Bruno Alexandre Meirinhos Preto

Licenciado em Engenharia Informática

Utilização de GPGPUs para a Identificação de Objectos em Imagens Tomográficas

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: Prof. Doutor Pedro Abílio Duarte de Medeiros

Júri:

Presidente:	Prof. Doutor Nuno Manuel Robalo Correia
Arguente:	Prof. Doutor Salvador Pinto Abreu
Vogal:	Prof. Doutor Pedro Abílio Duarte de Medeiros

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2011

Utilização de GPGPUs para a Identificação de Objectos em Imagens Tomográficas

Agradecimentos

À Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, mais especificamente a todos os professores que me forneceram, no decorrer do curso, as ferramentas necessárias para colocar em prática este Projecto;

Ao Professor Doutor Pedro Medeiros, por toda a sua ajuda e inteira disponibilidade para me orientar ao longo deste caminho, motivando-me sempre a melhorar;

Ao Professor Doutor Adriano Lopes, por toda a ajuda durante o período de estudo de estruturas de dados geométricas, que ocorreu em simultâneo com a realização da dissertação;

Ao Professor Doutor Fernando Birra, por todo o interesse e dedicação durante a fase de concepção do algoritmo;

A toda a equipa do Projecto "Ambiente de Resolução de Problemas para Caracterização Estrutural de Materiais por Tomografia" do Departamento de Engenharia de Materiais, mais especificamente ao Nuno Oliveira e ao Paulo Quaresma, pela sua disponibilidade e auxílio ao longo da elaboração da dissertação;

Aos meus pais e em especial à minha namorada, por toda a ajuda, compreensão e apoio em todo o meu percurso académico, bem como, durante esta recta final do mesmo;

O meu sincero Obrigado!

Resumo

Os materiais compósitos são utilizados em inúmeras áreas tais como a indústria automóvel, aeronáutica e espacial. Nestes materiais existe um elemento base (por ex. alumínio) ao qual são adicionados reforços (por ex. partículas de silício) que dão origem a um composto com propriedades físicas muito diferentes do material base.

Os investigadores na área da Ciência de Materiais utilizam imagens tomográficas para analisar a estrutura desses materiais compósitos. Esta análise pode exigir um conjunto considerável de recursos computacionais, quer pela quantidade dos dados envolvidos quer pela complexidade do processamento necessário.

Está em curso no DI-FCT-UNL um projecto cujo principal objectivo é suportar a execução das operações de processamento de imagens tomográficas de materiais compósitos em máquinas de secretária contendo uma combinação entre *Central Processing Units* (CPUs) e *General-Purpose Graphics Processing Units* (GPGPUs). Neste contexto, a presente dissertação tem como principal objectivo a concepção e implementação de algoritmos eficientes de análise de imagens tomográficas de materiais compósitos, mais particularmente a detecção de objectos nessas imagens a 3 dimensões. Esta detecção permitirá fazer a caracterização desses objectos segundo critérios definidos por investigadores da área.

Os algoritmos foram desenvolvidos tendo como alvo a plataforma OpenCL, que permite o desenvolvimento de aplicações portáveis em *hardware* heterogéneo incluindo CPUs clássicos e aceleradores dos quais os GPGPUs são a classe mais usada correntemente.

Após a análise de diversas alternativas, desenvolveu-se um algoritmo de identificação de objectos que se encaixa na classe *Connected-Component Labeling*. Esse algoritmo foi adaptado ao processamento de imagens a três dimensões; por outro lado, foi modificado de forma a conseguir extrair o máximo de desempenho do *hardware* GPGPU disponível.

A principal contribuição da tese é assim, a concepção, desenvolvimento e avaliação de um conjunto de algoritmos de identificação de objectos, que conseguem tempos de processamento de imagens de grande dimensão inferiores a 10s. Estes tempos são muito bons quando comparados com programas com a mesma funcionalidade, anteriormente

desenvolvidos, que executavam em multiprocessadores de memória distribuída e partilhada e cujos tempos de execução se mediam em minutos. Esta diminuição dos tempos de execução constitui uma contribuição importante para a construção de ambientes interactivos de análise de imagens tomográficas.

Palavras-chave: Algoritmos de processamento de imagens a três dimensões; Paralelização de aplicações; OpenCL; Tomografia; GPGPU.

Abstract

Composite materials are used in several areas such as the automotive, aeronautics and, space industries. These materials are built using a base element (for example aluminum) reinforced with particles (for example silicon): this combination leads to a composite material that can have physical properties that are very different from those of the base material.

X-Ray micro tomography with synchrotron's radiation is used to obtain tomographic images, allowing a Materials specialist to characterize the reinforcement population of these composite materials. This analysis may require a considerable amount of computing resources due to the size of the data and the complexity of the processing required.

At DI-FCT-UNL a project is running, among at the support of the processing of tomographic images of composite materials in desktop PCs containing multi-core CPUs and GPGPUs. This dissertation has the main objective of designing efficient algorithms for object identification in 3-D images of composite materials suited for execution in GPGPUs. This identification will allow the chacterization of the object population, according to criteria defined by a Materials researcher.

The algorithms were developed targeting the OpenCL platform that supports the development of heterogeneous applications running in classical CPUs and accelerators, in particular the most cost-effective ones, GPGPUs.

After the analysis of several alternatives, an algorithm for object identification of the *Connected-Component Labeling* fanmily was developed. The algorithm was adapted to the processing of 3D images, an modified in order to exploit efficiently the GPGPU hardware.

The main contribution of this thesis is the design, implementation and assessment of a set of object identification algorithms, that in the processing of very large 3D images, achieve execution times below 10 seconds. This is much better than the execution times of similar algorithms executing in distributed and shared memory multiprocessors, that executed in minutes. The reduction of execution time achieved is an important contribution to the building of interactive tomographic image analysis systems.

Keywords: Algorithm; Application parallelization; OpenCL; Tomography; SCIRun; GPGPU.

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Enquadramento

O presente trabalho é realizado no âmbito da Unidade Curricular Dissertação, do 2º ano de Mestrado em Engenharia Informática da Faculdade de Ciências e Tecnologia (FCT), da Universidade Nova de Lisboa (UNL) com a finalidade da obtenção do grau de Mestre em Engenharia Informática.

Esta dissertação surge na sequência das dissertações do P. Quaresma [36], T. Cadavez [7] e P. Paiva [33]. A área de trabalho é a implementação de algoritmos de manipulação de imagens tomográficas tridimensionais, utilizando a paralelização para diminuir os tempos de execução.

A dissertação apresenta como principais objectivos a identificação e análise dos estudos já existentes relativos ao processamento de dados tomográficos através de GPGPUs mais precisamente na identificação de objectos em imagens tridimensionais, obtidas através de micro tomografia computacional.

A dissertação realizada enquadra-se num projecto financiado pela Fundação de Ciências e Tecnologia do Ministério da Ciência, Tecnologia e Ensino Superior (MCTES) já existente nas Unidades de Investigação Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) e Centro de Investigação de Materiais (CENIMAT/FCT/UNL), cujo tema é "Ambiente de Resolução de Problemas para Caracterização Estrutural de Materiais por Tomografia".

De seguida, serão abordados alguns aspectos fundamentais para a definição do objectivo da dissertação e para o planeamento da sua execução.

1.1 Microtomografia

Uma das áreas mais activas em engenharia de materiais é o desenvolvimento de materiais compósitos. Estes materiais são construídos através da junção de um dado material com reforços de outros materiais, combinando assim as suas propriedades. O projecto no qual se insere a dissertação enquadra-se nessa técnica permitindo analisar, através de imagens tomográficas, os materiais compósitos. Para que isso seja possível, é utilizada Microtomografia de Raio X com radiação de sincrotrão (SXMT).

Segundo A. Velhinho em [48], a SXMT constitui um novo desenvolvimento da tomografia computorizada (CT), a qual consiste numa técnica de imagiologia médica, que tem obtido sucesso no estudo de materiais. Esta consiste em avaliar a estrutura de um objecto através de imagens tomográficas.

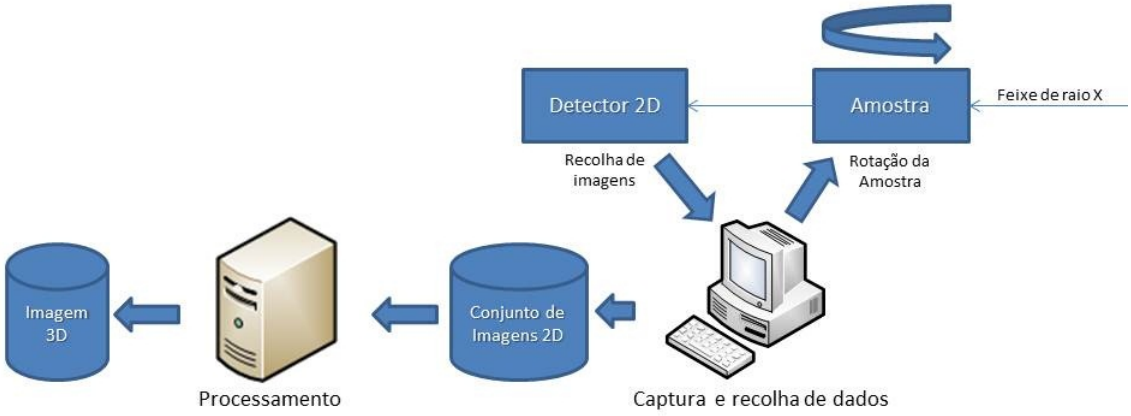


Figura 1.1: Microtomografia de Raio X com radiação de sincrotrão [48].

A técnica mencionada anteriormente, ilustrada na figura 1.1, analisa o material emitindo radiação X que, através da sua propriedade de penetração, é absorvida parcialmente por este, sendo a estrutura do material analisada de acordo com a intensidade da radiação recebida no detector. Como resultado obtém-se um volume de dados que corresponde a imagens bidimensionais que serão processadas, através de algoritmos específicos, originando uma imagem tridimensional que descreve a amostra.

1.2 Processamento dos dados

Para se proceder à análise dos dados tomográficos resultantes de amostras de materiais compósitos, os mesmos são representados através de uma matriz tridimensional, sendo manipulados por algoritmos específicos.

Um dos problemas presentes neste tipo de análise deve-se à elevada dimensão do volume de dados. Assim, caso seja efectuado o processamento sequencial, este torna-se bastante dispendioso devido ao tempo necessário para obter resultados, sendo que em determinados casos este tratamento pode demorar dias.

É importante referir que uma análise pode incluir na aplicação sequências de mais do que um algoritmo, sendo a ordem de aplicação relevante. Desta forma, devido ao tempo despendido em cada algoritmo, o tempo de resposta por sua vez também aumenta. Uma solução possível para melhorar essa eficiência, consiste em efectuar o processamento dos dados em paralelo.

1.3 O projecto

O projecto no qual a dissertação se insere, como já foi referido anteriormente, apresenta como tema "Ambiente de Resolução de Problemas para Caracterização Estrutural de Materiais por Tomografia".

Este projecto apresenta como finalidade o desenvolvimento de um ambiente computacional para resolução de problemas (*Problem Solving Environment* - PSE), que permita aos investigadores da área de Ciências dos Materiais efectuarem a análise de imagens tomográficas sem grandes conhecimentos informáticos, tal como se pode verificar na figura 1.2. O sistema irá ser constituído por um computador pessoal, com *hardware* específico, que poderá ser usado remotamente para desenvolvimento, mas que é usado em exclusivo e intensivamente por especialistas de materiais que pretendem analisar uma imagem.

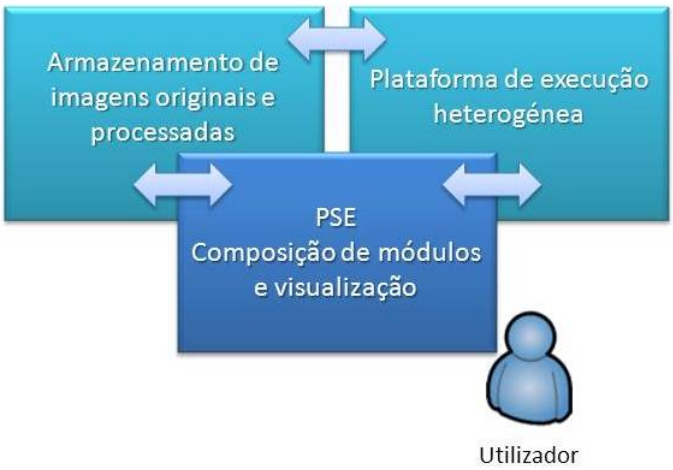


Figura 1.2: Funcionamento do sistema.

Durante o tratamento da informação, o utilizador tem a possibilidade de parametrizar os algoritmos e visualizar os seus resultados, finais e intermédios, em tempo real na máquina cliente, através do sistema SCIRun [35]. O utilizador tem também a possibilidade de combinar algoritmos, bem como definir os seus conjuntos de dados.

O sistema permite ao utilizador um elevado grau de liberdade, sem que este tenha conhecimentos acerca da implementação do sistema, bem como de linguagens de programação. Essa funcionalidade é proporcionada através de blocos, como se pode verificar na figura 1.3. Esses blocos correspondem a algoritmos, que se interligam através dos

canais de entrada e saída. Como é possível observar, o utilizador pode visualizar os resultados graficamente. Um aspecto interessante a salientar corresponde ao facto de cada bloco poder estar associado a uma interface gráfica, que permite configurar o algoritmo em causa.

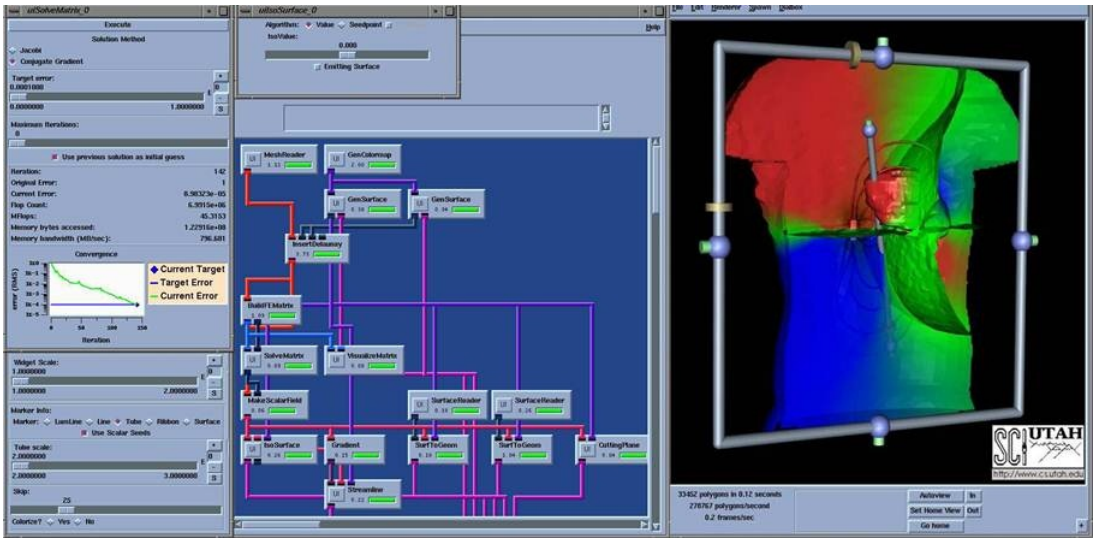


Figura 1.3: Sistema SCIRun [25].

Através do sistema SCIRun, o utilizador tem a possibilidade de desenvolver os seus próprios programas, baseados no repositório de algoritmos disponibilizados. O sistema também permite extensibilidade, de modo a que, caso exista necessidade de desenvolver novos algoritmos para o repositório, o sistema não necessite de ser modificado.

1.4 Estrutura da dissertação

A presente dissertação apresenta como principais objectivos a concepção e o desenvolvimento de algoritmos de identificação de objectos em volumes de dados tridimensionais, obtidos através de micro tomografia computorizada. Estes algoritmos, ao contrário dos implementados anteriormente, são executados através de aceleradores, visto permitirem um maior grau de paralelismo, o que os torna uma mais-valia no aumento da performance dos referidos algoritmos. De forma a implementar essas características, foi utilizada a *framework* OpenCL, que permite desenvolver aplicações para um conjunto diverso de dispositivos, mais especificamente GPGPUs. Para que isso seja possível, foi necessário rescrever os algoritmos de modo a que estes se adaptem a esta nova tecnologia. Tendo em conta o objectivo da dissertação supracitado, a mesma será intitulada "Utilização de GPGPUs para a Identificação de Objectos em Imagens Tomográficas".

A tecnologia Open Computing Language (OpenCL) vem introduzir *overheads* em relação aos algoritmos existentes, devido à necessidade de transferência de dados para o dispositivo. Esse *overhead* é compensado através do poder computacional do mesmo, permitindo executar eficientemente centenas de *threads* em simultâneo, o que, através de

uma boa decomposição do volume de dados, diminui consideravelmente o tempo de execução dos algoritmos. Uma vantagem desta arquitectura nessa tarefa de decomposição é inerente à capacidade de endereçar tridimensionalmente os *threads*.

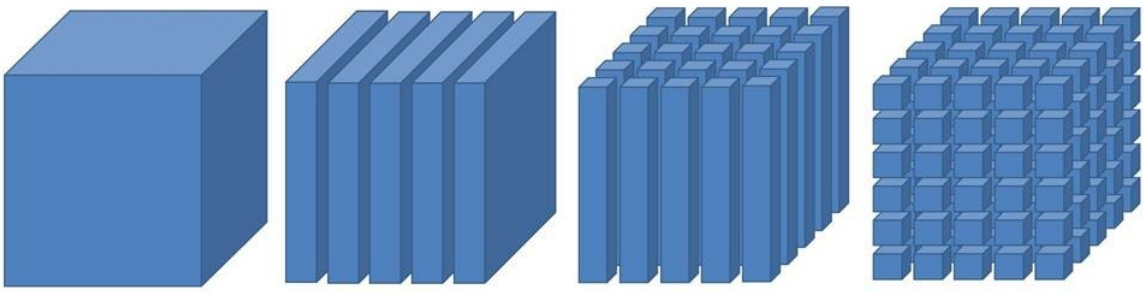


Figura 1.4: Divisão geométrica do volume de dados.

Um dos pontos essenciais da dissertação, como já foi referido anteriormente, consiste na definição de novas estratégias de paralelismo adequadas ao processamento com GPGPUs, distintas das estratégias regulares em memória partilhada ou distribuída, de forma a minimizar o tempo de execução dos algoritmos a um baixo custo. De entre várias estratégias de paralelização, uma das mais importantes é a divisão geométrica.

A divisão geométrica consiste em dividir a informação em blocos, como está ilustrado na figura 1.4, que irão ser processados em paralelo. Esta estratégia constitui uma mais-valia quando o processamento de um subconjunto de dados depende pouco dos subconjuntos vizinhos. Deste modo, todos os *threads* executam segundo um modelo *Single Program, Multiple Data* (SPMD), onde cada *thread* aplica o mesmo conjunto de operações sobre um subconjunto de dados, o que permite diminuir os tempos de execução.

O paralelismo geométrico apresenta algumas limitações, sendo uma delas a dependência entre os dados, causando uma redução significativa no desempenho da aplicação em relação a uma total independência dos dados. Esta situação pode ser controlada através de sincronização. Utilizando esta divisão num dos algoritmos a implementar, que consiste em analisar as partículas presentes numa dada matéria, é necessário sincronizar fluxos de execução, o que irá implicar uma redução do desempenho.

Uma outra limitação diz respeito à divisão equitativa do processamento, que ocorre quando o tempo varia consoante os dados. Esta limitação implica que *threads* diferentes, a processar conjuntos diferentes de dados de igual dimensão, apresentem tempos de execução diferentes. Um exemplo disso ocorre no cálculo do fractal de Monte Carlo, no qual certos pixéis apresentam um tempo de execução superior a outros. Uma forma de contornar esta limitação consiste em utilizar uma fila de trabalho, onde cada fluxo de execução obtém trabalho assincronamente. Desta forma, obtém-se uma distribuição de processamento mais equitativa.

No processamento de imagens tomográficas, a experiência tem mostrado a presença destes dois problemas, o que levou a repensar as estratégias de paralelização, de forma a decompor o problema em diversas tarefas, em que se procura reduzir o número de

pontos de sincronização.

Além da paralelização geométrica, é também utilizada a paralelização funcional, que consiste em dividir as tarefas ao invés dos dados. Este tipo de divisão apresenta uma maior complexidade do que a anterior, no entanto, é a melhor solução quando não existe a independência dos dados atrás descrita. Quando é utilizado o paralelismo geométrico, o *overhead* na sincronização pode reduzir drasticamente a performance do algoritmo, enquanto que através da divisão funcional, cada elemento de processamento executa uma dada tarefa sobre o mesmo conjunto de dados. Logo, o aumento do desempenho é obtido através da diminuição da complexidade das tarefas. O problema desta divisão consiste em efectuar uma distribuição equitativa do processamento entre *threads*.

O algoritmo de identificação de objectos desenvolvido no âmbito desta dissertação utiliza paralelismo funcional, uma vez que são executadas várias fases de processamento sobre o mesmo conjunto de dados. Este utiliza também paralelismo geométrico, não só na subdivisão do conjunto de dados a processar, como internamente em cada uma das fases de processamento.

1.5 Contributo da dissertação

Esta dissertação constitui um contributo significativo, essencialmente no que diz respeito ao aumento da performance dos algoritmos de identificação de objectos em imagens tomográficas, tirando partido da combinação do processamento de CPUs e *General-Purpose Graphics Processing Units* (GPGPUs), uma vez que estes possuem características bastante relevantes para o tipo de operações a realizar.

Devido à portabilidade e heterogeneidade oferecida pela plataforma OpenCL, o desenvolvimento de algoritmos através da mesma vem possibilitar que, futuramente, se utilizem outros dispositivos específicos para aumentar e melhorar o seu desempenho.

Esta dissertação vem também possibilitar a utilização dos algoritmos de um modo mais acessível pelos utilizadores, visto que cada algoritmo irá pertencer a um repositório de algoritmos passíveis de serem utilizados através da plataforma SCIRun.

Os contributos da presente dissertação residem essencialmente na capacidade de identificação de objectos em grandes volumes tridimensionais num reduzido espaço de tempo. Até ao momento, este facto apenas é obtido através de *clusters*, como poderá ser verificado posteriormente numa solução já existente.

Esta dissertação apresenta também novas técnicas de programação através de GPGPUs, que poderão ser posteriormente reutilizadas para rescrever outras soluções.

Nesta dissertação encontra-se uma análise detalhada dos aspectos intrínsecos às soluções, de forma a se obterem as vantagens e limitações da programação com GPGPUs, bem como o estudo de novas estratégias de paralelização que se adequam a este tipo de programação.

A combinação entre CPU e GPGPUs apresenta como vantagem a obtenção de um

elevado nível de paralelismo, a baixo custo, podendo-se assim obter uma melhoria significativa na relação custo / desempenho na execução dos algoritmos. Para que isso fosse possível, foi utilizada a plataforma OpenCL, que permite o desenvolvimento de aplicações portáveis em *hardware* heterogéneo incluindo CPUs clássicos e os chamados aceleradores dos quais os GPGPUs são a classe mais usada correntemente.

Para que os algoritmos possam ser combinados de um modo fácil e intuitivo, foi utilizada a plataforma SCIRun, que permite o desenvolvimento de programas sem que exista a necessidade de uma aquisição prévia de conhecimento acerca de linguagens de programação. Com este sistema, os investigadores além de criarem os seus próprios programas a partir do repositório de algoritmos, podem visualizar resultados finais ou intermédios.

Os algoritmos desenvolvidos constituiram um avanço significativo para a classe de algoritmos *Connected-Component Labeling*, visto processarem volumes de grandes dimensões em tempos de resposta diminuídos, permitindo a sua utilização em ambientes interactivos.

1.6 Organização da dissertação

No que se refere à estrutura da presente dissertação, esta será descrita seguidamente. Inicialmente surge o capítulo referente ao estado da arte no qual se abordam temas considerados imprescindíveis para o enquadramento e compreensão da dissertação, tais como: os aceleradores Field-Programmable Gate Arrays (FPGAs), Cell Broadband Engine (Cell BE), General-Purpose Graphics Processing Units (GPGPU), NVIDIA GT200 e GT 300, AMD RV770 e Intel Larrabee; os ambientes de desenvolvimento Compute Unified Device Architecture (CUDA), OpenCL. No terceiro capítulo encontra-se uma breve descrição acerca do tema major da presente dissertação, ou seja, acerca da identificação de objectos em imagens tomográficas, incluindo as soluções sequenciais e as soluções paralelas.

De seguida, encontra-se o quarto capítulo que corresponde à concepção dos algoritmos, que inclui as fases de decomposição do volume de dados em blocos, processamento, identificação de subobjectos, fusão de subobjectos, e fusão de identificadores de blocos distintos.

Posteriormente, apresenta-se a implementação dos algoritmos *One-Pass*, *Two-Pass* e *Object Identifier*, englobando uma descrição pormenorizada de todos os aspectos relacionados com os mesmos.

Seguidamente, surge o capítulo referente à avaliação, na qual se realiza uma descrição do *hardware*, do *software* e dos volumes de dados utilizados, seguida da avaliação realizada aos algoritmos desenvolvidos.

Por fim, encontra-se a conclusão da dissertação e algumas sugestões / hipóteses de trabalho futuro passível de ser realizado.

2

Estado da arte

No presente capítulo será realizada uma abordagem geral das áreas científicas relevantes para a elaboração da dissertação. Como o seu principal objectivo consiste na implementação de algoritmos para identificação de objectos em imagens tomográficas em plataformas de computação heterogénea, começa-se por discutir o *hardware* e o *software* que as compõem.

2.1 Aceleradores

No ano de 1965, Moore previu que o número de transístores colocados num único *chip* iria aumentar para o dobro em cada ano [26]. Até recente data, a frequência dos processadores tradicionais tem vindo a evoluir de acordo com a lei de Moore. Contudo, as limitações físicas suspenderam, e até inverteram ligeiramente, esse crescimento exponencial. A chave dessa restrição designa-se por *power density* [21], isto é, a energia dissipada por unidade de área. Essa energia dissipada nos processadores aproxima-se do limite físico que o silício pode suportar quando submetido a técnicas de arrefecimento. Para dar continuidade à referida técnica é imprescindível utilizar novas técnicas de arrefecimento como o azoto líquido; no entanto, essas técnicas são bastante dispendiosas [5].

A frequência não pode ser incrementada com a tecnologia actual, pelo que o aumento da performance é conseguida pelo aumento do número de transístores. Uma forma de explorar este maior número de transístores é adicionar *cores* ao *chip*, que comparativamente a um *single-core* permite um aumento da performance de 180%, utilizando apenas 85% da frequência e da potência [21].

Actualmente, CPUs *multi-cores* utilizam muitos dos seus transístores na unidade de controlo e em *caches*, sendo muita energia utilizada em unidades não computacionais. O

aumento do número de *cores* por si só, também causa problemas [15], sendo um deles o aumento do número de pedidos de acesso à memória em simultâneo, o que se traduz no aumento de pressão sobre a mesma [5].

Uma outra opção seria a exploração do paralelismo ao nível das instruções, estando este geralmente associado a processadores superescalares [43]. Essa solução apresenta a desvantagem de o *hardware* requerer cada vez mais recursos, como *instruction window* e *reorder buffers*, para resolver problemas relacionados com as dependências das instruções [8].

A combinação de arquitecturas tradicionais *multi-core* com aceleradores tem vindo a constituir uma alternativa às opções descritas anteriormente.

Um acelerador consiste num dispositivo *hardware* destinado à realização de operações específicas, que é adicionado aos sistemas de computação tradicionais [8]. Muitas vezes os *cores* dos aceleradores utilizam menos transístores e executam a frequências mais baixas do que os *cores* dos processadores tradicionais. Esta característica permite que no mesmo espaço de silício seja possível incorporar mais *cores* do que os usados nos CPUs tradicionais [5].

Existem também soluções baseadas em *Application-Specific Integrated Circuits* (ASICs) [37] que são dispositivos desenvolvidos especificamente para estarem adaptados a uma determinada tarefa, ou a um conjunto limitado de tarefas, não podendo ser novamente reconfigurados, o que os torna inadequados para aplicações em *High-performance computing* (HPC). Devido a essa desvantagem existe uma outra categoria designada FPGA, a qual permite a reconfiguração. Esta técnica será abordada na subsecção 2.1.1 do presente capítulo.

De entre os vários tipos de aceleradores salientam-se três: *Field Programmable Gate Array* (FPGA), *Cell Broadband Engine* (Cell BE) e *General-Purpose Graphics Processing Units* (GPGPU). Os mesmos serão descritos seguidamente.

2.1.1 Field-Programmable Gate Arrays (FPGAs)

Tal como um ASIC, um FPGA é uma implementação de um algoritmo em *hardware*; ao contrário do ASIC, esta solução após programada não fica permanente no *chip*, podendo assim ser reprogramada inúmeras vezes [50].

Tal como se encontra ilustrado na figura 2.1, a arquitectura de um FPGA é composta por blocos lógicos interligados por uma estrutura que permite um encaminhamento. Os referidos blocos contêm elementos de processamento que realizam operações simples, contendo também *flip-flops* para implementação de lógica sequencial. Relativamente à estrutura de ligação, esta permite interligar elementos lógicos da forma desejada [13]. Esta arquitectura é bastante genérica e flexível, permitindo assim um número elevado de combinações possíveis de circuitos a implementar. Todos estes elementos são programados através da tecnologia *Flash* ou *Synchronous Dynamic Random Access Memory* (SDRAM). Aos FPGAs é possível ainda adicionar memórias, multiplicadores, *hardware*

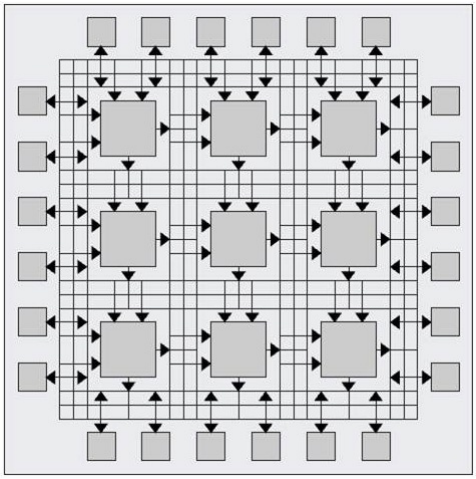


Figura 2.1: Representação abstracta de um FPGA [13].

para realizar funções aritméticas, e lógicas e até micro processadores [13].

O Xilinx Virtex [44] corresponde a um FPGA que utiliza SDRAM, composto por um conjunto de elementos heterogéneos, interligados através de uma malha de interligação complexa. Os elementos, por sua vez, são compostos por *Configurable Logic Blocks* (CLBs), *Digital Signal Processing Blocks* (DSPs), *Block RAM*, e *Input/Output Blocks* (IOBs).

