Pedro Miguel Ferreira Somsen

Licenciatura em Engenharia Informática

Programação Paralela Estruturada Aplicada a um Problema de Reconstrução de Imagem

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Co-orientadores: Pedro D. Medeiros, Professor Associado, FCT/UNL
Maria Cecília Gomes, Professora Auxiliar, FCT/UNL

Júri:

Presidente: Prof. Doutor Carlos Viegas Damásio

Vogais: Prof. Doutor André Mora
Prof. Doutor Pedro Medeiros

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Janeiro, 2016

Programação Paralela Estruturada Aplicada a um Problema de Reconstrução de Imagem

Copyright © Pedro Miguel Ferreira Somsen, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Resumo

A Digital Breast Tomosynthesis (DBT) é uma técnica que permite obter imagens mamárias 3D de alta qualidade, que só podem ser obtidas através de métodos de reconstrução. Os métodos de reconstrução mais rápidos são os iterativos, sendo no entanto computacionalmente exigentes, necessitando de sofrer muitas optimizações. Existem optimizações que usam computação paralela através da implementação em GPUs usando CUDA. Como é sabido, o desenvolvimento de programas eficientes que usam GPUs é ainda uma tarefa demorada, dado que os modelos de programação disponíveis são de baixo nível, e a portabilidade do código para outras arquitecturas não é imediata. É uma mais valia poder criar programas paralelos de forma rápida, com possibilidade de serem usados em diferentes arquitecturas, sem exigir muitos conhecimentos sobre a arquitectura subjacente e sobre os modelos de programação de baixo nível.

Para resolver este problema, propomos a utilização de soluções existentes que reduzam o esforço de paralelização, permitindo a sua portabilidade, garantindo ao mesmo tempo um desempenho aceitável. Para tal, vamos utilizar um framework (FastFlow) com suporte para Algorithmic Skeletons, que tiram partido da programação paralela estruturada, capturando esquemas/padrões recorrentes que são comuns na programação paralela.

O trabalho realizado centrou-se na paralelização de uma das fases de reconstrução da imagem 3D – geração da matriz de sistema – que é uma das mais demoradas do processo de reconstrução; esse trabalho incluiu um método de ordenação modificado em relação ao existente. Foram realizadas diferentes implementações em CPU e GPU (usando OpenMP, CUDA e FastFlow) o que permitiu comparar estes ambientes de programação em termos de facilidade de desenvolvimento e eficiência da solução.

A comparação feita permite concluir que o desempenho das soluções baseadas no FastFlow não é muito diferente das tradicionais o que sugere que ferramentas deste tipo podem simplificar e agilizar a implementação de um algoritmos na área de reconstrução de imagens 3D, mantendo um bom desempenho.

Palavras-chave: Reconstrução Imagem 3D, Processamento Paralelo em GPU, Algorithmic Skeleton Frameworks

---

Abstract

Digital Breast Tomosynthesis (DBT) is a 3D radiographic technique that allows one to acquire high quality 3D images of breast tissue. These images are only achieved through image reconstruction methods. The fastest reconstruction methods are the ones that use iterative techniques, but are computationally intensive. Their execution time must be reduced in order to allow its use in clinical settings. Some optimizations use parallel computing, namely though the offload of computations to Graphical Processing Units (GPUs); these parts of the reconstruction process are developed using CUDA, achieving reduction of the execution time. It is widely known that developing these kinds of programs using GPUs can be time-consuming. This happens because its development requires the use of low level programming models, and the portability to other architectures isn't straightforward. Being able to develop fast parallel programs that can be used in different architectures without the requirement of deep knowledge on the underlying architecture and low level programming models is really appealing.

To solve this problem, we propose the use of existing solutions that lower the effort in parallelizing a problem, allowing portability and at the same time ensuring an acceptable performance. We will use a framework (FastFlow) that supports Algorithmic Skeletons that take advantage of structured parallel programming, capturing recurring patterns that are common in parallel programming.

The dissertation work focused on the parallelization of one of the most time consuming steps of the image reconstruction process, the generation of the system matrix; this effort included the changing of the sorting algorithm present in the base implementation used. Several implementations targeting CPUs and GPUs were developed – using OpenMP, CUDA and FastFlow – allowing the comparison of these programming frameworks in turns of efficiency and ease of development.

The comparison made allow us to conclude that the performance of FastFlow-based solutions is close to the ones achieved by more traditional parallel programming environments. This suggests that the use of frameworks supporting algorithmic skeletons allow easier development of 3D image reconstruction implementations without significant performance overheads.

Keywords: 3D Image Reconstruction, GPU Parallel Processing, Algorithmic Skeletons Frameworks

---

Conteúdo

1. INTRODUÇÃO ... 1

1.1. MOTIVAÇÃO ... 1

1.2. OBJECTIVOS DA TESE ... 3

1.3. SOLUÇÃO PROPOSTA ... 4

1.4. CONTRIBUIÇÃO DA TESE ... 6

1.5. ESTRUTURA DO DOCUMENTO ... 6

2. TRABALHO RELACIONADO ... 9

2.1. PROCESSAMENTO E SÍNTESE DE IMAGEM ... 9

2.1.1. Processamento ponto a ponto ... 10

2.1.2. Convolução entre máscara e imagem ... 10

2.1.3. Segmentação de uma imagem ... 11

2.1.4. Image Registration (Alinhamento de Imagens) ... 11

2.2. PROCESSAMENTO PARALELO EM PROCESSAMENTO DE IMAGEM ... 11

2.2.1. Programação paralela e suas dificuldades/desafios ... 12

2.2.2. Eficiência da paralelização ... 14

2.2.3. Dificuldades ... 17

2.2.4. OpenMP ... 18

2.2.5. CUDA ... 19

2.3. PROGRAMAÇÃO PARALELA ESTRUTURADA COMO TÉCNICA GERAL PARA O PROCESSAMENTO PARALELO ... 21

2.3.1. Design Patterns ... 21

2.3.2. Algorithmic Skeletons ... 24

2.3.3. Algorithmic Skeletons Frameworks ... 27

2.3.4. FastFlow ... 28

2.4. EXEMPLOS DE PROCESSAMENTO PARALELO DE IMAGENS 33
2.4.1. Paralelização do algoritmo de reconstrução LM OSEM 33
2.4.2. Reconhecimento facial usando um ambiente paralelo baseado em Algorithmic skeletons 36
2.4.3. Processamento ponto a ponto de imagens digitais usando computação paralela 37
2.4.4. Experiência efectuada (segmentação) 38

3. ALGORITMOS ITERATIVOS DE RECONSTRUÇÃO DE IMAGEM 43
3.1. RECONSTRUÇÃO DE IMAGEM 43
3.1.1. Reconstrução Iterativa 45
3.2. DBT SYSTEM 51
3.3. DESCRIÇÃO DA IMPLEMENTAÇÃO DO TRABALHO EXISTENTE 52
3.4. CÁLCULO DA MATRIZ SISTEMA 52

4. DESENHO DA SOLUÇÃO 59
4.1. VERSÃO SEQUENCIAL 59
4.2. OPORTUNIDADES DE MELHORAMENTO 61
4.3. VERSÃO PARALELA USANDO OPENMP 66
4.4. VERSÃO PARALELA USANDO CUDA 66
4.5. FASTFLOW – CPU 68
4.6. FASTFLOW – GPU 69
High-Level Patterns 69

5. IMPLEMENTAÇÃO 71
5.1. SEQUENCIAL 71
Cálculo das intersecções eixo X 72
Cálculo das intersecções do eixo Y 72
Calculo das intersecções eixo Z 73
Ordenação Biblioteca 73
Nova Ordenação (proposta) 74
Cálculo das Distâncias 75
5.2. OPENMP 76
Cálculo das intersecções 77
Cálculo das distâncias 78
5.3. CUDA 79
Kernel cálculo das intersecções 79
Kernel apagar zeros das intersecções 80
Kernel calcular distâncias 81
Kernel apagar zeros das distâncias 82
5.4. FASTFLOW – CPU 82

Cálculo das intersecções ... *82*
Cálculo das distâncias ... *82*
5.5. FASTFLOW – GPU ... 83
5.6. ALTERAÇÕES FRAMEWORK ... 84
6. DISCUSSÃO DOS RESULTADOS ... 89
6.1. HARDWARE E SOFTWARE ... 89
6.2. NOVA ORDENAÇÃO ... 90
6.3. RESULTADOS ALTERAÇÕES FRAMEWORK ... 91
6.4. RESULTADOS CONSTRUÇÃO DA MATRIZ ... 93
6.4.1. Resultados com Ordenação Biblioteca ... 94
6.4.2. Resultados com Nova Ordenação ... 96
6.4.3. Comparação dos Resultados das Diferentes Ordenações ... 97
7. CONCLUSÃO E TRABALHO FUTURO ... 101
7.1. Comparação dos resultados obtidos com os objectivos iniciais ... 101
7.2. Os desenvolvimentos futuros que se poderão basear neste trabalho são: 103
REFERÊNCIAS ... 105
A. SIMPLIFICAÇÃO DO CÁLCULO DAS INTERSECÇÕES ... 109
1. DEMONSTRAÇÃO DA SIMPLIFICAÇÃO ... 109
1.1. Eixo X ... 110
1.2. Eixo Y ... 112
1.3. Eixo Z ... 114
2. COMPARAÇÃO DOS RESULTADOS ... 116

Lista de Figuras

Figura 2.1: Exemplos de convoluções de máscaras com a imagem original 11
Figura 2.2: Dependência dos dados para um algoritmo de processamento de imagem usando um operador em janela. 15
Figura 2.3: Decomposição *fine-grained* para um algoritmo de processamento de imagem usando um operador em janela 15
Figura 2.4: Decomposição coarse-grained para um algoritmo de processamento de imagem usando um operador em janela 16
Figura 2.5: (a) máscara usada na convolução; (b) pixels da imagem original usados na convolução [26] 16
Figura 2.6: Grafo do skeleton pipe [22] 25
Figura 2.7: Estrutura do grafo da *Farm,* segundo a interpretação do Fastflow *emitter* e *colector* são nós especiais que tratam do agendamento das tarefas para os *workers* e de reunir os seus resultados [22] 25
Figura 2.8: O grafo do Map em que cada item é avaliado em paralelo, segundo a interpretação do Fastflow [22]........ 26
Figura 2.9: Padrões principais do FastFlow que operam em stream. Adaptado de [32]. 28
Figura 2.10: Arquitectura da ferramenta FastFlow [32]. 29
Figura 2.11: Exemplos de combinações válidas de pipeline e farm (e também feedback). 31
Figura 2.12: Implementação do algoritmo LM OSEM (PSD) usando MPI e OpenMP........ 34
Figura 2.13: Implementação do algoritmo LM OSEM (PSD) usando *task parallel skeleton*. 34
Figura 2.14: Implementação do algoritmo LM OSEM (ISD) usando *data parallel skeletons*........ 34
Figura 2.15: Tempos de execução da reconstrução de uma imagem PET [17]. 35
Figura 2.16: Código para reconhecimento facial usando skeletons........ 37
Figura 2.17: Processamento de uma imagem (resolução 200x200x200) usando algoritmos de bi-segmentação (a) e histerese (b) 39
Figura 2.18: Histograma da imagem inicial em tons de cinzento........ 39
Figura 2.19: Distribuição da carga da imagem (a) pelos processos (b). 40

FIGURA 3.1: IMAGEM DE UM CORAÇÃO RETIRADA DE UM VÍDEO DE UMA RESSONÂNCIA MAGNÉTICA. A) RECONSTRUÇÃO DIRECTA B) RECONSTRUÇÃO ITERATIVA [10] 45
FIGURA 3.2: ESQUEMA DO CICLO DE EXECUÇÃO DOS ALGORITMOS DE RECONSTRUÇÃO PARA DBT (SART, ML-EM E OS-EM). 46
FIGURA 3.3: DIAGRAMA DO SISTEMA SIEMENS MAMMOMAT INSPIRATION FAZENDO DBT. ADAPTADO DE [1]. ... 51
FIGURA 3.4: EXECUÇÃO DE UM PROGRAMA IDL INVOCANDO CUDA [1]. 52
FIGURA 3.5: EXEMPLO DA ABORDAGEM ORIENTADA AO RAIO NO PLANO XY, EM QUE O "Y" REPRESENTA O COMPRIMENTO DO DETETOR E O "X" A ALTURA [1]. 53
FIGURA 3.6: ILUSTRAÇÃO DE UM RAIO I (RI) DE UMA CÉLULA DO DETETOR PARA O FOCO DO RAIO-X, ILUSTRANDO QUE O RAIO É APENAS ATENUADO PELOS VÓXEIS POR ONDE PASSA (VÓXEIS VERDES)[1]. 54
FIGURA 3.7: POSIÇÃO DO FOCO NOS PLANOS (A) XY E (B) YZ [1]. 55
FIGURA 3.8: ATRIBUIÇÃO DO VOXEL ÀS INTERSECÇÕES DE UM RAIO NO (A) PLANO XY (QUE TEM SEMPRE DECLIVE POSITIVO) E (B) PLANO YZ (QUE NESTE CASO TEM UM DECLIVE NEGATIVE) [1]. 56
FIGURA 4.1: INTERSECÇÕES ORDENADAS PELA COORDENADA X SEGUIDA PELA CÉLULA DO DETETOR. 60
FIGURA 4.2: A) POSIÇÃO DO FOCO NO EIXO XY B) POSIÇÃO DO FOCO NO EIXO XZ [1]. 62
FIGURA 4.3: ALGUNS EXEMPLOS DE DISTRIBUIÇÃO DE DADOS QUE PODEMOS ENCONTRAR. 63
FIGURA 4.4: ORDENAÇÃO DAS INTERSECÇÕES PARA UMA CÉLULA. 64
FIGURA 4.5: ESQUEMA DO PROGRAMA SEQUENCIAL EXECUTADO NUM ÚNICO CORE DO CPU. 65
FIGURA 4.6: ESQUEMA DO PROGRAMA PARALELO BASEADO NO ESQUEMA SEQUENCIAL. 66
FIGURA 4.7: ESQUEMA DO PROGRAMA PARALELO (GPUS) BASEADO NA VERSÃO SEQUENCIAL. 67
FIGURA 4.8: ESQUEMA DO PROGRAMA FASTFLOW (CPU) BASEADO NO ESQUEMA DO OPENMP. 68
FIGURA 5.1: ESTRUTURA DA INTERSECÇÃO. 72
FIGURA 5.2: EXEMPLO DA NOVA ORDENAÇÃO PARA UM VECTOR COM 4 INTERSECÇÕES POR CADA EIXO. 75
FIGURA 5.3: ESTRUTURA DOS ELEMENTOS DA MATRIZ. 76
FIGURA 5.4: ESQUEMA PARA REMOÇÃO DAS INTERSECÇÕES NULAS. 80
FIGURA 6.1: TEMPO MÉDIO DA ORDENAÇÃO USANDO A BIBLIOTECA DO C++ (LINHA AZUL); TEMPO MÉDIO DA ORDENAÇÃO USANDO O NOVO MÉTODO (LINHA MAGENTA). 91
FIGURA 6.2: TEMPO MÉDIO DE CADA MAPA NO CÁLCULO DA MATRIZ. 92
FIGURA 6.3: TEMPO MÉDIO PARA GERAR UMA PARTE DA MATRIZ SISTEMA. 93
FIGURA 6.4: RESULTADO DAS ORDENAÇÕES PARA TODAS AS ESCALAS USANDO O ALGORITMO DE ORDENAÇÃO DISPONIBILIZADO PELA BIBLIOTECA. 94
FIGURA 6.5: RESULTADO DAS ORDENAÇÕES PARA TODAS AS ESCALAS USANDO O NOVO ALGORITMO DE ORDENAÇÃO PROPOSTO. 96
FIGURA 6.6: FIGURA COM DIFERENTES VERSÕES DA GERAÇÃO DA MATRIZ PARA UMA ESCALA IGUAL A 1, INDICANDO O SPEEDUP OBTIDO USANDO O NOVO MÉTODO DE ORDENAÇÃO. 98
FIGURA 6.7: COMPARAÇÃO DOS TEMPOS DE EXECUÇÃO DA VERSÃO FASTFLOW GPU COM E SEM AS ALTERAÇÕES À FERRAMENTA. 98
FIGURA 6.8: COMPARAÇÃO DA FERRAMENTA FASTFLOW COM AS TECNOLOGIAS MAIS USADAS: OPENMP PARA CPU; CUDA PARA GPUS. 99

Lista de Tabelas

TABELA 6.1: CARACTERÍSTICAS DO SISTEMA USADO NESTE TRABALHO. 90
TABELA 6.2: CARACTERÍSTICAS DO GPU USADO NESTE TRABALHO. 90
TABELA A.1: COMPARAÇÃO DAS VERSÕES SEQUENCIAIS. 116
TABELA A.2: COMPARAÇÃO DAS VERSÕES OPENMP. 117
TABELA A.3: COMPARAÇÃO DAS VERSÕES CUDA. 117

Listagens

Listagem 2.1: Exemplo das diferenças entre código sequencial e código paralelo usando OpenMP.....19
Listagem 2.2: Programa CPU.....20
Listagem 2.3: Programa CUDA.....20
Listagem 2.4: Exemplo de uma etapa usando ff_node.....30
Listagem 2.5: Exemplos de geração (myNode1) e absorção (myNode2) de tarefas.....30
Listagem 2.6: Exemplo do uso do padrão pipeline.....31
Listagem 2.7: Exemplo do uso do padrão farm.....32
Listagem 2.8: Comparação entre o código gerado usando FastFlow e usando OpenMP.....32
Listagem 2.9: Exemplo de um parallel for reduce em comparação com o código sequencial.....33
Listagem 2.10: Parto do algoritmo de geração do histograma (etapa 1).....41
Listagem 2.11: Processamento de cada fatia por cada processo (etapa 2).....42
Listagem 2.12: Processamento de cada trabalho por cada processo (etapa 3, primeira fase).....42
Listagem 4.1: Exemplo de um padrão mapa a ser executado para GPU (CUDA).....69
Listagem 5.1: Parte do código para o cálculo das intersecções no eixo X.....72
Listagem 5.2: Parte do código para o cálculo das intersecções no eixo Y.....72
Listagem 5.3: Parte do código para o cálculo das intersecções no eixo Z.....73
Listagem 5.4: Código para a ordenação das intersecções usando a biblioteca do C++.....73
Listagem 5.5: Código responsável por ordenar as intersecções de um detetor e guardar no novo vector final ordenado.....74
Listagem 5.6: Código que calcula as distâncias e o voxel associado à mesma.....76
Listagem 5.7: Código da paralelização do ciclo do cálculo das intersecções.....77
Listagem 5.8: Código da paralelização do ciclo do cálculo das distâncias.....78
Listagem 5.9: Código do kernel responsável por calcular as intersecções.....79
Listagem 5.10: Código do kernel que calcula das distâncias da matriz.....81
Listagem 5.11: Código do kernel que elimina os zeros das distâncias.....82
Listagem 5.12: Código que paraleliza o cálculo das intersecções.....82
Listagem 5.13: Código que paraleliza o cálculo das distâncias.....82
Listagem 5.14:Código que exemplifica o corpo da função kernel CUDA.....83

Listagem 5.15: Métodos adicionados à classe "stencilReduceCUDA" da ferramenta FastFlow. 85
Listagem 5.16: (a) verificação da cópia do device para host; (b) verificação da cópia do host para device. 86
Listagem 5.17: Exemplo de uma task em que as variáveis Env1 e Env2 não são copiadas do Host para o Device. 87
Listagem A.1: Parte do código para o cálculo das intersecções no eixo X. 110
Listagem A.2: 1ª simplificação do código para o cálculo das intersecções no eixo X. 110
Listagem A.3: 2ª simplificação do código para o cálculo das intersecções no eixo X. 111
Listagem A.4: 3ª simplificação do código para o cálculo das intersecções no eixo X. 112
Listagem A.5: Parte do código para o cálculo das intersecções no eixo Y (crescente). 113
Listagem A.6: Simplificação do código para o cálculo das intersecções no eixo Y (crescente). 114
Listagem A.7: Parte do código para o cálculo das intersecções no eixo Z. 115
Listagem A.8: Simplificação do código para o cálculo das intersecções no eixo Z. 116

1

Introdução

1.1. *Motivação*

A Digital Breast Tomosynthesis (DBT) é uma técnica que permite a obtenção de imagens mamárias 3D de alta qualidade, que só podem ser obtidas através de métodos de reconstrução. Estes métodos têm um papel muito importante num ponto de vista clínico, onde é importante conseguir implementar um processo de reconstrução que é preciso e rápido [1]. Há dois grupos de algoritmos de reconstrução, analíticos e iterativos, onde os algoritmos que produzem a melhor qualidade de imagem são os iterativos. O problema é a intensidade de computações necessárias para gerar estas imagens, o que implica que para que estes métodos de reconstrução possam ser utilizados em práticas clínicas, é necessário optimizar o seu desempenho.

Tal pode ser feito através de programação paralela utilizando *Graphics Processing Units* (GPUs) dado que os estes são mais adequados para problemas de paralelização massiva em que há independência nos dados a processar. Assim, eles são bastante utilizados no processamento de imagem em que o processamento de vários pixels pode ser feito de modo independente, possibilitando utilizar simultaneamente as várias threads que compõem o GPU. No entanto, como é sabido, o desenvolvimento de programas eficientes que usam GPUs é ainda uma tarefa demorada, dado que os modelos de programação disponíveis são de baixo nível, exigindo grandes conhecimentos sobre as configurações *hardware* usadas. Tal inclui saber tirar partido das novas arquitecturas heterogéneas, que facilitam a comunicação entre os *Central Processing Units* (CPUs) e os GPUs, tendo como objectivo reduzir a latência da comunicação entre eles.

O trabalho aqui proposto centra-se neste contexto do processamento de imagem com recurso a GPUs, tendo como base de partida a paralelização em GPUs de algoritmos de reconstrução de imagem para a detecção de cancro na mama. Esta paralelização foi desenvolvida numa tese de Mestrado por Pedro Ferreira [1] a qual, por sua vez, partiu de implementações sequenciais realizadas pelo Prof. Nuno Matela e Nuno Oliveira [46] de três algoritmos iterativos, rápidos e precisos para reconstrução de imagem usando DBT, por eles desenvolvidos [44]. Para todos estes algoritmos (SART, ML-EM e OS-EM), existe uma fase comum com grande custo computacional, a geração da matriz sistema. Esta matriz representa a atenuação dos raios-X, emitidos pela máquina durante o exame, no corpo do paciente. O trabalho de Pedro Ferreira dedicou-se à implementação da geração da matriz utilizando técnicas de processamento paralelo, usando NVIDIA CUDA para programar GPUs.

No entanto, o trabalho previamente desenvolvido deparou-se com a dificuldade da programação em CUDA, dado que os modelos de programação disponibilizados são ainda de baixo nível, tornando difícil quer o desenvolvimento quer a compreensão destes programas e, consequentemente, a sua extensão. Estas dificuldades são acrescidas quando se trata de cientistas e engenheiros sem uma formação extensa em informática, nem em processamento paralelo, em particular. Para além disso, verifica-se que a implementação desenvolvida não consegue ser portada rapidamente para outra arquitectura, sendo necessário efectuar alterações ao programa para que este funcione como esperado, para além do investimento necessário em conhecer os detalhes da nova arquitectura. Adicionalmente, como é comum neste domínio, o conhecimento adquirido em termos das melhores opções de paralelização para o problema em questão é difícil de reutilizar para problemas similares. A arquitectura da solução de paralelização desenvolvida nem sempre é evidente, sendo por isso mais complicado de reproduzir, adaptar, ou até comunicar os problemas e suas soluções, àqueles que são menos experientes na área.

Sendo o problema de reconstrução de imagem transversal a diferentes domínios, desde a medicina à engenharia dos materiais (e.g. identificação de materiais compósitos), torna-se igualmente interessante estudar formas de sistematizar esse processamento tendo em vista a sua aplicação em diferentes domínios de uma forma mais rápida e de um modo mais independente da arquitectura computacional subjacente. Para mais, é cada vez maior a necessidade de disponibilizar resultados em ciclos temporais mais curtos (quer por razões económicas quer socais, e.g. caso da medicina), pelo que continua premente o problema de continuar a estudar formas de melhorar o desempenho do seu processamento.

1.2. *Objectivos da tese*

O trabalho proposto nesta tese teve como base o estudo de três algoritmos iterativos de reconstrução de imagem (SART, ML-EM e OS-EM) apresentados em [45] e, em particular, o estudo da paralelização em CUDA de uma fase desses algoritmos, nomeadamente o cálculo da matriz sistema, o qual foi apresentado em [1]. Com este estudo pretendeu-se avaliar as possibilidades de paralelização desses três algoritmos, bem como formas de melhorar o desempenho do cálculo da matriz de sistema, dado que é a etapa mais exigente em termos computacionais. Este estudo insere-se no contexto mais lato de estudar a viabilidade de utilizar esses algoritmos iterativos de reconstrução de imagem e sua paralelização, a áreas diversas em ciência e engenharia.

Como suporte a esses objectivos, foi necessário estudar os modelos de programação paralela existentes, quer os que foram usados na implementação de base, i.e. programação em GPU usando CUDA, quer modelos tradicionais de paralelização dos dados em CPU como seja o OpenMP, quer ainda modelos recentes de programação paralela estruturada com base em algorithmic skeletons. O objectivo geral deste estudo foi avaliar as potencialidades de cada modelo no contexto do problema particular de reconstrução de imagem, e tendo em vista a possível reutilização da solução para problemas similares, bem como permitir o desenvolvimento de diferentes implementações para diferentes plataformas, que pudessem ser comparadas em termos do seu desempenho.

Dadas as limitações temporais desta tese, o trabalho proposto restringiu-se à paralelização do cálculo da matriz de sistema usando os diferentes modelos computacionais. O objectivo foi assim a) produzir uma versão em CUDA que pudesse ter um melhor desempenho que a solução existente; b) produzir uma versão em CPU usando a API OpenMP que é a framework de utilização mais frequente quando as arquitecturas alvo são os multiprocessadores de memória partilhada; c) produzir uma versão usando um framework de suporte à programação paralela estruturada com base em algorithmic skeletons. Este último ponto prende-se com a crescente relevância da programação paralela estruturada dado que disponibiliza abstracções e mecanismos que permitem capturar soluções standard (i.e. comprovadas) para problemas recorrentes e reutilizá-las com um menor esforço em problemas similares.

Finalmente, a implementação dos três tipos de solução teve como objectivo avaliar de forma empírica a facilidade de utilização de cada modelo e, sobretudo, comparar os seus desempenhos como forma de avaliar a viabilidade das diferentes soluções, como parte integrante da paralelização dos algoritmos iterativos de reconstrução de imagem.

1.3. *Solução proposta*

Tendo em vista o objectivo de melhorar a versão em CUDA existente, a solução proposta passou necessariamente por um estudo detalhado dessa implementação e o levantamento dos seus potenciais pontos de melhoramento, tal como descrito na secção 4.2. Tal incluiu a identificação de boas práticas em Engenharia de Software e desenho de algoritmos, em particular, e no contexto particular da programação em GPUs, bem como a metodologia de aquisição dos dados e a sua organização.

Deste estudo resultou também a observação da possibilidade de melhorar o modo de ordenar os resultados no cálculo da matriz de sistema, alteração esta que é transversal às diferentes implementações. Como tal, foi incluída nas diferentes versões, produzindo melhorias significativas no desempenho tal como descrito na secção 6.4.

Tendo em vista o objectivo de produzir uma versão em CPU usando uma API popular como é o OpenMP, foi necessário analisar o algoritmo de base do cálculo da matriz de sistema, de um modo independente da sua implementação em CUDA de onde este trabalho partiu. A solução proposta tira também partido do modelo de paralelismo dos dados ("data paralelism") o qual é facilitado pelas directivas e funções disponibilizadas pela API do OpenMP, tal como descrito na secção 2.2.4. Os bons resultados obtidos, descritos na secção 6.4, permitem considerar a utilização desta solução sempre que não seja possível recorrer a uma arquitectura com base em GPUs.

Os objectivos de reduzir o esforço do desenvolvimento de aplicações paralelas, garantindo ao mesmo tempo um desempenho adequado, bem como de permitir reutilizar soluções em problemas similares e que sejam mais facilmente portáveis para diferentes arquitecturas, compreenderam as seguintes dimensões.

Foi estudado o conceito de programação paralela estruturada e suas vantagens, na forma como disponibiliza abstracções e mecanismos que permitem capturar a essência (i.e. as características principais) de soluções comprovadas e conhecimento de peritos na área de alto desempenho. Foram analisados exemplos comuns de abstracções neste domínio tais como padrões paralelos ("parallel design patterns") e algorithmic skeletons que capturam o código de soluções comprovadas, i.e. padrões/esquemas recorrentes na programação paralela, tal como apresentado na secção 2.3. Tanto os padrões paralelos como os skeletons são caracterizados por oferecerem um maior nível de abstracção, dado que capturam as interacções relevantes de um esquema de paralelização, sem compromisso com uma arquitectura paralela específica. Assim, é possível libertar o programador da responsabilidade de tratar da distribuição dos dados entre os processos, da sua comunicação e sincronização, facilitando a implementação de

programas paralelos [2]. Dado que o desenho de aplicações com base nestas abstracções não depende da arquitectura em que o programa será executado, permite também suportar portabilidade do código de um modo simples [2].

Foram estudadas alternativas em termos de frameworks de suporte à programação com base em algorithmic skeletons, com um levantamento das suas características principais. Um problema com os frameworks correntes é estarem limitados pelos skeletons existentes, visto que estes apenas se focam em soluções de algoritmos bem definidos [3]. Outro problema é a falta de especificação para definir e trocar skeletons entre diferentes implementações [3], onde a solução passaria por criar um standard completo, o que ainda não se verifica. Para além disso, o desempenho dos frameworks baseados em skeletons ainda se afasta das soluções dedicadas (e.g. específicas para uma arquitectura particular). No entanto, visto que o contexto da reconstrução de imagem é necessário em diferentes domínios, onde nem sempre é imprescindível obter o maior desempenho possível de uma arquitectura, torna-se muito útil poder simplificar a programação e compreensão do código pelo que a estratégia nos parece indicada.

Foi estudado um framework particular baseado em skeletons denominado FastFlow, o qual permite a execução em CPU e GPU, e de que modo os skeletons disponibilizados poderiam ser usados no desenho do cálculo da matriz de sistema, tal como descrito na secção 2.3.4. Estudou-se ainda a possibilidade de melhorar o código do FastFlow tornando-o mais adaptado ao problema particular do cálculo da matriz de sistemas. Foi realizada e avaliada esta implementação tal como descrito nas secções 5.6 e 6.3.

Tendo em vista o objectivo de comparar o desempenho dos diferentes modelos, foi feita a avaliação sistemática dos tempos de execução considerando diferentes cenários, tal como descrito na secção 6.4. Esta comparação procurou definir avaliações em condições o mais similares possível, chamando a atenção para as diferenças, sempre que tal não foi possível. Assim, os tempos do cálculo da matriz de sistema incluíram avaliações das implementações sequenciais (com e sem o novo modelo referido de ordenação de resultados da matriz de sistema), em OpenMP e CUDA (também com e sem esse novo modelo de ordenação), avaliações das implementações em FastFlow para CPU e GPU (novamente com e sem o modelo referido), e finalmente, avaliações para o código do framework Fastflow alterado.

1.4. *Contribuição da tese*

Tendo em vista a melhoria do desempenho e simplificação da programação de algoritmos de reconstrução de imagens, em particular da paralelização do cálculo da matriz de sistema, este trabalho apresenta as seguintes contribuições:

- Diferente abordagem na ordenação dos resultados no cálculo da matriz sistema (comparativamente ao trabalho de base apresentado em [1]).
- Diferentes implementações em CPU e GPU do código do cálculo da matriz de sistema com base no código IDL existente e na implementação em CUDA que serviu de base a este trabalho. As diferentes implementações incluíram o novo modo de ordenar os resultados do cálculo da matriz de sistema. Versões produzidas: versões sequenciais, implementações em CUDA, OpenMP e em FastFlow.
- Desenho, implementação e análise do algoritmo do cálculo da matriz de sistema com base em skeletons, no contexto da ferramenta FastFlow. Tal permite abrir caminho ao desenho da implementação dos três algoritmos iterativos de reconstrução de imagem com base em algorithmic skeletons, com produção de código para diferentes arquitecturas num tempo mais reduzido.
- Alteração (limitada) do código da ferramenta FastFlow, possibilitando um melhor desempenho para programas que usam GPU. A alteração está relacionada com as transferências de memória que são necessárias para este tipo de programas.
- Avaliação extensiva do desempenho das diferentes implementações, permitindo a sua comparação no contexto particular do cálculo da matriz de sistema.

1.5. *Estrutura do documento*

Este documento está organizado em 7 capítulos, sendo o primeiro capítulo a introdução. O resto do documento está organizado da seguinte maneira: no capítulo 2 é abordado o estado da arte sobre o processamento e síntese de imagem, reconstrução de imagem (focando o algoritmo de reconstrução iterativo); processamento paralelo em processamento de imagem, indicando as dificuldades e a motivação para a sua aplicação na reconstrução de imagem; e por fim a programação paralela estruturada, incluindo design patterns, algorithmic skeletons e as frameworks que os suportam, onde é apresentada uma breve introdução sobre a sua relevância e a sua potencial aplicação no algoritmo de reconstrução. No capítulo 3 é descrito o trabalho que serviu de base a esta tese, incluindo a reconstrução de imagem para DBT, como funciona a implementação de Pedro Ferreira e cálculo da matriz sistema. No capítulo 4 é apresentado o dese-

nho das implementações realizadas e no capítulo 5 são explicadas as implementações dos vários programas. No capítulo 6 é apresentada a discussão dos resultados de desempenho obtidos com as diferentes implementações (programação paralela em CPU e GPU, em OpenMP e CUDA, respectivamente, e com recurso a uma ferramenta de programação paralela estruturada, o FastFlow). No capítulo 7 podemos consultar as conclusões da tese e o trabalho futuro.

2

Trabalho Relacionado

Neste capítulo discutem-se vários temas que são relevantes para o desenvolvimento da tese, a saber, processamento e síntese de imagem, ambientes de programação paralela e sua aplicação à geração e processamento de imagens digitais.

2.1. *Processamento e Síntese de Imagem*

Uma imagem digital é uma representação numérica, normalmente binária, de uma imagem bidimensional. Há dois tipos importantes de imagem digital, imagem vectorial e imagem *bitmap*. Uma imagem vectorial é baseada em vectores e usa primitivas geométricas para a sua representação computacional. Estas primitivas geométricas podem ser linhas, polígonos, curvas de Bézier, círculos, elipses, entre outros [4]. Por outro lado, uma imagem *bitmap* é representada por uma matriz que contém a descrição de cada pixel. Uma imagem vectorial pode ser redimensionada sem que haja perda de definição, no entanto uma imagem bitmap, que é baseada em pixéis, quando é ampliada mostra uma diminuição da definição. Para o processamento digital de imagem, são mais frequentemente utilizadas imagens *bitmap*.

Em termos formais, uma imagem digital é uma função de duas dimensões $f(x,y)$ onde o x e o y são coordenadas cartesianas e f é a intensidade de qualquer ponto que se encontra no plano. Os valores de x, y e f são finitos e de quantidades discretas.

A forma de representar as cores e as relações entre elas pode ser feita pelo espaço de cores. A visão humana é tricromática, isto significa que existem três receptores nos nossos olhos que reagem à cor vermelha, verde e azul, no inglês *red* (R), *green* (G) e *blue* (B). O espaço de cores RGB funciona da mesma maneira, é constituído por três canais para representar cada uma das cores. Como já foi referido anteriormente, as cores de

uma imagem digital podem ser vistas como uma matriz de pixéis, em que cada elemento tem três valores, um para o vermelho, um para o verde e um para o azul, sendo que cada valor está compreendido entre 0 e 255.

Processamento de imagem pode ser considerado como qualquer forma de processamento de dados onde a entrada é uma imagem. A saída pode ser uma imagem ou um conjunto de características ou parâmetros relacionados com a mesma. Podemos considerar como característica, por exemplo, a cor dominante ou o número de ocorrências de um determinado objecto. Esta informação pode ser útil para os casos em que é necessário efectuar várias pesquisas, por certas características, num conjunto de imagens, onde não é necessário processar uma imagem cada vez que se efectua uma pesquisa.

