Ricardo José da Silva Martins

Licenciado em Eng. Informática - DI

Coastline Watch - Processamento de imagens com GPU em algoritmos de monitorização da linha da costa

Dissertação para obtenção do Grau de
Mestre em Engenharia Informática

Orientadores: Pedro Abílio Duarte de Medeiros,
Prof. Associado, Universidade Nova de Lisboa
Nuno Duro dos Santos, Engenheiro,
Bluecover Technologies, Lda

Júri:

Presidente:	Doutora Ana Maria Dinis Moreira, Profª Associada, DI - FCT/UNL
Vogais:	Doutor José Manuel Ribeiro Fonseca, Prof. Auxiliar, DEE - FCT/UNL
	Doutor Pedro Abílio Duarte de Medeiros, Prof. Associado, DI - FCT/UNL

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2015

Coastline Watch - Processamento de imagens com GPU em algoritmos de monitorização da linha da costa

Aos meus pais e ao meu irmão

AGRADECIMENTOS

Quero desde já agradecer em primeiro lugar o Prof. Pedro Abílio Duarte de Medeiros (do Departamento de Informática) pelo grande apoio e orientação que me deu durante esta dissertação. Conheço-o há alguns anos e é um excelente professor, aberto a dúvidas e disposto a ajudar os seus alunos. O primeiro contacto que tive com ele foi ao desenvolver o trabalho "Gestão de Recursos em Servidores Equipados com Aceleradores GPUs" para a cadeira de PIICEI e foi um dos trabalhos que gostei de desenvolver e que me fez despertar o interesse pela computação paralela até hoje.
Quero agradecer ao Eng. Nuno Duro (Bluecover Technologies), ao Prof. Gil Gonçalves (INESC Coimbra) e ao Dr. António Alves da Silva (Direção-Geral do Território) pelo suporte que me deram no desenvolvimento do serviço Coastline Watch. Na Bluecover Technologies, aprendi a programar e a obter experiências profissionais na área da Observação da Terra.
Quero agradecer ao Departamento de Informática da FCT-UNL que tem um dos melhores e mais reconhecidos cursos do país, e aos meus pais e o meu irmão pela força e suporte que me deram durante os meus anos da faculdade para conseguir conclui-lo.

RESUMO

As alterações às linhas de costa são um problema sério hoje em dia. As correntes e as ondas do mar provocam alterações à linha devido ao processo de erosão e de acreção. A costa oeste e sul de Portugal continental é banhada pelo Oceano Atlântico e a agitação marítima causa danos a habitações e infraestruturas próximas ao mar. Estas agressões à linha de costa trazem impactos económicos, ambientais e sociais do país.

O Coastline Watch é um sistema colaborativo que pretende monitorizar as alterações à linha da costa causados por fatores naturais (ondas e correntes), fatores humanos ou por alterações climáticas (subida do nível do mar). Tem uma infraestrutura flexível e colaborativa para processar pedidos, analisar e partilhar resultados com outros membros do projeto. Este sistema faz o processamento de um conjunto de imagens de alta resolução obtidas por satélite (e de UAVs), usando os algoritmos pansharpening, MNDWI+K-Means e Vectorização, e armazena os resultados para uma base de dados espacial OGC (Open Geospatial Consortium).

Apesar das vantagens deste sistema, existe o problema do processamento dos dados ser feito por CPU. Imagens de alta resolução demoram muito tempo a processar. Uma solução seria usar GPUs na computação paralela, visto serem uma alternativa para processar imagens e que apresentam uma relação desempenho/preço muito superior ao dos CPUs. Adquirir máquinas com GPUs rápidas e potentes para tirar maior partido da computação paralela não é a solução por causa dos custos. Há soluções *cloud* que têm instâncias com GPUs para a computação paralela.

Esta dissertação começa com a implementação e o desenvolvimento do sistema Coastline Watch, em que o utilizador submete um conjunto de imagens para o sistema na *cloud*. Depois, é descrita uma implementação paralela dos algoritmos mencionados usando GPUs. No final, é feita uma avaliação comparativa dos resultados do processamento paralelo com os do processamento sequencial.

Palavras-chave: gpu, cloud, computação paralela, linha da costa, dados abertos, cuda, mpi, aws

Abstract

Coast lines changes are a serious issue on these days. The sea currents and waves cause coast line changes due to the process of erosion and accretion. The West and South coast of mainland Portugal is bathed by the Atlantic Ocean and the sea turmoil causes damage to houses and infrastructures built near the sea. These agressions to the coast line cause impacts to the economy, to the environment and to the society of the country.

Coastline Watch is a collaborative system aimed to monitor the coastline changes caused by natural processes (waves and tides), by human intervention or global climate changes (sea levels rising). Has a flexible and colaborative infrastructure to process requests, analyze and share the resulting outcomes with other members of the project. This system does the processing of a set of high resolution images from satellite (and UAVs), by using pansharpening, MNDWI+K-Means and Vectorization algorithms, and stores its results to a OGC (Open Geospatial Consortium) spatial database.

Despite of the advantages of this cloud interface, there is a problem with the data processing being done by CPU. High resolution images take much time to process. One solution is to use GPUs on parallel computing, since they are an option to process lots of data, and they have a greater performance/price ratio than the CPU. Buying machines with faster and powerful GPUs to get the best of parallel computing is not the solution due to the costs. There are cloud solutions that have instances with GPUs for parallel computing.

This thesis starts with the implementation and development of the Coastline Watch system, where the user submits a set of images to the system on the cloud. In another part of the dissertation, the implementation of the parallel processing algorithms by using GPUs is described. Also included is a comparison between execution times in sequential and parallel implementations.

Keywords: gpu, cloud, parallel computing, coastline, open data, cuda, mpi, aws

SIGLAS

AWS	Amazon Web Services
CeCILL	CEA CNRS INRIA Logiciel Libre
CNES	Centre National d'Étude Spatiales
CUDA	Compute Unified Device Architecture
DFT	Discrete Fourier Transform
DRY	Don't repeat Yourself
EPSG	European Petroleum Survey Group
ESA	European Space Agency
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FOSS	Free and Open-Source Software
GCP	Google Cloud Platform
GDAL	Geospatial Data Abstraction Library
GIL	Global Interpreter Lock
GMES	Global Monitoring for Environment and Security
GPL	GNU General Public License
GPU	Graphics Processing Unit
GPGPU	General Purpose GPU
GRASS	Geographic Resources Analysis Support System
IaaS	Infrastructure as a Service
MNDWI	Modified Normalised Difference Water Index
MPI	Message Passing Interface
MTV	Model-template-view
MVC	Model-view-controller
NASA	National Aeronautics and Space Administration
NDWI	Normalised Difference Water Index
NIR	Near InfraRed
OGC	Open Geospatial Consortium
OGR	OpenGIS Simple Features Reference Implementation
OpenCL	Open Computing Language
OTB	Orfeo ToolBox
PaaS	Platform as a Service
QGIS	Quantum GIS
RDS	Relational Database Service
SaaS	Software as a Service
SAGA	System for Automated Geoscientific Analyses
SIG (GIS)	Sistema de Informação geográfica (Geographic Information System)
SVM	Support Vector Machine
SWIR (MIR)	Short Wave InfraRed (Middle InfraRed)
TMS	Tile Map Service
UAV	Unmanned Aerial Vehicles
USGS	US Geological Survey
VRT	Virtual Dataset
WGS	World Geodetic System

Conteúdo

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

1.1 Problema a resolver

As linhas de costa sofrem alterações ao longo do tempo por causa do processo de erosão e de acreção que é feito pelas correntes e ondas do mar. Existem outros fatores que permitem aumentar a erosão, como a força das ondas gerada pelo vento e pelas condições climatéricas, as falhas das próprias rochas que permitem que elas se fragmentam em pedaços, e fatores humanos como por exemplo o transporte de sedimentos e rochas do fundo do mar para a terra (dragagem) (Jackson (2014)). É um processo da natureza que tem sido acompanhado com atenção pelos geólogos. Estas alterações à linha de costa trazem impactos negativos para a economia (no turismo e prejuízos por destruição), para o ambiente (acidentes ambientais, como derrames) e para a sociedade do país (edifícios e cidades/locais situados perto do mar).

Portugal, que possui aproximadamente 900 kms de costa banhada pelo Oceano Atlântico, é um país que sofre com as alterações à linha de costa e que tem investido milhões de euros na requalificação da orla costeira (Greensavers - Sapo.pt (2014)) por forma a minimizar os perigos para a população e para as habitações. 85% do PIB português corresponde à produção na zona costeira e 75% da população portuguesa vive nessa zona. (Nuno Duro, Gil Gonçalves (2014))

Por estas razões, considera-se importante unir esforços para se desenvolver um sistema automático e rápido capaz de fazer a monitorização das linhas de costa.

1.2 Soluções existentes

Uma abordagem típica para a monitorização e obtenção das linhas de costa consiste na utilização de uma aplicação SIG [1] como o QGIS. O QGIS é uma aplicação da Open Source Geospatial Foundation (OSGeo) que permite ao utilizador criar, editar, visualizar e analisar informação geoespacial. Para fazer a análise da linha de costa, o utilizador teria que instalar localmente esse software juntamente com bibliotecas para processamento de imagens, e

- Obter, a partir de um repositório, um conjunto de imagens de satélite da zona de costa.
- Seguidamente fazer o processamento das imagens com recurso a algoritmos específicos para diferenciar na imagem as zonas de água com as zonas de terra.
- No final, fazer a vetorização para se obter um ficheiro vetorial com a geometria da linha de costa (.shp) que é depois importado para uma base de dados para poder ser visualizado posteriormente.

Existem também outras ferramentas que fazem essa detecção aplicando algoritmos diferentes, como o UNIBEST-CL+ ou o Delft3D da Deltares (Deltares (2015)) ou o MIKE21 com LITPACK da DHI (DHI (2015)). Mesmo assim, fazer este processamento manual na máquina do utilizador traz problemas para o utilizador: é pesado porque requer investimento na aquisição de uma máquina rápida para processar esses dados; e é demorado sendo necessário a intervenção do utilizador para executar as funções de processamento.

Há um interesse da comunidade científica no desenvolvimento de ideias cloud de monitorização de fatores da natureza como incêndios (FireHub - Copernicus Masters (2014)), contaminação de águas (CyanoLakes - Copernicus Masters (2014)), colheitas (Ceptu - Fieldsense (2014)) ou colónias de abelhas (Beehive Locations - Copernicus Masters (2015)). E também existe o interesse em fazer processamento de deteção das linhas de costa na cloud (Shaima AL-GABLI, Osman Hegazy (2014)). Por essas razões, considerou-se interessante desenvolver uma aplicação colaborativa na cloud para monitorização da linha da costa, daí nascer o projeto Coastline Watch da empresa Bluecover Technologies, Lda. (Nuno Duro, Gil R. Gonçalves, Ricardo Martins, António A. Silva (2015)) no âmbito do qual esta tese está a ser desenvolvida. O Coastline Watch pretende oferecer um serviço de monitorização na *cloud* capaz de detetar as alterações às dunas das praias, e também capaz de determinar e validar o impacto da erosão e acreção ao longo da linha da costa. Usando imagens obtidas por satélite e/ou de UAVs (Unmanned Aerial Vehicles, ou drones pilotados remotamente ou autonomamente) de uma determinada zona que são carregadas pelo utilizador, o sistema faz o processamento automático delas com recurso a algoritmos específicos para o efeito e no final apresenta ao utilizador a linha de costa e outros resultados no mapa.

[1] SIG/GIS: sistema de informação geográfica/geographic information system

Esta abordagem *cloud* tem associadas algumas questões:

- Flexibilidade e escalabilidade da infraestrutura: Como construir um serviço na cloud por forma a ser fácil de manter, gerir, escalar e armazenar grandes volumes de informação?

- Performance e latência no processamento: Como resolver a questão do tempo de processamento sequencial por CPU, sabendo que as imagens a processar são de grande resolução e os algoritmos são demorados e pesados?

- Custos da infraestrutura: Quais são os custos em utilizar e manter a infraestrutura? Existem soluções cloud baratas que podem ser usadas no Coastline Watch?

Contudo, este serviço visa também permitir resolver os tais problemas do processamento, mas nesta dissertação pretende-se ir mais longe e aplicar também a paralelização GPU no processamento. Não foram encontradas referências a processamento paralelo aplicada à área da deteção das linhas de costa com exceção do sistema MIKE21 (MIKE by DHI (2014)); trata-se de um sistema comercial e portanto desconhecem-se os detalhes técnicos. A aplicação da computação paralela ao processamento de imagens é habitual (Craig A. Lee, Carl Kesselman, Stephen Schwab (1996), Yegor V. Pushkin, Rauf Kh. Sadykhov, Leonid P. Podenok, Andrey V. Dorogush, Valentin V. Ganchenko (2014), Mamta Bhojne, Anshu Pallav, Abhishek Chakravarti, Sivakumar V (2013), Rodrigo Alonso, Sergio Nesmachnow (2012), In-Kyu Jeong, Min-Gee Hong, Kwang-Soo Hahn, Joonsoo Choi, Choen Kim (2012)) havendo uma breve descrição de alguns esforços realizados no capítulo 2.

1.3 O que foi feito

O trabalho conducente à dissertação foi incluído no projeto Coastline Watch e teve duas fases. Numa primeira fase, foi desenvolvida a parte do sistema acessível via web que permite o processamento de imagens na cloud. Numa segunda fase, foi feita a paralelização dos algoritmos de processamento de imagens.

1.3.1 Disponibilização do serviço na cloud

A figura seguinte mostra a organização geral da aplicação Coastline Watch especificada de acordo com o documento (Nuno Duro, Gil R. Gonçalves, Ricardo Martins, António A. Silva (2015)). A decomposição visa tornar a aplicação flexível e escalável.

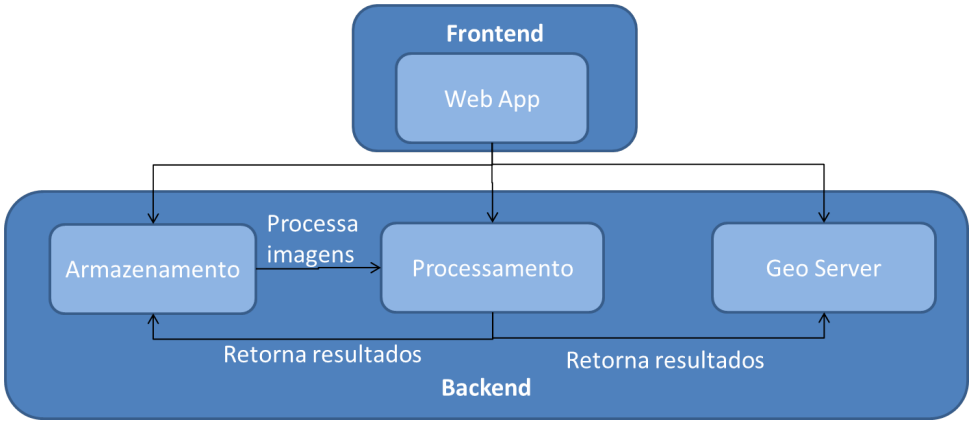

Figura 1.1: Organização geral da aplicação

Seguidamente, descrevem-se brevemente os módulos da aplicação:

- Um *frontend* com uma aplicação web em que o utilizador envia dados, visualiza-os e seleciona o script para os processar.
- Um *backend* com um serviço de armazenamento de imagens, um serviço que executa os scripts definidos para processar as imagens e um GeoServer para armazenar os resultados do processamento (linhas de costa).

A decomposição em módulos tem as seguintes vantagens:

- Melhor tolerância a falhas: a ideia de fragmentar a aplicação em vários serviços permite evitar que ela seja um ponto de falha único para o caso de um dos seus serviços (frontend, backend, geoserver) também falhar.
- Ser escalável: cada serviço faz o seu escalonamento sem comprometer o normal funcionamento.
- Sobrecarga: se a aplicação não tivesse sido particionada, acabaria por ficar sobrecarregada, lenta e sensível a falhas. A sua partição em vários serviços permite contornar essa questão por forma a que elas trabalhem independentemente.

1.3.2 Paralelização

Na segunda fase do trabalho, foi procurado diminuir os tempos de execução dos algoritmos de processamento de imagens através da sua paralelização. Os quatro algoritmos usados são:

- *Pansharpening*: Junção de várias imagens numa imagem melhorada.
- *MNDWI+K-Means*: A aplicação dos algoritmos *Modified Normalised Difference Water Index (MNDWI)* e *K-Means* permite detetar as zonas onde existe água (rios, mares, lagos...)

- Vetorização: A partir da imagem K-Means processada, obtém-se uma linha de costa.

A tabela seguinte mostra os tempos de processamento do script original usando duas diferentes máquinas:

- Portátil da empresa: Intel Core i7-4510U 2Ghz, 1TB disco, 8GB de memória e Windows 8.
- Cluster AWS EC2 Free tier: Intel Xeon 3.3Ghz, 8GB disco, 1GB de memória e Ubuntu 14.04.

	Portátil Empresa	AWS EC2
Criar Raster Virtual	0:00:01	0:00:18
Criar imagem multiespectral de 7 bandas	0:02:16	0:01:27
Criar imagem pancromática de 1 banda	0:00:17	0:00:24
Juntar a multiespectral com a pancromática	0:12:52	0:04:04
Processar MNDWI	0:07:17	0:13:30
Processar K-Means com k=5 + Converter para Uint16	0:02:27	0:02:04
Criar grelha para a vetorização	0:00:21	-
Fazer vetorização	0:07:15	-
TOTAL	0:32:47	-

Tabela 1.1: Tempos de processamento do script original no portátil e AWS EC2

Pelo menos os dois primeiros algoritmos prestam-se à paralelização, pelo que se espera uma redução significativa dos tempos de processamento. A natureza dos processamentos sugere que a utilização de CUDA em GPUs nVidia (Kamran Karimi, Neil G. Dickson, Firas Hamze (2009)) poderá ser um caminho a seguir. Acresce que existem alguns serviços cloud que vendem instâncias (máquinas virtuais) com GPUs nVidia, como a Amazon (Amazon (2014)).

1.4 Contribuições da dissertação

As contribuições desta dissertação são as seguintes:

- Participação na construção do serviço na Cloud: O autor deste documento esteve envolvido na construção do Coastline Watch cuja descrição foi feita em (Nuno Duro, Gil R. Gonçalves, Ricardo Martins, António A. Silva (2015)). O protótipo permitirá avaliar a viabilidade de execução remota do processamento de imagens de satélite.
- Desenvolvimento de versões paralelas dos algoritmos de obtenção das linhas da costa em CUDA: Um teste à execução do script original para o processamento de linhas de costa demorou aproximadamente 33 minutos. A aposta na paralelização desses algoritmos permite conseguir assim reduzir este tempo de processamento. Pretende-se reduzi-lo para valores muito menores, considerando-se um sucesso se o tempo de processamento for inferior a 5 minutos.

1.5 Estrutura do documento da dissertação

Este documento foi estruturado em 5 capítulos:

- O capítulo 1 tem uma pequena introdução do problema a resolver e explica a forma como vai ser resolvida.
- O capítulo 2 faz uma abordagem das tecnologias mais relevantes para o desenvolvimento e disponibilização do serviço Coastline Watch, incluindo as tecnologias disponíveis para utilização de GPUs.
- O capítulo 3 fala das escolhas tecnológicas feitas para a implementação do serviço Coastline Watch. De seguida, é descrita a forma como é feito o processamento sequencial para obtenção da linha da costa, e explica-se o que cada algoritmo de processamento faz e as oportunidades de paralelização existentes.
- O capítulo 4 descreve os tempos de execução do script original e justifica as razões dos resultados. De seguida é feita uma descrição detalhada da implementação do serviço na cloud e da paralelização dos algoritmos. No final, são apresentados os tempos de execução na versão paralela para comparar com os tempos anteriores.
- O capítulo 5 é feita uma avaliação do trabalho realizado nesta dissertação, em que se faz um resumo do trabalho realizado nesta dissertação e uma previsão do trabalho futuro.

CAPÍTULO 2

TRABALHO RELACIONADO

No capítulo anterior, foi feita uma introdução ao projeto com um resumo da aplicação a implementar. Neste capítulo, ir-se-á descrever a aplicação mais em detalhe e também explicar as tecnologias envolvidas, começando por fazer uma breve introdução à *Computação em Cloud*. Para cada um dos módulos da aplicação, foram analisadas diversas alternativas disponíveis para a sua implementação. O final deste capítulo está dedicado à paralelização GPU, em que é feita uma descrição das formas de paralelizar e das frameworks de programação de GPUs existentes.

2.1 Organização da aplicação

A aplicação/serviço está estruturada em dois componentes: um *frontend* (que é a interface do utilizador) e um *backend* (o "motor" do serviço que trata de executar os pedidos do utilizador para processar as imagens). Para compreender melhor essa estrutura, tem-se o esquema ilustrado na figura 2.1.

A interação dos utilizadores com o sistema é feita a partir do componente *frontend* que é uma aplicação *web*. O componente *backend* está estruturado em:

- Infraestrutura de armazenamento: Serve para guardar as imagens de entrada do utilizador (imagens .tif) e os resultados do processamento como ficheiros de saída (ficheiros .shp que são ficheiros vetoriais que armazenam as geometrias)
- Infraestrutura de processamento: Destina-se a executar os *scripts* de processamento de imagens.
- GeoServer: base de dados OGC (Open Geospatial Consortium) para disponibilizar e importar as linhas de costa depois de produzidas pelos scripts/algoritmos de processamento. (ver secção: Armazenamento das linhas de costa)

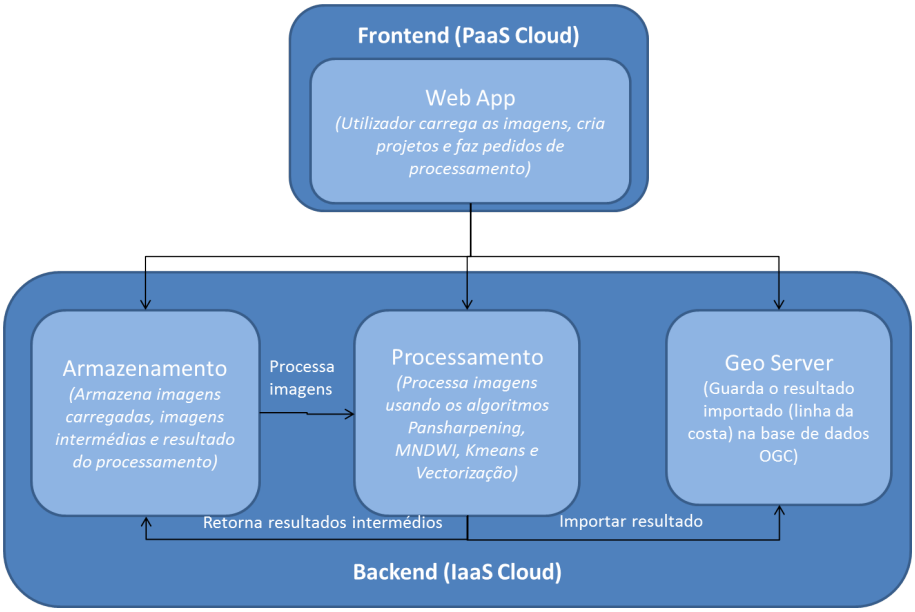

Figura 2.1: Esquema detalhado do serviço implementado

2.2 Computação em Cloud

Como a aplicação em desenvolvimento tem uma grande dependência da cloud, será feita uma introdução dos princípios gerais da cloud e o tipo de serviços por ela oferecidos. A filosofia geral da Computação em Cloud é fornecer aos utilizadores um vasto conjunto de serviços (como a virtualização, armazenamento, base de dados, monitorização, e outros) de forma flexível. A computação cloud baseia-se em dois tipos de tecnologias que tiveram uma grande evolução nos últimos anos:

- Máquinas virtuais/Virtualização: Possibilidade de executar virtualmente múltiplos sistemas operativos dentro de uma máquina física. A computação cloud hoje em dia permite a criação de máquinas virtuais a pedido do utilizador
- Redes de computadores de alta velocidade e baixa latência: a generalização deste tipo de redes permitiu a transferência de grandes volumes de dados entre computadores.