Os CLBs e os DSPs implementam a semântica dos blocos lógicos, sendo unidades semelhantes às *Arithmetic Logic Unit* (ALU) [44].

Em termos de operações de vírgula flutuante, o desempenho do Virtex-6 SX475T FPGA pode atingir 116 Gflops com um consumo de aproximadamente dez watts, ao contrário dos tradicionais CPUs que consomem centenas de watts. A razão desses consumos é definida pela menor frequência utilizada, sendo de 100 a 300 MHz a utilizada pelo FPGA e de 2 a 3 GHz a utilizada pelos CPUs [44].

Devido ao paralelismo existente nas aplicações deste FPGA e à frequência por este utilizada, em certos problemas, existe um aumento de velocidade de aproximadamente dez vezes mais do que a velocidade dos CPUs, consumindo quatro vezes menos energia [44]. Em algumas aplicações os FPGAs permitem diminuir os tempos de execução em cerca de 240 vezes face aos CPUs [44].

Apesar das vantagens apontadas, o uso desta solução só é justificável economicamente se o sistema a desenvolver for produzido em média ou longa escala.

2.1.2 Cell Broadband Engine (Cell BE)

O Cell BE é a primeira implementação da arquitectura *Cell Broadband Engine Architecture* (CBEA), consistindo num *chip* com cerca de 241 milhões de transístores numa área de 235 mm^2 com nove *cores* que executam segundo um modelo de memória partilhada, interligados através de um *bus*. O desempenho em operações de vírgula flutuante do Cell pode atingir cerca de 204.8 Gflops em *single precision* e 14.6 Gflops em *double precision*,

Ao Cell BE está associado um conjunto de *software* que suporta o desenvolvimento

de aplicações. Essas ferramentas, além de *debuggers*, incluem também utensílios para monitorizar diversos aspectos desta arquitectura heterogénea, que são relevantes para a diminuição do tempo de execução das aplicações.

Esta arquitectura é composta por vários componentes, tal como se pode observar na figura 2.2. De seguida apresenta-se uma breve descrição de cada um deles.

O *Memory Interface Controller* (MIC) apresenta como função interligar, através de dois canais, o sistema de memória aos restantes componentes do *chip*. A memória designa-se *eXtreme Data Rate Dynamic Random Access Memory* (XDR DRAM) e efectua oito transferências de dados por *clock cycle*, ao invés da tradicional DRAM que efectua duas ou quatro, isto faz com que não se necessite de frequências de *clock* elevadas [39]. Este controlador suporta uma largura de banda de 25.6 Gbytes/s na ligação à memória central [11].

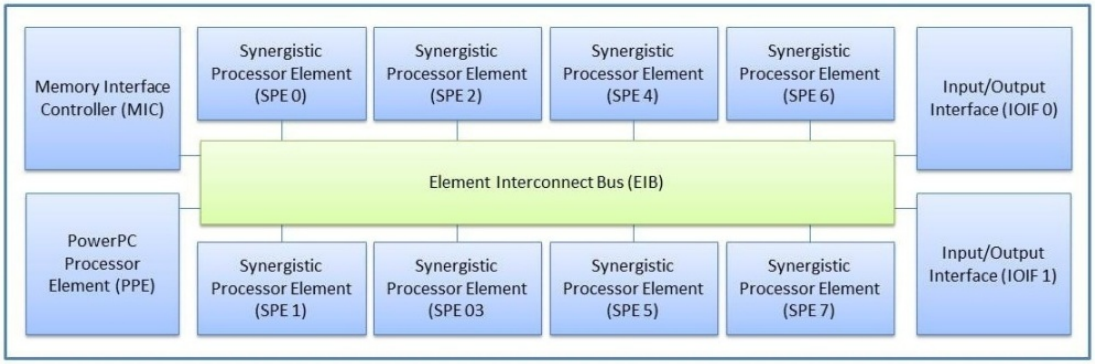


Figura 2.2: Arquitectura Cell BE [4].

O *PowerPC Processor Element* (PPE) é composto por um *PowerPC Processor Unit* (PPU) e por um *PowerPC Processor Storage Subsystem* (PPSS). O PPU é uma arquitectura RISC *PowerPC* de 64 bits que suporta a execução de dois *threads* em simultâneo *symmetric multithreading* (STM). Este possui também uma unidade *VMX engine* para processamento de vírgula flutuante, suportando instruções *Single Instruction, Multiple Data* (SIMD). O PPU inclui duas *caches* L1 de 32 KB, sendo uma para instruções e a outra para dados [39]. O PPSS inclui uma *cache* unificada L2 de 512 KB, e tem como função a ligação do PPU aos restantes componentes do processador [4].

O *Input / Output Interface* (IOIF) tem como função ligar o sistema a periféricos exteriores. Cada IOIF baseia-se na tecnologia Rambus FlexIO, com uma largura de banda total próxima de 76.8GB/s [39].

O *Synergistic Processor Element* (SPE) é composto por dois módulos, o *Synergistic Processor Unit* (SPU) e o *Memory Flow Controller* (MFC). Os SPUs são compostos por *cores* RISC, desenhados especialmente para executar operações SIMD. Cada SPU é composto por dois *pipelines* paralelos que executam a 3.1 GHz. Estes possuem 256KB de memória local para guardar instruções e dados, e possuem também registos de 128 bits. A função mais importante do MFC é controlar o *Direct Access Memory* (DMA), visto ser este o responsável por mover bytes da memória central para a memória local [39].

O *Element Interconnect Bus* (EIB) tem uma largura de banda de 204.8 Gbytes/s para

transferências internas ao *chip*. Este componente é constituído por quatro anéis de comunicação, que transportam os pedidos no sentido horário, relativos ao DMA entre todos os componentes, e por dois anéis em sentido inverso [11]. Cada anel tem uma largura de banda de 16 bytes e suporta três transferências de dados em simultâneo [39].

O Cell desempenhou um papel importante na chamada de atenção para o potencial dos aceleradores; também foi pioneiro na exploração de arquitecturas de *cores* heterogéneos, em que cada *core* executa as partes de um programa que lhe são mais adequadas.

Embora o potencial deste processador seja elevado, as dificuldades na sua programação acabaram por levar a IBM/Sony/Toshiba a abandonarem o seu desenvolvimento.

2.1.3 General-Purpose Graphics Processing Units (GPGPU)

A motivação inicial para os GPUs serem desenvolvidos foram os jogos de computadores; em geral, os GPUs executam *rendering* de imagens em 2D e 3D, libertando o CPU dessa tarefa. Actualmente, os GPUs são utilizados de forma mais abrangente, designando-se assim *General-Purpose Graphics Processing Units* (GPGPUs), não sendo totalmente dedicados a *rendering* de imagens. Estes possuem centenas de *cores*, que conseguem executar dezenas de milhares de *threads* em paralelo. A referência [5] prova que a performance dos GPGPUs se aproxima dos três teraflops, constituindo assim uma mais-valia para HPC [5].

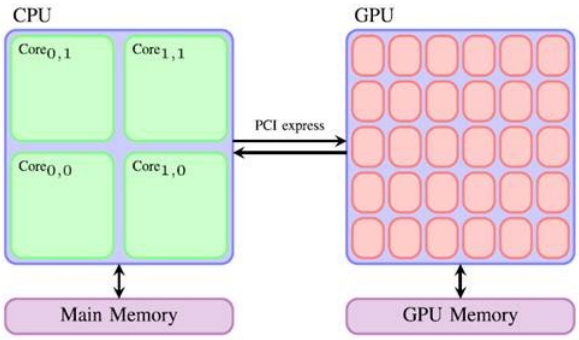


Figura 2.3: Combinação do CPU e GPGPU [5].

O GPGPU é constituído em torno de um processador simétrico *multi-core* que é controlado pelo CPU, tal como se pode observar na figura 2.3. O GPGPU funciona de modo autónomo em relação ao CPU, tratando-se assim de uma computação heterogénea, existindo concorrência na execução e na transferência entre memórias [5].

Um factor importante a salientar na performance dos GPGPUs diz respeito ao tempo de transferência da informação entre o CPU e o GPGPU. Em aplicações onde a quantidade de dados a transferir é diminuta e a complexidade de computação é elevada, torna-se possível atingir *speedups* elevados no que concerne a soluções optimizadas que utilizam CPUs *multi-cores*. Um exemplo disso é a simulação de *Monte Carlo* da equação *Black-Scholes* [22] na qual a capacidade computacional do NVIDIA S870 supera 23 vezes a de um *core* CPU e 3 vezes uma plataforma com 8 *cores* CPU; o NVIDIA S870 um 1U *rack* que contém quatro GPGPUs NVIDIA C870 [22].

Quando o volume de dados necessário a transferir entre CPU e GPGPU aumenta, a performance do GPGPU diminui podendo ficar inferior à do CPU. Na simulação realizada para calcular a transformação discreta de Fourier [22] é possível observar que até um determinado ponto os CPUs obtêm uma performance superior à dos GPGPUs. Esses resultados devem-se ao *overhead* causado pela transferência de dados entre CPU e GPGPU. O GPGPU obtém uma performance superior quando a capacidade computacional proveniente do paralelismo compensa todos os *overheads* [22].

Devido às vantagens da combinação entre CPUs e GPUs, actualmente, surgiu uma nova tecnologia designada *Accelerated Processing Units* (APU), que consiste na referida combinação no mesmo *chip*. Essa tecnologia apresenta como grande vantagem a redução do *overhead*, relativo à transferência entre CPU e GPU, visto que utiliza *hypertransport* como meio de comunicação [3].

A AMD possui uma arquitectura APU, designada AMD Fusion [3], utilizada nos APUs Llano e Ontario da AMD. A mesma pode ser programada através das plataformas DirectCompute e OpenCL.

A Intel também possui uma arquitectura semelhante designada Intel Microarchitecture ou SANDY BRIDGE [18].

De entre várias arquitecturas de GPGPUs destacam-se três: NVIDIA GT200 [30], NVIDIA Fermi [10] e AMD RV770 [2]. Além das referidas, existe a Intel Larrabee [40] e a Intel Knights Ferry, que consistem em arquitecturas híbridas entre um *multi-core* CPU e um GPGPU. Seguidamente serão abordadas apenas as arquitecturas referidas acima.

2.1.3.1 NVIDIA GT200 e GT 300

A arquitectura GT200 [30], ilustrada na figura 2.4, é implementada com 1400 milhões de transístores com cerca de 240 *cores* CUDA. Esta é programada através de CUDA segundo um modelo SPMD, utilizando um grande número de *threads* organizados em blocos. Estes blocos são divididos em *warps*, que são constituídos por 32 *threads*, e podem comunicar e sincronizar-se entre si utilizando memória partilhada. Estes são escalonados para os *Streaming Multiprocessors* (SM) em tempo de execução, sendo cada um executado de acordo com o modelo SIMD. Um SM é constituído por oito *Scalar Processors* (SP), contendo ainda uma *Double Precision Unit*, duas *Special Functions Units*, 16KB de memória partilhada, 8 KB de *cache* de memória constante e registos de 32 bits [5].

O *hardware* é responsável por resolver ocorrências de fluxos de código divergentes em *threads*, o que se traduz numa diminuição de performance quando os *threads* pertencem ao mesmo *warp*. Caso a divergência ocorra entre *threads* de *warps* diferentes, não existe qualquer impacto na performance [5].

Todos os *threads* do mesmo bloco têm acesso à mesma memória partilhada, caso se encontrem dentro do mesmo *warp* esse acesso é sincronizado de forma automática, caso apenas pertençam ao mesmo bloco, necessitam de barreiras de sincronização [5].

O SM mantém activas muitas *warps*, o que permite alterá-las automaticamente sem

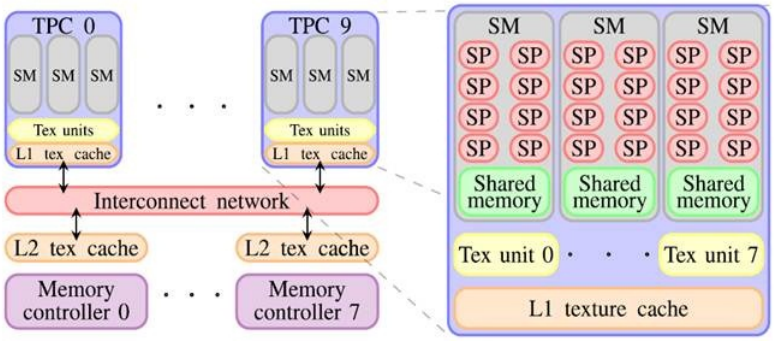


Figura 2.4: Arquitectura NVIDIA GT200 [5].

GPGPU	GT200	GT300
Transístores	1.4 Milhões	3.0 Milhões
CUDA Cores	240	512
Double Precision Floating Point Capability	30 FMA ops/clock	256 FMA ops/clock
Single Precision Floating Point Capability	240 MAD ops/clock	512 FMA ops/clock
Special Function Units (SFUs) / SM	2	4
Warp schedulers (por SM)	1	2
Memória Partilhada (por SM)	16 KB	48 KB ou 16 KB
L1 Cache (por SM)	Apenas para texturas	16 KB ou 48 KB
L2 Cache	Apenas para texturas	768 KB
ECC Memory Support	Não	Sim
Concurrent Kernels	Não	Até 16
Load/Store Address Width	32-bit	64-bit

Tabela 2.1: Comparação entre as arquitecturas GT200 e GT300 [10].

overheads, característica que constitui uma enorme vantagem para o aumento da performance. Os registos e a memória partilhada restringem o número de *warps* que cada multiprocessador pode executar [5].

A arquitectura GT200 possui oito controladores de memória de 64 bits, constituindo um agregado de 512 bits de interface para a memória central do GPGPU [5].

A NVIDIA possui uma nova arquitectura designada GT300, mais conhecida por Fermi [10]. Esta arquitectura tem como base os mesmos conceitos da GT200, no entanto é implementada com 3000 milhões de transístores com cerca de 512 *cores* CUDA, que por sua vez estão organizados em dezasseis *Streaming Multiprocessor* (SM). Este GPGPU comunica com a memória usando 6 canais de 64 bits, ou seja, uma interface de 384 bits, que pode ir até 6GB de GDDR5 DRAM. Comparativamente à anterior, esta arquitectura apresenta as seguintes melhorias: o número de SP é aproximadamente o dobro; a performance em precisão dupla aumentou significativamente, aproximando-se de metade da velocidade da precisão simples; a memória é protegida por ECC; cada SM possui em *cache* L1 16 ou 32 KB e a *cache* L2 partilhada entre SMs possui 768 KB; é possível a execução de *kernels* concorrentemente, ao contrário da arquitectura GT200; torna mais transparente o acesso à RAM do CPU, o que permite a execução do código C++ directamente no GPGPU [5].

Um SM na arquitectura Fermi é constituído por 23 *cores* CUDA, dezasseis *Load/Stores*

Units (LSU), quatro *Special Functions Units* (SFU), dois *Warp Schedulers*, duas *Dispatch Units*, uma *cache* L1 e uma memória partilhada de 64 KB. Cada *core* CUDA é constituído por uma *Fully Pipelined Arithmetic Logic Unit* (ALU) e por uma *Floating Point Unit* (FPU) [10].

Um LSU apresenta como função permitir que endereços de origem e destino sejam calculados por dezasseis *threads* por *clock*. O SFU é uma unidade cuja função consiste em executar as chamadas instruções transcendentais, como o cálculo do seno e do co-seno. Relativamente aos *Warp Schedulers* e aos *Dispatch Units*, estes são responsáveis pelo escalonamento no interior de um SM [10].

2.1.3.2 AMD RV770

A geração AMD FireStream teve origem na arquitectura RV770 [2], descrita na figura 2.5. A mesma equivale à arquitectura da NVIDIA de rede de blocos, utilizando um modelo SPMD sobre uma rede de grupos [5].

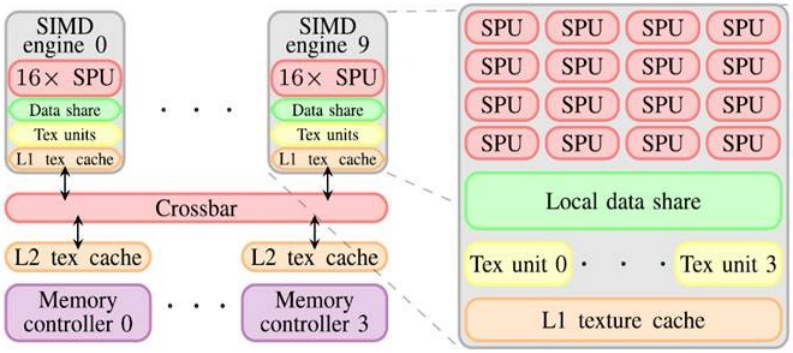


Figura 2.5: Arquitectura AMD RV770 [5].

Todos os grupos processam o mesmo *kernel*, e os *threads* presentes em cada grupo podem comunicar-se e sincronizar-se através da memória local. Cada grupo é constituído por um *array* de *threads* com total independência de fluxo de código, sendo todos eles escalonados para os *SIMD engines* em tempo de execução. Um *SIMD Engine* consiste num *core* da arquitectura RS770, que contém 16 *Shader Processing Uunits* (SPU), 16 KiB de memória local, registos de 32-bits, quatro *Texture Units* e uma *Texture cache* L1 [5].

Os grupos são divididos em *wavefronts* de 64 *threads*, equivalentes aos *warps* da NVIDIA, sendo executadas de acordo com o modelo SIMD [5]. Os SPU introduzem um nível de paralelismo com cinco unidades de precisão simples que são programadas através de *Very Long Instructions Words* (VLIW), e que possibilitam processar aritmética transcendental e precisão dupla [5].

Tal como ocorre na memória partilhada da NVIDIA, também na AMD todos os *threads* dentro de um grupo acedem à mesma memória local através de barreiras de sincronização, não podendo aceder às memórias locais dos restantes grupos. A arquitectura RV770 contém ainda *caches* de instruções e de memória constante, *texture cache* L2, e memória

global partilhada entre *SIMD Engines*. A memória global permite partilhar memória entre *SIMD Engines*, contudo, esta função não se encontra acessível ao programador [5].

Ambas as arquitecturas, NVIDIA GT200 e AMD RV770, são conceptualmente idênticas visto que operam segundo o mesmo modelo de execução. Os *SIMD Engines* da AMD são duas vezes maiores que os do *hardware* da NVIDIA e executam a metade da frequência. A performance das referidas arquitecturas é semelhante, aproximando-se de um teraflop em precisão simples [5].

A AMD possui uma nova arquitectura designada de RV870 [47]. Esta detém algumas diferenças em relação à RV700, as quais se apresentam ilustradas na tabela 2.2, mas que se baseiam essencialmente no dobro do número de componentes no mesmo *chip*, e um aumento significativo no *clock* dos mesmos.

GPGPU	SPs	Texture Units	SIMD Engines	Core Clock	Mem Clock	Mem Type
RV770	800	40	10	750Mhz	900 Mhz	DDR5
RV780	1600	80	20	850Mhz	1200 Mhz	DDR5

Tabela 2.2: Comparação entre AMD RV770 e RV870 [47].

2.1.3.3 Intel Larrabee

A arquitectura Intel Larrabee [40], ilustrada na figura 2.6, é considerada um híbrido entre CPUs e GPGPUs, como já foi referido anteriormente. Esta baseia-se em diversos *cores* que executam uma extensão do conjunto de instruções do Pentium [5].

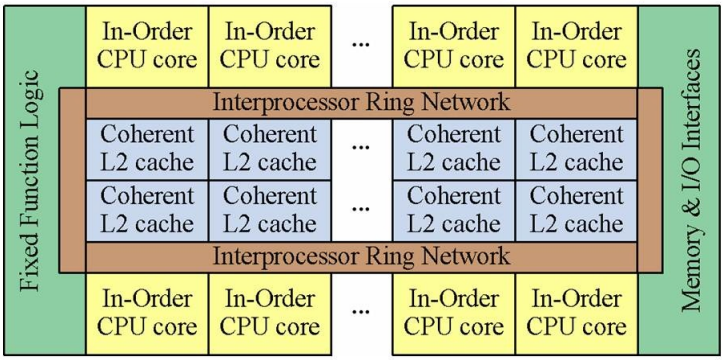


Figura 2.6: Arquitectura Intel Larrabee [40].

Cada *core* consiste num *chip* Pentium com uma unidade vectorial que pode operar 16 elementos de uma vez, que processa instruções em precisão simples e dupla. Cada *core* é super escalar, com uma unidade escalar e outra vectorial, permitindo a execução simultânea de 4 *threads* [5].

A arquitectura Intel Larrabee assegura a coerência de *caches* L2. Os *cores* têm a possibilidade de comunicar e partilhar informação, através de uma rede bidireccional em anel de 1024 bits. Uma característica a salientar desta arquitectura é o facto de esta não

possuir *hardware* dedicado para a geração de imagens como os GPUs, sendo este aspecto implementado por *software* [5].

Actualmente, a Intel possui a arquitectura *Intel Many Integrated Core* (MIC) [42], que vem suceder a Intel Larrabee. A Intel Knights Ferry provém da arquitectura MIC e contém 32 *cores* a 1.2 GHz, podendo ter 128 *threads*. Esta possui também 8 MB de *cache* partilhada coerente e 1 a 2 GB GDDR5 [42].

2.1.4 Comparação

A tabela 2.3 descreve alguns factores comparativos, quantitativos e qualitativos relativos às arquitecturas descritas nesta secção, tendo também a descrição do processador de um CPU Intel Core i7-965 Quad Extreme como referência.

Como é possível observar na tabela, as Unidades SIMD do processador Intel Core i7-965 Quad Extreme têm a capacidade de processar 16 instruções em precisão simples por ciclo a 2.2 GHz, ou seja, 102.4 gigaflops, e cerca de metade em precisão dupla. Este processador consome cerca de 130 watts, sendo cerca de 0.8 instruções em precisão simples por watt. Segundo a sugestão do construtor, o preço deste dispositivo ronda os 999 USD, isto é, cerca de um dólar por 100 megaflops [5].

	CPU	GPGPU (AMD/NVIDIA)	CBEA	FPGA
Composition	Symmetric multicore	Symmetric multicore	Heterogeneous multicore	Heterogeneous multicore
Full cores	4	–	1	2
Accelerator cores	0	10/30	8	2016 DSP slices
Intercore communication	Cache	None	Mailboxes	Explicit wiring
SIMD width	4	64/32	4	Configurable
Additional parallelism	ILP	VLIW/Dual-issue	Dual-issue	Configurable
Float operations per cycle	16	1600/720	36	520
Frequency (GHz)	3.2	0.75/1.3	3.2	<0.55
Single precision gigaflops	102.4	1200/936	230.4	550
Double:single precision performance	1:2	1:5/1:12	1:2	1:4
Gigaflops/watt	0.8	5.5/5	2.5	13.7
Megaflops/USD	70	800/550	>46	138
Accelerator bandwidth (GB/s)	N/A	109/102	204.8	N/A
Main memory bandwidth (GB/s)	25.6	8	25.6	6.4
Maximum memory size (GiB)	24	2/4	16	System dependent
ECC support	Yes	No	Yes	Yes

Tabela 2.3: Propriedades das arquitecturas [5].

No que se refere ao GPGPU NVIDIA Tesla C1060, este é capaz de processar 720 instruções em precisão simples por ciclo a 1.3 GHz, ou seja, 936 gigaflops, e um duodécimo em precisão dupla. Este dispositivo consome no máximo 187.8 watts, sendo cerca de 5 gigaflops por watt. No que diz respeito ao preço, segundo o construtor, este GPGPU custa cerca de 1699 USD, o que significa cerca de 550 megaflops por USD [5].

O GPGPU AMD FireStream 9270 tem a capacidade de efectuar 1600 operações concorrentes por *clock* a 750 MHz, o que resulta em cerca de 1.2 teraflops em precisão simples, e um quinto em precisão dupla. Este GPGPU consome no máximo 220 watt, o que significa aproximadamente 5.5 gigaflops por watt. O preço deste GPGPU é de 1499 USD, o que corresponde a 800 megaflops por USD [5].

Relativamente ao IBM QS22 blade [17], este tem a capacidade de 230 gigaflops em precisão simples, e um pouco menos de metade em precisão dupla. O consumo deste

dispositivo é de 90 watts, o que implica 2.5 gigaflops por watt. O preço deste acelerador é bastante elevado, sendo de 9995 USD, resultando em 46 megaflops por USD [5].

Os FPGAs variam bastante consoante a sua implementação, sendo usados nesta comparação os seguintes: Virtex-5 LX330; Virtex-5 SX240T e Virtex-6 SX475T. O LX330 tem a capacidade de efectuar 230 gigaflops em precisão simples, o SX240T tem a capacidade de 250 gigaflops e o SX475T tem a capacidade de 550 gigaflops. O custo destes dispositivos é bastante elevado, sendo de 1700 USD o LX330, 2000 USD o SX240T e de 4000 USD o SX475T. O consumo é de 17, 18 e 14 gigaflops por watt, respectivamente [5].

Como é possível verificar no artigo [5], os preços do CPU da Intel e dos aceleradores constatados pelos autores do mesmo, na altura da sua realização, diferem bastante entre si.

Na situação actual, o melhor compromisso desempenho/custo/facilidade de programação é apresentado pelos GPGPUs. Como esta arquitectura foi escolhida para o projecto, apresenta-se seguidamente dois ambientes usados no desenvolvimento de aplicações para GPGPUs.

2.2 Ambientes de desenvolvimento

Segundo os autores do artigo [41], inicialmente apenas era possível programar GPUs através de APIs gráficas. No entanto, o aparecimento da linguagem OpenGL Shading Language (GLSL) [38] veio simplificar a forma de programar, embora esta se expresse através de vértice, texturas, fragmentos ou *blending*.

Numa tentativa de desenvolver uma linguagem de programação de alto nível, surgiram duas linguagens designadas BrookGPU [6] e Sh [23]. Estas baseiam-se no modelo de programação por *streaming*, que é um paradigma de programação que está associado ao módulo SIMD do *hardware*. Nesta abordagem, os múltiplos elementos de processamento, como por exemplo os *cores* de um GPU, são usados aplicando a mesma operação a todos os elementos de um conjunto de dados; a vantagem é que não é preciso programar explicitamente a alocação, sincronização e comunicação entre os processadores. Os programas deste modelo são constituídos por conjuntos de fluxos de dados, como forma de *input* e *output*, e por *kernels*. Os *kernels* são executados em paralelo por cada unidade de processamento, onde a comunicação com o dispositivo é efectuada através de *streams* [41].

O projecto Microsoft Accelerator [46] apresenta objectivos semelhantes à linguagem Brook, embora seja suportado por uma linguagem de mais alto nível como o C# e tal como o Sh permite a compilação *just-in-time*. Uma comercialização do Sh, designada RapidMind [24], permite a compilação *just-in-time* e baseia-se na linguagem C++. Esta veio acrescentar heterogeneidade, podendo suportar vários tipos de dispositivos [41].

O sistema PeakStream [34] é inspirado na linguagem Brook e vocacionado para suportar operações em *arrays*. É efectuada a compilação *just-in-time*, no qual o código é vectorizado de forma a maximizar a performance em arquitecturas SIMD [41].

A AMD lançou o seu sistema no final de 2006, designado *Close To Metal* (CTM), que permite ao programador o acesso a funções de baixo nível nos GPGPU da ATI da série R5XX e R6XX. A AMD também disponibilizou uma camada de abstracção a mais alto nível semelhante ao Brook, embora também suporte Brook directamente na arquitectura R6XX [41].

Em seguida encontram-se descritas as duas arquitecturas mais utilizadas actualmente na programação de GPGPUs, denominadas Compute Unified Device Architecture (CUDA) e OpenCL.

2.2.1 Compute Unified Device Architecture (CUDA)

CUDA [29] é uma arquitectura *hardware* e *software* para o desenvolvimento e execução de aplicações em GPGPUs, permitindo o seu uso em computação paralela, sem a necessidade de recorrer a uma API orientada para as operações gráficas.

A pilha do *hardware* CUDA é composta por diversas camadas, como está ilustrado na figura 2.7, sendo essas: interligação entre o sistema operativo e o GPU que é realizada através de um *device driver*; interface de programação; ambiente de execução (*runtime environment*); e várias bibliotecas, por exemplo CUFFT [31] e CUBLAS [28]. Existem dois aspectos importantes a salientar, um refere-se à interface de programação que consiste numa extensão à linguagem de programação C, e o outro reside no facto de o dispositivo *hardware* ter de obedecer a um dado conjunto de requisitos para suportar esta arquitectura [29].

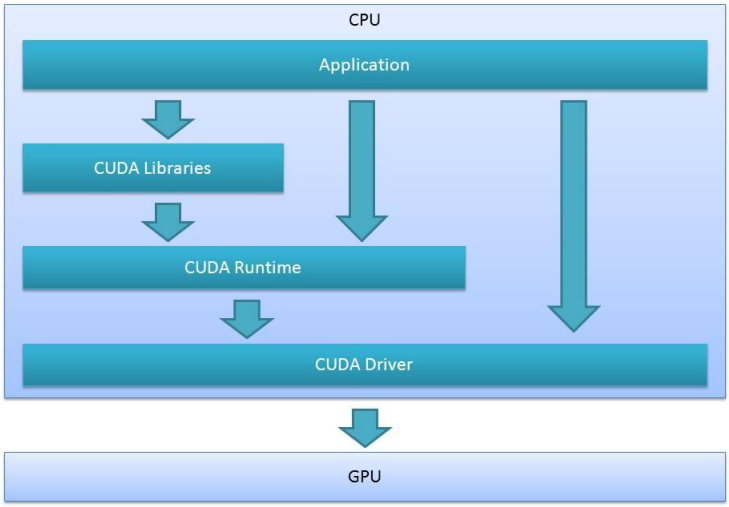


Figura 2.7: Compute Unified Device Architecture Software Stack [29].

A arquitectura CUDA expõe ao programador uma hierarquia de memória que inclui: a memória do CPU; a memória do global do GPU; a memória de texturas; e a memória privada dos SMs.

2.2.1.1 Modelo de programação

Um programa é composto por duas partes, sendo uma ANSI C que executa no CPU, e outra que executa num dispositivo como GPGPU, escrita em CUDA. Esta segunda parte é compilada utilizando o compilador NVIDIA C *compiler* (nvcc), e consiste numa extensão ao ANSI C, como já foi referido anteriormente, a qual é composta por um conjunto de funções que executam em paralelo designadas por *kernels* [20]. Os *kernels* dão origem a um grande número de *threads* executados em paralelo. Os *threads* que executam num *kernel* são organizados numa grelha de blocos, de igual tamanho, endereçados através de um *array* de uma ou duas dimensões, designado por *block ID*, tal como se pode observar na figura 2.8. Cada bloco é constituído por um conjunto limitado de *threads*, endereçados através de um *array* de uma, duas ou três dimensões, designado por *thread ID* [29].

