Pedro Miguel Galvão Farelo Lourenço

Licenciatura em Engenharia Informática

Programação Paralela Baseada em Skeletons para Processamento de Imagens 3D

Dissertação para obtenção do Grau de
Mestre em Engenharia Informática

Orientadores: Maria Cecília Gomes, Prof. Auxiliar,
Universidade Nova de Lisboa
Pedro Medeiros, Prof. Associado,
Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Carlos Damásio
Arguente:	Prof. Manuel Barata
Vogal:	Prof. Maria Cecília Gomes

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2016

Programação Paralela Baseada em Skeletons para Processamento de Imagens 3D

À minha família e amigos pelo apoio que me deram

AGRADECIMENTOS

Gostaria de agradecer aos meus orientadores, Professora Maria Cecília Gomes e Professor Pedro Medeiros, por todo acompanhamento, disponibilidade, apoio e ajuda que me permitiram completar esta tese.

Agradeço também à Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, por todos os anos que aqui passei que me permitiram aprender e crescer enquanto pessoa e profissional.

Agradeço ainda ao Bruno Preto pela disponibilidade que apresentou para me ajudar neste projeto.

Por último agradeço a toda a minha família e amigos, por todo o apoio e ajuda que me deram durante estes anos.

A todos o meu Obrigado.

RESUMO

A melhoria de desempenho obtida através da computação paralela permitiu o aumento da sua utilização na resolução de problemas computacionalmente exigentes em muitas áreas em ciência e engenharia.No entanto, devido à complexidade da criação de programas paralelos são necessárias ferramentas que simplifiquem o seu desenvolvimento. Um tipo de problemas que é resolvido com programação paralela é o processamento de imagem em áreas como a Ciência dos Materiais e a Medicina. À semelhança de outras áreas, também nestas e para este tipo de problemas, é possível encontrar soluções e estratégias de paralelização comuns, e que capturam o conhecimento acumulado ao longo do tempo. O conhecimento sobre estes padrões e a sua disponibilização permitem assim simplificar o desenvolvimento desses programas paralelos mas é necessário existirem ferramentas que os implementem com um desempenho adequado. Os padrões devem também ser de fácil adaptação e reutilização em problemas similares, melhorando a produtividade no desenvolvimento de programas em diversas áreas que necessitem de processamento de imagem.

No contexto da computação paralela, em geral, existem já ferramentas que disponibilizam padrões de paralelização permitindo que não peritos na área possam desenvolver os seus programas de um modo mais simples. Os *algorithmic skeletons* são uma das soluções existentes para capturar esses padrões, existindo *frameworks* que os implementam libertando os programadores da necessidade do conhecerem os detalhes da arquitetura alvo. Os *algorithmic skeletons* podem também ser aplicados aos problemas de processamento de imagem, capturando diretamente ou por composição padrões nesses domínio. No entanto, as ferramentas de *algorithmic skeletons* existentes não disponibilizam padrões otimizados com propriedades adaptativas que possam ter em conta, quer as características do sistema em execução (e.g. carga do sistema versus consumo de energia, etc.), quer da imagem em processamento (e.g. imagens com mais ou menos objetos).

Neste contexto, este trabalho começou por estudar e comparar as implementações de um algoritmo de processamento de imagem usando dois *framework* de *algorithmic skeletons* que permitem gerar código para GPGPUs, de modo a identificar os padrões subjacentes e o *framework* mais adequado. Seguiu-se como contribuição a extensão do *framework* FastFlow com uma arquitetura de medição do estado de execução do *skeleton farm*, e a extensão deste com propriedades adaptativas. É possível alterar o número de *workers* de uma *farm*,

controlar a distribuição de tarefas pelos vários *workers*, e escolher se a a execução do *skeleton* é feita em CPU ou GPU.

Palavras-chave: computação paralela, processamento de imagem, algorithmic skeletons

Abstract

With the performance improvement resulting from parallel computing several areas other than Computer Science are increasingly developing parallel programs to solve their complex problems. However, due to the difficulty of creating parallel programs, the tools for their development need to offer mechanisms that may simplify that task to non-experts in the area.

Image processing in domains such as Materials Science and Medicine is an example of a type of problem that benefits from parallel programming. Moreover, there is already an accumulated knowledge in the area in terms of effective solutions and parallelization strategies that it is useful to capture and reuse. This allows reducing the effort of adapting those solutions to similar problems, most of all to novel users in the area of image processing. Nevertheless, there is a lack of tools in the domain that may give support to such reuse of solutions while guaranteeing an adequate performance or which have to conform to some pre-defined quality of service.

In the area of parallel computing, there are already tools that allow representing common/proven solutions in the form of parallel patterns, simplifying the task of non-experts in parallel programming. Frameworks of Algorithmic Skeletons, which capture such parallel patterns, are an example of those tools where users simply select, compose and parameterise those skeletons, with no concern about the target parallel arquitectures details.

Similarly, algorithmic skeletons can be applied to image parallel processing problems, capturing patterns in this domain. However, the existing tools do not provide patterns specifically optimized for image processing; also, current frameworks do not support adaptive properties that can take advantage of the system features.

The contributions made by this thesis were the study and comparison of implementations of an image processing algorithm using two algorithmic skeletons frameworks that can generate GPGPU code, an extension of the FastFlow framework with an architecture that measures the state of the farm skeleton execution, and the extension of the farm skeleton with adaptive properties, which allow to change the number of farm workers, to control the distribution of tasks among the workers, and to choose the skeleton's backend.

Keywords: parallel computing, image processing, algorithmic skeletons

CONTEÚDO

Lista de Figuras

LISTA DE TABELAS

ACRÓNIMOS

API Application programming interface.

CUDA Compute Unified Device Architecture.

GPGPU General-purpose computing on graphics processing units.

GPU Graphics processing unit.

OMP OpenMP.

CAPÍTULO 1

INTRODUÇÃO

1.1 Motivação

O foco na computação paralela tem vindo a aumentar recentemente com o aparecimento de diversas arquiteturas de paralelismo intensivo como forma de compensar as dificuldades em melhorar o desempenho de um único processador (e.g. o aumento do número de transístores implica uma maior dispersão de energia que torna os circuitos instáveis, limitando o aumento do relógio do CPU). A existência dessas arquiteturas multi- e *many-core* permite um aumento muito significativo do desempenho de muitas aplicações, mas resultou num aumento da relevância no desenvolvimento de programas paralelos que saibam explorar as capacidades dessas arquiteturas, em particular para problemas que requerem o processamento de um grande número de cálculos ou de grande quantidade de informação, em tempo aceitável.

A criação de programas paralelos é porém mais complexo do que a criação de programas sequenciais, dado que requer um conhecimento da arquitetura alvo e do seu modelo de programação. Isto inclui saber como fazer uma boa divisão dos dados bem como coordenar diferentes unidades de execução (processos ou *threads*). A programação paralela apresenta assim problemas e dificuldades que não estão presentes na programação sequencial, o que faz com que seja mais fácil introduzir erros no programa. Tal pode levar a que as operações não executem na ordem pretendida e se obtenham resultados errados, bem como obrigar a que as várias *threads* fiquem bloqueadas à espera de recursos usados por outras *threads*, impedindo assim que o programa termine a sua execução.

Também para facilitar e agilizar a criação de programas no domínio do processamento de imagens, é útil que o desenho da aplicação seja mais centrado no domínio do problema e não nos detalhes da sua paralelização, que varia de arquitetura para arquitetura. Tal como acontece em diversas áreas que necessitam de computação paralela, os programas de processamento de imagem apresentam esquemas recorrentes ou padrões nas soluções

utilizadas, tanto a nível de algoritmos como a nível de estratégias de paralelização.

O uso destes padrões permite simplificar o desenvolvimento de várias soluções desta área dado que os novos utilizadores podem reutilizar o conhecimento adquirido por peritos na área da paralelização. Basta para isso conhecer os padrões disponíveis (que formam a base da construção de um vocabulário na área) e como os aplicar, que os detalhes da sua implementação são transparentes, facilitando assim a portabilidade dos programas. Torna-se assim útil disponibilizar ferramentas que implementem essas soluções recorrentes tal que sejam facilmente parametrizadas para o problema em questão ou adaptadas para problemas similares. Tal é vantajoso mesmo que estas soluções possam ser menos eficientes que as soluções dedicadas, dado que aumentam a produtividade dos programadores.

No contexto da programação paralela existem já soluções que capturam padrões de paralelização, disponibilizando-os sob a forma de abstrações no contexto de uma linguagem de programação ou de um *framework*. Os *frameworks* de *algorithmic skeletons* são uma dessas soluções em que o *skeletons* disponíveis representam soluções recorrentes, ou padrões,de programação paralela. Um *skeleton* particular captura as comunicações e interações (e.g. dependências de dados) presentes num padrão, escondendo do programador os detalhes da sua instanciação numa arquitetura particular. O desenvolvimento de programas paralelos consiste assim na seleção, composição e instanciação desses *algorithmic skeletons* para o problema que se pretende resolver, cabendo ao *framework* a geração automática de código para uma ou mais arquiteturas. Mesmo os programadores menos experientes podem assim paralelizar as suas aplicações de modo mais simples com a vantagem de o seu código ser mais facilmente portável para diferentes arquiteturas, dependendo das potencialidades da ferramenta de suporte.

O uso de *algorithmic skeletons* é assim uma possível solução na simplificação de programas de processamento paralelo de imagens, com a vantagem de o código representado pelos *skeletons* ser implementado por peritos na área e ter sido previamente testado, apresentando por isso uma menor probabilidade de existirem erros de programação. A utilização de *algorithmic skeletons* no desenvolvimento de programas paralelos tem sido inclusivamente estudada em vários trabalhos revelando-se uma estratégia promissora para o aumento da produtividade dos programadores.

1.2 Problema

Os *frameworks* baseados em *skeletons* não garantem em geral um desempenho ao mesmo nível que uma solução otimizada para uma arquitetura alvo. Embora os *skeletons* sejam parametrizados de acordo com o problema particular em que são usados, representam sempre soluções mais genéricas que não são tão eficientes como as possíveis por parte de um programador experiente, com conhecimento profundo de uma arquitetura particular e dos detalhes do modelo de dados do algoritmo a paralelizar. Outro problema é a necessidade do desenho da aplicação ter de se adaptar aos padrões disponíveis, e caso isso não aconteça, a única possibilidade é adaptar os *skeletons* já existentes. As ferramentas

com base em *skeletons* têm assim de ser flexíveis, permitindo a criação de outros *skeletons* genéricos ou *skeletons* otimizados mais próximos do domínio de aplicação, que possam por sua vez ser reutilizados noutras aplicações.

O problema geral, no qual este trabalho se insere, é assim contribuir para simplificar o desenvolvimento de aplicações paralelas no contexto de processamento de imagem e por utilizadores não informáticos, com recurso a uma abordagem baseada em *algorithmic skeletons* mais próximos deste domínio de aplicação, sem comprometer o desempenho das aplicações. Este problema geral está a ser tratado no contexto de uma tese de doutoramento em curso, a qual pretende tirar partido da programação paralela estruturada para simplificar desenvolvimento de aplicações para o processamento de *streams* de imagens 3D. Para tal, pretende-se disponibilizar uma arquitetura de suporte a *skeletons* com propriedades autonómicas que possam tirar partido quer das características do sistema em execução quer das características das imagens em processamento. O trabalho em desenvolvimento está ainda nas suas fases iniciais requerendo o estudo aprofundado das potencialidades e limitações apresentadas pelos *frameworks* de *skeletons* correntes, bem como de que modo soluções de reconfiguração dinâmica neste contexto, podem ser combinadas com soluções mais próximas do domínio de aplicação, e.g. que tenham em conta as características das imagens a processar.

Tendo em vista contribuir para a resolução do problema geral acima descrito, o trabalho apresentado nesta tese teve um carácter exploratório e centrou-se num conjunto restrito de problemas. O primeiro foi identificar algoritmos básicos na área de processamento de imagem, tal que a sua disponibilização possa ser útil em diversos problemas, incluindo em áreas distintas. Segundo, identificar que ferramentas na área da programação paralela estruturada, i.e. baseadas *algorithmic skeletons*, são passíveis de ser alteradas tendo em vista: a execução em arquiteturas heterogéneas; recolha de dados em tempo de execução como suporte a alterações dinâmicas necessárias; permitir oferecer soluções que tenham em conta as características das imagens e tirem partido das características do sistema em tempo de execução. Terceiro, contribuir para o estudo das características/componentes iniciais de uma arquitetura de suporte a propriedades adaptativas para o processamento de imagens. Quarto, implementação de versões iniciais destes componentes no contexto de um *frameworks* de *skeletons* particular, tendo em vista não degradar o desempenho das aplicações.

1.3 Solução proposta

O primeiro objetivo deste trabalho foi estudar um conjunto de problemas típicos no contexto do processamento de imagem e avaliar a sua paralelização com base em programação estruturada. Deste modo, foi estudada a classe de problemas de segmentação de imagem bem como as soluções de paralelização possíveis usando padrões paralelos e o seu mapeamento em *algorithmic skeletons* comuns.

Como não seria possível avaliar a otimização via reconfiguração dinâmica de um conjunto alargado de padrões comuns no contexto do processamento de imagem, o trabalho focou-se nesse padrão particular que captura algoritmos de segmentação de imagem. Isto vem do facto de se tratarem de estratégias recorrentes usadas e de a segmentação ser um padrão sobre o qual existe conhecimento prévio no grupo de investigação no qual este trabalho se insere. Em concreto, foi avaliado um algoritmo particular de segmentação denominado *ObjectIdentifier* descrito em [41]. Foram estudados os seus detalhes em termos de estratégias de paralelização usadas, como forma a identificar as estratégias recorrentes que correspondessem a padrões de paralelização mais simples.

O segundo objetivo foi avaliar quais ferramentas na área da programação paralela estruturada poderiam servir de base ao trabalho a desenvolver. Para tal, o algoritmo *ObjectIdentifier* foi implementado em dois *frameworks* de *skeletons* parametrizáveis denominados FastFlow [1] e SkePU [21], o que permitiu um primeiro estudo mais aprofundado destes *frameworks* e a escolha do *framework* Fastflow como plataforma de desenvolvimento deste trabalho.

O terceiro objetivo foi identificar os componentes básicos para a construção de uma arquitetura com propriedades autonómicas no contexto da programação paralela estruturada, e o quarto proceder à sua implementação e avaliação no contexto do *framework* escolhido. Assim, a solução proposta por este trabalho é descrita no capítulo 4 e consiste na extensão do *framework* FastFlow com um *skeleton* com propriedades adaptativas, ou reconfiguração dinâmica (e.g. permitindo no futuro garantir uma certa qualidade de serviço). O padrão escolhido foi o *skeleton farm* dado que é um padrão básico para o processamento de imagens digitais cujo modelo de execução se mapeia tanto no processamento em CPU como em GPU. As propriedades adaptativas inicialmente consideradas foram: alterar o número de *workers* em execução numa *farm*; modificar a divisão dos dados pelos vários *workers*; e definir se se usa CPU ou GPU (ou ambos) no processamento de dados. A estratégia particular que foi implementada garante um equilíbrio de carga pelos diferentes *workers* de uma *farm*, com base nos estados de cada *worker*. A terceira propriedade foi também implementada permitindo a execução de uma *farm* ou em CPU ou em GPU, ficando para trabalho futuro a divisão do trabalho entre CPU e GPU.

Por forma a implementar as propriedades adaptativas descritas, um dos componentes básicos identificados e implementados suporta uma arquitetura de medição do estado do sistema, em particular no contexto de execução de um *skeleton farm*. Esta arquitetura é fundamental para a reconfiguração dinâmica de uma *farm* particular, de modo a saber que tipo de reconfigurações são necessárias em cada instante, ou quais as mais indicadas para cada tipo de parametrização. No entanto, a arquitetura de medição foi desenhada de maneira a ser genérica o suficiente para servir de base à reconfiguração dinâmica de outros *skeletons* que possam incluir outras propriedades adaptativas adicionais, por exemplo, *checkpointing* do estado de execução de um padrão paralelo particular, por forma a que seja substituído por outro padrão paralelo. A validação da solução proposta é feita no capítulo 5, sendo usado o *framework* estendido para implementar o algoritmo de segmentação

ObjectIdentifier.

1.4 Contribuição da tese

O trabalho desenvolvido nesta tese tem como objetivo geral contribuir para simplificar a programação de aplicações na área de processamento de imagem, com recurso a estratégias de programação paralela estruturada, sem comprometer o seu desempenho. Neste contexto, foram feitas as seguintes contribuições:

- Estudo da viabilidade da implementação de um algoritmo de segmentação de imagem denominado *ObjectIdentifier* em duas ferramentas baseadas em *skeletons* que geram código para GPGPUs, e sua comparação.
- Extensão do FastFlow, um *framework* baseado em *skeletons*, cada um representando um padrão de execução paralela recorrente, e suportado por esqueletos de código para CPU e GPU. Uma das extensões consistiu no desenvolvimento de uma arquitetura de medição do estado de execução de um *skeleton* particular, em concreto de uma *farm*.
- Extensão do padrão *farm* do *framework* FastFlow com as seguintes propriedades adaptativas:
  - reconfiguração dinâmica do número de *workers* de uma *farm*;
  - controlo do equilíbrio de carga dos *workers* de uma *farm* (distribuição dos trabalhos pendentes pelos *workers* em melhor estado);
  - escolha da execução da *farm* em CPU ou GPU.

  Estas extensões inserem-se no objetivo (futuro) mais geral de disponibilizar padrões paralelos com propriedades autonómicas que cumpram diferentes qualidades de serviço parametrizáveis, para a área de processamento de imagem.

A solução proposta foi validada, no contexto do *framework* FastFlow estendido, com a implementação do algoritmo *ObjectIdentifier*, um algoritmo paralelo para processamento de imagem com base em segmentação.

1.5 Descrição do documento

No capítulo 2 apresenta-se uma síntese do estado da arte da computação paralela, processamento de imagem, computação paralela estruturada, suas características e exemplos, e propriedades adaptativas. Neste capítulo faz-se uma descrição do que é cada tópico, bem como de algumas das suas características, sendo também dados exemplos de cada um.

O capítulo 3 apresenta uma descrição mais detalhada dos *frameworks* estudados e algumas das suas características. É aí apresenta também uma descrição do algoritmo *object*

identifier e sua implementação nos *frameworks* estudados. O capítulo finaliza com uma comparação entre os *frameworks*, sendo apresentadas as razões para o *framework* escolhido.

No capítulo 4 é feita a descrição da solução proposta usando a implementação do algoritmo *Object Identifier* usando as extensões feitas ao *framework* FastFlow.

No capítulo 5 é feita a validação da solução proposta usando a implementação do algoritmo *ObjectIdentifier* usando as extensões feitas ao *framework* FastFlow.

O capítulo 6 apresenta as conclusões sobre o trabalho desenvolvido e resultados obtidos. São apresentadas algumas propostas de trabalho futuro em termos de possíveis utilizações adicionais do trabalho desenvolvido, bem como em termos do melhoramento e expansão deste.

CAPÍTULO 2

ESTADO DA ARTE

Neste capítulo é descrito o que é a computação paralela, o processamento de imagem e a programação paralela estruturada, dado que são os três domínios que compõe o trabalho a realizar nesta tese. Para cada uma destas secções é descrita a sua relevância e limitações, e exemplos de aplicação. São ainda apresentadas as características relevantes de cada domínio,tais como as arquiteturas de computação paralela existentes e os seus modelos de programação, as operações de processamento de imagem mais relevantes, e os tipos de ferramentas que existem na programação paralela estruturada.

2.1 Computação paralela

Na computação paralela usam-se vários processadores em simultâneo para a resolução de um problema. Este tipo de computação contrasta com a computação sequencial onde o programador define a lógica do programa considerando que é executada uma operação de cada vez. Na computação paralela cada um dos processadores em uso pode executar as suas operações ao mesmo tempo do que os outros.

Nas sub-secções que seguem é feita a caracterização da computação paralela em termos da sua relevância, vantagens e limitações, arquiteturas e modelos de programação e a sua adequação a diferentes tipos de problemas. São ainda referidos os *algorithmic skeletons* como uma estratégia de simplificação da programação paralela (que serão apresentados com mais detalhe na secção 2.3 do documento), bem como diferentes métricas de desempenho usadas em computação paralela. Para cada métrica são apresentadas as fórmulas para o seu cálculo, bem como uma descrição de como estas fórmulas são obtidas.

2.1.1 Âmbito

Tradicionalmente, as melhorias de desempenho de um programa eram obtidas através do aumento da frequência de relógio do CPU. Recentemente, tal tornou-se cada vez mais difícil devido a problemas como a dificuldade de dissipar o calor gerado e o elevado consumo de energia [44]. A solução encontrada foi a produção de CPUs com vários núcleos de processamento. Isto aumentou o foco na computação paralela que passou a ser uma das principais formas de obter melhoria de desempenho.

A computação paralela permite tornar o processamento de dados mais rápido, ou processar uma quantidade maior de dados no mesmo tempo, quando comparado com a computação sequencial. Assim, é possível resolver problemas que não terminariam em tempo aceitável com processamento sequencial, como por exemplo a análise de grandes quantidades de dados na área de *data mining*. A computação paralela também permite resolver problemas de grande dimensão, onde existe grande dependência entre componentes que devem estar em execução ao mesmo tempo, existindo troca de informação entre eles. Um desses tipos de problemas é a previsão do tempo, onde é necessário calcular a evolução da pressão, da velocidade do vento e da temperatura, sendo estes cálculos dependentes entre si. Outro problema é o cálculo de sistemas de partículas, onde se pretende calcular a movimentação de cada partícula, sendo necessário saber as posições e forças exercidas pelas outras partículas em cada instante.