As cloud oferecem as seguintes vantagens:

- Escalabilidade: A cloud pode aumentar ou diminuir os seus recursos dependendo do fluxo no acesso ao serviço. O escalonamento e o aprovisionamento dinâmico de recursos é feito automaticamente, sem intervenção humana.
- Custos: A computação cloud permite às empresas que paguem por aquilo que usufruem. Mais, permite às empresas pouparem nos custos de manutenção e também nos problemas que venha ter no futuro, só quem vende os serviços cloud é que fica encarregue pela manutenção dos seus servidores.

Contudo, as cloud têm também as suas desvantagens:

- Segurança: A segurança depende da responsabilidade do utilizador e do fornecedor do serviço cloud. Por um lado o fornecedor tem a responsabilidade de manter a infraestrutura atualizada e segura, e o utilizador tem que manter seguro o acesso à cloud com a utilização de chaves únicas de acesso. Este problema afeta todas as clouds, mas no caso das clouds privadas há um controlo mais rigoroso no acesso.(Toby Velte, Anthony Velte, Robert Elsenpeter (2010))

- Privacidade: A privacidade é um aspeto importante num serviço cloud. Deve-se confiar num serviço cloud que saiba encriptar dados importantes do utilizador. Caso contrário, esses dados podem ser vistos, roubados e utilizados por qualquer um.

Numa cloud, existem vários tipos de serviços, como ilustrado na figura 2.2:

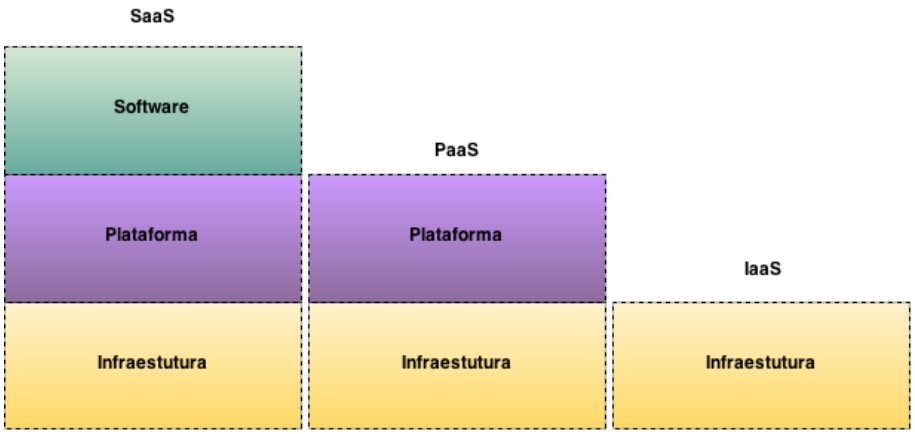

Figura 2.2: Representação simplificada dos tipos de serviços cloud

- Software as a Service (SaaS): Serviço completo que inclui Infraestrutura, Plataforma e Software. Fornece aplicações ao utilizador via web e permite que o utilizador possa usar o software na cloud, sem se preocupar com a instalação, manutenção, ou saber o tipo de infraestrutura do serviço.

- Platform as a Service (PaaS): Serviço que inclui Infraestrutura e Plataforma. Fornece os componentes necessários para o serviço puder executar, como por exemplo base de dados, frameworks, runtime e sistema de operação. O utilizador tem que se preocupar com a implementação e manutenção das aplicações, sem precisar de saber o tipo de infraestrutura usado no serviço.

- Infrastructure as a Service (IaaS): Serviço que só tem a Infraestrutura para o funcionamento dos componentes do serviço, como por exemplo máquinas virtuais, servidores, armazenamento e rede. O utilizador tem que lidar com a gestão e instalação dos componentes do serviço.

2.3 Soluções existentes

Nesta secção, é feita uma abordagem das tecnologias relevantes presentes nas seguintes vertentes: Backend Cloud (para o armazenamento e processamento dos dados), Frontend Cloud (para interação com utilizador), Ambientes de programação (Framework), Processamento de imagens e Paralelização da aplicação com GPUs. Em cada uma delas, foram escolhidas algumas das tecnologias disponíveis e que têm interesse para o projeto.

2.3.1 Backend Cloud

Como dito no início deste capítulo, o Backend está estruturado em três componentes: armazenamento (para armazenar imagens e outros ficheiros intermédios), o processamento (para processar as imagens e gerar os resultados) e um GeoServer (ver secção: Armazenamento das linhas de costa). A interação do serviço com o utilizador é feita a partir da frontend (enviar pedidos, fazer upload de imagens, submeter scripts de processamento).

Os seguintes critérios vão ser aplicados na escolha do Backend Cloud:

- Natureza do serviço oferecido, de preferência IaaS para que seja possível ter uma instância virtual para processamento, um servidor de armazenamento para as imagens e uma base de dados OGC para importar e apresentar os resultados ao utilizador no frontend depois do processamento.
- Custo, pretendendo-se que possa ser gratuito para um uso inicial
- Suporte de instâncias/máquinas virtuais com GPUs (pagas)
- Tipos de API para desenvolvimento suportadas. Pretende-se que sejam fáceis de usar e que suportem um conjunto alargado de linguagens de programação
- Documentação disponível

Das clouds do tipo IaaS estudadas, pode-se destacar 3: a Amazon Web Services (AWS), a Google Cloud Platform e a IBM SmartCloud Enterprise.

2.3.1.1 Amazon Web Services

A Amazon Web Services (AWS) é um serviço Cloud fornecido pela Amazon. Foi uma das primeiras empresas a lançar o conceito de Cloud, em 2006, e que hoje continua a crescer. A origem do AWS pode ser associada ao artigo que Benjamin Black e Chris Pinkham escreveram em 2003 a descrever uma infraestrutura da Amazon completamente automatizada, normalizada e que dependia dos serviços web para executar serviços como armazenamento (Benjamin Black (2009)). E no final chegaram a propor uma ideia de venderem esses serviços ao público por forma a gerarem lucro à Amazon. Foi daí então que surgiu o AWS.

Os serviços mais usados no AWS são o EC2 (Amazon Elastic Compute Cloud) e o S3 (Amazon Simple Storage Service). O EC2 é um serviço escalável para a computação e processamento de dados, fácil de usar e muito rápido para inicializar instâncias (usa o Xen para a virtualização).

No EC2, existem 7 tipos de instância de processamento divididas em 5 classes: Utilização normal (T2 e M3), Otimizado para Computação (C3), Otimizado para Memória (R3), Otimizado para Armazenamento (I2 e HS1) e, o mais importante para este trabalho, Computação Paralela GPU (G2). Estas instâncias possuem diferentes capacidades de memória e de armazenamento, suportam Amazon Linux, Debian, CentOS e Windows e todas elas correm em CPUs Intel Xeon. Só o G2 é que possui uma GPU da nVidia com 1536 cores e 4GB de memória.

Existem 3 modalidades diferentes de pagamento no EC2:

- On-demand: em que o utilizador paga por hora pelo que utiliza.
- Reserved Instances: em que paga uma vez por 1 ou 3 anos ficando mais barato o seu preço por hora.
- Spot Instances: em que paga o serviço pelo valor no spot price, que varia consoante os recursos disponíveis.

O S3 é um serviço dedicado ao armazenamento que permite cifrar os dados (usando SSL ou encriptação do lado do cliente), sistema de notificações de eventos e monitorização, disponibilidade do serviço muito elevada e uma baixa taxa de perda de dados. Ao contrário de outros serviços que utilizam mecanismos pesados para verificação e reparação de dados, o AWS possui um mecanismo simples que calcula os checksums no tráfego de envio e receção de dados por forma a detetar os pacotes de dados corruptos e garantir assim a fiabilidade dos mesmos.

No S3, existem 3 tipos de armazenamento: Standard Storage (o normal, replicado, suporta perda de dados com concorrência), Reduced Redundancy Storage (barato, garante durabilidade, não é replicável como no standard) e Glacier Storage (optimizado para dados a que se faz acesso com pouca frequência). Os preços são taxados ao GB e variam também da quantidade de armazenamento utilizado por mês.

O AWS disponibiliza outros serviços como o EBS, DynamoDB, Elastic MapReduce ou RDS que não se descrevem aqui. Disponibiliza também um marketplace para os utilizadores puderem instalar software, no caso de pretenderem transformar o AWS num serviço tipo SaaS.

Existe o "free-tier" para os utilizadores puderem experimentar gratuitamente todos os serviços da AWS por um ano, e que oferece uma instância t2.micro, 750 horas EC2 por mês, 5GB de armazenamento S3, 30GB de espaço no EBS, e outras funcionalidades. (Amazon (2014a)).

A AWS disponibiliza uma API que permite a integração da nossa aplicação com os serviços da Amazon e que suporta as seguintes linguagens: Python, Ruby, Java, PHP,

.NET, Node.js e Javascript. A documentação do AWS é completa, incluindo exemplos de implementação de serviços com várias APIs.

2.3.1.2 Google Cloud Platform

O Google Cloud Platform (GCP) é um serviço Cloud da Google e que é usado em alguns dos seus serviços, como a pesquisa Google, YouTube e Gmail. Em 2008, a Google lançou o App Engine, um serviço PaaS que permite o desenvolvimento e alojamento de aplicações web na Google. Em 2013, o GCP foi apresentado ao público integrando serviços como o Google App Engine, Google Compute Engine, Google Cloud Storage, Google Cloud Datastore, Google Cloud SQL e outros. (Toby Velte, Anthony Velte, Robert Elsenpeter (2010))

O GCP possui o Google Compute Engine e o Google Cloud Storage, que são serviços semelhantes ao EC2 e S3 da Amazon. O Google Compute Engine está focado na computação e é um serviço que permite os utilizadores lançarem máquinas virtuais ”on-demand”, e o Google Cloud Storage está focado no armazenamento dos dados.

O GCP disponibiliza, no seu Compute Engine, 5 tipos de instâncias divididas em 4 classes: Standard (n1-standard), High Memory (n1-highmem), High CPU (n1-highcpu) e Shared Core (f1-micro e g1-small), mas não possui nenhuma instância dedicada à computação paralela GPU. Estas instâncias suportam CentOS, Debian, RHEL e Windows e usam KVM para executar as imagens na arquitetura x86 64-bit. O KVM é conhecido pelo pouco *overhead* no lançamento de instâncias da máquina virtual. (Google (2014b)).

Em termos de pagamento, é o mais simples, mas pouco cómodo: todas as instâncias são cobradas por um mínimo de 10 minutos, depois desse tempo é cobrado ao minuto.

Por fim, tal como o AWS, a GCP possui também uma SDK que suporta só Python (Google (2014a)). Relativamente à documentação disponibilizada, é fraca, tem pouca informação disponibilizada e que mostrou ser mais confusa que a da AWS.

2.3.1.3 IBM SmartCloud Enterprise

A IBM oferece aos utilizadores e empresas serviços cloud PaaS, IaaS e SaaS que podem ser privadas, públicas ou híbridas. Nos anos 70, numa altura em que a computação era feita através de mainframes, a IBM foi pioneira na criação de máquinas virtuais por forma a tirar maior partido das mainframes e diminuir assim os custos. A partir dos anos 90, começaram a ser lançados servidores com virtualização. Em 2007, com o aparecimento das clouds, a IBM apostou numa cloud com suporte de virtualização.

O SmartCloud Enterprise tem serviços IaaS integrados de armazenamento e de processamento. A IBM adquiriu a SoftLayer para apostar no desenvolvimento de novas soluções cloud.

O SmartCloud Enterprise disponibiliza 9 tipos de instâncias (servidores virtuais) divididas em 2 versões: 32-bit (copper, bronze, silver e gold) e 64-bit (copper, bronze, silver, gold, platinum). Estas instâncias podem ser executadas em SUSE, RedHat ou em

Windows, sendo que as diferenças entre instâncias estão no número de CPU virtuais, no tamanho da memória virtual e no tamanho de armazenamento da instância. Este serviço fornece instâncias com suporte à computação paralela com GPU. (Softlayer (2015))

A IBM fornece documentação para a API REST do SmartCloud (*IBM SmartCloud Entry* (2013)).

2.3.2 Frontend Cloud

Como em muitos sistemas, numa comunicação entre o utilizador e o sistema, é necessário haver um frontend. Um frontend é uma abstração do sistema com uma interface gráfica para o utilizador. No nosso sistema, o frontend é uma página web alojada num servidor virtual residente na cloud.

Os seguintes critérios vão ser usados na escolha do Frontend Cloud:

- Uma cloud que seja PaaS
- Gratuita para um uso inicial
- Fácil extensão dos serviços oferecidos (addons)
- Suporte de um conjunto alargado de ambientes (frameworks) de programação.

Existem bastantes serviços cloud PaaS disponíveis com diferentes características, mas para este projeto só foram escolhidas 5: Heroku, Bitnami, Google App Engine, Engine Yard e IBM Bluemix.

2.3.2.1 Heroku

O Heroku é um serviço cloud PaaS desenvolvido em 2007 que permite criar aplicações na hora. Esta plataforma utiliza AWS EC2/S3 como IaaS, suporta linguagens como Ruby, Node.js, Python, Java e PHP e utiliza PostgreSQL como aplicação de base de dados. Tem suporte a add-ons (alguns pagos, outros gratuitos) que podem ser obtidos e integrados no projeto.

O Heroku tem contentores chamados ”dynos” que processam os pedidos do utilizador. São máquinas que estão a correr por detrás do serviço e que processam dados e tarefas definidas pelo utilizador. Estas dynos correm em AWS, e trazem vantagens como por exemplo escalonamento, gestão de carga e a gestão de pedidos. (Neil Middleton, Richard Schneeman (2013))

Existem 3 tipos de Dynos: 1X (com 512MB RAM), 2X (com 1024MB RAM) e Performance Dyno (6GB RAM, baixa latência e melhor qualidade de serviço). Os preços deste serviços são definidos de acordo com o tipo de ”dyno” a usar e o número de recursos para escalar.

Este serviço disponibiliza uma versão gratuita com uma dyno 1X, 750h mensais de utilização e uma base de dados PostgreSQL com limite máximo de 10 mil entradas.

2.3.2.2 Bitnami

O Bitnami, apesar de ser uma empresa mais focada na produção e fornecimento de pacotes de software para instalação, também fornece um serviço cloud PaaS de fácil implementação, em que o utilizador seleciona o software que deseja instalar, a plataforma faz a sua instalação e põe a correr no servidor.

O Bitnami utiliza AWS EC2/S3 como serviço IaaS (havendo também a possibilidade de utilizar Google Cloud Platform ou Microsoft Azure), suporta linguagens como Ruby, Python, nodejs e PHP e utiliza MySQL como aplicação de base de dados. Como dito anteriormente, este serviço tem suporte para addons (ou ”stacks"que são pacotes completos de instalação e que existem para Ruby e Python) que podem ser instalados a partir da dashboard da Bitnami (Amazon (2011)).

Um dos problemas do Bitnami é o preço pela utilização do serviço. Como o Bitnami usa AWS, o utilizador tem que pagar o serviço Bitnami, mais o serviço fornecido pela AWS, o que acaba por ser caro. Contudo, existe na Bitnami uma opção free-tier, com as mesmas condições oferecidas pela AWS (t2.micro, 12 meses utilização livre), mas se o utilizador quiser mais do que isso, tem que pagar.

2.3.2.3 Google App Engine

O Google App Engine (GAE) é uma cloud PaaS da Google lançada em 2008 e que permite o desenvolvimento e alojamento de aplicações web.

O GAE utiliza o Google Cloud Platform como serviço IaaS desde 2012 e suporta linguagens como Python, Java, PHP e Go (e frameworks como Django, Flask, Spring e webapp2). Relativamente à base de dados, usa o Google Cloud SQL com suporte para MySQL. Não existe qualquer informação relativa à instalação de *addons*.

Relativamente aos preços, o Google tem uma política diferente dos seus concorrentes. Existe uma quota livre diária, mas a sua utilização diária está limitada a 28 horas-instância, 50 mil operações e 1 GB de armazenamento na base de dados, 5GB de armazenamento na cloud e outras restrições. Se o utilizador exceder essa quota, tem que pagar.

2.3.2.4 Engine Yard

O Engine Yard é uma cloud PaaS desenvolvida em 2006. É semelhante ao Bitnami, em que o utilizador seleciona o software que deseja instalar, a plataforma faz a sua instalação e põe a correr no servidor.

Esta plataforma utiliza AWS EC2/S3 ou Microsoft Azure como IaaS, suporta linguagens como Ruby, PHP, Java e Node.js e disponibiliza como sistemas de base de dados o PostgreSQL ou MySQL ou Riak. (Engine Yard (2014a)) Este serviço permite a instalação de *addons* através de um sistema de stacks, em que para cada linguagem existe um pacote com aplicações que o utilizador pode escolher e instalar na dashboard da cloud.

Relativamente aos preços, tem o mesmo problema do Bitnami. O utilizador para usufruir do serviço tem que pagar o serviço da Engine Yard, mais o serviço AWS, o que

fica caro no final do mês. Não há de momento uma versão free, mas existe uma versão de demonstração para os utilizadores, só que está limitado à utilização grátis de uma instância do tipo medium por 500 horas (Engine Yard (2014b)).

2.3.2.5 IBM Bluemix

O Bluemix é uma cloud PaaS desenvolvida pela IBM que oferece uma solução que permite o desenvolvimento e implementação de aplicações na cloud. Umas das inovações interessantes dessa cloud é a possibilidade de fazer desenvolvimento dentro da cloud, ou seja, o utilizador pode importar o código, editar e correr dentro da cloud. (Thoughts on Cloud (2014))

Esta plataforma utiliza o IBM SoftLayer como serviço IaaS e suporta a instalação de addons a partir do catálogo do Bluemix. Existem *addons starters* que instalam pacotes e runtimes para linguagens como Java, Javascript, Node.js e Ruby (algumas são free por 30 dias), serviços Watson, Mobile, DevOps, Web, Data Management (de momento traz suporte a aplicações de base de dados como ElephantSQL, SQL Database, existe versões experimentais do mongoDB, MySQL e postgreSQL) e outros mais.

O modo de pagamento desta plataforma consiste em pagar pelo uso do runtime, mais pelo uso do serviço/addon. Um runtime é cobrado à unidade de GB-hours, que define o tempo e a memória que a aplicação utiliza e é gratuita durante o período de demonstração, até ao máximo de 2GB por instância. Depois desse período o serviço fornece 375 GB-horas gratuitas por mês por instância ou distribuídas pelas várias instâncias. Por cada serviço é cobrado por um valor fixo mensal. (Bluemix (2014))

2.3.3 Ambientes de programação (Framework)

Implementar um site para o Frontend "manualmente"acaba por ser uma tarefa complicada para o programador. Existem frameworks de desenvolvimento de sites que possuem mecanismos para poupar tempo na implementação e que seguem a arquitetura MVC (model-view-controller) (Bernardo Pires (2014)). Os critérios de escolha de uma framework são o suporte de uma linguagem fácil, poderosa e rápida de aprender, uma boa qualidade de documentação e um bom suporte por parte dos serviços cloud. A seguir descrevem-se em detalhe dois desses frameworks: Ruby on Rails e Django.

2.3.3.1 Ruby on Rails

Ruby on Rails (ou Rails), é uma framework web em Ruby (Rails (2014)). Tem como princípios:

- O DRY (don't repeat yourself) para permitir ao utilizador obter a informação a partir de uma fonte (tabela da base de dados, objeto), sem ter que repetir código e permitir assim um rápido desenvolvimento.

- O CoC (convention-over-configuration) que permite ao Rails definir automaticamente convenções para os nomes do modelo, controlador e vistas com base no nome do objeto criado pelo utilizador, permitindo assim uma melhor organização na estrutura de ficheiros e acelerar a sua implementação.
- A reutilização, rápida implementação e integração de componentes.

O Ruby on Rails utiliza a arquitetura MVC para a integração de componentes:

- Os modelos (Models) permitem definir o acesso aos dados, validações, relações. No Rails são representados em ficheiros separados. Primeiro são criados a partir de migrações (generate migration) e depois existe uma operação - rake db:migrate - para gerar as tabelas, sem perder ou afetar as existentes.
- As vistas (Views) definem a forma como os dados são mostrados ao utilizador e como é que este interage com a aplicação. No Rails, são representadas por páginas html.erb (Embedded Ruby) que têm código HTML e Ruby.
- Os controladores (Controllers) recebem pedidos a partir das Views e depois comunicam com os Modelos. No Rails, o utilizador deve definir ações para um determinado modelo. Seguem o princípio do CRUD em que se pode criar (Create), ler (Read), atualizar (Update) e destruir (Destroy).

Em relação aos componentes, o Ruby on Rails já integra alguns componentes como por exemplo o sistema de migração de base de dados, para o utilizador fazer as operações de alteração à BD de uma forma mais fácil e automática. Suporta a instalação de *gems*, que são componentes adicionais que podem ser integrados a um projeto, como por exemplo o Devise (para autenticação), Bootstrap (para a interface) e outros mais que podem ser instalados a partir do RubyGems. A documentação é boa e fácil de seguir. O Ruby on Rails é bastante suportado pela maioria dos serviços cloud, como o Heroku, Bitnami, Engine Yard, Cloogle Cloud Platform e outros.