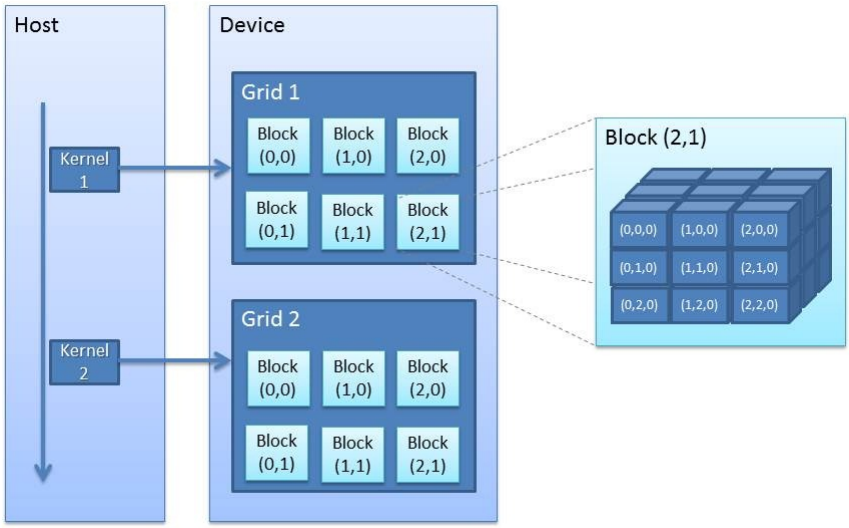


Figura 2.8: *Thread Batching* [20].

Os blocos de *threads* executam o mesmo *kernel*, segundo um modelo SPMD, podendo cooperar eficientemente através de partilha de dados, utilizando memória partilhada. Estes também se podem sincronizar para coordenar os acessos à memória, permitindo assim especificar pontos de sincronização no *kernel*. Contudo, *threads* de blocos diferentes não conseguem comunicar entre si nem sincronizar-se. Este modelo permite executar *kernels* eficientemente em vários dispositivos sendo o tempo de execução dependente do número de *cores* que o *hardware* disponibiliza [29].

2.2.1.2 Modelo de memória e transferência de dados

Existem diversos espaços de memória disponíveis a serem acedidos pelos *threads*, como está ilustrado na figura 2.9. Como é possível observar, existe um conjunto de registos e uma memória local por *thread*. Um *thread* apresenta também disponível uma memória partilhada entre *threads* do mesmo bloco. Existem também outras três memórias acedidas

por *threads* da mesma grelha, podendo também ser acedidas pelo *host*, sendo estas: uma memória global; uma memória constante; e uma memória para texturas [29].

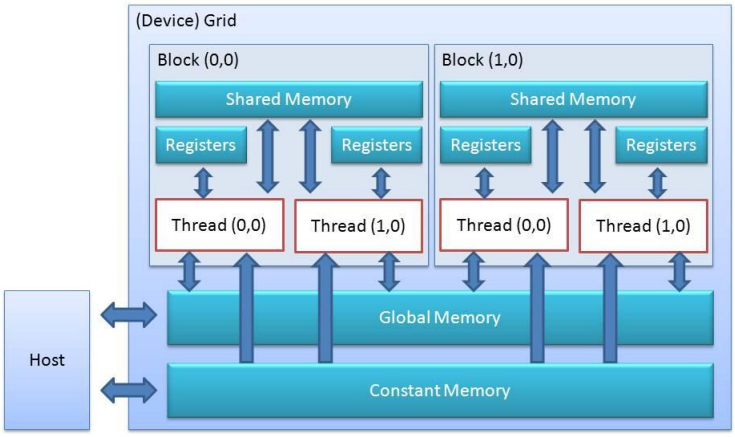


Figura 2.9: Modelo de memória [20]

A transferência de dados entre o programa que executa no *host* e o que executa no dispositivo CUDA é efectuada através da memória global ou da memória constante, não tendo acesso à memória partilhada entre blocos, nem à memória local de cada *thread*. Relativamente à memória constante, esta apenas permite operações de leitura [20].

2.2.1.3 Modelo de execução

A execução de programas CUDA tem início no *host* (CPU). Quando um *kernel* é invocado, a execução é efectuada no dispositivo como GPGPU, através de um grande número de *threads* executados em paralelo. Após todos os *threads* terminarem a sua execução no dispositivo, o programa do *host* continua até ser invocado um novo *kernel* [20]. O dispositivo efectua o escalonamento de blocos de *threads* para os seus multiprocessadores, que os executam de modo paralelo. Cada bloco de *threads* pode ser dividido em grupos de *threads* que executam segundo um modelo SIMD, designados *warps* [29].

2.2.2 OpenCL

O OpenCL (Open Computing Language) [27] é uma norma para programação paralela de uso geral, incorporando CPUs, GPGPUs e outros processadores, permitindo desenvolver *software* que executa em diferentes plataformas sem necessidade de alterar o código fonte. Este sistema pode ser usado também para o desenvolvimento de aplicações gráficas que conjugam algoritmos de computação paralela com *pipelines* gráficos [27].

O OpenCL é uma API que fornece uma linguagem de programação multi-plataforma que: suporta modelos de programação baseados em dados e tarefas; adopta um subconjunto na norma ISO C99 com uma extensão para paralelismo; está de acordo com a norma IEEE 754 para vírgula flutuante; define um perfil de configuração para dispositivos embebidos; inter-opera de forma eficiente com OpenGL, OpenGL ES e outras APIs gráficas

[27].

Para a descrição do OpenCL é possível enumerar quatro modelos hierárquicos: modelo da plataforma; modelo de memória; modelo de execução; e modelo de programação. Os mesmos serão descritos seguidamente, sendo também abordado um aspecto importante para a compreensão do funcionamento do OpenCL, o conceito de *memory objects*.

A *framework* OpenCL é constituída pelos componentes OpenCL Platform layer, OpenCL Runtime e OpenCL Compiler. A platform layer possibilita que um programa *host* procure OpenCL devices e visualize as suas definições para originar contextos. O *runtime* permite que um programa *host* manipule contextos apenas por ele originados. O OpenCL compiler cria os executáveis que possuem os OpenCL *kernel*. A linguagem de programação OpenCL C é implementada por um compilador que suporta um subconjunto da linguagem ISO C99 com uma extensão para paralelismo [27].

2.2.2.1 Modelo da plataforma

O modelo da plataforma encontra-se ilustrado na figura 2.10, na qual é possível visualizar que este consiste num *host* ligado a um ou mais dispositivos. Um *OpenCL device* é composto por um ou mais *compute units* (CUs), que por sua vez são constituídas por um ou mais *processing elements* (PEs) nos quais é efectuado o processamento [27].

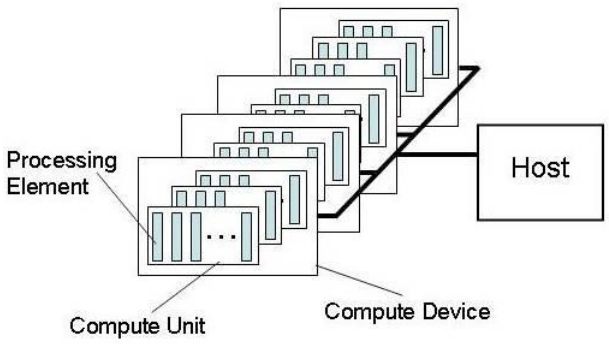


Figura 2.10: Modelo da plataforma [27].

Os comandos submetidos pelas aplicações OpenCL são executados nos *processing elements*, através do *host*. Os *processing elements* executam um só fluxo de instruções segundo o modelo SIMD, ou o modelo SPMD [27].

A plataforma OpenCL foi desenvolvida para comportar dispositivos com características distintas numa única plataforma. Esses dispositivos apresentam diversas versões da especificação OpenCL, sendo de salientar a versão da plataforma, a versão do dispositivo e a versão da linguagem OpenCL C [27].

2.2.2.2 Modelo de execução

Existem duas partes a considerar na execução de programas OpenCL, *kernels* que executam num ou mais *OpenCL devices*, e *host program* que executa num *host*. O contexto e a gestão da execução do *kernel* é determinado pelo *host program* [27].

Quando um *kernel* é executado implica a definição de um espaço de endereçamento, sendo que em cada ponto desse espaço é executada uma instância do *kernel* (*work-item*), identificado por um *global ID*. Cada *work-item* executa o mesmo código, porém, este pode enverdar por diferentes caminhos de execução, sendo estes definidos pelo código e os dados operados, que podem variar entre *work-items* [27].

Os *work-items* estão organizados em *work-groups* que constituem um grão mais elevado de decomposição do espaço de endereçamento, sendo identificados por um *group ID*. Estes associam-se a um só *local ID* dentro de um *work-group*, podendo assim cada *work-item* ser identificado univocamente pelo *global ID* ou pela junção do *local ID* com o *group ID*. Os *work-items* num determinado *work-group* executam, de forma concorrente, em *processing elements* num único *compute unit* [27].

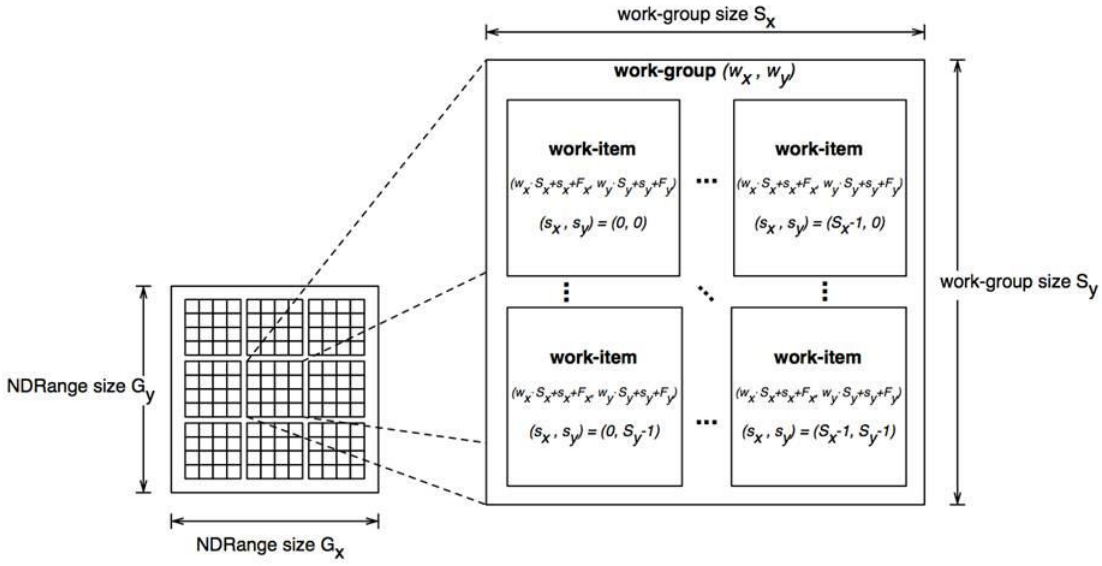


Figura 2.11: Espaço de endereçamento multidimensional [27].

NDRange é a designação dada ao espaço de endereçamento multidimensional do OpenCL, podendo suportar entre uma a três dimensões, como se pode observar na figura 2.11 [27].

2.2.2.3 Modelo de memória

Na execução de um *kernel* existem quatro regiões diferentes de memória - como está ilustrado na figura 2.12 - às quais os *work-items* têm acesso: a memória global, que possibilita o acesso de leitura e escrita para todos os *work-items* de todos os *work-groups*; a memória

constante, que corresponde a uma memória global mas que se mantém constante durante a execução do *kernel*; a memória local, que é partilhada por *work-items* do mesmo *work-group*; e a memória privada, que é somente visível ao *work-item* associado [27].

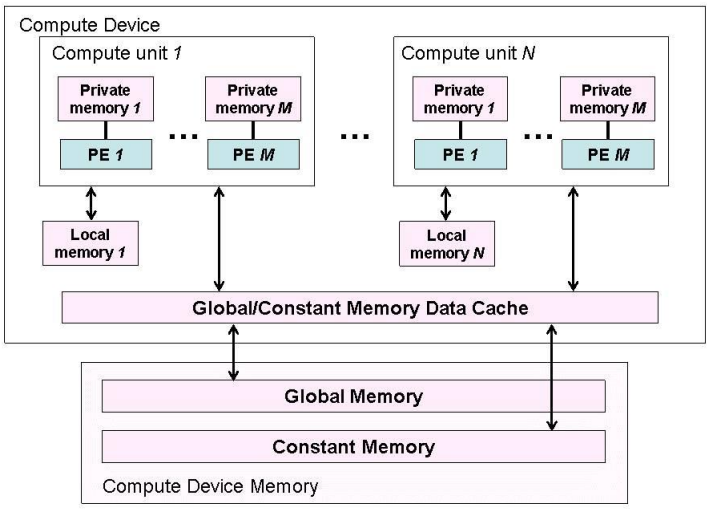


Figura 2.12: Arquitectura de um dispositivo OpenCL [27].

A referir que a memória local é implementada por regiões dedicadas da memória nos *OpenCL device*, caso isso não seja possível, esta é mapeada em secções da memória global [27].

2.2.2.4 Modelo de programação

O modelo de execução suporta os modelos de programação *data parallel* e *task parallel*, bem como uma combinação dos mesmos [27].

O modelo de programação *data parallel* diz respeito à computação de sequências de instruções aplicadas a múltiplos elementos em memória. O espaço de endereçamento que lhe está associado determina os *work-items* e o mapeamento da sua informação. Este modelo é definido segundo uma hierarquia, na qual existem duas formas de especificar a sua divisão: o modelo explícito, no qual o programador selecciona a quantidade de *work-items* a executar em paralelo e também a sua forma de divisão; e o modelo implícito, no qual o programador apesar de também definir o número de *work-items* a executar em paralelo, delega à implementação OpenCL a divisão por *work-groups* [27].

O modelo de programação *task parallel* define que uma instância do *kernel* é executada independentemente de qualquer espaço de endereçamento, o que corresponde à execução do *kernel* numa *compute unit* com um *work-group* constituído apenas por um *work-item* [27].

2.2.2.5 Objectos de memória

Os objectos de memória apresentam como função o armazenamento de informação a ser utilizada pelos *kernels* OpenCL. Estes podem ser do tipo *buffer* ou do tipo *image*. Os objectos do tipo *buffer* são compostos por elementos de uma só dimensão, que variam entre escalares, vectores ou estruturas definidas pelo utilizador. Os objectos do tipo *image* podem armazenar texturas de duas e três dimensões, bem como *frame-buffers* ou *imagens* [27].

Uma das grandes diferenças entre os objectos consiste na forma de acesso, a qual é efectuada sequencialmente nos objectos do tipo *buffer*, através de um apontador. Os objectos do tipo *image* são armazenados de forma opaca para o utilizador, não podendo ser acedidos directamente através de apontadores. Para aceder a esses elementos, é necessário utilizar funções do OpenCL [27].

Uma outra diferença reside na forma de armazenamento dos elementos, sendo esta igual à acedida pelo *kernel* nos objectos do tipo *buffer*. Relativamente aos objectos do tipo *image*, o formato usado para guardar os elementos pode não ser o mesmo que o utilizado dentro do *kernel*, que normalmente utiliza vectores de quatro componentes [27].

2.2.3 Comparação CUDA e OpenCL

Segundo a análise presente no artigo [19], ambos os ambientes de programação abordados anteriormente, OpenCL e CUDA, oferecem interfaces de programação para GPGPUs, embora o OpenCL seja mais abrangente, o que possibilita que seja utilizado também em outros tipos de dispositivos.

A arquitectura CUDA pertence à NVIDIA, ao contrário do OpenCL que é uma norma aberta proposta pela Khronos Group. Um aspecto importante a salientar do ambiente CUDA diz respeito à possibilidade de programação a diversos níveis, da camada existente entre a aplicação e o GPGPU, como já foi referido na secção relativa ao CUDA, o que fornece ao programador a possibilidade de programar directamente com o *driver*, usufruindo de um acesso directo ao dispositivo *hardware*. Um outro aspecto importante está relacionado com a transferência de dados entre o CPU e o GPGPU, sendo que o OpenCL permite definir programas em que diferentes componentes executam em diferentes dispositivos, sendo necessário utilizar uma representação de dados independente do dispositivo, o que adiciona *overheads* à transferência de dados [19].

Outro aspecto relaciona-se com a portabilidade do código em OpenCL pois este, ao contrário do CUDA, permite a compilação do código *just-in-time*. Esse aspecto é interessante porque permite utilizar um mesmo programa em diversos dispositivos diferentes, visto que o código para os mesmos é apenas compilado na altura da execução. A desvantagem dessa técnica diz respeito ao acréscimo de tempo de execução [19].

O artigo [19] contém uma comparação entre estas duas arquitecturas, que executam

no mesmo GPGPU com *kernels* semelhantes. A aplicação utilizada é designada de *Adiabatic QUantum Algorithms* (AQUA), que se baseia numa simulação de Monte Carlo. Segundo os resultados do artigo, a execução dos *kernels* em OpenCL apresenta uma performance inferior, que varia entre os 13% e os 63% em relação ao CUDA. Existe também uma diminuição de performance na execução *end-to-end* das aplicações em OpenCL, que varia entre 16% e 67%. Relativamente ao tempo de transferência entre CPU e GPGPU, a diferença entre o OpenCL e o CUDA é também significativa [19].

Embora na arquitectura CUDA se tenham obtido melhores resultados, é necessário ter em consideração que o GPGPU utilizado é NVIDIA GeForce GTX-260, sendo este desenhado para ser preferencialmente programado através de CUDA. Logo, a arquitectura CUDA ajusta-se melhor ao *hardware* do que o OpenCL, que foi desenvolvido para ser genérico [19].

Em suma, o CUDA apresenta uma maior eficiência comparativamente ao OpenCL, mas apresenta as desvantagens de ser uma arquitectura homogénea, ao contrário do OpenCL que é heterogénea, e de não possuir as características de portabilidade que o OpenCL possui. Desta forma, a selecção entre estas duas arquitecturas é efectuada consoante a necessidade da aplicação.

2.3 Dificuldade na programação com GPGPUs

A programação para estes dispositivos apresenta diversos aspectos que a tornam bastante difícil. Um desses pontos diz respeito à necessidade de realizar um novo código, quando se migra um algoritmo sequencial ou paralelo regular para GPGPUs. Um outro aspecto é referente à existência de apenas instruções básicas, fazendo assim com que a programação seja efectuada a um baixo nível. Além dos aspectos já referidos, a principal dificuldade reside nas optimizações a realizar para se poder melhorar o desempenho de um dado algoritmo. De forma a optimizar os algoritmos desenvolvidos para GPGPU, é necessário ter diversos aspectos em consideração, tais como: a organização dos *threads*; a utilização das hierarquias de memória; a forma de aceder à memória global; a comunicação entre o CPU e o GPGPU; e a potencial divergência entre os fluxos de controlo.

Estes aspectos têm de ser considerados quando se pretende ajustar os algoritmos aos dispositivos, de forma a tirar o máximo partido do *hardware*.

2.3.1 Organização dos *threads*

A organização dos *threads* consiste em definir a dimensão dos grupos, de forma a que estes possam ser processados em simultâneo no mesmo multiprocessador. Este valor deverá ser um múltiplo da dimensão dos *warp* de *threads* do *scheduler* do multiprocessador.

Quando este ajuste não ocorre, o *scheduler* preenche os *warps* com *threads* que não realizam qualquer processamento, desperdiçando assim capacidade de processamento.

Um outro aspecto importante de referir sobre os *warps* diz respeito aos recursos utilizados pelos *threads*, tais como o número de registos e a memória constante. Caso a soma dos recursos utilizados pelos *warps* seja superior aos recursos disponibilizados pelo multiprocessador, o *scheduler* não coloca em execução o número máximo de *warps* suportados pelo multiprocessador.

Para se poderem definir os valores correctos para maximizar o número de *threads* a serem executados em simultâneo num dado multiprocessador, a nVidia disponibiliza uma ferramenta (folha de cálculo) que dadas as características do multiprocessador e os recursos utilizados por cada *thread*, devolve a taxa de ocupação máxima.

2.3.2 Hierarquias de memória

Relativamente às hierarquias de memória, a sua análise é pertinente para optimizar os algoritmos, dado que as latências no acesso são distintas.

A recomendação para minimizar o tempo de resposta consiste em evitar a utilização da memória global, visto apresentar latências de acesso superiores às restantes. Assim, é aconselhado transferir os dados para a memória local no início do processamento, para que estes possam ser manipulados localmente no multiprocessador. Quando o processamento termina, os dados são novamente enviados para a memória global para que possam ser transferidos para o CPU.

No que diz respeito aos acessos à memória global, estes podem ser realizados de forma eficiente caso sejam coalescentes. Este tipo de acessos consiste em aproveitar o bloco de memória transferido para o multiprocessador quando este acede a uma dada célula de memória. É benéfico para o desempenho procurar combinar acessos à memória de diferentes *threads*, transferindo num único acesso à memória dados de vários *threads*.

Deste modo, quando os *threads* do mesmo *warp* necessitam de aceder à memória global, caso estes sejam contíguos, ou que estejam presentes num mesmo bloco de memória, o multiprocessador apenas efectua uma transferência. Caso contrário é necessário transferir vários blocos para que todos os *threads* obtenham as células requeridas.

2.3.3 Fluxo de controlo

Um outro aspecto relevante a referir diz respeito ao fluxo de controlo. Este é bastante importante dado que cada multiprocessador executa cada *warp* em modo SIMD. Assim, caso existam fluxos distintos entre *threads* do mesmo *warp*, estes têm de ser executados em instantes distintos, executando apenas um fluxo de cada vez.

2.3.4 Comunicação entre o CPU e GPGPU

Uma das grandes limitações da programação com GPGPUs consiste nas transferências entre o CPU e GPGPU. Essas transferências são realizadas entre a memória RAM e a memória global do GPGPU através do *bus* PCI-E, como se encontra ilustrado na figura 2.13. Este *bus* é partilhado em diversos dispositivos, como placas de redes, som, entre

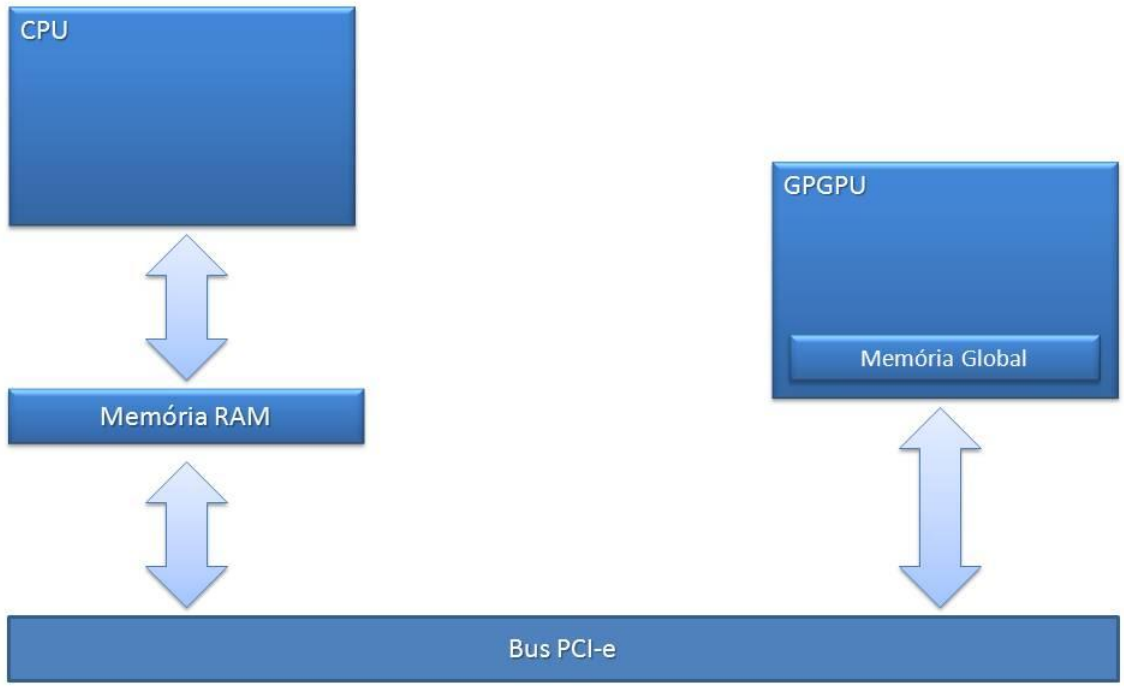


Figura 2.13: Comunicação entre CPU e GPGPU.

outras. Uma limitação dessa utilização é que quando um dispositivo está a utilizar o *bus*, os outros necessitam de aguardar. Sendo as transferências realizadas através do *bus* PCI-E, a taxa de bytes enviados por segundo está limitada à largura de banda do mesmo. Teoricamente este *bus* no modelo x16 na versão 2.0 pode transferir até 8GB/s. Este valor na prática é bastante menor, visto existir concorrência nos acessos bem como *overheads* nas transmissões dos bytes, que reduzem 25% da performance. Um aspecto importante de referir sobre este *bus* diz respeito à possibilidade de poder transmitir e receber bytes em simultâneo.

Para efectuar as transferências entre CPU e GPGPU existem diversas formas, sendo elas a cópia regular, cópia com *page-locked*, ou através de mapeamento.

A cópia regular consiste em definir um conjunto de dados para efectuar a transmissão, este está sujeito aos *overheads* do sistema de gestão de memória do sistema operativo, dado que as páginas relativas a esses dados podem não estar presentes em RAM.

Relativamente à cópia através de *page-lock*, esta consiste em reservar uma região de memória em RAM que não pode ser retirada através das operações de *swap-out*. Deste modo, quando iniciada uma transferência, esses dados já estão em RAM. Este tipo de transferências consegue efectuar taxas na ordem dos 5GB/s.

No que concerne ao mapeamento, este consiste em definir uma região em RAM, também *page-locked*, que irá ser acedida em tempo de execução pelo GPGPU, sempre que este necessite de dados presentes nessa região. Este modo permite efectuar uma extensão à memória do GPGPU.

2.3.5 Conclusões

Em suma, a programação através de GPGPUs torna-se bastante difícil, dado necessitar rescrever integralmente todo um algoritmo sequencial ou paralelo, sendo que este tipo de programação utiliza um paradigma totalmente distinto dos restantes. Além do paradigma de programação, a programação com GPGPUs situa-se a um baixo nível, ainda bastante limitada, dada a insistência de objectos e estruturas de dados dinâmicas. Após a conclusão de um dado algoritmo, é necessário efectuar optimizações que consistem em ajustá-lo ao dispositivo. Este tipo de optimizações muitas vezes implicam rescrever total ou parcialmente o algoritmo, entrando muitas vezes num ciclo, como ilustrado na figura 2.14.

Figura 2.14: Ciclo de desenvolvimento.

Muitas vezes o ajuste dos mecanismos referidos anteriormente entram em conflito, ou seja, a manipulação de um degrada o desempenho do outro. Embora seja difícil a programação com estes dispositivos é bastante importante para a criação de soluções eficientes, proporcionando tempos de resposta bastante baixos, impossíveis de obter através de outros mecanismos.

3

Identificação de objectos em imagens tomográficas

Uma operação bastante importante no estudo de materiais compósitos consiste na identificação dos objectos presentes numa dada amostra. Essa identificação tem como base um agrupamento de voxels interligados segundo um nível de conectividade previamente definido, que quando apresentam um valor distinto dos voxels que os rodeiam dão origem a objectos. Dado que cada voxel é representado por um cubo num espaço tridimensional, a sua conectividade com os restantes pode ser realizada através das faces, das arestas ou dos vértices.

Este problema enquadra-se num conjunto de problemas, que são actualmente resolvidos através da identificação do volume de dados com etiquetas sendo estas distintas entre objectos. Esta classe de problemas é designada de *Connected-component labeling*. Assim, dado um qualquer volume bidimensional ou tridimensional constituído apenas por voxels, todos eles são processados de forma que sejam substituídos por um identificador que representa o objecto de forma unívoca, como ilustrado na figura 3.1. Desta forma é possível obter todos os voxels de um dado objecto, através do seu identificador, para analisar posteriormente de forma isolada.

Esta forma de resolução de problemas tem vindo a ser alvo de investigação dada a sua elevada complexidade, variando esta em função da dimensão do volume de dados, bem como da morfologia dos objectos.

A sua utilização no projecto é bastante pertinente, uma vez que realiza a transição dos dados não estruturados para dados estruturados, o que torna mais fácil a análise dos materiais em estudo.

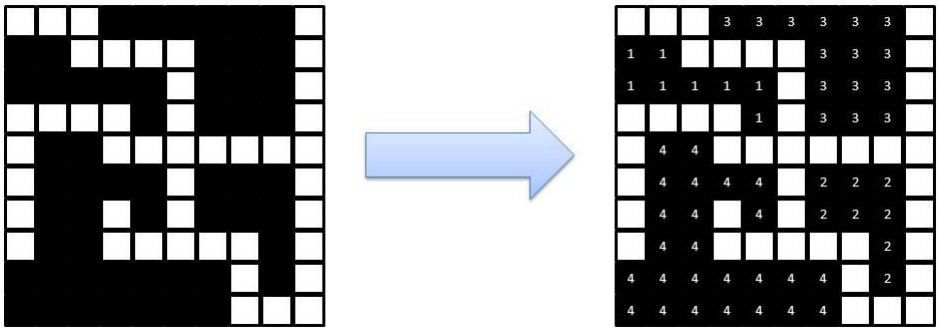


Figura 3.1: Algoritmo *Connected-component labeling*.

Uma outra utilização desta classe de algoritmos é na detecção de padrões [1].

De entre diversas soluções, destacam-se as soluções puramente sequenciais [51], soluções utilizando *clusters* [12], e soluções utilizando GPGPUs [14] [16], sendo nestas últimas que se incorpora a solução desenvolvida na presente dissertação.

De seguida, estão descritas diversas soluções para o problema da identificação de objectos, as quais estão agrupadas em dois grandes grupos, o grupo das soluções sequenciais e o grupo das soluções que utilizam paralelismo.

3.1 Soluções sequenciais

As soluções que processam sequencialmente os volumes de dados, apresentadas no artigo [51], consistem em soluções lineares no tempo de resposta. Estas podem ser agrupadas em três grupos de algoritmos, *multi-pass*, *two-pass* e *one-pass*, segundo o número de passagens que efectuam ao volume de dados.

Os algoritmos *multi-pass* realizam diversas passagens ao volume de dados até se obter a solução. O algoritmo mais conhecido desta classe é descrito em Suzuki et al. [45], e efectua até quatro passagens no volume de dados, recorrendo a uma tabela de conectividade entre identificadores para reduzir o número de passagens.

Relativamente à classe de algoritmos *two-pass*, estes efectuam duas passagens ao volume de dados, acedendo apenas a posições contíguas de memória. Este algoritmo utiliza o endereçamento do volume de dados para atribuir inicialmente identificadores únicos aos voxels. Para efectuar a junção de identificadores, é utilizada a estrutura de dados *Disjoint-Set Forests*.