Existem várias técnicas para o processamento de imagem, sendo que as mais simples são aquelas que apenas usam um pixel de cada vez para efectuar os cálculos necessários, que são, entre outras, o filtro, o equilíbrio das cores, os histogramas de imagem e de cor. Como técnicas mais complexas, temos a segmentação e a *Image Registration*. As seguintes secções explicam com maior detalhe algumas destas técnicas, ilustrando, sempre que possível, com um exemplo.

2.1.1. Processamento ponto a ponto

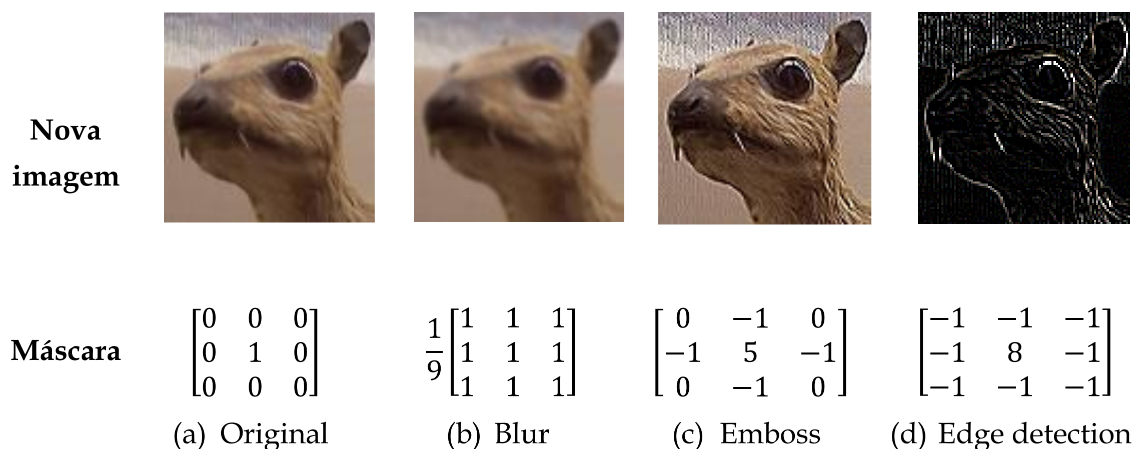
No tipo de processamento ponto a ponto, a transformação que é aplicada muda apenas um pixel. Seja $f(x,y)$ o valor de um pixel em que x e y são coordenadas de uma imagem, $g(x,y)$ uma transformação de $f(x,y)$ e T a função aplicada a $f(x,y)$. A função T mapeia $f(x,y)$ em $g(x,y)$ e a transformação apenas afecta um pixel nas coordenadas (x,y) (Equação 2.1).

$$\mathbf{g(x,y) = T[f(x,y)]}$$ **Equação 2.1**

Existem vários tipos de processamento de imagem deste tipo, entre eles os filtros *negative, grayscale, brightening, darkening* e *thresholding*.

2.1.2. Convolução entre máscara e imagem

Este algoritmo é normalmente utilizado como um filtro para mudar as características de uma imagem. Nas imagens digitais, para cada máscara utilizamos uma matriz, por exemplo 3x3, em que é aplicada a sua convolução com a imagem original. Cada pixel gerado resulta do cálculo da média ponderada dos pixéis vizinhos, sendo que o pixel a ser processado encontra-se no centro da matriz. O resultado final é uma nova imagem com um novo efeito. Os efeitos mais conhecidos são o *blur*, o *sharpen* e *edge detection.*



$$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix} \quad \frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix} \quad \begin{bmatrix}0&-1&0\\-1&5&-1\\0&-1&0\end{bmatrix} \quad \begin{bmatrix}-1&-1&-1\\-1&8&-1\\-1&-1&-1\end{bmatrix}$$

Figura 2.1: Exemplos de convoluções de máscaras com a imagem original

2.1.3. Segmentação de uma imagem

A segmentação refere-se ao processo de divisão de uma imagem em vários segmentos. O objectivo principal desta técnica é simplificar e/ou mudar a representação de uma imagem em algo mais informativo e fácil de analisar. Esta técnica é tipicamente usada para localizar objectos e fronteiras numa imagem. Foi realizada uma experiência na secção **2.4.4.**

2.1.4. Image Registration (Alinhamento de Imagens)

Image registration corresponde ao alinhamento geométrico de várias imagens do mesmo conjunto de objectos. Uma das facetas da *image registration* é a análise multi-temporal em que o alinhamento é feito em relação a imagens dos mesmos objectos adquiridas em alturas diferentes. Envolve a existência de uma imagem de referência da qual é extraído um conjunto de características (*features*); numa 2ª fase essas *features* são procuradas nas imagens seguintes. As mudanças nas características entre as várias imagens permitem elaborar um modelo que deve ser ajustado às alterações entre imagens [5].

Um exemplo da aplicação deste a imagens médicas permite lidar com as alterações causadas pela respiração ou por outra alteração anatómica.

2.2. *Processamento Paralelo em Processamento de Imagem*

Uma vez que esta tese se centra na paralelização de algoritmos iterativos e de reconstrução de imagem, nesta secção analisam-se alguns aspectos do processamento paralelo e da sua aplicação ao processamento de imagens

2.2.1. Programação paralela e suas dificuldades/desafios

A programação paralela caracteriza-se pelo uso de vários recursos computacionais para resolver um problema. Esses problemas são divididos em várias partes, em que cada parte vai ser executada em simultâneo por vários processadores. A exploração da concorrência é a chave para a programação paralela. Em teoria, utilizar mais recursos para resolver um problema, vai diminuir o seu tempo de execução. Estes recursos podem ser um único computador com vários processadores, vários computadores interligados entre si ou uma combinação dos dois. Tem que se ter cuidado para garantir que a sobrecarga da gestão da concorrência não aumente o tempo de execução do programa. Para além disso, a divisão equilibrada do trabalho pelos vários processadores pode por vezes, ser complicada.

Tradicionalmente, grande parte do software desenvolvido utilizava apenas um único processador, onde as instruções são executadas sequencialmente. O crescimento do desempenho dos processadores tem sido impulsionado pelo aumento da densidade de transístores nos circuitos integrados. À medida que o tamanho dos transístores diminui, o seu tempo de computação diminui, permitindo aumentar a frequência do relógio que dita o número de operações por unidade de tempo que o circuito integrado consegue fazer, bem como o seu consumo de energia. No entanto, o aumento da frequência do relógio não pode prosseguir porque não é possível dissipar o calor gerado no interior do circuito. Como o número de transístores que se podem colocar continua a aumentar, optou-se por colocar múltiplos processadores num único circuito integrado [12].

Contudo, adicionar mais processadores não vai aumentar o desempenho de programas sequenciais, porque tais programas não podem tirar partido da sua existência. Por outro lado, o desenvolvimento de algumas aplicações é mais fácil, se forem construídas como um conjunto de processos cooperantes (e.g. simulações na área da meteorologia compostas por diversos módulos). Surge assim a necessidade de existência de ambientes de programação que suportem mecanismos para a gestão de processos, e para a sua comunicação e sincronização. Quando se escreve um programa em que existem vários processos cooperantes, a criação de processos e as operações de comunicação e sincronização podem ser realizadas de duas formas:

- ***Explicitamente:*** O programa contém as instruções que especificam que processos são inicializados e executados de forma paralela. O programador tem controlo absoluto sobre a execução paralela.
- ***Implicitamente:*** O compilador tem a tarefa de adicionar as instruções necessárias para o programa executar de forma paralela.

Assim, uma dimensão a considerar na paralelização é a divisão do trabalho que tem de ter em atenção a arquitectura subjacente, ou seja, o modelo de programação pode e deve tirar partido do sistema paralelo sobre o qual vai executar. Existem dois tipos principais de sistemas paralelos, de memória partilhada e memória distribuída:

- Em sistemas de memória partilhada, os processos podem partilhar o acesso à memória do computador usando-a como forma de comunicação e coordenação entre si. Isto é suportado porque cada processador pode ler e escrever em todas as posições de memória, podendo cada CPU observar todas as alterações efectuadas pelos processos, tendo no entanto que ter em linha de conta a consistência dos dados. Podemos coordenar os processos fazendo com que cada um examine e actualize posições de memória partilhada e, por exemplo, execute partes diferentes do código, de acordo com esses valores num determinado instante.
- Em sistemas de memória distribuída, cada processador tem apenas acesso à sua memória privada, obrigando a que a troca de informação seja realizada por troca de mensagens por uma rede de interligação. Isto quer dizer que o programador de aplicações tem de ter à sua disposição mecanismos explícitos para transferir informação entre os espaços de endereçamento de processos que residem em CPUs distintos. O desempenho destes sistemas depende de dois factores:
    - A taxa de transferência entre CPUs (em bytes / segundo);
    - A latência de comunicação que é o tempo gasto na comunicação entre dois CPUs mesmo quando a quantidade de informação transferida é muito pequena.

Quando os processos trabalham independentemente uns dos outros, desenvolver programas paralelos não é uma tarefa muito complicada. Tudo se torna mais complexo quando os processos necessitam de coordenar o seu trabalho e trocar informação entre si. Nos sistemas de memória partilhada, este factor pode limitar a eficiência da paralelização, devido a latências, limitação da largura de banda e também consistência dos dados (é importante distribuir bem o trabalho entre todos e sincronizar cada um deles antes de começarem a executar a sua parte do trabalho).

Pela lei de Amdahl [13], a fracção de código sequencial nestes programas, ou seja, a parte do código que não pode ser paralelizado (e.g. leitura de dados de um ficheiro), limita a aceleração que se pode atingir utilizando vários CPUs, visto que o tempo de execução nunca vai ser inferior ao tempo de execução da parte intrinsecamente sequencial do código.

À medida que o poder computacional aumenta, o número de problemas que podemos considerar resolver também aumenta, devido ao facto de conseguirmos proces-

sar estes problemas num mais curto espaço de tempo. Conseguimos assim manipular grandes volumes de dados e desenvolver modelos muito mais detalhados, não estando limitados apenas pelo poder computacional de um único computador. Por exemplo, na área de processamento de imagens, é possível diminuir o tempo de processamento dividindo partes da imagem entre os processos a executar em paralelo.

2.2.2. Eficiência da paralelização

No processamento paralelo, o programador cria várias tarefas que trabalham em conjunto para resolver um problema. A ideia principal é subdividir o problema em tarefas mais simples, que possam ser executadas ao mesmo tempo em diferentes CPUs, tentando sempre dividir o trabalho de forma uniforme para cada tarefa. Essa subdivisão está relacionada com a **Granularidade** que está associada ao quociente entre o número de computações efectuado por um elemento de processamento e as operações de comunicação e sincronização efectuadas. Pode ser classificada em [14]:

- ***De grão grosso (Coarse-grained):*** Poucas tarefas com computações mais intensas. Um exemplo desta decomposição é apresentado na secção 2.2.2.1.
- ***De grão fino (Fine-grained):*** Maior número de tarefas mais pequenas com computações menos intensas. Um exemplo desta decomposição é apresentado na secção 2.2.2.1.

A redução do tempo de execução que se consegue pela paralelização de um algoritmo depende de múltiplos factores alguns dos quais se referem seguidamente:

- A parte intrinsecamente sequencial do mesmo.
- O tempo gasto (*overhead*) na comunicação e sincronização entre processos.
- As dependências entre processos, como quando um dado processamento depende dos resultados de uma etapa anterior.
- As assimetrias na distribuição de trabalho pelos vários processadores

A natureza do problema a resolver e a qualidade da concepção da estratégia de paralelização vai definir a Escalabilidade, isto é a capacidade de um algoritmo manter a sua eficiência aumentando, na mesma proporção, o número de processos e o tamanho do problema. Define-se speedup (aceleração) como a relação entre o tempo de execução S_S (execução com um único processador) e S_N (execução com N processadores).

Equação 2.2: Cálculo do speedup.

$$S = \frac{S_S}{S_N}$$

2.2.2.1. Distribuição dos dados

Muitos dos algoritmos de processamento de imagem apresentam um paralelismo natural da seguinte forma: os dados, da imagem de entrada, necessários para calcular uma parte da imagem de saída estão espacialmente distribuídos. No caso mais simples, uma imagem de saída é obtida pelo processamento independente de cada pixel da imagem de entrada. Por outro lado, podemos utilizar uma vizinhança (janela) de pixéis, da imagem de entrada, para gerar a imagem de saída, como se pode ver na Figura 2.2 [15]. Como os valores dos pixéis da imagem de saída não dependem uns dos outros, podem ser calculados independentemente uns dos outros e em paralelo, desde que a imagem de saída esteja localizada noutra zona da memória.

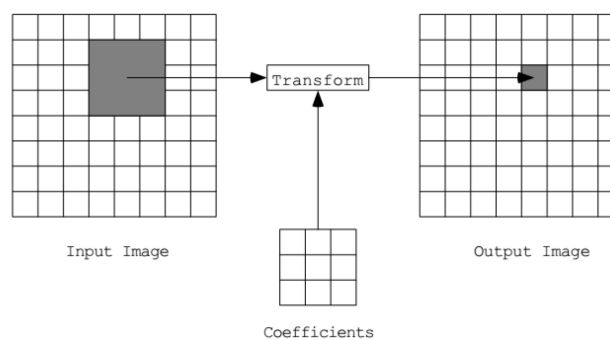


Figura 2.2: Dependência dos dados para um algoritmo de processamento de imagem usando um operador em janela.

A decomposição paralela com granularidade fina (*fine-grained)* utilizando um algoritmo de processamento de imagem baseado num operador em janela, atribuiria um pixel de saída para cada processo e também a informação necessária da janela. Cada processo teria que efectuar as computações necessárias para calcular o seu pixel de saída. A Figura 2.3 [15] ilustra um exemplo da decomposição de uma imagem com granularidade fina.

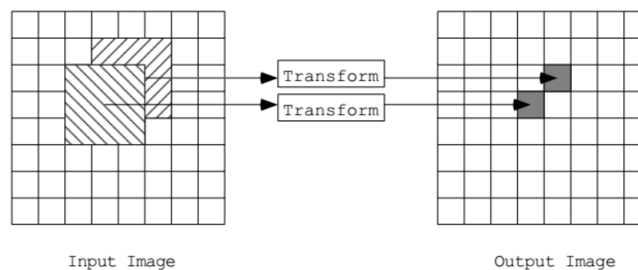


Figura 2.3: Decomposição *fine-grained* para um algoritmo de processamento de imagem usando um operador em janela

Uma decomposição com granularidade mais alta atribuiria grandes regiões contínuas da imagem para cada um dos processos, sendo o número de processos consideravelmente mais baixo. Cada processo teria que efectuar as operações à sua região da imagem utilizando a janela apropriada. Dependendo do tamanho das janelas que estão a ser utilizadas, as regiões de sobreposição têm que ser devidamente atribuídos para possibilitar o cálculo dos pixéis de saída. A Figura 2.4 [15] ilustra um exemplo da decomposição de uma imagem com granularidade mais alta.

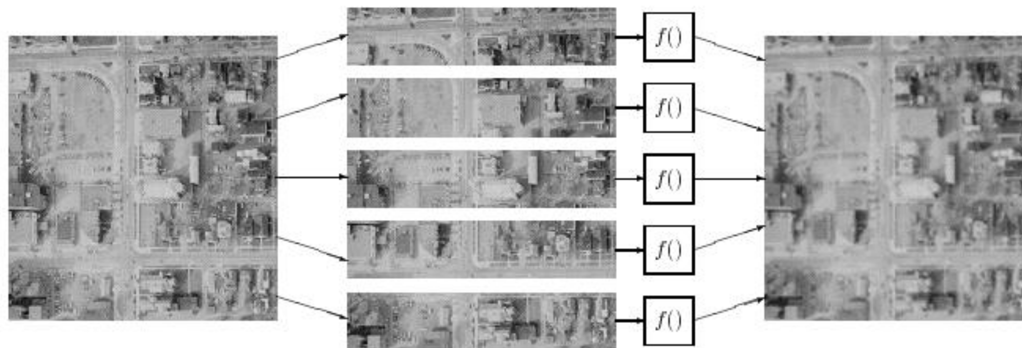


Figura 2.4: Decomposição coarse-grained para um algoritmo de processamento de imagem usando um operador em janela

2.2.2.2. Convolução entre máscara e imagem

As convoluções são muito exigentes a nível computacional. Tomemos como exemplo [16] a convolução de uma imagem $N \times N$ com uma máscara $M \times M$. O número total de operações necessárias é $M^2 \times N^2$ multiplicações, $(M^2 - 1) \times N^2$ adições e $(2 \times M^2 \times N^2 + N^2)$ loads/stores. Daqui surge a necessidade de desenvolver soluções eficientes para melhorar a velocidade do processamento destas convoluções. À medida que as máscaras aumentam de tamanho, a convolução torna-se cada vez mais computacionalmente exigente, devido ao crescimento quadrático de M^2 no número de computações. Para simplificação, tomemos como máscara uma matriz 3 × 3 (Figura 2.5).

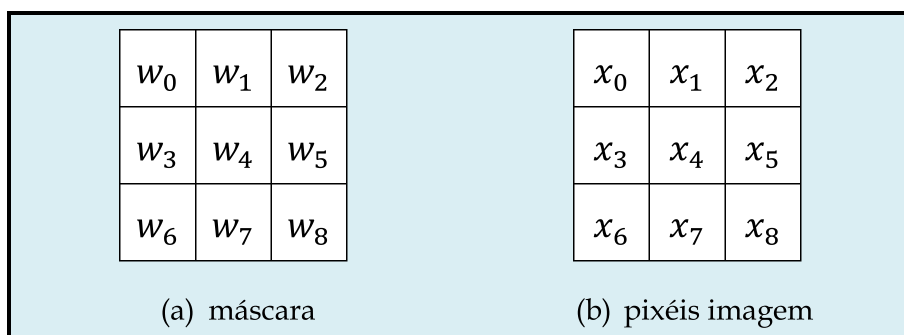


Figura 2.5: (a) máscara usada na convolução; (b) pixels da imagem original usados na convolução [26]

2.2.3. Dificuldades

O processamento de imagem com operações que envolvem apenas um pixel, é um exemplo de paralelismo perfeito (*embarassing parallelism*). Nestes casos, não é necessário muito esforço para dividir a carga entre as tarefas e não existe qualquer dependência entre cada pixel e os seus resultados, por isso podem ser obtidos em paralelo sem qualquer problema. Na verdade, nem todas as operações efectuadas são assim tão simples, existindo operações com complexidades muito mais elevadas que envolvem, para o cálculo de um dado pixel, vários dos seus vizinhos.

Nestes problemas mais complexos, surgem a maior parte das dificuldades que também estão presentes em várias aplicações paralelas. O programador tem que pensar como vai decompor o problema em causa, como vai integrar as soluções parciais e como vai ser executada a comunicação entre cada tarefa. Como as imagens podem variar a sua resolução com bastante frequência, estas soluções necessitam de ser escaláveis, o que pode ser um desafio para quem está a desenvolver o programa. O aumento to tamanho de uma imagem, pode levar à diminuição de eficiência do uso da arquitectura hardware, o que se reflecte numa diminuição do speedup.

Muitas vezes, é necessário utilizar primitivas de paralelização de baixo nível para conseguir obter os melhores resultados desejados. Isto exige a que o programador tenha que conhecer grande parte destas primitivas de baixo nível, por exemplo *pthreads*. Este nível de programação revela-se trabalhoso, e pode ser necessário investir bastante tempo no desenvolvimento de programas até que se atinga o grau de optimização desejado. Contudo, também existem primitivas com uma maior abstracção, que permitem ao programador desenvolver aplicações paralelas com maior facilidade, obtendo, ainda assim, um bom desempenho.

As diferentes arquitecturas existentes são um problema para a paralelização de aplicações, porque muitas vezes a portabilidade é das mesmas para novas arquitecturas o que obriga a reescritas profundas. Isto torna o processo de programação mais complexo, visto que o programador tem que conhecer mais modelos de programação. Uma solução passa por criar programas com um nível de abstracção mais alto e com compiladores para arquitecturas diversas, que podem ser executados em arquitecturas diferentes sem que haja diminuição no seu desempenho.

Não se pretende aqui fazer uma síntese dos âmbitos de programação paralela – ver por exemplo a referência [40][41], seguindo-se uma apresentação de alguns desses âmbitos usados nesta tese.

2.2.4. OpenMP

Para a paralelização em CPU foi escolhido o OpenMP dado que é uma API muito conhecida para programação paralela explícita em memória partilhada e está disponível em C++. A API do OpenMP usa um modelo paralelo de execução `fork-join`. Várias `threads` realizam as tarefas definidas explicitamente ou implicitamente pelas directivas do OpenMP. No C/C++, as directivas do OpenMP são especificadas usando o mecanismo **`#pragma.`** Esta API está desenvolvida para suportar programas que executam correctamente na versão paralela e na versão sequencial (as directivas são ignoradas e o programa é compilado em modo sequencial) [29].

```
#pragma omp directive-name [clause, …] new-line
     structured-block
```

Cada directiva começa por **#pragma omp**, seguida por `directive-name` que tem que ser uma directiva OpenMP válida. Um exemplo é a directiva parallel que forma uma equipa de *threads* e inicia uma execução paralela. As cláusulas (`[clause, …]`) são opcionais, podem aparecer em qualquer ordem e se necessário repetidas.

As cláusulas podem ser uma das seguintes [29]:

```
if(scalar-expression)
num_threads(integer-expression)
default(shared|none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction(operator: list)
```

(A nova linha (`new-line`) é obrigatória, seguido do bloco estruturado (`structured-block`) que vai ser executado em paralelo).

A Listagem 2.1 ilustra um exemplo que mostra a facilidade com que se consegue construir um programa paralelo a partir do programa sequencial usando OpenMP.

```
//Sequencial

for(long i=0;i<N;++i){
        A[i]+=1;
}
```

```
//OpenMP

#pragma omp parallel for
num_threads(workers)
for(long i=0;i<N;++i){
        A[i]+=1;
}
```

Listagem 2.1: Exemplo das diferenças entre código sequencial e código paralelo usando OpenMP.

2.2.5. CUDA

As GPU (Graphics processing unit) apresentam relação desempenho/custo/consumo muito favoráveis em certas classes de problemas, nomeadamente os de processamento de imagem. O principal ambiente de programação em causa para este dispositivo é o CUDA [41] que é apresentado brevemente em seguida.

Convém relembrar os seguintes conceitos, host refere-se ao CPU e à sua memória (*host memory*) e device refere-se ao GPU e à sua memória (*device memory*). Como já foi referido um *kernel* é uma função que é executada N vezes por N *threads* de CUDA. A palavra-chave `__global__` do CUDA C/C++ indica uma função que corre no device e é chamada pelo host:

```
__global__ void nome_kernel(argumentos)
```

Para se chamar esta função, para além de especificar os argumentos da função, como é feito no C, é necessário indicar o número de blocos e o número de *threads* por bloco:

```
nome_kernel<<<numero_blocos, numero_threads>>>(argumentos);
```

onde ambos podem ser do tipo inteiro (`int`). O número total de threads é igual a: `numero_blocos x numero_threads`. Cada *thread* em execução tem um identificador único, que é calculado combinando o identificador da *thread* com o identificador do bloco, estes identificadores podem ser acedidos usando as variáveis `threadIdx` e `blockIDx` respectivamente:

```
int index = threadIdx.x + blockIdx.x * M;
```

em que M é a dimensão do bloco. Estas variáveis são compostas por 3 componentes, para se poder usar índices unidimensionais, bidimensionais e tridimensionais. Também podemos saber a dimensão de um bloco no *kernel* usando a variável `blockDim`:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

A memória host e device são duas entidades separadas, os apontadores do device referem a memória do GPU, enquanto que os apontadores do host referem para a memória do CPU. A API do CUDA disponibiliza funções que permitem manipular a memória do device muito semelhante ao equivalente no C. Os apontadores do host são tratados como no C convencional. Para reservar memória usa-se o `malloc()` e para libertar usa-se o `free()`. Para os apontadores do device, o mesmo pode ser feito usando as funções `cudaMalloc()` e `cudaFree()`. Enquanto o `malloc()` devolve um apontador para o objecto reservado, `cudaMalloc()` escreve o apontador no apontador do device e devolve qualquer erro que tenha ocorrido:

```
cudaMalloc( (void **) &device_pointer, size);
```

Para transferir os dados do host para o device e vice-versa, é usada a função `cudaMemcpy()`, semelhante à função `memcpy()` do C mas com um argumento extra indicando o tipo de transferência, se esta é do host para o device ou do device para o host (`cudaMemcpyHostToDevice` ou `cudaMemcpyDeviceToHost`, respectivamente).

Segue-se um pequeno exemplo que mostra as diferenças entre um programa em C e um programa em CUDA. Neste exemplo [31] o que se pretende fazer é incrementar um valor *b* a todos os *N* valores do vector *a*.

```
1   void increment_cpu(float *a, float b, int N)
2   {
3         for (int idx=0; idx < N; idx++)
4             a[idx] = a[idx] + b;
5   }
6
7   void main()
8   {
9         …
10        increment_cpu(a,b,N);
11  }
```

Listagem 2.2: Programa CPU.

```
1   __global__ void increment_gpu(float *a, float b, int N)
2   {
3        int idx = threadIdx.x + blockIdx.x * blockDim.x;
4        if(idx < N)
5            a[idx] = a[idx] + b;
6   }
7
8   void main()
9   {
10       …
11       dim3 dimBlock(blocksize);
12       dim3 dimGrid( ceil( N/(float)blocksize) );
13       increment_gpu<<<dimGrid, dimBlock>>>(device_a, device_b, N);
14  }
```

Listagem 2.3: Programa CUDA.

2.3. *Programação Paralela Estruturada como Técnica Geral para o Processamento Paralelo*

A programação paralela estruturada consiste no desenvolvimento de programas paralelos com base em abstracções que capturam esquemas recorrentes de paralelização, quer ao nível do modelo de programação, quer ao nível do modelo de execução paralela. A sua utilização no contexto de reconstrução de imagens 3D permite definir uma aplicação usando abstracções que capturam a execução paralela mas escondem os seus detalhes.

Como já foi visto na secção 2.2, paralelizar um programa é uma tarefa complicada, em que é necessário indicar explicitamente como é realizada a distribuição da carga pelos processos, como vão ser integrados os resultados parciais e como é feita a comunicação entre eles. Isto implica que o programador necessita de dominar as primitivas de baixo nível que permitem o desenvolvimento destes programas paralelos. Para além disso, a existência de diferentes arquitecturas dificulta a portabilidade sendo necessário conhecer os detalhes de muitas e novas arquitecturas, por vezes bem distintas. Tal limita o objectivo de criar programas de alta qualidade, que não são apenas eficientes e escaláveis, mas também são genéricos e bem programados [20].

O processamento paralelo de imagem é necessário em diferentes domínios, nos quais os engenheiros e cientistas não têm muitos conhecimentos de programação. O desenvolvimento de programas paralelos torna-se menos complicado quando estes têm acesso a um modelo que permite obter paralelização sem grandes conhecimentos na área por recurso a abstracções mais simples e independentes de diferentes arquitecturas. Desta forma, o programador consegue obter programas paralelos com bons resultados, sem a necessidade de perceber os detalhes de implementação.

2.3.1. Design Patterns

Os design patterns são expressões de alto nível que generalizam algoritmos paralelos que representam um problema paralelo recorrente [20]. Um design pattern não é um modelo acabado, que pode ser transformado directamente em código, mas sim uma descrição de como podemos resolver um problema que pode ser usado em várias situações. A ideia é registar as experiências dos especialistas de uma maneira que as suas soluções possam ser utilizadas por outros enfrentando problemas semelhantes. A ideia de capturar soluções recorrentes na área de arquitectura, e os apresentar de uma forma textual, ilustrando a sua utilização, implementação e relação com outros padrões, surgiu pela primeira vez no trabalho de Alexander [37]. Neste trabalho existe uma série de padrões por onde escolher e também uma linguagem de padrões que in-

troduz uma nova abordagem para o desenho das aplicações. Nesta linguagem, os padrões estão organizados numa estrutura que leva o utilizador através de uma série de padrões que permite o desenho de sistemas complexos usando padrões. Em cada ponto de decisão, o utilizador escolhe o padrão mais apropriado (apesar do uso do termo linguagem, a linguagem de padrões não é uma linguagem de programação). Inspirado neste trabalho, surgiu o livro sobre Design Patterns [42] no contexto dos sistemas distribuídos e do modelo Object-Oriented, sendo seguido por outros trabalhos.

No livro [38] a linguagem de padrões paralelos está organizada em 4 espaços, formando uma hierarquia:

1. **Encontrar Concorrência:** exposição da concorrência de um problema que pode ser explorada;
2. **Estrutura do Algoritmo:** estruturar o algoritmo para tirar vantagem da potencial concorrência;
3. **Estruturas de Suporte:** ligação entre o nível superior e o nível inferior. Existem dois grupos de padrões neste espaço, aqueles que representam as abordagens da estruturação do programa e aqueles que representam as estruturas mais comuns de memória partilhada;
4. **Mecanismos de Implementação:** como é que os padrões dos níveis mais altos são mapeados em ambientes de programação específicos. É usado para descrever mecanismos comuns de gestão e interacção de processos.

Segue-se um exemplo da utilização desta abordagem no problema da reconstrução de imagens médicas. Para resolver este problema, é necessário criar modelos de propagação da radiação pelo corpo dos pacientes. A ideia é que temos uma partícula (raio) que passa pelo corpo do paciente, e é atenuada pelos diferentes órgãos por onde passa, continuando até sair do corpo e atingir um detetor, definindo uma trajectória. Para uma reconstrução é necessário milhares, e por vezes milhões, de raios.

É natural associar cada tarefa a cada raio (task decomposition pattern). Estas tarefas são fáceis de gerir concorrentemente porque são completamente independente umas das outras. Tendo as tarefas definidas, consideramos então a decomposição dos dados. Cada tarefa necessita de manter a informação que define a trajectória do raio. Para além disso, as tarefas necessitam de ter acesso ao modelo do corpo. Como o modelo é apenas acedido para leituras, não existem problema se estamos perante um sistema de memória partilhada, cada tarefa pode ler os valores quando precisa.

Visto que as tarefas são independentes umas das outras, a única questão com que temos que nos preocupar quando escolhemos a estrutura do algoritmo é como mapear

as tarefas. Para este problema, o princípio de organização parece ser como as tarefas estão organizadas. De seguida, consideramos a natureza do nosso grupo de tarefas, se estas estão organizadas numa hierarquia ou se não existe organização. Para este problema, as tarefas não estão organizadas e não existe uma estrutura hierárquica entre elas, por isso escolhemos o padrão Data Parallel. Todo este processo de identificação dos padrões está explicado no livro [38], este exemplo apenas ilustra como é que identificamos os padrões para um problema real recorrente.

Kurt Keutzer e Tim Mattson [39] identificaram diversos padrões e estruturaram-nos em camadas usando uma abordagem de engenharia de software. Desenhar programas paralelos é complexo. Para gerir esta complexidade, a sua abordagem passa por dividir o problema numa hierarquia em camadas. Cada nível da hierarquia trata de uma parte do problema. Quem gere o desenvolvimento de uma aplicação paralela tem que perceber todas as camadas da hierarquia, mas na prática a maior parte dos programadores numa aplicação pode restringir o seu trabalho ao nível que está directamente associado à sua parte da aplicação. As duas primeiras categorias estão no mesmo nível da hierarquia, cooperando para criar uma camada de arquitectura de software identificada como "camada de produtividade". Os padrões a este nível são utilizados por engenheiros e cientistas, e capturam soluções próximas dos seus domínios de aplicação. Estas duas categorias definem a arquitectura de software de alto nível de uma aplicação.

1. **Padrões de estruturação:** descreve a organização da aplicação e a forma como os elementos computacionais que formam a aplicação interagem.
2. **Padrões computacionais:** descrevem as classes essenciais que formam a aplicação. Estes padrões descrevem o que é computado no que foi definido pelos padrões de estruturação.

As categorias seguintes definem a "camada de eficiência", cujos padrões capturam conhecimento adquirido na área de paralelismo.

3. **Estratégias do algoritmo:** definem estratégias de alto nível para explorar a concorrência numa computação para ser executada num computador paralelo.
4. **Estratégias de implementação:** como é que o programa em si está organizado e estruturas de dados comuns específicas para programação paralela.
5. **Padrões de execução paralela:** abordagens usadas para suportar a execução de um algoritmo paralelo. Por exemplo, construção de blocos básicos para coordenação das tarefas.

De notar que nas primeiras duas categorias não é dito nada sobre paralelismo. Nestas categorias o importante é saber quais os padrões mais relacionados com o domínio da aplicação, dando uma visão de alto nível sobre o problema. Por outro lado, nas categorias seguintes, é feita uma descrição focada na implementação de programas paralelos. Esta divisão permite que numa camada superior seja possível usar os padrões sem necessidade de preocupar com o paralelismo, permitindo um desenvolvimento mais rápido das aplicações. A camada inferior permite a programadores mais experientes implementarem os padrões, de forma optimizada, para estes sejam disponibilizados na camada superior.

2.3.2. Algorithmic Skeletons

O conceito de Algorithmic Skeleton foi primeiro definido por Murray Cole [43]. Existe toda a vantagem em disponibilizar skeletons dessas soluções recorrentes para serem utilizados no desenvolvimento de programas paralelos. Um skeleton representa um esqueleto de código que captura a essência de um padrão de paralelização que implementa. Os skeletons são caracterizados por oferecerem: simplicidade, aumentando o nível de abstracção dado que escondem os detalhes de implementação, libertando o programador deste conhecimento; portabilidade, retirando ao programador a responsabilidade de compreensão detalhada dos padrões; desempenho, dando acesso a padrões optimizados com implementações específicas para uma arquitectura; optimização, documentando as dependências do algoritmo, o que seria impossível extrair num programa não estruturado, e permitindo a definição de funções de custo que dão suporte à previsão do tempo de execução de um programa [21].

Os algorithmic skeletons que se descrevem a seguir são exemplos disponibilizados pelo FastFlow e têm a sua semântica própria.

2.3.2.1. Pipe

O *parallel pattern Pipe* modela uma série de etapas $s_1, \dots, s_n$, em que cada uma delas aplica uma função $f_1, \dots, f_n$ a cada um dos processos independentes que estão a percorrer os estados numa dada ordem [22].

Este skeleton pertence à família de *stream parallel skeletons* e pode ser representado como um grafo (Figura 2.6) em que cada etapa *i* representa a entrada para a estapa *i + 1*. Pode ter um número variável de etapas, em que cada uma delas também pode ser um skeleton.

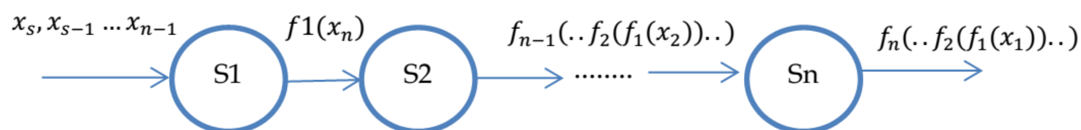


Figura 2.6: Grafo do skeleton pipe [22]

2.3.2.2. Farm

O *Farm pattern*, também conhecido como *master-slave*, representa a replicação de uma acção e a sua execução em paralelo num conjunto de unidades de execução; cada acção numa sequência de acções diferentes pode ser replicada e executada em paralelo. Este padrão suporta computações *embarransingly parallel* em que as acções executadas em cada unidade de execução são independentes entre si. No padrão *Farm u*ma função f pode assim ser aplicada a todas as tarefas de um fluxo de dados $x_1, \dots, x_n$. Este skeleton pode ser visto como um grafo de nós replicados, em que cada um deles avalia a função f para uma dada entrada (Figura 2.7) [22]. Nesta interpretação particular da *Farm* na figura, temos três tipos de nós:

- os *workers* (nomeadamente $w_1, \dots, w_s, s \leq n$) que representam os executores do skeleton.
- o *emitter* que representa o nó responsável por enviar as tarefas do fluxo de entrada para os *workers* que estão disponíveis.
- o *collector* que representa o nó responsável por agrupar os resultados produzidos pelos *workers*.