2.3.3.2 Django

Django é uma framework web em Python (Django (2014)). Tem os seguintes princípios:

- O MTV (model-template-view)
- O DRY (tal como no Rails)
- A reutilização, rápida implementação e integração de componentes.

Como dito anteriormente, o Django segue uma arquitetura diferente do MVC (model-view-controller), porque a framework é que fica responsável pelos Controladores; o utilizador só se deve preocupar com os Modelos e as Views, e dá ao utilizador os Templates. Temos assim uma arquitetura MTV que será descrita a seguir:

- Os Modelos no Django, ao contrário do Rails são representados num único ficheiro models.py. Neste caso, a migração é feita com um python manage.py migrate para gerar as tabelas, sem perder ou afetar as existentes.
- Os Templates no Django representam a forma como queremos mostrar os dados. São como as Views no Ruby on Rails. No Django, são páginas com código HTML e Python.
- As Views no Django, representam os dados que queremos mostrar. São parecidos com os Controladores no Ruby on Rails.

Em relação aos componentes, o Django já integra alguns componentes, como por exemplo o sistema de migração, sistema de autenticação, ferramentas para o Google Sitemaps e ferramentas contra ataques web. É possível instalar mais componentes, como por exemplo o South para o sistema de migração (só para versões antigas do Django), o django-scaffold, alguns sistemas de base de dados (PostgreSQL, MySQL, SQLite, Oracle), que podem ser instalados a partir do Django Packages. A documentação parece ser bastante boa, tal como no Rails. Ao contrário do Ruby in Rails, o Django é suportado por poucos serviços cloud, como o Bitnami ou Google Cloud Platform. Existem serviços na internet que usam esta framework como o Pinterest, Instagram ou Disqus.

2.4 Imagens geográficas por satélite

As imagens obtidas por satélite são o input do sistema de processamento automático de deteção da linha de costa. Depois desse processamento, a linha da costa é importada para a base de dados OGC para ser apresentada ao utilizador num visualizador de mapas. Esta secção descreve o tipo de imagens a processar e a forma como esses dados são armazenados e apresentados ao utilizador.

2.4.1 Tipos de imagens a processar

As imagens geográficas são obtidas por Internet através de serviços de dados abertos (open data) que disponibilizam as imagens gratuitamente para o utilizador descarregar. De momento esses serviços disponibilizam imagens de dois satélites: o Landsat e o Sentinel. Seguidamente dá-se informação sobre os dois satélites mencionados, nomeadamente a resolução e outros detalhes da imagem disponibilizada e o tempo de revisita do satélite.

2.4.1.1 Landsat

O Landsat é um programa da NASA [1] que já conta com mais de 40 anos de existência e 7 satélites lançados. O objetivo do Landsat é obter imagens espaciais para estudo e obtenção de métricas (como estatística populacional, crescimento da urbanização global, zonas de

[1] National Aeronautics and Space Administration, agência norte-americana de Aeronáutica e Espaço

costa...). (NASA (2014a), NASA (2014b)). No serviço Coastline Watch, de acordo com a especificação do documento (Nuno Duro, Gil R. Gonçalves, Ricardo Martins, António A. Silva (2015)), vão ser processadas as imagens dos satélites Landsat 5, 7 e 8, cuja descrição será feita a seguir:

- O Landsat 5 foi um dos satélites que mais tempo esteve em atividade desde 1984 a 2012. Tinha um MultiSpectral Scanner (MSS) e um Thematic Mapper (TM) que dava imagens com melhor detalhe e detetava sete bandas (banda verde, banda azul, banda vermelha, banda near-infrared, duas bandas mid-infrared e uma banda térmica).

- O Landsat 7 foi lançado em 1999 com Enhanced Thematic Mapper Plus (ETMP), uma iteração do Enhanced Thematic Mapper (ETM) do Landsat 6 que era uma evolução do TM com uma oitava banda (pancromática) com resolução espacial [2] de 15 metros.

- Foi lançado em 2013 o Landsat 8 que possui novos instrumentos Operational Land Imager (OLI) e Thermal Infrared Sensor (TIRS). Este satélite tem um tempo de revisita de aproximadamente 16 dias, 11 bandas espectrais e resolução espacial de 15 metros (Nuno Duro, Gil R. Gonçalves, Ricardo Martins, António A. Silva (2015)). As imagens obtidas por esse satélite têm as seguintes características: formato GeoTIFF, utilizam o Universal Transverse Mercator (UTM) como projeção do mapa e o WGS-84 como modelo matemático de representação da superfície da terra (datum).(USGS (2014))

2.4.1.2 Sentinel

Os Sentinel são programas da agência Espacial Europeia (ESA - European Space Agency) que operam sob o Programa Copernicus [3]. A ESA pretende realizar 7 missões e em cada missão vai haver dois satélites Sentinel (A e B) por forma a cumprir os requisitos de revisita e de cobertura do programa. Estes satélites têm radares e instrumentos para obtenção de imagem multi-espectral para monitorização da terra, oceano e atmosfera.(ESA (2014a))

O program Sentinel é muito recente, tendo sido colocado em órbita o Sentinel-1 em 2014 e o Sentinel-2A em Junho de 2015. O serviço Coastline Watch está preparado para receber imagens do Sentinel-2A (Nuno Duro, Gil R. Gonçalves, Ricardo Martins, António A. Silva (2015)), que possui as seguintes características: 5 dias de tempo de revisita, 13 bandas espectrais e resolução espacial de 10 metros. As imagens obtidas por esse satélite são JPEG2000 (ESA (2014b)), usam o Universal Transverse Mercator (UTM) como projeção do mapa e o WGS-84 como modelo matemático de representação da superfície da terra (datum). (ESA (2014c))

[2] Resolução Espacial é o tamanho em metros correspondente a cada pixel da imagem. Quanto menor for esse tamanho, mais nítida é a imagem.

[3] Copernicus ou GMES - Global Monitoring for Environment and Security

2.4.2 Armazenamento das linhas de costa

Para o armazenamento, pretende-se um serviço capaz de guardar resultados do processamento para uma base de dados Open Geospatial Consortium (OGC). Existem 3 abordagens diferentes para fazer isso:

- Usar uma plataforma SaaS
- Usar uma plataforma IaaS/PaaS e fazer instalação manual
- Criar uma base de dados PostGIS no serviço cloud

A primeira abordagem consiste na aquisição de um serviço SaaS geoespacial, como por exemplo, o AcuGIS (AcuGIS (2014)). O AcuGIS vende serviços SaaS cloud do tipo GeoServer, PostGIS, PostgreSQL, Neatline (serviço de mapas para estudos históricos), Drupal (Content Management System) e Cartaro (Content Management System Geoespacial), com tudo incluído, sem configurações ou instalações iniciais. Estes serviços trazem um painel de controlo para a gestão do serviço, e a possibilidade do utilizador puder instalar mais algum software se desejar. A questão é que este serviço é pago e o seu custo elevado.

A segunda abordagem consiste na utilização de um servidor cloud do tipo IaaS ou PaaS e instalação de raíz o serviço e as suas dependências software. O IaaS traz flexibilidade mas é mais difícil porque requer algum controlo manual (os backups são feitos manualmente pelo utilizador) e tem custos acrescidos. O PaaS é mais controlado, existe por exemplo o GeoPaas que fornece um serviço Paas geoespacial. O uso do serviço requer um registo no serviço cloud OpenShift, o que permite fazer uma cópia do repositório GeoPaas para instalar como um serviço cloud. A pouca atividade e reputação que o GeoPaas tem não permitem que seja utilizada neste projeto.

A terceira abordagem consiste em implementar uma base de dados PostGIS no serviço cloud. O PostGIS é uma extensão geográfica e espacial para o PostgreSQL. O Amazon AWS disponibiliza o serviço RDS (Relational Database Service) que está integrada no plano free-tier e que permite ao utilizador gerir, configurar e fazer o escalonamento de uma base de dados na cloud. O RDS suporta PostgreSQL, podendo assim integrar funcionalidades do PostGIS na base de dados do Coastline Watch a custo zero. Do lado do backend, será desenvolvido um script na instância EC2 que executa o comando psql para fazer a importação das linhas de costa através da funcionalidade *shp2pgsql*. No lado do frontend, é feita uma ligação remota à instância para executar o script para a importação.

2.4.3 Visualização das linhas de costa

O serviço Coastline Watch deve apresentar as linhas de costa (shorelines) num mapa e dar também métricas ao utilizador (como por exemplo a área) para comparar a evolução da linha da costa em relação aos outros anos. Para fazer isso, o sistema precisa de obter essas linhas da base de dados e fazer a sua conversão para informação do tipo GeoJSON [4] para

[4]GeoJSON: ficheiro JSON de dados geográficos

depois apresentá-los num serviço de visualização de mapas.
Na implementação do serviço, foram escolhidos dois serviços de visualização de mapas que suportam a visualização das linhas de costa: o Google Maps e o OpenLayers.

O Google Maps é um serviço de mapas mais conhecido. Lançado em 2005, este serviço é utilizado para localização e planeamento. Possui funcionalidades como o "Street View"(visualização de imagens a 360 graus) , imagens satélite e um "route planner"(para definir trajetos). O Google Maps também está presente no Android e no iOS, mas dispõe também de uma API em JavaScript para integração na página web. O suporte do GeoJSON existe através da operação *map.data.loadGeoJson*. A Google utiliza a projeção EPSG (European Petroleum Survey Group):3857 (ou EPSG:900913) que é a projeção de Mercator na forma esférica da Terra, usando coordenadas X e Y em metros.

O OpenLayers, por outro lado, não é um serviço de mapas. É uma API em JavaScript bastante poderosa permitindo ao utilizador inserir as camadas que pretende visualizar, sejam mapas ou vetores (até é possível inserir o Google Maps ou OpenStreetMaps como camadas). Também suporta GeoJSON, usando *newOpenLayers.Format.GeoJSON()* e depois ser integrado dentro do *OpenLayers.Layer.Vector()*. O Openlayers usa a projeção EPSG:4326 (ou WGS84, World Geodetic System) que é a projeção na forma elipsoidal da Terra usando coordenadas latitude e longitude em graus. No Openlayers, é possível usar outras projeções, sendo necessário fazer a sua respetiva tradução/conversão (no caso de se usar camadas Google Maps no OpenLayers, é necessário definir o EPSG:3857).

2.5 Processamento de imagens

O processamento das imagens é necessário para a deteção da linha da costa. Esse processamento requer primeiro que o utilizador faça o descarregamento das imagens geográficas a partir de um repositório de imagens geográficas, para depois iniciar a execução do script desenvolvido que executa 4 algoritmos:

- Pansharpening: Junção de várias imagens numa imagem melhorada.

- MNDWI + K-Means: A aplicação dos algoritmos *MNDWI* (Modified Normalised Difference Water Index) e *K-Means* permite detetar as zonas onde existe água (rios, mares, lagos...)

- Vetorização: A partir da imagem K-Means processada, obtém-se uma linha de costa que é um ficheiro vetorial do tipo .shp que depois pode ser importado para a base de dados OGC.

As imagens geográficas a processar são imagens TIFF de grande resolução e tamanho, pelo que se espera que este processamento seja muito demorado e exigente.

2.5.1 Bibliotecas de processamento

O script de processamento funciona com recurso às bibliotecas de processamento Orfeo Toolbox (OTB), Geospatial Data Abstraction Library (GDAL), OpenGIS Simple Features Reference Implementation (OGR) e System for Automated Geoscientific Analyses (SAGA) que disponibilizam algoritmos necessários para executar e que trazem suporte (ou bindings) para linguagens como Python, Java, C++ ou Matlab. A seguir são detalhadas essas bibliotecas de processamento.

2.5.1.1 Orfeo Toolbox (OTB)

O Orfeo Toolbox é uma biblioteca de processamento de imagens desenvolvida pelo Centre National d'Études Spatiales (CNES) e distribuída sob licença CEA CNRS INRIA Logiciel Libre (CeCILL). Esta biblioteca é muito completa e que dispõe, por exemplo, ferramentas/algoritmos para: (Orfeo Toolbox (2015))

- Extração: índices radiométricos, extração de linhas
- Manipulação: conversão de imagens, rasterização
- Aprendizagem: classificação K-Means, Markov, SVM
- Filtragem: smoothing
- Fusão: pansharpening, segmentação
- Análise: análise de imagens e vetores
- Outros: BandMath, Calibração, obter valor do pixel...

O OTB tem uma interface gráfica chamada Monteverdi GUI, e disponibiliza executáveis que podem ser evocados a partir da linha de comandos. Esta biblioteca disponibiliza bindings para Python.

2.5.1.2 Geospatial Data Abstraction Library (GDAL)/ OpenGIS Simple Features Reference Implementation(OGR)

O GDAL/OGR é uma biblioteca para tradução de formatos de dados geográficos e distribuída com a licença Free and open-source software (FOSS). A biblioteca GDAL é dedicada ao processamento de rasters (imagens), enquanto o OGR está dedicada ao processamento de vetores.(GDAL (2014a), GDAL (2014c)) A biblioteca GDAL contém ferramentas para:

- Manipulação: cópia de imagem (gdal_translate), editar informação (gdal_edit), comparação de imagens (gdalcompare), cálculo de imagem (gdal_calc)
- Conversão: conversão de coordenadas (gdalwarp), conversão de imagens (rgb2pct, pct2rgb)

- Criação: gerar TMS - Tile Map Service (gdal2tiles), criação de grelhas (gdal_grid), gerar VRT - Virtual Dataset (gdalbuildvrt), juntar imagens (gdal_merge)
- Extração: raster para vetor (gdal_polygonize), vetor para raster (gdal_rasterize), contornos (gdal_contour)
- Análise de imagem: obter informação (gdalinfo), visualizar DEM (gdaldem)

A biblioteca OGR tem ferramentas para:

- Conversão: converter o ficheiro vetorial para outros formatos (ogr2ogr)
- Criação: tile index (ogrtindex) e referências lineares (ogrlineref)
- Análise: listar informação de um determinado ficheiro vetorial (ogrinfo)

O GDAL/OGR não possui uma interface gráfica própria, mas é possível chamar as suas funções a partir da interface do QGIS. Tem executáveis que podem ser evocados a partir da linha de comandos e bindings para o Python.

2.5.1.3 System for Automated Geoscientific Analyses (SAGA)

O SAGA é um software GIS (SAGA (2014)) que é distribuído sob licença FOSS. No caso da API a licença é GPL (General Public License). Esta biblioteca tem ferramentas/algoritmos para:

- Grid (análise, cálculo, filtragem, interpolação, ferramentas de manipulação)
- Imagens/Rasters (classificação, SVM, fotogrametria, segmentação, ferramentas de processamento)
- Importação e exportação (grids, imagens, vetores...)
- Projeções (georeferenciação)
- Vetores/Shapes (extrair linhas, pontos, polígonos, ferramentas de manipulação)
- Simulações (erosões, incêndios, impactos humanos)

O SAGA tem uma interface gráfica chamada SAGA-GUI, possui executáveis que podem ser evocados a partir da linha de comandos e tem bindings para Python.

2.5.1.4 Outras bibliotecas

Outras bibliotecas de processamento foram tidas em conta, como o GRASS (Geographic Resources Analysis Support System) e o Processing/Sextante.

- O GRASS é um software GIS, disponibiliza uma biblioteca de processamento e é distribuído sob licença GPL. Esta biblioteca (GRASS (2014)) tem ferramentas para Rasters, Vetores, Base de dados, desenvolvidas na linguagem R. O GRASS tem uma interface gráfica chamada GRASS-GUI, tem executáveis que podem ser chamados a partir da linha de comandos e tem suporte para o Python. (GRASS Wiki (2014))
- O Processing (anteriormente conhecido como Sextante) é uma biblioteca de processamento que existe para o QGIS e que utiliza todos os algoritmos das bibliotecas GDAL/OGR, OTB, GRASS e SAGA (desde que estes estejam também integrados no QGIS). Esta biblioteca é interessante para o projeto porque através de um "import Processing"no Python temos acesso a todos os algoritmos, sendo mais fácil para o desenvolvimento dos scripts. As experiências feitas mostraram que há dificuldades na invocação das funcionalidade do Processing por um programa Python isolado.

2.5.2 Pansharpening

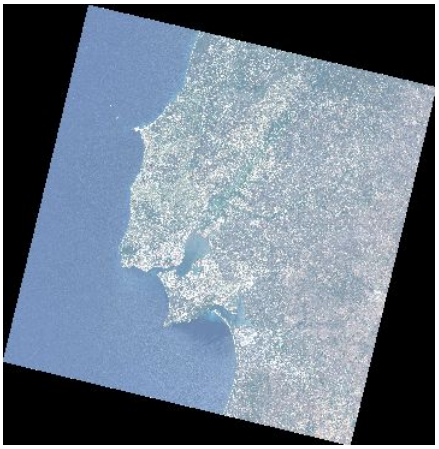

Figura 2.3: Pansharpening

Um conjunto de imagens Landsat de uma determinada zona tem 11 imagens que representam as 11 bandas. Cada banda deteta uma cor ou componente através de um conjunto de frequências do espectro eletromagnético e as suas imagens são representadas na escala de cinzas. Para este caso, vão ser usadas as primeiras 8 bandas. No Landsat 8, a banda 1 deteta azuis escuros e violetas, banda 2 deteta azuis, banda 3 deteta verdes, banda 4 deteta vermelhos, banda 5 é o Near Infrared (NIR) para detetar biomassa e linhas da costa, bandas 6 e 7 são o Short-wave Infrared (SWIR) para detetar humidade no solo e na vegetação, e banda 8 é a banda pancromática que junta todas as bandas numa só, sendo capaz assim de detetar mais luz e formar uma imagem mais nítida. (Landsat (2013)).
O pansharpening permite juntar uma imagem de banda pancromática (banda 8) de alta resolução de 15 metros com a imagem multi-espectral de baixa resolução (conjunto de imagens com bandas 1 a 7) de 30 metros para obter uma imagem mais nítida de alta resolução de 15 metros.

2.5.3 Modified Normalised Difference Water Index (MNDWI) + K-Means

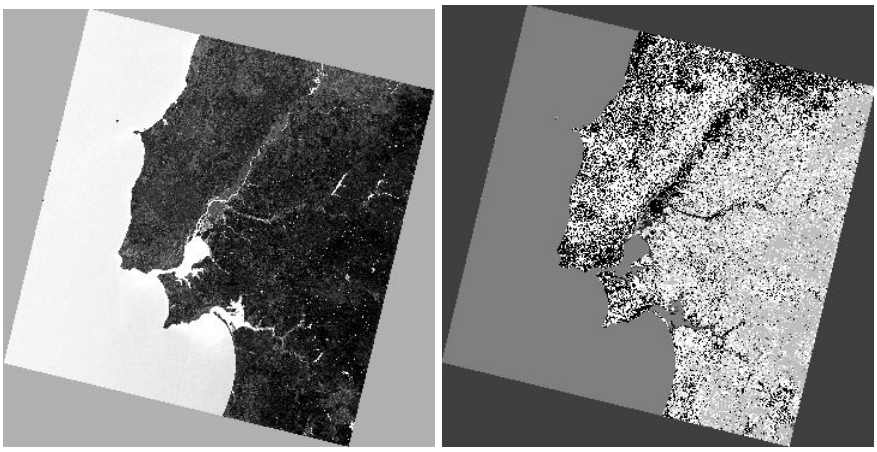

Figura 2.4: MNDWI (à esquerda) e K-Means (à direita)

Na segunda fase do processamento, para diferenciar as zonas de água de outras zonas para a deteção da linha de costa, é utilizado o MNDWI e K-Means. Primeiro o MNDWI é um índice de água que permite diferenciar numa imagem as zonas que têm água das zonas que não têm água. O MNDWI é baseado no NDWI (Normalised Difference Water Index) desenvolvido pelo McFeeters (S. K. McFeeters (1996)) que utilizava as bandas GREEN (banda 3) e NIR (banda 5) da imagem e calculava, para cada pixel, a divisão da diferença das bandas desse pixel com a soma delas desse pixel, ou seja:

$$NDWI = \frac{GREEN - NIR}{GREEN + NIR} \tag{2.1}$$

Contudo, Hanqiu Xu provou que o NDWI não era suficientemente bom(Hanqiu Xu (2006)). Num dos testes que realizou, as zonas de deserto e cascalho eram assumidas pelo NDWI como água, e o problema era que só considerava o fator da vegetação e que descartava outros fatores considerados importantes. (Graciela Schneier-Madanes, Marie-Françoise Courel (2010)) Por essa razão é que Xu desenvolveu o MNDWI, uma versão modificada do NDWI, que usa as bandas GREEN (banda 3) e SWIR1/MIR (banda 6) da imagem e calcula da mesma forma que o NDWI, ou seja:

$$MNDWI = \frac{GREEN - MIR}{GREEN + MIR} \tag{2.2}$$

Os resultados obtidos permitiram concluir que o MNDWI tinha uma precisão na extração perto dos 96,5%, sendo este melhor algoritmo que o NDWI e capaz de diferenciar melhor as zonas de água de outras zonas. (Hanqiu Xu (2006))

Na fase a seguir, a imagem MNDWI é processada com K-Means. O K-Means é um algoritmo de aprendizagem não supervisionada que tem um conjunto de n observações que são depois particionadas em k *clusters*. O algoritmo funciona da seguinte forma:

- 1) Com base numa pequena amostra (sample), definir aleatoriamente valores para k classes.

- 2) Para cada observação (pixel da imagem) dessa amostra, associá-la a uma classe que tenha um valor mais próximo da observação.

- 3) Para cada classe, ajustar o seu valor (centróide) com base na média de valores do seu conjunto de observações associado.

- 4) Repetir os passos 2 e 3 até não ser mais possível ajustar.

Depois disso, sabendo a lista de classes e os respetivos valores, começa-se por percorrer todos os pixéis da imagem e associar cada pixel a um número de classe (prediction).

A ideia do KMeans neste processamento é de reduzir o número de cores na imagem (quantização da cor), removendo assim algum ruído existente na imagem e permitir que depois a vetorização consiga detetar/desenhar facilmente a linha da costa.

2.5.4 Vetorização

Figura 2.5: Vetorização

O último passo consiste em gerar uma linha de costa com base na imagem obtida anteriormente. A linha de costa é representada por um vetor com pontos definidos por latitude e longitude que depois no sistema é armazenada para a base de dados OGC (Open Geospatial Consortium).

Para a versão de processamento sequencial, não é necessário implementar de raiz esses algoritmos porque já existem bibliotecas opensource que já incluem esses algoritmos para o processamento, como o OTB, QGIS e SAGA e que trazem suporte para linguagens como IDL, Python, Java, C++ e Matlab. Para a diminuição dos tempos de execução usando GPUs, poder-se-á fazer modificações ao código fonte das bibliotecas existentes, ou utilizar bibliotecas com facilidades semelhantes e que utilizem processamento GPU.