A estrutura de dados *Disjoint-Set Forests* [9] tem como objectivo a realização de uniões entre conjuntos de forma eficiente, através de dois quaisquer elementos pertencentes a conjuntos distintos. Esta estrutura apresenta a vantagem de permitir manipular conjuntos partindo de um qualquer elemento, sem necessitar de modificar todos os outros. Cada conjunto da estrutura de dados é representado internamente numa árvore, a qual é constituída por todos os elementos do grupo, tendo como raíz o representante do grupo,

tal como ilustrado na figura 3.2. Cada elemento do conjunto é constituído por um identificador que representa o elemento, e por uma ligação a um nó do conjunto.

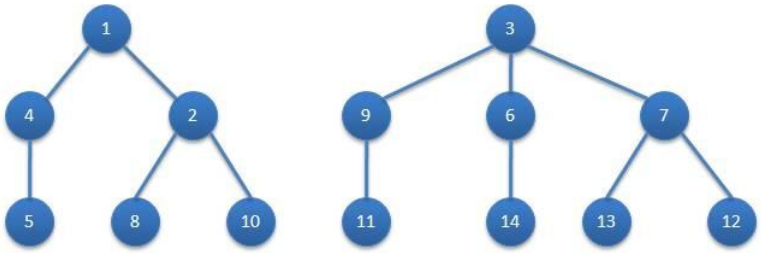


Figura 3.2: Estrutura de dados *Disjoint-Set Forests*.

Para a manipulação dos conjuntos, esta estrutura disponibiliza três operações: criação de um grupo associado a um identificador único; obtenção do representante do grupo através de qualquer elemento do mesmo; e a união de dois grupos através dos seus representantes.

O algoritmo *two–pass* inicia a estrutura de dados com tantos grupos quanto o número de identificadores existentes. Seguidamente, efectua a primeira passagem, que consiste em unir o grupo ao qual pertence o identificador do voxel corrente com o dos seus vizinhos. Assim, no final da primeira passagem, existem tantos grupos quanto objectos e cada identificador pertence a um só grupo.

A segunda passagem consiste em atribuir aos voxels do volume de dados o representante do grupo ao qual eles pertencem. Assim, cada voxel de um mesmo objecto possui o mesmo identificador.

Este algoritmo tem a vantagem de apenas efectuar acessos contíguos em memória no que diz respeito ao volume de dados, visto não efectuar acessos irregulares como o algoritmo anterior.

Por fim, existe a classe de algoritmos *one-pass*, que apenas realiza uma passagem ao volume de dados. Estes consistem em percorrer todo o volume de dados, e sempre que for detectado um voxel pertencendo a um objecto, este é explorado de forma a percorrer todos os voxels com conectividade ao voxel em exploração.

Embora estes algoritmos apenas realizem uma passagem ao volume de dados, realizam também acessos irregulares à memória, o que penaliza bastante o seu desempenho.

Estas soluções têm a grande desvantagem de proporcionarem tempos de resposta bastante elevados, e de apenas poderem ser aplicadas a conjuntos de dados limitados. Estas desvantagens tornam estes algoritmos inapropriados para o contexto do projecto, dado que não permitem tempos de resposta adequados a um ambiente interactivo, tal como é demonstrado posteriormente na análise das soluções. Contudo, estes apresentam a grande vantagem de não necessitarem de *hardware* específico, apresentando portabilidade e um custo menor.

3.2 Soluções paralelas

Dada a elevada complexidade do problema, uma forma de reduzir o tempo de resposta é utilizar paralelismo. Deste modo, é possível decompor o processamento de forma a distribui-lo pelas unidades de processamento disponíveis.

Tal como já foi referido, de entre várias soluções, destacam-se as soluções que utilizam *clusters* e GPGPUs.

No que se refere às soluções utilizando *clusters*, destaca-se a solução *Data-Parallel Mesh Connected Components Labeling and Analysis* [12], que consiste em dividir o volume de dados por diversos nós, que processam um subconjunto do volume de dados original. Durante esse processamento existem pontos de sincronização de forma a obter informações e a conectividade entre os subconjuntos processados em nós distintos. Para identificar os objectos, esta solução utiliza uma estrutura de dados designada *Disjoint-Set Forests*, que permite detectar conjuntos disjuntos através de uniões. Como é possível observar na figura 3.3, o algoritmo é decomposto em quatro fases.

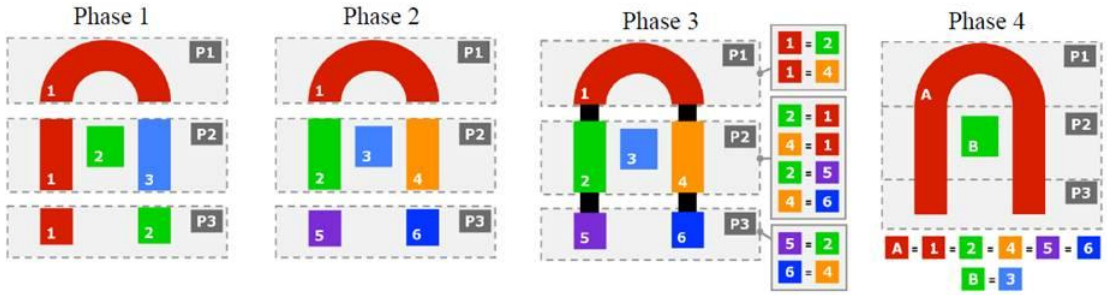


Figura 3.3: Fases do algoritmo *Data-Parallel Mesh Connected Components Labeling and Analysis* [12].

A primeira fase consiste em identificar os objectos presentes no subconjunto de dados, atribuindo-lhes uma etiqueta única para cada objecto, utilizando a estrutura de dados *Disjoint-Set Forests*. Nesta primeira fase, os identificadores são únicos apenas dentro do mesmo subconjunto do volume de dados a ser processados.

Seguidamente, é realizada uma outra fase que globaliza os identificadores, de forma que objectos diferentes possuam identificadores distintos. Essa fase é realizada através da propagação de mensagens entre nós.

Por fim, é necessário unir objectos que se encontrem divididos em um ou mais subconjuntos. Essa união consiste em propagar por *broadcast* as estruturas locais a cada nó, para que todos consigam validar os seus identificadores de forma a encontrar relações com outros objectos.

Esta solução tem a vantagem de suportar grandes volumes de dados, sendo estes posteriormente decompostos e atribuídos aos nós disponíveis. O problema desta abordagem reside na relação custo/escalabilidade, dada a necessidade de adicionar novos nós ao *cluster* para aumentar a capacidade computacional. Embora a escalabilidade seja

um problema, a principal limitação da solução reside essencialmente nas latências da comunicação entre os nós, que tornam a interactividade limitada.

Relativamente às soluções que utilizaram GPGPUs, destacam-se as seguintes: *Parallel Graph Component Labelling with GPUs and CUDA* [14] e *Connected Component Labeling* (CCL) [16].

No que se refere ao artigo *Parallel Graph Component Labelling with GPUs and CUDA* [14], este analisa um conjunto de soluções que permitem a identificação de objectos numa dada amostra. As soluções apresentadas consistem essencialmente em implementações em GPGPU de soluções sequenciais existentes para CPU, sendo descritas as suas vantagens e limitações.

No que diz respeito à solução CCL, esta consiste em efectuar todo o processamento no GPGPU, tirando partido da memória e da localização da informação, ocorrendo em quatro fases, tal como se encontra ilustrado na figura 3.4.

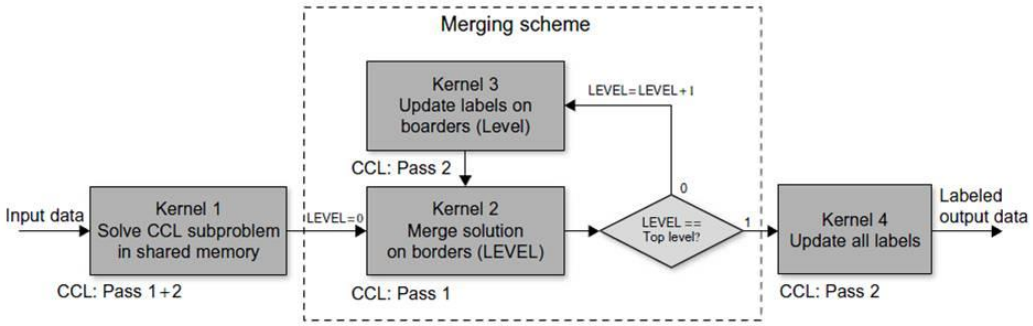


Figura 3.4: Fases do algoritmo *Parallel Graph Component Labelling with GPUs and CUDA* [16].

Na primeira fase ocorre a identificação unívoca de subobjectos dentro do mesmo multiprocessador, tirando partido assim dos tempos de acesso à memória local. Durante esta fase primeiramente copia-se um subconjunto do volume de dados para a memória local, e de seguida cada voxel correspondente a um objecto é etiquetado com o valor de indexação do mesmo na memória. Posteriormente, cada voxel valida o valor dos vizinhos, e adopta-o caso um deles possua um identificador inferior. Esta operação é repetida até que nenhum voxel seja alterado. De forma a efectuar uma convergência mais rápida, no final de cada iteração, cada voxel, após encontrar um identificador menor, valida se este já está associado a um inferior, detectando assim a transição entre identificadores.

Seguidamente, o algoritmo necessita de fundir os subobjectos, para isso é realizado um novo processamento, no qual se validam as fronteiras entre cada bloco anteriormente processado e se detectam as respectivas relações. De forma a melhorar o desempenho, o referido processamento é efectuado em diversas iterações de forma hierárquica, garantindo um número fixo de iterações. Assim, em cada iteração apenas são unidos quatro blocos, dando origem a um bloco maior para a fase seguinte. Essa operação é repetida até que o bloco maior corresponda ao volume de dados. Um aspecto bastante importante a

referir é que este algoritmo necessita de uma estrutura auxiliar para relacionar identificadores antigos com os novos, visto não ser possível efectuar uma previsão para o número total de identificadores presentes na amostra.

Este algoritmo apresenta enormes vantagens face aos descritos anteriormente em termos da diminuição dos tempos de execução. No entanto, apesar de todas as suas vantagens, o algoritmo apenas é aplicável a imagens bidimensionais com volumes relativamente pequenos, comparados com os utilizados pelo algoritmo desenvolvido.

3.3 Conclusões

A solução *Data-Parallel Mesh Connected Components Labeling and Analysis* permite a identificação de objectos tridimensionais de grandes dimensões, mas para isso recorre a um *cluster* para obter um elevado poder computacional agregado. Esse aspecto não se adequa às características do projecto, elevando assim os seus custos. Além de possuir um elevado custo, possui também uma limitação associada às latências na comunicação entre os nós, o que torna o tempo de resposta elevado para um ambiente interactivo.

Relativamente à solução *Connected Component Labeling*, esta apenas efectua o processamento de imagens bidimensionas, não permitindo assim a detecção dos objectos tridimensionais em volumes de dados resultantes da micro tomografia computacional.

Embora a solução tenha essa limitação, esta possui um elevado rácio entre o custo e a capacidade computacional. Posteriormente, apresenta-se a solução desenvolvida, que combina as duas soluções já existentes de forma a realizar o processamento de imagens tridimensionais através de GPGPUs com tempos de resposta adequados a um ambiente interactivo.

4

Concepção dos algoritmos

O algoritmo desenvolvido baseou-se num algoritmo sequencial designado *flood fill*, que pertence à classe de algoritmos *one-pass*, sendo utilizado em computação gráfica para o preenchimento de objectos. Esse algoritmo consiste em percorrer cada pixel de uma determinada imagem e, caso encontre um pixel pertencente a um objecto, inicia a fase de preenchimento. A fase de preenchimento consiste em pintar o pixel corrente e propagar esse processamento para os seus vizinhos, até encontrar as fronteiras do objecto em causa.

O algoritmo de *flood fill*, quando processado de forma sequencial, apresenta uma elevada complexidade computacional limitando a dimensão do conjunto de dados, sendo assim ineficiente para a identificação de objectos em grandes volumes de dados, como é possível comprovar na análise realizada no capítulo seguinte. Para que o processamento de grandes volumes de dados seja possível é necessário paralelizar o algoritmo, de forma a dividir os grandes volumes de dados pelas diversas unidades de processamento, tanto ao nível do CPU como do GPGPU.

O algoritmo desenvolvido utiliza paralelismo ao nível do CPU e do GPU, de forma a ser possível melhorar o desempenho quando há uma elevada dimensão do volume de dados a processar. A necessidade do uso de paralelismo ao nível do CPU deve-se à impossibilidade de colocar todo o volume de dados na memória de um GPU. Como alternativa, poder-se-ia mapear os dados de forma que estes permanecessem em memória RAM até que fossem necessários na execução do GPU. Dada a elevada latência nas transferências entre CPU e GPU, esta alternativa apresenta um custo bastante elevado, uma vez que os mesmos dados necessitam ser transferidos diversas vezes. Desta forma, o algoritmo desenvolvido, descrito na figura 4.1, divide o volume de dados em subconjuntos, que são posteriormente processados em paralelo por um ou mais GPUs. Após essa operação é realizado um processamento adicional, no qual se obtém a conectividade

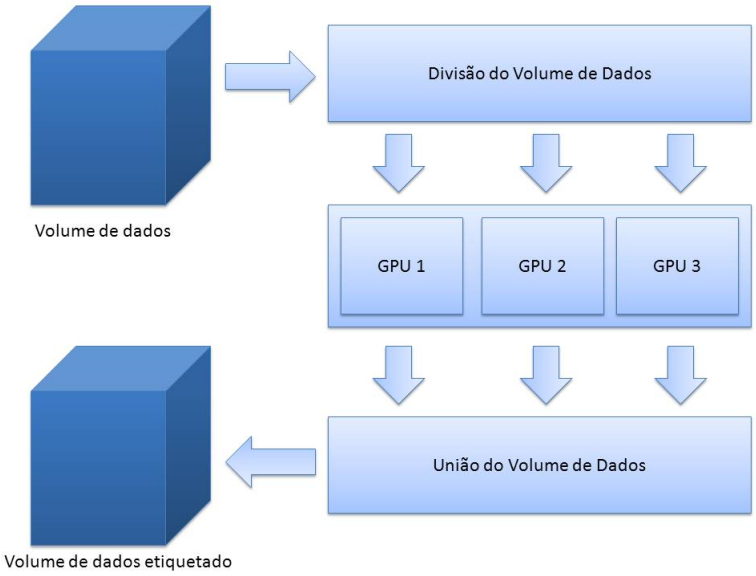


Figura 4.1: Fases do algoritmo desenvolvido.

entre objectos presentes em subconjuntos distintos.

Para a obtenção de uma solução com melhor eficiência tornou-se necessário analisar pormenorizadamente cada fase do algoritmo, através de diversas técnicas, de forma a localizar os seus pontos críticos. A solução final resultou de diversas melhorias realizadas à solução, após serem identificadas as suas limitações.

De seguida, encontram-se descritas as diversas fases pelas quais foi necessário passar ao longo do desenvolvimento do algoritmo.

4.1 Decomposição do volume de dados em blocos

Como já foi referido anteriormente, para que seja possível processar um elevado volume de dados, dada a limitação da memória dos GPGPUs, o algoritmo começa por dividir o conjunto de dados em blocos, que serão posteriormente processados isoladamente. Uma desvantagem desta divisão diz respeito à necessidade de computação adicional para calcular as ligações entre os subobjectos de blocos distintos, como será descrito seguidamente. De forma a reduzir o impacto no desempenho, a divisão é apenas efectuada ao nível da dimensão em z, como se encontra ilustrado na figura 4.2, fazendo assim com que os dados de cada bloco estejam contíguos em memória.

Na primeira abordagem realizou-se a divisão do volume de dados em blocos de dimensão fixa, colocando-os numa fila de trabalho. Assim, cada GPU quando se encontra disponível retira um bloco assincronamente da fila de blocos.

O cálculo da dimensão de cada bloco é efectuado no início do algoritmo, baseando-se na análise do montante de memória disponível em todos os dispositivos disponíveis. Após se obterem esses valores, é seleccionado o mínimo para todos os blocos passando estes a serem processados em qualquer dispositivo.

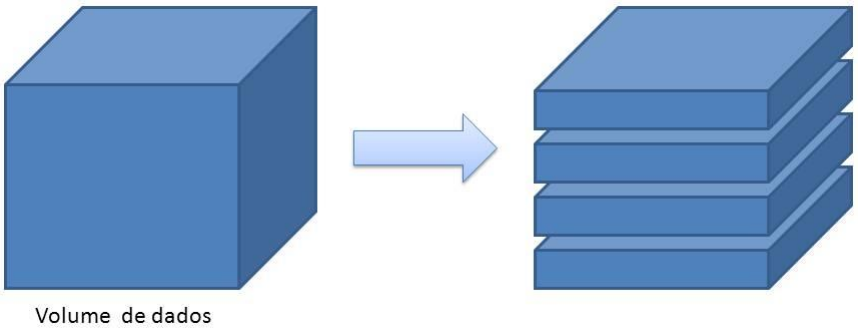


Figura 4.2: Divisão do volume de dados.

Esta solução tem a vantagem de permitir a distribuição dos dados de forma equitativa pelos dispositivos, visto que são estes que controlam a distribuição dos blocos. A grande desvantagem desta solução corresponde ao facto da dimensão dos blocos ser fixa, o que limita a solução em ambientes heterogéneos ao dispositivo com menor capacidade de armazenamento.

De forma a ultrapassar a desvantagem supracitada, ou seja, para se maximizar a solução em ambientes heterogéneos, foi modificada a distribuição dos blocos, de forma que estes tenham dimensões distintas consoante a memória do dispositivo que os processa. Deste modo, ao invés de inicialmente existir uma fila com blocos, existe apenas um apontador para o volume de dados, que é incrementado em exclusão mútua sempre que é realizado um pedido. Assim, cada dispositivo quando requer um bloco envia a dimensão máxima. Esta abordagem permite tirar o melhor partido dos recursos, bem como reduzir a complexidade associada à fila de trabalho.

4.2 Processamento

Esta fase tem como finalidade identificar todos os objectos presentes em cada bloco de dados, de modo a que cada um seja representado com um identificador único.

Cada bloco é processado em paralelo pelos GPGPUs, tirando assim partido de todos os dispositivos existentes. Estes blocos apresentam uma característica importante de referir, que é o facto de estes poderem ser inseridos integralmente na memória do GPU, fazendo assim com que não seja necessário transferir mais do que uma vez os dados, visto serem as transferências a principal limitação dos GPUs. Desta forma, os dados são transferidos no início para um GPU, e apenas são retirados no fim do processamento de todos os *kernels*.

Nesta etapa, os dados de entrada são representados por voxels compreendidos entre 0 e 255, ou seja, numa escala de cinzentos, o que implica que cada voxel corresponda a um byte. Uma vez que cada voxel é convertido num identificador, é necessário um inteiro de quatro bytes para o representar.

Tal como está ilustrado na figura 4.3, o processamento no GPU é constituído por três

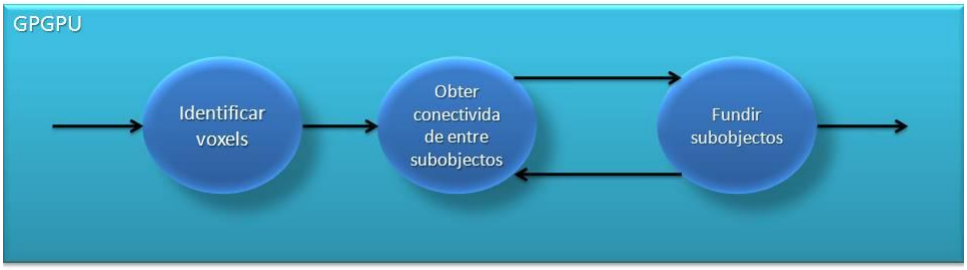


Figura 4.3: Processamento no GPGPU.

kernels, os quais são executados sequencialmente para cada subconjunto dos dados. O primeiro *kernel* tem como principal função atribuir identificadores aos voxels do subconjunto, de forma a que estes possam posteriormente identificar univocamente um objecto. O segundo *kernel* tem a função de obter conectividade entre identificadores, ou seja, detectar identificadores que se referem ao mesmo objecto. Por fim, o terceiro *kernel* tem como função alterar, no volume de dados, os identificadores detectados pelo *kernel* anterior. De forma a estruturar a execução em GPUs, tornam-se necessárias duas etapas, sendo a primeira constituída apenas pelo primeiro *kernel*, para a identificação dos elementos do bloco de dados, e a segunda para a junção dos identificadores, de modo a existir apenas um identificador por objecto. De seguida, encontra-se descrita a implementação das duas fases em GPU, bem como as variantes associadas.

4.2.1 Identificação de subobjectos

Uma primeira tentativa de realizar paralelismo consistiu em dividir geometricamente os dados do bloco em processamento, de forma que cada elemento resultante seja processado por um *thread*, dando origem a um conjunto de subobjectos, tal como se pode verificar na figura 4.4.

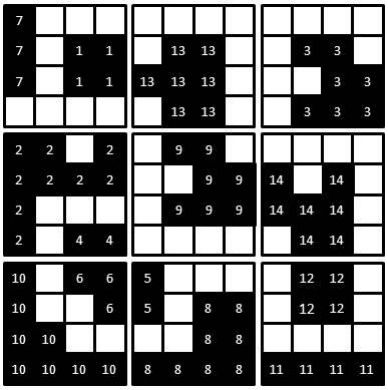


Figura 4.4: Identificação dos objectos através de divisão geométrica.

Tal como na versão iterativa, o algoritmo necessita de uma estrutura auxiliar para guardar os voxels a processar. Dado que não existem mecanismos de alocação de memória dinâmica no GPU, foi necessário alocar previamente memória para ser distribuída

posteriormente aos *threads*, de forma a guardar os voxels a expandir, através de um gestor de memória.

Nesta primeira abordagem, a utilização dos GPGPUs apresenta a vantagem de permitir uma maior decomposição do volume de dados, comparativamente à proporcionada pelos CPUs. Apesar da vantagem mencionada, foram identificadas algumas limitações na solução. Uma dessas limitações consiste na utilização da memória adicional para guardar os voxels a expandir. Esta, sendo alocada previamente em memória global, necessita de ter dimensão suficiente para alocar os voxels a expandir em qualquer fase do algoritmo. Devido à quantidade limitada de memória nos GPGPUs o problema agrava-se, penalizando bastante o desempenho do algoritmo. Além da quantidade de memória reduzida, a utilização de um gestor de memória teve como resultado um aumento da complexidade computacional da solução bastante elevado. Uma outra desvantagem da utilização desta memória adicional, reside no aumento de acessos à memória global durante o processamento.

Para melhorar esta abordagem, a dimensão de cada conjunto de voxels a processar por cada *thread* poderia ser igual à dimensão máxima da memória local a cada multiprocessador. Desta forma, ao invés de se utilizar a memória global seria utilizada a memória local, melhorando bastante o desempenho do algoritmo. Todavia, esta melhoria influencia negativamente o escalonamento de *threads* do GPU. Dado que cada *thread* necessita de toda a memória de um multiprocessador, apenas é possível colocar em execução um *thread* de cada vez, devido a este utilizar todos os recursos do mesmo, o que implica que o nível de paralelismo seja limitado ao número de multiprocessadores no GPU.

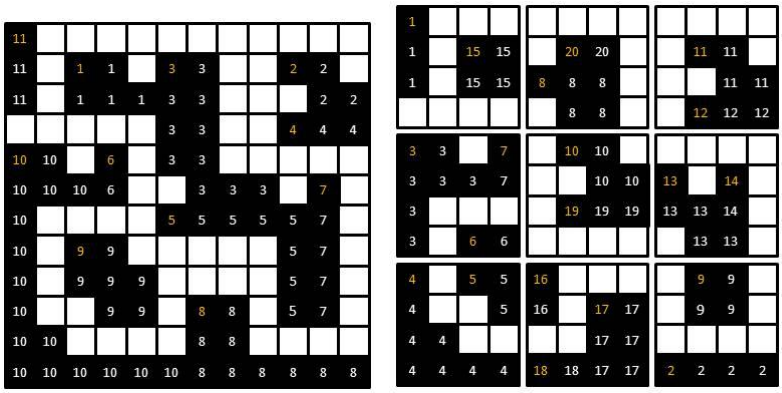
De forma a melhorar a solução, foi necessário também reduzir o volume de dados atribuídos a cada *thread*. Deste modo, tendo em conta a enorme capacidade de paralelismo presente nos GPGPUs, o volume de dados foi reduzido a um voxel por *thread*, eliminando assim a necessidade de memória para expandir a propagação. Assim, foram desenvolvidas duas soluções distintas, as quais se distinguem pela sua complexidade. Ambas consistem em numerar cada voxel, conforme os identificadores dos vizinhos.

4.2.1.1 Primeira solução

Esta solução, ilustrada na figura 4.5a, numera os voxels com base na propagação de identificadores entre vizinhos. Assim, não é necessário utilizar estruturas de dados para numerar os voxels.

Para que essa numeração seja possível, cada *thread* entra em espera activa até que um voxel vizinho seja numerado. Para que a propagação ocorra, existem voxels que são numerados inicialmente, caso contrário os restantes *threads* entravam em *deadlock*. O factor utilizado pela solução, consiste em numerar voxels que tenham os vizinhos da direita, esquerda e de trás com identificadores a branco. Todos os identificadores atribuídos são únicos, distribuídos de forma atómica através de primitivas OpenCL.

Estas melhorias têm como vantagens permitir escalabilidade do algoritmo, bem como



(a) Utilizando memória global. (b) Utilizando memória local.

Figura 4.5: Numeração através de propagação de identificadores.

a redução do tempo de resposta.

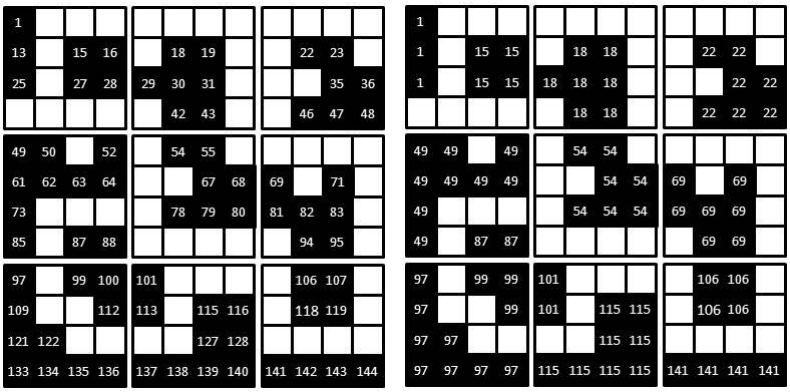
De forma a reduzir o tempo de acesso à memória, cada grupo transfere inicialmente os dados que irá aceder da memória global para a local, e uma vez que os acessos à memória local são bastante inferiores do que à memória global, diminui bastante o tempo de execução. A desvantagem desta abordagem reside no facto dos identificadores apenas serem propagados dentro da mesma memória local, como ilustrado na figura 4.5b, criando assim mais subobjectos.

Uma outra vantagem desta alteração relaciona-se com a existência de *cache* para a memória global dos novos GPGPUs, que visa reduzir o tempo de acesso à memória. Esta opção arquitectural dificulta a comunicação por memória global entre *threads*, isto é, embora a memória global seja partilhada, a verdade é que a consistência com a *cache* não é garantida e assim as alterações realizadas por um *thread* não são visíveis aos *threads* noutro multiprocessador. Posto isto, existem *threads* que entram em espera activa sem nunca obterem um identificador, o que provoca um *deadlock*. Através desta nova abordagem esse factor não ocorre.

4.2.1.2 Segunda solução

Esta solução assemelha-se bastante à solução anterior, no que diz respeito à forma de propagação de identificadores, visto que cada *thread* também valida os identificadores dos vizinhos, que pertençam ao mesmo grupo. A grande diferença consiste em todos os voxels serem numerados inicialmente, utilizando o seu índice na memória global, como está ilustrado na figura 4.6a.

Ao contrário da solução anterior, a propagação apenas termina quando nenhum *thread* do mesmo grupo efectua qualquer alteração ao estado do seu voxel. O processamento consiste em validar os identificadores dos vizinhos, seleccionando o menor, de forma a que todos os voxels de um dado objecto presente no grupo tenham o mesmo identificador, como se pode observar na figura 4.6b. Desta forma, a fusão de subobjectos



(a) Numeração inicial (b) Numeração após propagação

Figura 4.6: Numeração com base no índice da matriz

é efectuada totalmente dentro de cada grupo, fazendo assim com que na fase seguinte apenas seja necessário efectuar as junções entre grupos, para contemplar objectos que estejam presentes em mais do que um grupo em simultâneo.

Como é possível analisar pela descrição do algoritmo, este possui uma complexidade computacional superior à solução anterior. Uma outra desvantagem desta abordagem reside nos identificadores finais dos subobjectos, devido a estes serem calculados inicialmente através do índice, no final não serão contíguos, o que causa um grande impacto no desempenho da fase seguinte.

4.3 Fusão de subobjectos

Nesta fase os identificadores atribuídos na fase anterior são unidos, de forma a que cada objecto seja representado por um identificador único.

A primeira abordagem a esta solução consistiu em criar um grafo em que os vértices correspondem a identificadores, e as arestas às relações entre eles. Após a criação do grafo, é efectuado um percurso em profundidade no mesmo, de forma a encontrar os identificadores que constituem os objectos. Por fim, os identificadores presentes no bloco de dados são alterados consoante o resultado da pesquisa.

Cada *thread* nesta abordagem é mapeado num voxel que verifica os seus vizinhos e, caso exista algum com um identificador distinto, é criada uma aresta no grafo entre os dois vértices correspondentes.

Esta abordagem adequa-se à programação com GPGPUs, visto efectuar uma elevada decomposição do processamento por um vasto número de *threads*. Embora a decomposição dos dados seja adequada, a utilização do grafo, como já foi referido anteriormente, possui diversos aspectos negativos, devido à quantidade de memória utilizada na representação por matriz de adjacências, tal como se pode observar na figura 4.7.