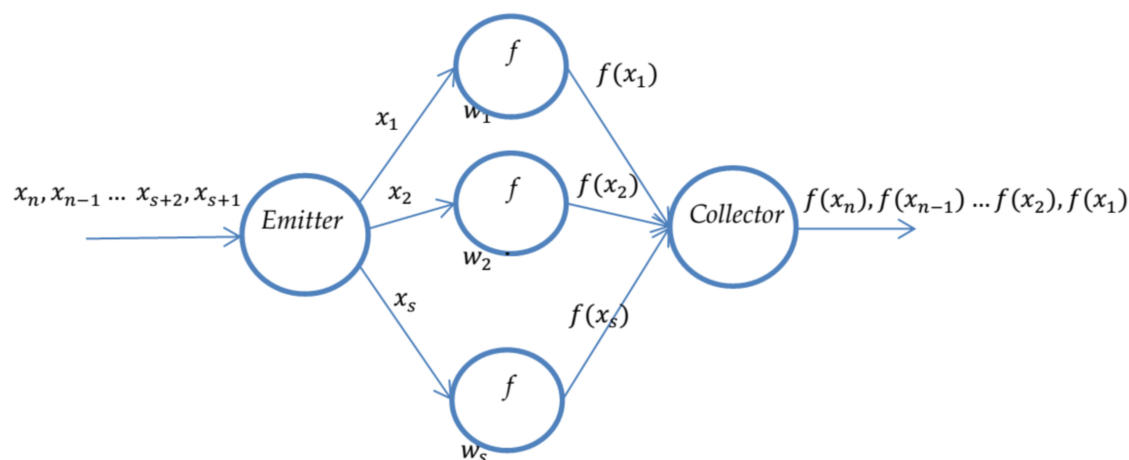


Figura 2.7: Estrutura do grafo da *Farm*, segundo a interpretação do Fastflow *emitter* e *colector* são nós especiais que tratam do agendamento das tarefas para os *workers* e de reunir os seus resultados [22]

2.3.2.3. Map

O *Map pattern* modela as computações paralelas em que uma função *f* tem que ser aplicada a todos os itens (*xi)* que constituem uma estrutura de dados (incluindo-se no modelo de *data paralelism)*. O máximo grau de paralelismo é igual ao número de itens na estrutura de dados. De uma forma geral, divide-se a estrutura de dados em várias partições e efectua-se a computação de todas as partições em paralelo. Este *pattern* é muito semelhante ao *farm*, onde o tamanho do fluxo de entrada não implica o grau de paralelismo, onde a estratégia de agregação dos dados produz um fluxo de valores (ordenado ou não). No caso deste *pattern*, o grau de paralelismo está exactamente definido pelo número de partições da estrutura de dados que será dividida uniformemente entre os executores, onde a estratégia de agregação produz uma nova estrutura de dados (ordenada).

O skeleton pertence à família dos *data parallel skeletons* e o seu grafo (Figura 2.8) [22] pode ser visto como uma instância do grafo do *farm*, tendo as seguintes características:

- o número de *workers* (executores) é proporcional ao numero de partições em que a estrutura de dados vai ser dividida;
- o *emitter* divide o único elemento de entrada (estrutura de dados) entre os *workers*;
- o *collector* produz um único elemento que representa a estrutura de dados onde os resultados foram agrupados ordenadamente.

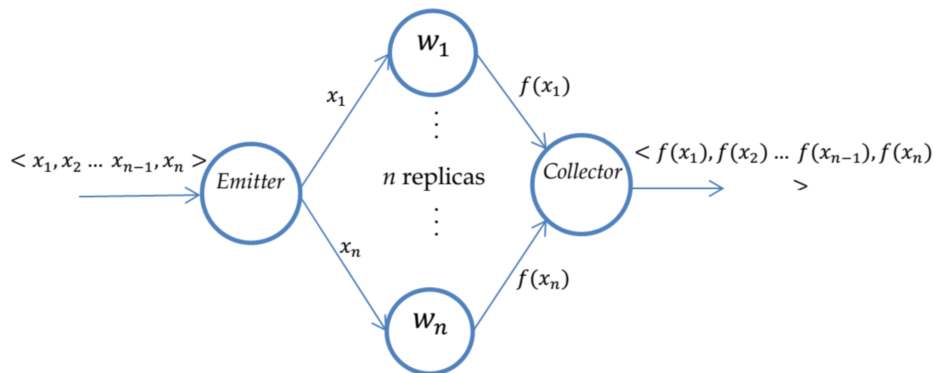


Figura 2.8: O grafo do Map em que cada item é avaliado em paralelo, segundo a interpretação do Fastflow [22].

A utilização destes skeletons é feita ou através do uso de linguagens de programação estendidas (que disponibilizam os skeletons como novas abstracções), ou de bibliotecas/frameworks que disponibilizam a sua utilização através de uma API (seja ela gráfica ou funcional).

2.3.3. Algorithmic Skeletons Frameworks

Algorithmic skeleton frameworks (ASkF) fornecem um conjunto de Algorithmic skeletons, com funcionalidades paralelas genéricas, que são parametrizados pelo programador para gerar um programa paralelo específico. Estas ferramentas surgiram como forma de disponibilizar esqueletos de soluções de paralelização já existentes, bastando ser parametrizados para problemas específicos. Os esqueletos, semelhantes às bibliotecas de software que são normalmente utilizadas, podem ser acedidos através de extensões às linguagens ou interfaces bem definidas. Contudo, o controlo dos dados, monitorização dos recursos e a portabilidade do programa paralelo resultante é efectuado pelo ASkF em vez do programador, tirando a necessidade de preocupação com as implementações paralelas, visto que estas já estão implementadas.

A utilização de ASkF fornece uma metodologia de alto nível de programação paralela, que permite uma descrição abstracta dos programas [23], por exemplo, através de uma linguagem visual em que os skeletons são escolhidos de um menu, compostos e parametrizados para o problema em questão. ASkF promovem a portabilidade focando-se na descrição da estrutura do algoritmo em vez da descrição detalhada da sua implementação. A programação paralela estruturada fornece um comportamento consistente e claro entre plataformas, em que a estrutura subjacente depende de uma implementação em particular de um dado ASkF.

Como já foi referido anteriormente (secção 3.1), o processamento de imagem pode ser mais rápido com a utilização de GPUs. Esta é uma tarefa complicada, que pode ser simplificada com a utilização de ASkF. Fastflow [24] é uma das ferramentas que permite ao programador escrever programas paralelos que possam tirar partido de computações envolvendo o GPU. O problema que surge é que o desempenho oferecido por estas ferramentas podia ser melhor, mas torna-se complicado devido ao nível de abstracção destas ferramentas (a disponibilização de soluções genéricas implica sempre um compromisso em termos de desempenho). No contexto da reconstrução de imagem nem sempre é imprescindível obter o maior desempenho possível de uma arquitectura, sendo também importante a simplificação da programação e da compreensão do código, uma vez que se pretende que os utilizadores que não são especialistas em informática a usem em diversas áreas científicas. Daí o uso destas ferramentas ser uma mais-valia para aqueles que desejam criar programas com relativa facilidade e ao mesmo tempo obter um desempenho razoável.

2.3.4. FastFlow

FastFlow [24] é uma framework de programação paralela em C++, gerando código optimizado para determinado tipo de arquitecturas. Esta ferramenta é especialmente utilizada para o desenvolvimento de aplicações *streaming* e *data-parallel*, focando-se em plataformas heterogéneas compostas por clusters de plataformas de memória partilhada, possivelmente equipados com aceleradores tais como NVIDIA GPGPUs.

Os criadores desta ferramenta tinham como objectivo permitir aos programadores desenvolverem aplicações com características de programação paralela (e.g. tempo de desenvolvimento, portabilidade, eficiência e desempenho da portabilidade) através de abstracções de programação paralela adequadas e suporte cuidadosamente concebido para tempo de execução [24].

Esta ferramenta suporta vários skeletons, entre eles: pipeline, farm, parallelFor, parallelForReduce, MapReduce, StencilReduce, PoolEvolution e MacroDataFlow. Estes skeletons estão implementados em C++/CUDA/OpenCL. No contexto de aplicação, podemos desenvolver programas de forma simplificada usando as implementações disponibilizadas pela ferramenta.

O conceito de stream é bastante importante no FastFlow. Uma stream é uma sequência de valores (possivelmente infinitos), que originam de uma fonte. Por exemplo, uma stream de imagens, stream de matrizes, stream de ficheiros, etc. Uma aplicação de stream pode ser vista como um grafo em que os vértices são nós de computação (sequenciais ou paralelos) e as arestas são canais que trazem streams de dados. Numa computação baseada em stream, tipicamente a primeira etapa recebe dados de uma fonte e produz tarefas para as próximas etapas.

O FastFlow trabalha com dois padrões principais, que operam em stream: pipeline e task-farm (ou simplesmente farm).

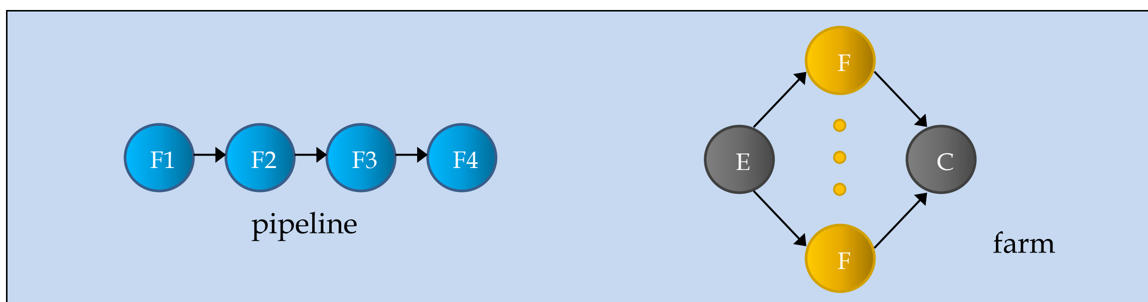


Figura 2.9: Padrões principais do FastFlow que operam em stream. Adaptado de [32].

Pipeline: para cada x, é feita a computação $F4(F3(F2(F1(x))))$. Cada elemento da pipeline é denominado de etapa.

Farm: também denominado de "master-worker". Os elementos de computação são o Emitter (E), Worker (efectua computação de F) e Collector (C). O Emitter faz o planeamento das tarefas para os Workers e o Collector reúne as tarefas dos Workers.

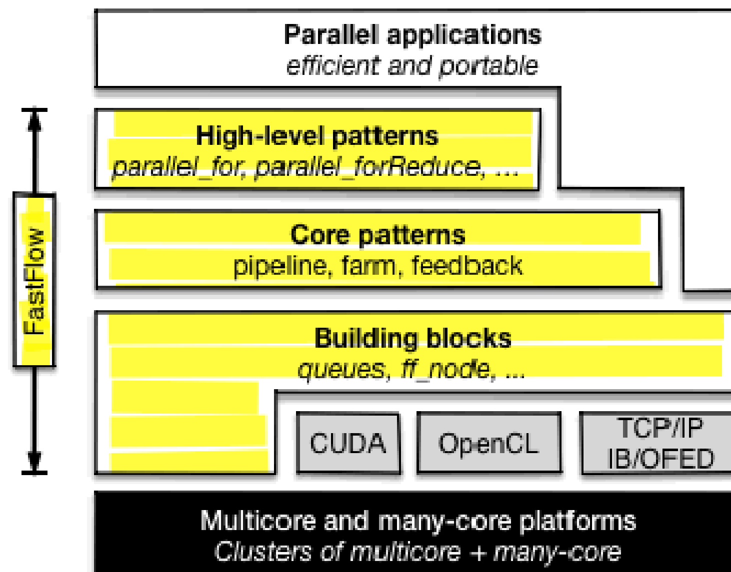


Figura 2.10: Arquitectura da ferramenta FastFlow [32].

A arquitectura do FastFlow está organizada em três camadas principais (Figura 2.10):

Building blocks

Fornece os blocos básicos para construir o suporte dos core patterns. Um processo (ff_node) é sequencial e os caminhos dos dados entre dois processos estão devidamente identificados. As unidades abstractas de comunicação e sincronização são conhecidas como canais e representam uma stream de dependência de dados entre dois processos. Um ff_node é uma classe do C++, após o construtor entra num loop infinito:

1. Recebe uma tarefa pelo canal de entrada (por exemplo, um apontador);
2. Executa o código do trabalho na tarefa;
3. Envia uma tarefa para o canal de saída (por exemplo, um apontador).

A seguinte listagem mostra como esta classe pode ser usada para criar uma etapa nos padrões (Listagem 2.4):

```
1  struct myNode: ff_node_t<TIN,TOUT> {
2      int svc_init() { //opcional
3          //chamada uma vez para inicialização
4          return 0;
5      }
6
7      TOUT *svc(TIN *task) {
8          //fazer algo à task de entrada
9          //é chamada cada vez que uma task está disponível
10         return task; //também pode ser EOS,GO_ON,…
11     };
12
13     void svc_end() { //opcional
14         //chamada uma vez para terminação
15         //chamada se EOS é recebido pelo input
16         // ou se foi gerado pelo próprio nó
17     }
18 };
```

Listagem 2.4: Exemplo de uma etapa usando ff_node.

São aqui apresentados dois valores de retorno especiais:

- **EOS** (End-Of-Stream), que significa final da stream de dados;

- **GO_ON**, que significa que o nó não tem mais tarefas para enviar pelo canal de saída, mas que fica a espera de tarefas que cheguem pelo canal de entrada.

```
1  struct myNode1: ff_node_t<Task> {
2   Task *svc(Task *) {
3      //gera N tarefas e depois EOS
4      for(long i=0;i<N; ++i)
5         ff_send_out(new Task);
6      return EOS;
7   };
8  };
```

```
1  struct myNode2: ff_node_t<Task> {
2   Task *svc(Task * task) {
3      //faz algo com a tarefa
4      do_Work(task);
5      delete task;
6      return GO_ON;
7      // não envia tarefa
8   };
9  };
```

Listagem 2.5: Exemplos de geração (myNode1) e absorção (myNode2) de tarefas.

O exemplo myNode1 é a típica primeira etapa do pipeline, produz tarefas usando a função `ff_send_out` ou simplesmente devolvendo a tarefa do método `svc`. Por outro lado, o exemplo myNode2 é a típica última etapa do pipeline, recebe as tarefas de entrada e não produz nenhum resultado.

Core patterns

Neste nível, são definidos padrões para construir grafos de ff_node(s) (vários ff_node). Como os grafos são uma rede de streaming, qualquer grafo FastFlow é construido usando dois padrões de streaming (farm e pipeline) e um padrão modificador (loopback, para construir redes cíclicas). Estes padrões podem ser agrupados para

construir grafos maiores e mais complexos (Figura 2.11). No entanto nem todos os grafos podem ser construídos, obrigando a correcção de todas as redes de streaming que podem ser geradas para que não existam deadlocks nem data-race. Seguem-se alguns exemplos válidos de combinações de pipeline e farm (e também feedback).

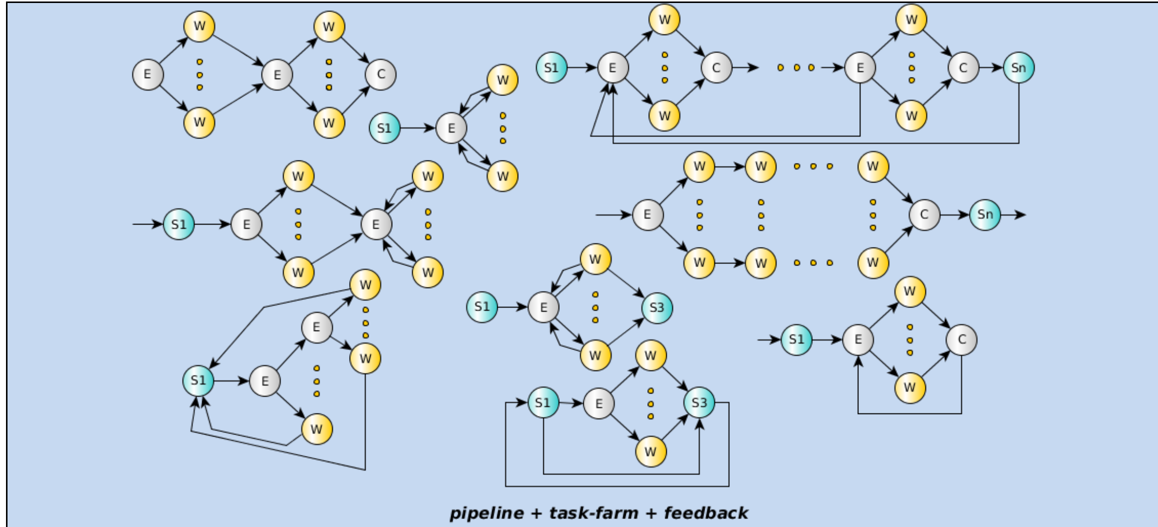


Figura 2.11: Exemplos de combinações válidas de pipeline e farm (e também feedback).

Já foi referida anteriormente a constituição dos padrões pipeline e farm. Seguem-se dois exemplos ilustrativos, um para cada padrão.

```
struct myNode1: ff_node_t<myTask> {
   myTask *svc(myTask *) {
      for(long i=0;i<10;++i)
         ff_send_out(new myTask(i));
      return EOS;
}};
struct myNode2: ff_node_t<myTask> {
   myTask *svc(void *task) {
      return task;
}};
struct myNode3: ff_node_t<myTask> {
   myTask *svc(void * task) {
      f3(task);
      return GO_ON;
}};
myNode1 _1;
myNode2 _2;
myNode3 _3;
ff_Pipe<> pipe(_1,_2,_3);
pipe.run_and_wait_end();
```

Listagem 2.6: Exemplo do uso do padrão pipeline.

Neste exemplo (Listagem 2.6), na primeira etapa são geradas 10 tarefas, usando a função `ff_send_out`, seguidas do EOS. A segunda etapa apenas devolve a tarefa recebida. Por último, a terceira etapa aplica a função `f3` a cada elemento da stream e não devolve nenhuma tarefa.

```
1   struct myNode: ff_node_t<myTask> {
2      myTask *svc(myTask * t) {
3         F(t);
4         return GO_ON;
5   }};
6
7   std::vector<std::unique_ptr<ff_node>> W;
8   W.push_back(make_unique<myNode>());
9   W.push_back(make_unique<myNode>());
10
11  ff_Farm<myTask> myFarm(std::move(W));
12
13  ff_Pipe<myTask> pipe(_1, myFarm, <...other stages...>);
14
15  pipe.run_and_wait_end();
```

Listagem 2.7: Exemplo do uso do padrão farm.

Neste exemplo (Listagem 2.7), os workers são ff_node(s) fornecidos por um std::vector. Se fornecermos diferentes ff_nodes ao vector facilmente conseguimos construir uma farm MISD (Multiple Instruction, Single Data) em que cada worker computa uma função diferente. Por norma, a farm tem um Emitter e um Collector, sendo que o Collector pode ser removido usando a função myFarm.removecollector(). O Emitter e o Collector podem ser redefinidos pelo programador, fornecendo os objectos ff_node adequados.

High-level patterns

A intenção desta camada é fornecer padrões flexíveis e reutilizáveis para resolver problemas paralelos específicos. Neste nível os padrões são apresentados como funções, tipicamente métodos pré definidos de uma classe que podem ser instanciados com uma função lambda do C++11. O seguinte exemplo, igual ao que já foi referido na secção 2.2.4 do OpenMP, mostra a semelhança entre paralelizar um ciclo `for` com OpenMP e FastFlow, usando padrões de alto nível:

```
//FastFlow

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

```
//OpenMP

#pragma omp parallel for
num_threads(workers)
for(long i=0;i<N;++i){
        A[i]+=1;
}
```

Listagem 2.8: Comparação entre o código gerado usando FastFlow e usando OpenMP.

Este primeiro padrão de alto nível pode ser usado para paralelizar ciclos com iterações independentes. Está implementado sobre o padrão task-farm com uma estraté-

gia de escalonamento adaptada. Apesar deste padrão ter uma sintaxe semelhante ao `parallel for` do OpenMP, a sua implementação é um pouco diferente.

Existe uma variante do `parallel for`, `parallel for reduce` (operação associativa). Esta variante executa uma redução local na parte do corpo do ciclo e uma redução final usando uma função para combinar os resultados (Listagem 2.9).

```
1  //codigo paralelo FastFlow
2  ParallelForReduce<double> pfr;
3  pf.parallel_for_reduce(sum, 0.0, 0,N,
4     [&A](const long i, double &sum) {
5        sum +=A[i];
6     },
7     [](double &sum, const double v) {
8        sum+=v;
9     }
10 );
```

```
1  //codigo sequencial: soma de
2  //todos os elementos de um arrray
3  double sum=0.0;
4  for(long i=0;i<N; i++)
5     sum += A[i];
6
7
8
9
10
```

Listagem 2.9: Exemplo de um parallel for reduce em comparação com o código sequencial.

Como podemos ver, são usadas duas funções lambda, a primeira para o corpo do ciclo e a segunda para a função de redução. As estratégias de escalonamento são as mesmas utilizadas no padrão parallel for.

2.4. *Exemplos de Processamento Paralelo de Imagens*

Nesta secção apresentam-se exemplos de utilização de computação paralela em processamento e reconstrução de imagens.

2.4.1. Paralelização do algoritmo de reconstrução LM OSEM

Ciechanowicz, Philipp, et al. [17] apresentou quatro implementações do algoritmo de reconstrução de imagens médicas LM OSEM (*List-Mode Ordered Subset Expectation Maximization*), semelhante ao MLEM explicado na secção 3.1, que utiliza a tecnologia PET (*Positron emission tomography*) para criação das imagens médicas. Duas destas implementações utilizavam as bibliotecas MPI, OpenMP ou Thread Building Blocks (TBB), enquanto que as outras duas usavam Algorithmic skeletons da biblioteca Muesli de Münster Skeleton. Foram comparados os tempos de execução, a eficiência e os estilos de programação. Os benchmarks foram executados num cluster multi-core, multi-processador com 20 nós cada um com dois processadores quad-core (AMD Opteron 2352, 2.1 GHz) e 32 GB de memória, com o sistema operativo Scientific Linux 5.2.

Normalmente uma reconstrução de uma imagem PET com dimensões $150 \times 150 \times 280$ processa aproximadamente $150 \times 150 \times 280$ eventos, demorando mais de duas horas para ser executado sequencialmente. Para resolver o problema, foram cria-

das duas abordagens. Na primeira, cada evento do subconjunto é processado independentemente, e denomina-se *projection space decomposition* (PSD). A segunda, denominada *image space decomposition* (ISD), baseia-se numa representação distribuída dos dados, sendo o seu tratamento realizado por um processo à parte.

Três das quatro implementações utilizam a abordagem PSD. A primeira implementação utiliza MPI e OpenMP (Figura 2.12), a segunda utiliza MPI e TBB e a terceira utiliza Muesli com *Task Parallel Skeletons* (Figura 2.13). A última implementação utiliza uma abordagem ISD e Muesli com *Data Parallel Skeletons* (Figura 2.14). Os detalhes de cada implementação não são aqui abordados (propositadamente).

```
1  for (l=0; l<numberOfSubsets; l++) {
2    events = readEvents(numberOfEvents,     // read fraction of subset
3                        offset);
4    #pragma omp parallel for private(path)
5    for (i=0; i<numberOfEvents; i++) {      // compute local c_l
6      ...
7      #pragma omp critical
8      for (m=0; m<path.length; m++)             // add (A_i)^t fp^-1 to local c_l
9        local_c[path[m].coord] += path[m].length / fp;  }
10   MPI_Allreduce(local_c, c, imageSize,    // compute global c_l
11                 MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
12   #pragma omp parallel for
13   for (j=0; j<imageSize; j++) {···}  }    // compute f_{l+1}
```

Figura 2.12: Implementação do algoritmo LM OSEM (PSD) usando MPI e OpenMP.

```
1  Initial<EventPacket> initial(getNextEvent);
2  Filter<EventPacket, Image> filter(processEvent);
3  Farm<EventPacket, Image> farm(filter, numberOfFilters);
4  Final<Image> final(updateImage);
5  Pipe pipe(initial, farm, final);
6  pipe.start();
```

Figura 2.13: Implementação do algoritmo LM OSEM (PSD) usando *task parallel skeleton*.

```
1  for (l=0; l<numberOfSubsets; l++) {
2    events = readEvents(numberOfEvents);    // read subset
3    for (i=0; i<numberOfEvents; i++) {      // compute c_l
4      path = computePath(events[i]);            // compute A_i
5      fp = f.foldIndex(&computeFp);             // compute fp = A_i f_l
6      c.mapIndexInPlace(&updateC);  }           // add (A_i)^t fp^-1 to c_l
7    f.mapIndexInPlace(&updateImage);  }     // compute f_{l+1}
```

Figura 2.14: Implementação do algoritmo LM OSEM (ISD) usando *data parallel skeletons*.

Resultados

Nas experiências efectuadas, para limitar o tempo de reconstrução, foram processados aproximadamente 10^6 eventos de um total de 6×10^7 eventos. Os eventos são divididos em 10 subconjuntos com 1 milhão de eventos cada. As implementações que usam MPI escalam praticamente da mesma forma com o aumento de nós no *cluster*. À medida que duplicamos o número de nós, obtemos um *speedup* médio de 1.8, baixando para 1.5 quando se duplica de 8 para 16 nós. A implementação baseada em OpenMP, obteve resultados ligeiramente melhores do que o TBB. Usando 8 threads, ambas as implementações obtiveram um *speedup* de 2.4. Este valor baixo em cada nó pode ser explicado pela existência de exclusão mútua na computação local de um valor (Figura 2.12, linha 7-11), que é usado para o próximo passo da iteração. Nas versões do Muesli, os valores são adicionados por um processo à parte, aumentando os requisitos de memória e comunicação do programa e diminuindo a penalidade causada pela exclusão mútua, como existe nos outros casos.

A implementação usando Muesli com *data parallel skeletons*, sofre um grande impacto no seu desempenho, causado pela elevada comunicação, sendo a implementação que obteve piores resultados. Por outro lado, a implementação usando Muesli com *task parallel skeletons* sofreu mais pela elevada necessidade de sincronização, ainda assim obtendo muito melhores resultados do que a implementação com data parallel skeletons.

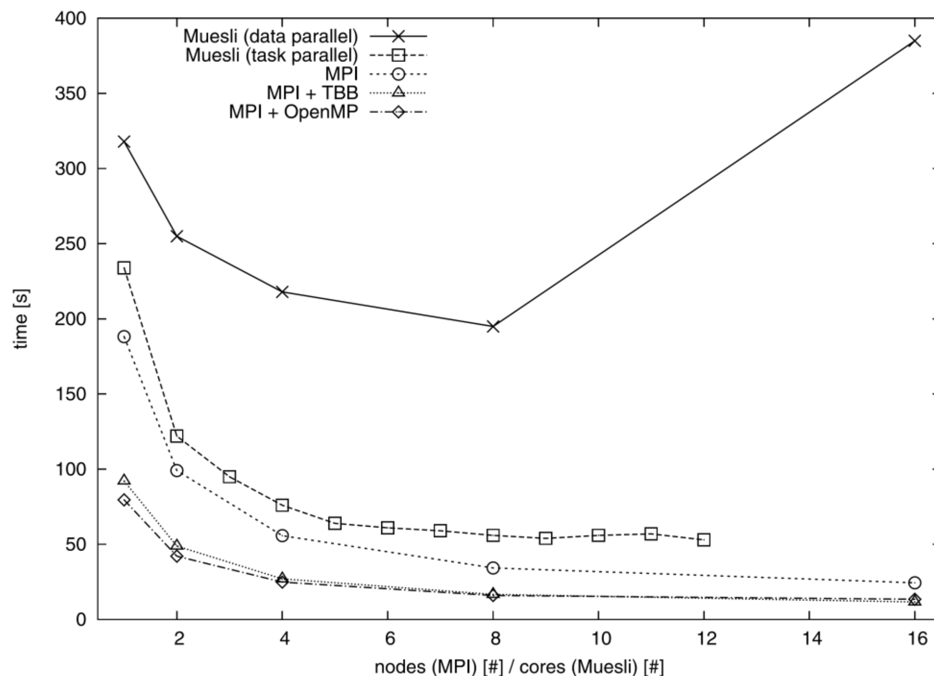


Figura 2.15: Tempos de execução da reconstrução de uma imagem PET [17].

Conclusões

Usando uma combinação de MPI e OpenMP ou TBB para programar multi-core clusters, permite implementar o algoritmo LM OSEM de forma eficiente, obtendo bons resultados. Contudo, este tipo de implementações requerem um nível de abstracção bastante baixo, exigindo um conhecimento detalhado de vários modelos de programação paralela.

Por outro lado, Muesli consegue paralelizar um programa sequencial sem grandes dificuldades. Como podemos verificar, o código gerado nestes dois programas, está muito bem estruturado e legível para qualquer um. Em ambos os casos, *task parallel* e *data parallel*, o nível de abstracção é muito mais alto, comparando com as outras implementações.

Esta experiência permite concluir que com o uso de skeletons é possível desenvolver programas simples, com uma abstracção mais alta, e que ainda assim obtenham um bom desempenho face aos programas mais complexos, com uma abstracção mais baixa. Se o programador desejar, também pode alterar um skeleton existente, para tentar obter os resultados desejados. Se a intenção é implementar programas rapidamente, o uso de skeletons é uma opção vantajosa, devido à sua simplicidade.

2.4.2. Reconhecimento facial usando um ambiente paralelo baseado em Algorithmic skeletons

Fatemi, Hamed, et al. [18] realizaram um estudo para avaliar o desempenho do uso de Algorithmic skeletons em aplicações de processamento de imagens. Para tal, implementaram dois algoritmos de reconhecimento facial numa plataforma paralela, em que o primeiro utilizava Algorithmic skeletons e o segundo foi optimizado da melhor maneira possível sem Algorithmic skeletons.

O reconhecimento facial pode ser dividido em duas fases principais, detecção e reconhecimento. Na primeira fase são detectados os tons da pele (caras), separando-as do resto da imagem como se fossem objectos. Na segunda fase, cada objecto é enviado para identificação numa base de dados com várias caras, sendo usado para este caso uma rede neural *Radial Basis Function (RBF)*. A razão pela qual se usou a rede neural RBF é pela sua capacidade de agrupar imagens semelhantes antes de as classificar.

Para a implementação do algoritmo, foram utilizados seis skeletons (Figura 2.16) e uma IMAP-board. Consultar paper para mais detalhes de implementação e referências sobre IMAP-board [18].

```
1  PixelToPixelOp(RGB, UV, &yc2ycbcr);
2  PixelToPixelOP(UV, skin, &Binarization);
3  ImageToObject(skin, obj, &labeling);
4
5  for(i=0; i < num_object; i++){
6      ObjectToObject(obj, hidden, &NeuralNet_hiddennode);
7      ObjectToObject(hidden, out, &NeuralNet_outputnode);
8      ObjectToValue(out, person, &Find_max);
9  }
```

Figura 2.16: Código para reconhecimento facial usando skeletons

Resultados

Para testar os algoritmos implementados, usaram uma imagem com tamanho 256×240 pixels e uma rede neuronal com 4608 nós de entrada, 15 nós escondidos e 6 nós de saída. Depois de efectuados vários testes, determinou-se que o algoritmo utilizando skeletons teve um tempo de execução 10% superior ao algoritmo optimizado. Com esta experiência, é de esperar que o uso de skeletons pode ser uma alternativa para este tipo de problemas, em que não obtemos um grande aumento no tempo de execução do algoritmo. Também liberta o programador das implementações de baixo nível e dos detalhes de paralelização.

Conclusões

Analisando o código fornecido na Figura 2.16, o facto de se optar por uma implementação recorrendo a Algorithmic skeletons, alivia o programador dos detalhes de paralelismo necessários para a implementação do algoritmo. A sua única tarefa é fornecer o código sequencial do algoritmo. Outra vantagem é que este nível de abstracção permite que estes tipos de programas possam ser executados em diferentes arquitecturas, apenas necessitando de alterar o código do skeleton para cada uma delas. Podiam ter realizado mais estudos com imagens de diferentes tamanhos, para verificar se o aumento do tempo de execução estava continuava a estar à volta dos 10%.

2.4.3. Processamento *ponto a ponto de imagens digitais usando computação paralela*

Olmedo, Eric, et al. [19] testaram o desempenho de vários algoritmos paralelos de processamento ponto a ponto de imagens digitais. Utilizaram as técnicas de *grayscale, brightening, darkening, thresholding* e *contrast change*. Todas estas técnicas aplicam, paralelamente, uma transformação num pixel de cada vez. Para tal, optaram por utilizar dois tipos de implementação, CUDA e OpenCV. Para os problemas em que a biblioteca OpenCV não disponibilizava as funções necessárias, implementaram as mesmas na linguagem C.

Foram implementados 16 algoritmos, 8 em CUDA, 3 em OpenCV e 5 em C. Os algoritmos implementados em C não sofreram um processo de optimização, o que pode justificar possíveis maus resultados no seu desempenho.

Nas implementações com CUDA, a imagem que vai ser processada tem que ser previamente transferida para a memória do GPU. Para a comparação dos tempos de execução, esta transferência não é tida em consideração.

Resultados e Conclusões

Utilizaram vários tamanhos de imagens, de maneira a poder verificar se os tempos de execução são proporcionais com o aumento do tamanho da imagem. As imagens utilizadas tinham 256×256, 512×512, 1024×1024, 1800×1400 e 4000×3000 pixels. Foi utilizado um desktop com AMD Phenom II Quad-core a 3.2GHz, 12 GB de RAM, sistema operativo 64-bit Linux Fedora 14 e versão OpenCV 2.3. Para o processamento com CUDA foi usada uma gráfica GeForce 430 GT com 96 cores e 1GB de RAM DDR3.

As transformações foram aplicadas em 10 imagens diferentes para cada resolução. A análise dos resultados permite concluir que em todas as técnicas, CUDA teve um melhor desempenho que OpenCV e C. Sendo que os resultados para OpenCV, estiveram muito próximos dos resultados de CUDA para resoluções mais baixas (funções de *grayscale, thresholding* e *negative*). Para imagens maiores, o CUDA apresenta vantagens em relação ao OpenCV.

2.4.4. Experiência efectuada (segmentação)

Esta experiência teve como objectivo principal a identificação de objectos em imagens. Apesar do contexto da implementação não estar directamente relacionado com o tema da tese, esta técnica é frequentemente utilizada no processamento de imagens médicas, surgindo algum interesse no estudo da sua implementação. É um bom ponto de partida para ganhar alguma experiência no processamento paralelo de imagens, nesta caso com o algoritmo de segmentação.

Para este exemplo em concreto, foram utilizadas imagens da área de Ciências dos Materiais. Estas imagens são capturadas através de raios-X, que deveriam conter apenas pixels pretos ou brancos, permitindo assim uma melhor compreensão da morfologia do material em análise. Porém, devido a questões físicas, existe a possibilidade dessa captura conter erros, resultando numa imagem com alguns pixels de cor indefinida. O objectivo deste processamento é remover esses erros e gerar uma imagem com pixels apenas pretos ou brancos.

Figura 2.17: Processamento de uma imagem (resolução 200x200x200) usando algoritmos de bi-segmentação (a) e histerese (b)

Podemos então dividir o processamento em três etapas:

1) É gerado um histograma de tons cinzento para identificar dois valores L1 e L2 para distinguir dois tipos de materiais numa amostra. Estes valores podem ser identificados pela visualização das curvas do histograma. O L1 representa os pixels mais próximos da cor preta correspondente a um tipo de material, e o L2 do branco, correspondente a outro material. Estes dois valores vão ser usados na segunda fase, servindo para transformar a imagem numa imagem com maior contraste entre os dois tipos de material.

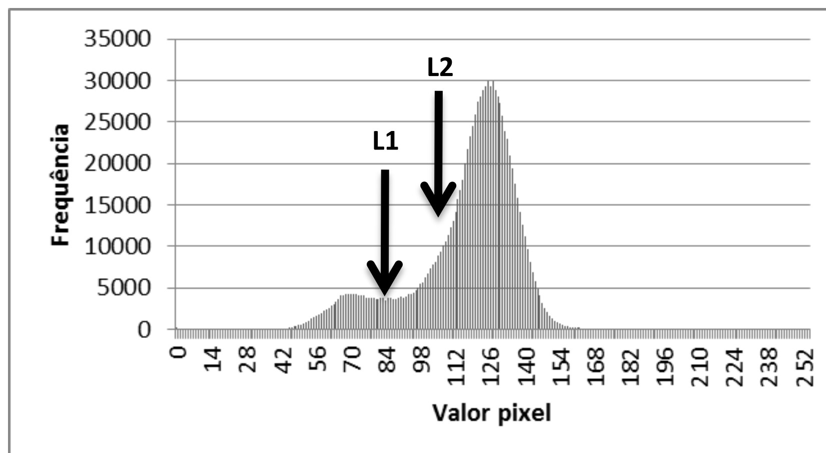


Figura 2.18: Histograma da imagem inicial em tons de cinzento

2) A partir dos valores L1 e L2, para todos os pixels da imagem com uma cor inferior ao valor L1 alteramos o seu valor para o pixel preto, para aqueles em que a sua cor tem um valor superior ou igual a L1 e inferior que L2 alteramos o seu valor para o pixel cinzento e para todos os outros alteramos para o pixel branco.

3) Partindo da imagem obtida na etapa 2, pretende-se chegar a uma imagem apenas com tons de preto e branco. Para cada pixel cinzento vamos avaliar a sua vizinhança, se existir maioria absoluta de pixels pretos, o seu valor muda para preto,

se a maioria forem brancos, o seu valor muda para branco, caso contrário mantém-se cinzento. Isto repete-se enquanto o número de cinzentos diminuir. Quando deixa de existir diminuição de cinzentos, em vez de mantermos a cor do pixel, opta-se por tomar a decisão baseada na maioria relativa de voxels pretos e brancos.

Implementação

O algoritmo foi implementado na linguagem C com a biblioteca *OpenMP*, que suporta primitivas para a programação paralela.

A distribuição da carga nas duas primeiras etapas pode ser feita dividindo uniformemente a imagem pelo número de processos, isto porque não existe qualquer dependência dos dados que são obtidos em paralelo e as operações executadas por cada processo são idênticas. Ou seja, para obter o melhor desempenho possível basta dividir o cubo por N threads. Por exemplo, para uma imagem com tamanho 100x100x100 pixels e com 4 threads, cada thread recebe 100x100x25 pixels. Podemos perceber melhor a partir da visualização da Figura 2.19.

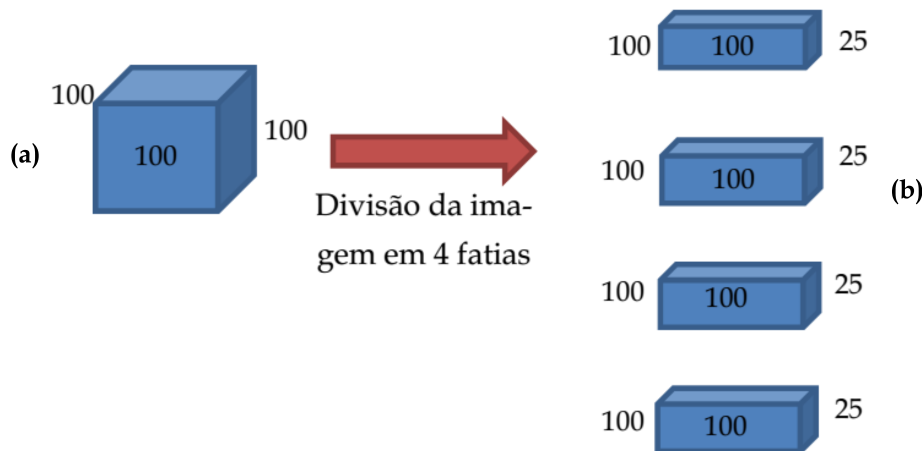


Figura 2.19: Distribuição da carga da imagem (a) pelos processos (b).

Para a última etapa, a mesma teoria não pode ser aplicada. Nesta fase não tratamos de todos os pixels, mas sim apenas daqueles que têm cor cinzenta. Se aplicássemos a divisão da Figura 2.19, poderíamos ter processos a terminar com tempos muito diferentes. Ainda assim, podemos comparar este processo ao da figura, só que para este caso, temos M fatias para N threads. Para minimizar este problema, a imagem é dividida em M partes (trabalhos), em que cada processo executa um trabalho de cada vez até não existirem mais trabalhos por executar. Desta forma, aumentando o valor de M conseguimos diminuir a perda de velocidade gerada por trabalhos muito grandes. O valor de M tem que ser inferior ou igual à altura da imagem. Para simplificar a explicação da implementação, vamos abordar cada etapa separadamente.

Etapa 1

Não foi dada muita importância à implementação do histograma, isto porque necessitávamos de visualizar o mesmo para escolher o melhor valor de L1 e L2, portanto optámos por uma implementação sequencial. Ainda assim, é muito importante paralelizar esta etapa para obter o melhor desempenho possível na geração do histograma. Não foi necessário para o nosso caso porque estamos a trabalhar apenas com uma imagem relativamente pequena (a maior imagem tem uma resolução de 400x400x400 pixels). A Listagem 2.10 mostra como é realizado o cálculo do histograma.