A computação paralela apresenta como principal vantagem o ganho de velocidade na resolução de problemas. No entanto, a criação de programas paralelos é mais complexa do que a criação de programas sequenciais. Um dos problemas presentes é garantir sincronia e consistência dos dados entre os vários processos que executam o programa. Isto faz com que seja mais fácil um programador introduzir erros no programa, especialmente quando apresenta pouca experiência na sua paralelização.

2.1.2 Arquiteturas

Em termos de arquiteturas a computação paralela pode fazer uso de arquiteturas de memória partilhada, memória distribuída, arquiteturas híbridas ou heterogéneas.

Arquiteturas de memória partilhada Neste tipo de arquiteturas existem vários processadores que operam de forma independente, ligados ao mesmo recurso de memória, tornando a partilha de dados entre processadores rápida e fácil. Quando um valor é atualizado por um dos processadores, todos os outros processadores têm acesso ao novo valor. No entanto, a partilha torna a sincronia entre processos mais difícil, visto que obriga a garantir que os dados são acedidos corretamente. Isto faz com que o programador tenha uma maior responsabilidade na garantia de sincronia entre processos. A figura 2.1.2 apresenta uma arquitetura de memória partilhada.

Este tipo de arquitetura não é facilmente escalável pois adicionar mais processadores cria mais tráfego no acesso a memória, e obriga a hardware mais dispendioso na construção

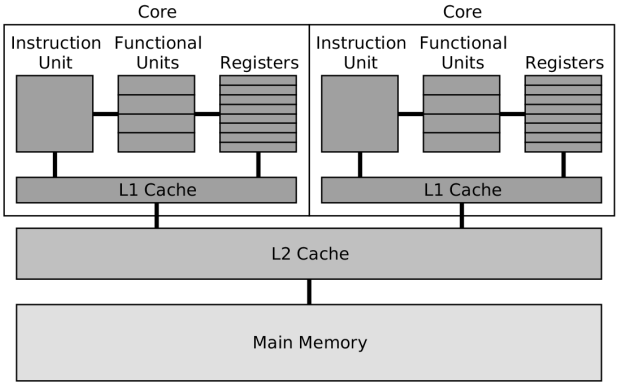

Figura 2.1: Arquitetura de memória partilhada. Retirado de [29]

da hierarquia de memória. No entanto, é ideal para problemas onde seja frequente a troca de informação entre processadores, por exemplo, no cálculo da difusão de calor pois para cada cálculo da temperatura de uma posição é preciso saber a temperatura calculada pelas posições adjacentes.

Arquiteturas de memória distribuída Nesta configuração cada processador tem uma memória local e comunica com os restantes processadores através de um dispositivo de entrada / saída (por exemplo uma interface de rede). A comunicação entre dois processadores é feita transferindo explicitamente os dados da memória de um processador para a do outro. Do ponto de vista do programador, os vários processos que compõem a aplicação comunicam entre si através de mensagens usando bibliotecas como o MPI [43].

Como cada processador tem uma memória local, as suas alterações não são visíveis nas memórias dos outros processadores. Isto permite ao programador não precisar de se preocupar com a possibilidade de outro processador ter alterado a memória e a mesma ficar com dados incoerentes. No entanto, obriga o programador a ser o responsável pela comunicação explícita entre processadores, sendo que também torna o acesso aos dados não uniforme, visto que aceder a dados presentes num processador remoto demora mais tempo do que aceder a dados locais. Um exemplo deste tipo de arquitetura são os *clusters*, que apresentam um grande número de computadores ligados entre si através da rede. A figura 2.1.2 mostra um *cluster*.

Este tipo de arquitetura permite aumentar mais facilmente o número de processadores, visto que cada um deles tem interface de memória separada. Assim sendo, é possível executar programas com um maior volume de dados. Devido aos custos da comunicação, este tipo de arquitetura é ideal para problemas onde não seja frequente a comunicação entre processadores, tais como inverter as cores de cada *pixel* de uma imagem. Como nesta operação cada *pixel* é independente dos outros, a imagem pode ser dividida e cada processador processa uma parte de forma independente.

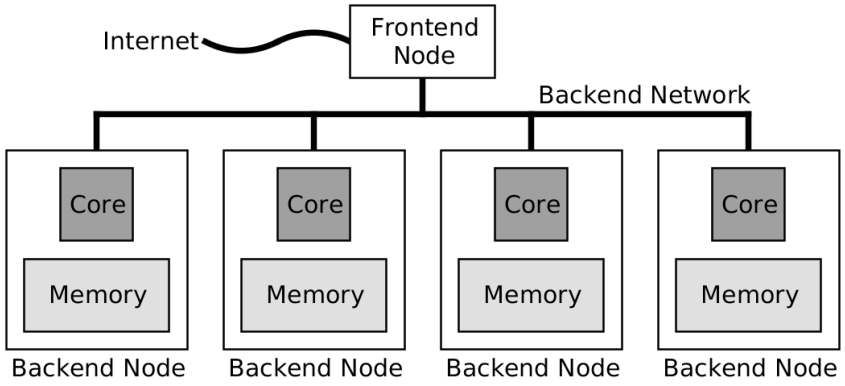

Figura 2.2: *Cluster* com arquitetura de memória distribuída. Retirado de [29]

Arquiteturas Híbridas Nestas arquiteturas são combinados os conceitos das arquiteturas de memória partilhada e memória distribuída. Vários processadores estão ligados à mesma memória, e cada conjunto de processadores com memória comum comunica com outros conjuntos de processadores através de uma interface de comunicação. Este tipo de arquitetura apresenta facilidade de comunicação entre processadores ligados à mesma memória, mas tal como na memória distribuída, a comunicação entre processadores através da rede é não uniforme. A figura 2.1.2 mostra um exemplo de uma arquitetura híbrida.

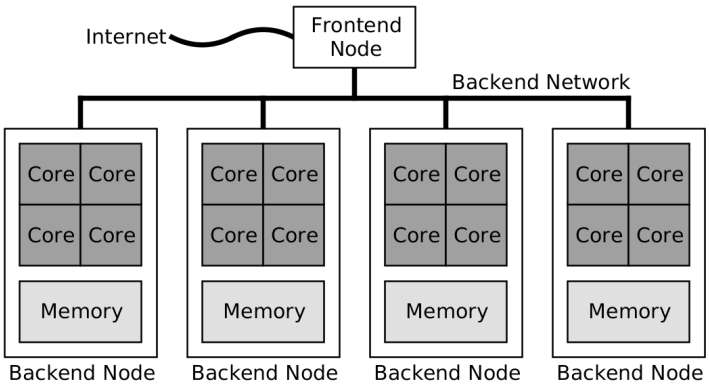

Figura 2.3: Arquitetura híbrida. Retirado de [29]

Tal como na memória distribuída, é fácil escalar este tipo de arquitetura pois cada conjunto de processadores é independente dos restantes. A programação é mais complexa pois combina problemas presentes nas arquiteturas anteriores, tais como a necessidade de garantir sincronia e consistência dos dados presentes em memória no mesmo conjunto de processadores e comunicação explícita entre conjuntos de processadores diferentes.

Arquiteturas Heterogéneas As arquiteturas heterogéneas são arquiteturas onde é usado mais do que um tipo de processador. Neste tipo de arquiteturas são incorporados processadores especializados de maneira a que estes tratem de tarefas específicas, tais como o GPU, ou *Graphics Processing Unit*, que foram concebidos para manipular e criar imagens.

Normalmente os processadores especializados são dispositivos separados do CPU, tendo uma memória privada. No entanto, já existem circuitos que integram CPU com GPU [31], partilhando os dois a mesma memória, como por exemplo o Intel Sandy Bridge [46]. A figura 2.1.2 mostra um exemplo de uma arquitetura híbrida.

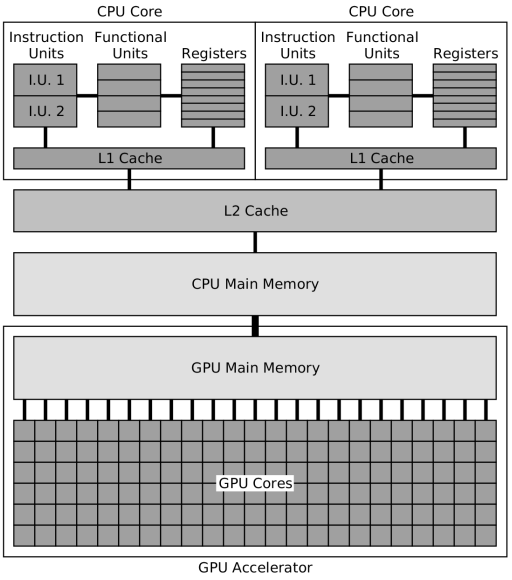

Figura 2.4: Arquitetura heterogénea. Retirado de [29]

Um GPU é composto por vários conjuntos de cores de processamento, sendo o número total de cores muito superior ao encontrado num CPU. Devido a esse número elevado de cores é possível obter maior paralelismo em GPU para outro tipo de computação, para além do processamento de imagens. Isto levou ao GPGPU, ou *general-purpose computation on graphics processing units*, onde é utilizado o GPU para a realização de computações que anteriormente seriam feitas em CPU. Um exemplo são as operações sobre matrizes, onde cada core de processamento pode ficar responsável por uma posição da matriz, como por exemplo cada core fazer uma soma da matriz resultante.

De maneira a ser possível implementar programas que fazem uso do GPU para computações genéricas, foram criadas várias *frameworks* tais como o CUDA [39], e *standards* como o OpenCL [38].

Quando comparado com a execução em CPU, a programação em GPUs oferece uma melhoria de desempenho na execução de certos programas paralelos, especialmente aqueles onde seja necessário processar uma grande quantidade de dados e onde exista pouca dependência entre o processamento aplicado a cada componente dos dados. No entanto apresenta alguns problemas, como a necessidade de transferir os dados a processar do CPU para o GPU, e o contrário com os resultados do processamento.

2.1.3 Modelos de programação

Os modelos de programação usados na conceção e desenvolvimento de aplicações paralelas podem se classificados de acordo com vários critérios. Esses modelos podem estar disponibilizados ao programador através de linguagens de programação que já apresentam suporte direto para programas paralelos, suportando entidades como *threads* ou envio de mensagens. Algumas dessas linguagens são o Chapel [11] e o Fortress [4]. Outra forma de suporte da programação paralela é através da adição a linguagens de programação tradicionais de anotações e bibliotecas. É nesta última abordagem que esta tese se concentra.

Paralelismo de dados e paralelismo de tarefas Uma possível classificação dos modelos de programação para arquiteturas paralelas divide-os em orientados para o paralelismo de dados e paralelismo de operações ou tarefas.

No modelo de paralelismo de dados, cada elemento de processamento (core) processa uma parte dos dados do problema, sendo que todos os núcleos processam da mesma maneira uma parte dos dados e são independentes das execuções dos outros núcleos. Um exemplo é o processamento de imagem, onde a mesma operação, por exemplo o cálculo do negativo da imagem, é aplicada a cada *pixel* podendo a imagem ser dividida em partes para processamento em cada core. A figura 2.1.3 apresenta um exemplo de paralelismo de dados, onde cada processador *Pn* processa uma parte do problema.

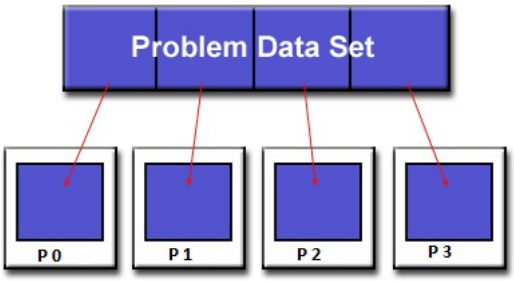

Figura 2.5: Paralelismo de dados, adaptado de [7]

No paralelismo de operações são executadas operações diferentes sobre os dados em cada core, podendo ser conjuntos diferentes de dados ou não. As várias operações podem estar relacionadas podendo uma usar como argumento o resultado de outra, como por exemplo num *pipeline*. Um exemplo é na execução de *queries* SQL onde diferentes processadores executam partes diferentes da mesma *query*. A figura 2.7 apresenta um exemplo de operações onde diferentes processadores, com núcleos diferentes, executam funções diferentes dos restantes.

Ambientes de programação para o paradigma de memória partilhada Existem vários modelos de programação que apresentam uso de memória partilhada de maneira a

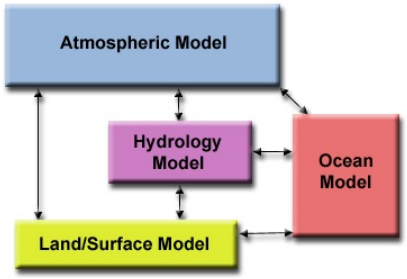

Figura 2.6: Exemplo de paralelismo de operações, retirado de [7]

fazerem comunicação entre processos, entre eles estão o POSIX threads (Pthreads) [10] e o OpenMP [14].

A programação usando Pthreads é feita usando uma API *Application Programming Interface* que inclui funções para a gestão de processos leves ou *threads* e para a sincronização entre eles. O modelo Pthreads é um modelo que entrega ao programador a responsabilidade de garantir que as *threads* em execução acedem à memória de maneira correta. De maneira a garantir que um valor é lido, ou atualizado, por uma *thread* sem erros devido à atividade de outras *threads*, estão disponíveis mecanismos de exclusão mútua. Este mecanismo permite que apenas uma *thread* aceda a uma zona crítica do código onde são acedidas variáveis partilhadas pelas várias *threads*. No entanto isto traz complexidade na criação de programas paralelos, pois obriga o programador a garantir que não existem bloqueios entre duas *threads* que necessitam de aceder às variáveis em exclusão mútua uma da outra, sendo esta situação conhecida como *deadlock*.

O modelo OpenMP consiste numa API e num conjunto de diretivas ou *pragmas*, que permite criar programas paralelos em sistemas com memória partilhada usando *threads*. Neste modelo a gestão de *threads* é implícita, sendo usadas as diretivas para especificar que partes do programa devem ser paralelizadas e como devem ser paralelizadas. Isto cria uma abstração de mais alto nível, tornando assim a criação de programas paralelos mais fácil para o programador pois este não precisa de especificar o comportamento de cada *thread*. Desta maneira apenas necessita de indicar uma parte do código como paralelo e compete ao compilador e ao sistema de suporte à execução definir que *threads* executam esse código.

Ambientes de programação para o paradigma de memória distribuída Para sistemas de memória distribuída existe o modelo MPI, *message passing interface*, em que o programador usa as funções de uma biblioteca para troca de mensagens entre processos. Estas operações permitem especificar para qual processo se pretende enviar a mensagem, e quais os dados que são enviados. Existem ainda operações que permitem enviar ou receber mensagens de vários processos. Visto que para cada operação de envio de mensagem é necessário haver uma operação que recebe a mensagem noutro processo, é possível garantir sincronia entre os processos. Existem diferentes implementações conformes com

o standard MPI tais como o OpenMPI [24] e o MPICH [23].

Adicionalmente, é possível combinar os modelos de memória distribuída com os modelos de memória partilhada, por exemplo usar MPI para fazer a comunicação entre várias máquinas e usar OpenMP para paralelizar a execução numa máquina.

Ambientes de programação para arquiteturas heterogéneas Para este tipo de arquiteturas é necessário existirem ambientes que permitam criar programas que executam em parte no CPU e em parte em GPU. O CUDA é uma *framework* criada pela NVIDIA e implementada nos GPUs que esta fabrica. O OpenCL é um *standard* criado para escrever programas que executam em diferentes tipos de processadores, não apenas GPUs. Comparativamente ao OpenCL, o CUDA apresenta como desvantagem o facto de apenas ser possível usar em GPUs fabricadas pela NVIDIA.

Na programação com CUDA é possível especificar quais as funções que executam em CPU e quais as que o são em GPU. Para as funções que executam em GPU é possível definir blocos, conjuntos de *threads* que podem cooperar entre si, sendo possível sincronizá-las, e que partilham um espaço de memória privado ao bloco. Para estas funções é possível definir o número de blocos que esta vai utilizar, e o número de threads presentes em cada bloco. De maneira a ser possível o GPU usar informação guardada na memória do CPU,ou vice-versa é necessário passar essa informação de um dispositivo para o outro, estando implementadas operações que fazem essa transferência. Um problema presente na programação com CUDA é a dificuldade em sincronizar *threads*, não sendo possível faze-lo entre *threads* de blocos diferentes.

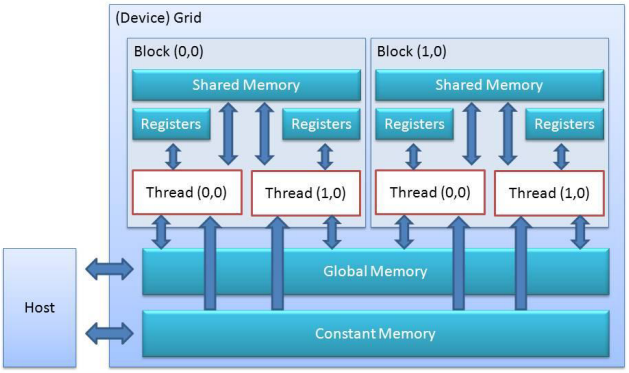

Figura 2.7: Modelo de memória do GPU, retirado de [30]

Algorithmic Skeletons Os *algorithmic skeletons* [13] são abstrações de alto nível de padrões de computação paralela e das comunicações e interações presentes em cada padrão. Alguns exemplos de padrões presentes em *skeletons* são o *pipeline*, onde diferentes fases do processamento são executados de forma paralela, e o padrão de *Master-Worker*, onde o processamento dos dados é dividido por vários *workers*. Existem várias *frameworks* que

implementam *algorithmic skeletons*, estando entre elas o FastFlow [1] e o eSkel [12]. Este tipo de modelo irá ser discutido em detalhe na secção 2.3.

2.1.4 Desenho de programas paralelos

Existem metodologias de desenho de programas paralelos que facilitam a criação dos mesmos, ou a transformação de programas sequenciais em programas paralelos. Uma dessas metodologias é a metodologia de Foster [22], que é composta por quatro passos de maneira a transformar a solução de um problema num programa paralelo.

- **Particionamento**: Nesta fase o problema é dividido em tarefas mais pequenas que possam ser executadas paralelamente. Esta divisão pode tanto ser feita ao nível dos dados, como da computação a ser feita.
- **Comunicação**: Neste passo são determinadas quais as comunicações necessárias entre as várias tarefas divididas anteriormente.
- **Aglomeração**: Aqui são agrupadas tarefas em tarefas maiores de maneira a melhorar o desempenho ou simplificar a programação.
- **Mapeamento**: O último passo consiste em atribuir tarefas a processos ou *threads*. Isto deve ser feito de maneira a minimizar a comunicação necessária, e garantir que cada processo ou *thread* recebe a mesma quantidade de tarefas.

2.1.5 Métricas de desempenho

O objetivo da criação de programas paralelos é melhorar o desempenho quando comparado com um programa sequencial. De maneira a avaliar o desempenho do nosso programa existem várias métricas sendo as mais importantes a aceleração ou *speedup* e a eficiência. O *speedup* permite comparar o tempo de execução de um programa executado por N CPUs e o tempo de execução do mesmo programa quando corrido num único CPU

De maneira a se calcular o *speedup* é necessário obter primeiro o tempo de execução do programa sequencial *Tsequencial* e do programa paralelo *Tparalelo*. Podemos considerar que o programa paralelo executa p vezes mais depressa que o programa sequencial, onde p é o número de núcleos usados. Se todo o programa pudesse ser paralelizado e não houvesse tempos não úteis (*overheads*) associados ao uso dos processadores extra, o tempo de execução com N processadores seria:

$$Tparalelo = \frac{Tsequencial}{p} \tag{2.1}$$

No caso anterior, para N CPUs o *speedup* seria N. De uma forma geral, o *speedup* é definido como:

$$S = \frac{Tsequencial}{Tparalelo} \tag{2.2}$$

Como num programa paralelo existe sempre uma fração que é sequencial irá haver sempre um limite de *speeudup* possível independentemente do número de núcleos. Esta observação é conhecida como a *lei de Amdahl* [5]. Considerando c como a fração do programa que é paralela, podemos obter o limite de *speedup* através da função:

$$S(p) = \frac{1}{(1-c) + \frac{c}{p}} \tag{2.3}$$

Se considerarmos que o número de processadores tende para o infinito temos:

$$S(p) = \frac{1}{1-c} \tag{2.4}$$

No entanto a *lei de Amdahl* não considera o tamanho do problema. Para vários programas se aumentarmos o tamanho do problema conseguimos reduzir a fração do programa que é sequencial. A *lei de Gustafson* [28] é outra métrica para definir o *speedup* possível de um programa paralelo, tendo em conta o tamanho do problema. Segundo a lei de Gustafson podemos considerar que o tempo de execução de um programa paralelo consiste numa parte sequencial s e numa parte paralela c executada por p processadores. O valor do tempo de execução paralelo é obtido através de:

$$T(p) = s + c \tag{2.5}$$

Sendo o tempo de execução sequencial igual a:

$$T(1) = s + cp \tag{2.6}$$

Daí podemos obter a seguinte função para o *speedup*:

$$S(p) = \frac{s + cp}{s + c} \tag{2.7}$$

Se definirmos $\alpha = \frac{s}{s+c}$ obtém-se:

$$S(p) = \alpha + p(1-\alpha) = p - \alpha(p-1) \tag{2.8}$$

Daí podemos ver que à medida que o tamanho do problema aumenta e com isso a fração sequencial diminui, o *speedup* do programa aumenta aproximando-se cada vez mais de p.

