x = np.uint32(window_size_x)
26     except Exception as e:
27         print("vglClConvolution: Error!! Impossible to convert window_size_x as a
           ↪  np.uint32 object.")
28         print(str(e))
29         exit()
```

Fonte – Repositório: <https://github.com/arturxz/TCC/>

Alguns objetos precisam ser convertidos para objetos *OpenCL* antes da chamada para execução do *kernel*. O Código 11 mostra exemplos de como garantir que as imagens já têm uma cópia em objeto *OpenCL* (linha 2); como fazer a cópia dos dados de `vglClShape` e `vglClStrEl`

(linhas 5 e 8 respectivamente) para *buffers OpenCL*; e a cópia de *arrays* para *buffers OpenCL*.

Código 11 – Exemplos de conversão prévia dos objetos Python para objetos *OpenCL*.

```
1  # GARANTE QUE A IMAGEM ESTÁ NA MEMÓRIA DO CD
2  vl.vglCheckContext(img_input, vl.VGL_CL_CONTEXT())
3
4  # COPIANDO vglClShape PARA MEMORIA DO CD
5  mobj_img_shape = img_input.getVglShape().get_asVglClShape_buffer()
6
7  # COPIANDO vglClStrEl PARA MEMORIA DO CD
8  mobj_window = window.get_asVglClStrEl_buffer()
9
10 # COPIANDO UM ARRAY PARA O CD
11 try:
12     mobj_convolution_window = cl.Buffer(vl.get_ocl().context, cl.mem_flags.READ_ONLY,
       ↪  convolution_window.nbytes)
13     cl.enqueue_copy(vl.get_ocl().commandQueue, mobj_convolution_window,
       ↪  convolution_window.tobytes(), is_blocking=True)
14     convolution_window = mobj_convolution_window
15 except Exception as e:
16     print("vglClConvolution: Error!! Impossible to convert convolution_window to
       ↪  cl.Buffer object.")
17     print(str(e))
18     exit()
```

Fonte – Repositório: <https://github.com/arturxz/TCC/>

As variáveis que não são *arrays* dos tipos `int`, `float` e `unsigned char` são convertidas automaticamente pelo *PyOpenCL* no enfileiramento do *kernel* para execução, desde que sigam a compatibilidade entre objetos descritos na Tabela 2.

Em seguida, é necessário compilar o *kernel* para execução, fazer a associação dos objetos aos argumentos na chamada de execução do *kernel*, o enfileiramento do *kernel* para execução e sinalizar no objeto de saída – que contém o resultado do processamento – que o ponteiro para o objeto *OpenCL* foi modificado e é o mais atual.

No Código 12 é mostrado na linha 2 um exemplo de como compilar um *kernel*. A `vgl_lib` guarda todos os *kernels* já compilados naquele contexto de forma que, se o *kernel* já foi compilado, ele não será compilado novamente, poupando tempo e recursos computacionais.

A linha 6 a 9 demonstram a associação dos objetos necessários para execução do *kernel* `vglClNdConvolution`. É importante observar que o primeiro argumento tem índice 0, seguindo a sequência numérica até o último argumento. Todos os argumentos que o *kernel* recebe devem ser associados nesse passo.

A linha 12 enfileira o *kernel* para execução no CD. Esse comando é blocante e o fluxo de execução do *wrapper* só continua depois que o *kernel* for executado. Na linha 15, a função `vglSetContext` sinaliza que a referência do objeto OpenCL de `img_output` foi alterado e nas linhas 16 e 17 é feita limpeza nas variáveis temporárias.

Código 12 – Exemplo de finalização da função *wrapper* Python.

```
1  # EXEMPLO DE OBTENÇÃO DO OBJETO DE PROGRAMA DO OPENCL
2  _program = vl.get_ocl_context().get_compiled_kernel("CL/vglClConvolution.cl",
   ↪  "vglClConvolution")
3  _kernel = _program.vglClConvolution
4
5  # EXEMPLO DE ASSOCIAÇÃO DE UM OBJETO A UM ARGUMENTO RECEBIDO PELO KERNEL
   ↪  vglClNdConvolution
6  _kernel.set_arg(0, img_input.get_oclPtr())
7  _kernel.set_arg(1, img_output.get_oclPtr())
8  _kernel.set_arg(2, mobj_img_shape)
9  _kernel.set_arg(3, mobj_window)
10
11 # EXEMPLO DE ENFILEIRAMENTO DE KERNEL PARA EXECUÇÃO PELO CD
12 cl.enqueue_nd_range_kernel(vl.get_ocl().commandQueue, _kernel,
   ↪  img_input.get_oclPtr().shape, None)
13
14 # SINALIZANDO QUE IMAGEM DE SAÍDA FOI MODIFICADA NO OBJETO OPENCL
15 vl.vglSetContext(img_output, vl.VGL_CL_CONTEXT())
16 mobj_img_shape = None
17 mobj_window = None
```

Fonte – Repositório: <https://github.com/arturxz/TCC/>

5

Configuração do sistema de execução

As configurações do computador em que este trabalho foi desenvolvido e os testes foram executados estão descritas na Tabela 4. A última atualização feita no software do sistema data do dia 14 de Abril de 2019.

Tabela 4 – Tabela de configurações do computador de testes e desenvolvimento.

Sistema Operacional	Ubuntu 18.04.2 LTS 64bits
Processador	AMD Ryzen 5 1400 @ 3.20GHz
Memória RAM	8GB DDR4 @ 2400MHz
Placa Mãe	Asus Prime B350-PLUS
Placa de Vídeo	GeForce GTX 1060 iGamer 6G GDDR5 192bits

Alguns problemas de compatibilidade de *drivers* com a VGL foram encontrados. Portanto, foi necessário usar dois *drivers* diferentes para as execuções do *OpenCL* em CPU: para a versão C++ foi usado o *driver AMD APP SDK*, e para a versão Python rodando em CPU, foi usado o driver *pocl*. Também foi usado o *driver* da NVIDIA para execução do *OpenCL* em GPU.

5.1 Configuração da *VisionGL*

A VGL exige a instalação de alguns pacotes para que funcione corretamente, como alguns pacotes do *OpenGL* e os arquivos de desenvolvimento do *OpenCV*. Como pacotes opcionais, o *CUDA*, *OpenCL*, biblioteca `libtiff`, para leitura de imagens do tipo *tiff*, a biblioteca do *OpenCV* e alguns outros [1]. Com a exceção da biblioteca *OpenCV* completa, os outros pacotes podem ser instalados via gerenciador de pacotes do Ubuntu, e essa foi a abordagem empregada para instalação das bibliotecas.

[1] Os pacotes opcionais estão descritos no repositório <https://github.com/ddantas/visiongl>.

Embora a página inicial do repositório mostre as etapas necessárias para instalar os pacotes, ela está com versões antigas de algumas bibliotecas. Para instalar as bibliotecas exigidas foram usados os comandos mostrados no Código 13.

Código 13 – Comando usado para instalar os pacotes exigidos no Ubuntu.