```
1  while(current_pos < size_of_image){ //para todos os pixeis da imagem
2         int pixel = image[current_pos]
3         histogram[pixel] ++; //aumentar a frequencia
4         current_pos++;
5  }
```

Listagem 2.10: Parto do algoritmo de geração do histograma (etapa 1).

Etapa 2

Nesta segunda etapa, cada uma das threads tem que processar uma fatia da imagem, sendo que o número de threads é passado como argumento pelo utilizador. É criado um apontador para a posição de memória onde a fatia começa, sendo realizada a bi-segmentação para cada um dos seguintes pixels até ao final da fatia. Na Listagem 2.11, podemos consultar o código que permitiu o cálculo da bi-segmentação.

Etapa 3

Nesta última fase foi implementado o algoritmo de histerese, em que foi criado um *parallel for* dinâmico para controlo da divisão do trabalho por cada thread. Basicamente funciona como uma lista de espera, em que cada thread, quando não está a executar nenhum trabalho, vai buscar o trabalho à lista. Para cada iteração, foi criada uma cópia da fatia da imagem obtida na Etapa 2. A primeira fase do algoritmo é aplicada nessa fatia, até não existirem mais mudanças de pixels. Quando já não há mais fatias por processar, passamos à segunda fase. A vizinhança utilizada para este processamento foi a mais simples possível, abrangendo apenas os 4 pixels adjacentes. Para obter melhores resultados poderíamos usar os 6 pixels adjacentes, adicionando assim o pixel da camada superior e inferior. A Listagem 2.12 ilustra a primeira fase desta etapa, sendo que a segunda fase é bastante semelhante, não existindo qualquer processamento iterativo.

```
1   //cada thread processa a sua fatia
2   //@image, imagem de entrada, em tons de cinzento
3   #pragma omp parallel shared(image) private(my_pixels,i) num_threads(n_threads)
4   {
5           i = omp_get_thread_num();
6           int start = width*height*plane/n_threads*i;
7           int end = (width*height*plane/n_threads);
8           my_pixels = &image[start]; //apontador para os pixeis da fatia
9           int k;
10          unsigned char current_char;
11          int counter = 0;
12          for(k = 0; k < end; k++){ //para todos os pixeis da fatia
13                  current_char = my_pixels[k]; //obter o pixel actual
14                  if(current_char < L1) current_char = 0; //preto
15                  else if(current_char < L2)current_char = 100; //cinzento
16                  else current_char = 255; //branco
17                  my_pixels[k] = current_char; //actualiza o valor do pixel
18          }
19  }
```

Listagem 2.11: Processamento de cada fatia por cada processo (etapa 2).