A *eficiência* de um programa paralelo indica como é que os processadores estão a ser utilizados para resolver o problema, comparativamente a quanto é que está a ser gasto em termos de sincronização e comunicação entre processos. A eficiência pode ser obtida através da fórmula:

$$E = \frac{S}{p} = \frac{Tsequencial}{p.Tparalelo} \tag{2.9}$$

Salvo casos especiais, a eficiência máxima possível de atingir é 1 o que indica que os recursos do computador estão a ser usados de forma ótima.

2.2 Processamento de imagem

Processamento de imagem é o uso de operações, como a criação de histogramas ou aplicação de filtros, de maneira a retirar e manipular informação contida numa imagem digital. O resultado deste processamento tanto pode ser a imagem original alterada, ou um conjunto de valores que caracterizam a imagem original, como por exemplo a intensidade de cada cor em cada *pixel*. Os valores obtidos podem ser usados para detetar e corrigir vários problemas presentes na imagem tais como problemas de contraste ou de brilho, ou detetar padrões e objetos como por exemplo caras numa fotografia.

Nas sub-secções seguintes descreve-se a relevância do processamento de imagem com aplicabilidade a diversas áreas científicas, e enumeram-se as suas características principais em termos da definição de uma imagem digital e operações que sobre ela podem ser aplicadas. Prossegue-se referindo as imagens a três dimensões, quais as operações que é possível realizar sobre elas, bem como o tipo de paralelismo possível. Termina-se com a apresentação em detalhe de um exemplo de processamento de imagem, bem como técnicas possíveis para a sua paralelização.

2.2.1 Âmbito

A importância do processamento de imagem tem vindo a crescer devido à sua utilidade e potenciais aplicações em áreas muito diversas. Com o processamento de imagem é possível detetar objetos, melhorar e restaurar imagens como fotografias, ou reconhecer padrões presentes na imagem. Em áreas como a medicina, o reconhecimento de padrões pode ser usado para ajudar a detetar tumores em pacientes [20]. O processamento de imagem também tem aplicações em áreas como a meteorologia e astronomia, por exemplo para retirar erros de imagens obtidas por telescópio, entre outros. Adicionalmente, a automatização do processamento de imagem permite avaliar um largo número de imagens e obter uma resposta muito mais rápida, quando comparado com um processamento manual das imagens, por exemplo na deteção de objetos em aeroportos, ou no reconhecimento de anomalias em imagens médicas.

As técnicas usadas para o processamento de imagem também podem ser aplicadas ao processamento de vídeos, visto que se tratam de várias imagens em sequência. Uma aplicação possível é a deteção da movimentação de um objeto ao longo de um vídeo.

Os problemas do processamento de imagem passam pela dificuldade em compreender, e aplicar, os cálculos, fórmulas e algoritmos usados, sendo necessário ter conhecimento na área de maneira a que os mesmos sejam bem aplicados. Outro problema é o tempo que o processamento pode demorar, visto que é necessário percorrer todos os *pixeis* da imagem. Caso a imagem apresente um grande número de *pixeis* o seu processamento pode ser bastante demorado.

2.2.2 Características

Numa imagem digital o que é processado são os valores de intensidade das várias cores de cada *pixel*. No caso de a imagem ser a preto e branco o valor de cada *pixel* refere-se à escala de cinzento; no caso da imagem se apresentar a cores, cada *pixel* apresenta três cores que representam os valores de intensidade de vermelho, verde e azul. Quando a imagem é processada estes são os valores usados, podendo ser contabilizados de maneira a se obter informações sobre a imagem, por exemplo o contraste de uma imagem, modificados de maneira a se criar uma imagem nova com base em valores antigos, ou comparados entre si de maneira a ser possível detetar diferenças de intensidade entre *pixeis* e com isso detetar objetos na imagem.

2.2.3 Operações

O processamento de imagem permite-nos realizar vários tipos de operações, entre eles a criação de histogramas onde são recolhidas estatísticas sobre a imagem, operações ponto a ponto onde cada ponto é modificado com base numa função, e filtros onde cada ponto é modificado com base nos pontos vizinhos.

Na criação de um histograma são recolhidas estatísticas com os valores de cada *pixel* o que nos permite detetar e corrigir erros de brilho e contraste. Numa imagem a preto e branco são recolhidos os valores de intensidade de cinzento, enquanto que numa imagem a cores podem ser criados histogramas com base no valor de intensidade de cada cor ou com base no brilho de cada *pixel*. A figura 2.8 apresenta um exemplo de histograma.

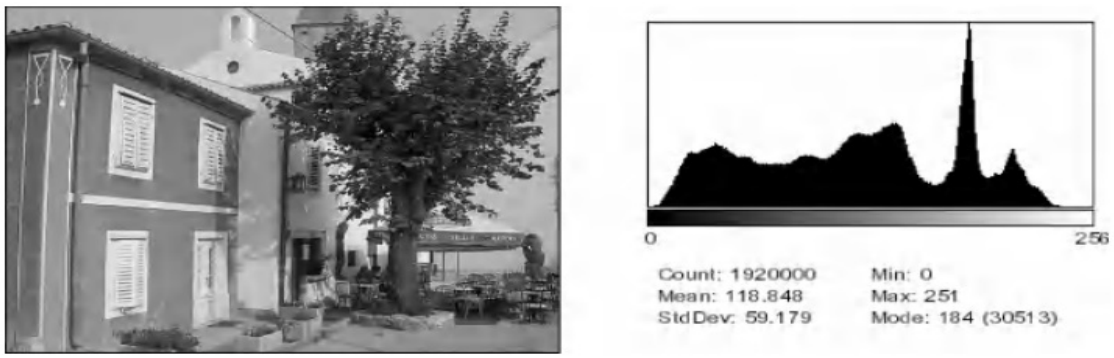

Figura 2.8: Imagem com histograma correspondente, retirado de [9]

As operações ponto a ponto aplicam uma função a cada *pixel* sem alterar o tamanho ou estrutura da imagem. O novo valor de cada *pixel* depende apenas do seu valor anterior sendo independente do valor de qualquer outro *pixel*. Algumas das operações possíveis são alterar o brilho e contraste da imagem ou inverter as cores da imagem. Para modificar o contraste cada valor é multiplicado pelo fator de contraste que se pretende, por exemplo se quisermos aumentar o contraste por 50% multiplicamos cada valor por 1.5. A fig 2.9 apresenta um exemplo da operação de alterar o contraste da imagem.

Figura 2.9: À esquerda a imagem original, à direita após ser modificado o contraste, retirado de [9]

As operações de filtro, tal como as funções ponto a ponto, aplicam uma função a cada *pixel* sem alterar o tamanho ou estrutura da imagem. No entanto o novo valor de cada *pixel* é calculado com base nos valores dos *pixeis* vizinhos na imagem original. Um exemplo da aplicação de um filtro é o efeito de névoa, onde os valores dos *pixeis* na nova imagem é a média dos valores de *n pixeis* vizinhos na imagem original. A figura 2.10 apresenta um exemplo do efeito névoa.

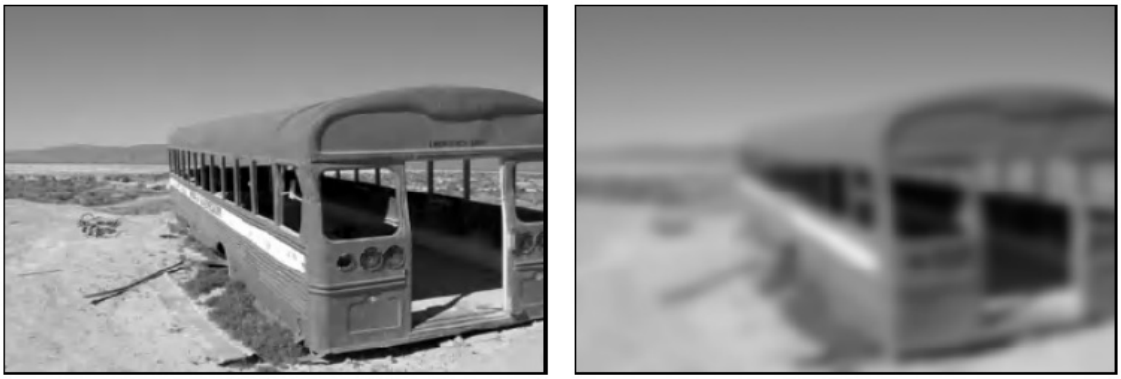

Figura 2.10: À esquerda imagem original, à direita após ter sido aplicado o efeito de névoa, retirado de [9]

Outra operação é a deteção de linhas na imagem. Estas são detetadas quando se encontra uma diferença na intensidade entre pixéis vizinhos. Com esta operação é possível detetar objetos com base nas suas linhas. A figura 2.11 apresenta um exemplo da operação de deteção de linhas.

Outra operação possível é a deteção de padrões ou objetos, como pessoas na imagem. Este tipo de operações tem particularmente em áreas como a de sistemas de transporte inteligente, de maneira a se detetar e reconhecer matrículas automaticamente, ou no processamento de imagens médicas, onde interessa saber se se encontra alguma anomalia na imagem fornecida.

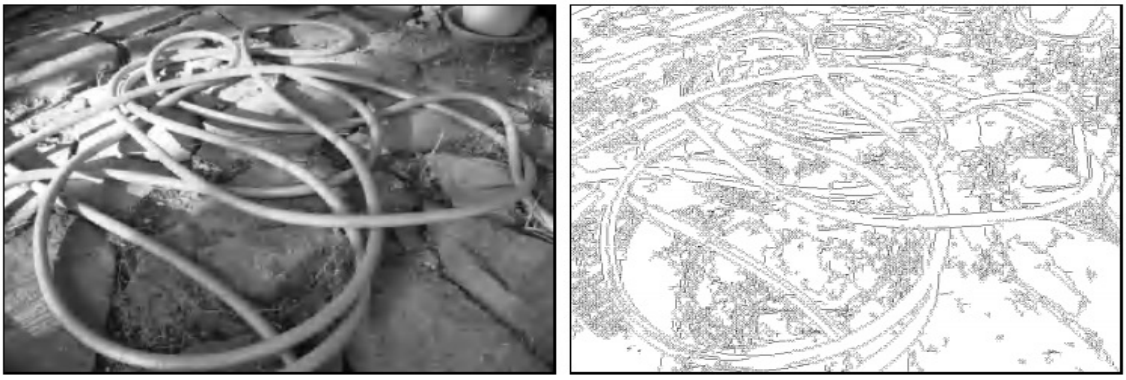

Figura 2.11: Exemplo de deteção de linhas, retirado de [9]

2.2.4 Processamento paralelo de imagens

Como o processamento de cada *pixel* da imagem pode ser independente dos restantes, o processamento da imagem pode ser paralelizado. Um tipo de paralelização possível será dividir a imagem em várias partes sendo cada parte processada paralelamente em diferentes núcleos. Esta divisão pode ser de vários níveis de grandeza, podendo ser desde processamento paralelo por um conjunto de linhas até processamento paralelo de cada *pixel*. Neste caso é preciso uma grande quantidade de processadores, normalmente encontrado dentro de GPUs.

Tendo em conta o exemplo dado na secção 2.2.6, o processamento em cada fase pode ser paralelizado. Na primeira fase apenas são recolhidas as cores de cada *pixel* presente na imagem, sendo essa recolha independente dos restantes *pixeis*. Na segunda fase as alterações que são feitas dependem apenas dos valores que são obtidos do histograma e não dos *pixeis* vizinhos sendo assim possível paralelizar o processamento de cada *pixel* nesta fase. Na última fase o novo valor de cada *pixel* depende dos valores dos seus vizinhos na imagem original. Assim em cada ronda podemos processar cada *pixel* paralelamente, sendo necessária sincronia apenas no fim de cada ronda de maneira a garantir que é sempre a mesma imagem que está a ser processada por todos os processos ao mesmo tempo.

Outro tipo de paralelização possível é através do uso de um *pipeline*, onde cada processo executa uma operação diferente na imagem, sendo estas execuções independentes entre si. Usando o exemplo da secção 2.2.6, cada uma das fases seria feita por processos diferentes. No entanto como cada fase é dependente dos resultados da fase anterior, não seria possível processar todas as fases ao mesmo tempo para a mesma imagem. No entanto seria possível processar diferentes imagens ao mesmo tempo, sendo processadas diferentes fases para cada imagem.

2.2.5 Imagens 3D

O processamento de imagem também pode ser aplicado a imagens a três dimensões, conhecidas como imagens 3D. Estas imagens guardam as intensidades das cores de cada ponto, usando a profundidade para além da altura e largura. Isto é muitas vezes conseguido ao alinhar e empilhar várias imagens a duas dimensões, que são paralelas numa dada direção e apresentam espaçamento entre si. Estas imagens são muito usadas na área médica, sendo possível obter imagens deste tipo através de tomografias axiais computorizadas (TAC) ou através de ressonâncias magnéticas.

As operações aplicadas para o processamento de imagens 3D são iguais às operações aplicadas para o processamento de imagens a duas dimensões. No entanto para operações onde são usados *pixeis* vizinhos, é necessário considerar a profundidade para os cálculos efetuados.

As imagens a três dimensões podem ser paralelizadas como uma imagem a duas dimensões, sendo possível paralelizar a imagem por plano que contém uma imagem a duas dimensões, sendo esta divisão possível apenas para imagens a três dimensões.

2.2.6 Exemplo; processamento de imagens tomográficas de compósitos

Um problema onde é usado o processamento de imagem é na área de Ciências dos Materiais, nomeadamente na análise de processos de fabrico de materiais compósitos. Nestes materiais, difunde-se num *material base* um conjunto de *reforços* de forma a conseguir um conjunto de propriedades que não existia no material base. O processamento de imagens tomográficas dos materiais produzidos permite avaliar a qualidade do processo de fabrico usado, através da caracterização da população de reforços, nomeadamente a sua localização e orientação na amostra.

Como a captura das imagens introduz alguns erros, estas não se apresentam apenas a preto (reforços) e branco (material base) mas também apresentam tons de cinzento. O objetivo do processamento da imagem é transformá-la de maneira a que apenas apresente a cor preta e branca, de maneira a ser mais fácil caracterizar os objetos aí presentes. O processamento de imagens passa por três fases, a criação de um histograma, uma operação ponto a ponto com base no histograma, e uma operação onde as modificações são feitas com base em *pixeis* vizinhos.

Na primeira fase é construído um histograma para todas as cores possíveis que cada *pixel* pode ter. O histograma é preenchido com o número de *pixeis* presentes na imagem com cada cor. Depois de criado o histograma este é analisado e são obtidos dois valores, um que representa um conjunto de *pixeis* com cores próximas do preto e outro valor que representa um conjunto de *pixeis* com cores mais próximas do branco, cada cor representando materiais diferentes que se pretende distinguir.

Na segunda fase cada pixel é alterado com base nos valores obtidos na primeira fase. Caso o *pixel* tenha uma cor menor que o valor que representa os *pixeis* pretos, este muda a sua cor para preto. Caso o *pixel* apresente uma cor maior do que o valor que representa os

pixeis brancos, este muda a sua cor para branco. No caso de nenhuma destas condições se verificar este fica com a cor cinzenta.

Na terceira fase partindo da imagem obtida da fase anterior, a cor de cada *pixel* é alterada com base nas cores dos *pixeis* vizinhos. Para cada *pixel* cinzento, se os seus vizinhos apresentarem uma maioria de cor preta este muda a sua cor para preto, se os seus vizinhos apresentarem uma maioria de cor branca este muda a sua cor para branco, caso nenhuma destas condições se verifique este fica com cor cinzenta. Isto é repetido até não existirem mais *pixeis* de cor cinzenta ou até não ser possível alterar os *pixeis* de cor cinzenta. No fim, caso ainda existam *pixeis* de cor cinzenta, o processo é repetido com a diferença de que se não for possível obter uma maioria relativa de cor branca ou preta a cor do *pixel* é escolhida aleatoriamente.

2.3 Programação paralela estruturada

A programação paralela estruturada consiste na disponibilização e utilização de soluções standard ou padrões que capturam esquemas de peritos na área da computação paralela. Permite assim fornecer ao programador modelos e abstrações que capturam esses padrões/ esquemas recorrentes, permitindo definir uma aplicação paralela a um nível mais elevado de abstração, sem preocupação com os detalhes de implementação de uma arquitetura particular.

Esta secção descreve as características principais da programação paralela estruturada com base em padrões paralelos e *algorithmic skeletons*, bem como os benefícios que daí resultam, mas também as limitações ainda existentes nesta estratégia de programação. São ainda descritos alguns dos *algorithmic skeletons* mais comuns e relevantes para este trabalho, bem como exemplos de *frameworks* de suporte à programação com base em *skeletons*. São ainda realçados duas *frameworks* em particular, FastFlow e SkePU,como candidatos ao suporte a este trabalho dado que suportam a geração de código para GPGPUs.

2.3.1 Âmbito

Os padrões de desenho paralelos (*parallel patterns*) [36] apresentam uma descrição de possíveis soluções para problemas recorrentes em computação paralela, e foram enunciados num formato textual à semelhança dos "*design patterns*"[25]. Os padrões paralelos inicialmente não foram acompanhados por implementações que possam ser usadas diretamente na criação de um programa paralelo, mas descreviam, de um modo geral, um problema e as soluções que podem ser aplicadas pelo programador para resolver este problema.

Os *algorithmic skeletons* [13] são abstrações ao nível das linguagens de programação e representam padrões de computação paralela. Cada *skeleton* captura as interações entre os elementos que constituem um padrão particular, escondendo os detalhes da sua implementação numa arquitetura específica, por exemplo, de que modo deve ser realizada a

sincronização e consistência de dados entre as atividades paralelas que constituem um padrão.

De um modo geral, para paralelizar o seu código, um programador deve saber identificar que partes do seu código pode ser paralelizado e escolher um ou mais *skeletons* que representam os modelos de paralelização mais adequados e, instanciar os seus parâmetros. Os parâmetros de entrada de um *algorithmic skeleton* consistem nos dados a serem processados, bem como funções e condições que seja necessário aplicar; os parâmetros de saída representam os dados processados.

Os *algorithmic skeletons* têm assim como objetivo tornar mais fácil a criação de programas paralelos dado que um programador não tem de se preocupar com os detalhes de implementação de uma solução de paralelização no contexto de uma arquitetura particular. Para mais, geralmente os *algorithmic skeletons* têm funções de custo associadas (e.g. em termos do tempo esperado para uma dada configuração de dados e arquitetura alvo) que permitem que um programador possa ter uma expectativa em termos dos tempos associados à execução do seu programa.

As *frameworks* de *algorithmic skeletons*, por seu lado, são ferramentas, ou bibliotecas, que implementam *algorithmic skeletons* que podem ser parametrizados pelo programador de maneira a ser obtido um programa paralelo concreto. Diferentes *frameworks* podem suportar diversas características, como seja a geração de código para arquiteturas distintas ou a utilização de standards de paralelização, e que promovem a sua utilização quer em sistemas computacionais tradicionais, quer em arquiteturas emergentes.

Dado que os *algorithmic skeletons* abstraem os detalhes do paralelismo numa arquitetura, tornando transparente as diferenças entre arquiteturas diversas, permitem agilizar a programação paralela para um programador. Tal é bastante relevante quer para programadores que pretendam reduzir o esforço de portar o seu código para outras arquiteturas, quer para programadores com pouca experiência em computação paralela. Este é o caso de programadores em áreas que não a informática, cada vez em maior número, que beneficiam da existência de estratégias que possam ser mais próximas do seu domínio de aplicação, sem preocupação com os detalhes de execução. Estes programadores inexperientes beneficiam ainda da redução dos erros de programação no seu código, visto que os *skeletons* representam soluções implementadas e testadas por peritos na área. Pela mesma razão, os programadores inexperientes podem obter melhorias de desempenho das suas aplicações dado que as soluções suportadas, embora com desempenho inferior a soluções desenvolvidas à medida, representam otimizações que um não perito na área teria dificuldade em saber desenvolver.

Uma das maiores limitações do uso de *algorithmic skeletons*, é precisamente o facto de a geração automática de código não conseguir produzir, para todos os casos, soluções tão eficientes como as que um perito conseguiria numa solução desenvolvida à medida. Outra desvantagem é que, caso um problema não se adapte de maneira rigorosa aos padrões de paralelização suportados pelos *skeletons* disponibilizados, a sua paralelização não é possível nesse contexto. Quando muito, os *skeletons* existentes podem ser adaptados ou

estendidos, reduzindo assim a facilidade que a abstração traz. Para além disso, não existe nenhum standard que especifique os vários *skeletons* e permita a sua portabilidade entre diferentes implementações de um modo simples.

2.3.2 Exemplos de *Skeletons*

De seguida são apresentados alguns dos padrões de paralelismo mais comuns capturados como *algorithmic skeletons*.

No padrão de *pipeline*, figura 2.12, o processamento dos dados passa por várias fases onde é aplicada uma função aos dados, sendo que saída de cada fase é a entrada da próxima fase. De maneira a usar este *skeleton*, são definidas quais as funções executar em cada etapa, sendo garantida que para um dado conjunto de dados, a etapa seguinte só começa a executar depois de todos os dados de entrada estarem processados.

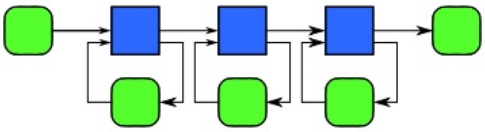

Figura 2.12: Padrão de pipeline. Retirado de [37]

No padrão de *Farm*, figura 2.13, também conhecido como *Master-Worker*, os dados são divididos por vários *workers*, ou *slaves*, que processam os dados da mesma maneira.