2.6 Paralelização

Foi referido anteriormente que o processamento das imagens podia ser lento e pesado por causa do tamanho das imagens a processar e da complexidade dos algoritmos a executar e que a paralelização de algumas partes do processamento permitiria reduzir os tempos de execução.

Um dos aspetos importantes do processamento paralelo é saber qual é a sua performance em relação ao processamento sequencial. A chamada Lei de Amdahl (Amdahl (1967)) consegue prever qual é a aceleração (speedup) máximo de uma aplicação usando vários processadores. O Speedup neste caso permite representar o ganho obtido na conversão da aplicação para processamento paralelo, ou seja:

$$S = \frac{T_{seq}}{T_{par}} \tag{2.3}$$

Em que T_{seq} é o tempo da aplicação com processamento sequencial e T_{par} é o tempo da aplicação com processamento paralelo, e para haver ganhos, o valor de S tem que ser maior que 1. Contudo, a Lei de Amdahl diz que "a menos que o programa sequencial seja todo paralelizado, o seu speedup vai ser limitado, independentemente do número de CPUs utilizados ", ou seja, para os casos em que não é possível paralelizar totalmente o programa, o nosso T_{par} vai ser:

$$T_{par} = P * \frac{T_{seq}}{n} + (1 - P) * T_{seq} \tag{2.4}$$

Em que o tempo vai ser a soma do tempo que demora a processar uma percentagem P de código paralelo (usando n núcleos) com o tempo que demora a processar uma percentagem $(1 - P)$ de código sequencial. Logo, o nosso speedup vai ser:

$$S = \frac{T_{seq}}{P * \frac{T_{seq}}{n} + (1 - P) * T_{seq}} = \frac{1}{\frac{P}{n} + (1 - P)} \tag{2.5}$$

Quando n tende para infinito, o seu speedup é $\frac{1}{1-P}$.

Para se fazer a paralelização dos algoritmos usados na deteção da linha de costa, foi definida a seguinte estratégia:

- Fazer o *profiling* das operações de processamento para determinar as fases mais demoradas.
- Uma vez identificadas as operações mais demoradas, analisar a documentação e código fonte com o objetivo de identificar as características do código crítico de forma a decidir a melhor abordagem de paralelização.

2.6.1 Formas de Paralelização

Existe um grande interesse da comunidade científica pela paralelização na área da Observação da Terra (OT). Uma aplicação de processamento de imagens pode ser paralelizada de várias formas:

- Usando computação distribuída/Grid: Várias máquinas eram ligadas entre si por rede. Uma aplicação como o MPI, Condor ou o Globus recebia uma tarefa a processar e distribuía pelas máquinas para cada uma processar a sua parte.
- Usando CPUs multi-core: Com os processadores multi-core, pode-se apostar na programação multi-processo com memória partilhada por forma a dar uso dos núcleos CPU para a execução paralela de dados, como o MPI ou OpenMP.
- Usando GPUs: Uma GPU é mais rápida a processar do que o CPU em problemas que envolvam grandes volumes de dados (imagens geográficas), por causa dos baixos tempos de acesso à memória e do grande número de núcleos.

A paralelização com computação distribuída/Grid dava uso das máquinas ligadas entre si por rede para processarem as imagens em paralelo, mas um problema dessa abordagem está nos custos, nos tempos de acesso e latências (por causa da velocidade da rede e das máquinas) e nas falhas da rede.

A paralelização usando os CPUs e GPUs tem como vantagem a utilização dos seus núcleos para processar partes de uma imagem em paralelo. Nas computações efetuadas em processamento de imagem, é frequente a aplicação da mesma operação a todos os pixeis de uma imagem. Neste contexto, parece ser pouco promissor fazer o processamento paralelo de imagens em CPU, uma vez que este está limitado pelo pequeno número de núcleos no CPU e pelo engarrafamento (bottleneck) existente no acesso à RAM. Pelo contrário, o processamento paralelo das imagens com recurso a GPUs aproveita as potencialidades dos milhares de núcleos e, supondo acessos à memória regulares, da elevada largura de banda de acesso à memória (Barry Wilkinson, C. Michael Allen (2005)). Assim, grande parte das operações de processamento de imagem podem ser feitas desta forma, pondo cada um dos processadores dos GPUs a processar um dos pixeis da imagem.

2.6.2 Processamento paralelo usando Python

Na paralelização desta aplicação, vai ser usada a linguagem Python para a sua implementação. A sua escolha em relação ao C recaiu na facilidade de programar. Enquanto a linguagem C não tem gestão automática de memória, requer compilação prévia e não integra funcionalidades de paralelização CPU e GPU (sendo necessário instalá-los à parte), o Python por outro lado torna isso mais fácil, não sendo necessário compilar, faz gestão automática da memória e integra funcionalidades de paralelização CPU integradas (threading e multiprocessing) mas não para GPU, sendo necessário instalá-lo.
A diferença entre threading e multiprocessing no Python é que enquanto o threading cria threads dentro do processo e a memória do programa é partilhada por todas elas, o multiprocessing cria processos mas sem qualquer partilha de memória, em que cada uma usa a sua memória. O grande problema do Python é o Global Interpreter Lock (GIL) (David Beazley (2010)) que é um lock que limita o acesso a uma rotina do código a uma só thread ou processo, enquanto as outras threads ou processos esperam que ela acabe, o que

torna a sua execução mais demorada. Usando um sistema de comunicação de dados de computação paralela, como o Message Passing Interface (MPI) ou outro similar, permite resolver esta questão.

2.6.3 Processamento paralelo com GPUs

As GPUs em cada geração estão mais evoluídas em termos de hardware. Por exemplo, a nVidia possui GPUs constituídas por vários streaming multiprocessors e cada uma tem 8 núcleos shader processor (SP), memória local partilhada pelas SP, 16384 registos e uma ALU (Pavel Karas (2010)). As Fermi têm 1536 threads e 512 núcleos distribuídos por 16 SM, e cada SM possui 32 SP, 32768 registos, 16 unidades de precisão dupla, 4 unidades de função especial e 16 unidades Load/Store. As Kepler têm 2048 threads e 2880 núcleos distribuídos por 15 SMX e que cada SMX possui 192 SP, 65536 registos, 64 unidades de precisão dupla, 32 unidades de função especial e 32 unidades Load/Store.

Inicialmente, as GPU eram só dedicadas a processar gráficos, como os jogos ou o processamento de vídeo e imagem. Com a evolução das GPU que permitiu dar mais poder de computação para processar gráficos, houve também um grande interesse da comunidade científica em puder usá-las para processar grandes volumes de outros dados. Inicialmente, tinha-se que programar usando APIs gráficas, como DirectX e OpenGL, que eram complicadas e demoradas. Por essa razão, surgiram as APIs dedicadas a processamento GPU (GPGPU, ou General Purpose GPU) (Gabriel E. Martinez Arroyo (2011)), e ao qual nesta secção vão ser destacadas duas delas: o CUDA e o OpenCL.

2.6.3.1 CUDA

O Compute Unified Device Architecture (CUDA) é uma plataforma proprietária da nVidia de computação paralela que usa os GPUs para aumentar a velocidade do processamento e que é suportado exclusivamente pelas gráficas nVidia a partir da micro-arquitetura Tesla (CUDA (2015)). É uma linguagem muito fácil de usar e que usa extensões da linguagem C para aceder à memória física do GPU. Possui ferramentas como compilador, debugger, um profiler (para analisar a performance da aplicação) e algumas bibliotecas científicas como o CUFFT e CUBLAS (Pavel Karas (2010)). Tem suporte para linguagens como C/C++, Fortran, Haskell, Java, MATLAB, Perl, Python e Ruby. No Python, o CUDA pode ser usado instalando a biblioteca PyCUDA.

Para passar a cadeia de processamento para o GPU, é necessário fazer modificações, uma vez que no desenvolvimento em CUDA não se resume em copiar o código fonte original, compilar e correr o executável. Um programa CUDA é constituído por:

- Um "host"que é o CPU e a sua memória,
- Um "device"que é o GPU e a sua memória

- Um "kernel"que é um método com o código a executar pelo GPU, e com apontadores como argumentos de entrada. A keyword __global__ no início permite que esse método seja chamado pelo host para o device executá-lo.

Um programa CUDA começa por reservar memória no device usando cudaMalloc, depois copia o input do host para a memória do device usando o cudaMemcpy e chama o kernel com o $kernel <<< N, M >>>$ para o device iniciar o processamento usando N blocos e M threads. Quando o device terminar o processamento, é necessário copiar a memória do device para o host e no final libertar a memória do device com o cudaFree.

Tal como descrito na secção "Formas de paralelização", a abordagem mais frequente no processamento de imagens consiste na aplicação da mesma operação a todos os pixeis. Quando um kernel é lançado, são chamados M threads que vão executar esse método em paralelo e cada uma processa um conjunto N de blocos que representam uma parcela da imagem. Mas a questão é procurar a combinação certa de N blocos e M threads que permita obter o melhor tempo de processamento para o caso de imagens com resolução aprox. de 15000x15000.

2.6.3.2 OpenCL

O OpenCL (OpenCL (2015)) é uma plataforma aberta de computação paralela com o mesmo princípio do CUDA. Foi desenvolvida inicialmente pela Apple e agora mantida pela Khronos. O OpenCL é suportado pela maioria das GPU, abrindo assim a possibilidade de se puder executar o mesmo código em qualquer GPU que tenha esse suporte. Usa também extensões do C para a programação, não possui uma ferramenta de debugging e profiler, mas existem ferramentas como o gDEBugger que fazem isso. Para usar a API do OpenCL no Python, existe uma biblioteca chamada PyOpenCL.

2.6.3.3 CUDA versus OpenCL

No caso da performance, o artigo (Kamran Karimi, Neil G. Dickson, Firas Hamze (2009)) apresenta um teste de performance de ambas as linguagens executando um gerador de números aleatórios Mersenne-Twister numa Geforce GTX260. Esse teste permitiu concluir que o CUDA é melhor a transferir dados de e para o GPU e a executar o kernel em relação ao OpenCL, porque a nVidia tem melhor suporte CUDA do que OpenCL, e a sua integração com OpenCL ainda está em desenvolvimento. (Pavel Karas (2010)) Não foram encontradas comparações mais recentes pelo que se supõe que esta diferença ainda existe.

Outro facto interessante é ambas as linguagens possuírem uma API e kernels parecidos entre si, o que abre a possibilidade de se puder converter código de uma linguagem para a outra. Existe software que converte automaticamente código CUDA para OpenCL, como por exemplo o CUDAtoOpenCL (Deepthi Nandakumar (2011)), que está em desenvolvimento, ou o Swan (Multiscalelab (2009)).

2.6.4 Serviços Cloud com suporte GPU

Relativamente aos serviços cloud que suportam GPU, o AWS disponibiliza duas instâncias pagas à hora para processamento GPU: (DevBlogs - Nvidia (2015))

- g2.2xlarge que usa nVidia GRID K520 (com 1536 cores e 4GB de memória), com um custo de $0.65/hora se for instância Ondemand ou $0.10/hora se for instância Spot.
- g2.8xlarge que usa 4 nVidia GRID K520 (com 4x1536 cores e 16GB de memória), com um custo de $2.6/hora se for instância Ondemand ou $0.40/hora se for instância Spot.

Também a SoftLayer da IBM disponibiliza instâncias para processamento GPU (Softlayer (2015)) pagas ao mês. Dá a possibilidade de escolher a GPU de acordo com as necessidades do cliente:

- nVidia Tesla K80 (com 2x2496 cores e 24GB de memória), com um custo de $1359/mês a $1529/mês.
- nVidia GRID K2 (com 2x1536 cores e 8GB de memória), com um custo de $1054/mês a $1224/mês.
- nVidia Tesla K10 (com 2x1536 cores e 24GB de memória), com um custo de $784/mês a $964/mês.

Existem outros serviços cloud que não foram falados anteriormente, mas que fornecem instâncias GPU nVidia, como é o caso do RapidSwitch ou o Penguin Computing. (DevBlogs - Nvidia (2015))

2.6.5 Computação paralela no processamento de imagens de satélite

Como dito na secção 1.2 deste documento, não existem referências a processamento paralelo aplicada à área da deteção das linhas, mas existem referências que associam computação paralela ao processamento de imagens de satélite. Reconhecendo que o principal problema do processamento de imagens é de ser demorado por serem imagens de grande resolução e tamanho em bytes, existe então um interesse em aplicar a paralelização para este caso para reduzir os tempos.

Uma das referências datada de 1996 apontava no uso da computação em grid para o processamento de imagens de satélite para deteção de nuvens em tempo real (Craig A. Lee, Carl Kesselman, Stephen Schwab (1996)). Os recursos das máquinas na altura eram limitadas para o processamento de imagens e era necessário recorrer à computação em grid para gerir e integrar vários recursos computacionais ligados por rede num só, permitindo também que o processamento seja feito de forma rápida.

Com o fácil acesso a clusters de computadores multi-core, começou-se a apostar no MPI para paralelizar esse processamento de imagens. O artigo de Mamta Bhojne, Anshu Pallav,

Abhishek Chakravarti, Sivakumar V (2013) faz até uma descrição geral dos passos que são feitos nesse processamento, e no qual aponta dois problemas desta abordagem: acesso à memória e acesso ao disco. Primeiro, atribuir várias threads para processarem as várias partes da imagem em paralelo permite reduzir o seu tempo, mas tem um custo adicional no tempo por causa da concorrência das threads no acesso à memória; segundo, este tipo de processamento requer também constantes leituras e escritas no disco, e sabendo que o disco é mais lento (se não for um SSD), também traz custos adicionais no tempo, e para resolver isso optou-se por uma abordagem paralela I/O multi-thread.

O artigo de Yegor V. Pushkin, Rauf Kh. Sadykhov, Leonid P. Podenok, Andrey V. Dorogush, Valentin V. Ganchenko (2014) descreve um exemplo de processamento de imagens de satélite Landsat para detetar as zonas onde há desflorestação. Essa deteção é feita usando os algoritmos Fuzzy C-means (FCM) e redes neuronais artificais de base radial (Radial Basis Function) que trazem um grande nível de complexidade computacional, daí se apostar no uso do MPI numa rede com várias máquinas em Grid para acelerar esse processamento.

O artigo de Rodrigo Alonso, Sergio Nesmachnow (2012) descreve um exemplo de processamento de imagens de satélite para estimar a quantidade de energia solar que é produzida no Uruguai. Usam duas imagens da mesma zona e hora do dia, uma com céu limpo e outra com o céu pouco nublado para determinarem os coeficientes de brilho para puderem também estimar a quantidade de energia que foi produzida. O problema descrito nessa publicação é de que são mais de 90 mil imagens a processar e que demora 15 horas a ser feito sequencialmente, daí ser necessário recorrer ao MPI para acelerar esse processamento. Os tempos obtidos permitem também demonstrar que não é possível com muito mais threads reduzir mais o seu tempo de processamento, e isso pode ser justificado pelos overheads no acesso à memória e ao disco.

Existe também um interesse em paralelizar esse processamento usando GPUs. Por exemplo, o artigo de In-Kyu Jeong, Min-Gee Hong, Kwang-Soo Hahn, Joonsoo Choi, Choen Kim (2012) descreve o processamento de imagens Landsat usando CUDA, aplicando fórmulas que permitem converter valores de pixel da imagem em valores de radiação espectral. O problema principal encontrado nessa abordagem é de saber como processá-las na GPU, sabendo que são imagens de grande tamanho e que ela não tem memória suficiente para a armazenar, e a solução encontrada passa por processar cada parte da imagem na GPU em vez da imagem toda. Os resultados obtidos neste documento provam que só é vantajoso usar o GPU em relação ao CPU para o caso de processamento de imagens de grande resolução.

É importante referir que algumas das publicações mencionadas chamam a atenção aos problemas e limitações encontrados na paralelização e que foram tidos em conta para esta implementação. Nos capítulos a seguir, serão descritos os esforços feitos para contornar estes problemas.

2.7 Súmula

Neste capítulo verifica-se que é possível tecnicamente, e com vantagens económicas, oferecer soluções de visualização da linha da costa baseadas na cloud. Os processamentos necessários requerem computações pesadas que podem ser paralelizados para puder tirar partido dos múltiplos processadores do CPU e GPU, e muitas delas permitem uma paralelização em que cada processador se dedica a processar uma parte das imagens, sendo neste caso necessário adaptar os GPUs a este tipo de abordagem para oferecer um compromisso preço/desempenho muito favorável.

CAPÍTULO 3

ORGANIZAÇÃO DA SOLUÇÃO

No capítulo anterior, foi feita uma abordagem à estrutura da aplicação Coastline Watch e as soluções existentes que podem ser usadas na aplicação e na paralelização. A primeira secção deste capítulo fala da disponibilização do Coastline Watch e as opções tomadas na implementação. A segunda secção fala da paralelização, onde é feita uma descrição detalhada dos algoritmos e as oportunidades de paralelização.

3.1 Disponibilização do Coastline Watch na cloud

3.1.1 Funcionamento

A disponibilização e implementação do serviço Coastline Watch está resumida na figura 3.1. O seu funcionamento é explicado através da descrição de uma utilização típica:

1. O utilizador regista-se e cria um projeto a partir da frontend.

2. a) Depois de criar, o utilizador começa por criar, a partir da frontend, uma referência de imagem.

   b) Dentro dela, faz o upload de uma imagem que é alojada no serviço de armazenamento.

   c) A referência é criada com uma "image path"associada que indica a localização da imagem na cloud. Para as próximas imagens, o utilizador repete o passo 2b.

3. O utilizador faz um pedido (request) a partir da frontend para processar as imagens com um determinado script.

4. a) Serviço de processamento obtém acesso à lista de pedidos pendentes na frontend.

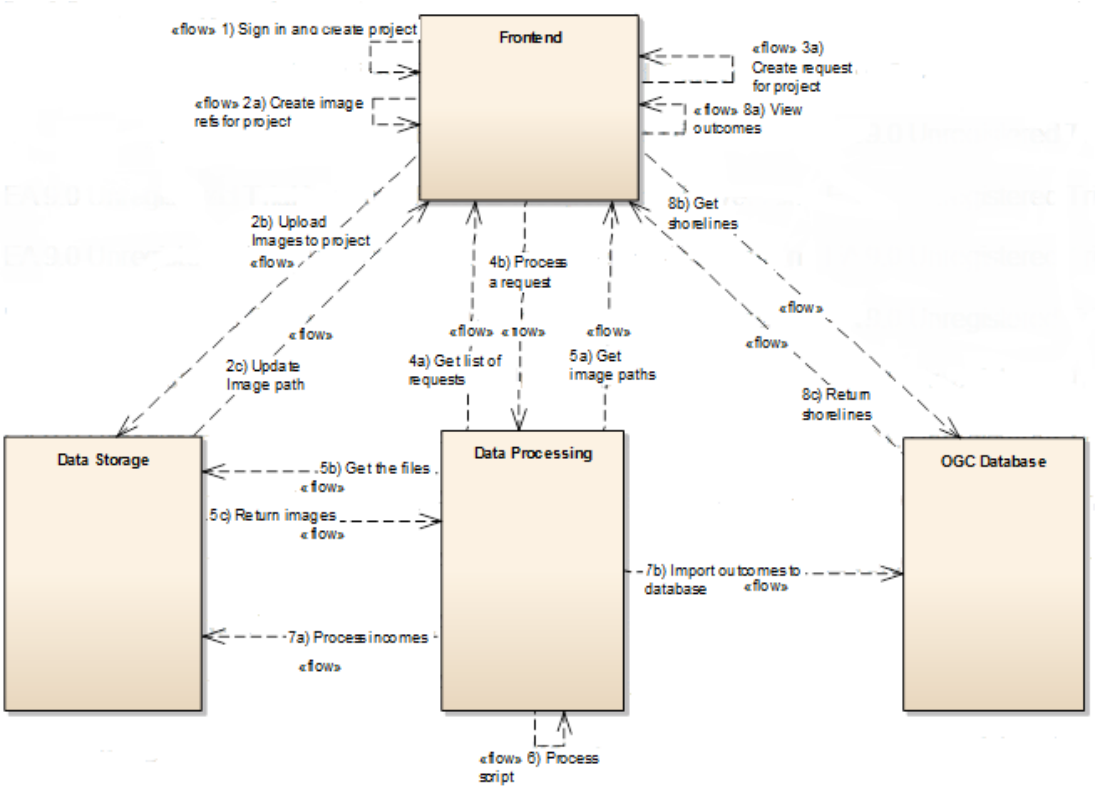

Figura 3.1: Data flow do sistema

b) Serviço de processamento processa o primeiro pedido na lista. O pedido do utilizador fica pendente até o serviço estar disponível.

5. a) Serviço de processamento vai à frontend buscar a localização das imagens do projeto.

   b) Serviço de processamento entra em contacto com o serviço de armazenamento que depois dá acesso dos ficheiros ao serviço de processamento.

6. Serviço de processamento inicia a execução do script de processamento automático. São geradas imagens intermédias durante o processamento. Essas imagens são criadas pelos algoritmos que depois são descartadas quando não são utilizadas pelos algoritmos seguintes para não ocupar espaço.

7. a) Depois do processamento, os resultados são alojados no serviço de armazenamento.

   b) O sistema executa o script de importação automática dos resultados para a base de dados.

8. a) Utilizador visualiza as linhas de costa na frontend.

   b) Frontend contacta a base de dados OGC para obter as geometrias dos produtos importados.

c) A base de dados OGC retorna essa informação, o frontend processa-a e depois carrega para o mapa.

3.1.2 Backend

No capítulo anterior, para o Backend do Coastline Watch foram considerados 3 serviços IaaS: Google Cloud Platform, IBM SmartCloud Enterprise, e Amazon Web Services (AWS).

O Google Cloud Platform não parece ter um serviço ao nível do Amazon Web Services. Não é um serviço atrativo porque não suporta ainda computação paralela GPU e não oferece um serviço gratuito "free-tier"tal como no AWS, havendo a possibilidade do utilizador ou uma empresa startup aderir ao Starter Credit.

O SmartCloud Enterprise não se revelou ser uma aposta interessante. Os problemas deste serviço resumem-se à falta de informação detalhada sobre o tipo de armazenamento e de processamento que utiliza, à utilização de uma API do tipo REST que não parece ser uma boa solução e à existência de instâncias de computação paralela GPU que são caras e pagas ao mês.

Sem contar com esses dois serviços, o AWS é o único serviço a apostar. Tem o free tier que dá a possibilidade ao utilizador puder experimentar por um ano a custo zero alguns dos serviços AWS, tem instâncias de computação paralela GPU pagas à hora e é um serviço que permite integrar o S3 com qualquer outro serviço cloud, como a SmartCloud (Amazon (2014b)) ou Heroku (Center (2015)).