```
1   //schedule dinamico, quando uma thread acaba o trabalho, vai buscar o que esta no
    topo da queue de trabalhos
2   //@image, imagem de entrada, obtida na etapa 2
3   //@new_image, imagem de saida para a fase dois
4   #pragma omp parallel for shared(image, new_image) private(my_pixels,k)
    schedule(dynamic, 1)
5   for(k = 0; k < NUM_JOBS; k++){
6           int start = (plane/NUM_JOBS)*k;
7           int end = start + (plane/NUM_JOBS);
8           my_pixels = malloc(width*height*plane/NUM_JOBS);
9           //copiar fatia para iteração
10          memmove(&my_pixels[0], &image[width*height*start], width*height*(end-
    start));
11          //aplicar a primeira fase até nao haver mudança de pixels
12          while(firstPhase(my_pixels, width, height, plane, start, end) == 1){}
13          //copiar fatia para a imagem da fase dois
14          #pragma omp critical
15          {
16                  memmove(&new_image[width*height*start], &my_pixels[0], end-start );
17          }
18          free(my_pixels);
19  }
```

Listagem 2.12: Processamento de cada trabalho por cada processo (etapa 3, primeira fase)

Resultados e Conclusão

Os testes realizados mostraram um aumento bastante significativo do desempenho em relação à versão sequencial, como era de esperar. O principal objectivo desta experiência era ganhar alguma sensibilidade na implementação de programas paralelos de processamento de imagem, para perceber quais as dificuldades e desafios que surgem à medida que vamos avançando no problema.

3

Algoritmos Iterativos de Reconstrução de Imagem

Este capítulo descreve o trabalho realizado anteriormente no IBEB (Instituto de Biofísica e Engenharia Biomédica) na área de Digital Breast Tomosynthesis, no qual esta tese se baseia. Discutem-se em detalhe os algoritmos iterativos de reconstrução de imagem com particular atenção às implementações da geração da matriz de sistema.

3.1. *Reconstrução de Imagem*

Esta tese centra-se numa área particular do processamento de imagem: a reconstrução de imagens 3D de objectos, a partir de sequências de dados recolhidos em diferentes posições dos objectos. Um exemplo deste tipo de processamento ocorre na tomografia computacional (*computed tomography - CT)* usada em medicina, em que múltiplas projecções de uma parte do corpo de uma pessoa são usadas para produzir uma imagem 3D dos órgãos. A reconstrução de uma imagem a partir da informação obtida é um exemplo de problema inverso, onde tipicamente se fazem as perguntas ao contrário [6]. Por exemplo, dada a pergunta "Se nós sabemos a estrutura dos órgãos de um paciente, que tipo de imagens de raios X iriamos obter?" a mesma pergunta inversa é "Dado um conjunto de imagens de raios X de um paciente, qual é a estrutura tridimensional dos seus órgãos?".

A técnica de *filtred back projection (FBP)* é uma das técnicas analíticas mais utilizadas para este tipo de problemas [7]. Esta técnica é muito pouco exigente a nível computacional, permitindo a qualquer máquina moderna calcular este tipo de imagens em poucos milissegundos. Este método assume um cenário ideal, em que conseguimos obter uma amostra uniforme com muitas projecções, com pouco ruído e onde os raios

X são lineares, ou seja, a sua direcção é uniforme e em linha recta. O que acontece na realidade é que as projecções podem ser muito dispersas umas das outras, são adquiridas numa amplitude inferior a 180º e, podem ter algum ruído [8]. Para além disso, os raios X podem ser não lineares, ou seja, podem ser curvados, refratados, ou sofrer outra alteração. Como tal, podem existir situações que impossibilitam a resolução directa do problema. Neste caso, o algoritmo tem que aproximar a solução, o que pode causar o aparecimento de artefactos na imagem reconstruida. Os artefactos são alterações indesejadas provocadas pelo algoritmo.

Existe outra técnica para a reconstrução de imagens que consiste na utilização de algoritmos iterativos baseados em modelos estatísticos. Enquanto o algoritmo anterior calcula a imagem de forma analítica, este algoritmo iterativo utiliza modelos estatísticos para aumentar a qualidade da imagem obtida em cada ciclo até não existir alteração relevante entre sucessivas iterações. Esta técnica consegue reconstruir imagens com melhor resolução, menor ruído e menor número de artefactos, tendo também a vantagem de, em determinadas circunstâncias, ser possível reduzir a dose de radiação utilizada para a amostragem. A maior desvantagem desta técnica é a necessidade de grande poder computacional, o que dificulta a sua aplicação. A solução para este problema pode passar pela utilização de algoritmos paralelos que utilizam os melhores GPUs disponíveis no mercado [9].

Mais concretamente, esta tese está centrada na reconstrução de imagens associada à técnica *digital tomosynthesis* em particular a *Digital breast thomosynthesis (DBT)*. Este processo tem algumas semelhanças com CT mas é um pouco diferente. Em CT as projecções são obtidas numa rotação de 180º à volta da zona em análise, sendo assim possível efectuar a reconstrução da imagem [7]. Na DBT, o ângulo de rotação é bastante inferior, entre 15º a 60º, sendo também a exposição a radiação inferior para esta técnica. Como não existe um grande número de projecções, o algoritmo FBP tem que ser adaptado para diminuir o impacto causado pela amostragem incompleta. O mesmo não acontece utilizando o algoritmo iterativo, este tem a vantagem de conseguir lidar com várias situações, nomeadamente: número limitado de projecções; projecções com distribuição não uniforme no ângulo de rotação; maior dispersão entre projecções; falta de projecções em certas orientações; raios X não lineares.

Apesar de ser necessário um grande poder computacional para os algoritmos iterativos, estes apresentam melhores resultados do que os FBP, quer permitindo obter imagens de melhor qualidade com a mesma exposição do paciente aos raios X, quer qualidade semelhante com uma dose de radiação menor. No artigo [10] a Figura 3.1

confirma essa diferença, comparando imagens obtidas através de ressonância magnética e algoritmos iterativos.

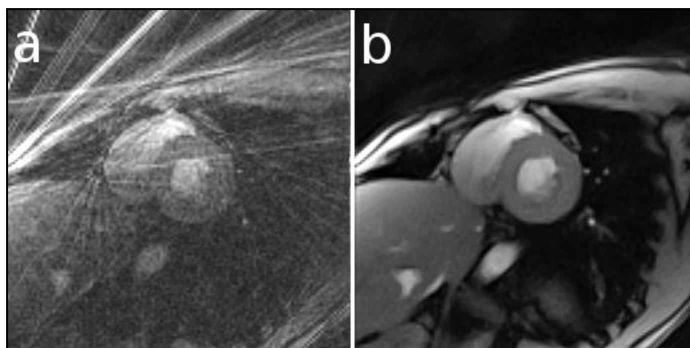


Figura 3.1: Imagem de um coração retirada de um vídeo de uma ressonância magnética. A) Reconstrução directa b) Reconstrução iterativa [10]

3.1.1. Reconstrução Iterativa

Para familiarização das técnicas utilizadas no algoritmo de reconstrução iterativa, a sua análise vai deliberadamente evitar os detalhes matemáticos. O objectivo é proporcionar uma compreensão bastante intuitiva de como funciona este algoritmo.

Todos estes métodos são iterativos, efectuando o mesmo ciclo de operações:

$$\boldsymbol{predict} \rightarrow \boldsymbol{compare} \rightarrow \boldsymbol{correct} \rightarrow \boldsymbol{predict} \rightarrow \boldsymbol{compare} \rightarrow \boldsymbol{correct} \ldots$$

Existem várias famílias de algoritmos de reconstrução de imagem [8], dos quais são relevantes para o trabalho os seguintes:

Métodos Algébricos

- *Algebraic Reconstruction Technique* (ART), SART, SIRT

Métodos Estatísticos

- *Expectation Maximization* (EM)
- *Maximum Likelihood Estimation* (MLE)
- *Maximum Likelihood and Expectation Maximization (ML-EM)*
- *Ordered Subsets – Expectation Maximization (OS-EM)*

Como este trabalho parte de um outro trabalho de reconstrução de imagem, segue-se uma explicação dos algoritmos de reconstrução iterativos SART, ML-EM e OS-EM, que foram usados no trabalho anterior. Para além disso, todos eles apresentam resultados satisfatórios a nível clínico. Segue-se uma breve explicação sobre estes três algoritmos em geral, seguida das características que os distinguem uns dos outros.

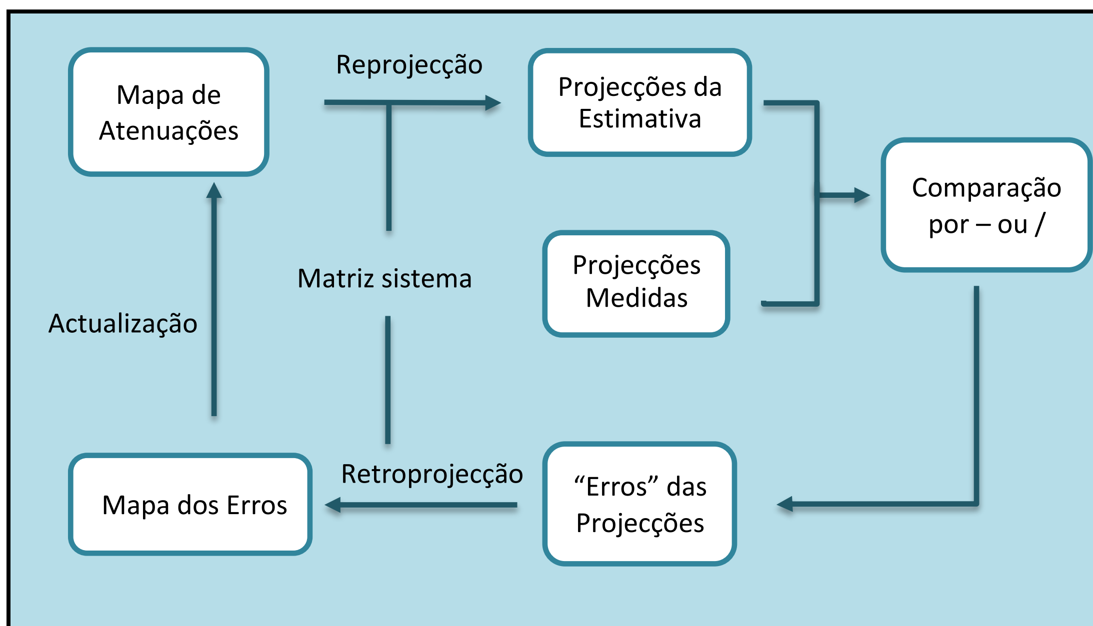


Figura 3.2: Esquema do ciclo de execução dos algoritmos de reconstrução para DBT (SART, ML-EM e OS-EM).

A Figura 3.2 é um exemplo de como se comportam a maior parte dos algoritmos de reconstrução iterativa.

Começamos com uma imagem inicial 3D estimada (Mapa de Atenuações), que ao fim de sucessivas transformações, reprojecção, retroprojecção e actualização, gera a imagem final, uma aproximação 3D do mama da paciente. É no início da reprojecção que necessitamos da matriz sistema. Esta matriz representa a atenuação dos diferentes órgãos, no voxeis, para um dado raio, e é utilizada para gerar as projecções da imagem 3D estimada. Todos os algoritmos usam os elementos da matriz sistema de forma diferente, o que se reflecte no tempo total de execução. A matriz sistema necessita de ser recalculada para cada exame que é realizado, porque a posição do foco pode ser ligeiramente diferente de exame para exame e a região de interesse (área ocupada pela mama) também pode alterar. Estas projecções são comparadas com as projecções reais que foram inicialmente medidas. Na fase da retroprojecção, vamos calcular o novo valor da imagem final, com a ajuda dos elementos da matriz sistema e o resultado da comparação entre as projecções. De seguida, é feita a actualização do voxel na imagem estimada.

A matriz sistema é sempre calculada da mesma forma para qualquer um dos métodos, sendo que a forma como os dados são utilizados diferem entre os métodos identificados anteriormente. Para qualquer método de reconstrução temos que ter sempre em conta o número de iterações, a posição do foco, que depende da potência do raio x,

o centro de rotação e o *binning* (se as células do detetor são agregadas) e a escala (qual o factor de agregação das células). Este factor de agregação tem um impacto directo na qualidade da imagem. Se aumentarmos este factor, a qualidade da imagem vai diminuir (porque o número de raios a ter em consideração vai ser inferior). Podemos ter algoritmos que convergem mais facilmente para o resultado final, diminuindo o número total de iterações. Isto é uma vantagem, visto que a matriz sistema tem que ser recalculada a cada ciclo (explicado na secção 3.4). Dizemos que estamos a convergir para a solução quando não existem muitas diferenças entre iterações sucessivas [11].

Uma forma de diminuir este tempo de reconstrução passa por utilizar a última matriz, que foi gerada na iteração corrente, na iteração seguinte. Isto poupa a geração de uma matriz por iteração, a partir da segunda iteração. Outras técnicas são específicas para cada método. Segue-se uma explicação breve sobre cada um deles.

3.1.1.1. "Algebraic Reconstruction Technique" (ART)

Para além de indicar o número de iterações e o binning, para este método temos que indicar o coeficiente de relaxação (lambda). Este valor vai impedir grandes oscilações no valor de cada voxel entre iterações consecutivas. A seguinte fórmula mostra como é calculado o valor de cada voxel para cada iteração.

Equação 3.1: Cálculo do valor de cada voxel (j) para cada iteração (k) na direcção (i).

$$f_j^{(k)} = f_j^{(k-1)} + \lambda^{(k)} \frac{Y_i - \sum_l a_{il} f_l^{(k-1)}}{\sum_l {a_{il}}^2} a_{ij} \qquad [27]$$

Cada uma das fases identificadas na Figura 3.2 está representada nesta fórmula da seguinte forma:

Reprojecção

$$f_j^{(k)} = f_j^{(k-1)} + \lambda^{(k)} \frac{Y_i - \boxed{\sum_l a_{il} f_l^{(k-1)}}}{\sum_l {a_{il}}^2} a_{ij}$$

É nesta fase que necessitamos da matriz sistema (a_{il}) pela primeira vez. Como podemos ver, o cálculo de cada raio pode ser efectuado de forma independentemente. Para cada raio multiplica-se o valor da imagem 3D estimada pelo valor da matriz sistema (podendo isto ser designado como projecções da estimativa).

Comparação:

$$f_j^{(k)} = f_j^{(k-1)} + \lambda^{(k)} \frac{Y_i - \sum_l a_{il} f_l^{(k-1)}}{\sum_l {a_{il}}^2} a_{ij}$$

Para este método a comparação consiste na subtracção dos valores das projecções reais (matriz Y) com as projecções da imagem estimada.

Retroprojecção:

$$f_j^{(k)} = f_j^{(k-1)} + \lambda^{(k)} \frac{Y_i - \sum_l a_{il} f_l^{(k-1)}}{\sum_l {a_{il}}^2} a_{ij}$$

A retroprojecção consiste no cálculo do novo valor que vai ser usado para actualizar a imagem 3D estimada. Este cálculo envolve o resultado da comparação e os valores da matriz sistema.

Actualização:

$$f_j^{(k)} = f_j^{(k-1)} + \lambda^{(k)} \frac{Y_i - \sum_l a_{il} f_l^{(k-1)}}{\sum_l {a_{il}}^2} a_{ij}$$

Nesta última fase, aplicamos o coeficiente de relaxação ao resultado obtido anteriormente e actualizamos o valor do voxel na imagem 3D estimada.

Dos três métodos referidos, o ART é o mais lento. Isto acontece porque existe dependência entre ângulos (i) quando estamos a realizar a actualização, o que torna mais difícil a sua paralelização. A seguir, podemos ver que o mesmo não acontece para o método ML-EM.

3.1.1.2. "Maximum Likelihood – Expectation Maximization" (ML-EM)

Como nos outros métodos, no ML-EM também temos que indicar o número de iterações e o binning. A fórmula seguinte mostra como é efectuado o cálculo de cada voxel, para cada iteração:

Equação 3.2: Cálculo do valor de cada voxel (j) para cada iteração (k) na direcção (i).

$$f_j^{(k)} = \frac{f_j^{(k-1)}}{\sum_{i'} a_{i'j}} \sum_i \frac{a_{ij} Y_i}{\sum_{j'} a_{ij'} f_{j'}^{(k-1)}} \qquad [27]$$

Podemos ver logo à partida que existe uma diferença no modo como os elementos da matriz sistema são utilizados. Cada uma das fases identificadas na Figura 3.2 está representada nesta fórmula da seguinte forma:

Reprojecção:

$$f_j^{(k)} = \frac{f_j^{(k-1)}}{\sum_{i'} a_{i'j}} \sum_i \frac{a_{ij} Y_i}{\boxed{\sum_{j'} a_{ij'} f_{j'}^{(k-1)}}}$$

Esta fase é semelhante ao método ART, em que cada projecção da imagem estimada é gerada a partir dos valores da matriz sistema multiplicados pelos valores da imagem 3D estimada.

Comparação:

$$f_j^{(k)} = \frac{f_j^{(k-1)}}{\sum_{i'} a_{i'j}} \sum_i \frac{a_{ij} \boxed{Y_i}}{\boxed{\sum_{j'} a_{ij'} f_{j'}^{(k-1)}}}$$

As diferenças começam a surgir aqui. No método ART a comparação era efectuada pela diferença das projecções reais com as projecções da imagem estimada. Neste caso é efectuada uma divisão, gerando o valor dos erros das projecções.

Retroprojecção:

$$f_j^{(k)} = \frac{f_j^{(k-1)}}{\sum_{i'} a_{i'j}} \boxed{\sum_i \frac{a_{ij} Y_i}{\sum_{j'} a_{ij'} f_{j'}^{(k-1)}}}$$

Com o valor dos erros das projecções gerados anteriormente, multiplicamos o valor da matriz sistema para o dado voxel j.

Actualização:

$$f_j^{(k)} = \boxed{\frac{f_j^{(k-1)}}{\sum_{i'} a_{i'j}} \sum_i \frac{a_{ij} Y_i}{\sum_{j'} a_{ij'} f_{j'}^{(k-1)}}}$$

Uma outra diferença é que a actualização pode ser efectuada em paralelo entre os ângulos, isto porque o valor final intermédio pode ir sendo calculado e gerado no fim depois de terem sido feitos todos os cálculos intermédios para todos os ângulos (i). Este factor diminui o tempo necessário para gerar a imagem final. Ainda assim podemos fazer melhor, usando o método OS-EM, que difere apenas como estes resultados parciais são gerados.

3.1.1.3. Ordered Subsets –Expectation Maximization (OS-EM)

Para este método necessitamos de indicar, para além do que foi indicado no ML-EM, o número de subsets utilizados. Este método é muito semelhante ao ML-EM, como podemos ver pela fórmula apresentada de seguida:

Equação 3.3: Cálculo do valor do voxel (j) para cada iteração (k) na direcção (i).

$$f_j^{(k)} = \frac{f_j^{(k-1)}}{\sum_{i' \in S(k)} a_{i'j}} \sum_{i \in S(k)} \frac{a_{ij} Y_i}{\sum_{j'} a_{ij'} f_{j'}^{(k-1)}} \qquad \textbf{[27]}$$

Uma das limitações do ML-EM reside no facto de ter uma velocidade de convergência muito lenta. Isto pode ser ultrapassado agrupando várias projecções e fazer a actualização das distribuições de actividade com base num dos sub-grupos (subsets S(k)).

A desvantagem é que existe um ponto de sincronismo entre cada actualização dos mapas de atenuações. A vantagem é que se actualizarmos o valor para um grupo de projecções, aproximamo-nos do valor esperado mais rápido, diminuindo o tempo de execução do algoritmo. Uma das formas de resolver o problema passa por criar vários conjuntos de projecções consecutivas. Esta abordagem vai melhorar a velocidade de convergência, mas não tão bem como desejamos, isto porque projecções consecutivas "vêem" o objecto de posições muito semelhantes, o que não permite ter uma boa noção do objecto. Uma forma de contornar esta limitação passa por criar os grupos em que as projecções têm grandes intervalos entre elas. Se escolhermos grupos com projecções muito próximas umas das outras, a imagem gerada vai conter apenas detalhes de uma parte do objecto irradiado. Se aumentarmos a distância entre as projecções, a imagem gerada vai conter detalhes de todo o objecto, permitindo assim uma "visão" maior do objecto a ser irradiado, melhorando a velocidade de convergência para uma solução ideal.

Esta reflexão permite concluir directamente que a construção da matriz sistema é o que tem maior influência no tempo total de execução do algoritmo. Vimos que para o ML-EM em comparação com o ART conseguimos diminuir o tempo de execução permitindo a paralelização entre ângulos, só que ainda assim a velocidade de convergência pode ser muito lenta. O método OS-EM permite uma velocidade de convergência mais rápida, diminuindo mesmo o número de iterações necessárias para a criação da imagem final.

3.2. *DBT System*

O sistema DBT usado no trabalho de Pedro Ferreira [1], no qual este trabalho se baseia, foi o Siemens MAMMOMAT Inspiration (Figura 3.3) que estava instalado no Hospital da Luz (Departamento de Imagiologia), Lisboa, Portugal.

A aquisição de imagens usando DBT é feita através de um tubo que emite raios-X e que percorre um arco de 50º à volta da mama da paciente (-25º a +25º). São obtidas 25 projecções num detector fixo, com um incremento aproximado de 2º por imagem. A partir destas 25 projecções é possível reconstruir imagens com 1mm de espessura.

A precisão geométrica do sistema DBT, enquanto se está a fazer o exame, é essencial para a sua reconstrução. Desta forma, o anglo de cada projecção é medido durante o exame para depois ser usado no algoritmo de reconstrução. Para a máquina usada, o centro de rotação fica 4.7 cm acima da superfície do detector e a distância entre o tubo de raio-X e o eixo de rotação é de 62.5 cm. O detector é um painel plano com tamanho de 2816 x 3584 células em que cada uma delas tem um tamanho de 85μm x 85μm, ocupando uma área de aproximadamente 24 $cm\ x$ 30 cm.

No trabalho de Pedro Ferreira foram utilizadas projecções de exames DBT realizados em mamas comprimidas a 60 mm. A imagem 3D reconstruida pretende-se que tenha 1mm de espessura em cada camada, por isso necessitamos de 60 camadas. Desta forma, podemos dizer que a região de interesse vai ser definida pelo volume irradiado pelos raio-X, sendo a sua dimensão definida por:

$$DetetorDetetor_{Dim_x} \times Detector_{Dim_y} \times NCamadas = 2816 \times 3584 \times 60$$

resultando em 605,552,640 voxels.

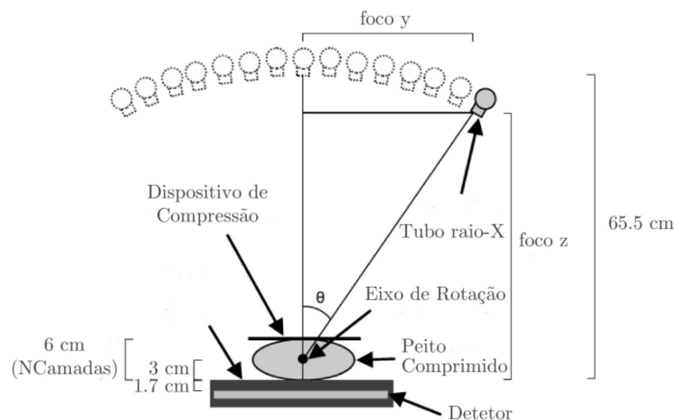


Figura 3.3: Diagrama do sistema Siemens MAMMOMAT Inspiration fazendo DBT. Adaptado de [1].

3.3. *Descrição da Implementação do Trabalho Existente*

O trabalho de Pedro Ferreira baseou-se numa implementação em IDL (Interactive Data Language [47]) para a reconstrução de imagens obtidas por DBT. Esta versão suportava vários algoritmos de reconstrução: SART, ML-EM e OS-EM. Todos os algoritmos iterativos estão de acordo com o esquema da Figura 3.2, em que a matriz sistema é calculada à medida que necessitamos de a usar.

Quando pensamos em paralelização podemos pensar em três fases principais do processo de reconstrução iterativo da DBT:

1. Cálculo da matriz sistema.
2. Reprojecção.
3. Retroprojecção.

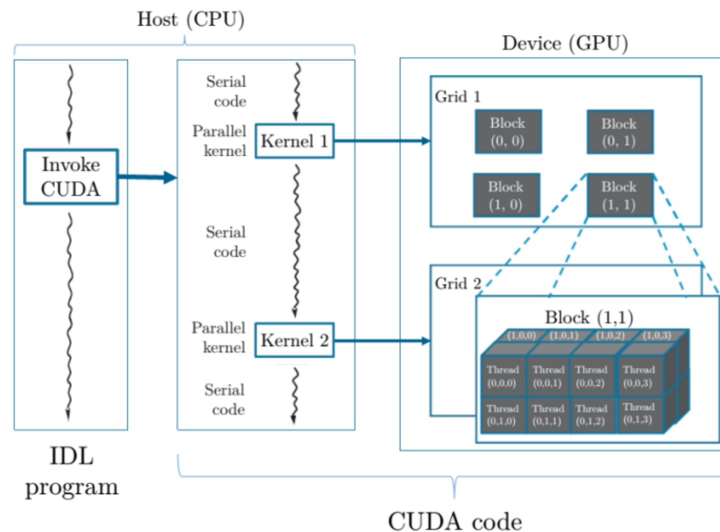


Figura 3.4: Execução de um programa IDL invocando CUDA [1].

O trabalho de Pedro Ferreira focou-se na paralelização do cálculo da matriz sistema usando CUDA, visto que esta fase era a computacionalmente mais exigente em relação às outras [1]. O seu trabalho teve como objectivo integrar esta nova implementação no código IDL existente, permitindo assim que o código CUDA fosse invocado a partir do código IDL (Figura 3.4).

3.4. *Cálculo da Matriz Sistema*

Tal como foi referido anteriormente, a matriz sistema descreve a atenuação dos voxeis para cada raio emitido pelo foco e recebido pelo detetor. A matriz sistema é in-

dependente do algoritmo de reconstrução iterativo e depende apenas da geometria do sistema DBT.

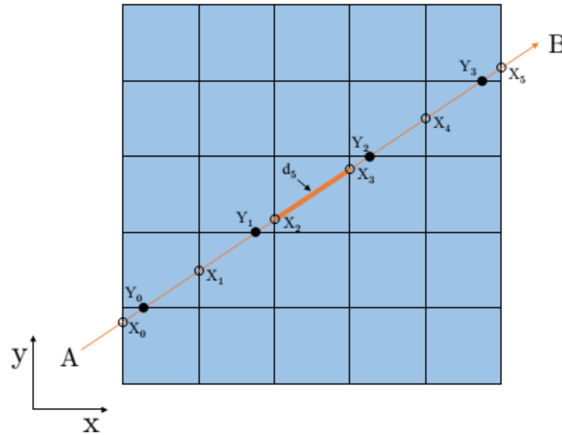


Figura 3.5: Exemplo da abordagem orientada ao raio no plano xy, em que o "y" representa o comprimento do detetor e o "x" a altura [1].

O cálculo da matriz sistema na implementação IDL usa uma abordagem orientada ao raio (*ray driven approach*). A construção da matriz tem como base o algoritmo descrito por Siddon em [34]. Existem outras alternativas mais eficientes para este cálculo, tendo sempre como base o algoritmo do Siddon, como por exemplo a técnica utilizada por Xiaolin Wu descrita em [35]. A abordagem usada na tese de Pedro Ferreira tem como base o algoritmo do Siddon. Esta abordagem calcula eficientemente a distância entre duas intersecções seguindo a trajectória de cada raio pelo volume da região de interesse. A Figura 3.5 ilustra um exemplo a duas dimensões (*plano xy*) da trajectória de um raio e as suas intersecções com as linhas horizontais e verticais.

A generalização para 3 dimensões é directa, em que o raio do ponto A (A_x, A_y, A_z) ao ponto B (B_x, B_y, B_z) pode ser representado pelas seguintes equações paramétricas

$$X(\alpha) = A_x + \alpha(B_x - A_x)$$

$$Y(\alpha) = A_y + \alpha(B_y - A_y)$$

$$Z(\alpha) = A_z + \alpha(B_z - A_z)$$

Equação 3.4 – Equações paramétricas do raio

em que $0 < \alpha < 1$, sendo 0 no ponto A e 1 no ponto B.

Esta abordagem consiste em desenhar raios a partir de cada célula do detector para o foco do raio X, determinando as intersecções de cada raio com a FOV (*Field Of View*). A FOV representa o espaço ocupado pela mama da paciente. A distância das

intersecções é calculada, onde após normalização representam a contribuição relativa de cada voxel para a atenuação do raio correspondente.

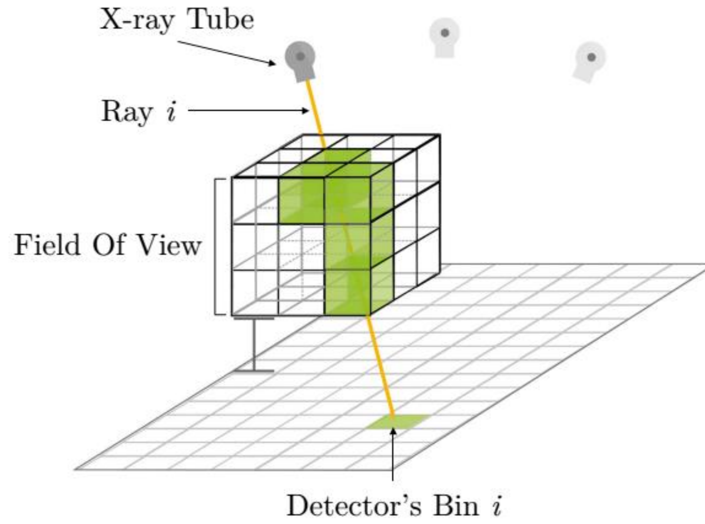


Figura 3.6: Ilustração de um raio i (Ri) de uma célula do detetor para o foco do raio-X, ilustrando que o raio é apenas atenuado pelos vóxeis por onde passa (vóxeis verdes)[1].

Para calcular as intersecções é necessário resolver a Equação 3.4. Para tal, temos que considerar que o ponto A é a posição da célula do detector e o ponto B é a posição do foco.

O ponto A é calculado da seguinte forma:

$$A_x = (n_x + 0.5) \times xbinsize$$

$$A_y = (n_y + 0.5) \times ybinsize$$

$$A_z = -17mm$$

em que $xbinsize = ybinsize = 85\mu m$ e $n_x, n_y \in \mathbb{N}$ em que $0 \leq n_x < Detector_{Dim_x}$ e $0 \leq n_y < Detector_{Dim_y}$. O ponto B é calculado da seguinte forma:

$$B_x = \frac{Detector_{Dim_x}}{2} \times xbinsize = \frac{2816}{2} \times 0.085mm$$

$$= 199.68$$

$$B_y = d_1 \times \sin(\theta) + \frac{Detector_{Dim_y}}{2} \times 0.085mm$$

$$= 625mm \times \sin(\theta) + \frac{3584}{2} \times 0.085mm$$

$$= 625mm \times \sin(\theta) + 152.32mm$$

$$B_z = d_1 \times \cos(\theta) + d_2$$

$$= 625mm \times \cos(\theta) + 30mm$$

Em que $d1$ é a distância entre o eixo de rotação e o tubo de raios X e $d2$ é a distância entre suporte da mama e o eixo de rotação. B_x é constante durante a aquisição DBT o que implica simetria geométrica Figura 3.7. Desta forma, não existe necessidade em calcular a parte direita da dimensão x da matriz sistema. Portanto, todos os raios desenhados a partir das células com coordenadas $(B_x \pm x_{celula}, y_{celula}, 0)$ são processados de igual forma.

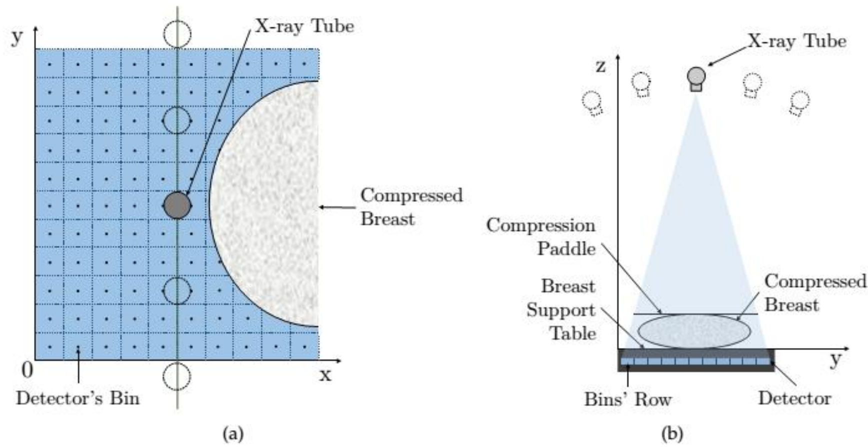


Figura 3.7: Posição do foco nos planos (a) xy e (b) yz [1].

Depois de efectuado o cálculo de A e B, já podemos encontrar as intersecções dos raios com os voxeis. Em primeiro lugar, assumimos que um raio intersecta sempre um eixo (x, y ou z) quando um ponto de um raio é igual a um número natural multiplicado pelo tamanho da célula na mesma direcção. De seguida, para cada coordenada de intersecção num eixo, calculamos o parâmetro α, que permite calcular as restantes coordenadas. Por exemplo, a partir da Equação 3.4 para as intersecções com a coordenada x temos:

$$\alpha = \frac{X(\alpha) - A_x}{B_x - A_x}$$

$$Y(\alpha) = A_y + \alpha(B_y - A_y)$$

$$Z(\alpha) = A_z + \alpha(B_z - A_z)$$

Onde efectuamos em primeiro lugar o cálculo de α, seguido de $Y(\alpha)$ e $Z(\alpha)$. O mesmo é feito para os restantes eixos. Neste momento temos 3 conjuntos, um para cada eixo, que representam as intersecções para um dado raio. Cada um destes conjuntos tem que estar organizado por ordem crescente para preservar a ordem das intersecções. É então necessário agrupar os conjuntos num só, em que dois termos adjacentes representam as intersecções para um voxel em particular. De seguida, consideramos as

dimensões da FOV para remover as intersecções que ocorrem fora da mesma (intersecções fora da FOV não contribuem para a imagem final porque não pertencem à área da mama da paciente). Por fim, depois de ter todas as intersecções que pertencem à FOV ordenadas, é efectuado o cálculo das distâncias entre intersecções consecutivas (N intersecções resultam em N-1 distâncias) em que cada distância é atribuída a um voxel.

Para determinar qual o voxel a que a distância está associada, consideramos que cada raio está na direcção da célula do detector para o tubo de raio X. O declive do raio pode ser positivo ou negativo e a coordenada do voxel é calculada de maneira diferente:

- Positivo: as coordenadas são obtidas subtraindo 1 às coordenadas da intersecção sempre que é um múltiplo do tamanho da célula, e arredondando (para baixo) caso contrário (Figura 3.8 (a));
- Negativo: todas as coordenadas são arredondadas (para baixo) (Figura 3.8 (b)).

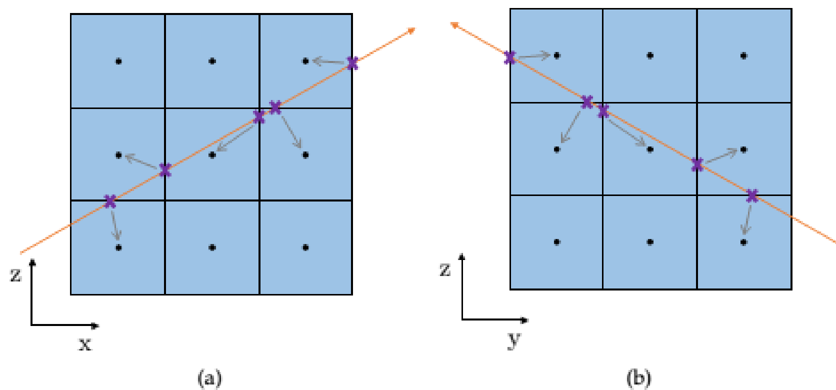


Figura 3.8: Atribuição do voxel às intersecções de um raio no (a) plano xy (que tem sempre declive positivo) e (b) plano yz (que neste caso tem um declive negativo) [1].

No fim obtemos a matriz com os valores da distância para cada voxel associado a uma célula. Para além disso, a matriz tem que ser recalculada a cada iteração do algoritmo para cada um dos ângulos devido às suas dimensões, podendo atingir um tamanho de aproximadamente 2GB, (o número de iterações depende do algoritmo utilizado), sendo que, como já foi referido anteriormente, é possível aproveitar a última matriz, da iteração corrente, na próxima iteração.

Existe uma grande necessidade de conseguir paralelizar estes algoritmos, que são bastante demorados e envolvem grandes volumes de dados. Existem esforços na área para paralelizar algoritmos deste género. Os mais promissores são os algoritmos iterativos, onde Dana Schaa et. al [36] aceleraram este tipo de algoritmos usando GPUs. Os

tempos obtidos foram bastante bons, permitindo afastar cada vez mais a preocupação do tempo quando se trata da aplicação clinica destes algoritmos.

Neste trabalho, o cálculo da matriz sistema, vai ser efectuado usando a framework FastFlow [32], que é capaz de produzir código CUDA [31]. Para tal, é necessário paralelizar o problema em CUDA. A intenção é avaliar a framework em relação à facilidade de programação e o seu desempenho, quando comparada com a opção mais popular: CUDA. Para além disso, esta framework também é capaz de produzir código para paralelização em CPU, semelhante ao popular OpenMP [29]. Desta forma será necessário paralelizar o problema usando OpenMP. Com estas implementações será possível então avaliar a framework na geração de código para CPU e GPU.

4

Desenho da Solução

Tal como foi referido anteriormente na secção 2.3.4, um dos âmbitos desta tese é estudar a usabilidade da ferramenta FastFlow na criação de programas que tiram partido do uso dos GPUs usando CUDA. Para tal, foram criados cinco programas distintos usando C++. Em primeiro lugar era necessário criar uma versão sequencial capaz de calcular a matriz sistema. Esta primeira versão foi a base para a criação das versões paralelas seguintes. Foram criadas duas versões paralelas, uma usando CPU e outra GPU, sendo que a versão CPU tirou partido da API OpenMP (Open Multi-Processing) e a versão GPU tirou partido da API CUDA (Compute Unified Device Architecture). Estas duas versões serão a base de comparação com os programas gerados pela ferramenta FastFlow, a qual gera código para CPU e GPU. Segue-se uma descrição do desenho de cada uma destas cinco implementações.

4.1. *Versão Sequencial*

Nesta primeira versão, o essencial é tentar replicar o que foi explicado na secção 3.4. A linguagem de base escolhida para a versão sequencial foi o C++ dado que é comum a todas as ferramentas utilizadas, incluindo framework FastFlow. C++ é uma linguagem de programação multi-paradigma muito utilizada em aplicações paralelas dado o seu desempenho, e que permite, por exemplo, uma manipulação baixo nível da memória do sistema [30].

Para cada iteração dos algoritmos iterativos (SART, ML-EM ou OS-EM), é necessário calcular 25 matrizes sistema, uma para cada ângulo do foco de raio-X. Podemos então dividir o cálculo de cada matriz em duas partes principais: cálculo das intersecções e cálculo das distâncias.

4.1.1. Cálculo das intersecções

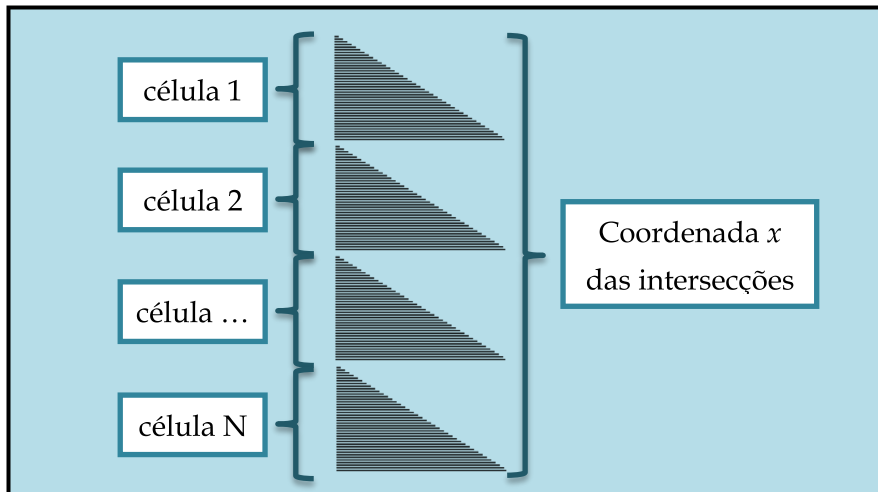


Figura 4.1: Intersecções ordenadas pela coordenada x seguida pela célula do detetor.

Nesta etapa, é necessário calcular as intersecções para cada célula do detetor, com as dimensões da região de interesse. O ciclo que percorre todas as posições do detector vai guardar todas as intersecções dos eixos x, y e z. A forma como as intersecções são calculadas foi explicada na secção 3.4. Todas as intersecções que forem encontradas fora da região de interesse são descartadas. Quando todas as intersecções tiverem sido calculadas, é necessário ordená-las pela coordenada x e de seguida pela célula do detetor. A Figura 4.1 ilustra como as intersecções estão ordenadas após esta fase.

Depois de estarem calculadas as intersecções e devidamente ordenadas, segue-se o cálculo das distâncias.

4.1.2. Cálculo das distâncias

Nesta altura temos N intersecções. Para cada par de intersecções, pertencentes à mesma célula do detetor, é calculada a distância euclidiana entre as duas. O voxel associado à distância é determinado como foi explicado anteriormente na secção 3.4. Neste programa sequencial, temos noção de quantas distâncias foram calculadas até ao momento, daí não existir grande dificuldade em saber como guardar os resultados finais. Um simples contador incremental pode determinar a posição final da distância no vector dos resultados. O mesmo não se aplica a programas paralelos, em que a posição de uma distância, calculada por uma thread, pode depender da posição de outra distância, calculada por uma outra thread. A maneira de contornar este problema é explicada na secção seguinte.

Este desenho foi baseado no trabalho de Pedro Ferreira [1]. À medida que se foi desenvolvendo o programa, surgiram pequenos detalhes sobre os dados que podem ser exploradas para melhorar os tempos do cálculo da matriz.

4.2. *Oportunidades de melhoramento*

1)

O cálculo das intersecções para uma célula do detetor pode ser visto como um raio que atravessa a região de interesse com origem no foco e destino no detetor, como foi visto na secção 3.4 (Figura 3.6). Actualmente são calculadas todas as intersecções possíveis, mesmo que estas estejam fora da região de interesse. Um raio, por norma, descreve o mesmo caminho, entra na região de interesse e a certa altura sai da mesma. As intersecções são calculadas por ordem, por isso, caso se obtenha uma intersecção fora da região de interesse, é possível deduzir que as próximas intersecções estão fora da mesma e que não vão ser utilizadas no resto do cálculo da matriz. Com este método, temos a vantagem de encurtar o ciclo do cálculo das intersecções de um eixo e de evitar calcular intersecções que mais tarde não iam ser utilizadas. Não estamos a adicionar carga ao algoritmo ao verificar se cada intersecção está dentro da região de interesse, porque isso teria que ser feito durante ou depois de calcular as intersecções.

2)

O segundo detalhe é que a carga imposta ao algoritmo de ordenação pode ser diminuída, onde a implementação anterior usava a biblioteca de ordenação do próprio C++. Em cada ciclo do cálculo das distâncias (cada ciclo é responsável por uma célula do detetor), se ordenarmos as intersecções da célula em questão, conseguimos obter na mesma o resultado da Figura 4.1. Isto porque, como sabemos a célula actual, podemos guardar os resultados ordenados dessa célula. Desta maneira o que fazemos é ordenar os resultados parciais para cada célula, sendo que no fim obtemos da mesma forma os resultados ordenados pela célula. Com este método, são ordenadas na mesma N intersecções, com a diferença de não ser necessário ordenar pela célula, isto porque as intersecções já são obtidas ordenadas pela célula (inicio na célula 0 até célula M, em que M representa o número de células).

3)

Após uma interpretação mais exaustiva na obtenção das intersecções para cada um dos três eixos (x, y e z), foi possível constatar que os resultados podem ser adquiridos de forma ordenada (por x, nomeadamente).

Tal como foi referido na explicação do cálculo da matriz sistema (secção 3.4), para cada célula, necessitamos de encontrar todas as intersecções para os três eixos (x, y e

z). Segue-se uma explicação sobre as características de cada plano que permitem esta ordenação por x.

Eixo X

Neste eixo, o que vamos incrementando é a coordenada X, se começarmos na célula do detetor e formos avançando para o foco, os resultados estarão sempre ordenados pela coordenada X (Figura 4.2, seta 1).

Eixo Y

A identificação da ordenação neste eixo é semelhante ao eixo X. Se a coordenada Y da célula do detetor for inferior à coordenada Y do foco, calculamos as intersecções incrementando a coordenada Y a partir da célula e com destino no foco (Figura 4.2, seta 1). Caso contrário, se a coordenada Y da célula do detetor for superior ou igual à coordenada Y do foco, calculamos as intersecções decrementando a coordenada Y a partir da célula e com destino no foco (Figura 4.2, seta 2). Em ambos os casos as intersecções são obtidas ordenadas pela coordenada X.

Eixo Z

Só necessitamos de calcular as intersecções para uma das metades da dimensão x, como foi explicado na secção 3.4. A metade escolhida é a que vai desde a posição $0 \rightarrow Detetor_{Dim_x}/2$. Desta forma, podemos obter as intersecções ordenadas começando na célula do detetor e incrementar a coordenada Z até chegar ao foco. Os resultados vêm ordenados pela coordenada X, como pode ser verificado pela seta 3 na Figura 4.2 b).

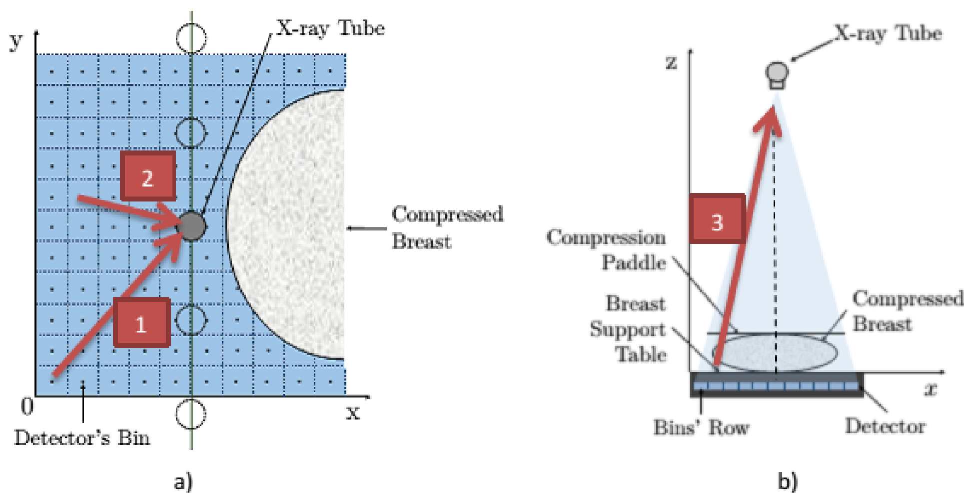


Figura 4.2: a) Posição do foco no eixo xy b) Posição do foco no eixo xz [1].

A maior parte dos algoritmos de ordenação comportam-se de maneira diferente para diferentes tipos de distribuição de dados. Os dados podem estar distribuídos aleatori-

amente, parcialmente ordenados, invertidos ou com poucas ocorrências únicas (Figura 4.3). É difícil escolher o algoritmo certo quando não sabemos como é que os dados vêm ordenados, por isso os algoritmos usados pela biblioteca do C++ têm que ser genéricos e estáveis, fazendo aproximadamente $N * log_2(N)$ (em que N é o número de elementos) comparações entre os elementos e trocas entre eles até esse valor [28].

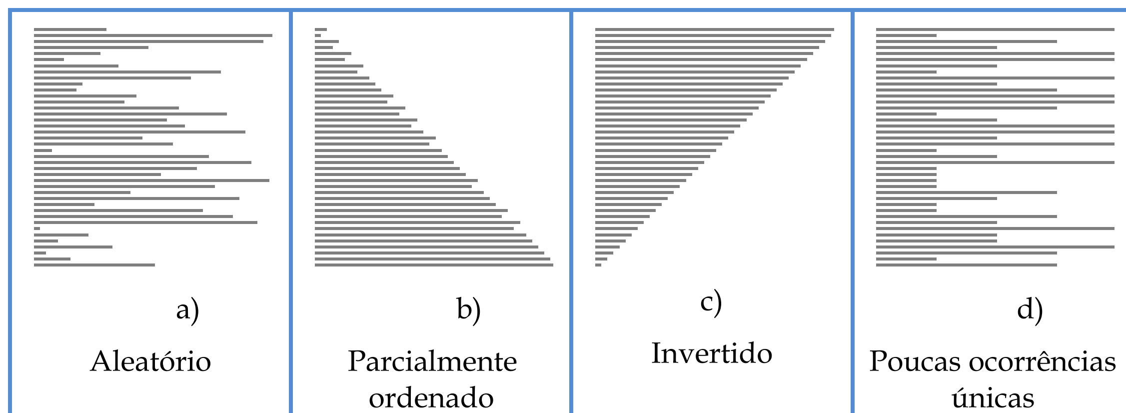


Figura 4.3: Alguns exemplos de distribuição de dados que podemos encontrar.

Mas para este caso, já constatámos que os dados podem ser obtidos parcialmente ordenados, ou seja, ordenados pela coordenada X em cada plano. Só que o objectivo final é ter todas as intersecções ordenadas pela coordenada X, independentemente do seu eixo. É então que surge a possibilidade de criar um algoritmo de ordenação adaptado para este problema.

4)

Algoritmo de ordenação adaptado

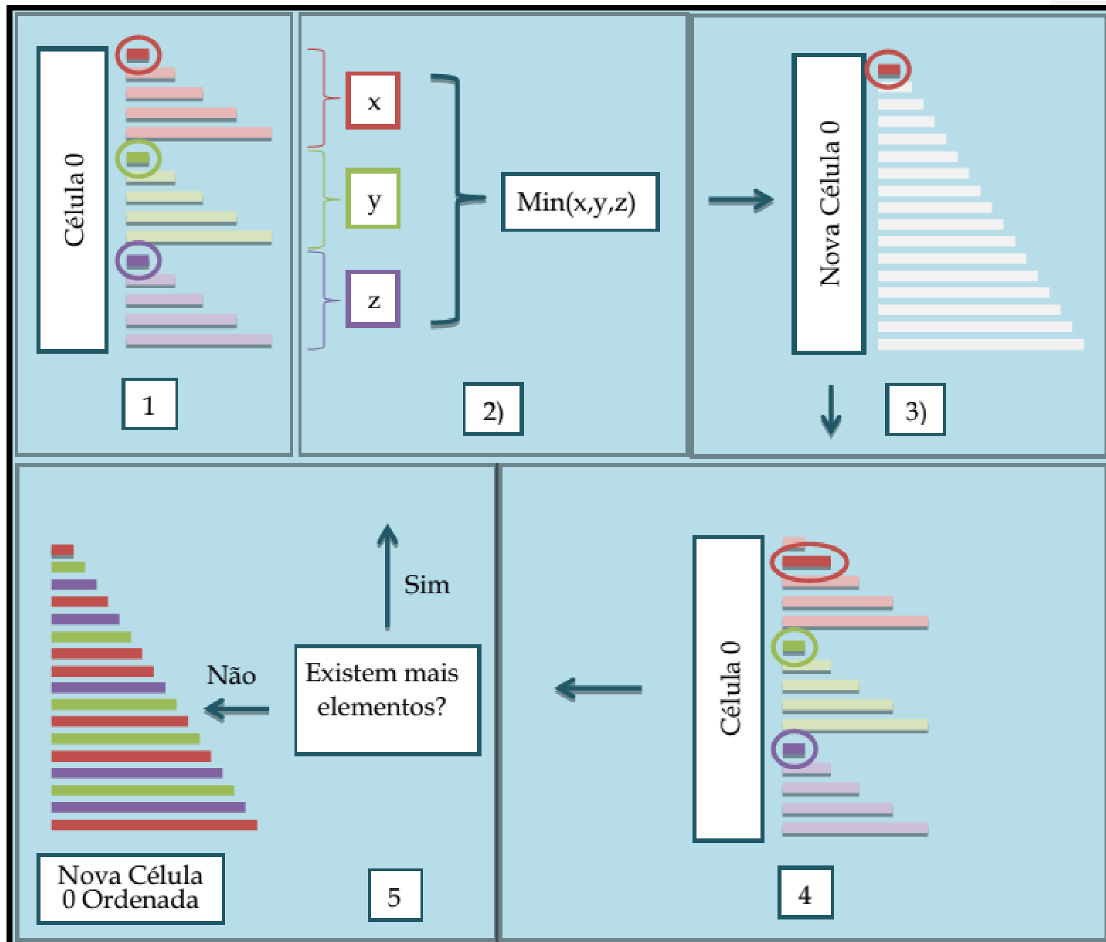


Figura 4.4: Ordenação das intersecções para uma célula.

Para conseguir aplicar este algoritmo, é necessário saber o número de intersecções em cada eixo. Já se sabe que as intersecções estão ordenadas pela coordenada X em cada eixo, assim o algoritmo pode ser dividido nas seguintes etapas (Figura 4.4):

1) Para cada eixo, escolher a intersecção com o valor da coordenada X mais baixo. Como as intersecções já estão ordenadas pela coordenada X basta escolher a primeira intersecção de cada eixo.
2) Escolher o eixo com o valor da coordenada X, da intersecção, mais baixo. Para tal, basta fazer o mínimo entre as três coordenadas (Min(x,y,z)).
3) O valor mínimo do eixo que foi escolhido na etapa anterior é copiado para uma nova célula (célula com a ordenação final).
4) Para o eixo que foi escolhido, se existirem mais intersecções, avançar para a próxima, caso contrário o plano já não é usado para a escolha do próximo valor mínimo.

5) Se ainda existirem elementos em pelo menos um eixo, voltar para o passo 2), caso contrário, obtemos a nova célula ordenada e termina o ciclo.

O que permite a utilização deste algoritmo é o conhecimento da estrutura dos dados para cada célula. Apesar do número de intersecções variar entre as células do detetor, a sua estrutura é sempre a mesma. Cada eixo (x, y e z) tem as intersecções ordenadas pela coordenada X. Os algoritmos de ordenação genéricos têm que funcionar para todos os casos e não tiram partido do conhecimento da estrutura dos dados. No máximo são realizadas N comparações e N cópias, em que N é o número de intersecções da célula (soma das intersecções do plano x, y e z).

A Figura 4.5 mostra o esquema do programa sequencial atendendo ao que foi dito anteriormente.

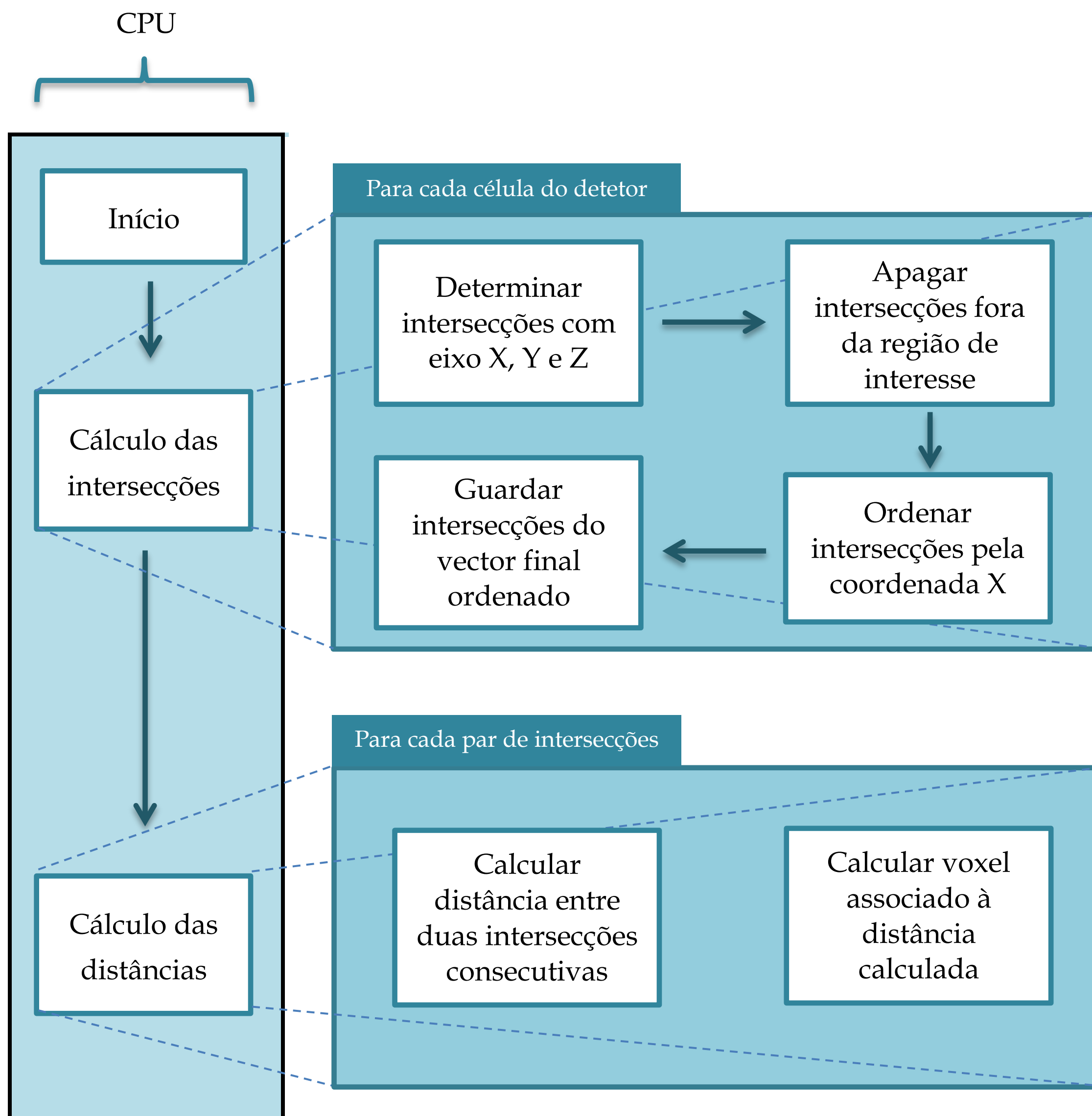


Figura 4.5: Esquema do programa sequencial executado num único core do CPU.

4.3. *Versão Paralela usando OpenMP*

Para gerar o código paralelo que calcula a matriz sistema, foi usando o código sequencial explicado na secção anterior, 4.1 Versão Sequencial. A paralelização é obtida distribuindo as células do detetor pelas threads. Como o número de threads é inferior ao número de células, o escalonador é responsável por dividir o trabalho total entre as threads existentes. Esta divisão depende da cláusula de escalonamento usada.

Com o simples uso da directiva `parallel for` é bastante fácil paralelizar os ciclos do programa sequencial obtendo bons resultados (Figura 4.6).

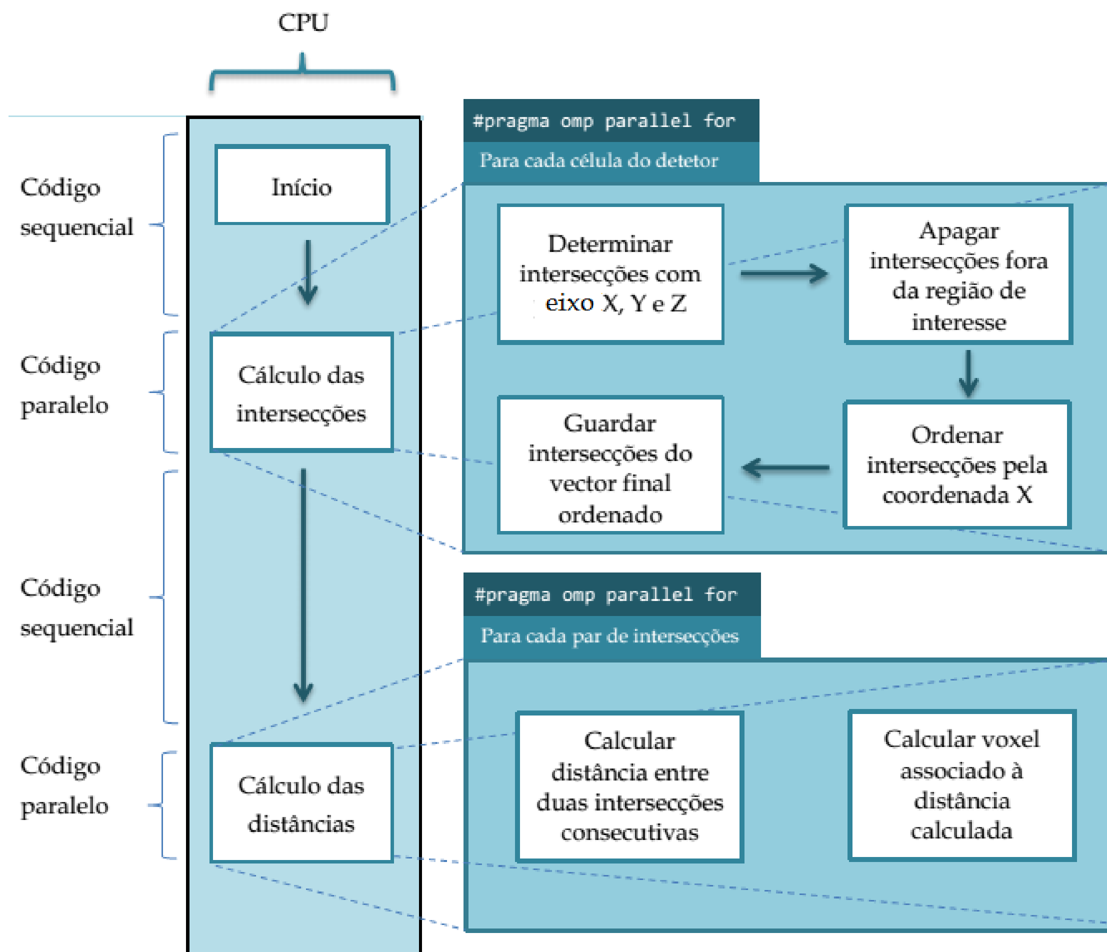


Figura 4.6: Esquema do programa paralelo baseado no esquema sequencial.

4.4. *Versão Paralela usando CUDA*

Nesta altura já é possível prever como é que será realizado o programa CUDA do cálculo da matriz sistema, tendo sempre em atenção como manipular os grandes volumes de dados envolvidos no cálculo da mesma. Como existem limitações na memória do GPU, é necessário adaptar a implementação à memória disponível. Isto implica

ter que calcular a matriz por partes, dividindo o tamanho do detetor tentando maximizar o uso do GPU em cada uma das partes. O capítulo da implementação explica como isto é feito. Sabendo isto, apenas temos que aplicar as técnicas de programação em CUDA para paralelizar os dois ciclos principais da geração da matriz (Figura 4.7). No caso do cálculo das intersecções, teremos uma *thread* para cada uma das células do detetor (relembro que apenas interessam as células de uma parte, visto que a matriz é calculada em várias partes). Para o caso do cálculo das distâncias, teremos $N - 1$ threads, isto porque a última intersecção não tem nenhum par associado para calcular uma distância, em que N é o número de intersecções obtidas no ciclo anterior (cálculo das intersecções). É necessário ter atenção às transferências de dados entre o *host* e o *device*, porque a existência de muitas transferências pode deteriorar bastante o desempenho do programa CUDA.

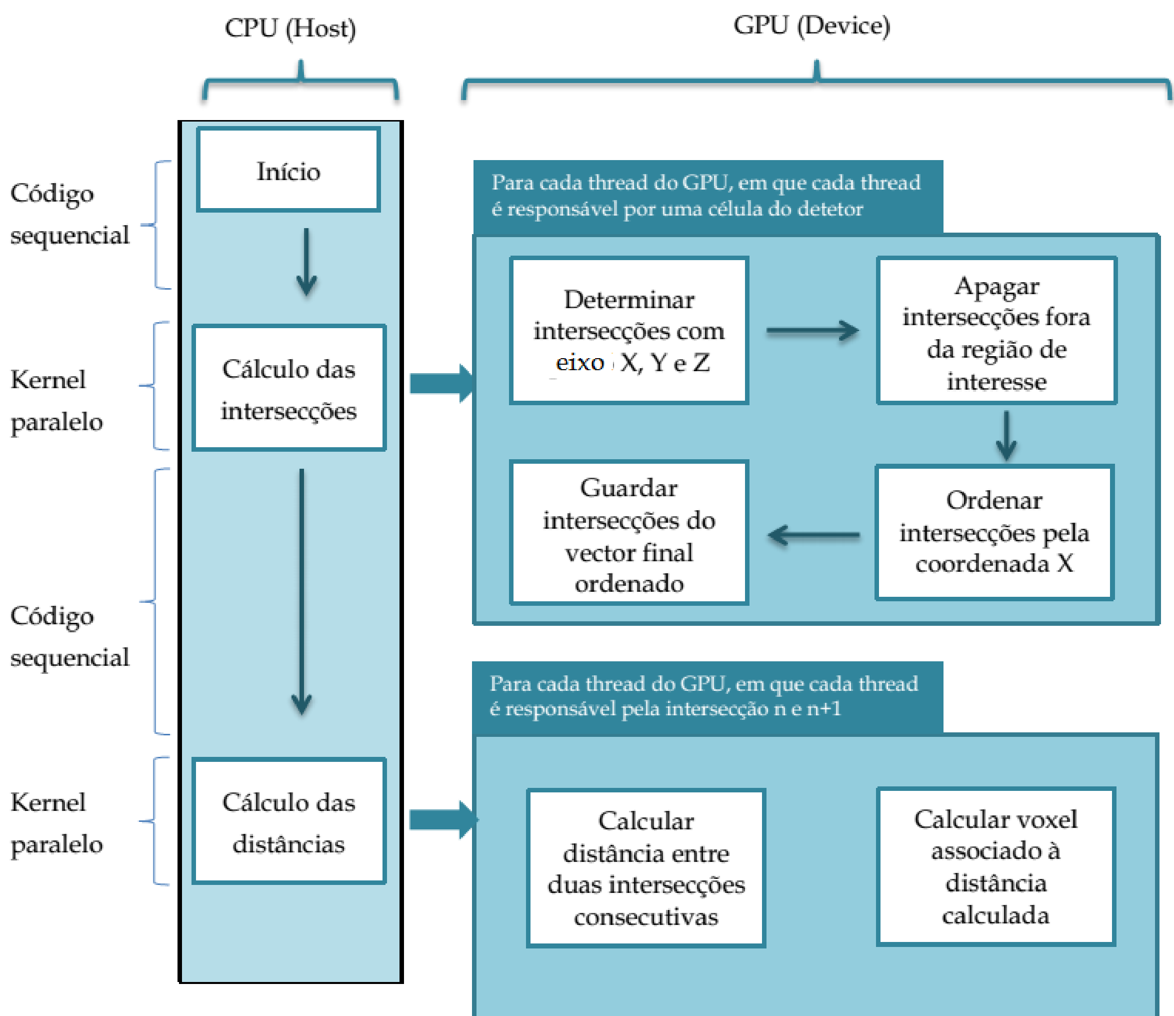


Figura 4.7: Esquema do programa paralelo (GPUs) baseado na versão sequencial.

4.5. *FastFlow – CPU*

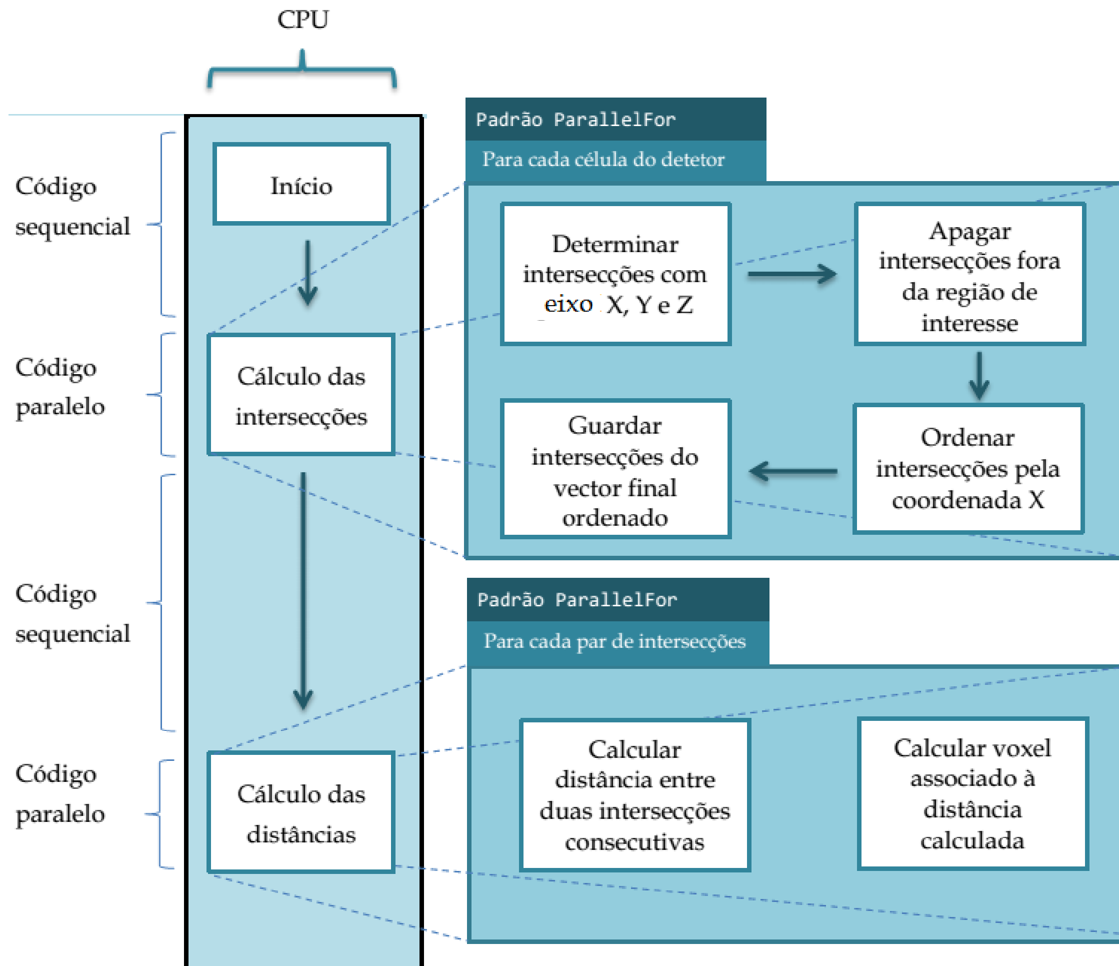


Figura 4.8: Esquema do programa FastFlow (CPU) baseado no esquema do OpenMP.

O esquema usado para a versão FastFlow (CPU) (Figura 4.8) é bastante semelhante à versão OpenMP (Figura 4.6). Na versão OpenMP é usado o **#pragma omp** parallel for para criar um ciclo paralelo. Na versão FastFlow é usado o padrão de alto nível ParallelFor, que permite paralelizar o corpo de um ciclo (usando funções lambda). Este padrão de alto nível é muito simples de usar, facilitando ainda mais a tarefa ao programador que não tem que se preocupar com os core-patterns. Para além disso já foi testado e comprovado obter bons resultados [33].

Existe ainda mais um padrão, Map. Este padrão é apenas um ff_node que serve para empacotar o padrão ParallelForReduce. Este padrão pode ser usado como etapa no pipeline e como worker numa farm.

4.6. *FastFlow – GPU*

High-Level Patterns

Execuções iterativas de kernels em GPGPUs são tratadas por um padrão bastante flexível `stencil-reduce` [32]. Este padrão está especialmente desenvolvido para algoritmos do tipo MapReduce. Acima deste padrão estão implementadas três variantes, map, reduce e map+reduce. Com estas variantes é possível desenvolver grande parte dos problemas que usam GPUs. Neste momento, o desenvolvimento destes programas pode ser feito escrevendo as funções em CUDA ou OpenCL. Isto permite ao programador usar as directivas CUDA/OpenCL no kernel, não sendo o seu uso obrigatório. O FastFlow encarrega-se das transferências de dados e sincronizações entre o Host e o Device e vice-versa.