```
# PACOTES REFERENTES AO OPENGL
sudo apt install glew-utils libglew-dev libglew2.0 freeglut3 freeglut3-dev

# ARQUIVOS DE DESENVOLVIMENTO DO OPENCV
sudo apt install libopencv-dev
```

Ainda, como pacotes opcionais, foram instalados o *OpenCL*, bem como a biblioteca de imagens *Tiff*. Para instalar o *OpenCL* são necessários dois softwares diferentes: o *OpenCL Instalable Client Driver (ocl-icd)* e o *driver OpenCL*. O primeiro é referente ao software intermediário entre o *host* e o CD; já o *driver OpenCL* é o que habilita um dispositivo a ser usado como *Computing Device* pelo *OpenCL*. Como neste trabalho será utilizada uma GPU NVIDIA, o *driver OpenCL* é instalado em conjunto com o driver da GPU. Para instalação desses pacotes, foi usado o que é mostrado no Código 14.

Código 14 – Comando usado para instalar os pacotes opcionais no Ubuntu.

```
# PACOTES REFERENTES AO OPENCL
sudo apt install nvidia-driver-390 xserver-xorg-video-nvidia-390 ocl-icd-libopencl1
 ↪  ocl-icd-opencl-dev

# ARQUIVOS PARA BIBLIOTECA DE IMAGENS TIFF
sudo apt install libtiff-dev libtiff5 libtiff5-dev libtiffxx5
```

Com esses pacotes instalados, foi possível compilar e instalar a biblioteca. Seguindo as instruções do repositório, o arquivo `Makefile_linux` foi editado de forma a apontar para os caminhos corretos das bibliotecas, procedimento que foi feito apenas com os *wrappers OpenCL* e com suporte à biblioteca `libtiff`. Dessa forma, os comandos demonstrados no Código 15 foram executados.

Código 15 – Comando usado para instalar a biblioteca VGL.

```
# INSTALAÇÃO DA VisionGL
make -f Makefile_linux all
```

5.2 Configuração do Python

A linguagem de programação Python possui duas versões: Python 2 e Python 3. A versão 2 não é compatível com a versão 3, o que significa dizer que um programa escrito em Python 2 pode não funcionar quando executado num interpretador Python 3. O Python 3 foi lançado oito anos depois que o Python 2, com o objetivo de ajustar as falhas de design que a linguagem trazia (van Rossum, 2007).

Por contar com esses ajustes de design, bem como a maior longevidade de suporte, o Python 3 foi escolhido para o desenvolvimento deste trabalho. O Python 3, bem como seus módulos podem ser instalados por meio do gerenciador de pacotes do Ubuntu na forma descrita pelo Código 16.

Código 16 – Comando usado para instalar o Python e seus módulos.

```
1 # INSTALAÇÃO DO PYTHON
2 sudo apt install python3
3
4 # MÓDULOS DO PYTHON
5 sudo apt install python3-numpy python3-skimage python3-pyopencl python-tifffile
```

O módulo *skimage* é um pacote manipulador de imagens, que também possui algumas funções de processamento de imagens. Esse pacote foi escolhido por permitir que *plugins* personalizados sejam especificados para abertura das imagens. O módulo *skimage* interpreta uma imagem como um objeto `ndarray` do *Numerical Python (NumPy)* que, assim como o *skimage*, faz parte do ecossistema de bibliotecas científicas do Python.

O *NumPy* permite, entre outras coisas, a manipulação de *arrays* n-dimensionais (os `ndarrays`) e o módulo *PyOpenCL* é o módulo Python que dá acesso à API *OpenCL* (subseção 3.1.2). Já o pacote *tifffile* é o módulo que permite aos programas Python abrirem imagens no formato *TIFF*.

5.3 Instalando a biblioteca VGL versão Python

Antes de baixar a versão Python da VGL, é preciso se certificar de que as imagens da *VisionGL* foram baixadas. Para isso, basta entrar na pasta da VGL e executar o *script* fornecido pela própria biblioteca, inserindo os comandos mostrados no Código 17 em um terminal.

Código 17 – Comando usado para baixar as imagens da VGL.

```
# DOWNLOAD DAS IMAGENS PARA TESTES
cd visiongl && sh get_images.sh
```

Com as imagens baixadas, é preciso clonar o repositório com a versão Python da VGL na pasta *visiongl/src*, nomeada como "*py*", como mostra o Código 18.

Código 18 – Comando usado para baixar o repositório da versão Python da VGL.

```
# DOWNLOAD DO REPOSITÓRIO DA VERSÃO PYTHON DA VGL
cd visiongl/src && git clone https://github.com/arturxz/tcc.git py
```

Com a pasta clonada, o último passo é copiar o `Makefile_python` para a pasta raiz da VGL e gerar um *link* simbólico na pasta `visiongl/src` que aponte para *visiongl/src/py/vgl_lib*. Considerando um terminal aberto no diretório raiz da biblioteca VGL. Para fazer isso, os comandos mostrados no Código 19 devem ser inseridos no terminal:

Código 19 – Comando usado para copiar arquivo `Makefile` para localização correta e criação de link simbólico para a biblioteca `vgl_lib`.

```
# COPIANDO ARQUIVO MAKEFILE
cp src/py/Makefile_python ./

# GERANDO LINK SIMBÓLICO
ln -s src/py/vgl_lib src/
```

Por fim, os comandos mostrados no Código 20 mostram como gerar os *wrappers* Python para os *kernels OpenCL* da VGL e como executar o *benchmark* da versão Python.

Código 20 – Comando usado para executar o *benchmark* da versão Python da VGL.

```
# GERANDO WRAPPERS PYTHON
make -f Makefile_python py

# EXECUTANDO O BENCHMARK PYTHON
make -f Makefile_python rundemopy
```

Caso se deseje executar o *benchmark* Python da VGL sem usar o `Makefile_python`, basta executar o arquivo *benchmark_cl.py* ou *benchmark_cl3d.py* na pasta *visiongl/src/py*. Ambos os *scripts* esperam três argumentos:

1. Caminho da imagem que será usada no teste;
2. Quantidade de vezes que cada um dos *kernels* do *benchmark* serão executados;
3. Pasta onde as imagens resultantes do processamento dos *kernels* serão armazenadas.

Dessa forma, o Código 21 exemplifica a execução do *benchmark* para imagens 3D usando a imagem E1154S7I_3d.tif, executando cada *kernel* dez vezes e armazenando os resultados na pasta *tmp/*.

Código 21 – Comando usado baixar as imagens da VGL.