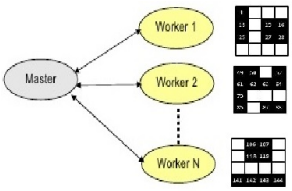

Figura 2.13: Padrão Master-Slave. Retirado de [6]

No padrão de *Map*, figura 2.14, o conjunto de dados recebidos na entrada é dividido em vários subconjuntos, sendo aplicado o mesmo processamento a cada subconjunto. Neste *skeleton* é necessário passar qual a função de mapeamento a ser usada.

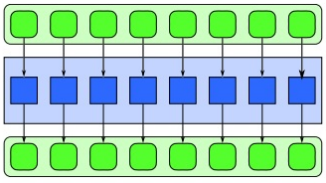

Figura 2.14: Padrão de Map. Retirado de slides de suporte a [37]

O padrão de *Reduce*, figura 2.15, consiste em agregar um conjunto de dados de maneira a se obter um resultado final. Para este padrão é necessário fornecer a função de agregação dos dados. Este padrão costuma ser usado em conjunto com o padrão *Map*, sendo usado os resultados do *Map* como input para a função de agregação do *Reduce*.

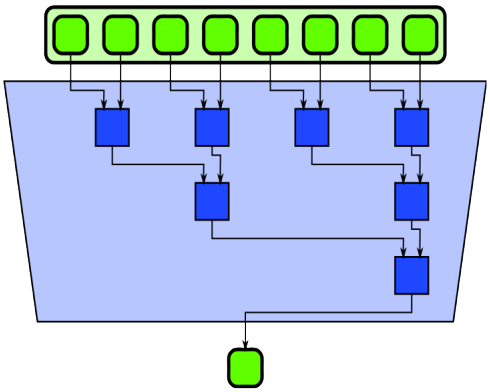

Figura 2.15: Padrão de Reduce. Retirado de slides de suporte a [37]

O padrão de *Stencil*, figura 2.16, é uma generalização do padrão *Map* onde o mapeamento não é feito apenas com base num elemento dos dados, mas onde são também usados elementos vizinhos na função de mapeamento.

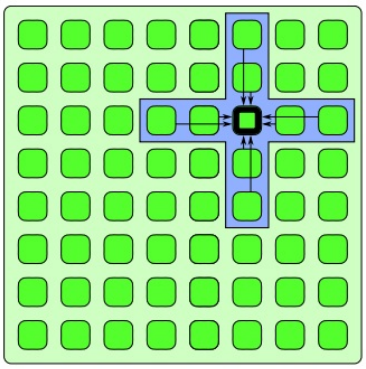

Figura 2.16: Padrão de Stencil. Retirado de slides de suporte a [37]

O padrão de *Divide and Conquer*, ou D&C, consiste em duas fases. Na primeira fase o problema é divido em problemas mais pequenos até se obter um caso base, suficientemente simples para ser resolvido. Na segunda fase o problema é resolvido e agregado com as outras divisões do problema já resolvidas. Para parametrizar este padrão é necessário indicar qual é a condição de paragem da subdivisão, como é que se agregam os resultados e qual a função a executar quando se chega ao caso base.

2.3.3 Tipos de ferramentas

De acordo com [26] é possível classificar cada *framework* de *algorithmic skeletons* com base nos paradigmas de programação suportados. Cada *framework* pode pertencer a pelo menos

uma das seguintes categorias:

Coordenação: As *frameworks* pertencentes a este tipo de classificação usam uma linguagem de alto nível para descrever o comportamento do algoritmo, sendo a interação com a estrutura tratada por outra linguagem. Isto permite ao programador gerar o código paralelo através do uso do *framework* e gerir a comunicação através de outra linguagem e de bibliotecas como o MPI. Este tipo de *frameworks* oferecem uma clara distinção entre a coordenação e a comunicação do programa paralelo. A principal desvantagem é que obriga o programador a aprender uma nova linguagem de maneira a poder usar a *framework*. Alguns exemplos de *frameworks* com esta classificação são o SCL [17], o Skil [8] e o SAC [27].

Funcional: Este tipo de *frameworks* apresentam-se como extensões de linguagens de maneira a permitirem o uso de *skeletons*. Estas *frameworks* geram programas paralelos ao traduzirem o seu código para outras linguagens como C, ou usando compiladores específicos. Este tipo de *framework* apresenta a vantagem de que não obriga o programador a aprender uma nova linguagem, visto que o *skeleton* é uma extensão de uma linguagem que este já conhece. Alguns exemplos de *frameworks* são o Eden [33] e o SkiPPER [42].

Orientada a objetos: Estas *frameworks* apresentam-se como bibliotecas que estendem a linguagem e incluem os vários *skeletons* como classes. Como estas *frameworks* são bibliotecas, não obrigam o programador a ter que aprender uma linguagem nova para as poder utilizar, além disso não obrigam a que o código seja traduzido para outra linguagem ou seja necessário um compilador específico. Este tipo de *framework* apresenta alguma popularidade devido à popularidade das linguagens orientadas pelos objetos. Alguns exemplos são o FastFlow [1] e o Skandium [32], sendo o primeiro para C++ e o segundo para a linguagem Java.

Imperativa: Neste tipo de classificação as *frameworks* são APIs na linguagem em que estão incorporadas. Como estas *frameworks* são APIs não necessitam de usar compiladores específicos. Algumas *frameworks* deste tipo são o eSkel [12] e o SKElib [15].

2.3.4 Caracterização de *frameworks* de *skeletons*

É possível definir várias características que diferenciam as várias *frameworks*, e que são usadas na comparação entre ferramentas que se segue.

A *linguagem de programação* define que linguagem é utilizada pelo programador para criar o seu programa com base em *skeletons*. Os *skeletons* podem ser usados como extensões de uma linguagem, bibliotecas ou como uma nova linguagem específica da *framework*. A linguagem utilizada para programar não necessita de ser a mesma em que está escrito o código fonte do programa a executar. Neste caso é necessária uma ferramenta para traduzir o código do programador para a *linguagem de execução*

A *distribuição* indica que bibliotecas são usadas como suporte da gestão dos processos e das operações de comunicação e sincronização. Estas bibliotecas indicam como é feita a paralelização e que tipo de arquiteturas hardware são suportadas. Estas bibliotecas não

Tabela 2.1: Comparação entre *frameworks* de *algorithmic skeletons* adaptada de [26]

Framework	**Linguagem de Programação**	**Linguagem de Execução**	**Bibliotecas de distribuição**	**Suporte para skeletons contidos**	**Conjunto de skeletons suportados**
SCL	Linguagem própria	Fortran/C	MPI	Sim	Map,Farm
Skil	C	C	—	Não	Map
SAC	Linguagem própria	C	Threads	Não	
Eden	Extensão de Haskell	C	PVM/MPI	Sim	Map, D& C, Pipeline
SkiPPER	CAML	C	SynDex	Limitado	
FastFlow	C++	C++/ CUDA/ OpenCL	Pthreads/ CUDA/ OpenCL	Sim	Pipeline, Farm, Map, Reduce, Stencil
Skandium	Java	Java	Threads	Sim	Pipeline, Farm, Map, D& C
eSkel	C	C	MPI	Sim	Pipeline, Farm
SKElib	C	C	MPI	Não	Pipeline, Farm
SkePU	C++	C++/ CUDA/ OpenCL	CUDA/ OpenCL/ OpenMP		Map, Reduce, Farm

são disponibilizadas ao programador pelo *skeleton*, impedindo assim que haja alterações à maneira como a comunicação e paralelização do programa é feita. A biblioteca mais utilizada por vários *frameworks* é o MPI, no entanto existem vários *frameworks* que suportam o uso de diferentes sistema de apoio à execução.

Alguns *frameworks* apresentam a possibilidade de incluir *skeletons* dentro de outros, i.e. *skeleton nesting*. Isto permite a criação de padrões mais complexos usando padrões simples. Para isso o programador deve fornecer um *skeleton* como uma das funções a ser executada por outro *skeleton*.

Cada *framework* suporta diferentes conjuntos de *skeletons*, no entanto os conjuntos apresentados na tabela 2.1 não apresentam todos os *skeletons* das *frameworks* em comparação,mas apenas aqueles que foram descritos na secção 2.3.2.

No capítulo 3 descrevem-se duas *frameworks* de *algorithmic skeletons* que, por gerarem código para GPUs, são candidatas a ferramentas de suporte ao trabalho a desenvolver nesta tese.

2.4 Propriedades adaptativas

Um sistema com propriedades adaptativas é um sistema que apresenta a capacidade de avaliar e alterar o seu próprio comportamento, quando a avaliação mostra que este não é capaz de atingir os seus objetivos, ou quando é possível obter melhor funcionalidade ou desempenho do mesmo. Estas alterações podem ter origem no sistema em si, por exemplo falhas de componentes do sistema, ou podem ter origem no ambiente, por exemplo informação recolhida de sensores ou um aumento de pedidos pelos utilizadores. De maneira a que estas alterações sejam possíveis, um sistema com propriedades adaptativas deve monitorizar-se a si próprio e ao seu ambiente, detetar alterações significativas e decidir como reagir e executar as alterações escolhidas.

Esta alteração do comportamento é realizada através de um ciclo de realimentação (*feedback loop*). Este ciclo é constituído por quatro fases: monitorização (M), análise(A), planeamento(P) e execução (E), sendo este ciclo designado por ciclo MAPE.

- O processo de *monitorização* é responsável por recolher e correlacionar dados das várias fontes, como sensores, e converter os dados para padrões de comportamento. Este processo pode ser feito através de correlação de eventos ou através da deteção de valores acima de um determinado limite, sendo possível usar outro métodos.
- O processo de *análise* é responsável por analisar a informação recolhida pelo processo de monitorização, de maneira a detetar quando é necessário fazer alguma alteração.
- O processo de *planeamento* é responsável por determinar o que é necessário alterar e como é feita essa alteração de maneira a obter o melhor resultado.
- Por fim, o processo de *execução* é responsável por aplicar as alterações ao sistema, decididas pelo processo anterior.

A figura 2.17 apresenta um ciclo MAPE.

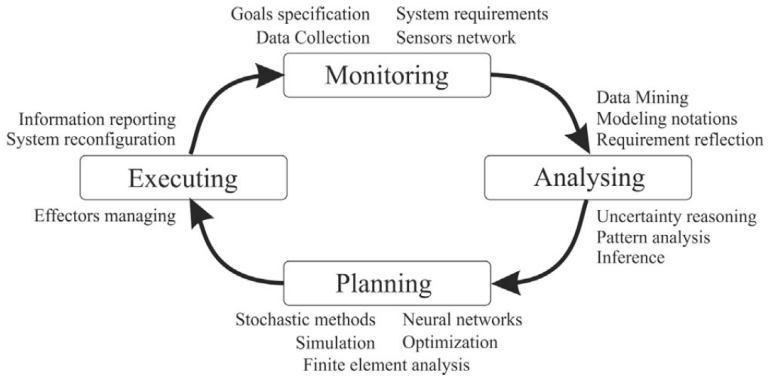

Figura 2.17: Fases de um ciclo MAPE. Retirado de [34].

Com sistemas de maior dimensão e maior complexidade, um único ciclo MAPE pode não ser suficiente, podendo ser usados vários ciclos no mesmo sistema.

Com base no ciclo MAPE são escolhidas as adaptações necessárias, se tal for o caso, de maneira a que o sistema atinja os seus objetivos. Alguns exemplos de adaptações possíveis ao sistema são:

- Ajustar parâmetros de maneira a se atingir os objetivos estipulados;
- Fazer uma nova distribuição de carga do sistema de maneira a se obter uma divisão justa pelos vários componentes do sistema, de maneira a obter melhor o número de tarefas completadas por unidade de tempo (*throughput*) ou diminuir o tempo de resposta;
- Alterar o método ou algoritmo em utilização, usando um algoritmo que seja mais eficiente;
- Alterar o alvo (*backend*) onde o algoritmo é executado, por exemplo, passando de CPU para GPU, ou vice-versa;
- Alterar um componente por outro que apresente melhor desempenho.

Existem vários padrões de desenho para a implementação de sistemas com propriedades adaptativas e reconfiguração dinâmica [35] [45]. Em [45] são apresentados padrões em que são usados vários ciclos MAPE em sistemas distribuídos, podendo cada subsistema executar uma ou mais partes do ciclo. Um exemplo de padrão é *Information Sharing*, onde cada subsistema executa o seu ciclo MAPE, existindo partilha de informação entre os componentes de monitorização de cada subsistema, mas não existindo comunicação entre os restantes componentes do ciclo. Outro exemplo é o padrão *Master/Worker*, onde existe um sistema central que comunica com vários subsistemas. Estes subsistemas são responsáveis pela monitorização e pela execução das adaptações, enquanto que o sistema central é responsável pela análise, da informação recolhida pelos subsistemas, e pela decisão das adaptações a serem feitas.

Em [35] é apresentado um sistema que usa um motor de inferência para decidir se é feita alguma alteração à estratégia usada pelo sistema, sendo apresentado um diagrama do sistema na figura 2.18. Para tomar uma decisão, o sistema faz uso de *triggers*, condições que levam a que seja necessária uma adaptação, e de um conjunto de regras, que representam uma relação entre um *triggers* e uma decisão. De maneira a alterar as regras, o sistema faz uso de algoritmos estatísticos que geram novas regras.

São apresentadas seguidamente exemplos de áreas em que já são usadas aplicações com propriedades adaptativas.

- Uma das áreas são as *redes de sensores*, que fazem uso de um grande número de sensores que são usados cooperativamente de maneira a capturar eventos ou monitorizar espaços de maneira eficaz. Estes sensores reagem a alterações no seu estado, por exemplo o estado da bateria, no seu ambiente, existência de ligação à rede, ou nos requisitos do utilizador.

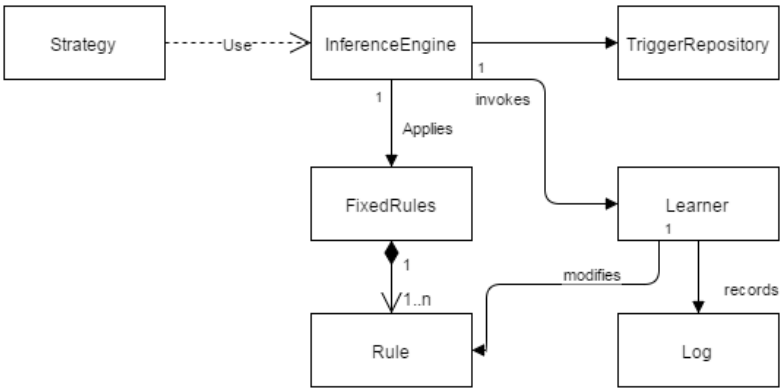

Figura 2.18: Diagrama adaptado, do sistema de reconfiguração dinâmica apresentado em [35].

- A área de *transportes* é outra área que faz uso de propriedades adaptativas, sendo que os sistemas nesta área consistem num conjunto de redes compostas por sistemas heterogéneos. As propriedades adaptativas são usadas na coordenação dos vários sistemas, sendo usadas no controlo de sistemas de transporte automatizados.

O uso de adaptação numa imagem pode ser usado para otimizar o seu processamento, uma vez que mesmo com uma estratégia de escalonamento dinâmico do trabalho (e.g. OpenMP), pode acontecer que um dos *workers* demore muito mais tempo que os restantes, devido às características da imagem, por isso pode ser útil o uso de adaptação dinâmica, para aumentar o número de *workers*, ou redistribuir o trabalho pelos restantes *workers*.

2.5 Conclusão

Neste capítulo foi descrito o estado da arte dos vários tópicos relacionados com o trabalho desenvolvido. Face à diversidade de arquiteturas paralelas e de ambientes de programação ficou claro que são necessárias formas de simplificar o desenvolvimento de programas paralelos nomeadamente para utilizadores que não são especialistas de informática. Uma dessas formas está relacionada com o uso de *algorithmic skeletons* e no capítulo seguinte é feita uma avaliação de duas dessas *frameworks* na programação de um algoritmo de segmentação de imagem 3D.

CAPÍTULO 3

AVALIAÇÃO DA SEGMENTAÇÃO USANDO FERRAMENTAS BASEADAS EM *skeletons*

Neste capítulo encontram-se descritas com maior profundidade os dois *frameworks* de *algorithmic skeletons* avaliados para servirem de suporte a esta tese, FastFlow e SkePU. Estes *frameworks* foram escolhidos devido à sua capacidade de gerarem código para CPUs e GPUs. Para cada um dos *frameworks* é feita uma descrição das suas características e mecanismos mais relevantes. É também feita uma descrição geral dos algoritmos de segmentação e uma descrição mais detalhada do algoritmo *object identifier*, sendo descrita a sua implementação em FastFlow e SkePU. Por fim é feita uma comparação entre os dois *frameworks* e escolhido o *framework* a ser usado neste projeto, bem como as razões para tal.

3.1 FastFlow

O FastFlow é um *framework* de *algorithmic skeletons* implementado em C++, que apresenta suporte para a criação de programas paralelos em plataformas heterogéneas. Este *framework* apresenta-se como um conjunto de bibliotecas de padrões recorrentes de paralelismo, que são parametrizáveis pelo programador. O FastFlow faz uso de *Pthreads* de maneira a conseguir paralelismo em CPU, sendo que permite o uso de CUDA ou OpenCL no paralelismo em GPUs.

O FastFlow está organizado como um conjunto de camadas que abstraem a comunicação, gestão de memória e a criação de *threads* ou processos, comuns aos programas e padrões paralelos. Este tipo de abstração tem como objetivo simplificar a programação paralela estruturada de alto nível, independente da plataforma, e suportar o desenvolvimento de programas paralelos eficientes para arquiteturas homogéneas e heterogéneas.

Os *skeletons* apresentados pelo FastFlow permitem garantir paralelismo de operações, como por exemplo o *pipeline*, e paralelismo de dados, como por exemplo os *skeletons*

farm ou *map*. Usando o FastFlow é possível combinar *skeletons* usando um *skeleton* como parâmetro de outro, por exemplo usar um *farm* como fase de um *pipeline*. No entanto, nem todos os padrões se encontram disponíveis para serem usados em GPU, sendo que apenas estão disponíveis os padrões de *map*, *reduce*, *map-reduce* e *stencil*, continuando a ser possível serem utilizados em conjunto com outros padrões.

3.1.1 Arquitetura

A arquitetura do FastFlow está organizada em três níveis, padrões de alto nível, padrões *core* e blocos básicos. Os *padrões de alto nível* apresentam-se como métodos de classes que podem ser instanciadas, permitindo que o programador paralelize pedaços de código sequencial já existente. Um exemplo deste tipo de padrões é o padrão *parallel-for* usado para paralelizar ciclos. Os *padrões core* permitem a criação de grafos de execução, podendo estes grafos ser cíclicos sem criarem *deadlocks*. Neste nível existem dois padrões (*farm* e *pipeline*) e um modificador (*feedback*) que permite criar grafos cíclicos. Os *blocos básicos* são responsáveis por garantir a paralelização, controlando a criação de *threads* e processos, e garantir a comunicação e sincronização.

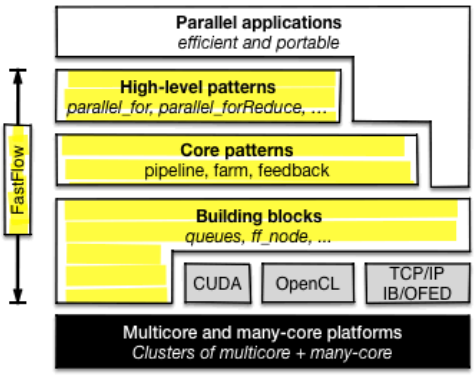

Figura 3.1: Arquitetura FastFlow

3.1.1.1 Padrões de alto nível

Estes padrões facilitam a paralelização de código sequencial já existente sendo utilizados em contextos específicos, por exemplo na paralelização de um ciclo. Dado que estes padrões são apresentados como métodos de classes basta aos utilizadores fazer a sua instanciação de acordo com o problema em questão. Por exemplo, para a paralelização de um ciclo, basta parametrizar o método parallel-for da classe com o mesmo nome [16] que implementa o padrão. O utilizador pode escolher o intervalo de valores do ciclo, o código a ser paralelizado e o número de *workers* a correrem paralelamente. Este padrão apresenta variações na sua parametrização possível, sendo possível determinar o tamanho do incremento em cada ciclo.

Listing 3.1: Paralelização de um ciclo usando o padrão Parallel-for

```
ff::ParallelFor pf;
  pf.parallel_for(0L,N,[&A](const long i) {
    A[i]+=1;
  },nworkers);
```

Para além deste padrão existem ainda os padrões de *parallel-reduce*, *mdf (macro-data-flow)*[3], *pool evolution* (algoritmo genético) e *stencil*. Existe ainda o padrão *stencil-reduce* para execução em GPGPUs, e que permite flexibilidade na escolha entre a execução de um map e um *stencil*, quer sozinhos quer acompanhados de um *reduce*.

Estes padrões estão todos implementados como *templates* em C++, permitindo que sejam estendidos pelo utilizador.

3.1.1.2 Padrões core

Neste nível encontram-se os padrões básicos usados pelo utilizador para criar grafos de execução paralela. Neste nível encontram-se dois padrões (*farm* e *pipeline*) e um modificador (*feedback*), que permite a criação de grafos cíclicos. Estes padrões podem ser usados em conjunto de maneira a criarem grafos complexos. Visto que estes padrões podem ser cíclicos, o FastFlow faz uso de *unbound SPSC buffer* [2] de maneira a evitar a criação de *deadlocks*. A implementação destes padrões faz uso dos blocos básicos presentes no FastFlow.