De forma a melhorar a solução, foi desenvolvida uma alternativa que teve como base a junção de subobjectos, dentro do GPU, de modo a resultar no objecto final. Esta fase do

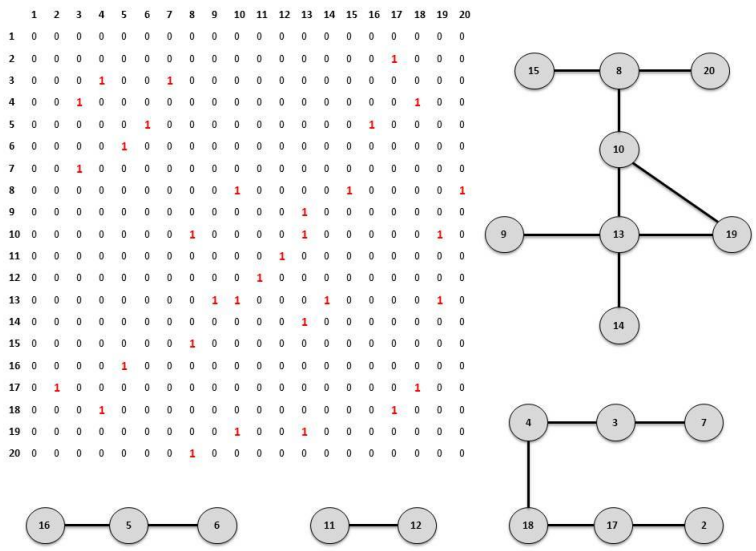


Figura 4.7: Matriz de adjacências

algoritmo, ilustrada na figura 4.8, é composta por três etapas. A primeira etapa consiste em mapear cada voxel num *thread*, validando de seguida todos os seus vizinhos, caso algum possua um identificador inferior, este associa o seu identificador antigo ao identificador novo encontrado numa estrutura auxiliar, para que assim todos os voxels com o valor antigo sejam alterados posteriormente.

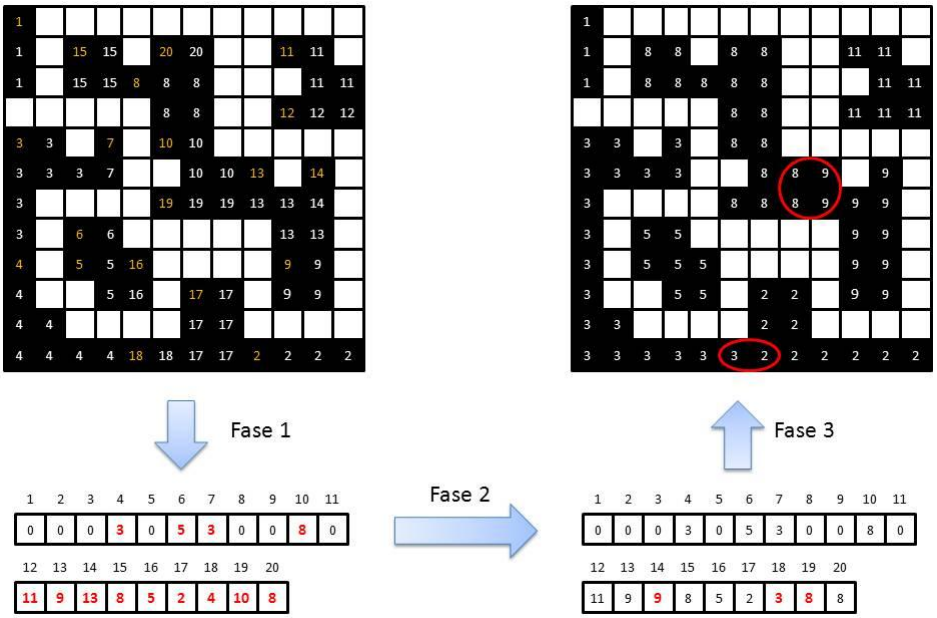
Figura 4.8: Fases do processamento em GPGPU.

A segunda etapa diz respeito à optimização do vector de alterações, o qual analisa a transitividade entre identificadores, de forma a encontrar o identificador representante do subobjecto resultante da fusão.

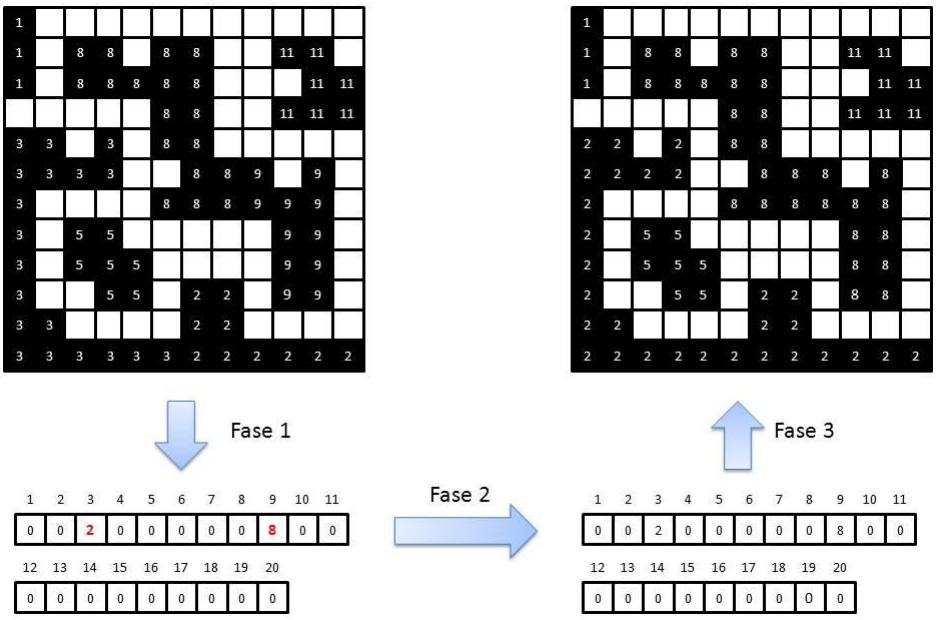
Na terceira etapa todos os voxels são novamente mapeados num *thread*, utilizando o GPGPU, o qual altera o identificador corrente de cada voxel para o correspondente no vector de alterações.

As fases supracitadas são repetidas até que todos os subobjectos do bloco sejam unidos, ou seja, até que não exista nenhuma modificação a efectuar no vector das alterações na primeira fase, tal como consta na figura 4.9.

Ao contrário da construção do grafo, sendo N o número de subobjectos identificados anteriormente, esta solução apenas necessita de um vector de N elementos, ao invés de N×N requerido pela matriz de adjacências, reduzindo assim a complexidade espacial do



(a) Primeira iteração



(b) Segunda iteração

Figura 4.9: Exemplo de fusão de objectos em GPGPU.

algoritmo. Uma desvantagem do vector utilizado é que este necessita de quatro bytes para cada elemento, uma vez que cada um deles é do tipo inteiro, ao contrário da matriz que ocupa apenas um byte por elemento, sendo uma aresta representada por um byte.

4.4 Fusão de identificadores de blocos distintos

Uma vez que o volume de dados foi inicialmente decomposto em blocos, é necessário obter a conectividade entre os objectos presentes em blocos distintos. Para realizar esta operação é utilizado um grafo, onde cada objecto representa um vértice, e a conectividade uma aresta. Devido à elevada dimensão do volume, não é possível realizar esta operação de forma eficiente no GPU.

Uma possibilidade seria apenas enviar as fronteiras de cada bloco, contudo, apresenta desvantagens no que concerne à representação do grafo, que devido à ausência de estruturas dinâmicas em GPUs, é realizada através de uma matriz de adjacências, que se torna ineficiente na ocorrência de diversos objectos.

Uma vez que esta operação foi realizada em CPU, e este não possui a limitação do GPU no que concerne à utilização de estruturas dinâmicas, foi utilizada uma estrutura dinâmica para representar o grafo, que consistiu numa tabela de dispersão, na qual a chave representa um nó e o valor a lista de vértices adjacentes. Esta abordagem deu origem a uma redução na complexidade espacial, bem como a uma melhoria na complexidade computacional da pesquisa em profundidade, necessária posteriormente para obter os identificadores relativos a cada objecto.

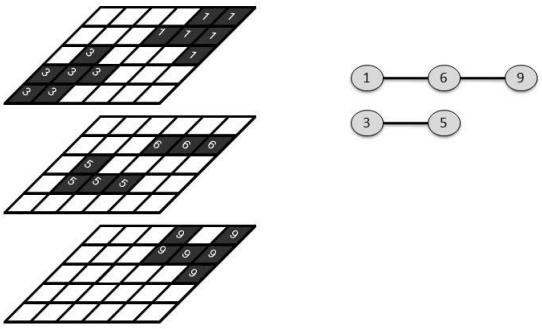


Figura 4.10: Fusão de identificadores de blocos distintos

Numa primeira abordagem desenvolveu-se uma solução sequencial, na qual são percorridos todos os voxels presentes nos planos relativos às fronteiras dos blocos, como se pode observar na figura 4.10. Para cada voxel é validado o seu adjacente no bloco vizinho, e caso sejam distintos, é adicionada uma aresta entre os dois vértices correspondentes. O grande problema desta solução ocorre quando existe um número elevado de blocos, visto que o processamento dos mesmos é realizado sequencialmente.

Um possível melhoramento será utilizar diversos *threads* para processar as fronteiras, no entanto, a desvantagem dessa abordagem reside no elevado número de pontos de sincronização existentes na inserção de arestas, uma vez que dois *threads* distintos podem aceder ao mesmo vértice.

Para melhorar a solução, reduzindo o sincronismo, foram criados diversos *threads* que

assincronamente obtêm blocos e processam as suas fronteiras. Assim, dado que os identificadores são únicos globalmente em cada bloco, não existem acessos concorrentes ao mesmo vértice do grafo, visto serem apenas criadas arestas com origem em identificadores do bloco a processar.

Para finalizar, para se obterem os identificadores que pertencem a um mesmo objecto, é realizada uma pesquisa em profundidade no grafo, que permite obter, de forma eficiente, listas de identificadores que relacionam os subobjectos do mesmo objecto.

5

Implementação dos algoritmos

Neste capítulo encontram-se descritas as implementações realizadas das duas classes de algoritmos sequenciais mais significativas, *one-pass* e *two-pass*. Seguidamente, é apresentada também a solução *Object Identifier*, que utiliza GPGPUs para a resolução do problema. As duas versões sequenciais tiveram como objectivo analisar o desempenho do processamento sequencial em CPU, bem como servir de termo de comparação para a solução *Object Identifier*. Para a classe *one-pass*, foi efectuada a implementação do algoritmo *Flood Fill* iterativo. Para a classe *two-pass*, foi realizada uma implementação que utiliza a estrutura de dados *Disjoint Set Forests*. O algoritmo *Object Identifier* é apresentado em três versões, as quais variam na fase de processamento da imagem, mantendo-se semelhantes nas restantes fases. Em seguida, encontram-se descritos os aspectos mais importantes de todas as implementações realizadas. Primeiramente, são apresentados os dados de entrada e saída adoptados pelas implementações, seguindo-se os detalhes das mesmas.

5.1 Input / Output

Os algoritmos implementados têm como parâmetros de entrada o volume de dados e as suas dimensões em x, y e em z. O volume de dados de entrada é representado por voxels numa escala de cinzentos, ou seja, com valor compreendido entre 0 e 255, sendo representado por um vector de elementos do tipo *unsigned char*, isto é, 8-bits por voxel. Dado que o volume de saída irá conter tantos identificadores quantos objectos encontrados, é necessário utilizar elementos com quantidade de memória suficiente para representar todos os identificadores possíveis, para isso foram utilizados *unsigned int*, que suportam identificadores de 32 bits que variam entre 0 e $2^{32} - 1$.

O algoritmo também possui a possibilidade de fornecer a lista de voxels relativa a

cada objecto encontrado. Esse *output* consiste numa *hashtable* em que a chave corresponde ao identificador do objecto e o valor a um vector de voxels. Para reduzir a quantidade de memória, é utilizado o índice no volume de dados para representar um dado voxel; assim apenas é necessário um inteiro ao invés de três.

5.2 Algoritmo One-Pass

Este algoritmo foi implementado de forma iterativa utilizando uma fila para guardar os voxels a explorar. Uma possível alternativa seria não utilizar a fila, e utilizar recursividade, porém essa alternativa iria limitar bastante a solução.

Como descrito no algoritmo 1, este realiza um percurso sequencial de todos os voxels, e sempre que encontra um voxel ainda não identificado, este é colocado na fila, e é iniciada a fase de exploração.

Na fase de exploração, são retirados os voxels da fila, caso estes não estejam identificados, são identificados, e todos os seus vizinhos são colocados na fila para que sejam tratados posteriormente, visto poderem pertencer ao objecto em questão.

A fase de exploração termina quando a fila estiver vazia, ou seja, quando já não existirem voxels por identificar para o objecto.

Algoritmo 1 Flood Fill Iterativo

```
counter ← 1
queue ← empty queue
for z = 0 → z_s do
  for y = 0 → y_s do
    for x = 0 → x_s do
      if image[x][y][z] = UNFILL then
        counter ← counter + 1
        elem ←Point(x, y, z)
        queue.push(elem)
        while queue.size() ⩾ 0 do
          v ← queue.pop()
          if image[v.x][v.y][v.z] = UNFILL then
            image[v.x][v.y][v.z] ← counter
            if x ⩾ 0 then
              neighbor ←Point(v.x − 1, v.y, v.z)
              queue.push(neighbor)
            end if
            (...) // Expansão para x + 1, y − 1, y + 1, z − 1 e z + 1.
          end if
        end while
      end if
    end for
  end for
end for
```

5.3 Algoritmo Two-Pass

Relativamente ao algoritmo *two-pass*, este consiste em efectuar duas passagens ao volume de dados, acedendo apenas a posições contíguas de memória no que concerne ao volume referido. Este algoritmo utiliza o endereçamento do volume de dados para atribuir inicialmente identificadores únicos aos voxels. Para efectuar a junção de identificadores, é utilizada a estrutura de dados *Disjoint Set Forests*.

Tal como é possível observar no algoritmo 2, este algoritmo começa por criar todos os grupos, ou seja, são criados tantos grupos quanto o número de voxels do volume.

Na primeira iteração, sempre que é encontrado um voxel com um identificador diferente de branco, este é unido com os grupos dos voxels vizinhos, de forma a unificar esses voxels num único objecto.

Após terminar a primeira iteração, todos os grupos estão associados de forma que exista um representante para cada objecto. Em seguida, o volume é novamente percorrido, e para cada voxel é pedido o representante do objecto em causa, ficando o identificador deste associado a todos os voxels do objecto.

Algoritmo 2 Two-Pass

```
nodes = [ ]
for i = 0 → size do
  if image[x][y][z] ≠ WHITE then
    nodes[i] = MakeSet(i)
    image[x][y][z] = i
  end if
end for
for z = 0 → z_s do
  for y = 0 → y_s do
    for x = 0 → x_s do
      if image[x][y][z] ≠ WHITE then
        currentIndex ← image[x][y][z]
        if x ⩾ 0 then
          neighborIndex ← image[x − 1][y][z]
          Union(Find(nodes[currentIndex]), Find(nodes[neighborIndex])
        end if
        (...) // Expansão para x + 1, y − 1, y + 1, z − 1 e z + 1.
      end if
    end for
  end for
end for
for i = 0 → size do
  image[i] =Find(nodes[i])
end for
```

5.4 Object Identifier

O algoritmo *Object Identifier* foi implementado utilizando a plataforma OpenCL, de forma a utilizar GPGPUs para efectuar o processamento dos dados. Além do paralelismo proporcionado pelos GPGPUs, é utilizado também paralelismo ao nível do CPU, de forma a maximizar a performance do algoritmo.

O algoritmo começa por pesquisar os dispositivos disponíveis no sistema, criando um *thread* para cada dispositivo, o qual será responsável por toda a comunicação entre o CPU e o dispositivo.

Cada *thread* começa por criar uma *command queue* para um dispositivo, para que possam ser enviados comandos. Seguidamente, é analisada a quantidade de memória disponível para alocação no dispositivo correspondente, de forma a calcular a dimensão de cada bloco de dados, podendo assim tirar o maior partido do dispositivo.

Após efectuar a extracção de informações sobre o dispositivo, é iniciada a fase de processamento, onde cada *thread* entra numa fase cíclica, em que em cada iteração do ciclo é processado um bloco de dados, como se encontra ilustrado na figura 5.1.

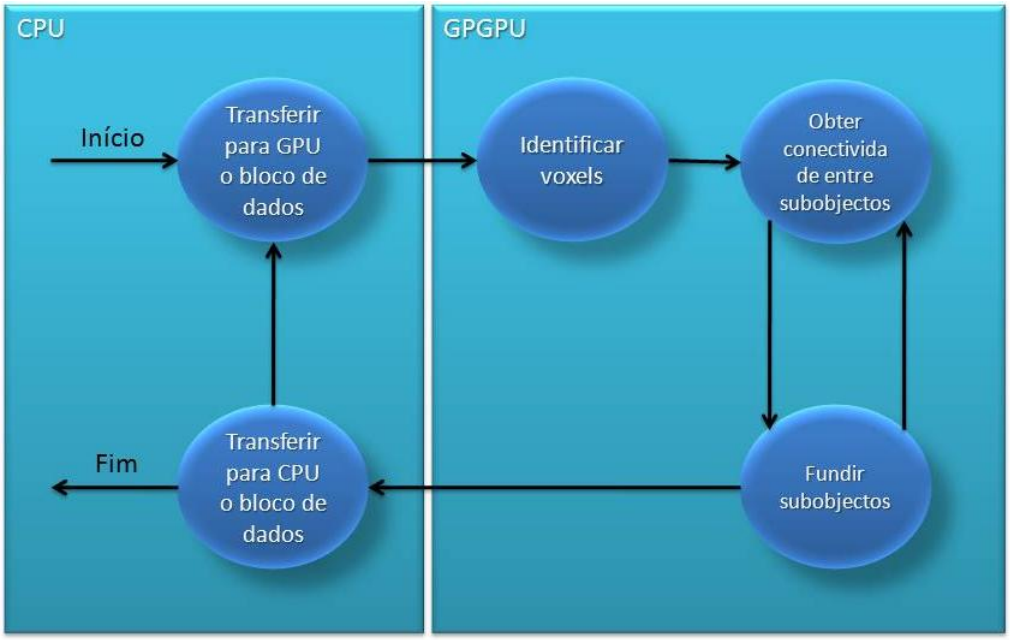


Figura 5.1: Processamento dos blocos.

Relativamente à representação dos blocos de dados, esta ocorre através de dois identificadores, um que indica o início do bloco no volume de dados, e outro que informa acerca da dimensão do mesmo. Através desta representação cada bloco de dados está contíguo em memória, melhorando assim os tempos de leitura do mesmo.

A forma adoptada na implementação da fila de trabalho consiste num contador atómico, que é implementado através de um apontador para o volume de dados que já foi atribuído. Deste modo, sempre que um *thread* requeira um bloco de dados, o contador é

incrementado atomicamente consoante o valor máximo suportado pelo dispositivo. Assim, é possível implementar uma fila assíncrona com distribuição equitativa de trabalho.

Após obter o bloco a processar, cada *thread* transfere-o juntamente com a dimensão do mesmo para o dispositivo, para que possa ser processado.

Tanto o bloco de dados como as dimensões ficam presentes no GPGPU até terminar a execução de todos os *kernels*. Desta forma, é possível reduzir o número de transferências entre o CPU e o GPGPU.

Após o processamento, cada *thread* transfere para o CPU o bloco de dados já processado. Posteriormente, ocorre um ponto de sincronização, que tem como objectivo aguardar que todos os *threads* terminem, ou seja, que todos os blocos de dados sejam processados e transferidos para o CPU.

De forma a relacionar os identificadores entre blocos distintos, dado que estes apenas estão divididos através da dimensão em z, são criados diversos *threads* que percorrem todo o plano relativo à fronteira entre os blocos, de modo a relacionar objectos presentes em mais do que um bloco.

A representação utilizada para relacionar esses identificadores realizou-se através de um grafo. Assim, sempre que seja detectada uma conectividade entre dois identificadores de blocos distintos, é criada uma aresta entre os nós correspondentes.

A estrutura utilizada para implementar o grafo consistiu numa *hashtable*, em que a chave representa os nós e o valor as arestas correspondentes. Esta lista de voxels é implementada com um *hashset*, para que ignore arestas duplicadas, e proporcione inserções com complexidade constante.

Por fim, é realizada uma pesquisa em profundidade através do algoritmo *Breadth-first Search*, como ilustrado na figura 5.2, de forma a obter os identificadores que compõem um dado objecto.

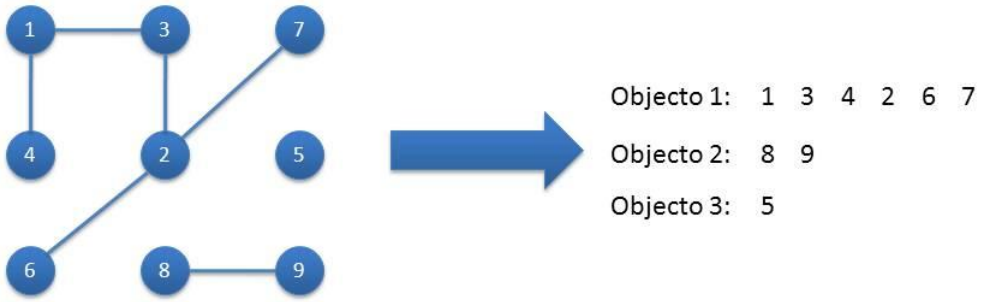


Figura 5.2: Algoritmo *Breadth-first Search*.

Para maximizar a utilização dos GPGPUs, é criada mais que uma *comand queue* para o mesmo GPGPU. Esse pormenor de implementação faz com que em dispositivos que permitam execuções em concorrência com transferências de memória, se possam transferir dados relativos a um bloco, enquanto o GPGPU processa outro bloco.

Uma outra vantagem do uso de várias *comands queues* para o mesmo GPGPU diz

respeito a uma nova tecnologia utilizada pelo Nvidia Fermi, que permite executar mais que um *kernel* em simultâneo, caso existam recursos para tal.

No que se refere ao processamento dos blocos, este é composto por três fases: identificação de subobjectos; detecção de conectividade entre subobjectos; junção de subobjectos. Para implementar as referidas fases foram realizadas três versões, que se distinguem no algoritmo bem como na interligação das fases.

A primeira e a segunda versão abordam duas formas distintas de numerar os voxels antes da junção em subobjectos.

A terceira versão tem como propósito melhorar a integração entre a fase de obtenção de conectividade entre subobjectos e a fase de junção dos subobjectos.

A realização destas três versões teve como objectivo analisar as limitações dos dispositivos bem como as formas de as contornar.

5.4.1 Primeira versão

Esta versão destaca-se por utilizar o índice dos voxels do bloco de dados para os numerar, sem necessitar de sincronismo para garantir identificadores únicos na numeração dos objectos.

A primeira fase de processamento de cada bloco, descrita no algoritmo 3, consiste em invocar um *kernel* que cria tantos *threads* quanto o número de voxels. Cada *thread* começa por numerar o voxel correspondente como o seu índice no bloco de dados, garantindo assim a unicidade entre identificadores. Seguidamente, inicia-se um ciclo em que em cada iteração os identificadores dos voxels vizinhos são analisados, caso exista um identificador inferior, este é adoptado. Após algumas iterações todos os objectos presentes no bloco de dados possuem um identificador único.

De forma a melhorar a performance dos algoritmos é utilizada a memória local do multiprocessador para realizar a propagação. Esta tem como função reduzir o número de iterações necessárias na propagação dos identificadores, bem como tornar os acessos à memória mais rápidos. A grande diferença no uso desta opção é que apenas são detectados subobjectos que estejam dentro da memória local, necessitando duma fase posterior para unir os subobjectos processados em multiprocessadores distintos. Mesmo que seja utilizada a memória global, iria ser necessária a fase posterior para unir os subobjectos, dado que alguns GPGPUs não garantem coerência na memória global entre os referidos multiprocessadores, visto utilizarem *caches* intermédias. Um outro problema resultante na utilização da memória global, reside no facto de tanto o OpenCL como o CUDA não permitirem pontos de sincronização entre *threads* de multiprocessadores distintos em memória global, o que não possibilita que a solução convirja apenas na primeira fase.

Esta fase termina armazenando os dados em memória global, para que possam transitar para a fase seguinte, uma vez que os dispositivos não permitem partilhar dados presentes em memória local entre *kernels*.

A segunda fase do algoritmo, descrito no algoritmo 4, consiste num *kernel* que gera,

Algoritmo 3 Kernel 1: Identificação dos Voxels.

```
__global unsigned char imageIn3D[];    // Imagem 3D original.
__global unsigned int imageOut3D[];    // Imagem 3D etiquetada.
__private Point3D globalID ← globalAddress(threadId);
__private Point3D localID ← localAddress(threadId);
__local unsigned int sharedMem[];      // Memória partilhada.
if imageIn3D[globalID.index] = BLACK then
    sharedMem[localID.index] ← globalID.index + 1;
else
    sharedMem[localID.index] ← 0;
end if
barrier(CLK_LOCAL_MEM_FENCE);
__local unsigned char isModify;
__private unsigned int currValue ← sharedMem[localID.index];
__private unsigned int newValue ← currValue;
while true do
    if localID.x > 0 then
        newValue ← sharedMem[localAddress(localID.x − 1, localID.y, localID.z)];
        if newValue > currValue) then
            isModify ← true;
            currValue ← newValue;
        end if
    end if
    (...) // Expansão para x + 1, y − 1, y + 1, z − 1 e z + 1.
    barrier(CLK_LOCAL_MEM_FENCE);
    if isModify = true then
        newValue ← currValue;
        while newValue ≠ sharedMem[newValue − 1] do
            newValue ← sharedMem[newValue − 1];
        end while
        if newValue ≠ 0 then
            sharedMem[localID.index]newValue − 1] ← new_value;
        else
            sharedMem[localID.index]newValue − 1] ← currValue;
        end if
    else
        break;
    end if
    barrier(CLK_LOCAL_MEM_FENCE);
    isModify ← false;
end while
imageOut3D[globalID.index] ← sharedMem[localID.index];
```

tal como o anterior, um *thread* por voxel.

Algoritmo 4 Kernel 2: Obter conectividade.

```
__global boolean isDone ← false;    // Afectada a true caso ocorra alterações.
__global unsigned int image3D[];    // Imagem 3D etiquetada.
__global unsigned int changes[];    // Vector de alterações.

__private Point3D globalID ← globalAddress(threadId);
__private Point3D localID ← localAddress(threadId);
__private unsigned int oldValue ← sharedMem[localID.index];
__private unsigned int newValue ← oldValue;
__private unsigned int newId ← BLACK;
if (globalID.x > 0) and (localID.x = 0) then
   newId ← image3D[globalAddress(globalID.x − 1, globalID.y, globalID.z)];
   if (newId ≠ WHITE) and (newValue > newId) then
      newValue ← newId;
   end if
end if
(...) // Expansão para x + 1, y − 1, y + 1, z − 1 e z + 1.
if newValue ≠ oldValue then
   isDone ← false;
   changes[oldValue] ← newValue;
end if
```

Cada *thread* começa por validar a sua posição, caso esteja numa fronteira analisa os identificadores dos vizinhos presentes em memória global, caso contrário termina a execução. Ao validar os identificadores vizinhos, caso estes possuam identificadores distintos, é estabelecida uma relação entre ambos num vector de alterações, para que posteriormente possam ser realizadas modificações de forma a existir apenas um identificador por objecto.

Para reduzir a quantidade de escritas ao vector de alterações, visto este se encontrar em memória global, cada *thread* utiliza um registo no multiprocessador para guardar o identificador menor entre todos os vizinhos em cada iteração. Assim, só efectua a escrita na memória global no final da validação dos vizinhos, caso o identificador obtido seja inferior ao corrente. Desta forma, apenas é efectuada uma escrita em memória global por iteração.

Para que a convergência do algoritmo seja maior, é realizado um processamento adicional que objectiva encontrar um representante único através da transitividade entre identificadores, como está ilustrado no algoritmo 5. Este processamento consiste em percorrer todo o vector de alterações, e para cada identificador é procurado o representante, actualizando todo o percurso. Assim, é possível acelerar a convergência do algoritmo, diminuindo o número de iterações.

Nesta versão a optimização do vector de alterações é realizado em CPU, numa perspectiva de evitar um processamento em memória global com acessos não contíguos. Em seguida, o vector de alterações é novamente enviado para o GPGPU para que sejam efectuadas as alterações aos identificadores no bloco de dados.

Algoritmo 5 Optimização do vector de alterações.

```
unsigned char changeVec[];    // Vector de alterações.
Stack<int> path ←new Stack<int>();
for i = 1 → changeVec.lenght() do
  newID ← changeVec[i];
  if newID ≠ 0 then
    path.push(i);
    while (changeVec[newID] ≠ 0) and (changeVec[newID] < newID) do
      path.push(newID);
      newID ← changeVec[newID];
    end while
    while path.isEmpty() do
      changeVec[path.top()] ← newID;
      path.pop();
    end while
  end if
  i ← i + 1;
end for
```

Após transferir as alterações, é invocado um terceiro *kernel* que tem como objectivo alterar os identificadores do bloco de dados, consoante as modificações presentes no vector de alterações, como pode ser visualizado no algoritmo 6.

Algoritmo 6 Kernel 3: Alteração dos identificadores.

```
__global unsigned int image3D[];    // Imagem 3D.
__global unsigned int changes[];    // Vector de alterações.

__private Point3D globalID ← globalAddress(threadId);
__private unsigned int currId ← image3D[globalID.index];

if currId ≠ WHITE then
  __private unsigned int newId ← changes[globalID.index + 1];
  if (newId ≠ WHITE) and (currId ⩾ newId) then
    image3D[globalID.index] ← newId;
  end if
end if
```

Este *kernel*, tal como os anteriores, cria um *thread* por voxel, o qual obtém os identificadores correspondentes no bloco de dados. Seguidamente, cada *thread* consulta o vector de alterações, caso exista um novo identificador associado ao valor do voxel, este é modificado e escrito no bloco de dados para que os objectos sejam apenas representados por um único identificador.

Esta forma de fusão não é efectuada em apenas uma iteração, dado que podem existir relações que não são detectadas apenas numa validação. Logo, torna-se necessário invocar os *kernels* dois e três até que não seja detectada nenhuma alteração.

Um aspecto bastante importante nesta versão corresponde ao vector de alterações. Este consiste num vector que apresenta como principal finalidade relacionar dois identificadores, no qual o maior terá de ser modificado para o menor. Para relacionar estes identificadores é atribuído à célula de memória, endereçada pelo índice correspondente

ao identificador a mudar, o valor do novo identificador.

Outro aspecto relevante a referir diz respeito às transferências entre CPU e GPGPU. Como é possível analisar através da figura 5.3, são enviados para o primeiro *kernel* o volume de dados e respectiva dimensão.