```
# EXECUTANDO BENCHMARK
./benchmark_cl3d.py ../../images/tif/E1154S7I_3d.tif 10 tmp/
```

6

Resultados e discussões

Neste capítulo serão apresentados os resultados alcançados no desenvolvimento do gerador automático de código para o *wrapper* Python na seção 6.1. Além disso, na seção 6.2 são relatados os resultados obtidos nos testes descritos na subseção 6.2.1.

6.1 Geração automática dos *wrappers*

O gerador automático de código para os *wrappers* Python foi implementado nos moldes especificados na seção 4.4. A versão Python dos *wrappers* apresentou uma quantidade significativamente menor de linhas em comparação com a versão C++.

Tabela 5 – Quantidade de linhas nos *wrappers* Python e C++.

Pasta *Kernels*	**Linhas no *wrapper* Python**	**Linhas no *wrapper* C++**	***Relação da quantidade de linhas***
ND	286	592	48,31%
CL	777	1804	43,07%
MM	1924	2878	66,85%

Fonte – Tabela produzida pelo autor.

6.1.1 *Wrapper* para as imagens binárias

Na subseção 4.3.3, foi relatada a especificidade das imagens binárias e a justificativa do porquê de o `global_work_size` precisar ser calculado de forma diferente nos *wrappers* dessas imagens. No entanto, ao fazer o cálculo de forma equivalente no Python, o processamento das imagens resultavam numa imagem preta ou numa imagem parcialmente processada (com apenas uma fração da imagem de saída processada e o restante da imagem preta).

Observando a ineficácia de usar a mesma abordagem utilizada nos *wrappers* C++, tentou-se executar o cálculo da `global_work_size` da mesma forma que os outros *wrappers* (com uma tupla contendo a altura, largura e profundidade da imagem). Nessa abordagem, apenas o procedimento de cópia (`vglClNdBinCopy.cl`) resultou numa imagem de saída apropriada. As demais funções resultaram numa imagem preta.

Nenhuma outra abordagem foi tentada e o *wrapper* Python para processamento de imagens binárias usando a VGL não foi implementado.

6.2 Benchmark dos *wrappers*

Para fazer uma comparação de desempenho entre a versão escrita em Python e C++, os *kernels* destacados na Tabela 6 foram executados em ambas versões processando as mesmas imagens numa quantidade igual de vezes. Cada uma das versões da VGL executará 3 testes: com uma, dez e cem execuções de cada *kernel*. Além de verificar a diferença da velocidade com a qual as linguagens manipulam e pré-processam os dados, será possível observar como o tempo de execução escala com mais execuções de *kernel* sendo enfileiradas ao CD.

Foram medidos os tempos de execução das funções, que consistem na medida do tempo gasto para executar cada função uma, dez e cem vezes. Para medi-lo serão usados os recursos oferecidos pela linguagem (biblioteca *time* do C++ e Python) iniciando-se no momento em que a primeira chamada à função do *wrapper* acontecer e terminando quando a última execução retornar seu resultado. Todos os testes foram feitos cinco vezes e será calculada a média aritmética dos tempos de execução da função.

Para os testes nos *wrappers* de processamentos para imagens 2D foi utilizada a imagem na Figura 16, anexada no Anexo A. Para os *wrappers* de processamentos para imagens 3D foi utilizada uma imagem com sete camadas. Algumas das camadas podem ser observadas nas imagens 14a, 14b, 15a e 15b, localizadas no Anexo A.

Para as janelas de convolução, foram usadas as janelas descritas no Código 26 para imagens 2D no C++; as descritas no Código 27 para imagens 3D no C++; as descritas no Código 24 para imagens 2D no Python; e no Código 25 para imagens 3D no Python. Esses códigos estão anexados no Apêndice B.

Tabela 6 – *Kernels* utilizados para o teste comparativo.

***Kernels* de imagens 2D**	***Kernels* de imagens 3D**	**Descrição**
`CL/vglClBlurSq3.cl`	`CL/vglCl3dBlurSq3.cl`	*Blur* com janela de 3x3 (2D) e 3x3x3 (3D)
`CL/vglClConvolution.cl`	`CL/vglCl3dConvolution.cl`	Convolução com janela de 3x3 (2D) e 3x3x3 (3D)
`CL/vglClConvolution.cl`	`CL/vglCl3dConvolution.cl`	Convolução com janela de 5x5 (2D) e 5x5x5 (3D)
`CL/vglClNot.cl`	`CL/vglCl3dNot.cl`	Inversão de cores da imagem
`CL/vglClThreshold.cl`	`CL/vglCl3dThreshold.cl`	*Threshold*
`CL/vglClCopy.cl`	`CL/vglCl3dCopy.cl`	Cópia CD-CD

Fonte – Tabela criada pelo autor.

O sistema utilizado para os testes é o descrito no Capítulo 5. Serão executados testes com o *wrapper* C++ usando o próprio processador como CD, pois foram encontrados problemas de compatibilidade entre a versão C++ da VGL e o driver *OpenCL* da GPU durante a execução dos testes. Na versão Python, serão executados os testes usando a CPU e GPU como CD.

Um trecho dos códigos utilizados para fazer o *benchmark* dos *wrappers* podem ser observados no Apêndice B. O Código 28 é referente ao *wrapper* C++ em imagens 2D; o Código 29 para o *wrapper* C++ em imagens 3D; o Código 22 para o *wrapper* Python em imagens 2D; e o Código 23 para o *wrapper* Python em imagens 3D.

6.2.1 *Benchmarking* da biblioteca

O *benchmark* dos *wrappers* foi executado com sucesso. Os resultados estão dispostos nas tabelas 7 a 18, mostrando os tempos de execução de cada uma das funções dos *wrappers* executando uma, dez e cem vezes os seus respectivos *kernels*.