3.1.1.3 uSPSC

De maneira a evitar os possíveis problemas de *deadlock* introduzidos pelo uso de *bounded queues* quando usados em grafos de execução complexos onde estão presentes ciclos, o FastFlow implementa uma *unbounded SPSC queue*, conhecida como *dSPSC*.

A *dSPSC queue* consiste numa lista ligada de nós que contêm o valor e um apontador para o próximo nó. Inicialmente a cabeça e o fim da lista apontam para nós que contêm valores nulos. Quando é feito um *push* é alocado e preenchido um novo nó, e o fim da lista passa a apontar para esse nó. Quando é feito um *pop*, os valores do nó são retornados, o apontador para a cabeça da lista é atualizado e o nó desalocado. Porém a constante alocação e desalocação de nós introduz algum *overhead*. De maneira a diminuir este *overhead*, a *dSPSC queue* faz uso de uma *bounded queue* para implementar uma *cache* de nós que podem ser reutilizados.

No entanto a obtenção da flexibilidade na *dSPSC queue* implica perda de desempenho em comparado com uma *bounded queue*, mas as duas soluções podem ser combinadas de maneira a se obter o melhor de cada solução. Tal é realizado numa *unbounded SPSC queue* denominada *uSPSC*, e que consiste numa *pool* de *bounded queues* ligadas através de uma *dSPSC queue*. A *dSPSC queue* faz uso de dois apontadores que apontam a *queue* de escrita e a *queue* de leitura, sendo que inicialmente apontam para a mesma *queue*.

Quando é feito um *push*, é verificado se a *queue* de escrita não se encontra cheia, para ser feito um *push* dos dados. Caso esta se encontre cheia, é pedido uma nova *queue* à *pool*, o apontador de escrita da *dSPSC queue* é atualizado para a nova *queue* e os dados são inseridos nesta.

No caso do método *pop*, é verificado se a *queue* de leitura se encontra vazia sendo retornado os dados neste caso. Caso esteja vazia, é verificado se ainda existem dados para consumir e, em caso afirmativo, o apontador de leitura passa para outra *queue*. A *queue* antiga libertada para ser reutilizada pela *pool* de *queues*. Caso os apontadores de escrita e leitura apontarem para a mesma *queue* isso significa que *dSPSC queue* se encontra vazia, caso contrário, é obtido o nó da próxima *queue*.

3.1.1.4 Blocos básicos

Neste nível encontram-se os blocos básicos necessários para a construção dos *padrões core*.Tal inclui as *queues*, os processos e *threads* de execução (*ff_node*) e o necessário para garantir a sua comunicação e sincronização, bem como a distribuição de tarefas pelas várias *threads* (*schedulers*).

Este nível é o responsável por garantir as funções básicas do *framework*: garantir e controlar a paralelização, através da criação, destruição e controlo dos vários nós e padrões usado em conjunto, bem como garantir a comunicação assíncrona entre os diferentes padrões e os nós presentes em cada padrão.

3.1.1.5 ff_node

O *ff_node* é o objeto onde o utilizador o código a executar em paralelo, sendo cada *ff_node* mapeado para uma *thread* de execução. Cada nó destes faz uso de uma *uSPSC queue* onde recebe as tarefas enviadas por outros nós, bem como faz uso de outra *queue* para enviar tarefas para outros nós, sendo estas *queues* não bloqueantes. Estas *queues* são responsáveis pela sincronização entre os diferentes nós, sendo que uma *queue* pertence a um par de nós, onde um deles é o produtor e o outro o consumidor das tarefas.

O FastFlow também apresenta nós que podem receber múltiplos *inputs* bem como enviar vários *outputs*, usando para isso as classes *minode* e *monode*, respetivamente. De maneira a conseguir obter estes nós, o FastFlow faz uso de uma *uSPSC queue* para canal de receção e outra para envio de tarefas do nó, bem como uma *thread* mediadora que garante que as tarefas são enviadas para os canais corretos. Visto que é possível estender as *threads* que garantem a distribuição de tarefas entre os nós, o utilizador pode implementar a sua própria política de distribuição.

3.1.2 Programação em FastFlow

O FastFlow apresenta um conjunto de *skeletons* ao programador para a criação de programas que correm em CPU e um conjunto de *skeletons* que correm em GPU. Isto leva a que a programação para cada um seja diferente.

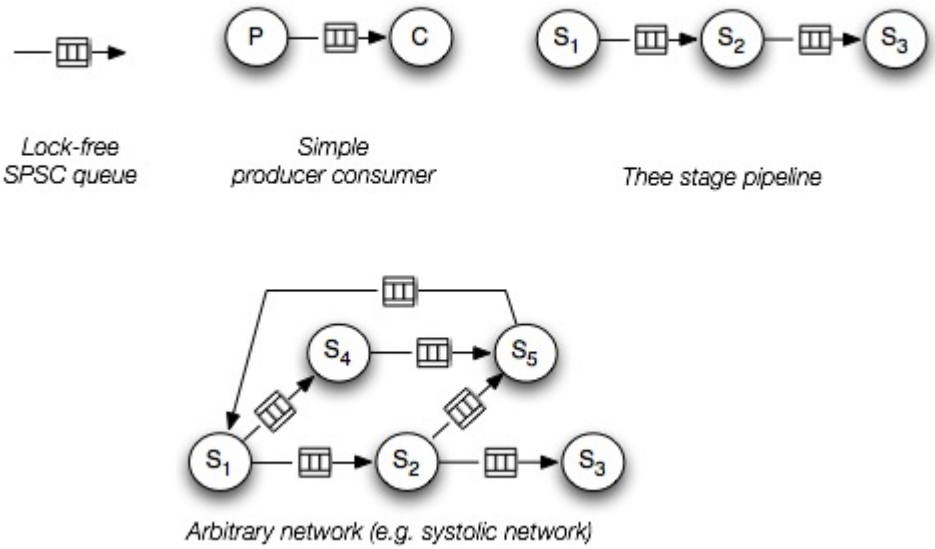

Figura 3.2: Exemplos de grafos possíveis de construir em FastFlow

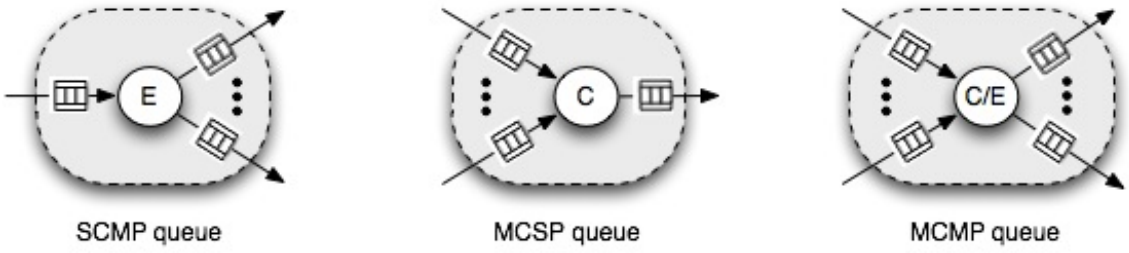

Figura 3.3: Exemplos de *minode* e *monode* em FastFlow

3.1.2.1 Programação em CPU

A maior parte do *skeletons* implementados pelo FastFlow são para CPU, estando incluídos neste grupo os *skeletons* de *farm*, *pipeline*, *parallel-for*, *map* e *stencil*.

O FastFlow faz uso de uma classe *ff_node* que o programador estende para inserir o código a ser paralelizado,em particular, o código a ser executado pelo nó deve ser implementado no método *svc*. Este apresenta como parâmetro um apontador para uma tarefa cujo tipo também deve ser definido pelo programador, e que o nó pode consultar ou modificar. No fim do método, é retornado novamente um apontador para a tarefa, modificada ou nova, a ser enviada ao próximo nó.

Na *main* do programa o utilizador cria uma instância do *skeleton* pretendido e insere os nós a serem usados, antes da sua execução. No caso do *pipeline*, cada nó introduzido representa uma fase, sendo as fases executadas pela ordem em que são inseridas. No caso de um *farm*, o utilizador insere a quantidade de nós que pretende correr em paralelo. Tem também a possibilidade de introduzir um nó emissor, que é o primeiro a executar e

que envia tarefas para os nós principais, e um nó coletor, que recebe as tarefas enviadas pelos nós principais e é o último a executar. Como os *skeletons* estão implementados como extensões da classe *ff_node*, eles podem ser usados pelos outros *skeletons*, podendo o utilizador inserir uma *farm* como uma fase do *pipeline*, por exemplo.

Listing 3.2: Exemplo de programação de um simples *farm* em FastFlow

```
struct Worker: ff_node {
  void *svc(void *t) {
    cout << "Hello I'm the worker\n";
    return t;
  }
};
int main(int argc, char *argv[]) {
  vector<ff_node *> Workers;
  for(int i=0;i<nworkers;++i) Workers.push_back(new Worker);
  ff_farm<> myFarm(Workers);
  myFarm.run_and_wait_end();
  return 0;
}
```

3.1.2.2 Programação em GPU

A programação em FastFlow em GPU depende do uso dos padrões de *stencil-reduce* e suas variações. De maneira a usar estes padrões, o utilizador tem disponível um conjunto de *macros* que lhe permitem indicar que tipo de função se pretende executar em GPU. Em cada macro o utilizador também define o tipo dos valores de entrada e de saída, bem como de valores auxiliares. No fim da *macro*, o utilizador indica o código a ser executado.

De maneira a que a *macro* obtenha valores para processar, o utilizador deve estender a classe *baseCUDATask*, onde indica onde se encontram os valores de entrada e auxiliares, bem como onde os resultados devem ser guardados.

Listing 3.3: Exemplo simples de programação em GPU

```
FFMAPFUNC(mapF, unsigned int, in, return in + 1;);

class cudaTask: public baseCUDATask<unsigned int, unsigned int, unsigned char,
    unsigned char, int, char, char> {
public:
  void setTask(void* t) {
      cudaTask *t_ = (cudaTask *) t;
      setInPtr(t_->in);
      setOutPtr(t_->in);
      setSizeIn(inputsize);
  }
  unsigned int *in;
};

int main(int argc, char * argv[]) {
```

```
16    cudaTask *task = new cudaTask();
17    task->in = new unsigned int[inputsize];
18    for (size_t j = 0; j < inputsize; ++j)
19      task->in[j] = j;
20    FFMAPCUDA(cudaTask, mapF) *myMap = new FFMAPCUDA(cudaTask, mapF)(*task);
21    myMap->run_and_wait_end();
22  }
```

3.2 SkePU

O SkePU é um *framework* de *algorithmic skeletons* implementado como uma biblioteca em C++, que permite a criação de programas paralelos que podem executar em múltiplos CPUs, através do uso de OpenMP para garantir o paralelismo, ou em um ou vários GPUs, através do uso de CUDA ou OpenCL.

O SkePU faz uso de um conjunto limitado de *skeletons*, tendo disponíveis os padrões de *map*, *reduce*, *scan* e *farm*, bem como algumas variações. No entanto todos estes padrões podem ser usados de igual maneira em CPU como em GPU.

O SkePU apresenta ainda um mecanismo que permite escolher automaticamente qual a implementação dos *skeletons*, em CPU ou GPU, que apresenta melhor desempenho [19]. Para fazer isso, o SkePU gera vários planos de execução com base no problema e nos *skeletons* escolhidos, sendo selecionado o plano de execução que apresenta melhor resultado. Ao utilizador é permitido afinar como são gerados os planos de execução pelo SkePU, podendo especificar o número de *threads* a serem usadas e qual o *backend* em que o *skeleton* é executado, considerando um intervalo particular de tamanho de dados a ser processado.

Existem ainda estruturas de dados inteligentes [18], *vector* e *matrix*, das quais o *framework* faz uso. Estas estruturas são responsáveis por identificar os dados que se encontram em CPU e GPU, e garantir que se encontram válidos e atualizados. De maneira a obter o melhor desempenho possível, estas estruturas apenas fazem transferência de dados entre CPU e GPU quando é estritamente necessário.

3.2.1 Estruturas de dados inteligentes

O SkePU faz uso de estruturas de dados inteligentes, *vector* e *matrix*, para a gestão de memória. De maneira a isso ser possível, cada estrutura mantém um registo das diferentes cópias de dados nas diferentes memórias. De maneira a diminuir a comunicação entre o GPU e o CPU, a transferência de dados apenas é feita quando é acedido um dos elementos da estrutura pelo CPU.

Cada registo de cópia presente numa estrutura é identificada por <*devID*, *offset*, *n*>, onde *devID* é o identificador do *device*, *offset* é o índice do primeiro elemento na cópia principal, e *n* é o número de elementos guardados nesta cópia. Num mesmo *device* é

possível existirem várias cópias dos dados que se sobrepõem. Cada cópia guarda *flags* de *modified*, *valid* e *last used*.

Quando é feita uma leitura de um ou vários elementos na estrutura na cópia principal, todas as cópias, presentes nos *devices* que contêm elementos que serão acedidos, transferem as suas cópias para a cópia principal e limpam as suas *flags* de *modified*. No fim, a cópia principal é definida como válida. Quando é feita uma escrita na cópia principal e esta não se encontra válida, são copiadas todas as cópias presentes em *device* que contêm elementos que vão ser escritos e foram modificadas, sendo que essas cópias depois são declaradas como inválidas.

Quando é feita uma leitura no *device*, a estrutura procura por uma cópia com <*devID, offset, n*>, e caso esta não exista é criada. Caso já exista uma cópia e esta não se encontre válida, os dados são copiados de outras cópias presentes nos vários *devices*, incluindo o *device* para o qual a cópia está a ser feita, ou então da cópia principal. Quando é feita uma escrita no *device*, é procurada uma cópia com <*devID, offset, n*>, caso esta não exista é criada. Caso esta cópia exista, as outras cópias que contêm elementos que serão escritos são declaradas como inválidas. Depois de ser feita a escrita na cópia do *device*, esta é indicada como modificada e válida.

A diferença de uma estrutura *matrix* para uma estrutura *device* é que esta é identificada por <*devID, offset, nrows, ncols*>, onde *nrows* indica o número de linhas na cópia do *device* e *ncols* indica o número de colunas. As cópias em *device* podem conter qualquer sub-matriz da matriz da cópia original. Quando é feita a cópia de dados entre diferentes cópias, podem ser necessárias várias operações de cópia para cada intervalo de elementos na matriz.

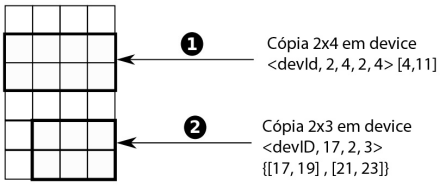

Figura 3.4: Exemplos de de estrutura *matrix* e as suas cópias.

3.2.2 Escolha automática de skeletons

De maneira a escolher qual o *backend* em que os *skeletons* vão executar o SkePU oferece três opções ao utilizador: através do código; em tempo de compilação através do uso de *flags*, e.g. usando a *flag* SKEPU_CUDA; ou então, quando são específicados os múltiplos *backends*, o utilizador pode deixar o SkePU escolher através do seu algoritmo de escolha de *skeletons*.

Visto que seria impossível calcular qual a melhor configuração de execução, o algoritmo assume que, se para duas dimensões de um mesmo problema a melhor configuração

possível é igual para os dois casos, o mesmo é verdade para os tamanhos dos problemas entre esses dois valores.

O algoritmo começa por criar uma árvore com um nó que contém os tamanhos possíveis para o problema (e.g. um intervalo de valores). Caso a configuração de execução escolhida seja igual para o valor inicial e final do intervalo do nó, essa é a configuração escolhida para esse intervalo de valores. Caso as configurações sejam diferentes para estes dois valores, o nó é dividido e são calculados as melhores configurações para os cantos/limites dos novos nós. Este processo é repetido até já não ser possível para os nós existentes, ou então ao atingir parâmetros definidos pelo utilizador. O utilizador pode especificar como parâmetros que terminam o algoritmo, a profundidade máxima da árvore, o número de nós máximo da árvore ou então pode definir um *timeout*.

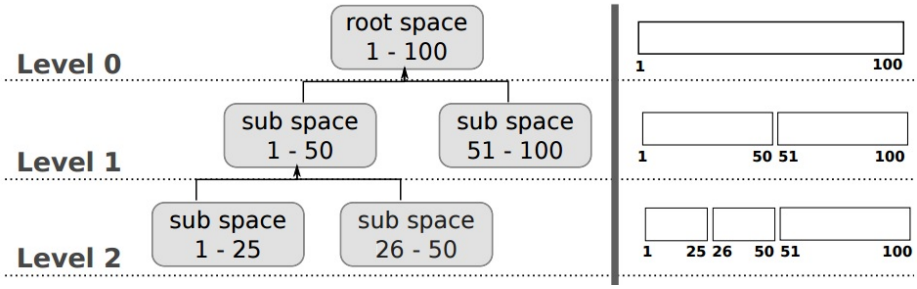

Figura 3.5: Exemplos de árvore de planos de execução.

O utilizador também pode determinar quais os planos de execução associados a um *skeleton* na execução do seu programa, utilizando a classe *Tuner* para criar os planos de pretendidos. Para cada intervalo de valores que o utilizador indicar, a implementação do *skeleton* faz uso do *backend* escolhido e outros parâmetros, e.g. o número de *threads* caso execute em CPU ou o número de blocos caso execute em GPU.

No fim da execução do algoritmo ficamos com uma lista dos melhores planos de execução em que, para cada intervalo de tamanhos do problema, indica qual o melhor *backend* para executar, o número de *threads* a usar e o número de blocos. De seguida segue-se um exemplo de possíveis planos de execução que se podem obter no fim do algoritmo:

1	750000	CL_BACKEND	32	32768
750001	1250000	CL_BACKEND	32	512
1250001	2250000	CL_BACKEND	128	2048
2250001	3750000	CL_BACKEND	32	2048
3750001	4250000	CL_BACKEND	32	16384
4250001	5250000	CL_BACKEND	128	32768
5250001	5750000	CL_BACKEND	128	16384
5750001	6250000	CL_BACKEND	128	32768
6250001	7250000	CL_BACKEND	512	16384
7250001	8250000	CL_BACKEND	256	16384

3.2.3 Programação em SkePU

Todos os *skeletons* disponibilizados em SkePU conseguem executar tanto em CPU como em GPU sem ser necessário fazer qualquer alteração. Isto leva a que existam algumas limitações à programação das funções a serem paralelizadas, dado que é necessário que seja possível executarem tanto em CPU como GPU. Isto leva a que apenas tipos primitivos de dados e estruturas possam ser usados.

O programador define as funções a serem paralelizadas através de *macros* que o SkePU fornece. Nestas macros, o utilizador indica o tipo de função a ser executada, o tipo dos dados e qual o código a ser usado por essa função. Na função *main* são inicializados os *skeletons* bem como as estruturas de dados que vão ser usadas, sendo depois o *skeleton* executado.

Listing 3.4: Exemplo de multiplicação de matrizes em SkePU

```
1  BINARY_FUNC(mult_f, float, a, b, return a*b; )
2
3  BINARY_FUNC(plus_f, float, a, b, return a+b; )
4
5  int main(){
6     skepu::MapReduce<mult_f, plus_f> dotProduct(new mult_f, new plus_f);
7     skepu::Vector<float> v0(20, (float)2);
8     skepu::Vector<float> v1(20, (float)5);
9     float r = dotProduct(v0, v1);
10    return 0;
11 }
```

3.3 Algoritmos de segmentação

Os algoritmos de segmentação de imagem são algoritmos de processamento de imagem que têm como principal objetivo particionar a imagem em grupos de *pixels*, ou *voxels*, homogéneos, ou que estejam relacionados por algum critério. Estes algoritmos são usados para que seja mais fácil a análise posterior à imagem, sendo muitas vezes usadas na deteção de objetos ou formas.

Existem vários tipos de algoritmos de segmentação de imagem. Entre eles encontram-se os algoritmos de *thresholding*, onde todos os *pixels* com um valor abaixo de uma determinada constante são transformados em *pixels* pretos, sendo transformados em brancos caso se verifique o oposto. Outro tipo de algoritmo são os algoritmos de *clustering*, onde são criados *n* grupos que representam um valor de uma característica da imagem, cor, brilho, entre outros. A cada *pixel* é atribuído o grupo mais próximo do seu valor, sendo o valor destes grupos recalculados com base na média dos *pixels* que o compõe. Este processo repete-se até se atingir um número de iterações definido pelo utilizador, ou até já não existir alteração nos grupos formados. Um exemplo deste tipo de algoritmo é o algoritmo *k-means*. As imagens 3.6 e 3.7 mostram a aplicação do algoritmo *k-means*.

Figura 3.6: Imagem original

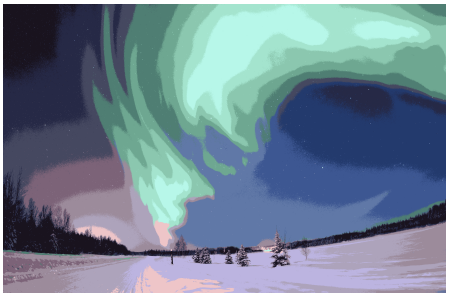

Figura 3.7: Resultado final

Os algoritmos de *watershed* são outro tipo de algoritmos usados na segmentação de imagem. A ideia deste algoritmo vem da geografia, transformando a imagem a preto e branco numa carta topográfica onde, quanto maior for o nível de cinzento de um *pixel*, maior será a sua altura. Desta maneira, se imaginarmos que é inserida água em qualquer ponto na imagem, esta irá convergir para um mínimo local, enchendo uma bacia. Onde duas bacias se encontram, é desenhada uma linha divisora, sendo que no fim do algoritmo cada bacia representa uma região da imagem.