3.1.3 Frontend

A frontend necessita de uma framework de desenvolvimento web e de um serviço cloud PaaS de alojamento. No capítulo anterior, foram tidos em conta 5 serviços cloud PaaS e 2 frameworks de desenvolvimento web.

3.1.3.1 Serviço Cloud

Relativamente aos serviços cloud PaaS, foram obtidas as seguintes conclusões:

- O Heroku possui um serviço gratuito, bom suporte, e uma documentação bastante compreensível. O único problema é o tamanho das dynos na versão gratuita que pareceu demasiado baixos para as exigências das aplicações.

- O Bitnami possui um serviço gratuito, bom suporte. Mas não foi considerada uma boa escolha por causa dos preços que pratica e a ideia de instalação de aplicações por stacks que por um lado tira algum trabalho ao utilizador, mas tira a possibilidade do utilizador fazer isso manualmente.

- O Google App Engine não serve porque como é um serviço proprietário da Google, está associado ao Google Cloud Platform.

- O Engine Yard tem uma versão de demonstração, mas não foi tida em conta por causa do mau suporte. Este serviço tem uma política de preços que obriga o utilizador pagar pelo plano e pela instância AWS.
- O IBM Bluemix tem uma versão de demonstração, não possui uma documentação acessível e tem preços muito elevados.

A razão para a escolha do Heroku deve-se à experiência do autor e o bom suporte do serviço. Alguns serviços têm versões de demonstração mas com condições que limitam a sua utilização, como na Engine Yard (Engine Yard (2014b)) ou no IBM Bluemix (Bluemix (2014)), por isso não foram tidas em conta.

3.1.3.2 Framework de desenvolvimento Web

Relativamente às frameworks de desenvolvimento web, o Ruby on Rails e o Django possuem algumas semelhanças, como o DRY (Don't Repeat Yourself), suportam adição de componentes e possuem boa documentação.

Existem diferentes opiniões acerca do Django e do Rails (Bernardo Pires (2014), Bradford (2014)).

- O Django utiliza a arquitetura diferente MTV (model/template/view), o que traz alguma confusão para quem está habituado à arquitetura MVC. Um problema desta framework é ter poucos componentes no repositório, o que permite concluir que o Django não tem uma comunidade muito grande, nem tem um grande suporte na cloud.
- O Ruby on Rails é uma linguagem com uma boa expressividade permitindo um desenvolvimento de aplicações mais fácil. Tem um grande número de componentes existentes no repositório e uma grande comunidade e suporte.

A escolha do Ruby on Rails acabou por ser ditado pela familiaridade e experiência do autor com esta plataforma. Mudar para o Django podia acabar por trazer consequências e dissabores no desenvolvimento do projeto, por causa da arquitetura usada e pelo fraco suporte que tem.

3.2 Informação geográfica na cloud

3.2.1 Tipo de imagens a processar

Relativamente ao tipo de imagens a processar para este projeto, numa primeira fase experimental vão ser processadas imagens obtidas pelos satélites Landsat 5, Landsat 7 e Landsat 8, fazendo o download das mesmas a partir do repositório de imagens da US Geological Survey (USGS). No futuro, será aberta a possibilidade de se processar também imagens obtidas pelo satélite Sentinel-2. De acordo com o documento (Nuno Duro, Gil

R. Gonçalves, Ricardo Martins, António A. Silva (2015)), foram definidos os seguintes critérios na escolha das imagens a processar:

- Imagens tiradas em duas alturas do ano (verão e inverno) ou obter imagens tiradas na mesma altura em anos diferentes
- Imagens com pouca nebulosidade.

3.2.2 Armazenamento e visualização dos dados

Em relação à forma como vão ser armazenados os dados (GeoServer), foi optado ir pela terceira abordagem descrita na secção "Armazenamento das linhas de costa"do capítulo 2, utilizando o AWS RDS para criar uma base de dados geográfica com PostGIS. Esta abordagem tem como vantagens a melhor integração com os outros serviços backend e frontend, e o RDS possuir um sistema automático de backups à base de dados em caso de perda.

Em relação à forma de apresentar ao utilizador os resultados no mapa, numa primeira fase foi utilizado o Google Maps. Contudo, o Google Maps era limitado e decidiu-se mudar para o OpenLayers que era mais flexível e que permitia converter as projeções das linhas de costa usando *transform("EPSG:4326", "EPSG:3857")*.

3.3 Processamento de imagens

No serviço Coastline Watch, esse processamento é realizado de forma automática pela componente de processamento do backend da Cloud. O script tem que ser carregado primeiro pelo utilizador para o serviço de armazenamento na cloud (AWS S3), de seguida o utilizador faz o pedido a partir da frontend do sistema para executar esse script e processar assim o conjunto de imagens carregadas também pelo utilizador para o serviço de armazenamento na cloud. O script de processamento executa 4 algoritmos: Pansharpening, MNDWI, K-Means e Vectorização. O seu processamento é feito utilizando as bibliotecas de processamento Orfeo Toolbox (OTB), GDAL e SAGA.

3.3.1 Pansharpening

Segundo o Cookbook do Orfeo Toolbox (Orfeo Toolbox (2015)), na teoria o algoritmo de Pansharpening consiste na junção de 3 ou mais bandas da imagem Multi-espectral (XS) com a imagem Pancromática (Pan). Na prática, essa junção consiste numa primeira fase aplicar a transformada de Fourier como um filtro ”low-pass” sobre a Pancromática (PanSmooth), depois dividir a Multi-espectral (XS) pela PanSmooth, e multiplicar no final pela Pancromática. O algoritmo Pansharpening é representando matematicamente por:

$$Pansharpening = \frac{XS}{PanSmooth} * Pan \tag{3.1}$$

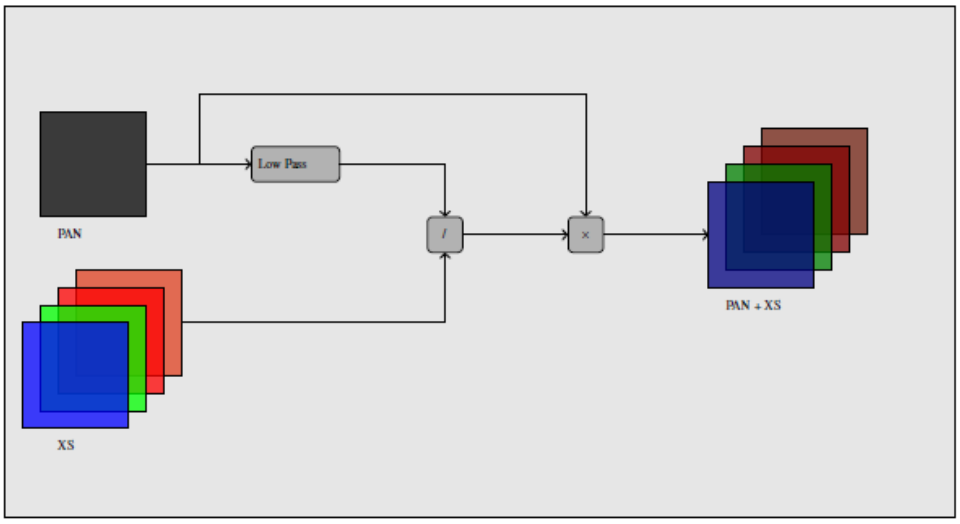

Figura 3.2: Diagrama do algoritmo Pansharpening (retirado do Orfeo Toolbox Cookbook)

A transformada de Fourier consiste em decompor os sinais de uma imagem para depois serem representadas como frequências. Existe a transformada inversa de Fourier que faz a inversa para decompor as frequências em sinais da imagem (R. Fisher, S. Perkins, A. Walker e E. Wolfart. (2003)). A fórmula normal de Fourier é representada por:

$$F(k,l) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} f(i,j)e^{-i2\pi(\frac{ki}{N}+\frac{lj}{N})} \tag{3.2}$$

E a sua transformada inversa é representada por:

$$f(i,j) = \frac{1}{N^2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} F(k,l)e^{i2\pi(\frac{ki}{N}+\frac{lj}{N})} \tag{3.3}$$

Em que F(k,l) é a imagem Fourier e f(i,j) é a imagem original no domínio espacial.
Inicialmente surgiu a DFT (Discrete Fourier Transform, ou Transformada Discreta de Fourier) que em vez de usar a totalidade das frequências, só utiliza uma amostra delas, o suficiente para puder representar a informação da imagem original. O problema da DFT é de ter uma complexidade $O(N^2)$ [1], e para contornar esse problema, surgiu o FFT (Fast Fourier Transform, ou Transformada Rápida de Fourier), que faz a decomposição da matriz DFT para reduzir a complexidade para $O(N * log(N))$. No FFT, existem vários algoritmos, como o Cooley-Tukey, Bluestein, Rader ou fatorização por números primos.

Para correr o pansharpening, é necessário gerar um ficheiro VRT, um ficheiro virtual que armazena em XML as referências aos ficheiros de entrada .tif e as respetivas bandas, utilizando o executável do GDAL *gdalbuildvrt* (GDAL (2014b)). Feito isso, pode-se então gerar uma imagem temporária multi-espectral de 7 bandas (bandas 1 a 7) com base no

[1] N representa o tamanho dos dados

ficheiro VRT, usando outro executável GDAL *gdal_translate*. Repete-se o mesmo procedimento também para a imagem temporária pancromática de 1 banda (banda 8) a partir do ficheiro VRT. No final procede-se à união das imagens multi-espectral e pancromática para se obter uma imagem pansharpened, recorrendo ao executável OTB *otbcli_pansharpening*. O diagrama a seguir explica como está a ser feita a execução do Pansharpening:

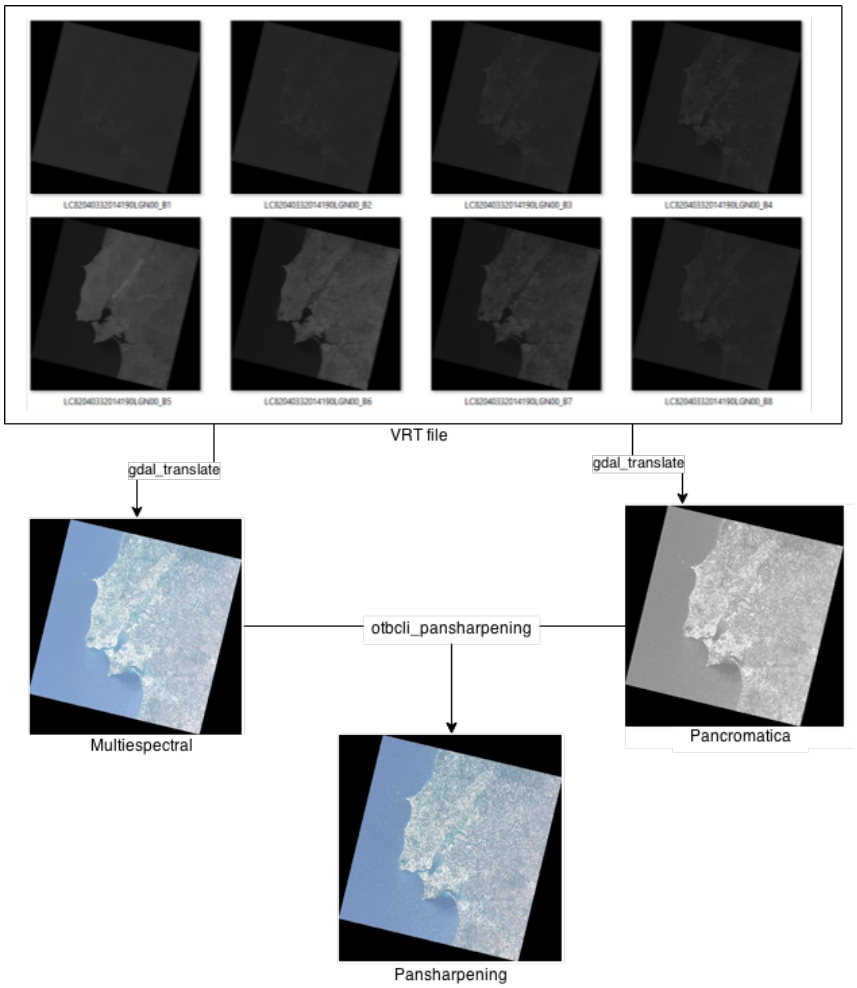

Figura 3.3: Diagrama de execução do pansharpening

3.3.2 Modified Normalised Difference Water Index (MNDWI)

Tal como abordado no capítulo anterior, o MNDWI é um versão melhorada do NDWI proposta por Xu que consiste na divisão simples da diferença da banda GREEN com o MIR, pela soma de ambas as bandas, permitindo assim diferenciar as zonas de água com outras zonas:

$$MNDWI = \frac{GREEN - MIR}{GREEN + MIR} \tag{3.4}$$

Isso é feito usando o executável do OTB, o *otbcli_RadiometricIndices*, passando neste caso como argumentos o nome do ficheiro input, o número da banda GREEN e banda MIR,

o nome do ficheiro output e o tipo de índice radiométrico (Water:MNDWI). O resultado obtido é uma imagem na escala de cinzas que representa o mar, os lagos e rios com pixeis brancos, e as zonas de terra, floresta, prédios e outros com pixeis de cores escuras ou negras.

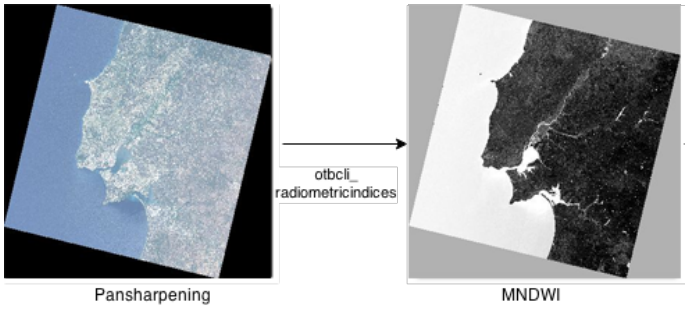

Figura 3.4: Diagrama de execução do MNDWI

3.3.3 K-Means

O K-Means é usado para diferenciar melhor as zonas de água de outros tipos de zonas na imagem MNDWI. A razão da sua utilização é que a imagem MNDWI tem "ruído" (por exemplo há pixeis de cores diferentes em zonas onde há mais pixeis escuros ou claros) e se não for tratado, o algoritmo de vetorização enfrenta dificuldades na deteção da linha da costa. O K-Means será executado com 5 clusters para representar as 5 zonas diferentes da imagem.
A classificação K-Means é feita usando o executável OTB, o *otbcli_KMeansClassification* que com base na imagem de entrada MNDWI e um número k de centróides (definimos k=5 para representar as 5 classes), gera uma imagem na escala de cinzas com os pixeis associados ao respetivo número de classe a que foi atribuído. Essa imagem definiu os pixeis de classe 2 os que pertencem a zonas com água.

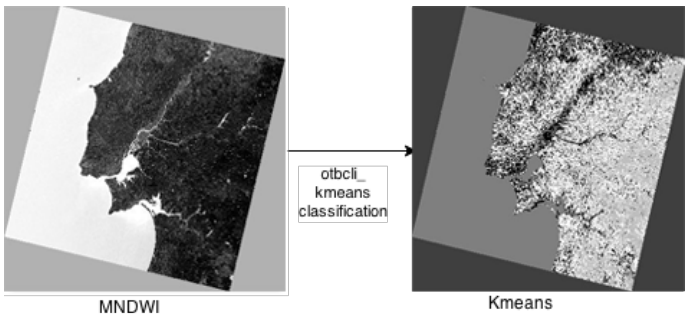

Figura 3.5: Diagrama de execução do K-Means

3.3.4 Vetorização

A vetorização é usada para determinar o polígono a partir de uma imagem de entrada. Este processo é feito usando dois executáveis SAGA: o *saga_cmd io_gdal* para gerar a grelha .sgrd com base na imagem do K-Means, e o *saga_cmd shapes_grid* que usa essa grelha e faz a vetorização das zonas com pixeis associados à classe que for definida (para detetar as zonas com água, é necessário definir nos parâmetros a classe 2). No final é gerado um ficheiro .shp que contém o polígono/vetor que depois pode ser visualizado a partir do QGIS.

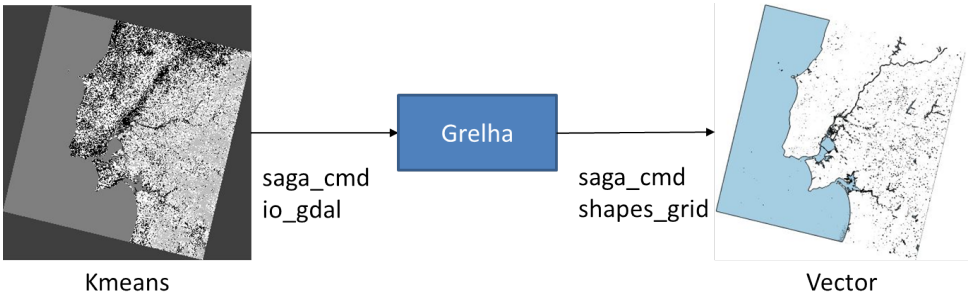

Figura 3.6: Diagrama de execução da Vetorização

3.4 Paralelização

Nesta secção, é feita uma descrição detalhada de cada um dos algoritmos originais de processamento, a sua respetiva oportunidade de paralelização e a opção tomada.

3.4.1 Pansharpening

O artigo (Weihua Sun, Bin Chen, David W. Messinger (2013)) mostra as vantagens da utilização do CUDA no pansharpening, contudo o seu método de processamento não é o mesmo que o usado no OTB. Enquanto que o pansharpening do OTB usa o low-pass filter (tal como descrito no cookbook), o artigo fala na utilização da difusão anisotrópica (anisotropic diffusion) e que cada pixel da imagem pansharpened é uma mistura linear de pesos dos seus pixeis vizinhos mais próximos. Há também outro artigo (PCI Geomatics (2009)) que mostra resultados de uma implementação paralela do algoritmo pansharpening desenvolvido por uma empresa e que recorre das potencialidades do CUDA e do multithreading para acelerar o processamento. Até ao momento, não foram encontrados artigos que descrevam uma implementação OpenCL do pansharpening. Estes documentos provam o interesse da comunidade científica em usar mais o CUDA do que o OpenCL para paralelizar o pansharpening.
Pode-se então concluir que o algoritmo Pansharpening é possível de ser paralelizado. Para o filtro "low-pass", não é necessário implementar manualmente a transformada de Fourier, porque apesar de ser complicada de implementar, existe uma biblioteca CUDA que pode

ser chamada para fazer isso, o cuFFT. A divisão e a multiplicação no Pansharpening são facilmente implementáveis no CUDA. Como descrito no capítulo 2, será utilizado também o Message Passing Interface (MPI) para contornar os problemas de paralelização no Python.

Numa fase inicial, irá ser implementada a versão sequencial do algoritmo de Pansharpening, tal como foi descrito. Na fase seguinte, procede-se à implementação da versão paralela do mesmo, usando como base o que já foi implementado da versão sequencial. O capítulo a seguir descreve a implementação detalhada de ambas as versões do algoritmo, as escolhas feitas e os seus resultados.

3.4.2 Modified Normalised Difference Water Index (MNDWI)

Existe um artigo (Sivakumar V., Ankit Goyal (2014)) que descreve uma implementação CUDA do MNDWI e do qual se conseguiu obter um speedup na ordem das 13 vezes usando a aproximação "um pixel por thread" (pixel wise parallel approach). Até ao momento, ainda não há artigos a descrever uma implementação OpenCL deste algoritmo. A sua implementação é acessível. As operações de divisão, soma e subtração podem ser facilmente implementados no CUDA. Como descrito no capítulo 2, será utilizado também o MPI para contornar os problemas de paralelização no Python.

Da mesma forma que no pansharpening, numa fase inicial, irá ser implementada a versão sequencial do algoritmo de MNDWI, tal como foi descrito. Na fase seguinte, procede-se à implementação da versão paralela do mesmo, usando como base o que já foi implementado da versão sequencial. O capítulo a seguir descreve a implementação detalhada de ambas as versões do algoritmo, as escolhas feitas e os seus resultados.

3.4.3 K-Means

Como o algoritmo K-Means pode ser usado para várias finalidades, foram encontrados imensos documentos a descrever implementações K-Means para CUDA e em OpenCL. No CUDA, este documento (BAI Hong-tao, HE Li-li, OUYANG Dan-tong, LI Zhan-shan, LI He (2009)) prova que o CUDA no K-Means traz speedups de 70x com 1024 clusters, usando uma Geforce 8800GTX. Outro documento (Reza Farivar, Daniel Rebolledo, Ellick Chan, Roy Campbell (2008)) testou o CUDA no K-Means usando uma Geforce 8600GT e obteve speedups de 13x com 4000 clusters. No OpenCL, existe este documento (Thilina Gunarathne, Bimalee Salpitikorala, Arun Chauhan (2011)) que fez um teste com o OpenCL usando uma Tesla C1060 e obteve speedups razoáveis usando 300 clusters (o speedup aumentava à medida que se aumentava o tamanho da amostra).
Existem também algumas implementações do K-Means que podem ser usadas neste trabalho, tanto em CUDA (cukmeans (Ludwig Schmidt-Hackenberg (2014)), CUDA K-Means Clustering (Serban Giuroiu (2013)) como em OpenCL. Como descrito no capítulo 2, será utilizado também o MPI para contornar os problemas de paralelização no Python.

Numa fase inicial, irá ser implementada a versão sequencial do algoritmo de K-Means, tal como foi descrito. Na fase seguinte, procede-se à implementação da versão paralela do mesmo, usando como base o que já foi implementado da versão sequencial, e depois utilizar o cukmeans que é uma implementação CUDA do K-Means feita em Python. O capítulo a seguir descreve a implementação detalhada de ambas as versões do algoritmo, as escolhas feitas e os seus resultados.

3.4.4 Vetorização

De momento, não foram encontrados documentos a descrever a implementação de um algoritmo de vetorização em CUDA ou em OpenCL. Dada a complexidade e a falta de soluções CUDA para a vetorização, tem que se apostar em alternativas CPU, como por exemplo bibliotecas Python GDAL Polygonize e/ou Rasterio e Fiona que têm disponível online um exemplo funcional de vetorização que pode ser modificadas e usada para a versão paralela.

Da mesma forma que no pansharpening, numa fase inicial, irá ser implementada a versão sequencial do algoritmo de Vetorização que usa o GDAL Polygonize. Na fase seguinte, procede-se à implementação da versão paralela do mesmo, usando como base o que já foi implementado da versão sequencial, mas usando o Rasterio e Fiona. O capítulo a seguir descreve a implementação detalhada de ambas as versões do algoritmo, as escolhas feitas e os seus resultados.