A partir dos dados dessas tabelas, foram produzidos os gráficos expostos nas Figuras 8 a 13. Esses gráficos ilustram como o tempo necessário para rodar os *wrappers* aumenta quando mais *kernels* são executados.

Tabela 7 – Tempos das execuções do *kernel* `CL/vglClBlurSq3.cl`.

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.004521s	0.044654s	0.428767s
Wrapper Python CPU	0.002331s	0.003256s	0.024247s
Wrapper Python GPU	0.000642s	0.001970s	0.015230s

Fonte – Tabela criada pelo autor.

Tabela 8 – Tempos das execuções do *kernel* `CL/vglCl3dBlurSq3.cl`.

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.086522s	0.860225s	8.555013s
Wrapper Python CPU	0.002276s	0.003144s	0.034548s
Wrapper Python GPU	0.000705s	0.002010s	0.015923s

Fonte – Tabela criada pelo autor.

Tabela 9 – Tempos das execuções do *kernel* `CL/vglClConvolution.cl` (3x3).

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.004825s	0.047070s	0.480095s
Wrapper Python CPU	0.012267s	0.126593s	1.271658s
Wrapper Python GPU	0.001214s	0.007456s	0.057544s

Fonte – Tabela criada pelo autor.

Tabela 10 – Tempos das execuções do *kernel* `CL/vglCl3dConvolution.cl` (3x3x3).

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.098138s	0.978382s	9.774385s
Wrapper Python CPU	0.157070s	1.017481s	18.155471s
Wrapper Python GPU	0.002564s	0.017481s	0.155471s

Fonte – Tabela criada pelo autor.

Tabela 11 – Tempos das execuções do *kernel* `CL/vglClConvolution.cl` (5x5).

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.012367s	0.122656s	1.220655s
Wrapper Python CPU	0.01100s	0.121902s	1.259818s
Wrapper Python GPU	0.000682s	0.006149s	0.057037s

Fonte – Tabela criada pelo autor.

Tabela 12 – Tempos das execuções do *kernel* `CL/vglCl3dConvolution.cl` (5x5x5).

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.430130s	4.329214s	42.903538s
Wrapper Python CPU	0.360472s	4.16083s	42.26595s
Wrapper Python GPU	0.003585s	0.032449s	0.280147s

Fonte – Tabela criada pelo autor.

Tabela 13 – Tempos das execuções do *kernel* `CL/vglClInvert.cl`.

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.000977s	0.010368s	0.098271s
Wrapper Python CPU	0.000661s	0.001973s	0.024588s
Wrapper Python GPU	0.000648s	0.002184s	0.016350s

Fonte – Tabela criada pelo autor.

Tabela 14 – Tempos das execuções do *kernel* `CL/vglCl3dNot.cl`.

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.006001s	0.048467s	0.504920s
Wrapper Python CPU	0.000765s	0.003212s	0.036966s
Wrapper Python GPU	0.000628s	0.002338s	0.016759s

Fonte – Tabela criada pelo autor.

Tabela 15 – Tempos das execuções do *kernel* `CL/vglClThreshold.cl`.

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.001198s	0.010929s	0.108916s
Wrapper Python CPU	0.001263s	0.002821s	0.027203s
Wrapper Python GPU	0.000701s	0.002607s	0.002355s

Fonte – Tabela criada pelo autor.

Tabela 16 – Tempos das execuções do *kernel* `CL/vglCl3dThreshold.cl`.

Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.005177s	0.052179s	0.509527s
Wrapper Python CPU	0.000803s	0.00763s	0.04340s
Wrapper Python GPU	0.000650s	0.002419s	0.020139s

Fonte – Tabela criada pelo autor.

Tabela 17 – Tempos das execuções do *kernel* `CL/vglClCopy.cl`.

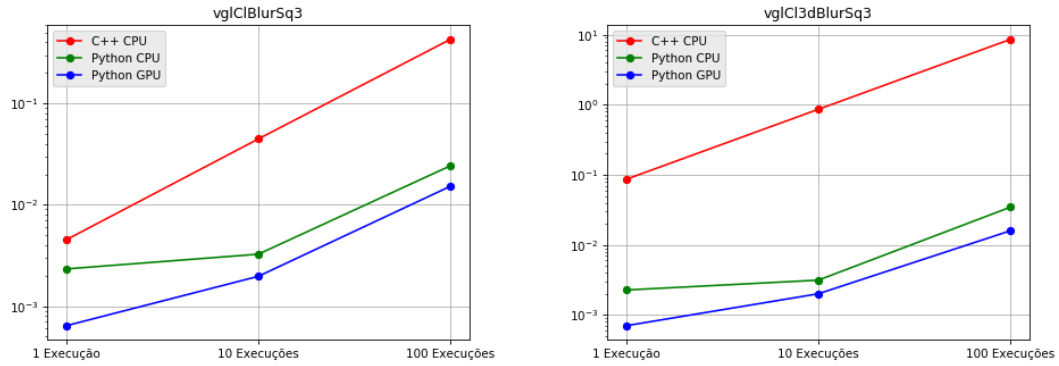
Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.001270s	0.012089s	0.088830s
Wrapper Python CPU	0.001062s	0.002421s	0.023586s
Wrapper Python GPU	0.000690s	0.002355s	0.017978s

Fonte – Tabela criada pelo autor.

Tabela 18 – Tempos das execuções do *kernel* `CL/vglCl3dCopy.cl`.

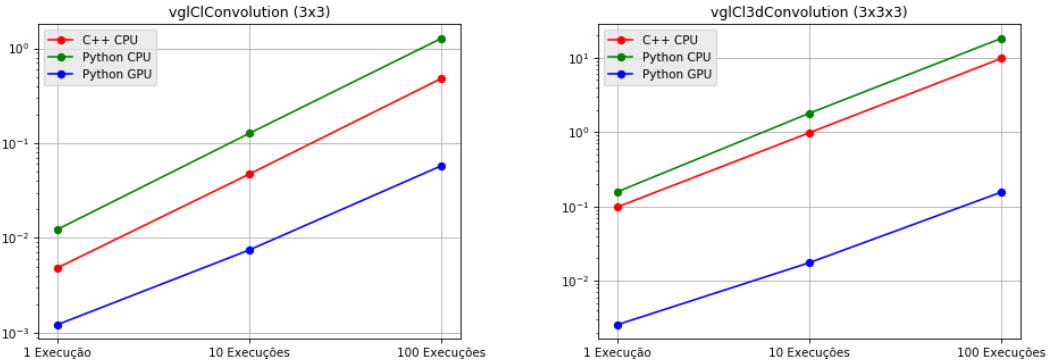
Versão do *wrapper* e CD	**Uma Execução**	**Dez Execuções**	**Cem Execuções**
Wrapper C++ CPU	0.004796s	0.048863s	0.440614s
Wrapper Python CPU	0.001159s	0.004136s	0.036417s
Wrapper Python GPU	0.000628s	0.002320s	0.018694s

Fonte – Tabela criada pelo autor.



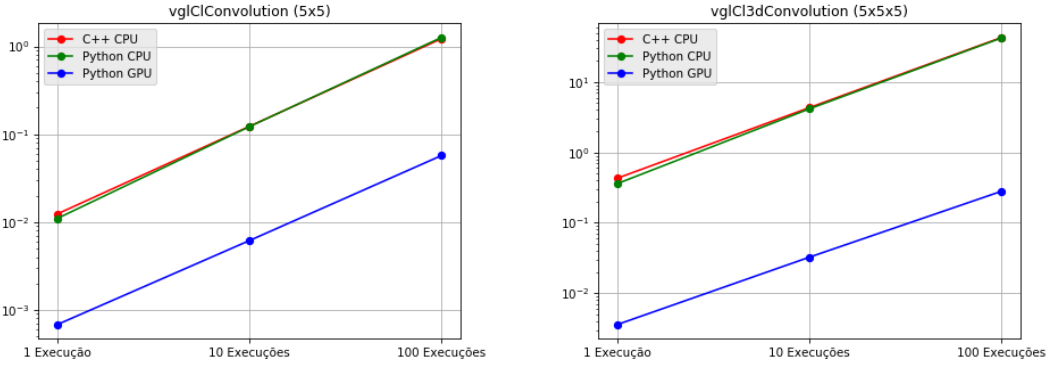
(a) *kernel* `vglClBlurSq3`. (b) *kernel* `vglCl3dBlurSq3`.

Figura 8 – Gráficos dos tempos de execução dos *benchmarks* nas funções de suavização.



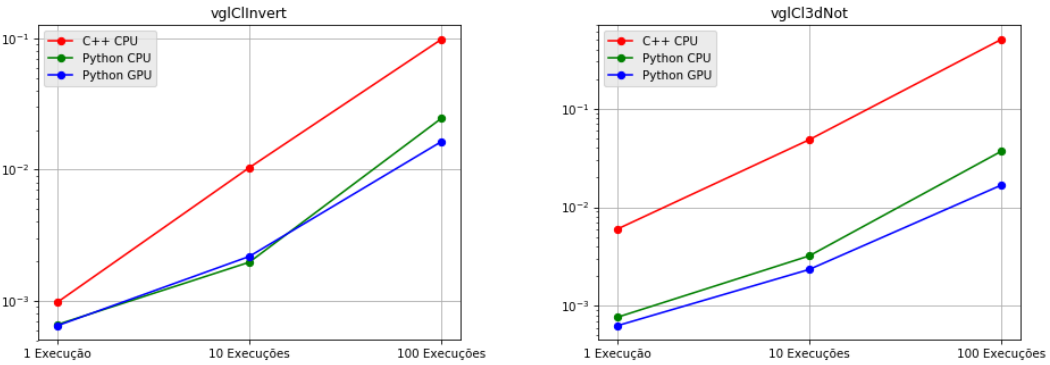
(a) *kernel* `vglClConvolution` (3x3). (b) *kernel* `vglCl3dConvolution` (3x3x3).

Figura 9 – Gráficos dos tempos de execução dos *benchmarks* nas funções de convolução usando janelas de tamanho 3 em suas dimensões.



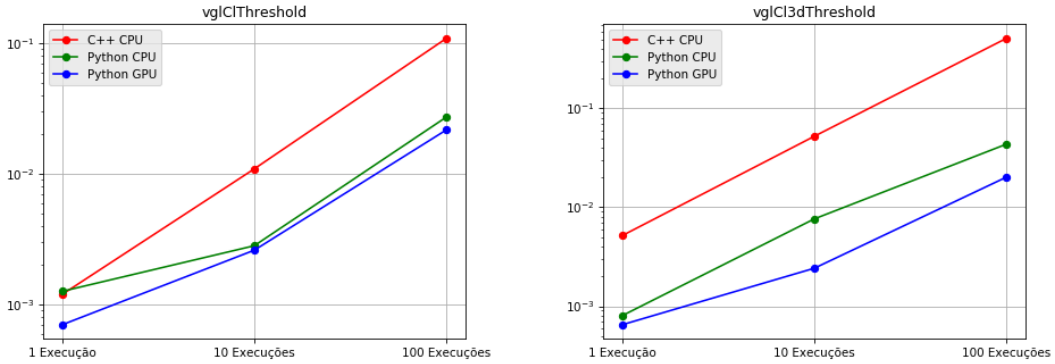
(a) *kernel* `vglClConvolution` (5x5) (b) *kernel* `vglCl3dConvolution` (5x5x5)

Figura 10 – Gráficos dos tempos de execução dos *benchmarks* nas funções de convolução usando janelas de tamanho 5 em suas dimensões.



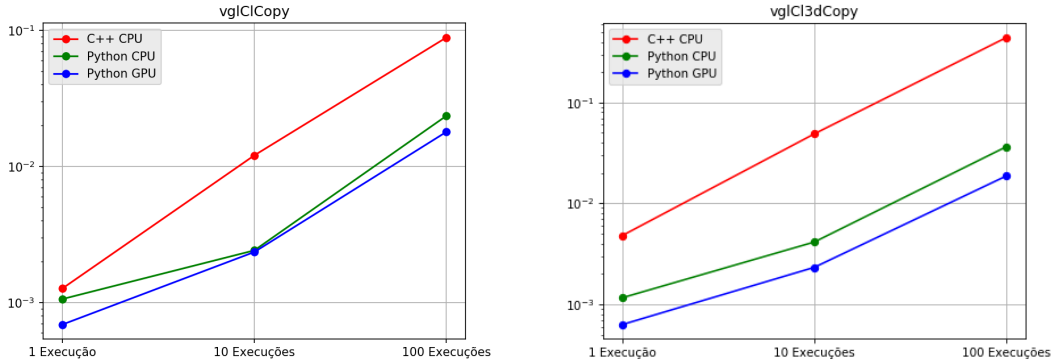
(a) *kernel* `vglClInvert`. (b) *kernel* `vglCl3dNot`.

Figura 11 – Gráficos dos tempos de execução dos *benchmarks* nas funções que calculam o negativo da imagem de entrada.



(a) *kernel* `vglClThreshold`. (b) *kernel* `vglCl3dThreshold`.

Figura 12 – Gráficos dos tempos de execução dos *benchmarks* nas funções de *threshold*.



(a) *kernel* `vglClCopy`. (b) *kernel* `vglCl3dCopy`.