Um exemplo de algoritmo de *watershed* é o algoritmo de Meyer, o qual faz uso de uma fila de prioridades para preencher as diferentes regiões da imagem. O algoritmo começa por escolher um conjunto de *pixels* iniciais que representam diferentes regiões, sendo atribuído a cada um, um identificador único. Para cada *pixel* inicial, os seus vizinhos são inseridos na fila de prioridades, sendo a ordenação da fila definida pelo nível de cinzento. De seguida, é retirado o *pixel* com menor prioridade e são verificados os seus vizinhos. Os vizinhos que não tenham um identificador atribuído são acrescentado à fila de prioridades, sendo verificados os identificadores dos restantes. Caso todos apresentem o mesmo identificador, este é atribuído ao *pixel*, caso contrário, este é definido como uma linha divisora entre regiões. Este processo repete-se até que a fila de prioridades fique vazia.

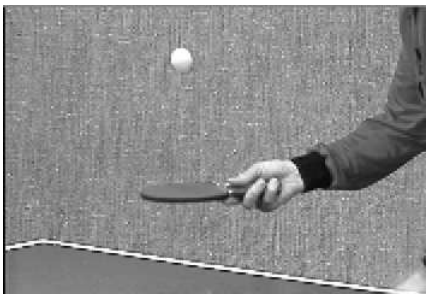

Figura 3.8: Imagem original

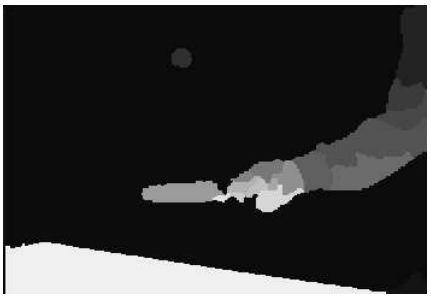

Figura 3.9: Linhas divisórias

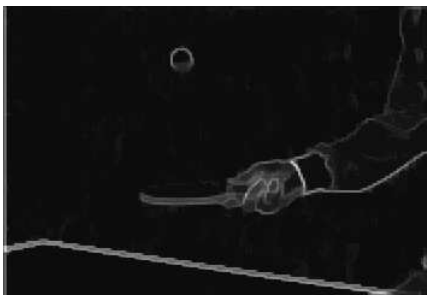

Figura 3.10: Resultado da segmentação com o algoritmo *watershed*

3.3.1 Algoritmo *Flood Fill*

O algoritmo *flood fill* é um algoritmo sequencial usado no processamento de imagem para o preenchimento de objetos ou áreas, sendo usado em programas de edição de imagem.

O algoritmo consiste em pintar, com a cor escolhida, o *pixel* escolhido inicialmente, e verificar os *pixels* vizinhos. Para aqueles que apresentarem uma cor igual à cor original do *pixel*, é aplicado o mesmo procedimento. Este processamento pode ser feito de forma recursiva, ou através do uso de uma fila onde são inseridos os *pixels* a serem processados posteriormente. As figuras 3.11 e 3.12 ilustram a aplicação do algoritmo, onde os *pixels* laranja são os quais são aplicados o algoritmo.

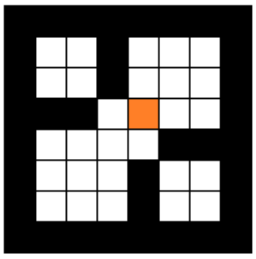

Figura 3.11: Condições iniciais do algoritmo *flood fill*.

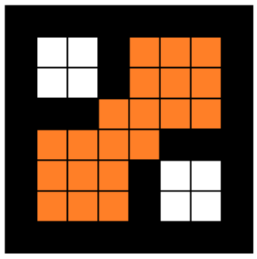

Figura 3.12: Resultado final do algoritmo *flood fill*.

3.3.1.1 *Object Identifier*

O algoritmo *object identifier* é um algoritmo desenvolvido para identificar objetos numa imagem a três dimensões, fazendo uso de paralelismo em CPU e GPU para o seu processamento. Este algoritmo é baseado no algoritmo sequencial de *flood fill* usado para o preenchimento de objetos. De maneira a ser possível processar um elevado volume de dados em GPU, estes têm que ser divididos em blocos que são processados paralelamente, devido às limitações de memória dos GPUs. O algoritmo encontra-se optimizado para processar as fronteiras entre os vários blocos nas etapas onde isso é necessário.

O algoritmo apresenta três etapas onde os blocos são processados de forma paralela, sendo cada etapa dependente do processamento feito pelas etapas anteriores. Na primeira

etapa, a cada *voxel* pertencente a um objeto, é atribuído inicialmente um identificador correspondente à sua posição na imagem (ver figura 3.13). Depois disso, são verificados os identificadores dos *voxels* vizinhos e selecionado aquele que apresenta o identificador menor, sendo menor que o identificador atual. Esta verificação é repetida até que nenhum *voxel* do bloco altere o seu identificador, ficando assim todos os *voxels* do mesmo objeto, presente no bloco, com o mesmo identificador (ver figura 3.14.

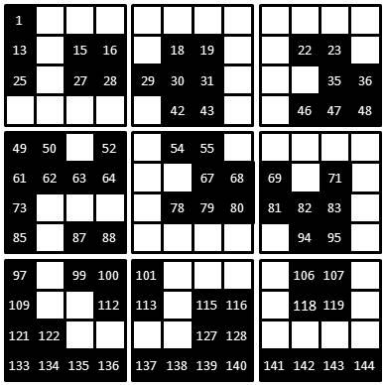

Figura 3.13: Numeração inicial com base no índice.

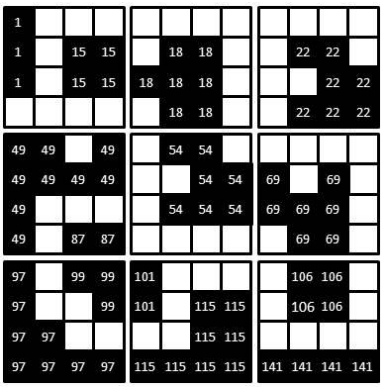

Figura 3.14: Numeração dos identificadores no fim da primeira fase do algoritmo

A segunda etapa consiste em obter a conectividade entre objetos em blocos distintos. Nesta fase, cada *voxel* verifica se se encontra numa fronteira de um bloco e, caso esteja, verifica os identificadores vizinhos por identificadores distintos. Quando tal é encontrado, é estabelecida uma relação num vetor de alterações de maneira a que mais tarde sejam realizadas modificações de maneira a se obter apenas um identificador por objeto. Depois disto é realizado um processamento adicional de otimização do vetor de alterações (ver figura 3.15). Este processamento consiste em percorrer o vetor de alterações e procurar o identificador representante, o identificador com menor valor, para cada identificador encontrado.

A terceira e última etapa consiste em atualizar os identificadores de cada *voxel* presente na imagem para os identificadores presentes no vetor de alterações, ficando assim todos

os objetos identificados por todos os blocos.

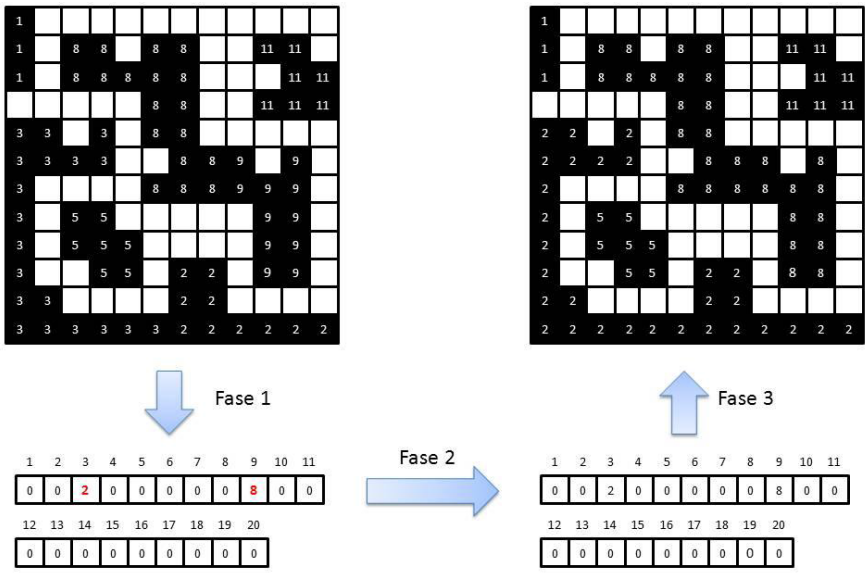

Figura 3.15: Terceira fase do algoritmo.

3.4 Conclusão

A escolha da *framework* a utilizar recaiu no FastFlow uma vez que apresenta vantagens em relação ao SkePU. Esta escolha deveu-se à falta de suporte no SkePU, que na altura desta escolha não tinha sido atualizado há vários meses, e mesmo quando se tentou contactar alguém da equipa de desenvolvimento do SkePU não foi obtida nenhuma resposta. Pelo contrário, sabíamos que o FastFlow tinha sido atualizado recentemente, bem como iria ser atualizado regularmente nos meses seguintes. Outra razão que levou à escolha do FastFlow, foi a existência dos mecanismos de escolha automática de *backend* no SkePU, o que leva a que a contribuição possível não tenha tanto impacto, visto que aquilo que se pretendia implementar em SkePU dificilmente teria um desempenho melhor do que o que já existia.

Na escolha de *framework* também teve impacto o algoritmo *Object Identifier*, uma vez que o FastFlow nos permite definir mecanismos de escalonamento que sejam mais adequados ao algoritmo, e.g. fazer a divisão dos blocos a serem processados em cada tarefa. O FastFlow também nos oferece maior controlo na escolha da execução do algoritmo, uma vez que nos permite escolher os números de nós que pretendemos em cada fase, bem como o *backend* em que estas vão ser executadas.

O FastFlow também já tinha sido comparado com outros modelos de programação paralela, tendo sido comparado com a implementação em CUDA e OpenMP do algoritmo da bi-segmentação e histerese. Para esta comparação foram usadas três imagens que apresentam os mesmos valores para largura, altura e profundidade, sendo usados como valores 100, 200 e 400. A imagem 3.16 apresenta os tempos obtidos nesta comparação.

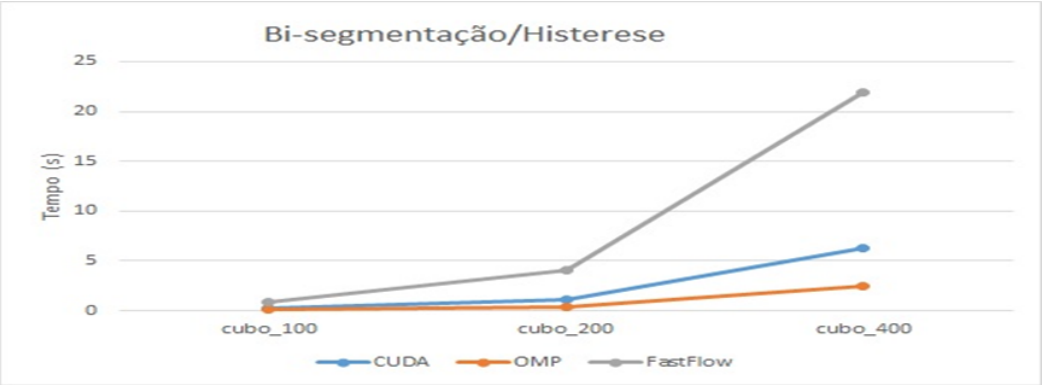

Figura 3.16: Comparação de tempos do algoritmo bi-segmentação/histerese entre CUDA, OpenMP e FastFlow.

CAPÍTULO 4

SOLUÇÃO PROPOSTA

Neste capítulo encontra-se uma descrição da arquitetura e da implementação da aplicação *Object Identifier* no contexto do *framework* FastFlow, e das propriedades adaptativas desenvolvidas na solução.

4.1 Arquitetura

O algoritmo *Object Identifier* é constituído por várias fases, as quais se descrevem nesta secção. As várias fases necessárias para a aplicação do algoritmo são também aqui descritas.Segue-se a descrição da arquitetura da solução proposta para a implementação de propriedades autonómicas e que inclui a extensão do *framework* FastFlow.

4.1.1 Object Identifier

A implementação do algoritmo *object identifier* divide-se em quatro passos, gerar ou ler a imagem, dividir a imagem em várias partes, aplicar o algoritmo *object identifier* e guardar a imagem depois do algoritmo aplicado. A imagem 4.1 mostra os vários passos da aplicação.

No primeiro passo o utilizador tem duas opções, gerar uma imagem nova ou ler uma imagem já existente. Caso o utilizador escolha ler uma imagem já existente, esta é carregada para memória e enviada para o próximo passo da aplicação. Caso o utilizador escolha gerar uma imagem nova, define o tamanho pretendido para a nova imagem e o tamanho dos blocos usados para gerar um padrão aleatório. Para cada bloco da imagem é selecionado aleatoriamente um de três padrões para o bloco da imagem, listas verticais, horizontais ou então um padrão de xadrez.

No segundo passo é feita a divisão da imagem em vários blocos que serão processados paralelamente pelas restantes fases. Aqui o utilizador define o tamanho de cada bloco em que a imagem deve ser dividida. Seguidamente, a esses blocos são aplicadas todas as

fases do algoritmo *object identifier*. Todas as fases do algoritmo estão implementadas com recurso a uma *ff_a_farm* (descrita na secção seguinte), o que permite ao utilizador controlar o número de nós para cada fase, e nós do tipo *ff_a_node*(também descrito na próxima secção) onde estão implementadas as versões em CPU e GPU do algoritmo, podendo assim o utilizador escolher em que *backend* pretende que cada fase corra. A imagem 4.2 mostra as várias fases da execução do algoritmo.

No último passo a imagem alterada pelo algoritmo é guardada num local especificado pelo utilizador.

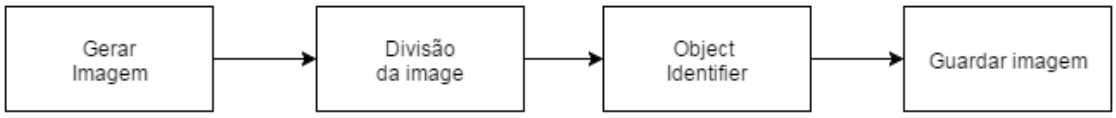

Figura 4.1: Execução completa do padrão Object Identifier

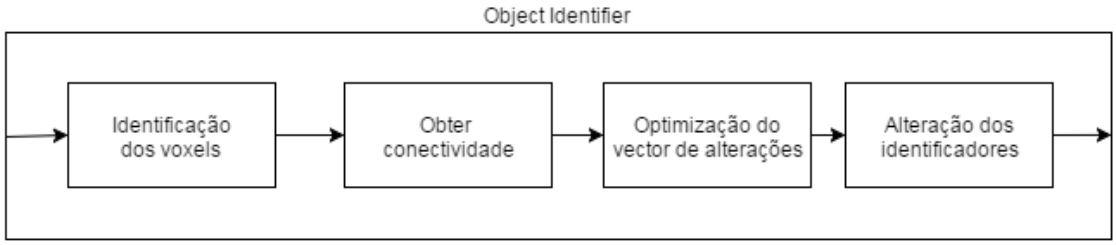

Figura 4.2: Execução detalhada do padrão Object Identifier

4.1.2 Arquitetura do sistema

Este trabalho insere-se num trabalho de doutoramento onde está a ser desenvolvido um sistema com propriedades adaptativas, com foco no processamento de imagem. A solução implementada faz parte de um conjunto de soluções que afetam o sistema, de maneira a que este cumpra os *Qualities of Service* (QoS) definidos. Este sistema segue um ciclo MAPE(ver 2.4) na sua maneira de se adaptar, como demonstra a figura 4.3.

Como a figura indica, na fase de monitorização é verificado se foi quebrado algum QoS, por exemplo, exceder o tempo de execução expectável, o tamanho das filas de cada nó ser irregular ou o sistema gastar mais energia do que o suposto, entre outros. Na fase seguinte, de análise, o sistema verifica o que é que pode mudar no sistema de maneira a cumprir o QoS, por exemplo, mudar o número de nós, alterar as filas de cada nó, mudar o *backend* de execução. Na fase de planeamento é feita uma decisão de como se vai mudar o sistema de maneira a se obter a otimização desejada, sendo essa decisão transmitida à fase de execução, que é a responsável por fazer as mudanças necessárias no sistema.

Idealmente para cada fase deste ciclo existe um processo correspondente responsável. No entanto de momento existe um processo chamado *supervisor* responsável pela monitorização, análise e planeamento, estando a execução do sistema e das suas adaptações a

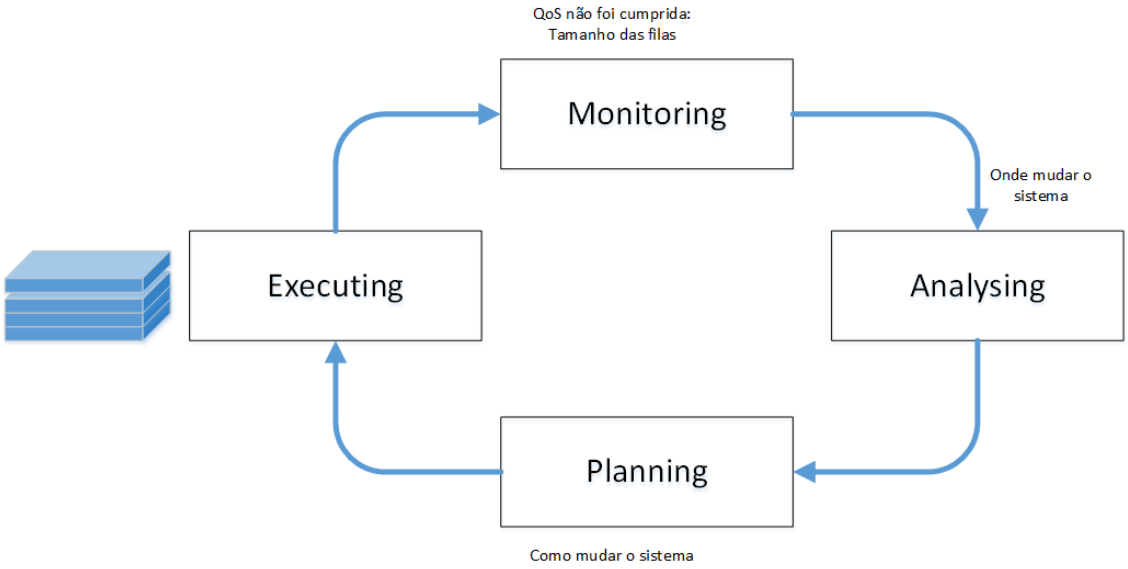

Figura 4.3: Ciclo MAPE do sistema

ser feita com a solução proposta que se encontra ligada a um adaptador responsável por comunicar com o *supervisor*. A solução proposta tem como base o padrão *farm*, presente no FastFlow. A figura 4.4 mostra o sistema atual e as várias fases que se inserem em cada fase, com as fases de *monitoring* , *analysing* e *Planning* a serem feitas pelo *supervisor* e a fase de *executing* a ser dividida pelo *Adapter/Wrapper* e pela *farm* da solução.

4.1.3 Arquitetura das propriedades adaptativas

De maneira a se implementarem propriedades adaptativas, foi necessário criar novas classes bem como estender classes já existentes no FastFlow, as quais devem então ser utilizadas para se ter acesso a essas propriedades.

A classe *ff_a_node*, que estende a classe *ff_node*, é a classe utilizada para a implementação de nós nos programas. Nesta classe está desenvolvida a propriedade adaptativa que possibilita escolher qual o *backend* em que o nó é suposto correr. Para suportar esta escolha, a classe apresenta dois métodos que o utilizador deve implementar com o seu código, *svc_cpu* e *svc_gpu*, onde é implementado o código a correr em CPU e GPU, respetivamente. Estes dois métodos substituem o método *svc* que costuma ser implementado pelo utilizador. A escolha é feita com base numa *flag* que indica qual o *backend* pretendido, podendo o utilizador alterar o seu valor quando desejar, i.e. dinamicamente.

Adicionalmente, no fim de cada tarefa, a classe *ff_a_node* também recolhe informação sobre o tempo de execução da última tarefa ou sobre o número de tarefas em espera para serem processadas pelo nó. Assim, no fim de cada tarefa, cada nó faz uma medição através da classe *measurer*, e gera um estado sobre a sua execução na forma da classe status.