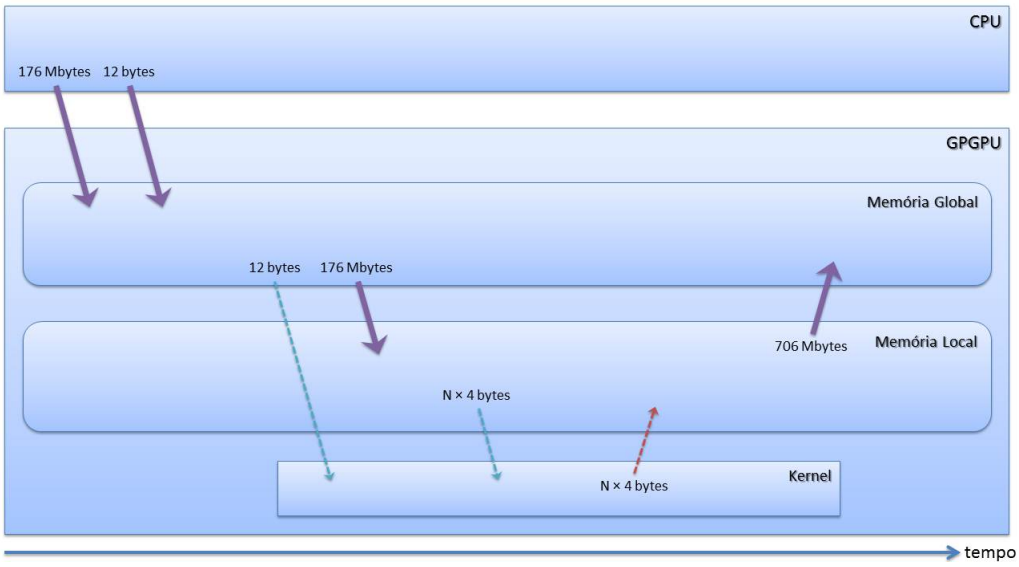


Figura 5.3: Transferências de memória do primeiro *kernel*.

Tanto o volume de dados como as dimensões mantêm-se em memória durante todo o processamento, sendo assim apenas necessário enviar o vector de alterações e uma variável de estado para o segundo *kernel*, como se pode observar na figura 5.4.

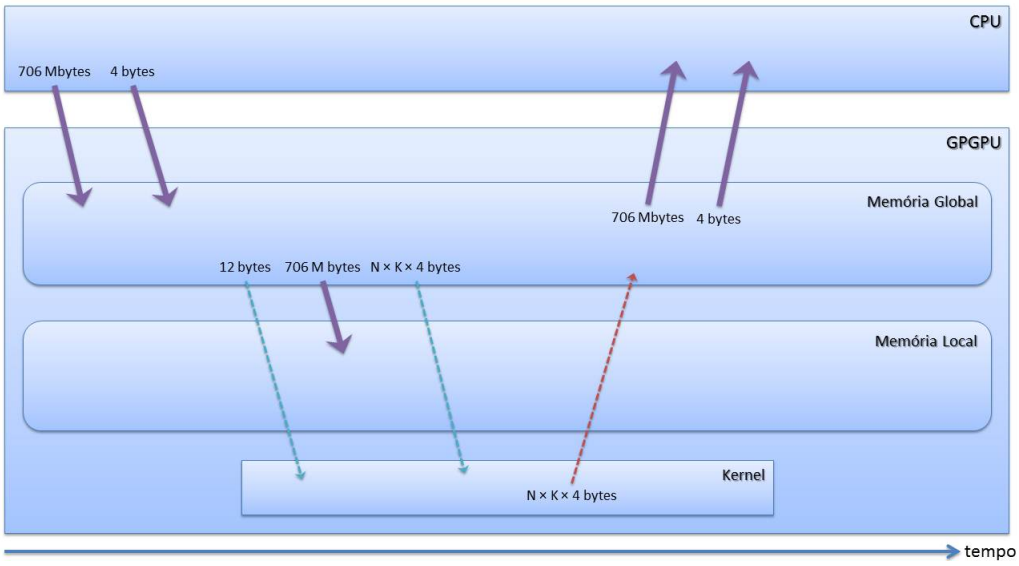


Figura 5.4: Transferências de memória do segundo *kernel*.

Após executar o segundo *kernel*, são transferidos para o CPU o vector de alterações e a variável de estado.

Por fim, é necessário enviar novamente o vector de alterações para o GPU para que as alterações sejam realizadas, tal como se encontra descrito na figura 5.5.

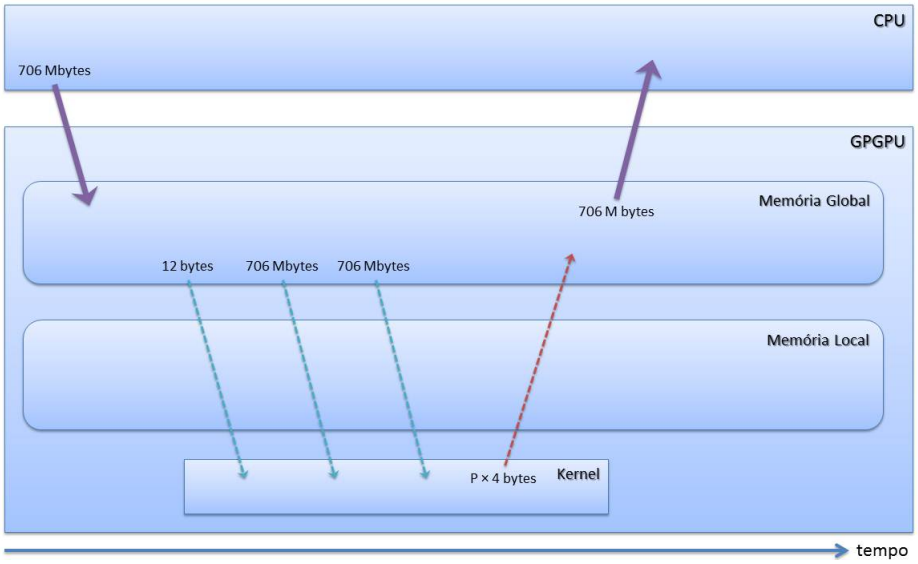


Figura 5.5: Transferências de memória do terceiro *kernel*.

Após terminar o terceiro *kernel*, o bloco de dados processado é transferido para CPU.

5.4.2 Segunda versão

Esta versão foi desenvolvida com o propósito de reduzir o número de identificadores utilizados, de forma a reduzir a dimensão do vector de alterações, dado esta constituir a principal limitação da versão anterior, como pode ser verificado no capítulo seguinte. A principal alteração realizada residiu na forma de distribuição dos identificadores, sendo esta agora efectuada através de instruções atómicas, ao contrário da versão anterior que utiliza o índice dos voxels no volume de dados.

As instruções atómicas permitem o incremento de uma variável global ou local de forma atómica.

Através deste mecanismo, sempre que se atribui um identificador a um voxel, a variável é incrementada atomicamente, garantido unicidade de identificadores.

No final da distribuição é possível obter um valor, regularmente bastante inferior à dimensão do bloco de dados, possibilitando assim definir uma dimensão para o vector de alterações para a segunda fase do processamento.

O *kernel* proposto para esta primeira fase do processamento do bloco de dados, ilustrado no algoritmo 7, tem início, tal como os anteriores, com a criação de um *thread* por voxel. Seguidamente, valida a vizinhança de cada um deles, caso o voxel tenha os vizinhos $x-1$, $y-1$ e $z-1$ a branco ou não existam, é requerido um identificador atómico para esse voxel. Todos os restantes entram em espera activa até que um dos vizinhos seja identificado. Deste modo, são utilizados menos identificadores, reduzindo bastante a dimensão do vector de alterações para a fase seguinte.

Algoritmo 7 Kernel 1: Identificação dos Voxels.

```
__global unsigned char imageIn3D[];   // Imagem 3D original.
__global unsigned int imageOut3D[];   // Imagem 3D etiquetada.
__global unsigned int counter;        // Contador de identifcadores.

__private Point3D globalID ← globalAddress(threadId);
__private Point3D localID ← localAddress(threadId);
__private unsigned int neighbor_0 ← localAddress(localID.x − 1, localID.y, localID.z);
__private unsigned int neighbor_1 ← localAddress(localID.x, localID.y − 1, localID.z);
__private unsigned int neighbor_2 ← localAddress(localID.x, localID.y, localID.z − 1);
__local unsigned int sharedMem[];     // Memória partilhada.
sharedMem[localID.index] ← imageIn3D[globalID.index];
barrier(CLK_LOCAL_MEM_FENCE);
if sharedMem[localID.index] = BLACK then
   __private unsigned int v_0 ← sharedMem[neighbor_0];
   __private unsigned int v_1 ← sharedMem[neighbor_1];
   __private unsigned int v_2 ← sharedMem[neighbor_2];
   if (v_0 = WHITE) and (v_1 = WHITE) and (v_2 = WHITE) then
      sharedMem[localID.index] ← atomicInc(counter);
   end if
end if
barrier(CLK_LOCAL_MEM_FENCE);
while sharedMem[localID.index] = BLACK do
   __private unsigned int value ← BLACK;
   if localID.x > 0 then
      value ← sharedMem[neighbor_0];
      if (value ≠ WHITE) and (value > 0) then
         sharedMem[localID.index] ← value;
         break;
      end if
   end if
   (...) // Expansão para y − 1 e z − 1.
end while
imageOut3D[globalID.index] ← sharedMem[localID.index];
```

Tal como na versão anterior, a convergência ocorre apenas em memória local, o que permite reduzir o número de iterações, assim como os tempos de acesso à memória.

Após a primeira fase, os subobjectos do mesmo grupo não se encontram unidos, sendo necessário efectuar essa junção numa segunda fase, como se pode verificar no algoritmo 8. Essa operação não aumenta significativamente o desempenho, dado que os *threads* que realizam essa junção não efectuam processamento na versão anterior.

Algoritmo 8 Kernel 2: Obter conectividade.

```
__global boolean isDone ← false;    // Afectada a true caso ocorra alterações.
__global unsigned int image3D[];    // Imagem 3D etiquetada.
__global unsigned int changes[];    // Vector de alterações.

__private Point3D globalID ← globalAddress(threadId);
__private Point3D localID ← localAddress(threadId);
__local unsigned int sharedMem[];       // Memória partilhada.
sharedMem[localID.index] ← image3D[globalID.index];
barrier(CLK_LOCAL_MEM_FENCE);
__private unsigned int oldValue ← sharedMem[localID.index];
__private unsigned int newValue ← oldValue;
__private unsigned int newId ← BLACK;
if globalID.x > 0 then
   if local.x > 0 then
      newId ← sharedMem[localAddress(localID.x − 1, localID.y, localID.z)];
   else
      newId ← image3D[globalAddress(globalID.x − 1, globalID.y, globalID.z)];
   end if
   if (newId ≠ WHITE) and (newValue > newId) then
      newValue ← newId;
   end if
end if
(...) // Expansão para x + 1, y − 1, y + 1, z − 1 e z + 1.
if newValue ≠ oldValue then
   isDone ← false;
   changes[oldValue] ← newValue;
end if
```

Nesta nova versão do segundo *kernel*, todos os voxels começam por obter o seu identificador, presente em memória global, e escrevem-no em memória local, para que os *threads* presentes no mesmo multiprocessador possam consultar esse valor.

De seguida, cada *thread* inicia um ciclo, em que para cada iteração são validados os voxels vizinhos, caso exista um identificador inferior, o mesmo é adoptado. Este ciclo termina quando não existir nenhuma alteração realizada pelos *threads* do mesmo multiprocessador.

Um aspecto importante de referir diz respeito aos acessos à memória, dado estes variarem consoante a posição dos vizinhos. Caso os vizinhos estejam presentes no mesmo multiprocessador, o acesso é efectuado através da memória local. Caso contrário, é obtido o identificador através da memória global.

Este detalhe de implementação faz com que se reduza o número dos acessos à memória global, o que implica que no final de cada iteração, caso algum *thread* efectue uma

alteração esta seja colocada em memória local e em memória global.

Relativamente às alterações efectuadas nas transferências de memória, nesta versão apenas é necessário transferir o contador de objectos para GPU antes da execução do primeiro *kernel*. Após a execução do *kernel*, o contador é novamente transferido para CPU, como está ilustrado na figura 5.6. Uma outra alteração, ilustrada na figura 5.7, diz respeito ao segundo *kernel*, que efectua acessos à memória local, ao contrário do semelhante da versão anterior.

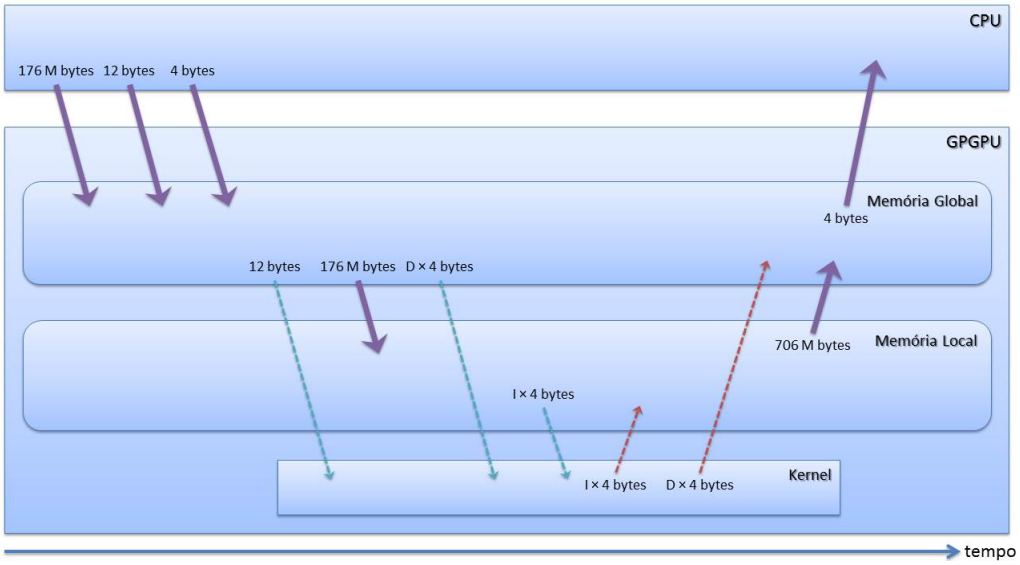


Figura 5.6: Transferências de memória do primeiro *kernel* modificado.

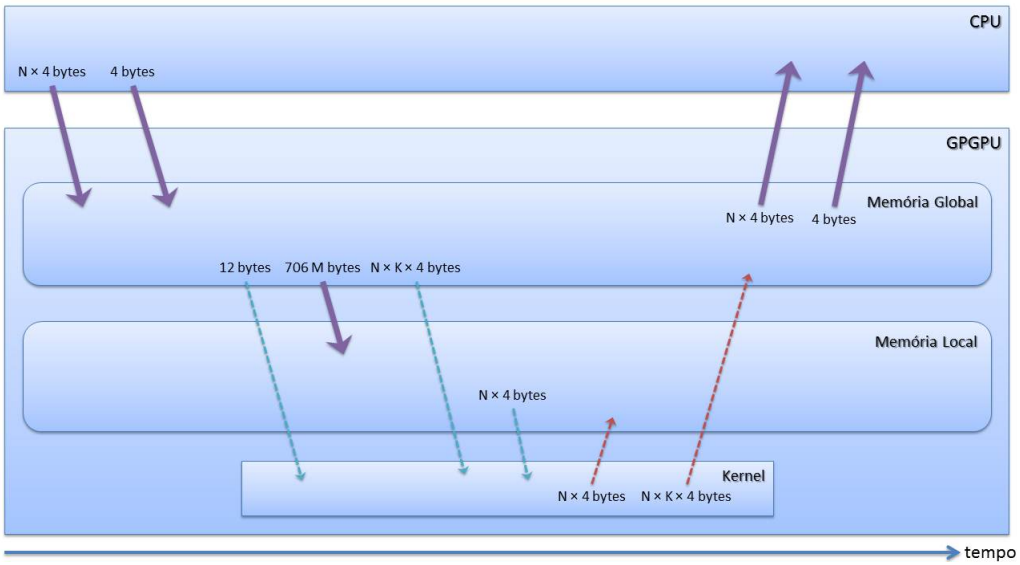


Figura 5.7: Transferências de memória do segundo *kernel* modificado.

5.4.3 Terceira versão

Esta versão apresenta como finalidade a redução do número de transferências realizadas entre CPU e GPGPU, durante a iteração da segunda e terceira fase do processamento do bloco de dados.

Para efectuar essa redução, é realizado o processamento do vector de alterações em GPGPU após ser executado o segundo *kernel*, como é descrito no algoritmo 9. Este aspecto faz com que seja apenas necessário iterar as execuções do segundo *kernel* até que não ocorra nenhuma alteração, transferindo apenas a variável que é afectada na ocorrência de alterações.

Algoritmo 9 Kernel 3: Alteração dos identificadores (Optimizado).

```
__global unsigned int image3D[];    // Imagem 3D.
__global unsigned int changeVec[];    // Vector de alterações.

__private Point3D globalID ← globalAddress(threadId);
__private unsigned int currId ← image3D[globalID.index];

if currId ≠ WHITE then
  __private unsigned int newId ← changeVec[globalID.index + 1];
  while (changeVec[newID] ≠ WHITE) and (changeVec[newID] < newID) do
    newID ← changeVec[newID];
  end while
  if (newId ≠ WHITE) and (currId ⩾ newId) then
    image3D[globalID.index] ← newId;
  end if
end if
```

Esta nova função consiste em cada *thread* consultar o identificador do seu voxel no volume de dados, sendo depois consultado o vector de alterações. Caso exista alguma alteração a realizar ao identificador presente no volume de dados, cada *thread* verifica se esse novo identificador também possui alguma alteração; caso possua é repetido este processamento, caso contrário esse novo valor é colocado associado ao identificador presente no volume de dados.

5.4.4 Aspectos importantes

Para melhorar os tempos de transferências entre CPU e GPGPU, foram utilizadas transferências de memória do tipo *pinned* para a recepção do volume de dados bem como para o envio e recepção do vector de alterações. Este detalhe, como já foi referido anteriormente no capítulo do estado da arte, consiste em manter as páginas *locked* em RAM de modo a que estas não sejam transferidas para o disco.

A opção de apenas efectuar este tipo de transferências para os blocos de dados e para o vector de alterações baseia-se no facto de estas serem as mais significativas em relação ao tempo. Uma outra razão diz respeito ao facto de que a utilização excessiva destes mecanismos pode piorar o desempenho, dado que o número de páginas livres para outras aplicações diminui.

Uma outra característica destas implementações refere-se à dimensão dos grupos de *threads*. Estes foram calculados utilizando uma ferramenta da nVidia, que através da dimensão dos grupos, da dimensão da memória partilhada e do número de registos utilizados, define a taxa de utilização do GPGPU. Assim, através dos recursos utilizados pelas implementações foi calculada a constante que maximiza a utilização do GPGPU, como poderá ser verificado no capítulo seguinte.

De forma a analisar as implementações efectuadas, de seguida encontra-se uma descrição dos tempos de resposta associados a cada volume de dados, bem como uma análise detalhada aos mesmos. No capítulo seguinte é possível verificar a diferença entre as soluções sequenciais e as soluções paralelas, e também a diferença de desempenho associado aos dispositivos utilizados.

6

Avaliação

Para analisar as implementações foram realizados testes a diversos níveis, de forma a obter as limitações inerentes a cada uma. A avaliação das implementações consistiu em analisar os tempos de resposta, assim como alguns aspectos inerentes às implementações que variam entre as sequenciais e as que utilizam paralelismo. A necessidade de efectuar esta distinção tem como fundamento as limitações inerentes a cada classe de soluções, dado que as implementações sequenciais estão limitadas pelo paralelismo e pelos acessos à memória, e as implementações paralelas estão limitadas pelas transferências entre CPU e GPGPU.

No que diz respeito à análise das versões sequenciais, esta começa por analisar a gestão de memória virtual realizada pelo sistema operativo. Esta análise é relevante dada a quantidade de memória utilizada pelos algoritmos, sendo que, tanto as operações de leitura como as de escrita em RAM e em memória secundária têm associadas latências significativas para o desempenho dos algoritmos.

Relativamente às versões que utilizam GPGPUs, o principal ponto em análise tem a ver com as transferências de memória entre CPU e GPGPU, dado estas serem a principal limitação do mesmo. A minimização da quantidade de dados a transferir aumenta o desempenho do algoritmo.

Um outro ponto bastante relevante de analisar diz respeito à diferença no tempo de resposta e das transferências de memória realizadas, em função do GPGPU em uso. Este ponto é bastante importante, dado que cada GPGPU possui aspectos únicos. Dado o funcionamento destes algoritmos, estes necessitam ser ajustados ao GPGPU em questão para que se maximize o seu desempenho. Nesta análise é possível detectar as limitações que estão associadas ao algoritmo e aos GPGPUs, podendo-se assim efectuar uma melhor análise à escalabilidade do algoritmo.

Por fim, são analisados os pontos relativos às optimizações nos acessos à memória, bem como no escalonamento dos *threads*. Estes pontos são também relevantes, dado que escolhas correctas a este nível significam elevadas taxas de ocupação dos recursos *hardware* do GPU, o que naturalmente diminui os tempos de execução.

6.1 Descrição do *hardware* e do *software* utilizados

No decorrer das experiências foi utilizado o *hardware* das estações de trabalho que são PCs, que têm como *hardware* base: um processador Xeon E5504 (4-core); 12 Gbytes RAM; um GPU para a visualização e um acelerador. Um dos aceleradores disponíveis é o NVIDIA c2050 (Fermi), que consiste num GPGPU com 448 *cores* CUDA, com 3 Gbytes de memória, sendo a sua capacidade de processamento de 1 Tflop em precisão simples e 515 Gflops em precisão dupla, efectuando um consumo de 238 W.

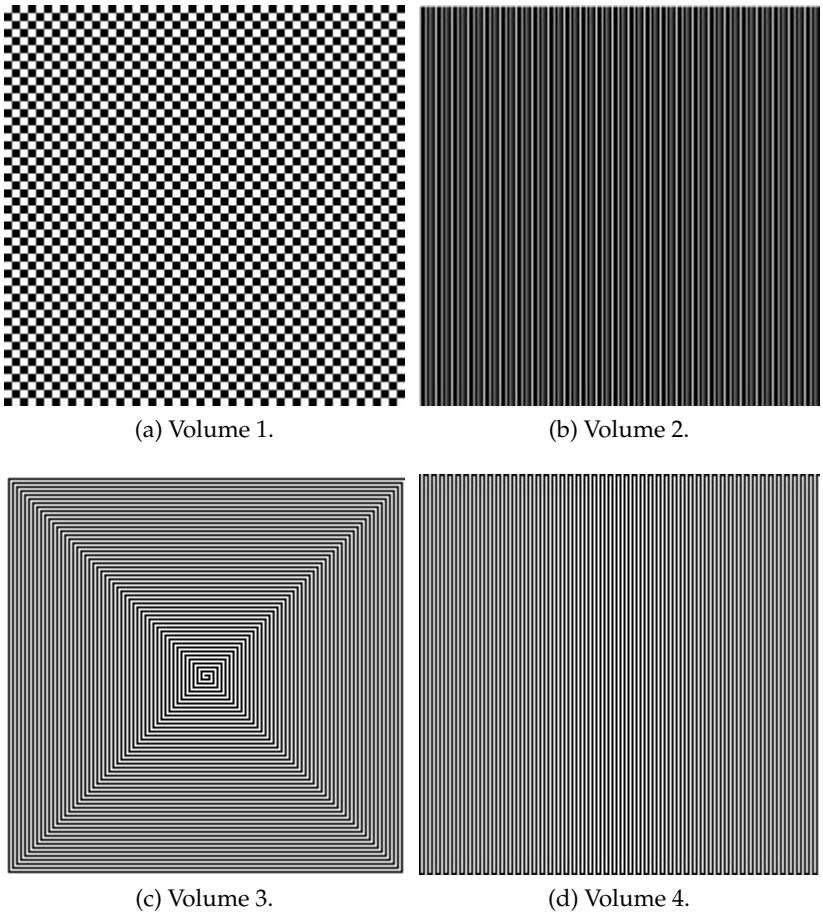
Também foi usada outra máquina com *hardware* base igual, mas equipada com dois aceleradores NVIDIA c1060 (Tesla), que possuem 240 *cores* CUDA, com 4 Gbytes de memória interna, com capacidade de 933 Gflops em precisão simples e 78 Gflops em precisão dupla, com um consumo de 187.8 W.

Relativamente ao *software* utilizado, este consistiu na plataforma OpenCL sobre o sistema operativo Linux na distribuição Ubuntu.

6.2 Volumes de dados utilizados

Para a análise dos algoritmos foram utilizados diversos volumes de dados tridimensionais fictícios, que variam na dimensão e na morfologia. Foi também utilizado um conjunto de dados obtido a partir de um material compósito obtido através de microtomografia computorizada. Relativamente à dimensão do volume de dados fictícios, foram utilizados volumes de dimensão variável entre $100 \times 100 \times 100$ e $1200 \times 1200 \times 1200$. Esta diferença visa analisar a escalabilidade dos algoritmos. No que concerne à morfologia, foram utilizados três tipos de volumes, como ilustrado na figura 6.1. O primeiro volume de dados consiste num xadrez tridimensional, que tem como finalidade criar um elevado número de objectos de pequenas dimensões. O segundo volume de dados consiste em diversos paralelepípedos que se estendem em toda a dimensão z, analisando assim o comportamento dos algoritmos em volumes com bastantes objectos de grandes dimensões. O terceiro volume de dados consiste num único objecto que o ocupa por completo, sendo a sua forma em espiral de forma a maximizar a sua dimensão, assim é possível analisar os algoritmos face a um único objecto de dimensões elevadas.

De forma a justificar os volumes de dados fictícios utilizados, a figura 6.2 apresenta imagens de conjuntos de dados reais. Estes dados permitem verificar a diversidade de objectos existentes, tendo alguns deles semelhanças com as amostras sintéticas atrás descritas; o conjunto de dados real corresponde a uma amostra com aspecto semelhante ao da figura 6.1.



(a) Volume 1. (b) Volume 2.

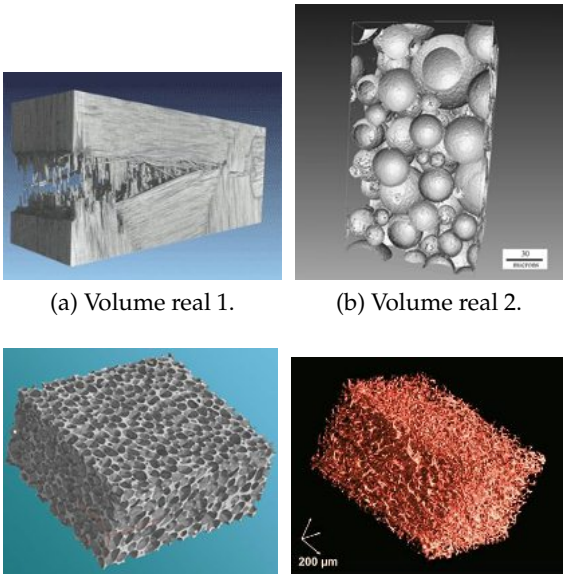
(c) Volume 3. (d) Volume 4.

Figura 6.1: Volumes de dados.

6.3 Algoritmo One-Pass

Este algoritmo foi descrito no capítulo anterior. Quando se codificou o algoritmo e se colocou o programa em execução, detectou-se que o seu grande problema reside na realização de acessos irregulares à memória, o que degrada bastante o seu desempenho devido à gestão de memória do sistema operativo que se baseia em páginas; esta irregularidade também prejudica a utilização das *caches* existentes entre a memória RAM e o CPU.

Os acessos irregulares à memória são realizados na fase de expansão, quando se percorrem voxels localizados em regiões de memória distintas. Embora na representação gráfica os voxels possam estar ligados, internamente estes encontram-se localizados em regiões distintas. Como é possível analisar na figura 6.3, a matriz tridimensional está internamente alocada numa região contígua de memória com apenas uma dimensão. Logo, para se aceder a um voxel com coordenada $y + 1$, é necessário realizar um deslocamento igual à dimensão máxima em x. Caso seja necessário aceder a um voxel com a coordenada $z + 1$, é necessário realizar um deslocamento igual à multiplicação das dimensões máximas de x e y.



(a) Volume real 1. (b) Volume real 2.

(c) Volume real 3. (d) Volume real 4.

Figura 6.2: Volumes de dados real.

Um aspecto importante de analisar diz respeito aos deslocamentos de memória realizados numa expansão. Quando esta ocorre, são validados todos os voxels vizinhos que possuem conectividade através das faces. Este facto faz com que seja necessário aceder às posições, $x-1$, $y-1$, $z-1$, $x+1$, $y+1$ e $z+1$, o que origina seis deslocamentos de memória, onde apenas dois estão contíguos como a posição corrente. Numa análise ao nível das *caches*, sendo o deslocamento superior à quantidade de memória transferida pelo sistema em cada acesso, não é possível tirar partido do mecanismo de previsão inerente às *caches* durante a expansão de um dado voxel, a não ser em dois acessos, $x-1$ e $x+1$. Porém, caso a *cache* possua dimensão suficiente para manter os blocos entre expansões, a situação melhora embora o padrão de acessos à memória continue a ser irregular.

Relativamente ao sistema de gestão de memória virtual do sistema operativo, a realização de acessos irregulares à memória torna necessárias mais transferências de páginas, sendo que a TLB não consegue guardar as correspondências entre o número de página virtual e o número da página física. Dado esse facto, é necessário aceder à memória RAM para obter essa correspondência o que aumenta bastante os tempos de resposta.

Um outro aspecto bastante importante de referir diz respeito à gestão das páginas entre a RAM e a memória secundária. Durante a fase de expansão, são modificadas diversas páginas, as quais estão associadas a voxels do objecto em questão. Quando os objectos apresentam dimensões elevadas, as páginas modificadas no início necessitam ser escritas para memória secundária para dar lugar a novas. Após a expansão, o percurso é retomado e as páginas relativas aos voxels a percorrer necessitam de ser novamente transferidas para a RAM, caso tenham sido modificadas. Este facto faz com que os tempos de acessos aumentem bastante.