Figura 13 – Gráficos dos tempos de execução dos *benchmarks* nas funções de cópia.

6.3 Discussões

Na maioria dos testes, o *wrapper* Python usando CPU como CD se mostrou mais rápido que o C++ usando CPU como CD. As exceções foram todos os casos das convoluções de imagens pelas janelas de 3x3x3 e 3x3; em cem execuções da convolução por janela 5x5; e em uma execução do *threshold* em imagem tridimensional. Nesses casos, o *wrapper* C++ foi mais rápido que a versão Python.

É importante salientar que, como mostrado no Capítulo 5, foram usados três *drivers OpenCL* para execução dos *wrappers* C++ e Python. Para execução do *wrapper* em C++ usando a CPU como CD foi usado o *driver AMD APP SDK*; para execução do *wrapper* Python usando a CPU como CD, foi usado o *driver pocl*; e para execução do *wrapper Python* usando a GPU como CD foi usado o *driver* fornecido pela NVIDIA. As implementações diferentes dos *drivers*,

em específico os *drivers* para usar a CPU como CD, podem ter influenciado nos resultados dos testes.

O *PyOpenCL* conta com a função de *real-time Code Generation* (RTCG), que faz uma série de otimizações no código do *kernel* enviado para compilação. Essa função de RTCG também leva em consideração o tipo de CD onde o *kernel* será executado (Klöckner et al., 2012). Essa otimização pode justificar que as versões Python da VGL sejam mais rápidas, sobretudo quando a carga de trabalho passa a ser maior no CD, pois as otimizações feitas aos *kernels* ficam evidenciadas.

Ainda, o *wrapper* Python executando em GPU se mostrou mais rápido que na CPU (sendo a única exceção a de dez execuções da inversão de cores em imagem bidimensional, onde usar a CPU como CD no Python mostrou-se pouco mais rápido que usando a GPU). Isso evidencia a superioridade da GPU em executar tarefas paralelas nos testes executados.

7

Conclusão

Este trabalho implementou um gerador automático de código escrito em Perl para a linguagem Python e uma versão Python da parte *OpenCL* da biblioteca *VisionGL*, proporcionando uma melhor compreensão em programação paralela e concorrente, com ênfase na utilização do *OpenCL* e o emprego da linguagem Python com a API *PyOpenCL*.

O Python, com sua API *PyOpenCL*, mostrou-se vantajoso por conseguir executar a maior parte dos processos disponíveis na VGL, com a vantagem de efetuar eventuais otimizações nos *kernels* executados por ele (Klöckner et al., 2012); e os códigos dos *wrappers* escritos em Python foram entre 43,07% e 66,85% menores que a versão escrita em C++. Em contrapartida, este trabalho não conseguiu efetuar os processamentos em imagens binárias na VGL por meio do Python, que se faz necessária para uma compatibilidade completa com a versão original da biblioteca.

Como trabalhos futuros, sugerimos a implementação e execução de testes em mais plataformas comparando o C++ e o Python na GPU, a fim de aferir se o Python leva vantagem em comparação ao C++ na VGL; a implementação de uma abordagem que consiga executar os *kernels* para imagens binárias na VGL; e a implementação de uma versão Python da VGL que utilize CUDA por meio da API *PyCUDA*.

Referências

Bi, D.; Sardella, E. *OpenCL™ Out-of-Order Queue on Intel® Processor Graphics*. 2017. <https://software.intel.com/en-us/articles/opencl-out-of-order-queue-on-intel-processor-graphics>. [Artigo online; Acessado em 10/04/2019.]. Citado na página 22.

Bradski, G.; Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. [S.l.]: "O'Reilly Media, Inc.", 2008. Citado na página 30.

Dagum, L.; Menon, R. OpenMP: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, IEEE, n. 1, p. 46–55, 1998. Citado na página 32.

Dantas, D. O.; Barrera, J. Automatic generation of wrapper code for video processing functions. *Learning & Nonlinear Models*, SBRN, v. 9, n. 2, p. 130–137, 2011. Citado 4 vezes nas páginas 15, 24, 25 e 33.

Dantas, D. O.; Leal, H. D. P.; Sousa, D. O. B. Fast 2d and 3d image processing with opencl. In: *2015 IEEE International Conference on Image Processing (ICIP)*. [S.l.: s.n.], 2015. p. 4858–4862. Citado 3 vezes nas páginas 15, 25 e 34.

Dantas, D. O.; Leal, H. D. P.; Sousa, D. O. B. Fast multidimensional image processing with opencl. In: *2016 IEEE International Conference on Image Processing (ICIP)*. [S.l.: s.n.], 2016. p. 1779–1783. ISSN 2381-8549. Citado 3 vezes nas páginas 15, 25 e 34.

Dlugosch, P. et al. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, v. 25, n. 12, p. 3088–3098, Dec 2014. ISSN 1045-9219. Citado na página 18.

Gamma, E. *Padrões de Projeto: Soluções Reutilizáveis*. [S.l.]: Bookman Editora, 2009. Citado na página 24.

IEEE, C. S. S. C. W. g. o. t. M. S. S. IEEE standard for binary floating-point arithmetic. In: INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. [S.l.], 1985. Citado na página 19.

Insight Toolkit. *Insight Toolkit: ITK*. 2019. <https://itk.org/ITK/project/about.html>. [Online; Acessado em 22-04-2019]. Citado 3 vezes nas páginas 27, 30 e 31.