A classe *measurer* é responsável pelas medições da classe *ff_a_node* e gerar o seu estado atual. As estratégias de medição são implementadas pelo utilizador, sendo que pode

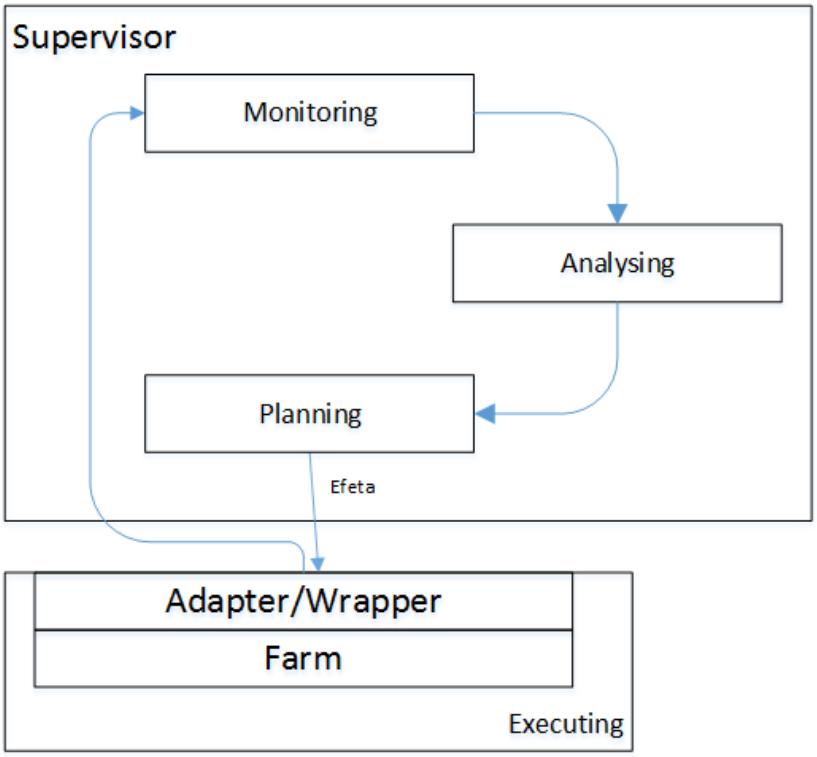

Figura 4.4: Sistema atual com respetivas fases do ciclo MAPE

ser feito uso da informação recolhida pelo nó que está a ser medido. Alguns exemplos de estratégias de medição são o tempo de execução médio das tarefas ou o número de tarefas já processadas. A classe *status* é uma classe que representa o estado do nó, sendo esta instanciada pela classe *measurer*. Nesta classe o utilizador implementa como é feita a comparação entre diferentes estados.

A classe *ff_a_farm* é uma extensão da classe *ff_farm*, e que é usada pelos utilizadores quando se pretende usar nós do tipo *ff_a_node* em conjunto com uma *farm*. Nesta classe é possível aumentar e diminuir o número de nós ativos no momento,i.e. nós que estão disponíveis para receber tarefas. A classe *a_scheduler* é responsável pela distribuição das tarefas pelos vários nós. De maneira a saber quais os nós disponíveis para receber tarefas, esta classe guarda o número de nós ativos. Quando um nó é retirado do conjunto de nós ativos, esta classe é ainda responsável por redistribuir as tarefas que ficaram por processar nesse nó, pelos restantes nós que continuam ativos.

4.2 Implementação

Nesta secção encontram-se os detalhes da implementação do algoritmo *object identifier* usando o padrão *farm* desenvolvido com propriedades adaptativas. Segue-se a descrição da implementação das classes necessárias que suportam as propriedades adaptativas desenvolvidas nesta solução.

4.2.1 Object Identifier

A implementação do algoritmo *object identifier* apresenta quatro fases durante a sua execução, as quais a seguir se detalham.

4.2.1.1 Leitura da imagem

Na primeira fase, o utilizador tem de carregar uma imagem a partir do *input* dado para memória. Uma vez que se espera que os valores da imagem sejam representados por *unsigned chars*, é preciso transformá-los para o tipo *unsigned int*, de maneira a ser possível representar todos os identificadores possíveis. Assim a imagem é percorrida e, para cada elemento que apresente a cor preta, o seu valor passa a ser 1 e para cada elemento de cor branca passa a ser 0.

4.2.1.2 Divisão da imagem

Nesta fase é feita a divisão da imagem em vários blocos, com tamanhos definidos pelo utilizador. Isto é feito de maneira a que todas as fases do algoritmo *object identifier* usem a mesma divisão no processamento. A imagem é dividida num vetor de blocos onde cada um guarda os valores iniciais e finais de x, y e z de cada bloco.

4.2.1.3 Algoritmo object identifier

O algoritmo *object identifier* está definido como uma classe que inicializa e executa todas as fases do algoritmo. Cada fase do algoritmo está implementada com uma *ff_a_farm* e nós do tipo *ff_a_node* onde estão implementadas as versões em CPU e GPU do algoritmo. Para cada nó, está também definida uma estratégia de medição com base no número de tarefas em espera, sendo o estado de cada nó esse valor. O algoritmo recebe, para além da imagem, o vetor com os blocos divididos na fase anterior, sendo estes divididos através dos emissores das *farms* de cada fase.

Ao executar o algoritmo, a classe executa cada fase de forma sequencial, ou seja, depois de todos os blocos terem sido processados por uma é que se segue a execução da fase seguinte. Isto é necessário porque o processamento de cada fase do algoritmo depende do resultado das fases anteriores. A classe também é responsável por inicializar o grafo que é usado nas duas últimas fases, bem como garantir que este é passado de uma fase para a outra.

4.2.1.4 Escrita da imagem

Depois de aplicado o algoritmo à imagem, é criado um ficheiro com o *output* dado pelo utilizador, sendo os dados da imagem copiados para esse ficheiro.

4.2.2 Propriedades adaptativas

Tal como referido, foram escolhidas duas propriedades adaptativas a serem implementadas, a possibilidade de alterar o *backend* em que o código de um nó corre, e a possibilidade de alterar o número de nós em execução, sendo possível fazer estas alterações em tempo de execução. Além disso, foi implementado uma distribuição de tarefas com base nos estados dos nós, de maneira a se escolher o nó em melhor condições para processar cada tarefa. Os estados são obtidos através de medições em cada nó, sendo que o utilizador escolhe que tipo de medição quer fazer e o que deve ser avaliado em cada estado. A descrição que se segue, pretende realçar os detalhes de implementação que suportam as propriedades descritas.

4.2.2.1 Alterar *backend* de cada nó

De maneira a ser possível alterar o *backend* em que cada nó corre o seu código, foi feita uma extensão da classe *ff_node* do FastFlow. Esta é a classe que o utilizador estende de maneira a indicar o código que deve ser paralelizado, o qual é inserido no método *svc* da classe que é chamado sempre que uma tarefa é recebida pelo nó.

A classe *ff_a_node* é uma extensão da classe *ff_node* que contém dois métodos virtuais, e que são implementados pelo utilizador para especificar o código do nó para CPU e para GPU. Estes métodos são chamados de *svc_cpu* e *svc_gpu* e substituem o método *svc* do *ff_node* que anteriormente tinha de ser implementado pelo utilizador. De maneira a saber qual dos dois métodos deve ser utilizado, a classe *ff_a_node* contém uma variável que guarda qual o *backend* a usar, podendo o utilizador definir qual o seu valor na inicialização da classe, ou mais tarde através do método *setEnv* que recebe o *backend* como parâmetro. Caso nenhum valor seja dado, este é inicializado com o *backend* de CPU. Nesta classe, o método *svc* está implementado de maneira a receber a tarefa a processar e, com base no *backend* escolhido, chamar o método apropriado, enviando a tarefa recebida como parâmetro. O resultado do método invocado é guardado, de maneira a que mais tarde seja retornado pelo método *svc*.

O código no *listing* 4.1 mostra como é feita a escolha do *backend* do nó em tempo de execução quando é recebida uma tarefa, e é executado o método *svc* do nó.

Listing 4.1: Implementação da classe *ff_a_node*

```
1  class ff_a_node : public ff_node {
2    ff_a_node(measurer * meas, Environment env = CPU):
3          m_env(env), last_time(0), buffer_size(0), measure(meas), status(Null){};
4
5    void * svc(void * task){
6          clock_t begin = clock();
7          void * res = NULL;
8          switch(m_env) {
9              case CPU:
10                 res = svc_CPU(task);
```

```
11                  break;
12              case GPU:
13                  res = svc_GPU(task);
14                  break;
15          }
16          clock_t end = clock();
17          last_time = end - begin;
18        buffer_size = inBuffer.length();
19          status = measure->strategy(this);
20        return res;
21      }
22  }
```

4.2.2.2 Alterar número de nós

Para se poder alterar o número de nós em execução num dado momento, foram feitas as extensões *ff_a_farm*, à classe *ff_farm* do FastFlow, e *a_scheduler* à classe *ff_loadbalancer*.

A classe *ff_loadbalancer* é a classe responsável por fazer a distribuição de tarefas pelos vários nós da *farm*. De maneira a ser possível indicar o número de nós ativos, i.e. os nós disponíveis a receber tarefas num determinado momento, esta classe foi estendida de modo a guardar uma variável com esse valor. Essa variável é usada na distribuição de tarefas durante a escolha dos nós que vão receber as tarefas, sendo apenas escolhidos os que estão ativos no momento.

Para alterar o número de nós ativos, a classe *ff_a_scheduler* apresenta dois métodos que permitem incrementar e decrementar esse valor. Quando é decrementado o número de nós, é verificado se existem pelo menos dois nós ativos no momento, de maneira a se garantir que existe pelo menos um nó ativo depois de reduzido o número de nós. Caso isso não se verifique, é retornada uma mensagem de erro a indicar que tem de existir pelo menos um nó ativo. Caso a redução seja possível, é decrementado o valor da variável que guarda o número de nós ativos. Ao nó retirado vão ser retiradas todas as tarefas que este tem em espera para processar, sendo estas redistribuídas pelos restantes nós.

Quando é invocado o método para incrementar o número de nós, o utilizador tem a possibilidade de usar uma instância de um nó como parâmetro. O método começa por verificar se o número de nós ativos é igual ou inferior ao número de nós que já estiveram ativos. Caso isso não se verifique, o número de nós ativos é incrementado. Caso se verifique, se o utilizador não usou nenhuma instância de um nó como parâmetro, é retornado uma mensagem de erro, devendo-se ao facto de não existirem mais nós possíveis de ativar. Caso o utilizador passe uma instância, essa instância é inserida no vetor de nós e o número de nós ativos é incrementado para incluir a instância dada pelo utilizador. Os métodos *increase_workers* e *decrease_workers* no *listing* 4.2, mostram a implementação do incremento e decremento de nós, respetivamente.

A classe *ff_a_farm* tem como objetivo substituir a classe *ff_farm*, inicializando uma *ff_farm* com a classe *a_scheduler* para a distribuição de tarefas. Esta classe é responsável

por chamar os métodos da classe *a_scheduler* para incrementar e decrementar o número de nós ativos.

Listing 4.2: Verificação e atualização dos apontadores em GPU

```
1    void increase_workers(ff_node * w = NULL){
2        if(activeWorkers <= 0 || activeWorkers == getNWorkers()){
3            if(w == NULL) printf("Must provide worker instance\n");
4            else{
5                register_worker(w);
6                activeWorkers = getNWorkers();
7            }
8        }else{
9            activeWorkers++;
10       }
11   }
12
13 void decrease_workers(){
14       if(activeWorkers <= 0) activeWorkers = getNWorkers();
15       if(activeWorkers > 1) {
16           activeWorkers--;
17           printf("getting Worker: %d\n", activeWorkers);
18           svector<ff_node*> w = getWorkers();
19           uSWSR_Ptr_Buffer *buf = w[activeWorkers]->get_in_buffer();
20           while(!buf->empty()){
21               void * task;
22               buf->pop(&task);
23               schedule_task(task);
24           }
25       } else
26           printf("MINIMUM WORKERS REACHED\n");
27   }
```

4.2.3 Persistência de dados em GPU

De maneira a programar para execução em GPU em FastFlow, o utilizador utiliza um conjunto de *macros* que representam diferentes padrões, para indicar qual o código que pretende executar. Para indicar quais os dados a serem processados, o utilizador faz uma extensão da classe *baseCUDATask* que indica onde estão guardados esses valores, sendo que depois inicializa a *macro* com uma instância da classe criada. A *macro* é executada da mesma maneira que os restantes padrões do FastFlow.

Sempre que é executada uma *macro*, os dados necessários são transferidos para GPU antes da execução. No fim da execução em GPU, são transferidos de volta apenas os dados do utilizador. Isto leva a que em execuções repetidas sejam novamente transferidos dados que já se encontram GPU, piorando significamente o desempenho. Assim foi acrescentada à classe *stencilReduceCUDA* mecanismos que permitem, em conjunto com a classe *baseCUDAtask*, reutilizar os dados já presentes em GPU, tanto de execuções anteriores da mesma *macro* como de *macros* diferentes.

O mecanismo assenta numa série de verificações antes das transferências de dados, onde se avalia se, na Task definida pelo utilizador, já estão definidos apontadores para posições de memória do GPU. Caso isto se verifique, são atualizados os apontadores guardados para serem usados pela *macro* por aqueles enviados pela *Task*, bem como é alterado o tamanho dos dados para cada apontador. Para além disso, também foram postas condições de maneira a que não seja possível haver transferência de dados, no caso da *Task* conter apontadores para GPU.

O código em 4.3 mostra como é feita a verificação da existência de apontadores em GPU a partir da *Task* definida pelo utilizador, e como são definidos os dados a reutilizar. Este exemplo apenas mostra os apontadores para os dados de entrada e o primeiro apontador para os dados auxiliares, no entanto este código é repetido para os restantes apontadores existentes.

Listing 4.3: Verificação e atualização dos apontadores em GPU

```
1       if(Task.getInDevicePtr() != 0){
2                   in_buffer = Task.getInDevicePtr();
3                   oldSize_in = Task.getBytesizeIn();
4               }
5
6               if(Task.getEnv1DevicePtr() != 0){
7                   env1_buffer = Task.getEnv1DevicePtr();
8                   oldSize_env1 = Task.getBytesizeEnv1();
9               }
```

De maneira a saber onde se encontram os dados a serem usados, a *macro* faz uso da função *setTask* definida pelo utilizador na classe *baseCUDATask*. Nesta função são indicados onde se encontram os dados de entrada, os dados auxiliares e onde se escrevem os dados de saída. Esta função recebe como parâmetro uma instância da classe *baseCUDATask*, definida pelo utilizador, com os dados inicializados. Para que a *macro* saiba onde verificar a existência de apontadores em GPU, é necessário definir nesta função onde se encontram os apontadores em GPU, usando para tal a instância da classe passada como parâmetro.

O código em 4.4 mostra como é feita a definição dos apontadores GPU a serem usados pela *macro*, através da instância da classe *baseCUDATask* referida neste exemplo como *t*.

Listing 4.4: Verificação e definição dos apontadores a serem usados pela *macro*

```
1       cudaTask *t_ = (cudaTask *) t;
2
3       if (t_->getInDevicePtr() != 0){
4         setInDevicePtr(t_->getInDevicePtr());
5       }
6
7       if(t_->getOutDevicePtr() != 0){
8         setOutDevicePtr(t_->getOutDevicePtr());
9       }
10
11      if(t_->getEnv1DevicePtr() != 0){
```

```
12          setEnv1DevicePtr(t_->getEnv1DevicePtr());
13        }
14        if(t_->getEnv2DevicePtr() != 0){
15          setEnv2DevicePtr(t_->getEnv2DevicePtr());
16        }
```

Caso seja a primeira execução da *macro*, é necessário definir na instância da classe *baseCUDATask*, que será reutilizada para execuções posteriores, onde se encontram os apontadores para os dados em GPU. O código em 4.5 mostra como tal é feita na função *endMR*, que executa na depois da *macro*.

Listing 4.5: Definição na *task* onde se encontram os dados em GPU

```
1  void endMR(void*) {
2          t->setInDevicePtr(getInDevicePtr());
3          t->setOutDevicePtr(getOutDevicePtr());
4          t->setEnv1DevicePtr(getEnv1DevicePtr());
5          t->setEnv2DevicePtr(getEnv2DevicePtr());
6          t->setEnv3DevicePtr(getEnv3DevicePtr());
7          t->setEnv4DevicePtr(getEnv4DevicePtr());
8          t->setEnv5DevicePtr(getEnv5DevicePtr());
9          t->setEnv6DevicePtr(getEnv6DevicePtr());
10 }
```

4.2.3.1 Persistência de dados entre fases do algoritmo *Object Identifier*

A implementação do algoritmo *Object Identifier* em GPU usando o FastFlow faz uso de três *macros* diferentes, uma para cada fase do algoritmo. De maneira a garantirmos a reutilização de dados já presentes em GPU pelo algoritmo, precisamos de usar apenas uma instância da classe *baseCUDATask* entre as diferentes fases. Isto significa que precisamos de utilizar uma extensão da classe *baseCUDATask* para as três fases. Daí que na função *endMR* da classe, temos que fazer uma distinção entre o que ocorre em cada fase.

O código em 4.6 mostra o que é necessário fazer no fim de cada fase do algoritmo. No fim da primeira fase, apenas precisamos de definir na instância da *task* onde se encontram os dados em GPU. No fim da segunda fase, é necessário copiar o grafo para CPU de maneira a que seja possível aplicar a otimização necessária, e é preciso definir como não existente o apontador para o grafo em GPU, de maneira a que, na fase seguinte, este seja transferido novamente com as alterações feitas. Na última fase, definida como *default*, apenas é preciso transferir a imagem de GPU para CPU.

Listing 4.6: Definição da função *endMR* no *Object Identifier*

```
1  void endMR(void*) {
2      switch(phase){
3        case 1:
4          t->setInDevicePtr(getInDevicePtr());
5          //Aqui acontece o mesmo que no c digo anterior
6          t->setEnv6DevicePtr(getEnv6DevicePtr());
```

```
7           break;
8         case 2:
9           cudaMemcpy(graph, getEnv3DevicePtr(), size*sizeof(int),
10                cudaMemcpyDeviceToHost);
11          t->setEnv3DevicePtr(0);
12          break;
13        default:
14          cudaMemcpy(img, getEnv1DevicePtr(), size*sizeof(int),
15                cudaMemcpyDeviceToHost);
16      }
17  }
```

4.3 Conclusão

Neste capítulo foi descrito a arquitetura e implementação das nossas soluções, tanto para as propriedades adaptativas, como para a persistência e reutilização de dados do GPU no FastFlow. O próximo capítulo vai validar e avaliar as soluções desenvolvidas.

CAPÍTULO 5

AVALIAÇÃO

Neste capítulo é feita uma análise do desempenho do algoritmo *Object Identifier* implementado em FastFlow com recurso aos padrões com propriedades adaptativas, comparativamente à implementação do algoritmo usando os padrões base do FastFlow.

É feita ainda uma validação das propriedades desenvolvidas nesses padrões, i.e. alteração do número de nós e alteração de *backend* entre CPU e GPU em tempo de execução. Para isso são usados quer testes simples quer a implementação do algoritmo *Object Identifier* que faz uso do *skeleton* desenvolvido,e que demonstram estas propriedades em funcionamento.

É ainda feita uma avaliação do uso de componentes de monitorização e de adaptação autonómica, que fazem uso de propriedades implementadas para o seu funcionamento. Neste caso a única adaptação autonómica testada foi a de redistribuição de *tasks* entre nós que correm o algoritmo *Object Identifier*, na versão em CPU. Estes componentes foram implementados pelo Bruno Preto no contexto do seu trabalho de doutoramento, no qual o meu trabalho se insere.

Por fim, é feita uma análise do desempenho obtido através da reutilização e persistência de dados na memória do GPU, sendo feita uma comparação usando o algoritmo *Object Identifier* onde apenas é feita uma transferência de dados inicial para GPU, sendo esses dados reutilizados por todas as fases do algoritmo, com uma versão onde é necessário transferir dados no inicio de cada fase do algoritmo.

5.1 Condições de teste

Todas as versões do algoritmo foram executadas na mesma máquina e com o mesmo número de *threads* que apresenta o melhor paralelismo possível, tendo sido usadas 4 *threads* nas execuções. A máquina usada na execução tem quatro cores, 12 GBytes de RAM e uma placa gráfica nVidia c2050, com 448 *cores* CUDA e 3 GBytes de memória. As versões

de *software* usadas são as seguintes: GNU/Linux Ubuntu 12.04.5 kernel 2320-99, CUDA 6.5 e FastFlow 2.0.4.

Todos os programas foram executados várias vezes usando imagens cúbicas, que apresentam valores iguais de altura, largura e profundidade, tendo sido obtidos os tempos de cada execução e calculado o tempo médio de execução para cada imagem e para cada programa e sua variação. Para testes foram usadas imagens reais, como 5.1 de tamanho 100, 200 e 400, bem como imagens geradas, como 5.2, tendo sido principalmente usadas imagens onde foi gerada uma esfera em imagens de tamanho 200, 400, 800 e 1024, e onde o raio da esfera variou entre 50, 100, 200, 400 e 500, sendo que o raio máximo da esfera dependeu do tamanho da imagem a ser testada.

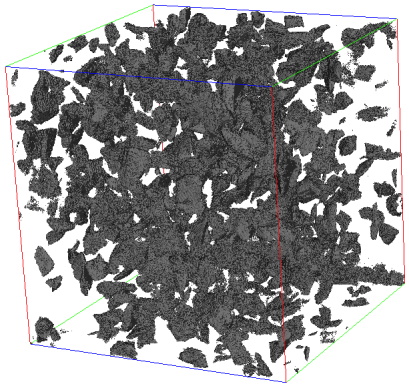

Figura 5.1: Exemplo de figura real usada em testes. Retirado de [40]

Figura 5.2: Exemplo de figura gerada usada em testes. Retirado de [40]

5.2 Validação das propriedades adaptativas

Nesta secção é feita a validação das várias propriedades adaptativas implementadas, sendo elas a alteração do número de nós e a alteração de execução de CPU para GPU. Estas validações são feitas com recurso a testes simples.

5.2.1 Alteração do número de nós

De maneira a validar a alteração do número de nós foi criado um programa simples usando uma classe *ff_a_farm* e foram implementado os nós com recurso à classe *ff_a_node*. Neste programa de teste, a única função do nó é contar o número de tarefas recebidas e

indicar esse valor no fim da execução da *farm*, bem como o seu *id*. Sempre que foi feita uma nova execução, o número de tarefas recebido foi reposto a zero.