```
1  //funcao do kernel
2  FFMAPFUNC(mapF, unsigned int, in, return in + 1;);
3  
4  //task que vai executar o kernel
5  class cudaTask: public baseCUDATask<>{
6  public:
7          void setTask(void* t){
8                  //define apontadores da task
9          }
10         ...
11         //funcoes que podem ser usadas antes ou depois do MapReduce
12 };
13 
14 int main(int argc, char * argv[]) {
15         ...
16         cudaTask *task = new cudaTask();
17         ...
18         FFMAPCUDA(cudaTask, mapF) *myMap = new FFMAPCUDA(cudaTask, mapF)(*task);
19         myMap->run_and_wait_end();
20         ...
21         return 0;
22 }
```

Listagem 4.1: Exemplo de um padrão mapa a ser executado para GPU (CUDA).

No exemplo da Listagem 4.1 podemos ver a base do desenho de programas para GPU usando a ferramenta FastFlow. Este exemplo em concreto usa o padrão map, que também foi o padrão usado para o algoritmo que calcula a matriz sistema. O kernel é criado usando a macro FFMAPFUNC, existindo várias macros que mudam consoante o número de parâmetros. É disponibilizada uma macro FFMAPCUDA que é responsável por criar o mapa para executar o kernel previamente definido. Para cada mapa é necessário identificar a tarefa CUDA que o vai executar, indicando os apontadores para as variáveis do Host e do Device.

O mesmo raciocínio que foi aplicado aquando do desenho da versão CUDA (Secção 4.4) foi agora aplicado para esta versão. Recorrendo à Figura 4.7, podemos ver que a diferença desta versão FastFlow vai ser na forma como o kernel CUDA é definido.

5

Implementação

Tal como foi indicado no capítulo anterior (Capítulo 4), foram criados 5 programas distintos em C++. A explicação da implementação de cada uma das versões é feita nesta secção. As grandes diferenças entre as várias implementações são encontradas no cálculo das intersecções e no cálculo das distâncias. Podemos ver a versão sequencial como a base, e que os programas paralelos derivam desta versão. Estes programas paralelos seguem um modelo de memória partilhada.

Neste capítulo vão ser explicadas as implementações das 5 versões: Sequencial; OpenMP; CUDA; FastFlow (CPU); FastFlow (GPU). Para todas as implementações são calculadas 25 matrizes (25 ângulos diferentes). Todas elas usam o mesmo método para calcular as intersecções no eixo x, y e z.

5.1. *Sequencial*

Para cada uma das células do detector são calculadas as intersecções com o eixo x, y e z. Para obter as intersecções ordenadas pela coordenada x, é necessário calcular as intersecções uma célula de cada vez. Como os resultados não são obtidos pela ordem desejada, estes são guardados num vector temporário que depois é copiado para o vector final, com os valores ordenados. A posição no vector final depende da célula em questão. Cada célula tem N posições reservadas no vector final, em que o N é igual ao número máximo de intersecções para uma célula. Desta forma é possível identificar a posição da célula no vector final, basta calcular o deslocamento para cada célula: $número_célula \times N$.

O cálculo das intersecções é resolver as equações apresentadas na secção 3.4. É bastante semelhante entre todos os planos, a única diferença encontra-se no ciclo e na

coordenada usada para calcular o *alpha*. Cada intersecção é caracterizada por ter uma coordenada x, coordenada y, coordenada z e a célula em que foi calculada (Figura 5.1).

```
1  struct intersection{
2    int bin;
3    float x, y, z;};
```

Figura 5.1: Estrutura da intersecção.

Cálculo das intersecções eixo X

```
1   for(x_integer = bin_x+1; x_integer <= detector_X/2; x_integer++){
2      float alpha;
3      intersection inter_x;
4      //Resolver a equação 2
5      inter_x.x = x_integer * x_bin_size; //coordenada X
6      alpha = (inter_x.x - (bin_x + 0.5) * x_bin_size) / slopevector_x;
7      //sabendo alpha, descobrir coordenada Y e Z
8      inter_x.y = (bin_y+0.5) * y_bin_size + alpha * slopevector_y; //coordenada Y
9      inter_x.z=distance_from_breat_support_table_to_detector+alpha*
    slopevector_z;//coordenada Z
11     inter_x.bin = bin_y + bin_x * detector_Y; //bin da intersecção
12     ...
```

Listagem 5.1: Parte do código para o cálculo das intersecções no eixo X.

O ciclo para o cálculo das intersecções com o eixo X é sempre crescente, começando na coordenada X da célula e terminando metade do detector (termina a metade devido à simetria no eixo X). O valor de *alpha* depende da coordenada X da intersecção, da coordenada X da célula e do valor de $B - A$ para o eixo X ().

Cálculo das intersecções do eixo Y

Semelhante ao eixo X, o eixo Y pode ter dois tipos de raios, crescentes e decrescentes. Se a coordenada Y da célula for superior à coordenada do foco então o ciclo é decrescente (com início na coordenada da célula), caso contrário o ciclo é crescente (com início na coordenada 0). O ciclo termina na posição do foco ou então nos limites do detector, caso a posição do foco exceda esses limites. O valor *alpha* depende da coordenada Y da intersecção, da coordenada Y da célula e do valor $B - A$ para o eixo Y (Equação 3.4).

```
1   while((y_int<= y_focus/y_bin_size && y_int<=detector_Y && crescente)
2      || (y_int>= y_focus/y_bin_size && y_int>0 && !crescente)){
3      float alpha;
4      intersection inter_y;
5      //Resolver a equação 2
6      inter_y.y = y_int * y_bin_size; //coordenada Y
7      alpha = (inter_y.y - (bin_y + 0.5) * y_bin_size) / slopevector_y;
8      //sabendo alpha, descobrir coordenada X e Z
9      inter_y.x = (bin_x + 0.5) * x_bin_size + alpha * slopevector_x; //coordenada X
10     inter_y.z= distance_from_breat_support_table_to_detector+
    alpha*slopevector_z;//coordenada Z
11     inter_y.bin = bin_y + bin_x * detector_Y; //bin da intersecção
12     ...
```

Listagem 5.2: Parte do código para o cálculo das intersecções no eixo Y.

Calculo das intersecções eixo Z

O ciclo para o cálculo das intersecções com o eixo Z é sempre crescente, correspondendo ao número de fatias da imagem final. O cálculo do valor *alpha* depende da coordenada Z e do valor B-A para o eixo Z (Equação 3.4).

```
1  for(z_integer = 0; z_integer <= N_slices-1 && stop_condition < 2; z_integer++){
2     float alpha;
3     intersection inter_z;
4     //Resolver a equação 2
5     inter_z.z = z_integer; //coordenada Z
6     alpha =(inter_z.z- distance_from_breat_support_table_to_detector) /
   slopevector_z;
7     //sabendo alpha, descobrir coordenada X e Y
8     inter_z.x = (bin_x+0.5) * x_bin_size + alpha * slopevector_x; //coordenada X
9     inter_z.y = (bin_y+0.5) * y_bin_size + alpha * slopevector_y; //cordenada Y
10    inter_z.bin = bin_y + bin_x * detector_Y; //bin da intersecção
11    ...
```

Listagem 5.3: Parte do código para o cálculo das intersecções no eixo Z.

Ordenação Biblioteca

Depois de calculadas as intersecções, estas estão guardadas num vector temporário que não se encontra ordenado. Para ordenar as intersecções, é utilizada a função do C++ `std::sort()` que ordena as intersecções pela coordenada X. De seguida, é calculada a posição da célula no vector ordenado final, para onde os valores são copiados. Esta posição é igual ao deslocamento, que é calculado através do número da célula e do número máximo de intersecções por célula.

```
1  inter_counter = inter_counter + tmp_vector.size();
2  std::sort(tmp_vector.begin(), tmp_vector.end()); //ordenar vector temporário
3  //inserir no vector principal
4  ...
5  int bin_vector_start = (bin_y) + (bin_x)*MAX_Y; //número da célula
6  int offset = bin_vector_start * (detector_X/2 + detector_Y +
   N_slices);//posição no vector final
7  unsigned int j;
8  for(j = 0; j < tmp_vector.size(); j++){
9     v[offset + j] = tmp_vector[j]; //copiar valores
10 }
```

Listagem 5.4: Código para a ordenação das intersecções usando a biblioteca do C++.

Nova Ordenação (proposta)

O código aqui apresentado ordena as intersecções como foi proposto no capítulo 4, secção 4.2. A comparação dos resultados pode ser consultada no capítulo 6.

```
//obter minimo
float min_x = vector[x_pos].x; //valor min plano X
float min_y = vector[x_counter + y_pos].x; //valor min plano Y
float min_z = vector[x_counter + y_counter + z_pos].x; //valor min plano Z
for(int k = 0; k < total_intersections; ++k){
        //descobrir min(min_x, min_y, min_z)
        int X_min = min(min_x, min(min_y, min_z));
        //copiar valor para vector final
        final_vector[offset + k] = X_min;
        //avancar contador do plano que foi escolhido
        if(X_min == min_x){ //plano X
                x_pos++;
                //alterar valor minimo do plano X
                //se existirem mais elementos
                if(x_pos < x_counter)
                        min_x = vector[x_pos].x;
                else
                        //nao ha mais elementos
                        min_x = MAX_VALUE;
        }
        else if(Y_min == min_x){
                ... //igual ao plano X
        }
        else if(Z_min == min_x){
                ... //igual ao plano X
        }
}
```

Listagem 5.5: Código responsável por ordenar as intersecções de um detetor e guardar no novo vector final ordenado.

Tal como na ordenação usando a biblioteca do C++, o objectivo é ordenar os resultados do vector temporário e copiar para o vector final. Nesta nova ordenação, em vez de se ordenar os valores todos e de seguida copiar para o vector final, os valores são copiados à medida que os elementos são obtidos por ordem. Desta forma, quando se descobre um valor mínimo entre os três eixos, esse valor é copiado para o vector final (ordenado). Quando já não existem mais elementos de um eixo, o valor mínimo desse eixo fica com um valor que é sempre superior à coordenada X das intersecções, isto para simular que já não existe nenhum elemento desse eixo, daí escolher um valor tão alto que nunca será seleccionado. Quando todas as intersecções tiverem sido copiadas para o vector principal, a ordenação está concluída. A Figura 5.2 ilustra um pequeno exemplo do algoritmo em execução. Este exemplo foi bastante simplificado para poder ilustrar o seu comportamento desde o início (vector não ordenado) ao fim (vector ordenado).

Figura 5.2: Exemplo da nova ordenação para um vector com 4 intersecções por cada eixo.

A alteração do número de intersecções por eixo não altera o comportamento do algoritmo, isto porque sabemos à partida quantas intersecções existem para cada eixo.

Cálculo das Distâncias

Para cada par de intersecções, que pertencem à mesma célula do detector, é calculada a sua distância euclidiana e de seguida é atribuído um voxel que vai estar associado à mesma. A atribuição do voxel segue o raciocínio apresentado na Figura 3.8. No final deste ciclo obtém-se um vector (matriz sistema) em que cada posição contém a distância, o seu voxel associado (normalizado) e a célula a que as intersecções estavam associadas (Figura 5.3).

```
1  struct matrix_value
2  {
3     int bin;
4     float distance;
5     unsigned int long xyz;
6  };
```

Figura 5.3: Estrutura dos elementos da matriz.

Neste código é possível ver como é que é calculado o voxel e como é que este é normalizado.

```
1  //distancia enclidiana
2  float distance = calculate_distance(first.x, first.y, first.z, second.x,
   second.y, second.z);
3  
4  //Atribuir distancia a voxel
5  //A partir das coordenadas das intersecções -> descobrir voxel correspondente
6  unsigned int long voxel_x, voxel_y, voxel_z;
7  //VOXEL X
8  if((second.x)/x_bin_size == round((second.x)/x_bin_size))
9     voxel_x = round((second.x)/x_bin_size) - 1;
10 else
11    voxel_x = (unsigned int long)((second.x)/x_bin_size);
12 //VOXEL Z (igual a X)
13 if(second.z == round(second.z))
14    voxel_z = (first.z) - 1;
15 else
16    voxel_z = (unsigned int long)(first.z);
17 //VOXEL Y (caso especial)
18 if(second.y < y_focus)
19    if((second.y)/y_bin_size == round((second.y)/y_bin_size))
20       voxel_y = (second.y)/y_bin_size - 1;
21    else
22       voxel_y = (unsigned int long)((second.y)/y_bin_size);
23 else
24    voxel_y = (unsigned int long)((second.y)/y_bin_size);
25 //valor normalizado
26 value.xyz = voxel_x + (unsigned int long)(voxel_y*detector_X) + (unsigned int
   long)(voxel_z*detector_X*detector_Y);
27 value.distance = distance;
28 value.bin = first.bin;
```

Listagem 5.6: Código que calcula as distâncias e o voxel associado à mesma.

5.2. *OpenMP*

O código, da versão sequencial, que calcula e ordena as intersecções e que calcula as distâncias foi reutilizado para esta versão com OpenMP, recorrendo às funções `calculate_intersections` e `calculate_distances`. Dito isto, só é relevante explicar como foi realizada a paralelização do código e que impacto é que isso teve no resto do programa. Usando o programa sequencial como base, é possível identificar que existem dois ciclos com boas oportunidades de paralelismo, cálculo das intersecções e cálculo das distâncias.

Cálculo das intersecções

Em vez de dividir o trabalho total pelo número de threads, o trabalho total é dividido em partes (NUM_JOBS). Isto é feito para diminuir o tempo em que as threads estão paradas por má distribuição de carga. Como cada célula tem um número diferente de possíveis intersecções, a carga nunca pode ser distribuída uniformemente. O número de trabalhos pode variar entre 1 e N, em que N é o número de células. Quanto mais próximo o valor está de N, menor é o tempo em que as threads estão paradas.

Para este caso foi usado um parallel for com uma estratégia de escalonamento dinâmica, ou seja, sempre que uma thread acaba um trabalho, vê se ainda existe mais trabalho para ser realizado. Esta estratégia permite diminuir o tempo que as threads estão sem trabalhar, diminuindo o tempo total do ciclo. No fim é feita uma redução (associativa) do contador de intersecções para cada trabalho realizado.

Em cada ciclo, a thread calcula o seu inicio e fim, sendo o tamanho do trabalho igual a $detector_X \times detector_Y \, / \, NUM_JOBS$.

```
1   //calcular interseccoes
2   #pragma omp parallel for shared(vector_inter, bin_intersection_count)
    schedule(dynamic, 1) reduction(+:inter_counter)
3   for(int k = 0; k < NUM_JOBS; ++k)
4   {
5      int start = k * ((MAX_X*MAX_Y) / NUM_JOBS);
6      int end = start + ((MAX_X*MAX_Y) / NUM_JOBS);
7      for(int xy = start; xy < end; ++xy){
8         int bin_x = xy / MAX_Y + x_iteration; //ymax
9         int bin_y = xy % MAX_Y + y_iteration; //ymax
10
11        calculate_intersections(bin_x, bin_y, x_focus, y_focus, z_focus,
12        detector_X, detector_Y, N_slices, x_bin_size, y_bin_size, vector_inter,
13        bin_intersection_count, inter_counter, x_iteration, y_iteration);
14     }
15  }
```

Listagem 5.7: Código da paralelização do ciclo do cálculo das intersecções.

Cálculo das distâncias

Há duas formas de encarar a paralelização deste ciclo. A primeira é criar um ciclo em que uma thread trata de um par de intersecções em cada iteração. A segunda é criar um ciclo em que uma thread trata das intersecções de uma célula. A primeira opção parece ser a mais indicada, isto porque podemos distribuir a carga de igual forma por todas as threads. O problema é que o número final de distâncias é igual ao $numero_interseccoes - numero_celulas$ isto porque N intersecções resultam em $N-1$ distâncias. O resultado do uso da primeira opção é num vector com valores nulos, que depois têm que ser procurados e retirados (implica alterar posições do vector final). A segunda opção possibilita saber qual a célula corrente e permite calcular facilmente o deslocamento para evitar ter valores nulos no vector final, daí ter sido a opção escolhida para este problema. O método de escalonamento é dinâmico, pelas mesmas razões apresentadas no cálculo das intersecções.

```
1   #pragma omp parallel for shared(matrix, bin_intersection_count) schedule(dynamic,
    1) reduction(+:total_distances)
2   for(int j = 0; j < MAX_X*MAX_Y; j++){
3      int bin_x = j / MAX_Y; //ymax
4      int bin_y = j % MAX_Y; //ymax
5      int n_elems = bin_intersection_count[bin_x][bin_y];
6      int start = bin_intersection_count_sum[bin_x][bin_y]-n_elems;
7      int bin_vector_start = bin_y + bin_x*MAX_Y;
8      int offset = bin_vector_start * (detector_X/2 + detector_Y + N_slices);
9      for(int i = 0; i <n_elems-1; i++){
10        intersection first = vector_inter[offset+i];
11        intersection second = vector_inter[offset+i+1];
12        matrix_value value;
13        calculate_distances(detector_X, detector_Y, N_slices, x_bin_size,
    y_bin_size, y_focus, first, second, value);
14        matrix[i + start] = value;
15        total_distances++;
16     }
17  }
```

Listagem 5.8: Código da paralelização do ciclo do cálculo das distâncias.

5.3. *CUDA*

O código, da versão sequencial, que calcula e ordena as intersecções e que calcula as distâncias foi em parte reutilizado para esta versão CUDA. Foram criados quatro kernels para esta versão. O primeiro kernel calcula as intersecções, o segundo kernel apaga todas as intersecções nulas, o terceiro kernel calcula as distâncias e o último kernel apaga as distâncias nulas. A razão pela qual existem valores nulos é porque o tamanho dos vectores é reservado para o pior caso (em relação à memória ocupada, neste caso o número máximo de intersecções/distâncias).

Kernel cálculo das intersecções

Para o caso do primeiro kernel, que calcula as intersecções, cada thread vai calcular as intersecções de uma dada célula e quando terminar vai registar num contador o número de intersecções encontradas. As intersecções são ordenadas e guardadas num novo vector, tal como na versão sequencial apresentada anteriormente. O que é importante nesta versão é como é que o paralelismo é obtido, e que neste caso uma thread ocupa-se de calcular as intersecções para uma célula. Por restrições de memória, a matriz final tem que ser calculada por partes. Para tal, é necessário ter variáveis que indicam a iteração corrente. O seguinte código (Listagem 5.9) é bastante intuitivo depois das explicações apresentadas sobre a linguagem CUDA.

```
1   __global__ void calculate_intersection_kernel(int total, float detector_X, float
2   detector_Y, float N_slices,float x_focus, float y_focus, float z_focus, float
3   x_bin_size, float y_bin_size, int n, int x_iteration, int y_iteration,
4   intersection *dev_vector_inter, unsigned int *counters, intersection *tmp){
5      int tid = threadIdx.x + blockIdx.x * blockDim.x; //id da thread
6      if(tid < total){
7         int bin_x = tid / MAX_Y + x_iteration; //coordenada X da célula
8         int bin_y = tid % MAX_Y + y_iteration; //coordenada Y da célula
9         //calculo das interseccoes, como na versao sequencial
          unsigned int counter = cuda_calculate_intersections(tid, bin_x, bin_y,
10        x_focus, y_focus, z_focus, detector_X, detector_Y,N_slices, x_bin_size,
11        y_bin_size, x_iteration, y_iteration, dev_vector_inter, tmp);
12        counters[tid] = counter;
13     }
14  }
```

Listagem 5.9: Código do kernel responsável por calcular as intersecções.

Kernel apagar zeros das intersecções

Em vez de se percorrer o vector todo à procura de intersecções nulas, é possível transferir os resultados directamente de um vector para outro, sendo que o novo vector não vai conter nenhuma intersecção nula. Cada thread sabe quantas posições é que tem que andar para trás para eliminar todas as intersecções que não foram aproveitadas. É possível saber esse valor porque este kernel recebe a soma acumulada das intersecções (entre cada bin). A seguinte figura explica bem como isso é feito (Figura 5.4).

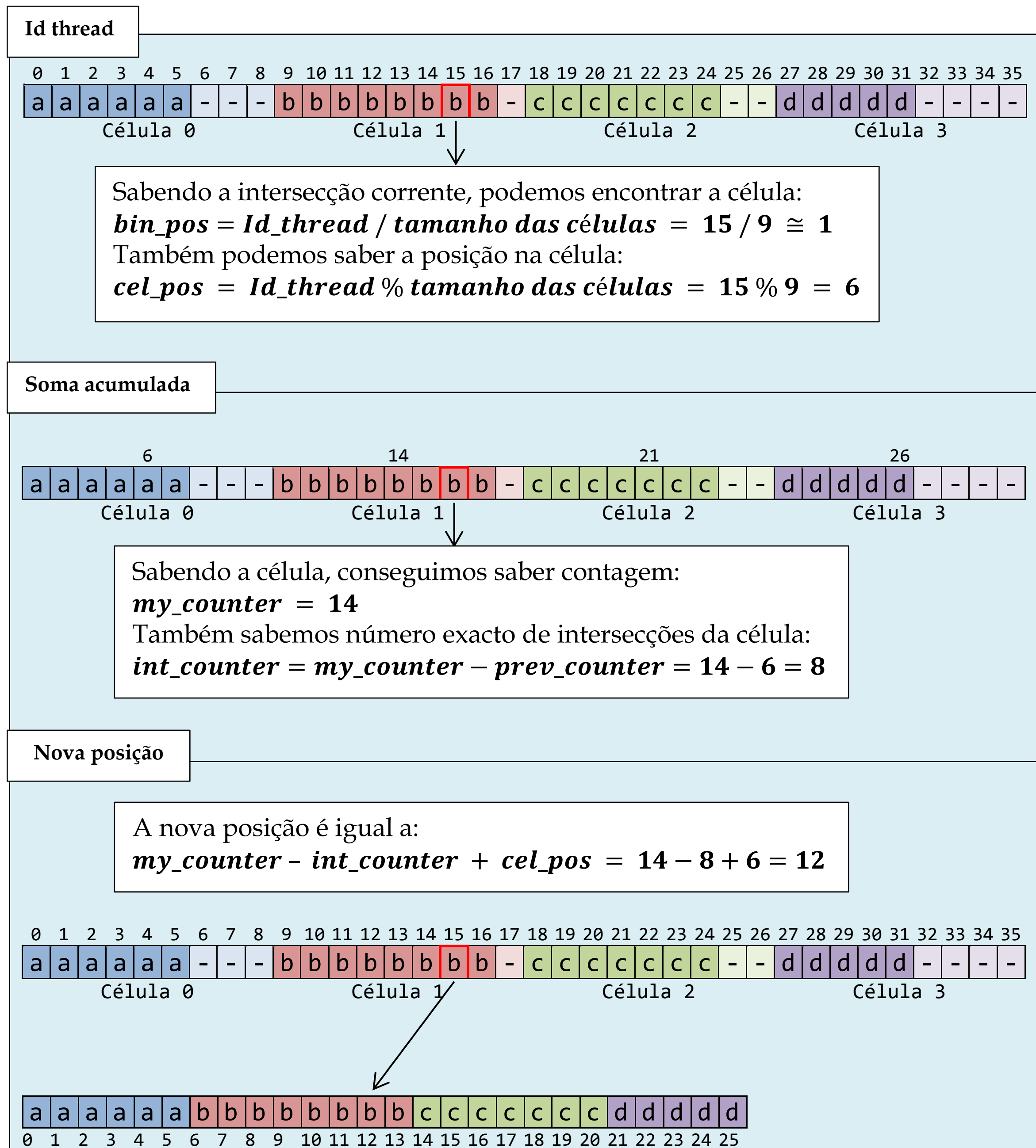


Figura 5.4: Esquema para remoção das intersecções nulas.

Kernel calcular distâncias

Este kernel é abordado de uma maneira um bocado diferente à que tinha sido usada até agora. Neste caso, cada thread é responsável por um par de intersecções, e não por um grupo de intersecções, como tinha sido até agora. O cálculo da distância e do voxel é igual ao que já foi apresentado anteriormente na versão sequencial. No final deste kernel, os valores estão guardados no vector `matrix`, só que ainda existem valores nulos, que necessitam de ser apagados (o próximo kernel mostra como isso é feito).

```
1   __global__ void calculate_distances_kernel(unsigned int num_intersections, float
2   detector_X, float detector_Y, float N_slices, float y_focus, float x_bin_size,
3   float y_bin_size, int n, intersection *dev_vector_inter, matrix_value *matrix,
4   int*stencil){
5      int tid = threadIdx.x + blockIdx.x * blockDim.x;
6      if(tid < num_intersections){
7      intersection first = dev_vector_inter[tid];
8      intersection second = dev_vector_inter[tid+1];
9      if(first.bin == second.bin)
10     {
11        float x = second.x - first.x; float y = second.y - first.y; float z =
    second.z - first.z;
12        float distance = x*x + y*y + z*z;
13        distance = sqrt(distance);
14        //Atribuir distancia a voxel
15        //A partir das coordenadas das intersecções -> descobrir voxel
    correspondente
16        unsigned int long voxel_x, voxel_y, voxel_z;
17        //VOXEL X
18        if(abs((second.x)/x_bin_size - round((second.x)/x_bin_size)) < 0.0001)
19           voxel_x = round((second.x)/x_bin_size) - 1;
20        else
21           voxel_x = (unsigned int long)((second.x)/x_bin_size);
22        //VOXEL Z (igual a X)
23        ...
24        //VOXEL Y (caso especial)
25        if(second.y < y_focus)
26           if(abs((second.y/y_bin_size) - round((second.y)/y_bin_size)) < 0.0001)
27              voxel_y = round((second.y)/y_bin_size) - 1;
28           else
29              voxel_y = (unsigned int long)((second.y)/y_bin_size);
30        else
31           voxel_y = (unsigned int long)((second.y)/y_bin_size);
32        matrix[tid].xyz = voxel_x + (unsigned int long)(voxel_y*detector_X) +
    (unsigned int long)(voxel_z*detector_X*detector_Y);
33        matrix[tid].distance = distance;
34        matrix[tid].bin = first.bin;
35
36        }
37     }
38  }
```

Listagem 5.10: Código do kernel que calcula das distâncias da matriz.

Kernel apagar zeros das distâncias

Como as únicas distâncias que vão estar a zero são aquelas em que o par de intersecções não pertence à mesma célula, sabemos que entre células vamos sempre ter uma distância que não tem valor. Este kernel elimina essa distância, em que cada thread atrasa uma distância N posições, sendo que N é igual ao número da célula actual.

```
1  __global__ void delete_zeros_distance_kernel( matrix_value *dev_matriz,
2  matrix_value *dev_new_matriz, int *positions, unsigned int n){
3     int tid = threadIdx.x + blockIdx.x * blockDim.x;
4     if(tid < n){
5        dev_new_matriz[tid] = dev_matriz[positions[tid]];
6     }
7  }
```

Listagem 5.11: Código do kernel que elimina os zeros das distâncias.

5.4. *FastFlow – CPU*

Para além de se usar funções lambda para descrever o conteúdo do ciclo, para paralelizar os ciclos do cálculo das intersecções e das distâncias foi utilizado um padrão de alto nível, `parallelFor`. A partir do código OpenMP é bastante fácil criar uma versão com o FastFlow, sendo apenas indicar as funções lambda que serão o corpo no ciclo.

Cálculo das intersecções

A função lambda `calc_inter` é muito semelhante àquela usada na versão OpenMP que calcula as intersecções (que também é semelhante à versão sequencial).

```
1  //calcular interseccoes
2  init_inter();
3  ff::ParallelFor pf(nworkers, false);
4  pf.parallel_for(0, NUM_JOBS, calc_inter);
5  //obter o total de interseccoes
6  for(int i = 0; i < NUM_JOBS;++i){
7     inter_counter+= counter_jobs[i];
8  }
```

Listagem 5.12: Código que paraleliza o cálculo das intersecções.

Cálculo das distâncias

Para este caso, a função lambda `calc_distances` também é muito semelhante àquela apresentada na versão OpenMP que calcula as distâncias (que também é muito semelhante à versão sequencial).