Intel. *Intel Xeon Phi*. 2019. <https://www.intel.com.br/content/www/br/pt/products/processors/xeon-phi/xeon-phi-processors.html>. [Online; Acessado em 22-04-2019]. Citado na página 26.

ISO, S. C99-iso. In: . [S.l.]: IEC, 1999. Citado na página 19.

Khokhar, A. A. et al. Heterogeneous computing: challenges and opportunities. *Computer*, v. 26, n. 6, p. 18–27, June 1993. ISSN 0018-9162. Citado 2 vezes nas páginas 17 e 18.

Khronos Group. *OpenGL*. 2019. <https://www.opengl.org/>. [Online; Acessado em 22-04-2019]. Citado na página 26.

Khronos Group. *OpenGL ES*. 2019. <https://www.khronos.org/opengles/>. [Online; Acessado em 22-04-2019]. Citado na página 27.

Kim, W.-J.; Kim, S.-D.; Kim, K. Fast algorithms for binary dilation and erosion using run-length encoding. *ETRI journal*, Wiley Online Library, v. 27, n. 6, p. 814–817, 2005. Citado na página 42.

Klöckner, A. et al. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, v. 38, n. 3, p. 157–174, 2012. ISSN 0167-8191. Citado 4 vezes nas páginas 14, 29, 61 e 62.

Landré, J. Programming with intel ipp (integrated performance primitives) and intel opencv (open computer vision) under gnu linux. 2004. Citado na página 31.

Liu, Y. et al. An itk implementation of a physics-based non-rigid registration method for brain deformation in image-guided neurosurgery. *Frontiers in neuroinformatics*, Frontiers, v. 8, p. 33, 2014. Citado na página 31.

Mell, P.; Grance, T. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Security and Technology, 2011. Citado na página 25.

Moulay, A.; Antoine, C. OpenCLIPP: Opencl integrated performance primitives library for computer vision applications. In: *Proc. SPIE Electronic Imaging, Intelligent Robots and Computer Vision XXXI: Algorithms and Techniques*. [S.l.: s.n.], 2014. v. 9025, p. 9025–31. Citado na página 31.

Munshi, A. et al. *OpenCL programming guide*. [S.l.]: Pearson Education, 2011. Citado 10 vezes nas páginas 14, 18, 19, 20, 21, 22, 23, 24, 32 e 34.

Nugteren, C.; Corporaal, H.; Mesman, B. Skeleton-based automatic parallelization of image processing algorithms for gpus. In: IEEE. *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. [S.l.], 2011. p. 25–32. Citado na página 14.

NVIDIA, N. P. P. *NVIDIA Performance Primitives*. 2011. Citado na página 31.

Odersky, M.; Spoon, L.; Venners, B. *Programming in scala*. [S.l.]: Artima Inc, 2008. Citado na página 28.

OpenCV. *OpenCV*. 2019. <https://docs.opencv.org/2.4/modules/ocl/doc/introduction.html/>. [Online; Acessado em 22-04-2019]. Citado na página 30.

OPENMP. *About OpenMP*. 2019. <https://www.openmp.org/about/about-us/>. [Online; Acessado em 22-04-2019]. Citado na página 32.

Padoin, E. L. et al. Performance/energy trade-off in scientific computing: the case of arm big. little and intel sandy bridge. *IET Computers & Digital Techniques*, IET, v. 9, n. 1, p. 27–35, 2014. Citado na página 17.

Passerat-Palmbach, J.; Hill, D. Opencl: A suitable solution to simplify and unify high performance computing developments. *chapter*, v. 8, p. 189–209, 2014. Citado 2 vezes nas páginas 28 e 29.

Pulli, K. et al. Real-time computer vision with OpenCV. *Communications of the ACM*, ACM, v. 55, n. 6, p. 61–69, 2012. Citado na página 30.

Sanders, J.; Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. [S.l.]: Addison-Wesley Professional, 2010. Citado na página 32.

Saxena, S.; Sharma, S.; Sharma, N. Parallel image processing techniques, benefits and limitations. *Research Journal of Applied Sciences, Engineering and Technology*, v. 12, p. 223–238, 01 2016. Citado na página 14.

Silberschatz, A. *Sistemas Operacionais com Java*. [S.l.]: Elsevier Brasil, 2008. Citado 3 vezes nas páginas 17, 25 e 26.

Stone, J. E.; Gohara, D.; Shi, G. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, v. 12, n. 3, p. 66–73, May 2010. ISSN 1521-9615. Citado 4 vezes nas páginas 14, 18, 19 e 35.

Valero, P. et al. A gpu-based implementation of the mrf algorithm in itk package. *The Journal of Supercomputing*, Springer, v. 58, n. 3, p. 403–410, 2011. Citado na página 31.

van der Walt, S. et al. scikit-image: image processing in python. *PeerJ*, v. 2, p. e453, jun. 2014. ISSN 2167-8359. Disponível em: <https://doi.org/10.7717/peerj.453>. Citado na página 32.

van Rossum, G. Python programming language. In: *USENIX annual technical conference*. [S.l.: s.n.], 2007. v. 41, p. 36. Citado 4 vezes nas páginas 29, 39, 43 e 50.

VIGRA. *Generic Programming for Computer Vision*. 2019. <https://ukoethe.github.io/vigra/>. [Online; Acessado em 22-04-2019]. Citado na página 30.

Wang, K. et al. An overview of micron's automata processor. In: IEEE. *2016 international conference on hardware/software codesign and system synthesis (CODES+ ISSS)*. [S.l.], 2016. p. 1–3. Citado na página 18.

Zahran, M. Heterogeneous computing: Here to stay. *Commun. ACM*, ACM, New York, NY, USA, v. 60, n. 3, p. 42–45, fev. 2017. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/3024918>. Citado 3 vezes nas páginas 14, 17 e 18.

Apêndices

APÊNDICE A – Códigos

Código 22 – Trecho de código para uma chamada ao *wrapper* Python para imagens 2D