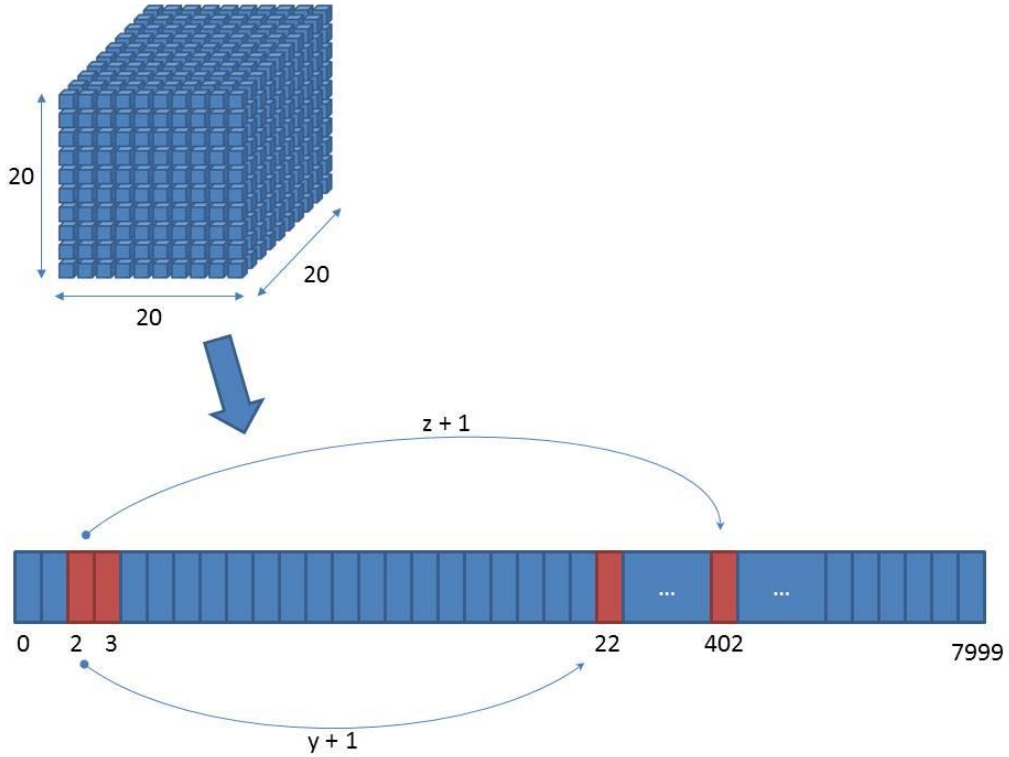


Figura 6.3: Localização do volume de dados em memória.

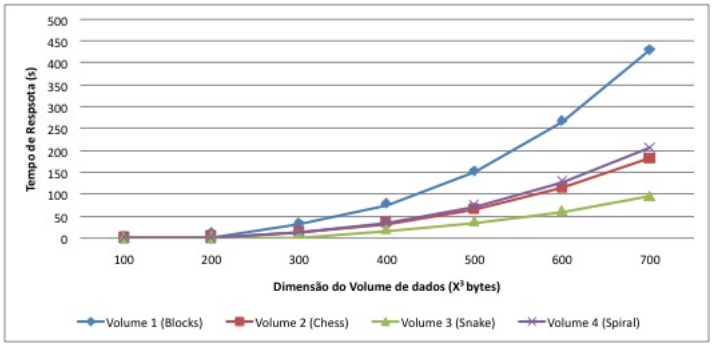
No que se refere à análise quantitativa do algoritmo, seguidamente serão apresentados os tempos de resposta do algoritmo em função da dimensão e morfologia dos volumes de dados utilizados.

	Blocks		Chess		Snake		Spiral	
Dimensão (X^3 bytes)	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s
100	1.05	0.95	0.54	1.87	0.28	3.55	0.51	1.96
200	8.98	0.89	4.29	1.87	2.25	3.55	4.26	1.88
300	31.67	0.85	14.47	1.87	7.58	3.56	14.94	1.81
400	76.71	0.83	34.27	1.87	17.91	3.57	35.67	1.79
500	151.94	0.82	66.65	1.88	35.05	3.57	73.50	1.70
600	266.76	0.81	115.58	1.87	60.37	3.58	127.54	1.69
700	430.68	0.80	183.21	1.87	96.10	3.57	206.97	1.66
800	645.56	0.79	274.94	1.86	143.24	3.57	309.94	1.65
900	927.63	0.79	390.17	1.87	205.29	3.55	444.10	1.64
1000	1278.57	0.78	535.54	1.87	282.18	3.54	615.80	1.62
1100	1713.71	0.78	713.36	1.87	376.77	3.53	810.71	1.64
1200	5606.99	0.24	1536.98	0.87	909.07	1.46	4748.37	0.28

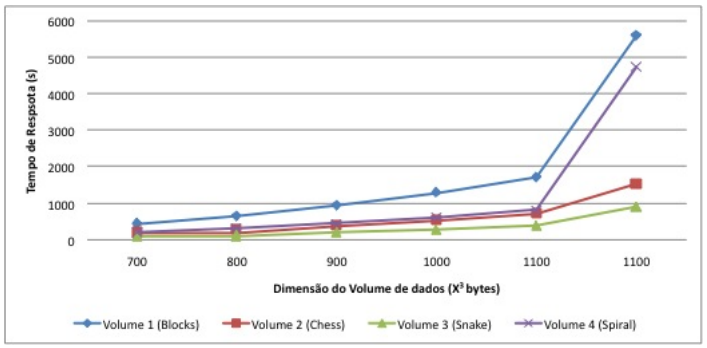
Tabela 6.1: Tempos de resposta do algoritmo *one-pass*.

Após a análise dos resultados obtidos, é possível concluir que o algoritmo possui tempos de resposta bastante elevados, não sendo adequado para ambientes interactivos. Um aspecto bastante importante de referir, referente à visualização do problema dos acessos irregulares, diz respeito à variação dos tempos de resposta em função do volume de dados. Esse aspecto pode ser visualizado no terceiro volume, que apresenta apenas a espessura de um voxel, fazendo assim com que tenha um único vizinho para expandir.

Quando utilizados volumes com uma elevada área, os tempos de resposta crescem



(a)



(b)

Figura 6.4: Tempo de resposta do algoritmo *one-pass*.

bastante, visto aumentar o número de expansões irregulares.

6.4 Algoritmo Two-Pass

Através do modo de funcionamento deste algoritmo, foi possível detectar que ao contrário do algoritmo anterior, este algoritmo tem a vantagem de apenas efectuar acessos contínuos em memória no que concerne ao volume de dados, visto não efectuar acessos irregulares a regiões de memória relativa a voxels, como o algoritmo anterior. O grande problema deste algoritmo reside na estrutura de dados, mais concretamente na operação *Find*, que visa obter o representante de um grupo. Embora a complexidade desta operação seja um aspecto limitativo da solução, a principal limitação reside na memória requerida pela estrutura de dados.

Relativamente à operação *Find*, quando o caminho de ligação entre o nó e a raíz da árvore é bastante extenso, torna-se necessário realizar bastantes acessos irregulares à memória, para ser possível obter o representante. Em casos extremos é necessário percorrer tantos nós quantos os voxels que estão associados ao grupo. Este problema pode ser minimizado através de duas heurísticas, usualmente utilizadas nesta estrutura de dados, que visam reduzir a sua complexidade. Essas heurísticas são designadas de *Union-by-Rank* e *Path Compression*.

A heurística *Union-by-Rank* ou união pela profundidade consiste em realizar uma modificação à operação *Union* de forma que a raíz da árvore com menor dimensão seja ligada ao nó representante da outra árvore, fazendo assim com que a árvore resultante possua uma menor profundidade. Esta alteração resulta numa redução significativa da complexidade computacional da operação *Find*, visto que assim nunca irá ocorrer o caso em que a profundidade da árvore seja igual ao seu número de elementos.

Relativamente à heurística *Path Compression* ou compressão do percurso, esta consiste em modificar a operação *Find*, de forma a ligar todos os nós de um dado conjunto directamente à raíz. Esta modificação é realizada após efectuar o percurso entre um dado nó e a raíz, para que posteriormente apenas seja consultado esse voxel e a raíz.

Como é possível analisar, esta operação reduz a complexidade amortizada da operação *Find*, dado que após uma compressão do percurso, as operações seguintes resultam num percurso bastante menor.

Embora sejam utilizadas as heurísticas supracitadas, o problema principal mantém-se, como é possível analisar nos resultados obtidos. Este algoritmo obtém bons tempos de resposta, quando os objectos presentes na imagem são relativamente pequenos, o que reduz o número de acessos irregulares nas operações realizadas pela estrutura de dados.

Relativamente à utilização das *caches*, uma vez que o percurso entre os voxels é realizado em conformidade com a localização dos mesmos em memória, é possível tirar partido deste mecanismo, mesmo na validação dos identificadores dos vizinhos, dado que os vizinhos numa dada iteração, estão alocados contiguamente com os vizinhos da iteração seguinte. No que concerne à estrutura de dados, esta também permite tirar partido da *cache*, tendo em conta o percurso utilizado numa pesquisa por o representante, este pode ser reutilizado na iteração seguinte, dado que o voxel processado anteriormente é vizinho do voxel da iteração corrente. Este facto pode não ocorrer, quando o número de nós validados é bastaste superior à dimensão da memória da *cache*.

Tal como o mecanismo de *cache*, este algoritmo também se ajusta ao mecanismo de gestão de memória virtual do sistema operativo, no que concerne ao percurso no bloco de dados, visto ser um percurso contíguo. Este facto é possível ser identificado através dos tempos de resposta, em comparação com o algoritmo anterior. O problema deste algoritmo ocorre quando os volumes são demasiado grandes, o que faz com que os acessos à memória aumentem de tal modo que a utilização destes mecanismos ainda agrava mais os tempos de resposta. Esse facto ocorre pela necessidade de percorrer as árvores dos grupos sempre que seja necessário unir dois grupos. Este aspecto faz com que existam muitos mais acessos à memória do que no algoritmo anterior.

Um outro aspecto relevante diz respeito à transferência de páginas entre a memória secundária e a RAM. Sendo o número de nós igual ao número de voxels, em imagens de grande dimensão, o sistema operativo necessita transferir páginas relativas a estes nós para a memória secundária. Assim, sempre que exista uma fusão, caso existam nós no percurso com páginas em memória secundária, torna-se necessário voltar a transferi-las para a RAM.

	Blocks		Chess		Snake		Spiral	
Dimensão (X^3 bytes)	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s
100	0.22	4.47	0.17	5.98	0.06	15.85	0.14	7.37
200	1.85	4.32	1.36	5.87	0.50	15.86	1.12	7.16
300	6.33	4.27	4.65	5.80	1.71	15.77	3.86	6.99
400	14.93	4.29	10.98	5.83	4.0645	15.75	9.04	7.08
500	29.49	4.24	21.48	5.82	7.98	15.66	17.67	7.07
600	51.19	4.22	37.23	5.80	13.80	15.65	30.65	7.05
700	2643.47	0.13	1561.75	0.22	448.8440	0.76	3458.36	0.10
800	> 3600.00	-	3417.76	0.15	> 3600.00	-	> 3600.00	-

Tabela 6.2: Tempos de resposta do algoritmo *two-pass*.

Como é possível constatar através dos resultados obtidos, este algoritmo apresenta melhores tempos de resposta face ao anterior, em imagens de pequenas dimensões. Este facto deve-se a apenas percorrer o volume de dados duas vezes de forma contígua. Através dos resultados é possível verificar que este algoritmo está limitado à dimensão do volume de dados, sendo que necessita de uma elevada quantidade de memória para efectuar o processamento. Quando é aplicado a volumes de dados superiores a $800 \times 800 \times 800$, o sistema operativo termina o processo, devido a não permitir o uso de tanta memória.

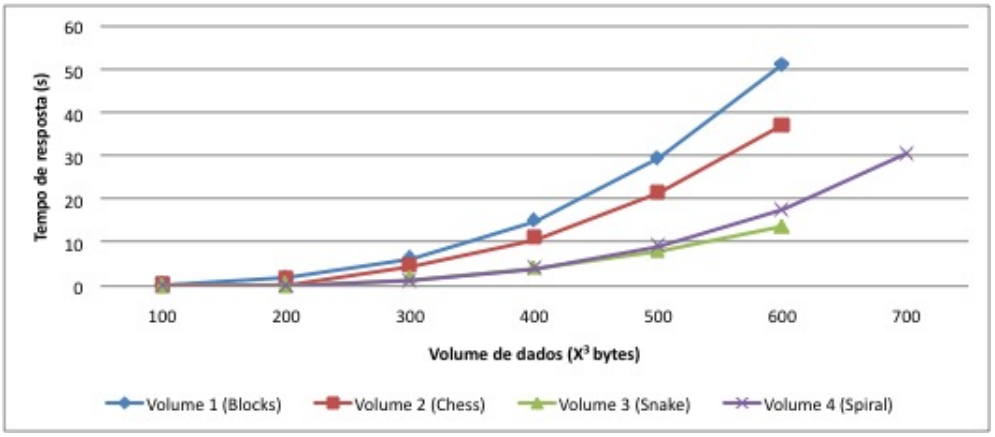


Figura 6.5: Tempo de resposta do algoritmo *two-pass*.

Esta limitação na dimensão das imagens, deve-se à quantidade de memória utilizada pela estrutura de dados. Esta necessita de oito bytes por cada voxel para poder guardar o identificador do nó e mais um byte para o apontador para o nó seguinte. Um outro aspecto relevante de extrair dos resultados obtidos diz respeito à variação do tempo em função da morfologia do volume de dados a processar. Verificam-se os piores resultados obtidos no primeiro e no segundo volume, sendo possível verificar que quanto maior a dimensão dos objectos, maior o tempo de resposta. Isto deve-se à árvore resultante, que possui uma maior profundidade.

6.5 Object Identifier

No que diz respeito ao algoritmo desenvolvido, este não possui os problemas descritos nas versões sequenciais, dado que quase todo o processamento do volume é realizado no GPGPU.

A análise desta implementação inicia-se pelos *threads* utilizados para processar os blocos. Estes, tal como já foi referido no capítulo anterior, obtêm os blocos assincronamente e efectuam todas as comunicações com o GPGPU de forma a processar os blocos.

O primeiro aspecto a analisar diz respeito à obtenção dos blocos. Esta operação necessita de sincronismo, devido a envolver operações sobre um contador global. Este ponto de sincronização não penaliza significativamente o tempo de resposta, visto que a função executada em exclusão mútua apresenta complexidade computacional constante. Torna-se bastante pertinente analisar as transferências de dados entre CPU e GPGPU, uma vez que constituem a principal limitação da programação através de GPGPUs.

Na tabela 6.3 é possível verificar o tempo de transferência dos blocos de dados em função do volume de dados e do tipo de transferência. Nesta análise quantitativa foram realizadas transferências de memória regulares e *pinned*.

	Envio		Recepção	
Dimensão (X^3 bytes)	Média (ms)	Desvio padrão (ms)	Média (ms)	Desvio padrão (ms)
100	0.17	0.00	0.63	0.00
200	2.46	0.03	4.99	0.00
300	9.09	0.00	16.85	0.00
400	21.87	0.23	39.93	0.00
500	64.46	2.85	109.82	0.00

Tabela 6.3: Tempos de transferência entre CPU e GPGPU.

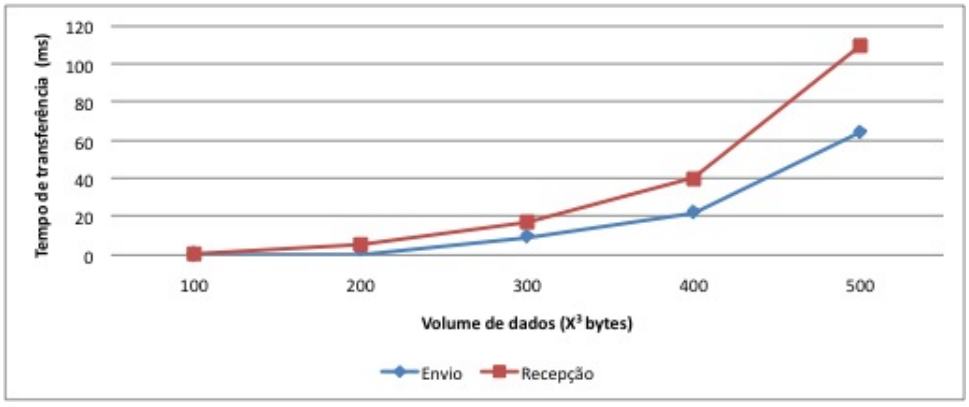


Figura 6.6: Tempos de transferência entre CPU e GPGPU.

Os tempos associados à emissão dos blocos para o GPGPU são inferiores à recepção, dado que o volume de dados enviado possui menos memória. Este utiliza para cada voxel um byte, enquanto que o volume de dados recebido utiliza quatro bytes para representar cada voxel.

No que se refere à forma de efectuar as transferências, apenas foram realizadas transferências *pinned* na recepção do volume de dados. Isto porque para realizar transferências de memória *pinned* é necessário criar afinidade entre o volume a transferir e um determinado GPGPU, bem como definir previamente a dimensão do bloco de dados. Sendo que os blocos são distribuídos assincronamente e possuem dimensão variável, não é possível garantir as condições necessárias à utilização de transferências de memória *pinned*, sendo apenas possível efectuar transferências de memória regulares.

Tal como é possível verificar na tabela 6.3 as transferências de memória *pinned* produzem taxas de transferências mais elevadas, visto que as páginas correspondentes estão em memória RAM quando iniciada a transferência. Segundo o artigo [32] este tipo de transferências consegue atingir taxas de cinco GBps em *bus* PCIe x16.

Segundo testes realizados à largura de banda, estas transferências são cinco vezes mais rápidas que as regulares.

Após terminar o processamento dos blocos, torna-se necessário construir o grafo que interliga objectos presentes em mais que um bloco de dados.

Nesta fase é necessário percorrer subconjuntos do volume de dados. Estes percursos consistem em iterar todos os voxels dos planos relativos a fronteiras entre os blocos de dados. Este processamento tal como já foi referido, é realizado em paralelo por diversos *threads*.

O percurso realizado pelos *threads* apresenta a particularidade de a localização dos voxels a iterar encontrar-se coerente com a localização dos mesmos em memória, o que tira partido dos mecanismos de *cache*.

	Criação		Pesquisa	
Volume	Média (ms)	Desvio padrão (ms)	Média (ms)	Desvio padrão (ms)
Blocks	111.29	3.09	11.00	0.26
Chess	556.17	3.21	239.78	0.50
Spiral	67.14	0.99	0.07	0.00
Snake	8.49	0.04	0.06	0.00

Tabela 6.4: Tempo de processamento na união dos blocos.

Como se pode analisar na tabela 6.4, o tempo de construção do grafo é bastante reduzido, o que se deve, essencialmente, à sua realização em paralelo e também ao facto de o número de blocos de dados ser bastante pequeno.

No que concerne à pesquisa em profundidade, visto existirem poucos blocos de dados, este também produz tempos de resposta diminutos.

De seguida, encontra-se descrita a análise efectuada ao processamento em GPGPUs. Para obter os resultados foram utilizadas duas *command queues* para o nVidia C2050. Tal como já foi referido no capítulo anterior, a solução foi implementada através de três versões, as quais produzem resultados pertinentes de analisar.

Um aspecto importante de referir relativo ao processamento consiste no número de *threads* utilizados pelos *kernels*. Este número baseia-se num múltiplo do utilizado pelo

scheduler do multiprocessador, de forma a maximizar a sua utilização.

Através de uma ferramenta da nVidia, que visa calcular a taxa de utilização do GPGPU, foi analisada a utilização dos registos dos multiprocessadores bem como a memória constante, de forma a obter a dimensão correcta para os grupos.

Esta análise foi realizada para o *hardware* utilizado, na qual se constatou que o número de *threads* que maximiza a utilização do GPGPU é de 255. Assim, cada *kernel* foi invocado para grupos de 255 *threads* indexados tridimensionalmente em blocos de $4 \times 4 \times 16$.

6.5.1 Primeira versão

Para analisar o processamento no GPGPU nesta primeira versão foram efectuados testes que descrevem os tempos: de execução dos *kernels*; de transferência nas fases intermédias; e do processamento realizado na optimização do vector de alterações em CPU.

Nesta análise é detectada também a coalescência dos acessos à memória global, visto ser uma das principais recomendações da nVidia para maximizar o desempenho dos GPGPUs, como se encontra referenciado no artigo [32].

	Blocks		Chess		Snake		Spiral	
Dimensão (X^3 bytes)	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s
100	0.02	40.88	0.02	55.16	0.03	32.36	0.03	30.51
200	0.17	46.14	0.12	67.30	0.28	28.90	0.22	36.95
300	0.67	40.51	0.38	71.41	1.10	24.59	0.70	38.39
400	1.82	35.10	0.87	73.49	3.16	20.27	1.61	39.67
500	3.94	31.72	1.70	73.52	7.90	15.82	3.51	35.58
600	6.43	33.57	2.96	72.92	14.87	14.53	6.10	35.39
700	9.02	38.02	4.66	73.67	30.05	11.41	9.95	34.47
800	11.77	43.49	6.77	75.67	47.84	10.70	15.32	33.43
900	16.49	44.22	10.64	68.50	73.28	9.95	23.69	30.77
1000	21.01	47.61	13.36	74.86	113.68	8.80	38.47	25.99
1100	29.26	45.50	17.94	74.21	174.71	7.62	51.08	26.06
1200	34.08	50.71	22.97	75.22	217.02	7.96	62.81	27.51

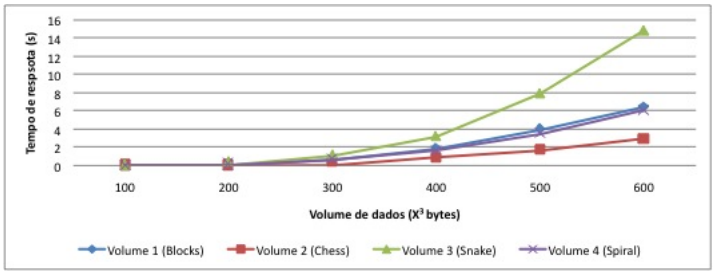
Tabela 6.5: Tempos de resposta da primeira versão do algoritmo *Object Identifier*.

O primeiro *kernel* apenas acede à memória global para efectuar transferências com a memória local. Essas transferências são efectuadas em coalescência, devido ao mapeamento dos *threads* ser igual ao mapeamento dos voxels, permitindo assim que cada multiprocessador transfira segmentos lineares de memória.

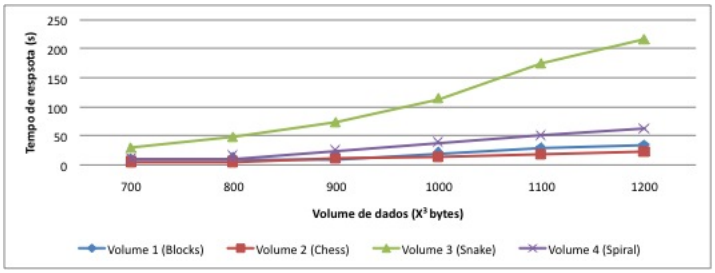
Relativamente ao número de iterações para a convergência, uma vez que esta é realizada através da memória local, no máximo são efectuadas tantas iterações quanto a dimensão da diagonal do bloco de memória local utilizada, dada a propagação ser efectuada através de inundação (flood fill).

Volume	Kernel 1 (s)	Kernel 2 (s)	Kernel 3 (s)	Optimização do vector de Alterações (s)
Blocks	0.22	0.09	0.02	0.26
Chess	0.21	0.11	0.02	0.27
Spiral	0.22	0.09	0.02	0.36
Snake	0.21	0.12	0.03	0.37

Tabela 6.6: Tempos de processamento da primeira versão do algoritmo *Object Identifier*.



(a)



(b)

Figura 6.7: Tempos de resposta da primeira versão do algoritmo *Object Identifier*.

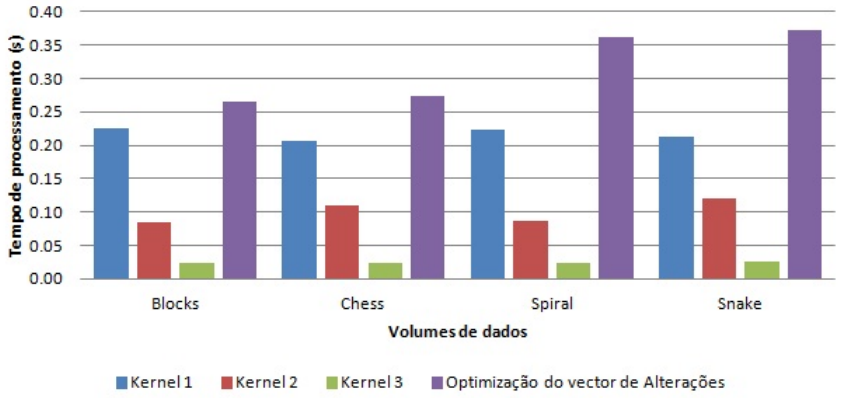


Figura 6.8: Tempos de processamento da primeira versão do algoritmo *Object Identifier*.

No que diz respeito ao tempo de execução do primeiro *kernel*, como se pode observar na tabela 6.6, o tempo médio de execução é bastante baixo, visto que cada *thread* efectua poucas operações, utilizando apenas a memória local para guardar valores intermédios.

O segundo *kernel* tem como objectivo unificar subobjectos processados por multiprocessadores distintos do *kernel* anterior.

Os acessos à memória realizados por este *kernel* são na sua grande maioria à memória global, o que adiciona alguma latência ao processamento.

Embora os acessos sejam realizados à memória global, é possível tirar algum partido de acessos coalescentes, dado ser possível definir um padrão de transferências por parte dos multiprocessadores.

O tempo de execução do segundo *kernel*, como se pode verificar na tabela 6.6, é bastante inferior ao *kernel* anterior, dado não possuir ciclos. Cada *thread* neste *kernel* efectua

um processamento de complexidade constante.

Embora o tempo unitário seja inferior ao anterior, o facto de ser executado diversas vezes, torna-o mais significativo no tempo total de processamento.

Um outro aspecto importante de analisar nesta versão, diz respeito à optimização do vector de alterações, o mesmo produz tempos de processamento bastante elevados, dada a dimensão do vector ser igual à do volume de dados.

	Envio		Recepção	
Volume	Média (ms)	Desvio padrão (ms)	Média (ms)	Desvio padrão (ms)
100	0.63	0.00	0.53	0.00
200	8.46	0.09	6.35	0.00
300	19.32	0.01	16.65	0.00
400	37.84	0.88	37.48	0.00
500	120.65	1.86	113.42	0.01

Tabela 6.7: Tempos de transferência do vector de alterações da primeira versão do algoritmo *Object Identifier*.

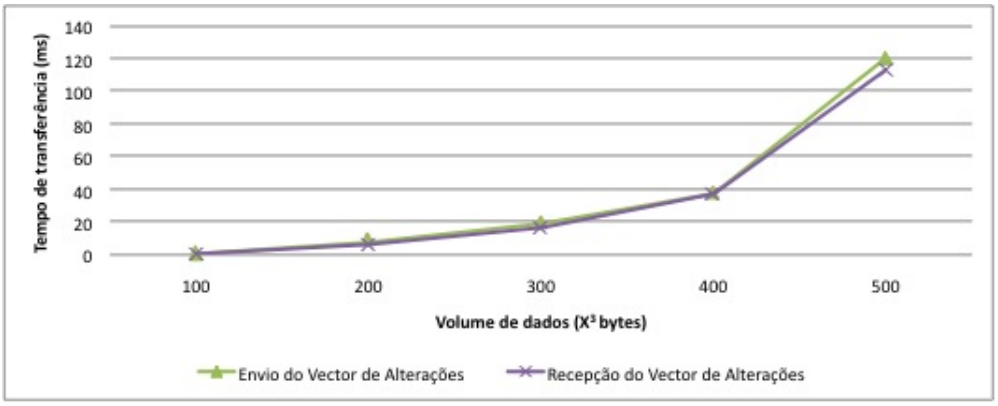


Figura 6.9: Tempos de transferência da primeira versão do algoritmo *Object Identifier*.

O grande problema associado ao vector de alterações reside nas transferências entre o CPU e o GPGPU, como ilustrado na tabela 6.7. Estes tempos derivam da elevada dimensão do vector, dado esta ser igual à do volume de dados, como foi descrito no capítulo anterior.

Este problema agrava-se devido a ser necessário efectuar diversas transferências durante as iterações do segundo e do terceiro *kernel*.

Relativamente ao terceiro *kernel*, este produz tempos de resposta bastante baixos, como ilustrado na tabela 6.6, devido a possuir também complexidade computacional constante. Embora o segundo *kernel* também possua uma complexidade constante, este terceiro é bastante mais rápido, uma vez que efectua apenas três acessos à memória, correspondendo a duas leituras e uma escrita.

No que diz respeito aos acessos realizados ao bloco de dados em memória global, estes são coalescentes, visto que cada *thread* apenas acede à posição correspondente.

No que se refere aos acessos ao vector de alterações, estes não são coalescentes, dado não existir um padrão entre os *threads* e as posições no vector.

Embora os acessos não sejam coalescentes, caso sejam utilizados GPGPUs com *caches* da memória global, os tempos de acesso ao vector de alterações podem ser minimizados, visto que cada posição do vector de alterações é consultada por diversos *threads*.

Para concluir a análise desta primeira versão, o gráfico presente na figura 6.10 apresenta as percentagens do tempo de resposta associadas às transferências e ao processamento em CPU e em GPGPU.

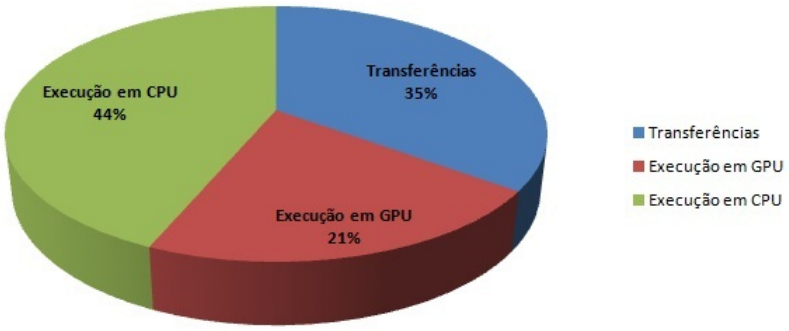


Figura 6.10: Decomposição do tempo total de resposta da primeira versão do algoritmo *Object Identifier*.

Neste gráfico é possível verificar que as transferências e o processamento em CPU constituem as principais limitações desta solução, visto possuírem uma elevada percentagem.

A percentagem de transferências é na sua grande maioria referente às transferências do vector de alterações.

No que se refere à percentagem do processamento, esta é também maioritariamente referente à optimização do vector de alterações. Assim, é possível concluir que a principal limitação desta solução reside na dimensão do vector de alterações.

6.5.2 Segunda versão

Esta versão, tal como já foi referido no capítulo anterior, teve como principal objectivo reduzir a dimensão do vector de alterações de modo a diminuir os tempos de transferência entre CPU e GPGPU.

O primeiro ponto relevante a analisar para o primeiro *kernel* diz respeito à utilização de instruções atómicas, dado ser a principal diferença comparativamente ao respectivo *kernel* na versão anterior.

Estas instruções necessitam de sincronismo no acesso à memória global, o qual é garantido através de *hardware*. Este sincronismo adiciona alguma latência ao tempo total de execução, como pode ser verificado na tabela 6.8.