```
1  //calcular distancias
2  init_distances();
3  ff::ParallelFor pf2(nworkers, false);
4  pf2.parallel_for(0L, MAX_X*MAX_Y, calc_distances);
```

Listagem 5.13: Código que paraleliza o cálculo das distâncias.

5.5. *FastFlow – GPU*

Tal como no programa FastFlow (CPU), esta versão GPU é bastante semelhante à versão CUDA. A diferença entre elas reside na sintaxe usada, visto que na versão FastFlow são usados Algorithmic skeletons que basta instanciar com os dados para obter os resultados da versão CUDA.

Uma diferença é que o FastFlow usa macros para definir o corpo dos kernels, em que este acaba por ser muito semelhante à versão CUDA. Segue-se apenas um exemplo ilustrativo para entender como é que isto é feito (macro FFMAPFUNC gera o corpo semelhante à função __global__ do CUDA). O número a seguir ao nome da macro depende do número de variáveis que são transferidas entre o host e o device.

```
1  //KERNEL TO DELETE 0's of intersections; 1 thread = 1 pos
2  FFMAPFUNC4(deleteZerosInter, unsigned int, unsigned int, tid, intersection,
3  vector_inter, intersection, new_vector, unsigned int, counters, unsigned int,
4  compass,
5          int bin_pos = tid / compass[0]; // posicao no vector de bins
6          int counter_pos = tid % compass[0]; //offset interseccao do bin
7          unsigned int my_pos_counter = counters[bin_pos];
```

Listagem 5.14:Código que exemplifica o corpo da função kernel CUDA.

As Tasks, referidas no capítulo do desenho da solução (Capítulo 4), servem para definir os apontadores das variáveis do host e do device. Tal como no CUDA, as variáveis que têm que ser copiadas entre o host e o device têm que ser identificadas para o padrão tratar das transferências. Nunca esquecer que é necessário referir o tamanho das variáveis que vão ser copiadas para o GPU.

5.6. *Alterações Framework*

A forma como a ferramenta FastFlow está implementada não dá muita liberdade ao programador de escolher as variáveis que são transferidas entre o host e o device. A metodologia usada para implementar os programas que usam CUDA, tornou a implementação que usa a ferramenta bastante lenta. Para entender melhor a raiz do problema, foi estudada a classe "stencilReduceCUDA" da ferramenta FastFlow que é responsável por executar o kernel CUDA e controlar a memória do GPU. Concluído esse estudo, foi possível enumerar as etapas do código a executar em GPU:

1) É definido o tamanho do vector de entrada. No melhor caso, existem threads suficientes para cada entrada do vector.
2) É definido o tamanho da grid e dos blocos do kernel que vai ser executado.
3) É executada a função `startMR()` que pode ser alterada pelo utilizador (função executada antes do MapReduce).
4) Para cada uma das variáveis da seguinte lista:
    a. apontador `in`;
    b. apontador `out`;
    c. apontador `env1`;
    d. apontador `env2`;
    e. apontador `env3`;
    f. apontador `env4`;
    g. apontador `env5`;
    h. apontador `env6`.

    São executados dois métodos `cudaFree()` e `cudaMalloc()`, após verificações de certas variáveis.
5) Caso seja necessário, é feita uma cópia do Host para o Device através da função: `cudaMemcpyAsync().`
6) Se o vector de entrada e o vector de saída forem diferentes, é executado um kernel que inicializa os valores do vector de saída.
7) Nesta fase, o algoritmo segue uma das três opções:
    a. Pure Map;
    b. Pure Reduce;
    c. Map Reduce.
8) Em todos eles é executada a função `beforeMR()`, seguido do kernel, seguido da função `afterMR()`. As duas funções referidas podem ser modificadas ao gosto do programador.
9) Depois de efectuado o kernel, os dados de output são copiados do Device para o Host.

Como o objectivo é minimizar as transferências entre o Host e o Device, dado que as cópias entre a memória do dispositivo e do host têm um custo elevado, temos

que nos focar nas etapas 5 e 9 da lista apresentada anteriormente. É nestas etapas que são efectuadas transferências de memória. Existem situações em que os dados devem permanecer no device, uma vez que vão ser necessários em cálculos consecutivos, em vez de terem de estar a ser sempre copiados entre o host e o device. Desta forma foram adicionados vários métodos que permitem ao programador controlar melhor as transferências efectuadas. Também foram adicionados métodos que permitem alterar o estado de uma variável, se vai ser copiada ou se não vai ser copiada. Como a ferramenta apenas permite copiar do Device para o Host o vector de saída, foram acrescentados métodos para alterar o seu estado (isto é, se é copiado ou não). Para o caso de copiar do Host para o Device, a ferramenta já permite que todas as variáveis sejam copiadas, daí a necessidade de criar um método para cada uma delas (variáveis apresentadas na etapa 4)) (Listagem 5.15).

```
1   //Copiar DEVICE->HOST?
2   bool getCopyHostOut() {return copyHostOut;}
3   //set copiar DEVICE->HOST ON/OFF
4   void setCopyHostOut(boolean state) {copyHostOut = state;}
5
6   //Copiar HOST->DEVICE?
7   bool getCopyDeviceIn() {return copyDeviceIn;}
8   bool getCopyDeviceOut() {return copyDeviceOut;}
9   bool getCopyDevice1() {return copyDevice1;}
10  bool getCopyDevice2() {return copyDevice2;}
11  //… até device 6
12
13  //G COPY HOST->DEVICE
16  void setCopyDeviceIn(boolean state) {copyDeviceIn = state;}
17  void setCopyDeviceOut(boolean state) {copyDeviceOut = state;}
18  void setCopyDevice1(boolean state) {copyDevice1 = state;}
19  void setCopyDevice2(boolean state) {copyDevice2 = state;}
23  //… até device 6
```

Listagem 5.15: Métodos adicionados à classe "stencilReduceCUDA" da ferramenta FastFlow.

Esta alteração é bastante fácil de ser utilizada, tanto na classe da ferramenta como pelo utilizador da mesma. Seguem-se dois exemplos de como isto pode ser feito, em que num primeiro exemplo podemos ver como é que estes métodos são usados na ferramenta (Listagem 5.16), e no segundo exemplo podemos ver como é que os métodos são usados pelo utilizador para controlar as variáveis que são transferidas (Listagem 5.17).

```
1  if(isPureMap()) {
2     Task.swap();                     // because of the next swap op
3     do {
4        Task.swap();
5        Task.beforeMR();
6        //CUDA Map
7        mapCUDAKernel<TkernelMap, Tin, Tout, Tenv1, Tenv2, Tenv3, Tenv4, Tenv5,
8        Tenv6><<<blockcnt,thxblock,0,stream>>>(*kernelMap,Task.getInDevicePtr(),
9        Task.getOutDevicePtr(),Task.getEnv1DevicePtr(), Task.getEnv2DevicePtr(),
10       Task.getEnv3DevicePtr(),Task.getEnv4DevicePtr(),Task.getEnv5DevicePtr(),
11       Task.getEnv6DevicePtr(), size);
12       Task.afterMR(task?task:(void*)oneShot);
13    } while (Task.iterCondition(reduceVar, ++iter) && iter < maxIter);
14    if(Task.getCopyHostOut())
15       cudaMemcpyAsync(Task.getOutPtr(), Task.getOutDevicePtr(),
                         Task.getBytesizeOut(), cudaMemcpyDeviceToHost, stream);
16    cudaStreamSynchronize(stream);
17 }
```

(a)

```
1  if(env1Ptr) {
2     if (oldSize_env1 < Task.getBytesizeEnv1()) {
3        cudaFree(env1_buffer);
4        if (cudaMalloc(&env1_buffer, Task.getBytesizeEnv1()) != cudaSuccess)
5        error("mapCUDA error while allocating memory on device (env1
   buffer)\n");
6        oldSize_env1 = Task.getBytesizeEnv1();
7     }
8     Task.setEnv1DevicePtr(env1_buffer);
9     if(Task.getCopyDevice1())
10       cudaMemcpyAsync(env1_buffer, env1Ptr, Task.getBytesizeEnv1(),
11                       cudaMemcpyHostToDevice, stream);
12 }
```

(b)

Listagem 5.16: (a) verificação da cópia do device para host; (b) verificação da cópia do host para device.

Para o utilizador poder usar estas alterações, só necessita de indicar na Task (dentro do método `setTask()` onde são definidas as variáveis do kernel) quais as variáveis que não vão ser transferidas (Listagem 5.17).

```
1   void setTask(void* t) {
2      if (t) {
3         cudaTaskInt *t_ = (cudaTaskInt *) t;
4         setInPtr(t_->in);
5         setOutPtr(t_->out);
6         setEnv1Ptr(t_->vector);
7         setEnv2Ptr(t_->vector);
8         setEnv3Ptr(t_->params_var);
9         setSizeIn(t_->size_in);
10        setSizeOut(t_->size_in);
11        setSizeEnv1(t_->size_vector);
12        setSizeEnv2(t_->size_vector);
13        setSizeEnv3(1);
14        if(FASTER){
15           setCopyDevice1(false);  //variavel env1 não é copiada HOST->DEVICE
16           setCopyDevice2(false);  //variavel env2 não é copiada HOST->DEVICE
17        }
18     }
19  }
```

Listagem 5.17: Exemplo de uma task em que as variáveis Env1 e Env2 não são copiadas do host para o device.

6

Discussão dos Resultados

Neste capítulo são apresentados os resultados da execução das diferentes versões as quais incluem a execução OpenMP em CPU, a execução CUDA em GPU, e usando a ferramenta FastFlow com códigos para CPU e GPU. Na secção 6.1 é feita uma descrição do hardware e do software utilizado. Durante o desenvolvimento do trabalho, foi encontrada uma forma diferente, daquela usada pelo Pedro Ferreira, para ordenação das intersecções, que designámos de nova ordenação. Desta forma também é realizada uma avaliação desta nova ordenação cujos resultados são apresentados na secção 6.2. Foram realizadas alterações à ferramenta FastFlow, sendo estes resultados avaliados na secção 6.3. A secção 6.4 avalia dois grupos de resultados para a geração da matriz. No primeiro grupo, podemos consultar os resultados usando o algoritmo de ordenação disponibilizado pela biblioteca. No segundo grupo, podemos consultar os resultados usando o novo algoritmo de ordenação que foi proposto. De seguida é realizada uma comparação entre os dois grupos. Dentro de cada grupo, existem programas desenvolvidos com e sem o FastFlow (Capitulo 4).

6.1. *Hardware e Software*

O hardware usado neste trabalho está detalhado na Tabela 6.1 e Tabela 6.2. A máquina usada tem GPUs com suporte a CUDA. Para uma informação mais detalhada das especificações dos dispositivos, consultar [48][49].

Foi usada a versão 7.5 CUDA Toolkit e o compilador **nvcc** (NVIDIA's CUDA Compiler) para GPUs NVIDIA. Para os programas que usam a linguagem C++ foi usado o compilador **gcc** (GNU Compiler Collection). Foi usada a versão v.2.0.8 da ferramenta FastFlow (recorrer a [50] para mais informações sobre a ferramenta).

Tabela 6.1: Características do sistema usado neste trabalho.

Características	Linux tomo-3
Sistema Operativo	Ubuntu 14.04.3 LTS
CPU	Intel® Xeon® CPU E5506 @ 2.13GHz
Memória	12GB
GPU	NVIDIA® Quadro® FX 3800 NVIDIA® GeForce GTX 680

Tabela 6.2: Características do GPU usado neste trabalho.

Características	NVIDIA® GeForce GTX 680
CUDA Cores	1536
Memory Size Total	2048MB
Memory Bandwidth (GB/sec)	192.2
Compute capability (version)	3.0
Microarchitecture	Kepler

6.2. *Nova ordenação*

Para testar a implementação da nova ordenação para as intersecções, descrita no capítulo 4 (Secção 4.2), foram realizados vários testes, desde a sua aplicação num vector simples até à sua utilização na construção da matriz sistema. A ideia era provar o conceito inicialmente proposto, em que é possível ordenar as intersecções mais rapidamente que o algoritmo genérico utilizado, por exemplo, pela biblioteca do C++ [28]. Partindo do exemplo usado anteriormente no capítulo 5 (Figura 5.2), foram realizados testes em vectores com dimensões bastante superiores, provando-se que, independentemente do tamanho do vector, a nova ordenação consegue sempre ordenar os resultados mais rapidamente.

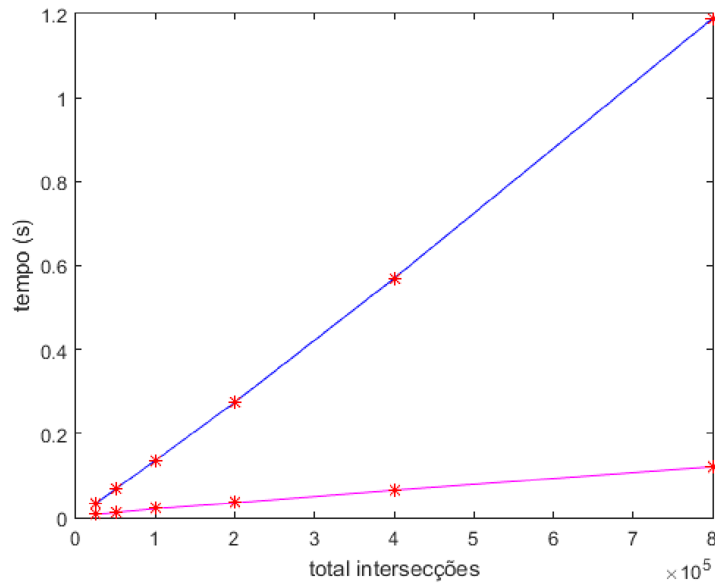


Figura 6.1: Tempo médio da ordenação usando a biblioteca do C++ (linha azul); tempo médio da ordenação usando o novo método (linha magenta).

O tempo da ordenação para os dois casos, uso da biblioteca C++ e nova ordenação, foi registado para diferentes números de intersecções (25.000, 50.000, 100.000,…), tal como mostra a Figura 6.1. Para ambas, o tempo aumenta linearmente com o aumento do número de intersecções. No entanto, a nova ordenação consegue obter tempos muito mais reduzidos comparativamente com a implementação da biblioteca. Na Figura 6.4 da secção 6.4 são apresentados os resultados do tempo da construção da matriz usando os dois tipos de ordenação. Pode-se observar que a vantagem do novo método de ordenação é mais evidente à medida que aumenta o número de intersecções. Assim sendo, a vantagem do novo método de ordenação é mais clara quando o factor de agrupamento passa a ser 1.

6.3. *Resultados Alterações Framework*

Nos testes realizados, foi possível confirmar a redução do tempo da construção da matriz. Basicamente o cálculo da matriz usando GPUs tem 4 etapas, que se traduzem a 4 mapas (na versão do FastFlow):

(a) Mapa do cálculo das intersecções;

(b) Mapa que elimina os zeros das intersecções;

(c) Mapa que calcula as distâncias;

(d) Mapa que elemina os zeros das distâncias;

Cada um destes 4 mapas vai executar um kernel no GPU. O teste em questão calculava as 25 matrizes sistema. Relembrar que para o caso do GPU era necessário dividir o detetor em várias partes. Para este teste foi usado um factor de escala igual a 16. Antes de ser aplicada a alteração na ferramenta, o tempo médio de cada mapa foi registado, obtendo os seguintes resultados:

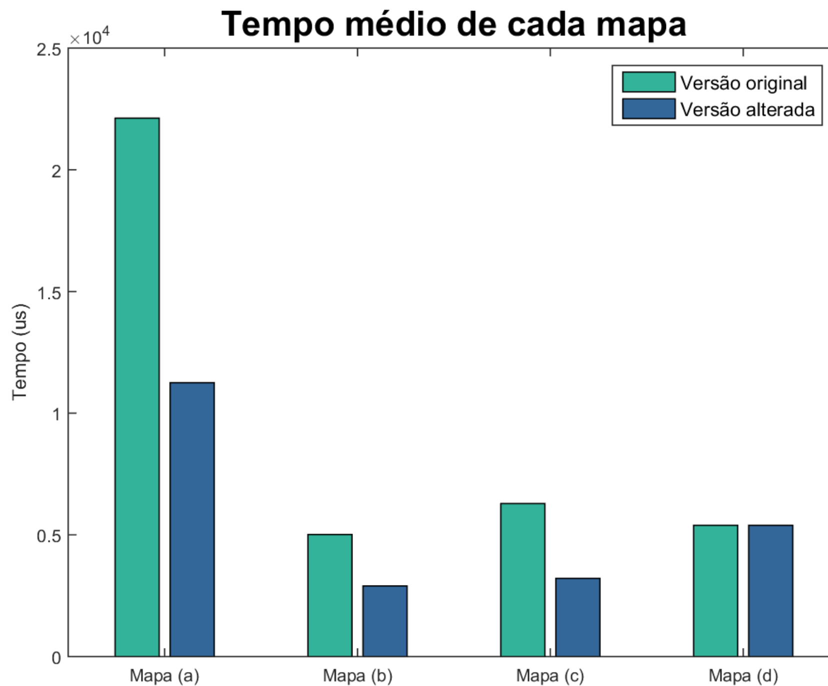


Figura 6.2: Tempo médio de cada mapa no cálculo da matriz.

Como podemos ver pela análise do gráfico, com a utilização das alterações efectuadas na ferramenta, foi possível diminuir o tempo de execução dos mapas por metade. O mapa (d) não mostra qualquer alteração porque não beneficiava das alterações que foram realizadas, mantendo o tempo de execução. O seguinte gráfico mostra o impacto desta alteração no tempo total dos 4 mapas (Figura 6.2).

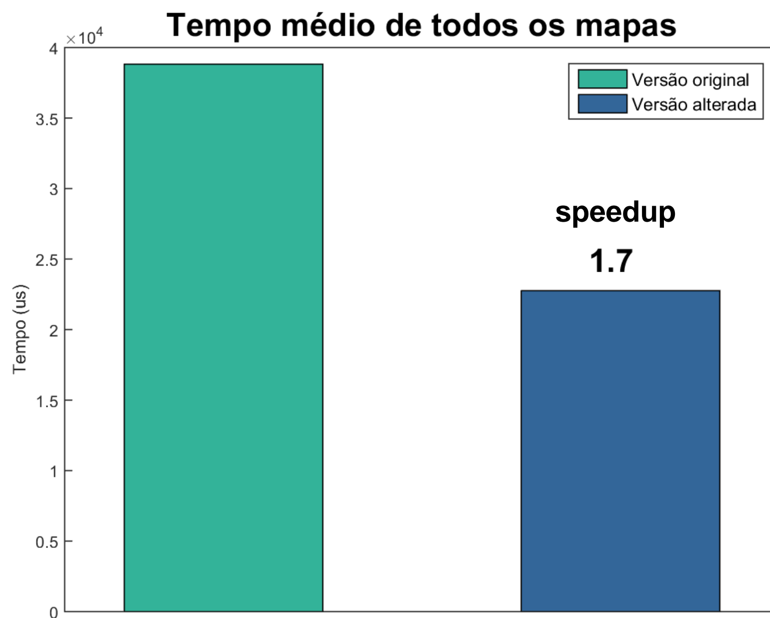


Figura 6.3: Tempo médio para gerar uma parte da matriz sistema.

Podemos verificar que esta alteração pode melhorar bastante o desempenho de certos programas, que fazem transferências desnecessárias entre o Host e o Device.

6.4. *Resultados Construção da Matriz*

Para efectuar tal avaliação, foi necessário criar programas que usam tecnologias semelhantes que possam ser usadas para comparar com os programas FastFlow. Este capítulo compara as diferentes soluções que foram descritas nos capítulos anteriores. Para isso, todas as versões foram testadas usando diferentes escalas (1, 2, 4, 8 e 16). Como para todos os algoritmos de reconstrução necessitamos da matriz sistema para diferentes ângulos, os tempos registados englobam a geração de 25 matrizes como foi descrito na secção 3.2. Como o tempo de execução aumenta à medida que diminuímos a escala, as escalas 1 e 2 são testadas apenas para a geração de 1 matriz (escolhida aleatoriamente uma vez, e usada para todos os testes). Para obter os tempos de execução, todas as versões foram testadas 5 vezes, retirando o melhor e o pior tempo. De seguida é calculado o valor médio entre os 3 tempos restantes e esse é o valor utilizado para a comparação dos resultados.

Atendendo ao que foi referido no início do capítulo, esta secção está dividida em três grupos. O primeiro grupo utiliza os métodos disponibilizados pelas bibliotecas para ordenar os resultados, enquanto o segundo grupo utiliza o novo método de ordenação. Em ambos estes grupos são apresentadas as versões descritas no capítulo 4. No último grupo é feita uma comparação entre estes dois grupos e também é feita uma análise das alterações feitas à ferramenta FastFlow para programas que usam GPUs.

6.4.1. Resultados com Ordenação Biblioteca

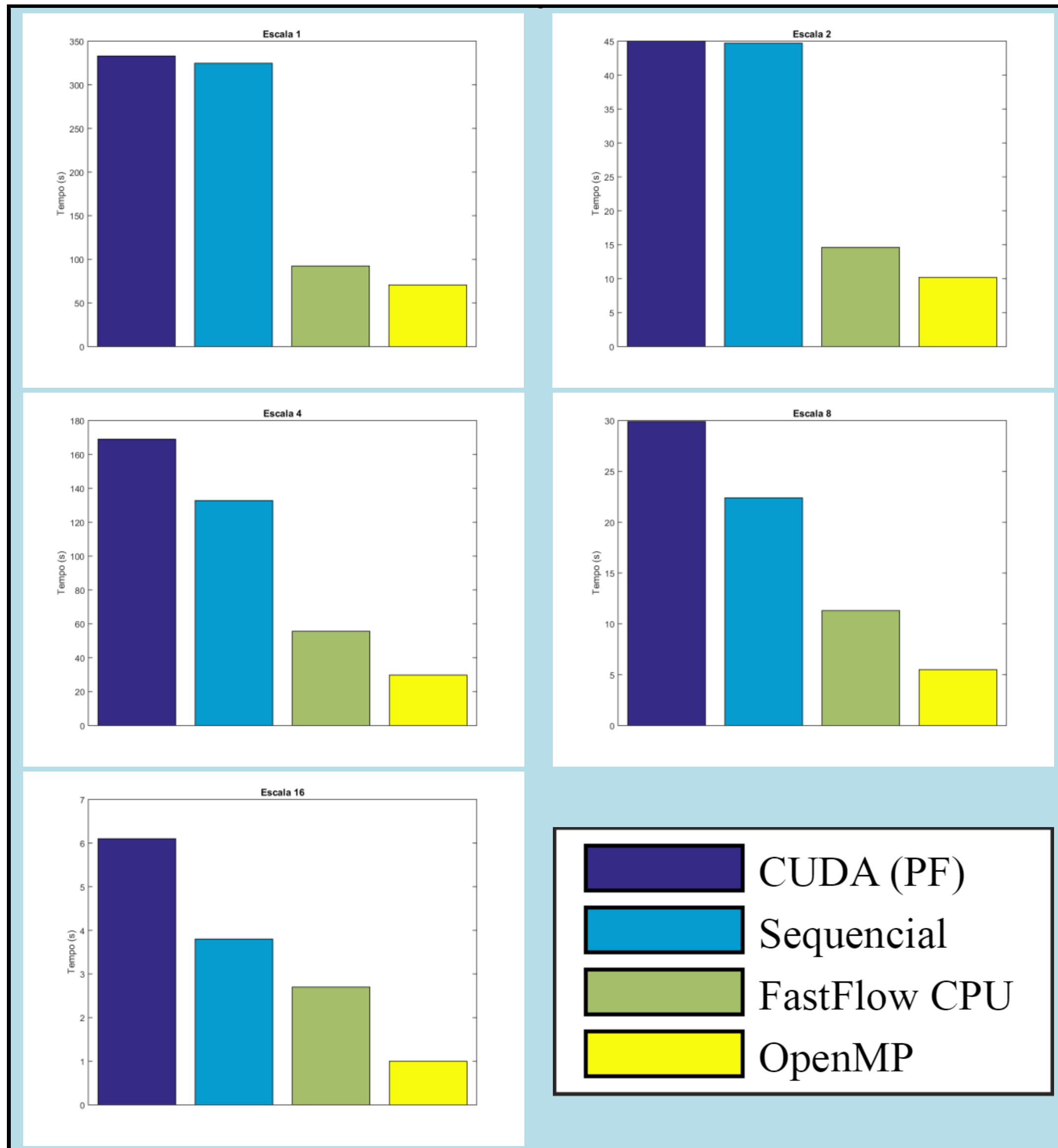


Figura 6.4: Resultado das ordenações para todas as escalas usando o algoritmo de ordenação disponibilizado pela biblioteca.

Para esta secção foram desenvolvidos 4 programas distintos, todos eles utilizam um algoritmo de ordenação disponível por uma biblioteca. Os programas CPU usam a biblioteca do C++ e o programa CUDA (versão de Pedro Ferreira (PF)) usa a biblioteca Thurst. O programa CUDA utiliza uma metodologia um pouco diferente para a ordenação dos dados. Em vez de ordenar as intersecções de uma célula do detetor, a ordenação é feita tendo em consideração todas as intersecções, o que permite que o programa sequencial seja mais rápido que o programa CUDA. A razão pela qual este programa se encontra nestes gráficos é para mostrar a redução do tempo de geração da

matriz alterando apenas a metodologia de ordenação. Desta forma, ainda não é apresentada nenhuma outra versão CUDA, visto que as versões usando a nova ordenação são apresentadas na secção seguinte.

A relação de ordem entre tempos de execução mantém-se para as várias versões independentemente da escala usada, sendo a versão OpenMP sempre a mais rápida. À medida que a escala aumenta, o volume de dados a ser processado diminui. A versão FastFlow consegue obter resultados próximos da versão OpenMP para as escalas menores. Esta diminuição torna o programa FastFlow mais lento. Uma justificação para este acontecimento pode ser o peso da ferramenta. Visto que quanto mais nos aproximamos de escalas maiores, menor será o número total de intersecções, consequentemente o tempo gasto no cálculo da matriz também será menor. Para estas escalas muito grandes, o tempo gasto pela ferramenta, que não seja no cálculo da matriz, vai ter um impacto maior no tempo final da geração da matriz.

Com a análise destes resultados podemos já verificar que o FastFlow consegue obter resultados próximos do OpenMP quando se trata de grandes volumes de dados.

6.4.2. Resultados com Nova Ordenação

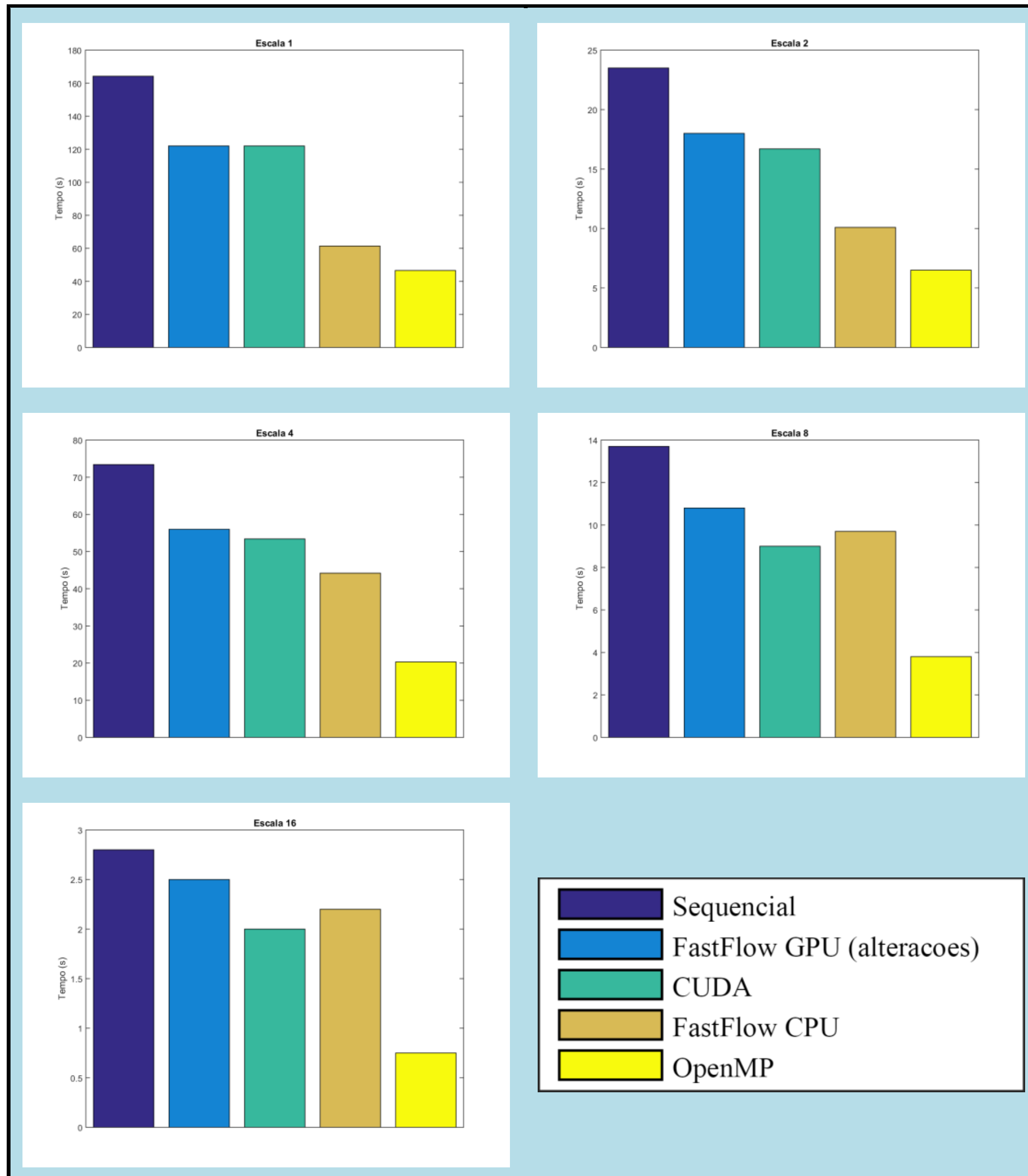


Figura 6.5: Resultado das ordenações para todas as escalas usando o novo algoritmo de ordenação proposto.

Para esta secção foram desenvolvidos 5 programas distintos, todos eles utilizam a nova metodologia de ordenação. A versão CUDA difere da versão de Pedro Ferreira na metodologia de ordenação e nas bibliotecas utilizadas. A versão FastFlow que usa GPUs é a que utiliza as alterações que foram efectuadas à ferramenta, visto que esta versão é a mais rápida. A comparação entre a versão que utiliza as alterações e a versão que não utiliza as alterações é feita na próxima secção.

Os testes efectuados neste grupo foram os mesmos do grupo anterior (Secção 6.4.1). Tal como foi comprovado anteriormente, o tempo da nova ordenação é inferior à ordenação da biblioteca (secção 6.2). Quando se aplica o conceito da nova ordenação para o cálculo da matriz, os resultados são bastante diferentes.

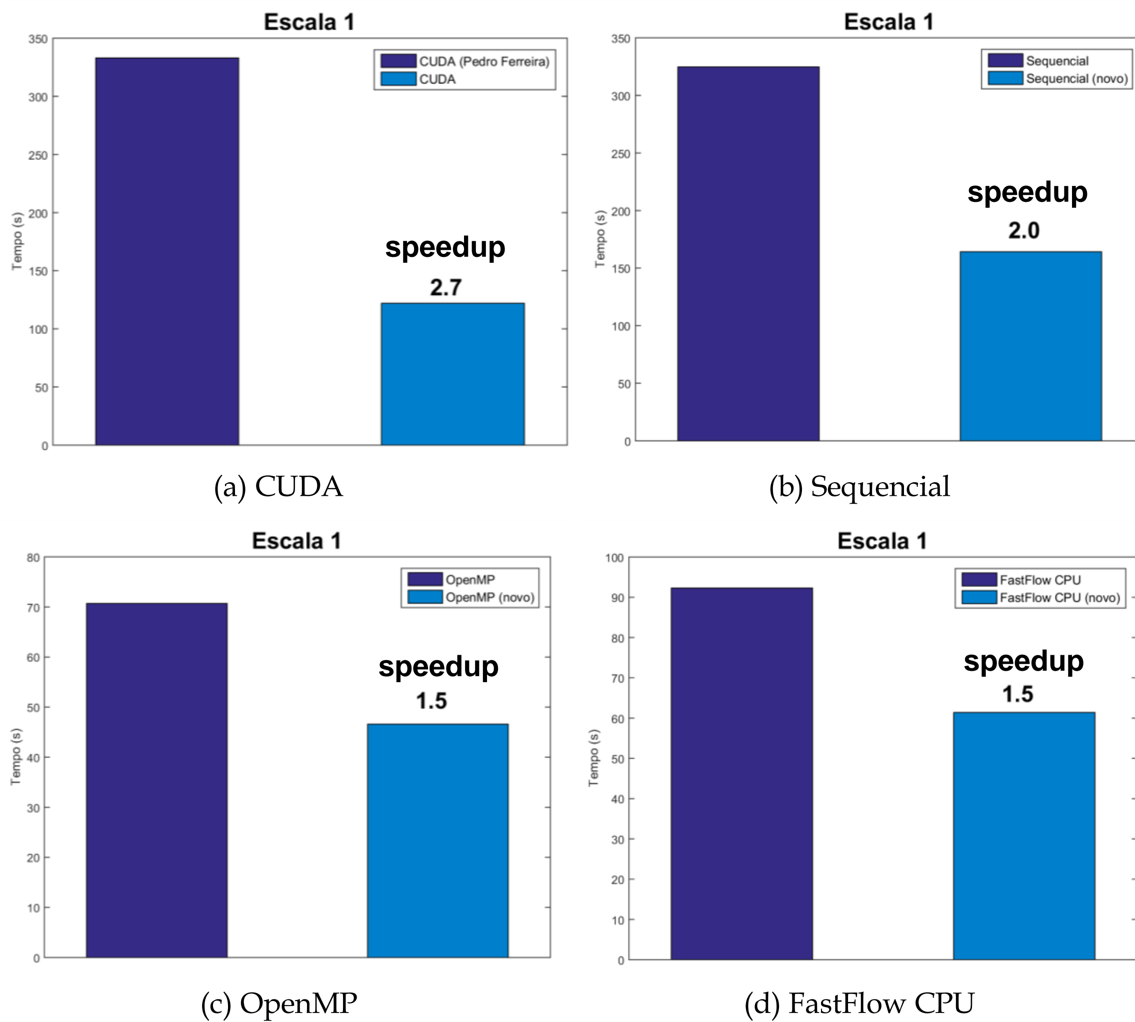
Tal como vimos na secção anterior, para escalas pequenas, o programa em FastFlow que usa apenas o CPU tem tempos de execução próximos da versão OpenMP. À medida que se aumenta a escala, o programa FastFlow diminui bastante o seu desempenho quando comparamos com o programa OpenMP. Da mesma forma, o programa FastFlow que usa GPUs também obteve resultados muito próximos para escalas mais pequenas. Também neste caso, à medida que aumentamos a escala, o desempenho deste programa diminui quando comparamos com a versão CUDA.

As versões do GPU, com e sem FastFlow, não conseguem obter tempos próximos das versões que usam CPU. Uma das justificações para esta causa pode ser que o tempo de processamento do GPU mais o tempo de transferências faz com que não seja possível reproduzir resultados capazes de competir com a implementação CPU.

6.4.3. Comparação dos Resultados das Diferentes Ordenações

Esta secção serve para corroborar o que foi referido anteriormente, que o tempo do cálculo da matriz consegue ser reduzido com o novo método de ordenação. Também aqui é feita a análise das alterações efectuadas à ferramenta FastFlow e o impacto no problema da geração da matriz.

Na Figura 6.6 podemos ver que o speedup obtido usando o novo método de ordenação é visível tanto para as implementações que usam GPU como para as implementações que apenas usam CPU. Para a versão CUDA, o speedup é superior porque a metodologia usada na versão mais lenta é diferente da metodologia usada pelos outros programas, tornando o programa mais lento. Ainda assim, nos programas paralelos que usam apenas CPU foi possível obter um speedup médio de 1.5. Estas comparações comprovam o que tínhamos previsto na secção 6.2, é possível reduzir o tempo da ordenação das intersecções usando a nova metodologia proposta.



(a) CUDA

(b) Sequencial

(c) OpenMP

(d) FastFlow CPU

Figura 6.6: Figura com diferentes versões da geração da matriz para uma escala igual a 1, indicando o speedup obtido usando o novo método de ordenação.

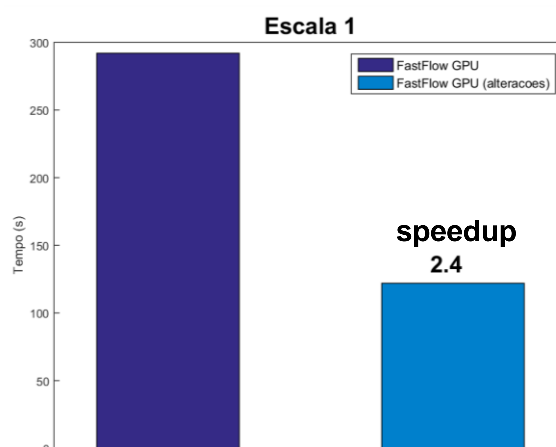
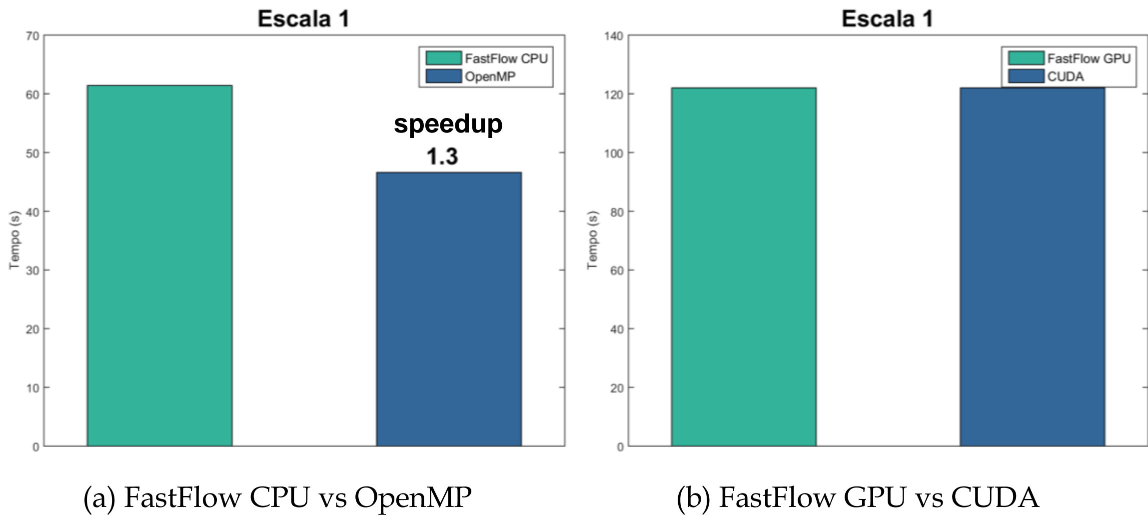


Figura 6.7: Comparação dos tempos de execução da versão FastFlow GPU com e sem as alterações à ferramenta.

A Figura 6.7 mostra o speedup obtido usando as alterações feitas à ferramenta. Nem todos os problemas podem beneficiar destas alterações. Para este problema em concreto, geração da matriz, as alterações permitem um speedup de 2.4 em comparação com a implementação que não usa as alterações.



(a) FastFlow CPU vs OpenMP (b) FastFlow GPU vs CUDA

Figura 6.8: Comparação da ferramenta FastFlow com as tecnologias mais usadas: OpenMP para CPU; CUDA para GPUs.

A análise destes dois gráficos (Figura 6.8) permite concluir que a ferramenta FastFlow é capaz de produzir resultados muito próximos das tecnologias mais utilizadas. Para o caso da implementação apenas em CPU, a versão OpenMP tem um speedup de 1.3. Este valor é aumentado à medida que se aumenta a escala. Para a implementação que usa GPUs, podemos dizer que tiveram os tempos muito semelhantes, com diferenças mínimas inferiores a 1%. Da mesma forma, à medida que se aumenta a escala, a versão CUDA obtém melhores resultados que a versão FastFlow.

Um dos objectivos desta tese era comprovar que o FastFlow pode ser uma boa alternativa às tecnologias mais utilizadas, quando se trata de desenhar programas paralelos para CPU e GPU. Ao mesmo tempo, um outro objectivo foi encontrar possíveis alterações que possam ser feitas ao algoritmo para melhorar a construção da matriz. A análise dos resultados permite concluir que o FastFlow pode ser uma boa alternativa para o desenho de programas paralelos quando se trata de escalas mais baixas. Como o suporte para GPU é ainda muito simples, foi necessário efectuar algumas alterações à ferramenta para adaptar ao tipo de problema que foi resolvido (construção da matriz sistema), de maneira a que esta implementação pudesse competir com as outras. Também foi utilizado um novo método de ordenação dos dados que permitiu uma redução no tempo total do cálculo da matriz.

7

Conclusão e Trabalho Futuro

Este capítulo começa por comparar os resultados conseguidos com os objectivos apresentados no capítulo 1. Seguidamente referem-se desenvolvimentos futuros que se poderão basear neste trabalho.

7.1.Comparação dos resultados obtidos com os objectivos iniciais

Como não tinha nenhuma experiência com algoritmos de reconstrução iterativos, foi necessário estudá-los exaustivamente para desenvolver um algoritmo na melhor condição possível. Foi necessário compreender como é que a matriz sistema é usada em cada um dos algoritmos estudados (SART, ML-EM e OS-EM) para saber o seu impacto na geração da imagem final. Pedro Ferreira fez esta avaliação e confirmou que a geração da matriz sistema tem um grande impacto na construção da imagem final. Como ainda não tinham existido muitas tentativas de paralelizar este código usando CUDA, Pedro Ferreira fez essa proposta, conseguindo incorporar o cálculo da matriz com o resto dos algoritmos. Os resultados obtidos foram bastante promissores, confirmando que o uso de GPUs pode reduzir o tempo dos algoritmos.

Programar para CUDA pode ser bastante complicado. Grande parte do trabalho realizado no decorrer desta tese focou-se na aprendizagem de um novo paradigma de programação, a programação paralela, que é menos óbvia do que a programação sequencial, daí ter demorado mais tempo no desenvolvimento das várias versões paralelas. A maior desvantagem em desenvolver estes programas é que este processo pode tornar-se longo, isto está relacionado com o tempo que leva até produzir código correcto e nas alterações que são feitas ao mesmo para melhorar o desempenho. Os debuggers existentes não são muito *user-friendly* dificultando ainda mais o processo de desenvolvimento.

Neste contexto, surgiu a possibilidade de desenvolver o cálculo da matriz usando a ferramenta do FastFlow, que permite desenvolver programas que tiram partido dos padrões recorrentes de programação paralela. Isto é feito usando Algorithmic Skeletons, como foi explicado ao longo da tese. O objectivo principal deste tipo de ferramentas é possibilitar ao programador uma forma de desenvolver programas paralelos sem necessitar de compreender as primitivas de baixo nível. Sabemos que o uso destas ferramentas pode significar uma perda de desempenho do programa final. Nesta tese foi possível constatar que o desempenho dos programas gerados, tanto para CPU como para GPU conseguem obter tempos próximos daqueles que usam as tecnologias mais comuns o que as torna numa opção interessante a considerar. A ferramenta FastFlow tira partido das funções lambda do C++ e são usadas tanto para os programas para CPU como para GPU. A familiarização deste tipo de funções pode ajudar o programador no desenvolvimento dos programas, tendo apenas que compreender como funciona a ferramenta.

Uma desvantagem desta ferramenta no desenvolvimento de programas que usam GPU é que a documentação é um bocado fraca e os padrões recorrentes disponíveis não são muitos. Não se pode dizer o mesmo do suporte para desenvolvimento de programas paralelos que usam CPU, que está muito bem documentada e possuí muitos padrões recorrentes para serem utilizados.

Adicionalmente, a alteração do código da própria ferramenta, embora pontual, permitiu diminuir o tempo de execução dos programas que usam GPU. É essencial reduzir o número de transferências de memória entre o *host* e o *device* e para o tipo de problema que estávamos a tratar, a ferramenta não dava muita liberdade ao programador para controlar quando e quais as variáveis a transferir. Esta pequena alteração permite ao utilizador, de uma forma muito transparente, controlar melhor estas transferências, melhorando o tempo de execução dos programas.

O uso deste tipo de ferramentas ainda não é muito comum na comunidade de programação. O seu uso pode facilitar bastante o desenvolvimento de programas paralelos que não necessitam de implementações paralelas bastante optimizadas. A facilidade de desenvolvimento de programas com a ferramenta faz com que o uso desta ferramenta possa vir a ser cada vez mais comum. Por outro lado, quando se necessita de programas optimizados a sua escolha pode não ser a melhor, sendo então essencial perceber o factor entre a optimização do programa e o tempo para o seu desenvolvimento.

Por outro lado, foi possível melhorar o tempo para calcular a matriz sistema usando diferentes abordagens na ordenação dos dados. Esta diferente abordagem

permitiu reduzir bastante o tempo da geração da matriz, consequentemente o tempo dos algoritmos. A redução do tempo fez com que a versão CUDA não fosse tão rápida ao ponto de compensar o uso de GPUs. Os resultados são bastantes promissores, tanto para as versões que usam a ferramenta FastFlow como para as que não usam, havendo uma grande potencialidade na sua aplicação real.

O desenvolvimento desta tese permitiu perceber que o uso de Algorithmic Skeletons pode facilitar o desenvolvimento de programas paralelos. Ao mesmo tempo, percebemos que o uso de CUDA pode melhorar bastante o tempo de um algoritmo. Da mesma forma, conseguimos concluir que nem todos os programas beneficiam da paralelização em CUDA, o que se pôde confirmar após às alterações feitas no algoritmo do cálculo da matriz. O uso de Algorithmic Skeletons é bastante promissor para vários tipos de programas paralelos, existindo uma grande variedade de opções por onde escolher.

7.2. Os desenvolvimentos futuros que se poderão basear neste trabalho são:

- Como o resto do algoritmo de reconstrução de imagem, do qual este trabalho partiu, está implementado de forma sequencial, usando IDL, surge uma grande oportunidade de paralelizar outras fases do algoritmo usando FastFlow. Existem várias fases com potencial oportunidade de paralelismo para trablho futuro, tais como a reprojecção e a retroprojecção. A portabilidade do código possibilitado por esta ferramenta é algo que não foi explorado nesta tese e que pode ser feito num trabalho futuro.
- Estava prevista uma avaliação da adaptação da solução proposta no contexto da área das Ciências dos Materiais, o que não foi possível fazer por restrições de tempo.
- Integrar a nova metodologia de ordenação nas intersecções no cálculo da matriz com as implementações dos algoritmos existentes.

Referências

[1] Ferreira, P. R. T. (2014). Optimization of breast tomosynthesis image reconstruction using parallel computing. Tese (Mestre em Engenharia Biomédica) – Faculdade de Ciências e Tecnologia Universidade Nova de Lisboa, Caparica.

[2] Cole, M. (2004). Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel computing, 30(3), 389-406.

[3] González-Vélez, H., & Leyton, M. (2010). A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience, 40(12), 1135-1160.

[4] Image processing Wikipedia book. Disponível: http://www.disi.unige.it/person/RovettaS/rad/image-processing-wikipedia-book.pdf (visitado em Fevereiro 2015).

[5] Zitova, B., & Flusser, J. (2003). Image registration methods: a survey. *Image and vision computing*, *21*(11), 977-1000.

[6] Medical Imaging. Disponível: http://en.wikipedia.org/wiki/Medical_imaging (visitado em Fevereiro 2015).

[7] Sechopoulos, I. (2013). A review of breast tomosynthesis. Part II. Image reconstruction, processing and analysis, and advanced applications. *Medical physics*, *40*(1), 014302.

[8] Mueller, K.(2011). Iterative Reconstructions Methods. Materiais do curso CSE 377/591 – Introduction to Medical Image, disponível em http://www3.cs.stonybrook.edu/~mueller/teaching/cse377/iterativeRecon_new.pdf

[9] Iterative Reconstruction. Disponível: http://en.wikipedia.org/wiki/Iterative_reconstruction (visitado em Fevereiro 2015).

[10] Uecker, M., Zhang, S., Voit, D., Karaus, A., Merboldt, K. D., & Frahm, J. (2010). Real-time MRI at a resolution of 20 ms. *NMR in Biomedicine*, *23*(8), 986-994.

[11] Hutton, Brian F. An Introduction to Iterative Reconstruction. Alasbimn Journal 5(18): October 2002. Article N° AJ18-6.

[12] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, *30*(3), 202-210.

[13] Amdahl, G. M. (2013). Computer Architecture and Amdahl's Law. *Computer*,*46*(12), 38-46.

[14] Saxena, S., Sharma, N., & Sharma, S. (2013). Image Processing Tasks using Parallel Computing in Multi core Architecture and its Applications in Medical Imaging. *International Journal of Advanced Research in Computer and Communication Engineering*, *2*(4).

[15] Squyres, J. M., Lumsdaine, A., McCandless, B. C., & Stevenson, R. L. (1996). Parallel and Distributed Algorithms for High Speed Image Processing. *Dept. of Computer Science and Engineering, University of Notre Dame, Computer Science and Engineering technical report TR*, *12*, 1996.

[16] Kim, C. G., Kim, J. G., & Lee, D. H. (2014). Optimizing image processing on multi-core CPUs with Intel parallel programming technologies. *Multimedia Tools and Applications*, *68*(2), 237-251.

[17] Ciechanowicz, P., Kegel, P., Schellmann, M., Gorlatch, S., & Kuchen, H. (2010). Parallelizing the LM OSEM Image Reconstruction on Multi-Core Clusters. *Parallel Computing: From Multicores and GPU's to Petascale*, *19*, 169.

[18] H. Fatemi, H. Corporaal, T. Basten, P. Jonker, R. Kleihorst, Implementing face recognition using a parallel image processing environment based on algorithmic skeletons, in: J.J. van Wijk, J.W.J. Heijnsdijk, K.G. Langedoen, R. Veltkamp (Eds.), 10th Annual Conference of the Advanced School for Computing and Imaging, Proceedings, Port Zlande, the Netherlands, 2004, pp. 351–357

[19] Olmedo, E., De La Calleja, J., Benitez, A., Medina, M. A., & por Computadora, L. D. P. (2012). Point to point processing of digital images using parallel computing. *International Journal of Computer Science Issues*, *9*(33), 3.

[20] Hammond, K., Aldinucci, M., Brown, C., Cesarini, F., Danelutto, M., González-Vélez, H., ... & Shainer, G. (2013, January). The Paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In Formal Methods for Components and Objects (pp. 218-236). Springer Berlin Heidelberg.

[21] Cole, M. (2004). Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel computing, 30(3), 389-406.

[22] Campa, S. et. al. (2011). Paraphrase Project: Initial generic patterns report D2.1.

[23] González-Vélez, H., & Leyton, M. (2010). A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience, 40(12), 1135-1160.

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

[25] Enmyren, J., & Kessler, C. W. (2010, September). SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications* (pp. 5-14). ACM.

[26] Wilkinson, B., & Allen, M.. (2006). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2/E*. Pearson Education India. (pp. 370-401 capítulo 12).

[27] Slides "Aplicação do algoritmo Maximum Likelihood Expectation Maximization (ML-EM) à imagiologia da mama" de Professor Nuno Matela IBEB.

[28] Sort C++. [Online]. Disponivel: http://www.cplusplus.com/reference/algorithm/sort/ (visitado a 15 Setembro 2015).

[29] OpenMP Application Program Interface. Disponível: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf (visitado a 16 de Setembro de 2015)

[30] C++, a brief description. Disponível: http://www.cplusplus.com/info/description/ (visitado a 16 de Setembro de 2015)

[31] CUDA C/C++ Basics. [Online]. Disponível: http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf (visitado a 16 Setembro 2015).

[32] FastFlow architecture. Disponível: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:architecture (visitado a 16 Setembro 2015).

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

[34] Siddon, Robert L. "Fast calculation of the exact radiological path for a three-dimensional CT array." *Medical physics* 12.2 (1985): 252-255.

[35] Wu, Xiaolin. "An efficient antialiasing technique." *ACM SIGGRAPH Computer Graphics*. Vol. 25. No. 4. ACM, 1991.

[36] Schaa, D., Brown, B., Jang, B., Mistry, P., Dominguez, R., Kaeli, D., Moore, R., Kopans, D. B. 2011. GPU Acceleration of Iterative Digital Breast Tomosynthesis. GPU Computing Gems, Morgan Kaufmann, Boston, 647-657.

[37] Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction*. Vol. 2. Oxford University Press, 1977.

[38] Mattson, Timothy G., Beverly A. Sanders, and Berna L. Massingill. "A pattern language for parallel programming." *Addision-Wesley, | ISBN-10* 321228111 (2004).

[39] Keutzer, Kurt, and Tim Mattson. "A design pattern language for engineering (parallel) software." *Intel Technology Journal* 13.4 (2010).

[40] Pacheco, Peter. *An introduction to parallel programming*. Elsevier, 2011.

[41] Barlas, Gerassimos. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014.

[42] Vlissides, John, et al. "Design patterns: Elements of reusable object-oriented software." *Reading: Addison-Wesley* 49.120 (1995): 11.

[43] Cole, Murray I. *Algorithmic skeletons: A structured approach to the management of parallel computation*. Diss. University of Edinburgh, 1988.

[44] Baptista, Mariana, et al. "Image quality and dose assessment in digital breast tomosynthesis: A Monte Carlo study." *Radiation Physics and Chemistry* 104 (2014): 158-162.

[45] Matela, Nuno Miguel de Pinto Lobo. "2d iterative image reconstruction for a dual planar detector for positron emission mammography." (2008).

[46] Di Maria, S., Baptista, M., Felix, M., Oliveira, N., Matela, N., Janeiro, L., ... & Silva, A. (2014). Optimal photon energy comparison between digital breast tomosynthesis and mammography: A case study. *Physica Medica*, *30*(4), 482-488.

[47] Bowman, Kenneth P. *An Introduction to Programming with IDL: Interactive data language*. Academic Press, 2006.

[48] Especificações NVIDIA GeForce GTX 680. Disponíivel:
http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications (visitado a 19 Janeiro 2016).

[49] Cuda capability e microarchitecture. Disponível:
https://en.wikipedia.org/wiki/CUDA. (visitado a 18 Janeiro 2016).

[50] Informações ferramenta FastFlow. Disponível:
http://calvados.di.unipi.itt/dokuwiki/doku.php/ffnamespace:about. (visitado a 18 Janeiro 2016).

Simplificação do cálculo das intersecções

Neste anexo é apresentada uma simplificação do cálculo das intersecções para os diferentes eixos (x, y e z). Os resultados desta simplificação são comparados com os resultados obtidos no capítulo 6, Discussão dos Resultados. O anexo está dividido em duas partes. Na primeira parte são apresentados todos os cálculos para simplificar o cálculo das intersecções. Na segunda parte, é feita a comparação com os resultados obtidos anteriormente.

Esta simplificação foi sugerida pelo Professor Carlos Viegas Damásio depois da apresentação desta tese.

1. Demonstração da simplificação

Os ciclos responsáveis pelo cálculo das intersecções têm elementos constantes que estão a ser recalculados a cada iteração. Resumidamente, para cada um dos eixos, à medida que vamos alterando o valor da coordenada (associada ao eixo) temos que calcular o alfa e as restantes coordenadas. A equação XX ilustra a situação para o eixo X, em que variamos a coordenada X, calculando o valor de alfa e das coordenadas Y e Z. O mesmo raciocínio é utilizado para os restantes eixos.

$$\alpha = \frac{X(\alpha) - A_x}{B_x - A_x}$$

$$Y(\alpha) = A_y + \alpha(B_y - A_y)$$

$$Z(\alpha) = A_z + \alpha(B_z - A_z)$$

Equação A.1: Equações paramétricas do raio (eixo X).

1.1.Eixo X

Para o eixo X, o ciclo que se pretende simplificar é o seguinte:

```
1   for(x_integer = bin_x+1; x_integer <= detector_X/2; x_integer++){
2      float alpha;
3      intersection inter_x;
4      //Resolver a equação 2
5      inter_x.x = x_integer * x_bin_size; //coordenada X
6      alpha = (inter_x.x - (bin_x + 0.5) * x_bin_size) / slopevector_x;
7      //sabendo alpha, descobrir coordenada Y e Z
8      inter_x.y = (bin_y+0.5) * y_bin_size + alpha * slopevector_y; //coordenada Y
9      inter_x.z=distance_from_breast_support_table_to_detector+alpha*
    slopevector_z;//coordenada Z
11     inter_x.bin = bin_y + bin_x * detector_Y; //bin da intersecção
12     ...
```

Listagem A.1: Parte do código para o cálculo das intersecções no eixo X.

Começamos por simplificar todos os termos que são constantes para o ciclo:

$$a_bin_x = \frac{(bin_x + 0{,}5) * x_bin_size}{slopevector_x}$$

$$a_bin_y = (bin_y\ + 0{,}5) * y_bin_size$$

$$bin = bin_x + bin_y + detector_X$$

Obtendo o seguinte código:

```
1   float a_bin_x = ((bin_x + 0.5) * x_bin_size)/slopevector_x;
2   float a_bin_y = (bin_y + 0.5) * y_bin_size;
3   float bin = bin_x + bin_y * detector_X;
4
5   for(x_integer = bin_x+1; x_integer <= detector_X/2; x_integer++){
6      float alpha;
7      intersection inter_x;
8      //Resolver a equação 2
9      inter_x.x = x_integer * x_bin_size; //coordenada X
10     alpha = inter_x.x / slopevector_x - a_bin_x;
11     //sabendo alpha, descobrir coordenada Y e Z
12     inter_x.y = a_bin_y + alpha * slopevector_y; //coordenada Y
13     inter_x.z=distance_from_breast_support_table_to_detector+alpha*
    slopevector_z;//coordenada Z
14     inter_x.bin = bin; //bin da intersecção
       ...
```

Listagem A.2: 1ª simplificação do código para o cálculo das intersecções no eixo X.

Substituindo agora o valor de alfa nas expressões de ***inter_x.y*** e ***inter_x.z*** tem-se:

$$inter_x.y = a_bin_y + \left(\frac{inter_x.x}{slopevector_x} - a_bin_x\right) * slopevector_y$$

$$= a_bin_y - a_bin_x\ * slopevector_y + inter_x.x * \frac{slopevector_y}{slopevector_x}$$

$$inter_x.z = distance + \left(\frac{inter_x.x}{slopevector_x} - a_bin_x\right) * slopevector_z$$

$$= distance - a_bin_x \; * slopevector_z + inter_x.x * \frac{slopevector_z}{slopevector_x}$$

onde distance é ***distance_from_breast_support_table_to_detector*** e o valor ***inter_x.x*** é o único que varia.

Separando as partes constantes para as expressões ***inter_x.y*** e ***inter_x.z*** ficamos com:

$$start_y = (a_bin_y \; - a_bin_x * slopevector_y)$$

$$start_z = (distance - a_bin_x * slopevector_z)$$

$$mult_y = \frac{slopevector_y}{slopevector_x}$$

$$mult_z = \frac{slopevector_z}{slopevector_x}$$

Obtendo o seguinte código:

```
1  float a_bin_x = ((bin_x + 0.5) * x_bin_size)/slopevector_x;
2  float a_bin_y = (bin_y + 0.5) * y_bin_size;
3  float bin = bin_x + bin_y * detector_X;
4  float start_y = (a_bin_y - a_bin_x * slopevector_y);
5  float start_z = (distance_from_breast_support_table_to_detector - a_bin_x *
   slopevector_z);
6  float mult_y = slopevector_y / slopevector_x;
7  float mult_z = slopevector_z / slopevector_x;
8
9  for(x_integer = bin_x+1; x_integer <= detector_X/2; x_integer++){
10    intersection inter_x;
11    inter_x.x = x_integer * x_bin_size; //coordenada X
12    inter_x.y = start_y + (inter_x.x) * mult_y; //coordenada Y
13    inter_x.z = start_z + (inter_x.x) * mult_z;//coordenada Z
14    inter_x.bin = bin; //bin da intersecção
      ...
```

Listagem A.3: 2ª simplificação do código para o cálculo das intersecções no eixo X.

Ainda é possível melhorar o ciclo interno evitando multiplicações no cálculo das coordenadas y e z. Sabemos que o valor da próxima coordenada depende apenas do valor de ***inter_x.x*** e sabemos que a cada iteração este valor aumenta por ***x_bin_size***. Sabendo isto, podemos utilizar variáveis auxiliares para manter o estado da coordenada e ir actualizando a cada iteração, obtendo o seguinte código:

```
1  float a_bin_x = ((bin_x + 0.5) * x_bin_size)/slopevector_x;
2  float a_bin_y = (bin_y + 0.5) * y_bin_size;
3  float bin = bin_x + bin_y * detector_X;
4  float start_y = (a_bin_y - a_bin_x * slopevector_y);
5  float start_z = (distance_from_breast_support_table_to_detector - a_bin_x *
   slopevector_z);
6  float mult_y = slopevector_y / slopevector_x;
7  float mult_z = slopevector_z / slopevector_x;
8
9  float bm_y = x_bin_size * mult_y;
10 float bm_z = x_bin_size * mult_z;
11 float curr_x = bin_x * x_bin_size;
12 float curr_y = start_y + curr_x * mult_y;
13 float curr_z = start_z + curr_x * mult_z;
14
15 for(x_integer = bin_x+1; x_integer <= detector_X/2; x_integer++){
16    intersection inter_x;
17    curr_x += x_bin_size;
18    curr_y += bm_y;
19    curr_z += bm_z;
20    inter_x.x = curr_x; //coordenada X
21    inter_x.y = curr_y; //coordenada Y
22    inter_x.z = curr_z;//coordenada Z
23    inter_x.bin = bin; //bin da intersecção
      ...
```

Listagem A.4: 3ª simplificação do código para o cálculo das intersecções no eixo X.

1.2. Eixo Y

O raciocínio aplicado para o eixo X é bastante semelhante para os restantes eixos. No caso do eixo Y, as equações paramétricas são as seguintes:

$$\alpha = \frac{Y(\alpha) - A_y}{B_y - A_y}$$

$$X(\alpha) = A_x + \alpha(B_x - A_x)$$

$$Z(\alpha) = A_z + \alpha(B_z - A_z)$$

Equação A.2: Equações paramétricas do raio (eixo Y).

Este eixo tem uma particularidade, o valor da coordenada Y tanto pode ser crescente, como decrescente, dependendo da posição inicial da célula em comparação com a posição do foco (a explicação pode ser encontrada no capítulo XX). Como tal, apenas é apresentada a simplificação para um dos casos, nomeadamente a situação em que o valor da coordenada é crescente. Como este exemplo é bastante semelhante ao eixo X, são apresentadas todas as simplificações matemáticas, seguidas do código final, evitando assim uma descrição detalhada da evolução do código.

```
1   for(y_integer = bin_y+1; y_integer <= detector_Y; y_integer++){
3      float alpha;
4      intersection inter_y;
5      //Resolver a equação 2
6      inter_y.y = y_int * y_bin_size; //coordenada Y
7      alpha = (inter_y.y - (bin_y + 0.5) * y_bin_size) / slopevector_y;
8      //sabendo alpha, descobrir coordenada X e Z
9      inter_y.x = (bin_x + 0.5) * x_bin_size + alpha * slopevector_x; //coordenada X
10     inter_y.z= distance_from_breat_support_table_to_detector+
    alpha*slopevector_z;//coordenada Z
11     inter_y.bin = bin_y + bin_x * detector_Y; //bin da intersecção
12     ...
```

Listagem A.5: Parte do código para o cálculo das intersecções no eixo Y (crescente).

Começamos por simplificar todos os termos que são constantes para o ciclo:

$$a_bin_x = (bin_x\ + 0{,}5) * x_bin_size$$

$$a_bin_y = \frac{(bin_y + 0{,}5) * y_bin_size}{slopevector_y}$$

Substituindo agora o valor de alfa nas expressões de ***inter_y.x*** e ***inter_y.z*** tem-se:

$$inter_y.x = a_bin_x + \left(\frac{inter_y.y}{slopevector_y} - a_bin_y\right) * slopevector_x$$

$$= a_bin_x - a_bin_y\ * slopevector_x + inter_y.y * \frac{slopevector_x}{slopevector_y}$$

$$inter_y.z = distance + \left(\frac{inter_y.y}{slopevector_y} - a_bin_y\right) * slopevector_z$$

$$= distance - a_bin_y\ * slopevector_z + inter_y.y * \frac{slopevector_z}{slopevector_y}$$

Separando as partes constantes para as expressões ***inter_y.x*** e ***inter_y.z*** ficamos com:

$$start_x = (a_bin_x\ - a_bin_y * slopevector_x)$$

$$start_z = (distance - a_bin_y * slopevector_z)$$

$$mult_x = \frac{slopevector_x}{slopevector_y}$$

$$mult_z = \frac{slopevector_z}{slopevector_y}$$

Sabemos que o valor da próxima coordenada depende apenas do valor de ***inter_y.y*** e sabemos que a cada iteração este valor aumenta por ***y_bin_size***. Sabendo isto, podemos utilizar variáveis auxiliares para manter o estado da coordenada e ir actualizando a cada iteração, obtendo o seguinte código:

```
1  float a_bin_x = (bin_x + 0.5) * x_bin_size;
2  float a_bin_y = ((bin_y + 0.5) * y_bin_size)/slopevector_y;
3  float bin = bin_x + bin_y * detector_X;
4  float start_x = (a_bin_x - a_bin_y * slopevector_x);
5  float start_z = (distance_from_breast_support_table_to_detector - a_bin_y *
   slopevector_z);
6  float mult_x = slopevector_x / slopevector_y;
7  float mult_z = slopevector_z / slopevector_y;
8
9  float bm_x = y_bin_size * mult_x;
10 float bm_z = y_bin_size * mult_z;
11 float curr_y = bin_y * y_bin_size;
12 float curr_x = start_x + curr_y * mult_x;
13 float curr_z = start_z + curr_y * mult_z;
14
15 for(y_integer = bin_y+1; y_integer <= detector_Y; y_integer++){
16    intersection inter_y;
17    curr_y += y_bin_size;
18    curr_x += bm_x;
19    curr_z += bm_z;
20    inter_y.x = curr_x; //coordenada X
21    inter_y.y = curr_y; //coordenada Y
22    inter_y.z = curr_z;//coordenada Z
23    inter_y.bin = bin; //bin da intersecção
      ...
```

Listagem A.6: Simplificação do código para o cálculo das intersecções no eixo Y (crescente).

1.3. Eixo Z

Tal como foi referido anteriormente, o raciocínio utilizado para os outros eixos é bastante semelhante para o eixo Z. No caso do eixo Z, as equações paramétricas são as seguintes:

$$\alpha = \frac{Z(\alpha) - A_z}{B_z - A_z}$$

$$X(\alpha) = A_x + \alpha(B_x - A_x)$$

$$Y(\alpha) = A_y + \alpha(B_y - A_y)$$

Equação A.3: Equações paramétricas do raio (eixo Y).

Como este exemplo é bastante semelhante ao eixo X, são apresentadas todas as simplificações matemáticas, seguidas do código final, evitando assim uma descrição detalhada da evolução do código.

```
1   for(z_integer = 0; z_integer <= N_slices-1; z_integer++){
2      float alpha;
3      intersection inter_z;
4      //Resolver a equação 2
5      inter_z.z = z_integer; //coordenada Z
6      alpha =(inter_z.z- distance_from_breat_support_table_to_detector) /
   slopevector_z;
7      //sabendo alpha, descobrir coordenada X e Y
8      inter_z.x = (bin_x+0.5) * x_bin_size + alpha * slopevector_x; //coordenada X
9      inter_z.y = (bin_y+0.5) * y_bin_size + alpha * slopevector_y; //cordenada Y
10     inter_z.bin = bin_y + bin_x * detector_Y; //bin da intersecção
11     ...
```

Listagem A.7: Parte do código para o cálculo das intersecções no eixo Z.

Começamos por simplificar todos os termos que são constantes para o ciclo:

$$a_bin_x = (bin_x\ + 0{,}5) * x_bin_size$$

$$a_bin_y = (bin_y\ + 0{,}5) * y_bin_size$$

Substituindo agora o valor de alfa nas expressões de ***inter_z.x*** e ***inter_z.y*** tem-se:

$$inter_z.x = a_bin_x + \left(\frac{inter_z.z - distance}{slopevector_z}\right) * slopevector_x$$

$$= a_bin_x - distance\ * \frac{slopevector_x}{slopevector_z} + inter_z.z * \frac{slopevector_x}{slopevector_z}$$

$$inter_z.y = a_bin_y + \left(\frac{inter_z.z - distance}{slopevector_z}\right) * slopevector_y$$

$$= a_bin_y - distance * \frac{slopevector_y}{slopevector_z} + inter_z.z * \frac{slopevector_y}{slopevector_z}$$

Separando as partes constantes para as expressões ***inter_z.x*** e ***inter_z.y*** ficamos com:

$$mult_x = \frac{slopevector_x}{slopevector_z}$$

$$mult_y = \frac{slopevector_y}{slopevector_z}$$

$$start_x = (a_bin_x\ - distance * mult_x)$$

$$start_y = (a_bin_y - distance * mult_y)$$

Sabemos que o valor da próxima coordenada depende apenas do valor de ***inter_z.z*** e sabemos que a cada iteração este valor aumenta por ***1***. Sabendo isto, podemos utilizar variáveis auxiliares para manter o estado da coordenada e ir actualizando a cada iteração, obtendo o seguinte código:

```
1  float a_bin_x = (bin_x + 0.5) * x_bin_size;
2  float a_bin_y = (bin_y + 0.5) * y_bin_size;
3  float bin = bin_x + bin_y * detector_X;
4  float mult_x = slopevector_x / slopevector_z;
5  float mult_y = slopevector_y / slopevector_z;
6  float start_x = (a_bin_x - distance_from_breast_support_table_to_detector *
   mult_x);
7  float start_y = (a_bin_y - distance_from_breast_support_table_to_detector *
   mult_y);
8
9  // float bm_x = 1 * mult_x; usamos mult_x como incremento
10 // float bm_y = 1 * mult_y; usamos mult_y como incremento
11 float curr_z = 0;
12 float curr_x = start_x + curr_z * mult_x;
13 float curr_y = start_y + curr_z * mult_y;
14
15 for(z_integer = 0; z_integer <= N_slices-1; z_integer++){
16    intersection inter_z;
17    inter_y.x = curr_x; //coordenada X
18    inter_y.y = curr_y; //coordenada Y
19    inter_y.z = curr_z;//coordenada Z
20    inter_y.bin = bin; //bin da intersecção
21    curr_z += 1;
22    curr_x += mult_x;
23    curr_y += mult_y;
      ...
```

Listagem A.8: Simplificação do código para o cálculo das intersecções no eixo Z.

2. *Comparação dos Resultados*

O código simplificado foi testado usando os mesmos critérios apresentados no capítulo 6 e os seus tempos de execução foram comparados com os tempos apresentados no mesmo capítulo. Foram executadas 3 versões: sequencial, OpenMP e CUDA. Os resultados são os seguintes:

Tabela A.1: Comparação das versões sequenciais.

Escala	*Sequencial (s)*	*Sequencial (simplificado) (s)*	*Melhoramento (%)*
16	2,8	2,75	1,82%
8	13,7	13,1	4,58%
4	73,4	68,3	7,47%
2	23,5	20,4	15,20%
1	164,2	140,3	17,03%

Tabela A.2: Comparação das versões OpenMP.

Escala	*OpenMP (s)*	*OpenMP (simplificado) (s)*	*Melhoramento (%)*
16	0,75	0,75	0,00%
8	3,8	3,7	2,70%
4	20,3	19	6,84%
2	6,5	5,9	10,17%
1	46,6	38,4	21,35%

Tabela A.3: Comparação das versões CUDA.

Escala	*CUDA (s)*	*CUDA (simplificado) (s)*	*Melhoramento (%)*
16	1,9	1,9	0,00%
8	9	9	0,00%
4	53,4	53,3	0,19%
2	16,7	16,7	0,00%
1	122	122	0,00%

Houve um aumento do desempenho para as versões sequenciais e OpenMP. Em ambas as situações, quanto menor for a escala, maior é o número de células e consequentemente maior é o número de intersecções para cada célula. Nestas situações (escalas menores), o desempenho obtido pela simplificação dos ciclos é maior quando comparando com as escalas menores.

Podemos concluir que esta simplificação permite reduzir o tempo do cálculo da matriz, obtendo os melhores resultados para as escalas menores.