```
        p = 0
        inicio = t.time()
        while(p < nSteps):
                vglClBlurSq3(img_input, img_output)
                p = p + 1
        vl.get_ocl().commandQueue.finish()
        fim = t.time()

        salvando2d(img_output, img_out_path+"img-vglClBlurSq3.jpg")
        vl.rgb_to_rgba(img_output)
    msg = msg + "Tempo de execução do método vglClBlurSq3:\t\t" +str( round(
     ↪  (fim-inicio), 9 ) ) +"s\n"
```

Fonte – Versão completa do código: <https://github.com/arturxz/TCC/blob/master/benchmark_cl.py>, acesso em 21/04/2019.

Código 24 – Trecho de código que mostra as janelas de convolução usadas para o *benchmarking* dos *wrappers em Python* para imagens 2D

```
convolution_window_2d_3x3 = np.array(((1/16, 2/16, 1/16),
                                      (2/16, 4/16, 2/16),
                                      (1/16, 2/16, 1/16) ), np.float32)
convolution_window_2d_5x5 = np.array(((1/256, 4/256,  6/256,  4/256,  1/256),
                                      (4/256, 16/256, 24/256, 16/256, 4/256),
                                      (6/256, 24/256, 36/256, 24/256, 6/256),
                                      (4/256, 16/256, 24/256, 16/256, 4/256),
                                      (1/256, 4/256,  6/256,  4/256,  1/256) ),
                                       ↪  np.float32)
```

Fonte – <https://github.com/ddantas/visiongl/blob/master/src/demo/benchmark_cl.cpp>, acesso em 21/04/2019.

Código 25 – Trecho de código que mostra as janelas de convolução usadas para o *benchmarking* dos *wrappers em Python* para imagens 3D