3.5 Súmula

Na implementação do serviço Coastline Watch, optou-se por usar soluções cloud de boa reputação e gratuitas nesta fase. O problema das soluções AWS e Heroku é que as versões free são limitativas em termos de recursos (pouca memória, pouco disco, base de dados pequena) e no futuro, será necessário apostar em soluções cloud pagas com melhores recursos.
Na paralelização, apostou-se na utilização do CUDA para paralelizar porque é mais usada pela comunidade científica do que o OpenCL. A abordagem mencionada na súmula do capítulo anterior e algumas das oportunidades de paralelização mencionadas neste capítulo vão ser aplicadas a todos os algoritmos, por forma a conseguir também ganhos no tempo total de processamento.

CAPÍTULO 4

IMPLEMENTAÇÃO E AVALIAÇÃO DOS RESULTADOS

No capítulo anterior, foi discutido a organização da solução, em que se explicou a forma como será disponibilizado o serviço Coastline Watch, e como vai ser feita a paralelização dos algoritmos de processamento. Neste capítulo serão apresentados os tempos de processamento do script original, seguido de um estudo aprofundado dos algoritmos envolvidos. A implementação desses algoritmos de processamento é descrita no final, incluindo uma apresentação dos tempos obtidos.

4.1 Disponibilização do Coastline Watch na cloud

Tal como descrito nos capítulos anteriores, o serviço Coastline Watch é composta por backend (com armazenamento, processamento e GeoServer) e um frontend.

4.1.1 Backend

Figura 4.1: Esquema de desenvolvimento

Primeiro foi preparado o serviço de armazenamento S3 em que foi criada uma bucket para alojar as imagens intermédias e os ficheiros .shp (e outros associados) da linha de costa.

Depois, foi preparada a instância de processamento EC2 do tipo t2.micro e usando o sistema operativo Ubuntu 14.04 Trusty. Para aceder à instância remotamente, foi criada a ligação do computador local ao serviço por SSH usando o PuTTY. Para a instância ter acesso à bucket na S3, foi utilizada uma aplicação chamada S3FS que faz um "mount", ligando assim a bucket à instância EC2.

Por fim, foi preparada a base de dados RDS para armazenar as linhas de costa importadas após o processamento. Dentro da RDS, foi necessário usar *CREATE EXTENSION postgis;* para suportar funcionalidades PostGIS quer no armazenamento, quer nas queries. No EC2, foram criados pequenos executáveis bash que fazem a importação das linhas de costa para a base de dados.

4.1.2 Frontend

A estratégia de desenvolvimento para este projeto consiste na implementação e teste de cada funcionalidade num ambiente de desenvolvimento (máquina local), e submissão para o ambiente de produção (Heroku) após testes, para garantir a solidez da aplicação.

Numa primeira fase, foram implementados os básicos da aplicação, como os Utilizadores e o sistema de autenticação. De seguida, foram implementadas as funcionalidades "core"da aplicação, como os Projects (Projetos), Images (Imagens) e Outcomes (resultados do processamento). No final, foi desenvolvida a funcionalidade dos Requests (pedidos de processamento), Processors (processadores ou scripts de processamento) e dos Resources (recursos ou servidores de processamento).

No Coastline Watch, existem 3 tipos de utilizadores registados: Administrador, Expert e Standard. Os Administradores ficam responsáveis pela gestão do serviço e adição de processadores (scripts de processamento), os Experts podem fazer pedidos de processamento com base nos scripts existentes, enquanto que os Standards podem visualizar os resultados processados. Tanto os Expert como os Standard podem também criar projetos, imagens e convidar utilizadores.

Cada imagem ou resultado ou processador tem uma "Sharebox"associada. A "Sharebox"pretende ser um repositório onde os utilizadores podem visualizar, fazer download, adicionar e remover ficheiros. A "Sharebox"faz a ligação remota à bucket do AWS S3 e foi implementada no Ruby on Rails usando a *gem* "aws-sdk-v1"desenvolvida pela Amazon.

Cada pedido criado no serviço é processado por um e só um recurso. Neste momento, é feito de forma manual em que o utilizador cria os pedidos, mas tem que carregar no botão para executar o pedido, mas no futuro, isso pode vir ser feito de forma automática.

Nos Outcomes, são guardados os resultados (ficheiros da linha de costa) depois do processamento e pode-se fazer a importação dos mesmos para a base de dados para serem visualizados no mapa. De momento, está a ser feito de forma manual, ao qual o utilizador

faz upload dos ficheiros da linha de costa e faz a importação manual, no futuro será feito de forma automática. A importação é feita chamando um script criado no EC2 que acede à base de dados RDS e faz a importação usando o *shp2pgsql*.

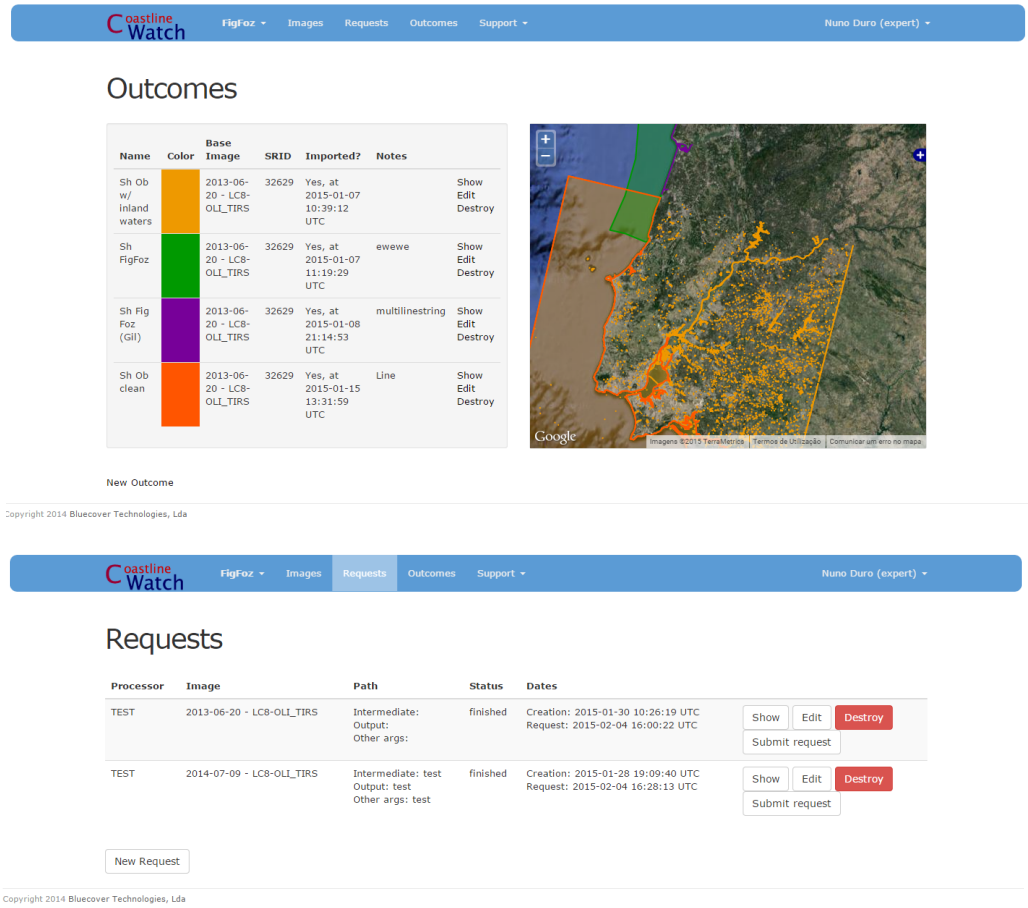

Figura 4.2: Imagens da aplicação Coastline Watch (Fevereiro 2015)

4.2 Resultados do script de processamento

Durante a preparação da dissertação, foi feito um primeiro teste ao script de processamento sequencial usando um conjunto de imagens da Lagoa de Óbidos obtidas no verão de 2014. Esse teste foi realizado em duas máquinas distintas:

- Portátil da empresa: Intel Core i7-4510U 2Ghz, 1TB disco, 8GB de memória e Windows 8.
- Cluster AWS EC2 Free tier: Intel Xeon 3.3Ghz, 8GB disco, 1GB de memória e Ubuntu 14.04.

Foram obtidos os seguintes tempos de processamento:

	Portátil Empresa	AWS EC2
Criar Raster Virtual	0:00:01	0:00:18
Criar imagem multi-espectral de 7 bandas	0:02:16	0:01:27
Criar imagem pancromática de 1 banda	0:00:17	0:00:24
Juntar a multi-espectral com a pancromática	0:12:52	0:04:04
Processar MNDWI	0:07:17	0:13:30
Processar K-Means com k=5 + Converter para Uint16	0:02:27	0:02:04
Criar grelha para a vetorização	0:00:21	-
Fazer vetorização	0:07:15	-
TOTAL	0:32:47	-

Tabela 4.1: Tempos de processamento do script original no portátil e AWS EC2

Concluiu-se que enquanto o processamento no portátil da empresa foi feito em aproximadamente 33 minutos, por outro lado o processamento na máquina EC2 não chegou a ser concluído porque o SAGA requer demasiados recursos de memória e a máquina não tinha capacidade suficiente. Em algumas situações, o EC2 foi mais lento que o portátil por duas razões:

- Tempo de acesso à bucket: Na criação do raster virtual, o EC2 teve que aceder remotamente às imagens alojadas na bucket no servidor S3, e isso traz um tempo adicional no tempo de execução.

- Muitos utilizadores: O tempo de processamento do algoritmo depende da quantidade de utilizadores a usar a EC2 naquele momento. Quando existem muitos utilizadores a usar a cloud, o processamento acaba por ser mais demorado. Noutras situações, é mais rápido.

Após a preparação desta dissertação, e dado que deixou de ser possível usar essas máquinas, repetiu-se o mesmo teste com o mesmo conjunto de imagens, usando outras duas máquinas:

- Desktop fixo do autor: AMD Phenom II x4 955BE 3.0Ghz, 4GB RAM, 500GB disco, Geforce GTX460 1GB, Ubuntu 14.04

- Máquina remota DI88: Intel Xeon E5506 2.13Ghz, 12GB RAM, 1.5TB disco, Geforce GTX680 2GB, Ubuntu Mate 14.04.

Foram obtidos os seguintes tempos de processamento:

Para este último caso, a diferença de tempos na vetorização em ambas as máquinas é grande. No Desktop, os 4GB de memória física não foram suficientes, o que levou a usar memória virtual do disco que é mais lenta e daí se ter obtido um tempo de processamento elevado. No DI88, os 12GB de memória física foram suficientes para a vetorização, daí ser mais rápido.

	Desktop	DI88
Criar Raster Virtual	0:00:02	0:00:02
Criar imagem multi-espectral de 7 bandas	0:01:23	0:01:40
Criar imagem pancromática de 1 banda	0:00:10	0:00:13
Juntar a multi-espectral com a pancromática	0:03:25	0:03:38
Processar MNDWI	0:04:19	0:02:20
Processar K-Means com k=5 + Converter para Uint16	0:01:24	0:00:39
Criar grelha para a vetorização	0:00:11	0:00:15
Fazer vetorização	0:08:12	0:01:19
TOTAL	0:19:06	0:10:06

Tabela 4.2: Tempos de processamento do script original no Desktop e máquina remota DI88

4.3 Estudo aprofundado dos algoritmos

Depois de analisar os resultados anteriores, foi feito um estudo para procurar os motivos dos algoritmos demorarem muito tempo e serem demasiado exigentes. A tabela abaixo representa o nível de exigência de cada passo do(s) algoritmo(s) em termos de recursos da máquina (CPU, Disco e Memória)

	CPU	Memória	Disco
Criar Raster Virtual	BAIXA	BAIXA	BAIXA
Criar imagem multi-espectral de 7 bandas	MEDIA-BAIXA	BAIXA	ALTA
Criar imagem pancromática de 1 banda	MEDIA-BAIXA	BAIXA	ALTA
Juntar a multi-espectral com a pancromática	MEDIA	BAIXA	ALTA
Processar Pansharpening com o algoritmo MNDWI	MEDIA-BAIXA	MEDIA-BAIXA	ALTA
Processar MNDWI com algoritmo K-Means com k=5	MEDIA	BAIXA	ALTA
Conversão MNDWI-KMEANS para uint16	MEDIA-BAIXA	BAIXA	ALTA
Criar grelha para a vetorização usando o MNDWI-KMeans	MEDIA-BAIXA	MEDIA-BAIXA	ALTA
Fazer vetorização	ALTA	ALTA	ALTA

Tabela 4.3: Tabela com o nível de exigência de recursos da máquina (CPU, Disco e Memória) para cada passo do algoritmo.

Estes resultados permitiram dar a entender que o grande problema dos algoritmos demorarem muito tempo a processar é de serem exigentes em termos de Disco e de CPU. No pansharpening, a geração de uma imagem única GeoTIFF de grande tamanho requer muitos acessos ao disco (leitura e escrita) para além do CPU. O MNDWI e K-Means requerem também muitos acessos ao disco. Na vetorização, o saga_cmd shapes_grid tem uma dependência muito grande dos recursos, deixando a máquina sem resposta por alguns minutos.
Com base na tabela acima e nos tempos obtidos no capítulo 3, foram escolhidos os seguintes passos:

- Juntar a multi-espectral com a pancromática (Pansharpening)
- Processar Pansharpening com o algoritmo MNDWI

- Processar MNDWI com algoritmo K-Means
- Fazer vetorização

4.4 Implementação

4.4.1 Abordagens possíveis

Feita a análise aos algoritmos originais, procedeu-se à escolha do tipo de abordagem a usar para implementar os algoritmos.

- Fazer download do código fonte das bibliotecas GDAL/OTB/SAGA e modificar para suportar paralelização e compilar. É possível fazer isso, mas como o código fonte delas está na linguagem C++, essa abordagem traz alguns problemas: primeiro, é necessário separar código CUDA do código C++, porque o nvcc só compila código do CUDA que está em C; e segundo, a sua implementação e estrutura não permite a fácil modificação dos mesmos para paralelização.
- Fazer uma implementação de raiz numa outra linguagem. Esta abordagem consiste na implementação dos algoritmos usando a linguagem Python.
  Optou-se pela segunda abordagem porque das bibliotecas estudadas anteriormente, o GDAL traz bindings para Python, o que significa que é possível recriar estes algoritmos de forma simples e rápida.

4.4.2 Bibliotecas Python a usar no processamento

4.4.2.1 CUDA

Sabendo que a abordagem escolhida obriga a que seja usado o Python para programar, foi necessário então procurar módulos/bibliotecas CUDA compatíveis com essa linguagem: PyCUDA, CopperHead, Reikna e SciKits.CUDA.

- O PyCUDA é uma das biliotecas CUDA mais conhecidas para Python e que continua a ser mantido. Possui duas formas de integrar código CUDA dentro do Python, como o SourceModule (para escrever um kernel em CUDA C) ou GPUArray (para fazer operações básicas com matrizes que são carregadas para a GPU). Existem bibliotecas externas que integram funções CUDA no PyCUDA.
- O scikit-CUDA é também mantido, utiliza o PyCUDA como base, mas integra algumas funções CUDA da nVidia, como o cuBLAS, cuFFT ou o cuSOLVER. (Scikit-cuda (2013))
- O CopperHead é uma outra biblioteca que tem o apoio da NVIDIA Research, mas não é muito atualizada.

- O Reikna é também mantido, utiliza o PyCUDA/PyOpenCL como base, mas não tem uma comunidade muito grande.

Optou-se por utilizar o PyCUDA versão 2014.1, porque é o mais utilizado pela comunidade e também porque as outras bibliotecas usam o PyCUDA como base. Uma vantagem do PyCUDA é de fazer automaticamente a compilação do código CUDA e o seu carregamento para a GPU quando é executado, ao contrário do CUDA em C em que é necessário compilar manualmente com o nvcc e depois correr a aplicação.

4.4.2.2 Transformada de Fourier (para o Pansharpening)

Como foi descrito no capítulo anterior, o Pansharpening usa o Fourier para a geração de uma imagem "low pass"da Pancromática no Pansharpening. No Python, existem os seguintes módulos/bibliotecas FFT.

- Para a versão sequencial, pode-se contar com o Fastest Fourier Transform in the West (FFTW). O FFTW é uma biblioteca FFT bastante utilizada para C que usa alguns dos algoritmos mencionados anteriormente, e é rápida porque usa planos de execução para acelerar a transformada (Ruobing Li (2012)).

- Para a versão paralela com CUDA existe o cuFFT e o pyfft: o cuFFT é da nVidia, usa os algoritmos Cooley-Tukey e Bluestein e está integrado no scikit-CUDA (Scikit-cuda (2013)), o pyfft é uma biblioteca para o pyCUDA (PyFFT (2013)) que é instalado à parte e utiliza uma implementação melhorada do algoritmo FFT de Cooley-Tukey para OpenCL, descrita no artigo seguinte (Vasily Volkov, Brian Kazian (2008)), mas não é mais mantida porque foi integrada no Reikna.

Para a implementação sequencial, existe uma versão Python dessa biblioteca com o nome de pyFFTW que vai ser usada para a versão sequencial do Pansharpening.
Para a implementação paralela, foi realizado um teste no Pansharpening usando o pyfft e o cuFFT para avaliar o tempo de processamento de ambas. A conclusão desse teste é que o pyfft é mais rápido porque usa uma implementação melhorada do algoritmo que aproveita ao máximo os recursos da GPU. (Ruobing Li (2012))

4.4.2.3 Outras bibliotecas usadas

Outras bibliotecas Python foram usadas:

- NumPy: usado em todos os casos para calcular matrizes.

- GDAL: usado para abrir e criar ficheiros TIFF e fazer a vetorização sequencial usando Polygonize. Na versão paralela, é usado só para obter informação geográfica de uma imagem e para criar ficheiros TIFF com base nessa informação.

- OGR: usado na vetorização sequencial para criar o ficheiro shp.

- tifffile: usado para abrir ficheiros TIFF na versão paralela.
- Scikits.Learn: usado para o processamento K-Means na versão sequencial.
- Rasterio: usado para obter informação geográfica de uma imagem na vetorização
- Fiona/Shapely: usado na vetorização paralela das linhas da costa.
- MPI4py: módulo MPI para Python usado para paralelização CPU.
- cukmeans: implementação K-Means para CUDA e usado na versão paralela.

4.4.3 Implementação Sequencial

Antes de se proceder ao desenvolvimento dos algoritmos de paralelização, foi decidido avançar com o desenvolvimento dos algoritmos sequenciais. A ideia desta abordagem era conseguir replicar os algoritmos originais do OTB/SAGA, usando Python, para depois ser mais fácil e directo converter os mesmos para a versão paralela.

4.4.3.1 Pansharpening

A versão sequencial deste algoritmo utiliza os módulos Python NumPy, GDAL e o pyFFTW, e segue a implementação definida no Orfeo Toolbox Cookbook. Numa primeira fase, a imagem pancromática é aberta com o GDAL e carregada para uma matriz NumPy, a seguir é criado também o ficheiro Pansharpening para guardar a imagem final. De seguida, a imagem pancromática é dividida em altura por 17 partes iguais (cada uma com tamanho de 15462 pixeis de largura por 926 pixeis de altura) por forma a não sobrecarregar a memória da máquina.
Cada parte é primeiro processada com uma transformada de Fourier usando o pyFFTW, para depois aplicar a máscara low-pass para filtrar as frequências mais altas, e no final aplicar a transformada inversa com o pyFFTW para se ter a parte dessa imagem PanSmooth. Depois é feita a divisão da parte Pancromática com a parte PanSmooth, e para cada banda da multiespectal, multiplicar o resultado da divisão pela sua respetiva parte da banda e escrever logo para a imagem final.

4.4.3.2 MNDWI

A versão sequencial deste algoritmo utiliza os módulos python NumPy e GDAL. Numa primeira fase, a imagem Pansharpening é aberta com o GDAL e carregada para uma matriz NumPy, a seguir é criado também o ficheiro MNDWI para guardar a imagem final. Da mesma forma como na pancromática, a imagem pansharpening é dividida em altura por 17 partes iguais (cada uma com tamanho de 15462 pixeis de largura por 926 pixeis de altura) por forma a não sobrecarregar a memória da máquina.
Para cada parte é feita o seu cálculo, mas para isso seria necessário obter as bandas 3 (GREEN) e 6 (MIR) da imagem Pansharpening. Após essa obtenção, pode-se proceder à sua divisão e depois escrever para a imagem final.

4.4.3.3 K-Means

A versão sequencial deste algoritmo utiliza os módulos python NumPy, GDAL e sklearn.cluster. Inicialmente, a imagem MNDWI é aberta com o GDAL e carregada para uma matriz NumPy, a seguir é criado também o ficheiro K-Means para guardar a imagem final. De seguida, será usada uma pequena amostra de 15462 pixeis obtida no centro da imagem e com ela, pode-se então começar a calcular os respetivos valores das classes (k=5) do K-Means para 1000 iterações. De seguida, a imagem é dividida em altura por 17 partes iguais (cada uma com tamanho de 15462 pixeis de largura por 926 pixeis de altura) e para cada uma é calculada a respetiva previsão (predict) onde para cada pixel é definida a sua classe. Neste algoritmo, a zona de água é representada pela classe 1.
O sklearn.cluster (SciKit Learn (2014)) usa o algoritmo de Lloyd garantindo assim uma complexidade razoável. Foi definido um número fixo no seed para permitir a geração fixa de números aleatórios para as suas classes, evitando assim geração de imagens/resultados diferentes em cada execução deste algoritmo.

4.4.3.4 Vetorização

A versão sequencial deste algoritmo utiliza os módulos python Numpy, GDAL e OGR. Inicialmente, a imagem K-Means é aberta com o GDAL e carregada para uma matriz NumPy. De seguida, com base nessa imagem, é criada uma máscara que representa as zonas com água a branco e as outras zonas a preto. Só com essa máscara é que se pode fazer a vetorização com o gdal.Polygonize para gerar o ficheiro com o polígono da linha da costa.