O programa consistiu em inicializar uma *ff_a_farm* e executar com 100 tarefas a serem divididas pelos vários nós. Depois dessa execução, o número de nós foi reduzido para apenas um e foi feita uma nova execução, sendo depois incrementado o número de nós e feita uma nova execução. O código desse programa de exemplo encontra-se em 5.1.

Listing 5.1: Programa para teste da alteração do número de nós

```
class Worker: public ff_a_node{
  void * svc_CPU(void * task){
   ntask++;}

  void svc_end(){
   printf("WORKER %d Received %d tasks, get_my_id(); ntask);}
}

int main(){
  ff_a_farm farm;

  vector<ff_node *> workers;
  for(int i = 0; i < 4; i++)
    workers.push_back(new Worker);
  farm.add_workers(workers);

  farm.run();
  while(nworkers > 1)
    farm.decrease_workers();

  farm.run();
  farm.increase_workers();
  farm.run();
}
```

Listing 5.2: Resultado do programa de teste de alteração do número de nós

```
Initial number of Workers: 4
WORKER 1 Received 25 tasks
WORKER 0 Received 25 tasks
WORKER 3 Received 25 tasks
WORKER 2 Received 25 tasks

Remove Workers
Removing Worker: 3
Removing Worker: 2
Removing Worker: 1
WORKER 0 Received 100 tasks

Add worker
WORKER 0 Received 52 tasks
```

```
15 WORKER 1 Received 48 tasks
```

O resultado do programa de teste está indicado em 5.2. Ao inicio estão presentes quatro nós na *farm*, daí o número de tarefas se distribuir igualmente pelos vários nós. Depois de removidos os nós suficientes para ficarmos apenas com um nó ativo, ao executar novamente a *farm* apenas o nó com *id* 0 recebe tarefas, recebendo todas as tarefas possíveis. Quando mais tarde é adicionado mais um nó ativo e executada a *farm*, o número de tarefas é novamente distribuído pelo número de nós ativos.

5.2.2 Alteração de *backend* do nó

Para testar a alteração de *backend* de um nó em tempo de execução, foi criado um programa simples que usa apenas um nó. Na implementação do código em CPU e em GPU, está apenas um simples *print* que indica o *backend* em que o nó está a executar, bem como o número de tarefas já recebidas. De maneira a mudar o *backend* do nó, foi feita uma verificação para quando este receber um certo número de tarefas, em concreto metade do número total de tarefas, o nó chama uma função para alterar o seu *backend* de CPU para GPU. O código do nó usado no programa de teste está descrito em 5.3.

Listing 5.3: Programa de para teste de alteração de *backend*

```
1  class Worker: public ff_a_node{
2    void * svc_CPU(void * task){
3      printf("Worker %d: Received task %d/%d in CPU, get_my_id(), ntask, totaltasks);
4
5      ntask++;
6      if(ntask >= totaltask/2)
7        setEnv(GPU);
8    }
9
10   void * svc_GPU(void * task){
11     printf("Worker %d: Received task %d/%d in GPU, get_my_id(), ntask, totaltasks);
12   ntask++;
13    }
14 }
```

Listing 5.4: Excerto do resultado do programa de teste

```
1 Worker 0: Received task 23/50 in CPU
2 Worker 0: Received task 24/50 in CPU
3 changing to GPU worker 0
4 Worker 0: Received task 25/50 in GPU
5 Worker 0: Received task 26/50 in GPU
```

Um excerto do resultado do programa de teste encontra-se em 5.4. Nesta execução foram enviada 50 tarefas para o nó, sendo que o nó alterou o seu *backend* ao receber a tarefa 25, demonstrado tanto pela linha que indica que está a ser feita uma alteração ao nó, como pelas linhas seguintes que apresentam uma mensagem diferente terminando a sua execução em GPU ao invés de CPU.

Tabela 5.1: Tempos médios de execução (ms) com e sem adaptação autonómica usando imagens reais

	cubo 100	cubo 200	cubo 400
sem adaptação	10,468	67,0753	476,973
com adaptação	11,113	69,451	487,833
Speedup	0,941	0,965	0,977

Tabela 5.2: Tempos médios de execução (ms) com e sem adaptação autonómica usando imagens geradas

	raio esfera	s/ adaptação	c/adaptação	Speedup
Cubo 100	50	37,836	42,945	0,881
Cubo 200	50	53,922	57,754	0,933
	100	293,303	293,166	1,0004
Cubo 400	50	177,923	205,051	0,867
	100	437,583	449,683	0,973
	200	2328,821	2331,410	0,998
Cubo 800	50	1324,710	1353,660	0,979
	100	1609,148	1596,679	1,007
	200	3553,210	3580,119	0,992
	400	19502,21	19552,553	0,997
Cubo 1024	50	2768,637	2787,431	0,993
	100	3015,760	3019,091	0,998
	200	5143,173	5118,136	1,005
	400	21639,352	21665,296	0,998
	500	39636,655	39519,663	1,002

5.3 Avaliação da monitorização e adaptação autonómica

Nesta secção é feita uma avaliação do sistema de monitorização e adaptação autonómica implementado, que faz uso das propriedades desenvolvidas. Como de momento se trata de um sistema rudimentar, a única adaptação possível foi redistribuição de *tasks* entre nós. Estes testes foram feitos usando o algoritmo *Object Identifier* a correr em CPU, usando imagens reais e imagens geradas com esferas de tamanhos diferentes.As tabelas 5.1 e 5.2 mostram uma comparação da execução usando adaptação autonómica e não usando, usando imagens reais e geradas respetivamente.

Usando as imagens reais podemos ver que se obtém piores resultados usando adaptação autonómica do que não usando, no entanto à medida que o tamanho da imagem aumenta a diferença de tempos vai reduzindo. O mesmo pode ser visto nas imagens geradas. À medida que se aumenta o tamanho da imagem o *speedup* obtido aproxima-se de 1 e por vezes é superior. Isto deve-se ao facto de que o tempo de execução de imagens mais pequenas é inferior, o que leva a que não hajam tantas oportunidades de ser feita uma adaptação que traga resultados positivos, fazendo com que o custo de monitorização e adaptação tenha maior influência no tempo de execução do que para imagens de tamanhos superiores.

Tabela 5.3: Tempos médios de execução (ms) com e sem persistência de dados em GPU

	Cubo 100	Cubo 200	Cubo 400
c/ transferencia	32,354	172,201	1245,284
s/ transferencia	24,299	125,457	884,312
Speedup	1,331	1,372	1,408

No entanto é preciso referir de que se trata de um sistema de monitorização e adaptação básico, que se acredita que à medida que este se for tornando mais complexo os tempos observados irão melhorar.

5.4 Avaliação da persistência de dados em GPU

Nesta secção é feita uma comparação dos desempenhos obtidos entre a persistência e reutilização de dados em GPU no FastFlow, e a transferência de dados sempre que é feita que uma execução em GPU usando o FastFlow. Para isso foi usado novamente o algoritmo *Object Identifier*, usando uma versão em que apenas eram transferidos os dados quando necessário e outra em que havia transferência de dados no antes e depois de cada fase do algoritmo. Nestes testes foram usadas as imagens reais usadas nos testes anteriores.A tabela 5.3 mostra os resultados destas execuções.

Como mostram os resultados, obtemos um *speedup* significativo ao garantir a reutilização de dados em GPU. Isto deve-se ao facto de reduzirmos o número de transferências para cerca de um terço do que se não garantirmos reutilização de dados, sendo que estas transferências apresentam um custo significativo.

5.5 Conclusão

Com estes testes podemos concluir que as propriedades adaptativas implementadas funcionam como esperado. Também se pode concluir que enquanto que o sistema de monitorização e adaptação autonómica apresenta resultados negativos para imagens pequenas, para imagens maiores aproxima-se do tempo de execução normal bem como chega a ser melhor algumas vezes. Como se trata de um sistema ainda básico, espera-se que à medida que o se sistema for desenvolvido e melhorado, os tempos obtidos também melhorem e as adaptações apresentem resultados positivos.

Em relação à persistência e reutilização de dados em GPU usando o FastFlow, pode-se concluir que traz uma vantagem significativa em termos de desempenho, sendo que esta melhoria aumenta à medida que se vai aumentando o tamanho da imagem, e com isso o tamanho dos dados que são necessários transferir para GPU.

CAPÍTULO 6

CONCLUSÃO E TRABALHO FUTURO

Este capítulo apresenta algumas conclusões em relação ao trabalho desenvolvido nesta tese, sendo apresentadas algumas propostas de trabalho futuro tornadas possíveis como resultado da solução proposta.

6.1 Balanço do trabalho

O objectivo do trabalho foi contribuir para o objectivo mais lato de construir uma arquitectura de suporte a propriedades autonómicas no contexto do processamento de imagem, e que possam tirar partido das características da imagem em processamento, quer isoladamente, quer no contexto de um *stream* de imagens. Para mais, o contexto de programação que se pretende oferecer visa utilizadores não peritos em computação de alto desempenho, e que previligiem um desenvolvimento de aplicações mais simples e célere, em detrimento da necessidade de se obter o melhor desempenho possível para uma arquitectura alvo particular. As ferramentas com base em *algorithmic skeletons* pretendem oferecer essa maior facilidade de desenvolvimento de aplicações paralelas sem contudo comprometer o seu desempenho. Este objectivo geral é a base de uma tese de doutoramento em desenvolvimento.

Como tal, o trabalho apresentado nesta tese, foi exploratório na sua essência e contribuiu para estudar, em detalhe, quer uma ferramenta particular que suporta *algorithmic skeletons*, quer o impacto da introdução de propriedades adaptativas nessa ferramenta e no contexto de um *skeleton* particular. O trabalho apresentado nesta tese contribuiu também para a definição da arquitectura de base que permitirá a reconfiguração dinâmica do processamento de imagens em diferentes contextos, bem como a sua instanciação num *framework* baseado em *skeletons* particular. Dado o seu carácter exploratório, foi assim necessário definir os componentes básicos necessários, e.g. de monitoração e escalonamento de tarefas, bem como as alterações ao *framework* para a execução quer em CPU quer em

GPU. Foi ainda relevante avaliar o custo da alteração do *framework* escolhido em termos de desempenho e que tipo de estratégias adaptativas, específicas para uma certa classe de imagens, vale ou não a pena explorar.

Assim, o objetivo deste trabalho insere-se no objetivo geral de simplificar a programação de aplicações na área de processamento de imagem, sem comprometer o seu desempenho. Tal simplificação passa por usar programação paralela estruturada com base em padrões paralelos e *algorithmic skeletons*. Estes capturam estratégias recorrentes de paralelização independentes da arquitetura alvo, garantindo a geração automática de código otimizado para diferentes arquiteturas. Dado que estas soluções genéricas apresentam sempre um desempenho menos satisfatório que soluções dedicadas, torna-se necessário desenvolver estratégias que permitam a otimização dessas estratégias baseadas em *skeletons*. Uma das formas é, tirando partido das características comuns a classes de aplicações, procurar melhorar o desempenho da execução de *skeletons* através da sua adaptação em tempo de execução, e de acordo com essas características das aplicações.

Tendo em vista contribuir para este objetivo geral, esta tese teve como objetivo estender um *framework* de *skeletons* com geração de código para arquiteturas heterogéneas, com a implementação de um *skeleton* com propriedades adaptativas. Estas permitem tirar partido da carga do sistema, bem como escolher qual o *backend* de execução.

Este trabalho teve também como objetivo contribuir para o objetivo geral de capturar características comuns a algoritmos de segmentação para processamento de imagem. Isto por forma a definir uma arquitetura comum a estes algoritmos, bem como mapear essas características em padrões paralelos básicos. Para tal foi estudado um algoritmo de segmentação denominado *Object Identifier*, e foi realizada a sua implementação usando o *framework* estendido. Esta implementação serviu para validação das estratégias de reconfiguração dinâmica que foram implementadas.

Em relação ao *skeleton* com propriedades adaptativas, foram atingidos os nossos objetivos, tendo sido implementado um *skeleton farm* que permite alterar o número de nós, bom como alterar o *backend* em que cada nó corre. Para além disso, foi implementado um sistema de distribuição de tarefas com base nos estados de cada nó, escolhendo o nó em melhores condições. Esta solução permite que o utilizador não necessite de criar dois programas em FastFlow para poder implementar uma solução para CPU e GPU. Permite ainda que o utilizador implemente as suas próprias condições para distribuição de tarefas por nós. No entanto, esta solução obriga o utilizador a implementar as estratégias de medição e a definir os estados usados na distribuição de tarefas, que não necessita de fazer ao usar os *skeletons* base do FastFlow. Esta solução apresenta um desempenho pior quando comparado com as implementações base do FastFlow, devendo-se isto à necessidade de fazer uma medição do nó depois deste processar uma tarefa, e à necessidade de comparar os estados dos vários nós quando é feita a distribuição das tarefas.

Em relação ao uso de monitorização e adaptação autonómica, esta foi testada usando uma das propriedades adaptativas implementadas. No entanto, como se trata de um sistema que ainda é básico, não foram obtidos resultados melhores significativos quando

comparado com a não utilização do sistema. No entanto, o desempenho obtido usando o sistema foi-se aproximando do obtido ao não utilizar o sistema quando o tamanho das imagens eram maiores. Isto leva a crer que à medida que o sistema for desenvolvido estes resultados obtidos vão melhorar, chegando ao ponto em que se obtém melhor desempenho usando o sistema do que não usando, sendo esses resultados obtidos em imagens com dimensões cada vez mais reduzidas.

No que se refere à persistência e reutilização de dados em GPU usando o FastFlow, os resultados foram positivos, tendo sido obtido um *speedup* significativo quando isto acontece. Isto deve-se à necessidade de fazer menos transferências de dados, transferências essas que tem um peso significativo no desempenho obtido.

6.2 Trabalho futuro

Com base no trabalho desenvolvido e nas limitações apontadas na secção anterior, algumas possibilidades de trabalho futuro são a adaptação de outros *skeletons* do FastFlow de maneira a incluir propriedades adaptativas, por exemplo o *skeleton pipeline*. Outra possibilidade de trabalho consiste em otimizar a medição feita pelos nós de maneira a que o seu desempenho se aproxime do desempenho obtido usando os nós base do FastFlow. Outra possibilidade ainda, é implementar o algoritmo *object identifier* com recurso a um *pipeline*, de maneira a ser possível diferentes blocos da mesma imagem serem processados ao mesmo tempo mas em fases diferentes do algoritmo.

Um trabalho interessante a explorar é a utilização o *skeleton* desenvolvido para suportar o cumprimento de qualidades de serviço. Por exemplo, garantir que o *skeleton* não ultrapassa um tempo máximo de execução ou um limite para os recursos disponíveis, ambos definidos pelo utilizador, sendo aplicadas as propriedades adaptativas sempre que se detete não ser possível cumprir, nas condições atuais, a qualidade de serviço proposta. De maneira a cumprir essas qualidades de serviço, o *skeleton* pode inicializar mais nós ou alterar o *backend* dos nós atuais.

BIBLIOGRAFIA

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick e M. Torquati. “FastFlow: high-level and efficient streaming on multi-core.(A FastFlow short tutorial)”. Em: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2011).

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin e M. Torquati. “An efficient unbounded lock-free queue for multi-core systems”. Em: *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 662–673.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick e M. Torquati. “Targeting heterogeneous architectures via macro data flow”. Em: *Parallel Processing Letters* 22.02 (2012), p. 1240006.

[4] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund et al. “The Fortress language specification”. Em: *Sun Microsystems* 139 (2005), p. 140.

[5] G. M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. Em: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[6] M. Arango e B. Kaponig. “Ultra-scalable architectures for telecommunications and Web 2.0 services”. Em: *Intelligence in Next Generation Networks, 2009. ICIN 2009. 13th International Conference on*. IEEE. 2009, pp. 1–7.

[7] B. Barney et al. “Introduction to parallel computing”. Em: *Lawrence Livermore National Laboratory* 6.13 (2010), p. 10.

[8] G. H. Botorog e H. Kuchen. “Skil: An imperative language with algorithmic skeletons for efficient distributed programming”. Em: *High Performance Distributed Computing, 1996., Proceedings of 5th IEEE International Symposium on*. IEEE. 1996, pp. 243–252.

[9] W. Burger e M. J. Burge. *Digital image processing: an algorithmic introduction using Java*. Springer Science & Business Media, 2009.

[10] D. Buttlar e J. Farrell. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. "O’Reilly Media, Inc.", 1996.

[11] B. L. Chamberlain, D. Callahan e H. P. Zima. "Parallel programmability and the chapel language". Em: *International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312.

[12] M. Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". Em: *Parallel computing* 30.3 (2004), pp. 389–406.

[13] M. I. Cole. "Algorithmic skeletons: A structured approach to the management of parallel computation". Tese de doutoramento. University of Edinburgh, 1988.

[14] L. Dagum e R. Menon. "OpenMP: an industry standard API for shared-memory programming". Em: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[15] M. Danelutto e M. Stigliani. "SKElib: parallel programming with skeletons in C". Em: *Euro-Par 2000 Parallel Processing*. Springer. 2000, pp. 1175–1184.

[16] M. Danelutto e M. Torquati. "Loop parallelism: a new skeleton perspective on data parallel patterns". Em: *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. IEEE. 2014, pp. 52–59.

[17] J. Darlington, Y.-k. Guo, H. W. To e J. Yang. "Parallel skeletons for structured composition". Em: *ACM SIGPLAN Notices*. Vol. 30. 8. ACM. 1995, pp. 19–28.

[18] U. Dastgeer e C. Kessler. "Smart containers and skeleton programming for GPU-based systems". Em: *International Journal of Parallel Programming* (2014), pp. 1–25.

[19] U. Dastgeer, L. Li e C. Kessler. "Adaptive implementation selection in the skepu skeleton programming library". Em: *Advanced Parallel Processing Technologies*. Springer, 2013, pp. 170–183.

[20] A. P. Dhawan, G. Buelloni e R. Gordon. "Enhancement of mammographic features by optimal adaptive neighborhood image processing". Em: *Medical Imaging, IEEE Transactions on* 5.1 (1986), pp. 8–15.

[21] J. Enmyren e C. W. Kessler. "SkePU: a multi-backend skeleton programming library for multi-GPU systems". Em: *Proceedings of the fourth international workshop on High-level parallel programming and applications*. ACM. 2010, pp. 5–14.

[22] I. Foster. *Designing and building parallel programs*. 1995.

[23] I. Foster e N. T. Karonis. "A grid-enabled MPI: Message passing in heterogeneous distributed computing systems". Em: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 1998, pp. 1–11.

[24] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation". Em: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2004, pp. 97–104.

[25] E. Gamma, R. Helm, R. Johnson e J. Vlissides. "Design patterns: Abstraction and reuse of object-oriented design". Em: *European Conference on Object-Oriented Programming*. Springer. 1993, pp. 406–431.

[26] H. González-Vélez e M. Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers". Em: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160.

[27] C. Grelck. "Shared memory multiprocessor support for functional array processing in SAC". Em: *Journal of Functional Programming* 15.03 (2005), pp. 353–401.

[28] J. L. Gustafson. "Reevaluating Amdahl's law". Em: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[29] A. Kaminsky. *BIG CPU, BIG DATA: Solving the World's Toughest Computational Problems with Parallel Computing*. 2015.

[30] D. B. Kirk e W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[31] G. Kyriazis. "Heterogeneous system architecture: A technical review". Em: *AMD Fusion Developer Summit* (2012).

[32] M. Leyton e J. M. Piquer. "Skandium: Multi-core programming with algorithmic skeletons". Em: *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE. 2010, pp. 289–296.

[33] R. Loogen, Y. Ortega-Mallén e R. Peña-Marí. "Parallel functional programming in Eden". Em: *Journal of Functional Programming* 15.03 (2005), pp. 431–475.

[34] F. D. Macías-Escrivá, R. Haber, R. del Toro e V. Hernandez. "Self-adaptive systems: A survey of current approaches, research challenges and applications". Em: *Expert Systems with Applications* 40.18 (2013), pp. 7267–7279.

[35] V. Mannava e T Ramesh. "A novel adaptive re-configuration compliance design pattern for autonomic computing systems". Em: *Procedia Engineering* 30 (2012), pp. 1129–1137.

[36] T. G. Mattson, B. Sanders e B. Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[37] M. McCool, J. Reinders e A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[38] A. Munshi et al. "The opencl specification". Em: *Khronos OpenCL Working Group* 1 (2009), pp. l1–15.

[39] C. Nvidia. "Compute unified device architecture programming guide". Em: (2007).

[40] B. Preto, F. Birra, A. Lopes e P. Medeiros. "Object Identification in Binary Tomographic Images Using GPGPUs". Em: *International Journal of Creative Interfaces and Computer Graphics (IJCICG)* 4.2 (2013), pp. 40–56.

[41] B. A. M. Preto. "Utilização de GPGPUs para a identificação de objectos em imagens tomográficas". Em: (2011).

[42] J. Sérot e D. Ginhac. “Skeletons for parallel image processing: an overview of the skipper project”. Em: *Parallel computing* 28.12 (2002), pp. 1685–1708.

[43] M. Snir. *MPI–the Complete Reference: The MPI core*. Vol. 1. 1998.

[44] H. Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. Em: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[45] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese e K. M. Göschka. “On patterns for decentralized control in self-adaptive systems”. Em: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 76–107.

[46] M. Yuffe, E. Knoll, M. Mehalel, J. Shor e T. Kurts. “A fully integrated multi-CPU, GPU and memory controller 32nm processor”. Em: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*. IEEE. 2011, pp. 264–266.