```
convolution_window_3d_3x3x3 = np.ones((3,3,3), np.float32) * (1/27)
convolution_window_3d_5x5x5 = np.ones((5,5,5), np.float32) * (1/125)
```

Fonte – <https://github.com/ddantas/visiongl/blob/master/src/demo/benchmark_cl3d.cpp>, acesso em 21/04/2019.

Código 23 – Trecho de código para uma chamada ao *wrapper* Python para imagens 3D

```
p = 0
inicio = t.time()
while(p < nSteps):
        vglCl3dBlurSq3(img_input_3d, img_output_3d)
        p = p + 1
vl.get_ocl().commandQueue.finish()
fim = t.time()

vl.vglSaveImage(img_out_path+"3d-vglCl3dBlurSq3.tif", img_output_3d)
msg = msg + "Tempo de execução do método vglCl3dBlurSq3:\t" +str( round( (fim-inicio),
 ↪  9 ) ) +"s\n"
```

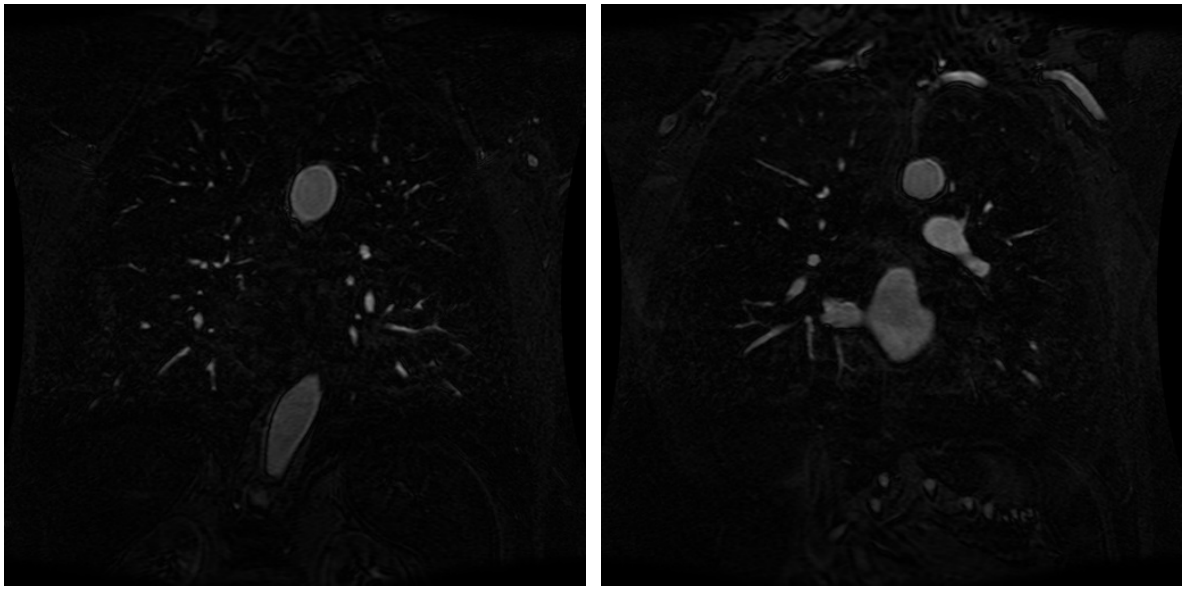
Fonte – Versão completa do código: <https://github.com/arturxz/TCC/blob/master/benchmark_cl3d.py>, acesso em 21/04/2019.

Anexos

ANEXO A – Imagens

Figura 14 – Primeira e segunda camadas da imagem 3D utilizada no *benchmarking* dos *wrappers*

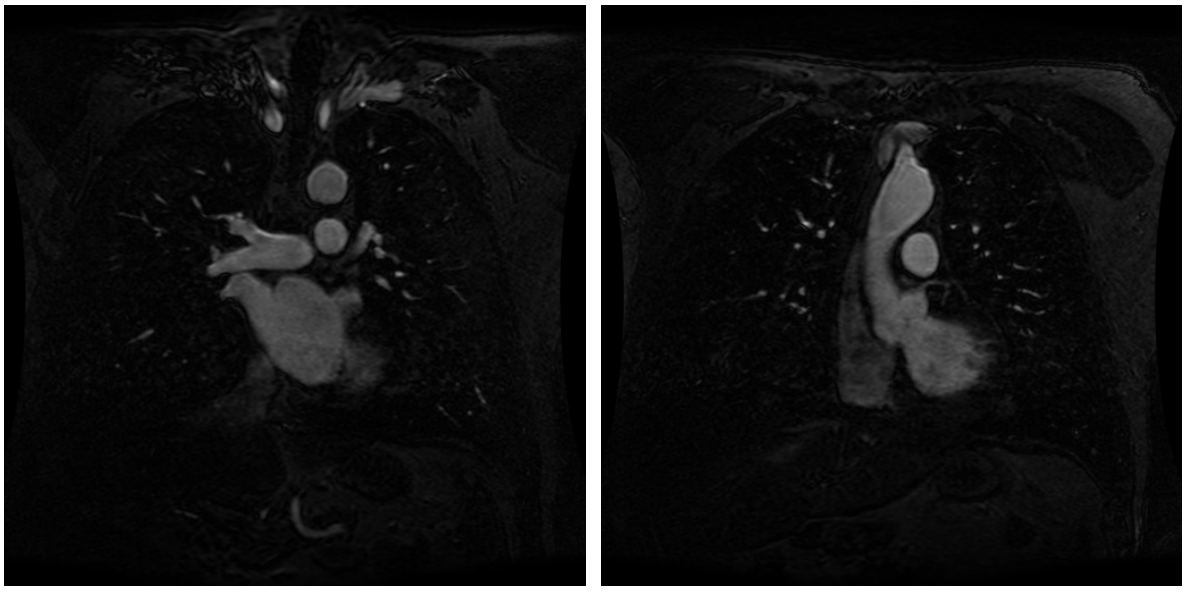
(a) Camada 1 (b) Camada 3



Fonte – <https://github.com/ddantas/visiongl_images/> (Acesso em 20/04/2019).

Figura 15 – Terceira e quarta camada da imagem 3D utilizada no *benchmarking* dos *wrappers*

(a) Camada 5 (b) Camada 7



Fonte – <https://github.com/ddantas/visiongl_images/> (Acesso em 20/04/2019).

Figura 16 – Imagem 2D utilizada no *benchmarking* dos *wrappers*.

Fonte – <https://github.com/ddantas/visiongl_images/> (Acesso em 20/04/2019).

ANEXO B – Códigos

Código 26 – Trecho de código que mostra as janelas de convolução usadas para o *benchmarking* dos *wrappers em C++* para imagens 2D

```
// Convolution kernels
float kernel33[3][3] = {{1.0f/16.0f,2.0f/16.0f,1.0f/16.0f},
                        {2.0f/16.0f,4.0f/16.0f,2.0f/16.0f},
                        {1.0f/16.0f,2.0f/16.0f,1.0f/16.0f},};
float kernel55[5][5] = {{1.0f/256.0f, 4.0f/256.0f, 6.0f/256.0f, 4.0f/256.0f,
 ↪  1.0f/256.0f},
                        {4.0f/256.0f, 16.0f/256.0f, 24.0f/256.0f, 16.0f/256.0f,
                         ↪  4.0f/256.0f},
                        {6.0f/256.0f, 24.0f/256.0f, 36.0f/256.0f, 24.0f/256.0f,
                         ↪  6.0f/256.0f},
                        {4.0f/256.0f, 16.0f/256.0f, 24.0f/256.0f, 16.0f/256.0f,
                         ↪  4.0f/256.0f},
                        {1.0f/256.0f, 4.0f/256.0f, 6.0f/256.0f, 4.0f/256.0f,
                         ↪  1.0f/256.0f}};
```

Fonte – <https://github.com/ddantas/visiongl/blob/master/src/demo/benchmark_cl.cpp>, acesso em 21/04/2019.

Código 27 – Trecho de código que mostra as janelas de convolução usadas para o *benchmarking* dos *wrappers em C++* para imagens 3D

```
// Convolution kernels
// 3x3x3 mask for mean filter
float* mask333 = (float*) malloc(sizeof(float)*27);
for(int i = 0; i < 27; i++)
    mask333[i] = 1.0f/27.0f;
// 5x5x5 mask for mean filter
float* mask555 = (float*) malloc(sizeof(float)*125);
for(int i = 0; i < 125; i++)
    mask555[i] = 1.0f/125.0f;
```

Fonte – <https://github.com/ddantas/visiongl/blob/master/src/demo/benchmark_cl3d.cpp>, acesso em 21/04/2019.

Código 28 – Trecho de código para uma chamada ao *wrapper* C++ para imagens 2D

```
int p = 0;
TimerStart();
while (p < nSteps)
{
    p++;
    vglClBlurSq3(img, out);
}
vglClFlush();
printf("Time spent on %8d Blur 3x3:                %s\n", nSteps,
 ↪  getTimeElapsedInSeconds());

vglCheckContext(out, VGL_RAM_CONTEXT);
sprintf(outFilename, "%s%s", outPath, "/out_cl_blur33.tif");
cvSaveImage(outFilename, out->ipl);
```

Fonte – Versão completa do código: <https://github.com/arturxz/TCC/blob/master/C%2B%2B/benchmark_cl.cpp>, acesso em 21/04/2019.

Código 29 – Trecho de código para uma chamada ao *wrapper* C++ para imagens 3D

```
int p = 0;
TimerStart();
while (p < nSteps)
{
    p++;
    vglCl3dBlurSq3(img, out);
}
vglClFlush();
printf("Time spent on %8d Blur 3x3x3:              %s\n", nSteps,
 ↪  getTimeElapsedInSeconds());

vglCheckContext(out, VGL_RAM_CONTEXT);
sprintf(outFilename, "%s%s", outPath, "/out_cl3d_blur333_%03d.tif");
vglSave3dImage(outFilename, out, imgIFirst, imgILast);
```

Fonte – Versão completa do código: <https://github.com/arturxz/TCC/blob/master/C%2B%2B/benchmark_cl3d.cpp>, acesso em 21/04/2019.