4.4.4 Implementação Paralela

4.4.4.1 Problemas encontrados

Durante a implementação, foram encontrados alguns problemas e limitações que tornaram a sua implementação e integração difíceis:

- O primeiro problema encontrado é o GIL (Global Interpreter Lock) que é um lock (ou mutex) que limita o acesso a uma rotina do código a uma só thread, enquanto as outras threads esperam que essa thread acabe. A Python implementou esse lock porque a gestão de memória do Python não é "thread-safe", o que traz o risco de corrupção dos dados em memória quando várias threads acedem ao mesmo tempo à memória (Python (2014)). Neste caso, o GIL limita bastante a paralelização, tornando a sua execução mais demorada que a versão sequencial. Uma solução seria remover o GIL na totalidade, o que foi feito uma vez, mas os resultados obtidos foram dececionantes (Guido van Rossum (2007)). Outra solução seria reimplementar o GIL para permitir mais flexibilidade na paralelização e melhores tempos de processamento. (David Beazley (2010))

- O outro problema é a limitação na velocidade da leitura e escrita de imagens do GDAL. A gestão de memória do Python não é "thread-safe", por essa razão o GDAL tem implementações "thread-safe" das suas drivers TIFF, file handlers e block caches (Blake Thompson - Mapbox (2015)). A versão 2 do GDAL vai trazer suporte multithreading para a leitura e escrita (GDAL (2015)), mas neste trabalho só foi usado o GDAL 1.11.2, disponível nos repositórios UbuntuGIS.

Para os problemas mencionados anteriormente, foram procuradas soluções capazes de resolver estes dois problemas, sem comprometer os tempos de execução:

- A solução possível para o problema do GIL seria utilizar MPI para permitir que vários processos a executar o mesmo código possam comunicar entre si por mensagens (envio e receção de dados). Existe um artigo (Li-Jun ZHAN, Cheng-Zhi QIN (2013)) que descreve uma implementação que usa o MPI juntamente com o GDAL.
- A solução para a leitura e escrita do GDAL passa pela utilização do módulo python tifffile por ser mais rápida. Mas esta biblioteca duas limitações:
  - Não permite gravar partes para a imagem de saída como no GDAL, sendo necessário ter a imagem de saída em memória para depois ser gravada para o disco.
  - Não dá para gravar algumas informações geográficas da imagem. Esta questão é importante, uma vez que no final do processamento (na vetorização) a linha da costa depende dessa informação e sem ela, acaba por não aparecer corretamente posicionada. Para contornar isso, é necessário abrir a imagem no início com o GDAL, guardar essa informação numa variável em memória e fechar a imagem para não ocupar memória. Essa informação depois é usada quando for criada uma imagem ou ficheiro de saída.

4.4.4.2 Pansharpening

A versão paralela deste algoritmo utiliza os módulos python MPI4py, Numpy, pyCUDA, pyfft, GDAL e tifffile. O código da versão paralela usa como base o código da versão sequencial, mantendo o número de partes a dividir a imagem. A utilização do MPI nesta versão serve para contornar a tal limitação do GDAL e que está preparada para funcionar só com 3 processos (-np 3) por se ter 2 ficheiros de entrada e 1 ficheiro de saída.

Os três processos estão programados da seguinte forma:

- Rank 1: Processo que lê a imagem multi-espectral com o tifffile e para cada banda da imagem são enviadas partes para o processo principal.
- Rank 2: Processo que lê a imagem pancromática com o tifffile e aplica o low-pass filter em que processa cada parte primeiro com uma transformada de Fourier usando o pyfft, para depois aplicar a máscara low-pass para filtrar as frequências mais altas,

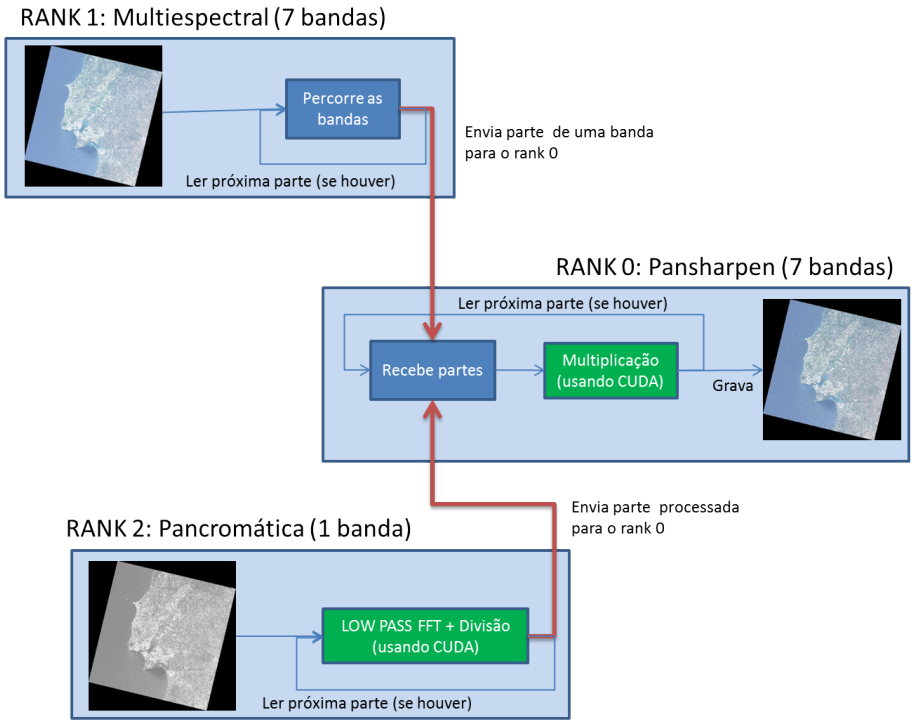

Figura 4.3: Diagrama do algoritmo paralelizado do Pansharpening

e no final aplicar a transformada inversa com o pyfft para se ter a parte dessa imagem PanSmooth. Depois faz a divisão da parte Pancromática com a parte PanSmooth e envia essa parte para o processo principal.

- Rank 0: É o processo principal que recebe as partes do rank 1 e do rank 2, faz a multiplicação delas e grava essa parte para a imagem usando o GDAL.

As multiplicações e divisões com matrizes são feitas com recurso ao gpuarray do pyCUDA, em que primeiro a matriz é carregada da memória para o GPU, depois fazer a multiplicação/divisão como se fosse uma matriz NumPy, no final descarregar a matriz do GPU para a memória usando get(). No início, para gravar as informações geográficas da imagem, escolheu-se o Rank 0 para primeiro abrir a imagem pancromática com o GDAL, guardar essa informação numa variável em memória e depois fechá-la.

4.4.4.3 Modified Normalised Difference Water Index (MNDWI)

A versão paralela deste algoritmo utiliza os módulos python MPI4py, NumPy, pyCUDA, GDAL e tifffile. Mais uma vez, foi necessário recorrer ao MPI para acelerar a paralelização, tal como no Pansharpening, mas só usando dois processos, porque só se tem um ficheiro de entrada e um de saída.

Os dois processos estão programados da seguinte forma:

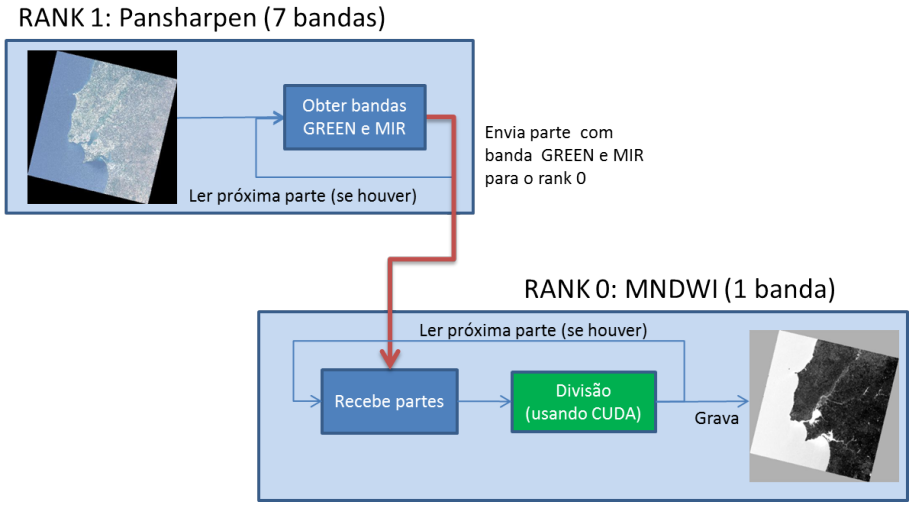

Figura 4.4: Diagrama do algoritmo paralelizado do MNDWI

- Rank 1: É o único processo que vai ler do ficheiro do pansharpening as bandas GREEN (banda 3) e MIR (banda 6) e manda-as para o rank 0.
- Rank 0: É o processo principal que recebe as partes do rank 1, faz o respetivo cálculo do MNDWI e grava essa parte para a imagem usando GDAL.

Por forma a tirar partido do CUDA, a soma, subtração e divisão é feita usando novamente o gpuarray do pyCUDA e aplicando essas operações como se fossem matrizes NumPy. No início, para gravar as informações geográficas da imagem, escolheu-se o Rank 0 para primeiro abrir a imagem pansharpened com o GDAL, guardar essa informação numa variável em memória e depois fechá-la.

4.4.4.4 K-Means

A versão paralela deste algoritmo utiliza os módulos python MPI4py, pyCUDA, NumPy, GDAL, tifffile e cukmeans. O cukmeans é uma implementação do K-Means em pyCUDA desenvolvida por Ludwig Schmidt-Hackenberg (Ludwig Schmidt-Hackenberg (2014)). O cukmeans usa uma interface similar ao do scipy k-means, e foi escolhida para este algoritmo porque é uma implementação pyCUDA do K-Means mais recente e outras implementações podiam trazer problemas de compatibilidade com a versão atual do pyCUDA por serem mais antigas. Mantendo a regra, foi necessário recorrer ao MPI para acelerar a paralelização, usando dois processos, porque só se tem um ficheiro de entrada e um de saída.

Os dois processos estão programados da seguinte forma:

- Rank 1: Lê imagem MNDWI usando tifffile, calcula o KMeans com base numa pequena amostra, e para cada parte determina o prediction. No final, essa parte é enviada para o rank 0.

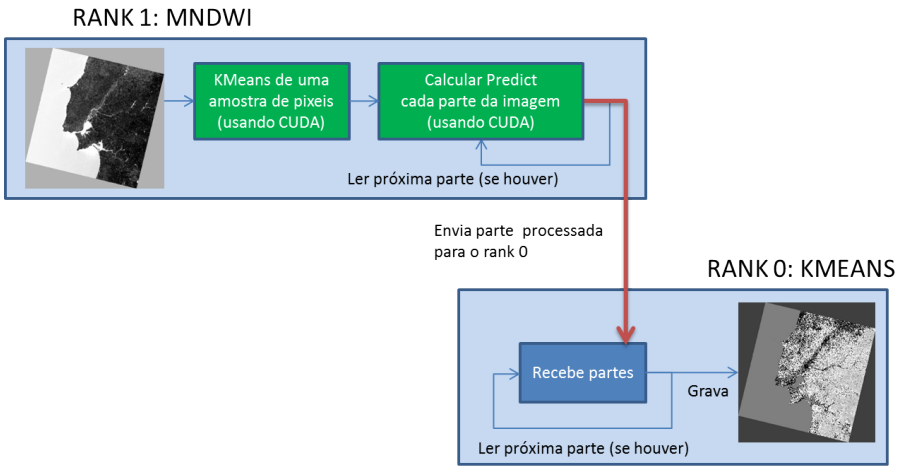

Figura 4.5: Diagrama do algoritmo paralelizado do K-Means

- Rank 0: É o processo principal que recebe partes do rank 1 e grava-as para a imagem final com o GDAL.

Relativamente à utilização do CUDA neste algoritmo, ao contrário dos anteriores algoritmos, foi utilizado o SourceModule no cálculo da previsão (predict). O SourceModule permite ao utilizador implementar um kernel na linguagem C, sem ter qualquer trabalho de o converter para Python. No início, para gravar as informações geográficas da imagem, escolheu-se o Rank 0 para primeiro abrir a imagem MNDWI com o GDAL, guardar essa informação numa variável em memória e depois fechá-la.

4.4.4.5 Vetorização

A versão paralela deste algoritmo utiliza os módulos python MPI4py, pyCUDA, NumPy, Rasterio, Fiona e shapely. No início do desenvolvimento desta versão, foi utilizado código existente no ficheiro rasterio_polygonize.py que faz a vetorização de uma imagem completa usando Rasterio e Fiona (está disponível para download na GitHub do rasterio). De seguida, sabendo que esta imagem vai ser dividida em 17 partes, foi modificada para executar a vetorização por partes da imagem e no final aplicou-se o cascaded_union do shapely para unir essas partes vetorizadas para formar um só vetor (polígono). No final, foi necessário recorrer ao MPI para acelerar a paralelização, usando só dois processos, porque se tem um ficheiro de entrada (imagem) e um de saída (ficheiro .shp)

Os dois processos estão programados da seguinte forma:

- Rank 1: Abre a imagem K-Means com o tifffile, e para cada parte dessa imagem, é gerada a sua respetiva máscara e enviada para o rank 0.

- Rank 0: É o processo principal que recebe partes do rank 1, faz a respetiva vetorização por partes e a sua união e grava para o disco usando o fiona.

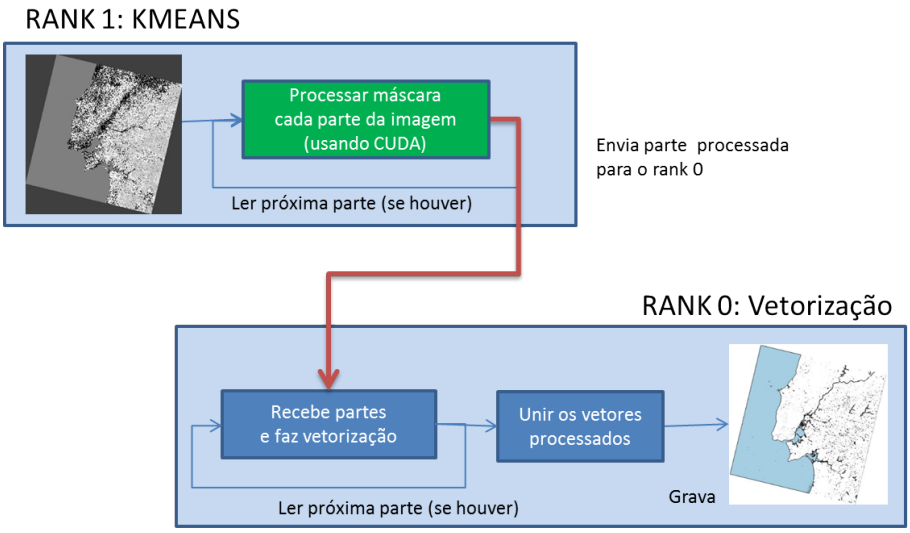

Figura 4.6: Diagrama do algoritmo paralelizado da Vetorização

Relativamente à utilização do CUDA neste algoritmo, foi utilizado o SourceModule do pyCUDA para correr um kernel para a criação de uma imagem de máscara que define as zonas com água (os pixeis com valor de classe 1) a branco e as restantes zonas a preto. No início, ao contrário das anteriores, para gravar as informações geográficas da imagem, escolheu-se o Rank 0 para primeiro abrir a imagem K-Means com o Rasterio, guardar essa informação numa variável em memória e depois fechá-la.

4.5 Resultados

Após a implementação dos algoritmos sequenciais e paralelos, foi feito um teste aos mesmos usando o conjunto de imagens da Lagoa de Óbidos obtidas no verão de 2014. Os resultados seguintes foram obtidos a partir das mesmas máquinas que executaram os algoritmos originais no capítulo anterior. De referir que estes tempos não incluem os tempos de criação das imagens multiespectral e pancromática, e que foi então assumido que elas foram previamente criadas.

- Desktop fixo do autor: AMD Phenom II x4 955BE 3.0Ghz, 4GB RAM, 500GB disco, Geforce GTX460 1GB, Ubuntu 14.04
- Máquina remota DI88: Intel Xeon E5506 2.13Ghz, 12GB RAM, 1.5TB disco, Geforce GTX680 2GB, Ubuntu Mate 14.04.

No geral, o tempo total permite provar que a paralelização usando o MPI e CUDA consegue reduzir o seu tempo para metade ou um quarto do tempo em relação ao script original (ou seja, tem-se um speedup de 2x a 4x). A razão para a ligeira subida de tempo ocorrida no DI88 na versão sequencial do MNDWI e Vetorização não é clara, podendo estar ligada à utilização da máquina por outros utilizadores.

	Desktop			DI88		
	Original	Sequencial	Paralela	Original	Sequencial	Paralela
Pansharpening	0:03:25	0:03:08	0:02:31	0:03:38	0:03:24	0:02:42
MNDWI	0:04:19	0:01:44	0:01:20	0:02:20	0:01:54	0:00:52
K-Means	0:01:24	0:00:56	0:00:31	0:00:39	0:00:57	0:00:41
Vetorização	0:08:22	0:02:05	0:00:23	0:01:34	0:01:56	0:00:30
TOTAL	0:17:30	0:08:01	0:04:45	0:08:11	0:08:11	0:04:45

Tabela 4.4: Tabela de tempos de processamento dos scripts original, com os algoritmos sequenciais e com os algoritmos paralelos nas máquinas Desktop e DI88.

4.5.1 Pansharpening

Figura 4.7: Imagem Pansharpening gerada na versão original (esquerda) e na versão paralela (direita) e sua respetiva amostra de nitidez

Os tempos foram razoáveis, só as imagens acabaram por ser ligeiramente diferentes. A imagem Pansharpening da versão original é mais limpa e nítida, mas ao implementar a versão paralela do Pansharpening tal como descrito no Orfeo Toolbox Cookbook, a sua imagem Pansharpening acabou por ficar menos nítida que a original (ou lapso na própria documentação e é um high-pass filter ou então foram feitas outras otimizações que

não foram documentadas). Também a imagem ficou mais ondulada por se ter aplicado o low-pass filter fourier com o CUDA a cada parte da imagem.

4.5.2 Modified Normalised Difference Water Index (MNDWI)

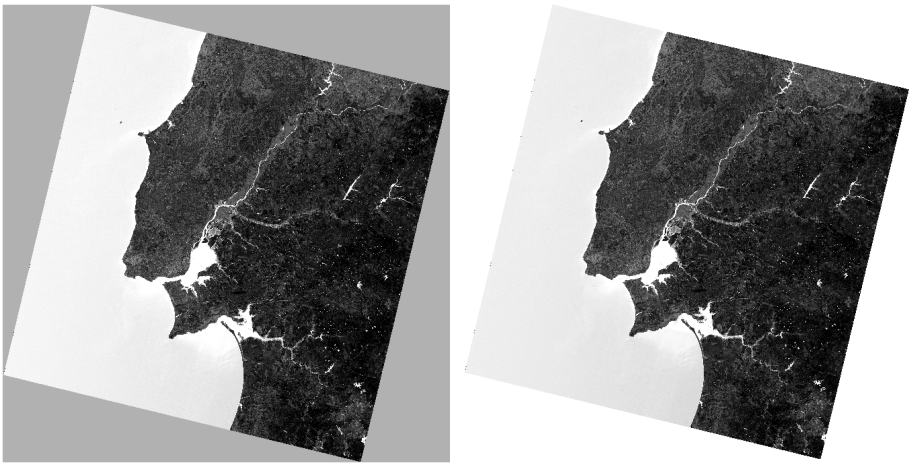

Figura 4.8: Imagem MNDWI gerada na versão original (esquerda) e na versão paralela (direita)

Apesar das diferenças de nitidez entre imagens Pansharpening, não foram notadas grandes diferenças nas imagens MNDWI geradas, acabando por se obter o mesmo resultado. Só a única pequena diferença são as margens cinza presentes na versão Original que não estão presentes na versão Paralela.

4.5.3 K-Means

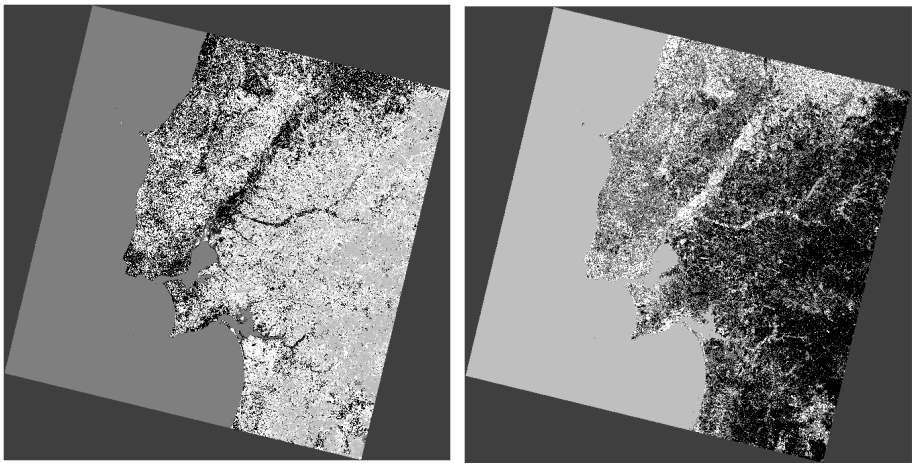

Figura 4.9: Imagem K-Means gerada na versão original (esquerda) e na versão paralela (direita)

A versão Paralela gerou uma imagem K-Means diferente da original. A razão para isso ter acontecido deve-se talvez a uma má escolha da amostra ou o tamanho da mesma. Por essa razão, as classes atribuídas à zona de água acabaram por ser diferentes em ambos os casos. Na versão original, o K-Means considerou-a como classe 2 e foi representada a cinza. Na versão paralela, o K-Means considerou-a como classe 3 e foi representada a cinza claro.

4.5.4 Vetorização

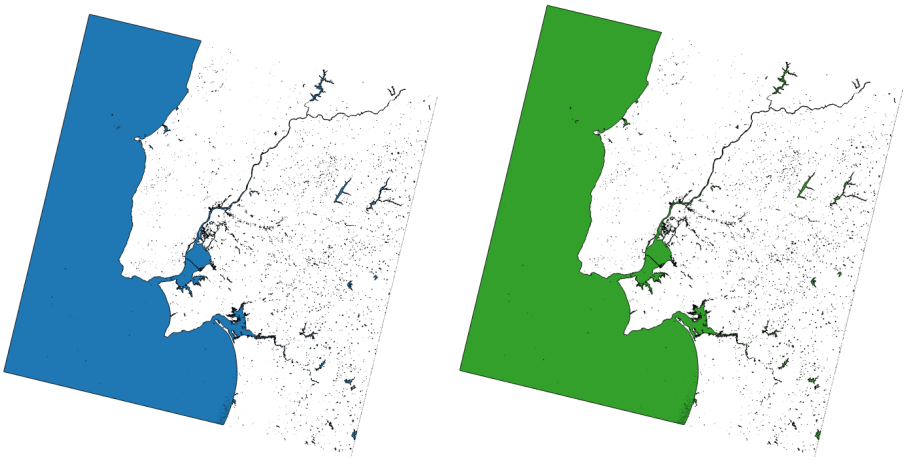

Figura 4.10: Vetor gerado na versão original (esquerda) e na versão paralela (direita)

A verdade é que não foi possível paralelizar na totalidade a vetorização usando o CUDA. Como alternativa, foi optado usar o Fiona na versão paralela em vez do Polygonize do GDAL na versão original. Apesar da versão Python do GDAL chamar as funções C/C++, é mais lento. O Fiona por outro lado foi desenvolvido em Python acabando por ser mais rápido (Sean Gillies - Fiona (2014)). Os vetores finais obtidos acabaram por ser iguais, não havendo grandes diferenças entre si.

CAPÍTULO 5

CONCLUSÕES

5.1 Resumo

O estudo e deteção das linhas da costa permite avaliar os impactos ambientais e sociais que esta traz para o país. A sua deteção era feita de forma manual pelo utilizador, obtendo as imagens geográficas e fazer o seu processamento manual para obter o ficheiro vetorial com a linha da costa.
A empresa onde o autor esteve a trabalhar tinha em desenvolvimento o Coastline Watch, um serviço online na cloud de monitorização que é capaz de detetar as alterações às dunas das praias e de determinar/validar o impacto da erosão e acreção ao longo da linha da costa com base na utilização de imagens obtidas por satélite. A vantagem deste serviço é de puder uma instância na cloud para tratar desse processamento, sem o utilizador ter de adquirir uma nova máquina.
Um dos problemas deste processamento manual é de ser é pesado e demorado. Pesado porque requer uma máquina com mais memória RAM e um bom disco para processar imagens geográficas muito grandes, e demorado porque requer um CPU mais rápido. Esses dois problemas podem ser resolvidos com recurso à paralelização que aproveita do multi-core dos CPUs e das potencialidades das GPUs para acelerar o processamento.
Nesta dissertação, foram estipulados os seguintes objetivos: desenvolver o Coastline Watch e desenvolver as versões paralelizadas dos algoritmos de processamento usando CPUs e GPUs.

5.1.1 Coastline Watch

No desenvolvimento do Coastline Watch, foi feito um estudo das soluções cloud e frameworks existentes para integrar o frontend e backend do serviço no capítulo 2. O resultado desse estudo é apresentado no capítulo 3, em que se escolheu o AWS para o Backend, o Heroku para a Frontend e o Ruby on Rails como framework para o desenvolvimento das páginas do serviço. O AWS tem um serviço free-tier que permite usar alguns dos serviços da Amazon gratuitamente por um ano, o Heroku tem um serviço free suficiente para alojar a frontend do serviço e o Ruby on Rails foi escolhido por ser muito usado para o desenvolvimento web.
Neste momento, o serviço Coastline Watch está disponível online mas encontra-se inoperacional. O AWS do backend teve que ser cancelado pela empresa por cobranças por funcionalidades extra que não estavam abrangidos pelo free tier. Nesta dissertação, estava planeado fazer-se um ensaio da sua disponibilização em máquinas virtuais com GPUs disponíveis na Cloud, mas não chegou a ser realizado por falta de disponibilidade e de recursos da empresa. Poderia ter sido interessante evoluir este serviço para suportar pelo menos uma instância cloud com processamento GPU para acelerar a deteção das linhas de costa, mas como descrito no capítulo 2, essas instâncias GPU envolvem custos adicionais.

5.1.2 Paralelização

No desenvolvimento das versões paralelizadas dos algoritmos, foram analisadas as diferentes formas de puder paralelizar o processamento e as soluções GPU existentes para a paralelização no capítulo 2. Durante o seu desenvolvimento, foram também encontrados dois problemas com o GDAL e com o GIL do Python que estavam a limitar a redução dos tempos de execução:

- Global Interpreter Lock do Python: O GIL limitava o speedup da paralelização porque a gestão de memória do Python não é "thread-safe".
- GDAL: As leituras e escritas feitas pelo GDAL que eram limitativas durante a paralelização por causa do GIL.

A ideia inicial seria só usar o CUDA, mas para se conseguir contornar esses dois problemas, foi necessário também incluir o MPI na paralelização. No capítulo 4, foi feito um estudo aos algoritmos de processamento por forma a saber que partes são possíveis de se paralelizar e para cada algoritmo implementou-se uma versão sequencial e uma versão paralela.
As versões paralelizadas dos algoritmos de processamento foram implementadas com sucesso, tendo sido obtidos resultados que permitiram demonstrar a redução dos tempos de processamento para menos de 5 minutos, usando MPI+CUDA.
Esta dissertação permitiu ao autor utilizar os seus conhecimentos obtidos nas cadeiras de

Computação de Alto Desempenho e de Sistemas de Computação em Cloud e aplicá-las num projeto a nível empresarial. Este projeto permitiu ao autor obter experiência na área de Sistemas de Informação Geográfica, para puder aplicar noutros projetos no futuro.

5.2 Trabalho futuro

No futuro, é possível fazer melhorias ou evoluções ao serviço Coastline Watch para puder integrar mais funcionalidades:

- Sistema para calcular métricas da linha de costa determinada: não chegou ainda a ser implementado, serve para comparar diferenças entre linhas de costa.
- Processar outras imagens: Possibilidade de processar imagens do satélite EEA Corine CLC e outros.
- Integrar uma instância GPU no serviço: só no caso de haver investimento e rentabilidade do projeto.
- Migrar o serviço para outro serviço frontend. Com as limitações do Heroku, para que a aplicação possa expandir mais, será necessário migrá-la para uma plataforma paga sem limitações de recursos.

Da mesma forma, serão também feitas melhorias ou evoluções à paralelização dos algoritmos de processamento no futuro, para diminuir os tempos de execução:

- Implementar de raiz a versão paralelizada da vetorização: até ao momento, não existe nenhuma implementação MPI+CUDA do algoritmo de vetorização.
- Implementar no Pansharpening o algoritmo para criar primeiro a imagem Multiespectral (XS) de 7 bandas usando MPI+CUDA: vendo os tempos obtidos no capítulo 4, é possível reduzir os seus tempos para metade usando a paralelização.

BIBLIOGRAFIA

AcuGIS (2014). *PostGIS - Spatial and Geographic Objects for PostgreSQL*. Última vez visitado a 23/09/2015. URL: http://postgis.net/.

Amazon (2011). *BitNami Cloud Hosting*. Última vez visitado a 15/11/2014. URL: aws.amazon.com/pt/blogs/aws/bitnami-cloud-hosting/.

Amazon (2014). *AWS | High Performance Computing - HPC Cloud Computing*. Última vez visitado a 07/02/2015. URL: http://aws.amazon.com/hpc/.

Amazon (2014a). *Free Tier AWS*. Última vez visitado a 18/10/2014. URL: aws.amazon.com/pt/free/.

Amazon (2014b). *IBM and Amazon Web Services*. Última vez visitado a 06/12/2014. URL: aws.amazon.com/ibm/.

Amdahl, G. M. (1967). "Computer Architecture and Amdahl's Law". Em: *AFIPS Conference Proceedings* 30. Reprinted in 2013, pp. 483–485.

BAI Hong-tao, HE Li-li, OUYANG Dan-tong, LI Zhan-shan, LI He (2009). *K-Means on commodity GPUs with CUDA*. Última vez visitado a 22/08/2015. URL: http://www.gpucomputing.net/sites/default/files/papers/908/K-MeansoncommodityGPUswit
pdf.

Barry Wilkinson, C. Michael Allen (2005). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson/Prentice Hall.

Beehive Locations - Copernicus Masters (2015). *BEEHIVE LOCATIONS - MONITORING HABITATS WITH SATELLITE DATA*. Última vez visitado a 12/12/2015. URL: http://www.copernicus-masters.com/index.php?kat=winners.html&anzeige=winner_dlr2015.html.

Benjamin Black (2009). *EC2 Origins*. Última vez visitado a 11/10/2014. URL: blog.b3k.us/2009/01/25/ec2-origins.html.

Bernardo Pires (2014). *Rails vs Django: An in-depth technical comparison*. Última vez visitado a 13/12/2014. URL: bernardopires.com/2014/03/rails-vs-django-an-in-depth-technical-comparison/.

Blake Thompson - Mapbox (2015). *Issues with Threading GDAL*. Última vez visitado a 30/07/2015. URL: https://github.com/mapbox/gdal-tiling-bench/issues/1.

Bluemix (2014). *Pricing*. Última vez visitado a 06/12/2014. URL: ace.ng.bluemix.net/\#/pricing/cloudOEPaneId=pricing.

Bradford, L. (2014). *Ruby on Rails vs Python and Django: Which Should a Beginner Learn?* Última vez visitado a 02/09/2015. URL: https://www.coursereport.com/resources/ruby-on-rails-vs-python-and-django-which-should-a-beginner-learn.

Center, H. D. (2015). *Using AWS S3 to Store Static Assets and File Uploads*. Última vez visitado a 13/09/2015. URL: https://devcenter.heroku.com/articles/s3.

Ceptu - Fieldsense (2014). *Easy crop monitoring using satellite technologies.* Última vez visitado a 12/12/2015. URL: http://www.ceptu.com/.

Craig A. Lee, Carl Kesselman, Stephen Schwab (1996). "Near-real-time Satellite Image Processing: Metacomputing in CC++". Em: *Computer Graphics and Applications, IEEE* 16(4), pp. 79–84.

CUDA (2015). *CUDA - Parallel programming and Computing platform*. Última vez visitado a 23/09/2015. URL: http://www.nvidia.com/object/cuda_home_new.html.

CyanoLakes - Copernicus Masters (2014). *CYANOLAKES - CYANOBACTERIA PUBLIC INFO SERVICE*. Última vez visitado a 12/12/2015. URL: http://www.copernicus-masters.com/index.php?kat=winners.html&anzeige=winner_ideas2014.html.

David Beazley (2010). *Understanding the Python GIL*. Última vez visitado a 30/07/2015. URL: http://www.dabeaz.com/GIL/.

Deepthi Nandakumar (2011). "Automatic translation of cuda to opencl and comparison of performance optimizations on GPUs". Tese de doutoramento. Graduate College of the University of Illinois.

Deltares (2015). *UNIBEST-CL+*. Última vez visitado a 08/12/2015. URL: https://www.deltares.nl/en/software/unibest-cl/.

DevBlogs - Nvidia (2015). *.NET Cloud Computing with Alea GPU*. Última vez visitado a 13/09/2015. URL: http://devblogs.nvidia.com/parallelforall/net-cloud-computing-with-alea-gpu/.

DHI (2015). *MIKE 21*. Última vez visitado a 08/12/2015. URL: http://www.mikepoweredbydhi.com/products/mike-21.

Django (2014). *Django*. Última vez visitado a 09/02/2015. URL: https://www.djangoproject.com/.

Engine Yard (2014a). *Stacks and Capabilities*. Última vez visitado a 15/11/2014. URL: www.engineyard.com/techstack.

Engine Yard (2014b). *Trial*. Última vez visitado a 22/11/2014. URL: www.engineyard.com/trial.

ESA (2014a). *Copernicus - Observing the earth*. Última vez visitado a 20/12/2014. URL: www.esa.int/Our_Activities/Observing_the_Earth/Copernicus/Overview4.

ESA (2014b). *SENTINEL-2 MSI Technical Guide*. Última vez visitado a 28/12/2014. URL: sentinel.esa.int/web/sentinel/sentinel-2-msi-wiki/-/wiki/Sentinel+Two/Level+1c+Product+Formatting.

ESA (2014c). *SENTINEL-2 MSI Technical Guide*. Última vez visitado a 28/12/2014. URL: sentinel.esa.int/web/sentinel/sentinel-2-msi-wiki/-/wiki/Sentinel+Two/Level+1c+Products.

FireHub - Copernicus Masters (2014). *FIREHUB - A SPACE-BASED FIRE MANAGEMENT HUB*. Última vez visitado a 12/12/2015. URL: http://www.copernicus-masters.com/index.php?kat=winners.html&anzeige=winner_bsc2014.html.

Gabriel E. Martinez Arroyo (2011). "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures". Tese de doutoramento. Faculty of the Virginia Polytechnic Institute e State University.

GDAL (2014a). *GDAL UTILITES*. Última vez visitado a 07/02/2015. URL: http://www.gdal.org/gdal_utilities.html.

GDAL (2014b). *gdalbuildvrt*. Última vez visitado a 11/01/2015. URL: www.gdal.org/gdalbuildvrt.html.

GDAL (2014c). *OGR UTILITES*. Última vez visitado a 07/02/2015. URL: http://www.gdal.org/ogr_utilities.html.

GDAL (2015). *GTiff – GeoTIFF File Format*. Última vez visitado a 03/09/2015. URL: http://www.gdal.org/frmt_gtiff.html.

Google (2014a). *Google Cloud SDK*. Última vez visitado a 25/10/2014. URL: cloud.google.com/sdk/.

Google (2014b). *Google Compute Engine - Frequently Asked Questions*. Última vez visitado a 09/02/2015. URL: https://cloud.google.com/compute/docs/faq.

Graciela Schneier-Madanes, Marie-Françoise Courel (2010). *Water and Sustainability in Arid Regions: Bridging the Gap Between Physical and Social Sciences*. Springer.

GRASS (2014). *GRASS GIS 6.4.5svn Reference Manual*. Última vez visitado a 07/02/2015. URL: http://grass.osgeo.org/grass64/manuals/index.html.

GRASS Wiki (2014). *GRASS and Python*. Última vez visitado a 07/02/2015. URL: http://grasswiki.osgeo.org/wiki/GRASS_and_Python.

Greensavers - Sapo.pt (2014). *Esposende, Faro e Olhão vão demolir 835 casas junto ao mar*. Última vez visitado a 04/10/2014. URL: greensavers.sapo.pt/2014/04/07/esposende-faro-e-olhao-vao-demolir-835-casas-junto-ao-mar/.

Guido van Rossum (2007). *All Things Pythonic - It isn't Easy to Remove the GIL*. Última vez visitado a 03/09/2015. URL: http://www.artima.com/weblogs/viewpost.jsp?thread=214235.

Hanqiu Xu (2006). "Modification of normalised difference water index (NDWI) to enhance open water features in remotely sensed imagery". Em: *International Journal of Remote Sensing* 27(14), pp. 3025–3033.

IBM SmartCloud Entry (2013). REST API Reference, version 3.2. IBM.

In-Kyu Jeong, Min-Gee Hong, Kwang-Soo Hahn, Joonsoo Choi, Choen Kim (2012). "Performance Study of Satellite Image Processing on Graphics Processors Unit Using CUDA". Em: *Korean Journal of Remote Sensing* 28(6), pp. 683–691.

Jackson, A. (2014). *Coastal Erosion*. Última vez visitado a 04/10/2014. URL: geographyas.info/coasts/coastal-erosion/.

Kamran Karimi, Neil G. Dickson, Firas Hamze (2009). "A Performance Comparison of CUDA and OpenCL". Tese de doutoramento. D-Wave Systems Inc.

Landsat (2013). *Landsat 8 Bands*. Última vez visitado a 17/01/2015. URL: landsat.gsfc.nasa.gov/?page_id=5377.

Li-Jun ZHAN, Cheng-Zhi QIN (2013). *Parallel Geospatial Raster Processing by Geospatial Data Abstraction Library (GDAL) — Applicability and Defects*. Última vez visitado a 19/08/2015. URL: http://www.geocomputation.org/2013/papers/30.pdf.

Ludwig Schmidt-Hackenberg (2014). *cukmeans.py - Kmeans in PyCUDA*. Última vez visitado a 22/07/2015. URL: https://github.com/shackenberg/cukmeans.py.

Mamta Bhojne, Anshu Pallav, Abhishek Chakravarti, Sivakumar V (2013). "High Performance Computing for Satellite Image Processing and Analyzing – A Review". Em: *International Journal of Computer Applications Technology and Research* 2(4), pp. 424–430.

MIKE by DHI (2014). *MIKE 21 Flow Model FM - Parallelisation using GPU - Benchmarking report*. Última vez visitado a 12/12/2015. URL: https://www.mikepoweredbydhi.com/-/media/shared\%20content/mike\%20by\%20dhi/flyers\%20and\%20pdf/product-documentation/gpu-benchmarking-report.pdf.

Multiscalelab (2009). *Swan: A simple tool for porting CUDA to OpenCL*. Última vez visitado a 17/01/2015. URL: gpgpu.org/2010/03/09/swan.

NASA (2014a). *Landsat Image Gallery*. Última vez visitado a 13/12/2014. URL: landsat.visibleearth.nasa.gov/.

NASA (2014b). *Landsat Image Gallery - Practical Uses*. Última vez visitado a 20/12/2014. URL: landsat.gsfc.nasa.gov/?page_id=2298.

Neil Middleton, Richard Schneeman (2013). *Heroku: Up and Running*. O'Reilly Media.

Nuno Duro, Gil Gonçalves (2014). "Remote sensing applications based on satellite open data (Landsat8 and Sentinel-2)". Em: *Conferência Nacional de Geodecisão - Barreiro*.

Nuno Duro, Gil R. Gonçalves, Ricardo Martins, António A. Silva (2015). "Collaborative and flexible processing infrastructure for Coastal Monitoring". Em: Submetido para o exp.at'15.

OpenCL (2015). *OpenCL - The open standard for parallel programming of heterogeneous systems*. Última vez visitado a 23/09/2015. URL: https://www.khronos.org/opencl/.

Orfeo Toolbox (2015). *The ORFEO Tool Box Software Guide Updated for OTB-5.0*. Última vez visitado a 17/07/2015. URL: https://www.orfeo-toolbox.org//packages/OTBSoftwareGuide.pdf.

Pavel Karas (2010). "GPU Acceleration of Image Processing Algorithms". Tese de doutoramento. Masaryk University Faculty of Informatics.

PCI Geomatics (2009). *GeoImaging Accelerator Pansharp Test Results*. Última vez visitado a 19/08/2015. URL: http://www.pcigeomatics.com/pdf/GXL_Pansharp_Test_Results.pdf.

PyFFT (2013). *PyFFT: FFT for PyCuda and PyOpenCL*. Última vez visitado a 22/08/2015. URL: http://pythonhosted.org/pyfft/.

Python (2014). *Global Interpreter Lock*. Última vez visitado a 30/07/2015. URL: https://wiki.python.org/moin/GlobalInterpreterLock.

R. Fisher, S. Perkins, A. Walker e E. Wolfart. (2003). *Fourier Transform*. Última vez visitado a 20/07/2015. URL: http://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm.

Rails, R. on (2014). *Ruby on Rails*. Última vez visitado a 09/02/2015. URL: http://rubyonrails.org/.

Reza Farivar, Daniel Rebolledo, Ellick Chan, Roy Campbell (2008). *A Parallel Implementation of K-Means Clustering on GPUs*. Última vez visitado a 23/08/2015. URL: http://www.researchgate.net/publication/221134288_A_Parallel_Implementation_of_K-Means_Clustering_on_GPUs.

Rodrigo Alonso, Sergio Nesmachnow (2012). "Parallel Computing Applied to Satellite Images Processing for Solar Resource Estimates". Em: *CLEI Electronic Journal* 15(3).

Ruobing Li (2012). *OpenCL Fast Fourier Transform*. Última vez visitado a 06/09/2015. URL: http://www.cs.nyu.edu/courses/fall12/CSCI-GA.2945-001/dl/ruobing-report.pdf.

S. K. McFeeters (1996). "The use of the Normalized Difference Water Index (NDWI) in the delineation of open water features". Em: *International Journal of Remote Sensing* 17(7), pp. 1425–1432.

SAGA (2014). *SAGA-GIS Module Library Documentation*. Última vez visitado a 07/02/2015. URL: http://www.saga-gis.org/saga_module_doc/2.1.4/index.html.

Scikit-cuda (2013). *Scikit-cuda*. Última vez visitado a 22/08/2015. URL: https://github.com/lebedov/scikit-cuda.

SciKit Learn (2014). *KMeans*. Última vez visitado a 22/07/2015. URL: http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

Sean Gillies - Fiona (2014). *The fiona user manual*. Última vez visitado a 22/08/2015. URL: http://toblerity.org/fiona/manual.html.

Serban Giuroiu (2013). *CUDA K-Means Clustering*. Última vez visitado a 22/08/2015. URL: http://serban.org/software/kmeans/.

Shaima AL-GABLI, Osman Hegazy (2014). *Coastline Change Detection on Tehama City Using Remote Sensing and Cloud Computing*. Última vez visitado a 12/12/2015. URL: http://www.researchpublish.com/download.php?file=Coastline\%20change\%20detection\%20on\%20Tehama\%20City-619.pdf\&act=book.

Sivakumar V., Ankit Goyal (2014). *Geospatial Information Extraction from multispectral sattelite imagery through GPU's based parallel computation approach*. Última vez visitado a 19/08/2015. URL: http://www.cibtech.org/J-ENGINEERING-TECHNOLOGY/PUBLICATIONS/2014/Vol-4-No-4/JET-010-011-SIVA_ANKIT_Manuscript_journal_format.pdf.

Softlayer (2015). *GPU Servers | | SoftLayer*. Última vez visitado a 13/09/2015. URL: http://www.softlayer.com/gpu.

Thilina Gunarathne, Bimalee Salpitikorala, Arun Chauhan (2011). *Optimizing OpenCL Kernels for Iterative Statistical Applications on GPUs*. Última vez visitado a 19/09/2015. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.300.4895&rep=rep1&type=pdf.

Thoughts on Cloud (2014). *Create and deploy a stand-alone Java application on IBM*. Última vez visitado a 22/11/2014. URL: thoughtsoncloud.com/2014/10/create-deploy-stand-alone-java-application-ibm-bluemix/.

Toby Velte, Anthony Velte, Robert Elsenpeter (2010). *Cloud Computing, A Practical Approach*. Última vez visitado a 09/02/2015. McGraw-Hill, Inc.

USGS (2014). *Landsat Processing Details*. Última vez visitado a 20/12/2014. URL: landsat.usgs.gov/Landsat_Processing_Details.php.

Vasily Volkov, Brian Kazian (2008). *Fitting FFT onto the G80 Architecture*. Última vez visitado a 06/09/2015. URL: http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf.

Weihua Sun, Bin Chen, David W. Messinger (2013). *Pansharpening of spectral image with anisotropic diffusion for fine feature extraction using GPU*. Última vez visitado a 19/08/2015. URL: http://spie.org/Publications/Proceedings/Paper/10.1117/12.2015084.

Yegor V. Pushkin, Rauf Kh. Sadykhov, Leonid P. Podenok, Andrey V. Dorogush, Valentin V. Ganchenko (2014). *Multispectral satellite images processing for forests and wetland regions monitoring using parallel MPI implementation*. Última vez visitado a 17/12/2015. URL: https://earth.esa.int/envisatsymposium/proceedings/posters/2P8/461312rs.pdf.