	Blocks		Chess		Snake		Spiral	
Dimensão (X^3 bytes)	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s
100	0.01	92.03	0.01	99.25	0.03	37.31	0.01	71.29
200	0.05	160.20	0.05	171.19	0.17	46.54	0.07	119.43
300	0.16	173.31	0.14	186.82	0.50	54.39	0.21	130.36
400	0.36	180.06	0.34	187.75	1.16	55.12	0.47	136.36
500	0.69	181.39	0.64	194.43	2.29	54.53	0.86	145.17
600	1.18	182.67	1.13	191.22	3.28	65.82	1.45	149.45
700	1.88	182.32	1.84	186.90	4.51	76.13	2.33	147.29
800	2.93	174.94	2.67	191.44	7.52	68.11	3.58	143.01
900	4.11	177.27	4.25	171.50	9.04	80.61	5.18	140.69
1000	5.65	176.92	5.80	172.46	11.75	85.07	7.10	140.83
1100	7.92	167.98	7.60	175.18	16.06	82.87	9.72	136.87
1200	10.22	169.02	10.87	159.01	21.20	81.52	12.31	140.37

Tabela 6.8: Tempos de resposta da segunda versão do algoritmo *Object Identifier*.

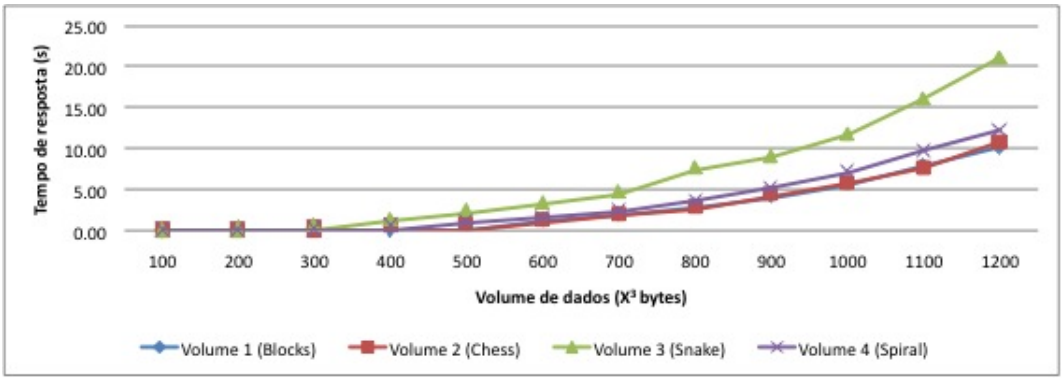


Figura 6.11: Tempos de resposta da segunda versão do algoritmo *Object Identifier*.

A convergência deste *kernel* é semelhante ao anterior, visto utilizar a memória local para efectuar a propagação. Embora o número de iterações seja igual ao anterior, o número de acessos à memória é menor, visto que após um *thread* mudar o seu voxel correspondente, termina.

Volume	Kernel 1 (s)	Kernel 2 (s)	Kernel 3 (s)	Optimização do vector de Alterações (s)
Blocks	0.18	0.10	0.03	0.01
Chess	0.09	0.07	0.03	0.02
Spiral	0.17	0.08	0.03	0.02
Snake	0.17	0.07	0.03	0.15

Tabela 6.9: Tempos de processamento da segunda versão do algoritmo *Object Identifier*.

O segundo *kernel* é semelhante ao anterior, com a particularidade de efectuar também a junção de subobjectos processados no mesmo multiprocessador.

Como se pode observar na tabela 6.9, estes tempos não diferem muito do *kernel* da versão anterior, visto que os *threads* que realizam este processamento adicional não efectuavam nenhum processamento na referida versão.

No que se refere à optimização do vector de alterações, este possui tempos bastante inferiores ao realizado na versão anterior, como descrito na tabela 6.9, dado processar vectores de menor dimensão.

Na tabela 6.10 é possível constatar a grande vantagem deste *kernel* através dos tempos

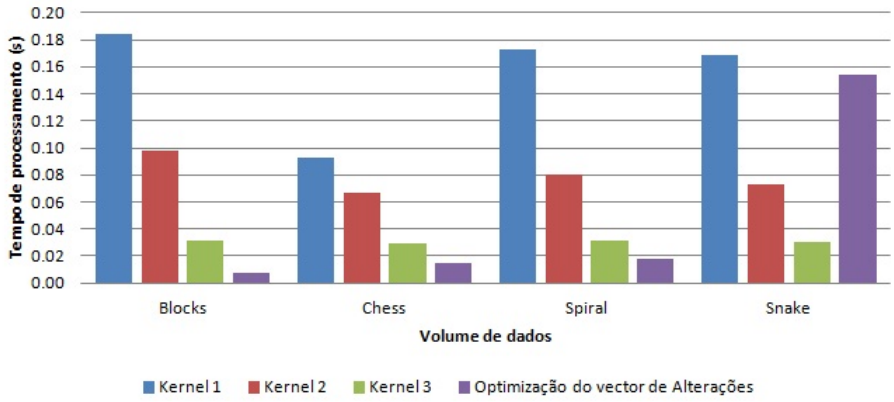


Figura 6.12: Tempos de processamento da segunda versão do algoritmo *Object Identifier*.

	Envio		Recepção	
Volume	Média (ms)	Desvio padrão (ms)	Média (ms)	Desvio padrão (ms)
100	0.02	0.02	0.02	0.02ss
200	0.11	0.14	0.11	0.13
300	0.37	0.45	0.35	0.41
400	0.86	1.07	0.81	1.02
500	1.67	2.09	1.59	1.99

Tabela 6.10: Tempos de transferência do vector de alterações da segunda versão do algoritmo *Object Identifier*.

de transferência do vector de alterações. Esta diferença deve-se à dimensão do vector de alterações ser bastante inferior à do vector da versão anterior.

Em suma, esta versão minimiza bastante o impacto do vector de alterações nos tempos de resposta do algoritmo, como se pode observar no gráfico da figura 6.14.

Através deste gráfico é possível verificar que as transferências ainda representam uma grande percentagem do tempo de resposta. Essa percentagem diz respeito à necessidade de transferir este vector diversas vezes, para que possa ser optimizado em CPU.

Ao observar o gráfico é possível verificar também que a optimização do vector de alterações também contribui para a redução dos tempos de execução.

Embora minimizado, o vector de alterações continua a ter um enorme impacto no tempo total de resposta, face à versão anterior. Logo, este continua a ser a principal limitação da solução.

6.5.3 Terceira versão

Quanto ao processamento, apenas é necessário analisar o segundo *kernel*, dado que o primeiro é igual ao anterior, e o terceiro é unificado com o segundo. Como é possível analisar na tabela 6.11 este algoritmo apresenta um tempo de resposta inferior ao da versão anterior.

Através da observação da tabela 6.12 é possível verificar que o tempo de execução

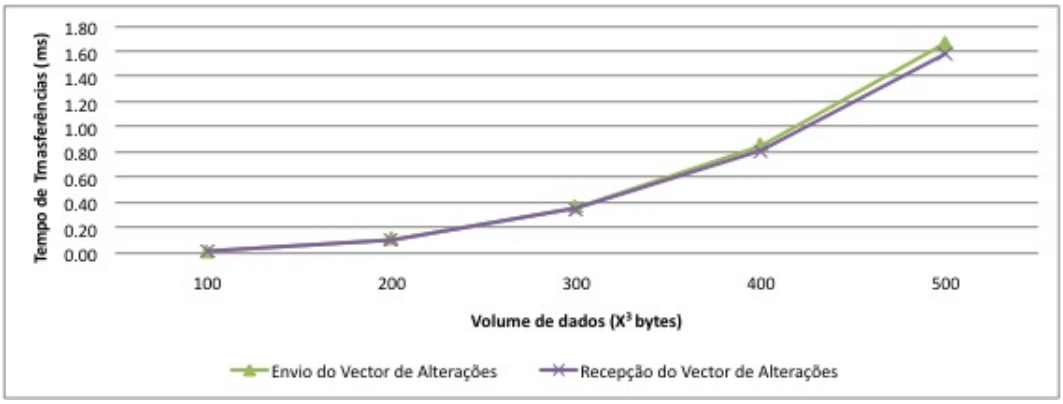


Figura 6.13: Tempos de transferência da segunda versão do algoritmo *Object Identifier*.

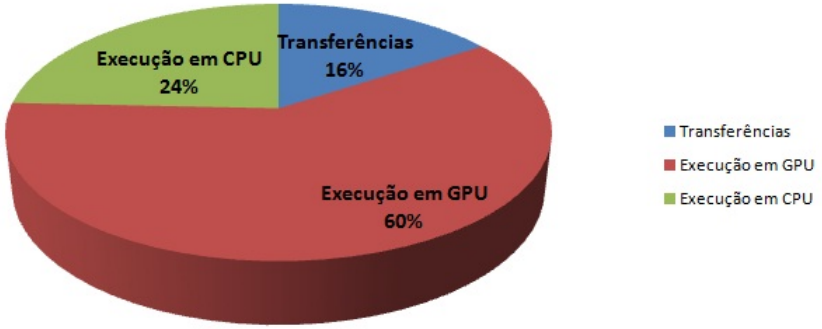


Figura 6.14: Decomposição do tempo total de resposta da segunda versão do algoritmo *Object Identifier*.

do segundo *kernel* é superior ao do anterior, visto efectuar a optimização do vector de alterações bem como a alteração dos voxels no bloco de dados. Embora seja superior, somando o segundo e o terceiro *kernel* da versão anterior, essa diferença torna-se quase insignificante.

É importante também referir que não são realizadas as transferências do vector de alterações, dada a sua optimização ser realizada em GPGPU, podendo assim permanecer no GPGPU durante todo o processamento.

Este aspecto implica que a diferença de tempos de resposta entre volumes distintos seja menor, visto que a principal diferença se encontra na dimensão do vector de alterações. Esta solução veio minimizar ainda mais o impacto do vector de alterações nos tempos de resposta do algoritmo, como é possível observar na figura 6.17.

As optimizações efectuadas vieram também reduzir a diferença dos tempos de resposta entre volumes distintos, como já foi referido anteriormente.

Esta solução apresenta-se limitada na fusão de identificadores no mesmo bloco de dados, os quais necessitam de diversas iterações para que sejam unificados. Essas iterações fazem com que seja necessário validar identificadores em memória global, o que origina tempos significativos.

	Blocks		Chess		Snake		Spiral	
Dimensão (X^3 bytes)	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s	Média (s)	MVoxels/s
100	0.01	94.78	0.01	107.08	0.02	64.51	0.01	85.09
200	0.05	157.43	0.04	178.34	0.08	97.36	0.06	133.47
300	0.16	170.44	0.13	200.76	0.26	103.37	0.18	146.29
400	0.37	174.28	0.31	207.02	0.62	103.82	0.44	144.40
500	0.71	175.61	0.61	205.16	1.21	103.57	0.83	150.29
600	1.25	172.62	1.13	190.66	2.07	104.54	1.48	145.89
700	1.98	173.18	1.75	196.53	3.25	105.51	2.27	150.94
800	2.98	172.06	2.66	192.42	4.84	105.88	3.54	144.66
900	4.20	173.69	4.39	166.24	6.81	107.09	4.99	146.13
1000	5.76	173.75	5.82	171.87	9.43	106.07	7.20	138.87
1100	7.83	169.97	7.70	172.89	12.73	104.57	9.31	142.99
1200	10.07	171.74	11.31	152.77	16.23	106.44	12.54	137.85

Tabela 6.11: Tempos de processamento da terceira versão do algoritmo *Object Identifier*.

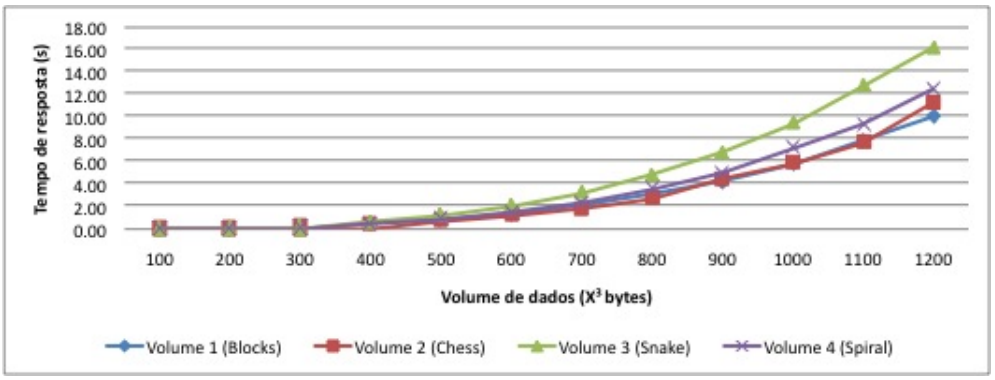


Figura 6.15: Tempos de resposta da terceira versão do algoritmo *Object Identifier*.

Em suma, neste algoritmo o tempo de resposta predomina no processamento em GPGPU, encontrando-se minimizado nas restantes operações necessárias. Para finalizar esta análise encontram-se na figura 6.18 os tempos de resposta desta solução em diversos GPGPUs. Na tabela é possível observar a superioridade do nVidia C2050, comparativamente aos restantes.

Um aspecto importante de referir diz respeito ao tempo de resposta da combinação nVidia C2050 e nVidia Quadro FX 3800. Esta solução produz tempos de resposta superiores à solução que utiliza apenas o nVidia C2050. Esta diferença acontece porque o acesso aos GPGPUs é efectuado através de um único *bus*. O *bus* quando está a ser utilizado faz com que todos os dispositivos não possam comunicar com o CPU. Dada a limitação das transferências do nVidia Quadro FX 3800, a utilização combinada com o nVidia C2050 produz tempos de resposta superiores.

Volume	Kernel 1 (s)	Kernel 2 (s)
Blocks	0.18	0.11
Chess	0.09	0.09
Spiral	0.17	0.10
Snake	0.18	0.10

Tabela 6.12: Tempos de processamento da terceira versão do algoritmo *Object Identifier*.

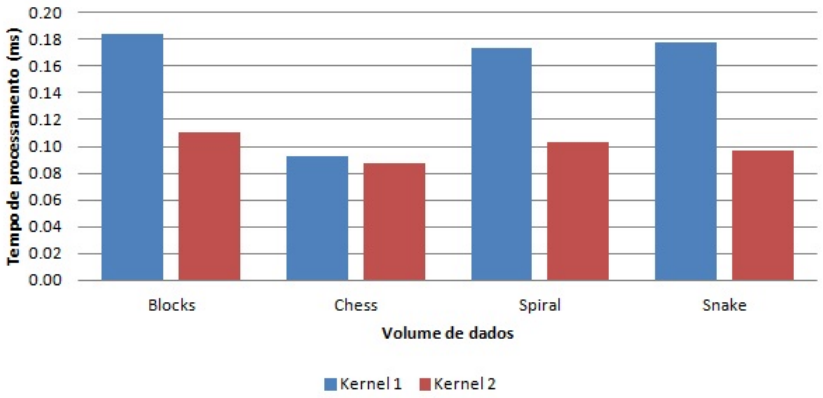


Figura 6.16: Tempos de processamento da terceira versão do algoritmo *Object Identifier*.

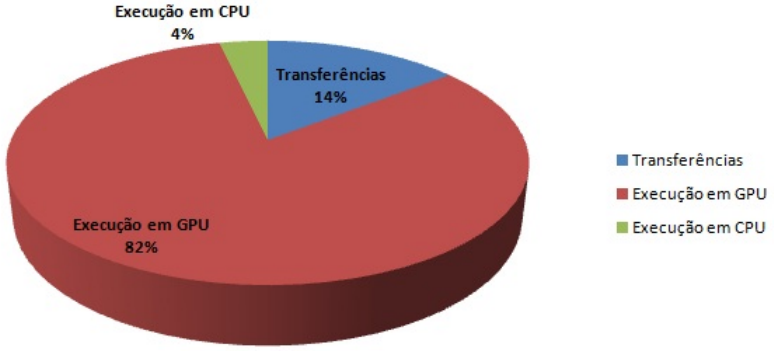


Figura 6.17: Decomposição do tempo total de resposta da terceira versão do algoritmo *Object Identifier*.

A utilização de diversos GPGPUs produz tempos de resposta superiores quando o processamento é predominante nas soluções. Este aspecto poderá aumentar o tempo de resposta desta solução após reduzir mais as transferências.

6.5.4 Resultados finais

Para terminar a análise do algoritmo *Object Identifier*, encontra-se na tabela 6.13 o tempo de resposta médio de cada versão implementada, bem como a quantidade de mega voxels que as mesmas conseguem processar por segundo. Na tabela referida é possível verificar as diferenças entre as versões sequenciais e paralelas, contudo, é possível verificar que a terceira versão do *Object Identifier* possui melhores resultados.

É possível observar que todas as versões apresentam tempos de resposta bastante inferiores aos das versões sequenciais. É possível verificar também que este algoritmo consegue processar imagens de elevada dimensão em tempos de resposta adequados para um ambiente interactivo.

Um outro aspecto relevante diz respeito à diferença entre os tempos de resposta, que aumentam consoante a dimensão do volume de dados. Este aspecto é bastante pertinente

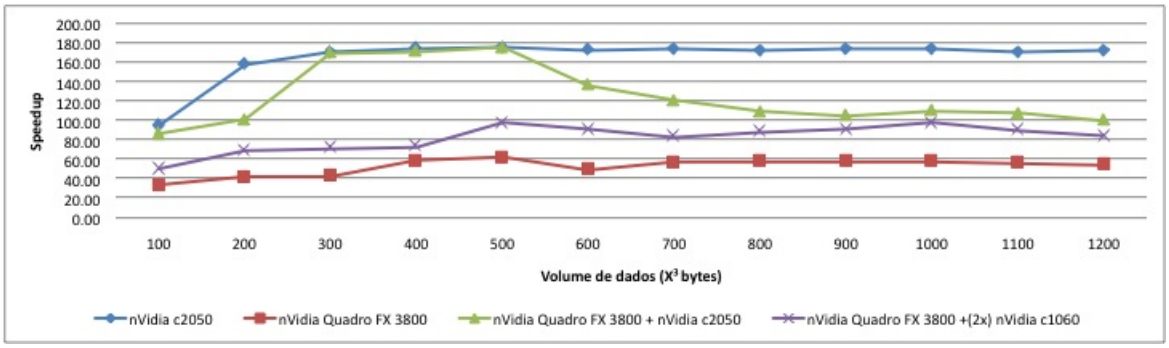


Figura 6.18: Tempos de resposta da terceira versão do algoritmo *Object Identifier* em diversos GPGPUs.

	One-Pass		Two-Pass		Object Identify v.1		Object Identify v.2		Object Identify v.3	
Dimensão (X^3 bytes)	Tempo médio (s)	Mvoxels/s	Tempo médio (s)	Mvoxels/s	Tempo médio (s)	Mvoxels/s	Tempo médio (s)	Mvoxels/s	Tempo médio (s)	Mvoxels/s
100	0.60	2.08	0.15	8.42	0.03	39.72	0.02	74.97	0.01	87.86
200	4.95	2.05	1.21	8.30	0.20	44.82	0.09	124.34	0.06	141.65
300	17.16	2.02	4.14	8.21	0.71	43.72	0.25	136.22	0.18	155.21
400	41.14	2.02	9.75	8.23	1.87	42.13	0.58	139.82	0.43	157.38
500	81.79	1.99	19.16	8.20	4.26	39.16	1.12	143.88	0.84	158.66
600	142.56	1.99	33.22	8.18	7.59	39.10	1.76	147.29	1.48	153.43
700	229.24	1.97	2028.10	0.30	13.42	39.39	2.64	148.16	2.31	156.54
800	343.42	1.97	> 3600.00	-	20.42	40.82	4.17	144.38	3.50	153.75
900	491.80	1.96	-	-	31.03	38.36	5.65	142.52	5.09	148.29
1000	678.02	1.95	-	-	46.63	39.32	7.58	143.82	7.05	147.64
1100	903.64	1.95	-	-	68.24	38.35	10.33	140.72	9.39	147.61
1200	3200.36	0.71	-	-	84.22	40.35	13.65	137.48	12.54	142.20

Tabela 6.13: Tempo médio de resposta das soluções desenvolvidas.

na análise da escalabilidade da solução, dado que quanto maior o volume de dados, maior o seu desempenho.

Dimensão (X^3 bytes)	Object Identify v.1 (s)	Object Identify v.2 (s)	Object Identify v.3 (s)
100	5.55	9.55	12.51
200	6.15	14.41	20.33
300	5.82	16.49	22.41
400	5.23	16.77	22.48
500	4.49	17.08	22.81
600	4.38	18.88	22.40
700	17.08	86.91	99.14
800	16.82	82.26	98.04
900	15.85	87.09	96.53
1000	14.54	89.49	96.16
1100	13.24	87.50	96.22
1200	38.00	234.46	255.30

Tabela 6.14: Tempo médio de resposta das versões do algoritmo *Object Identifier*.

Um outro ponto fundamental de apresentar diz respeito ao *speedup*. Tal como se pode observar no gráfico 6.19 da tabela 6.14, este aumenta consoante a dimensão do volume de dados, podendo-se assim se verificar a escalabilidade da solução. Este algoritmo, como já foi referido, foi desenvolvido de forma a permitir processar grandes volumes de dados em tempos de resposta reduzidos, de forma a permitir a análise de imagens tomográficas num ambiente interactivo.

Um outro aspecto importante a referir diz respeito ao volume de dados real referido

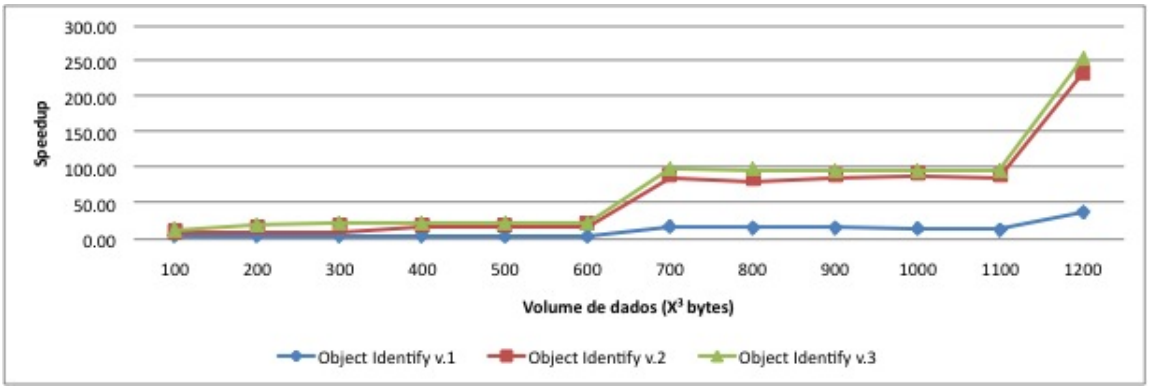


Figura 6.19: *Speedups* das diferentes versões do algoritmo *Object Identifier*.

inicialmente. Este possui as dimensões $200 \times 200 \times 200$, e obteve tempos de resposta de 320 ms nas versões sequenciais. Relativamente às versões do *Object Identifier*, estas tiveram tempos de resposta na ordem das dezenas de milisegundos. A primeira versão obteve um tempo de resposta médio de 98,03 ms. Relativamente à segunda versão, esta teve um tempo de resposta aproximadamente de 34,69 ms. Por fim, a terceira versão obteve um tempo de resposta superior à segunda visto que os dados são relativamente pequenos, tornado assim as transferências na segunda versão mais rápidas que os acessos à memória realizados pela terceira versão.

Em tema de conclusão a solução *Object Identifier* possui tempos de resposta bastante pequenos, adequados aos requisitos presentes na especificação do projecto. Através deste algoritmo foi possível identificar os objectos presentes em imagens tomográficas obtidas através de micro tomografia computacional.

Esta solução não se encontra limitada na sua concepção no que concerne à dimensão dos volumes de dados, assim, irá permitir melhores tempos de resposta e o aumento da referida dimensão, em função da evolução do *hardware* dos GPGPUs, dado estes estarem a ser cada vez mais utilizados para computação genérica. Este aspecto é bastante importante, uma vez que existem diversos avanços na captação de imagens através de micro tomografia, permitindo assim que cientistas de engenharia de materiais possam analisar de forma mais compacta os objectos de maiores dimensões. Um possível exemplo é a indústria aeronáutica, que cada vez mais aposta no uso de materiais compósitos para a concepção de aviões, sendo assim uma mais-valia a possibilidade de analisar materiais de maiores dimensões num reduzido espaço de tempo.

7

Conclusão e trabalho futuro

O presente capítulo tem como objectivo apresentar as conclusões obtidas com a dissertação, bem como algumas sugestões para possíveis trabalhos futuros relacionados com os algoritmos desenvolvidos.

7.1 Balanço do trabalho

Através do estudo realizado na presente dissertação, foi possível conceber e implementar um algoritmo que permite a identificação de objectos presentes em imagens tomográficas de grandes dimensões em ambientes interactivos.

Esta solução irá permitir a investigadores da área de engenharia de materiais a identificação de todos os componentes presentes nos objectos em estudo, de forma a poder distinguir cada um dos elementos que constituem os objectos. Através desta análise, é possível efectuar a distinção entre o material base e os reforços, que dão origem aos materiais compósitos. Esta solução irá posteriormente integrar um módulo do ambiente SCIRun, para que possa ser utilizado através de uma interface de alto nível, de forma a ser combinado com outros módulos segundo um *workflow*. Assim, os utilizadores poderão desenvolver programas a um mais alto nível sem que tenha qualquer conhecimento dos detalhes de implementação dos mesmos, como ilustrado na figura 7.1.

No que concerne ao desempenho do algoritmo, este utiliza paralelismo em CPU e GPGPU de forma a produzir um menor tempo de resposta. A solução inicia analisando os recursos disponíveis de forma a se auto-configurar para poder tirar o maior partido dos mesmos. Embora o algoritmo se ajuste ao *hardware*, este permite portabilidade, visto que esse ajuste é efectuado apenas durante a execução.

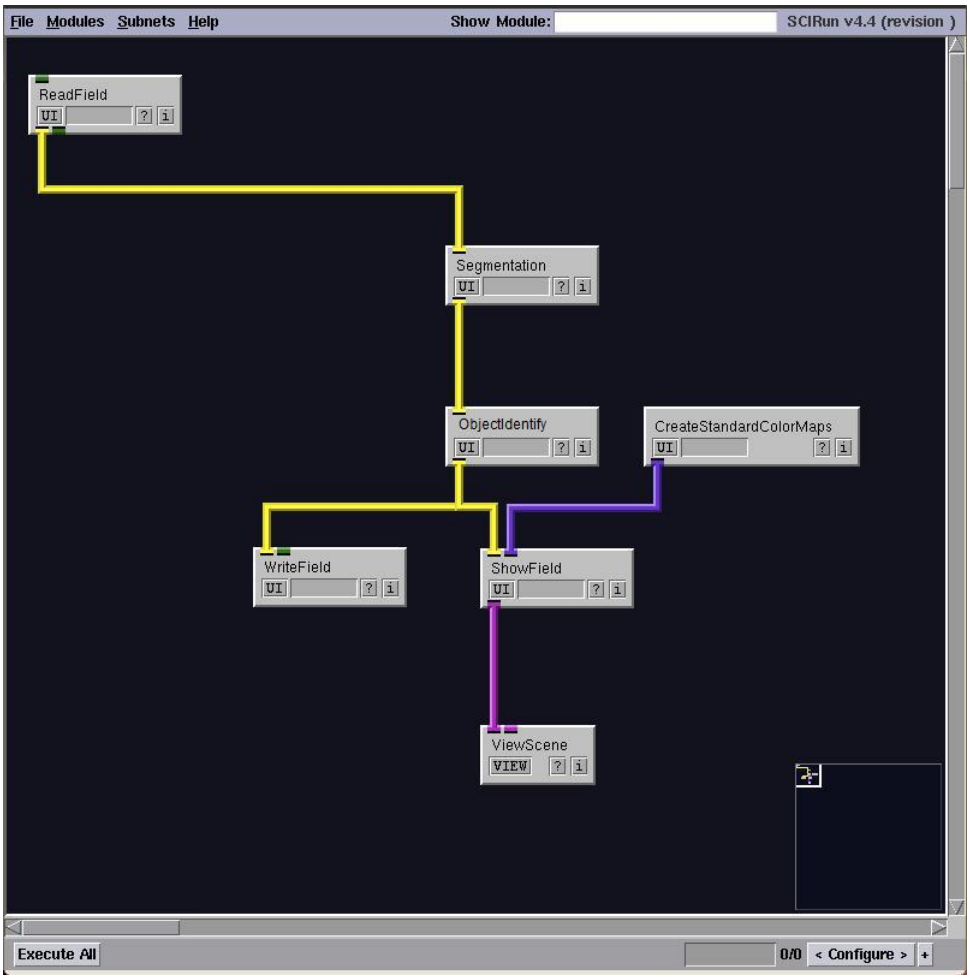


Figura 7.1: Ambiente de desenvolvimento SCIRun.

É importante referir que existem algumas limitações inerentes ao protótipo desenvolvido, tais como o facto de este apenas suportar imagens binárias e apenas efectuar a divisão dos volumes de dados segundo o eixo em z.

Além deste projecto, este tipo de soluções apresenta diversas aplicações, como na área da saúde, para a detecção de massas em determinadas áreas do corpo humano, bem como na identificação de padrões em imagens.

Neste estudo foi possível também analisar detalhadamente a programação através dos GPGPUs. Nessa análise foi possível detectar as limitações bem como a enorme capacidade de processamento dos mesmos.

Em relação às vantagens deste tipo de programação, estas residem essencialmente no elevado poder computacional que permite obter *speedups* elevados através da quantidade de unidades de processamento disponíveis.

Foi possível também verificar a portabilidade do código obtida através da compilação em *runtime* efectuada pela plataforma OpenCL.

As limitações, por sua vez, estão associadas principalmente à dificuldade da programação e aos meios de comunicação com os GPGPUs.

Este tipo de programação situa-se a um baixo nível onde apenas existem instruções básicas e algumas bibliotecas específicas.

O paradigma de programação associado a estas linguagens faz com que seja necessário rescrever totalmente um algoritmo sequencial, para que possa ser utilizado em GPGPUs.

Após a implementação de um dado algoritmo, é extremamente importante conhecer o modo de funcionamento do *hardware* para que o algoritmo possa ser optimizado.

No término da realização desta dissertação é possível concluir que o algoritmo desenvolvido constitui uma mais-valia para o projecto no âmbito do qual este foi desenvolvido, pois permitiu um avanço no estudo da identificação de objectos em imagens tomográficas. Ou seja, permitiu uma evolução de entre a classe de algoritmos *Connected-component labeling*.

Além das componentes técnicas que foi possível adquirir, desenvolver e até mesmo aperfeiçoar, com a realização desta dissertação foi possível desenvolver competências na área da investigação e aumentar a minha admiração por a referida área. A referir ainda que esta experiência em investigação foi bastante importante para a minha vida futura enquanto Engenheiro Informático, abrindo-me novos caminhos para um futuro promissor.

7.2 Trabalho Futuro

Após a análise detalhada à solução foi possível identificar alguns temas que podem ser explorados futuramente de forma a melhorar o desempenho da solução. Além desses temas existe também a integração do algoritmo no ambiente SCIRun, dado este ser o ambiente definido pelo âmbito do projecto de investigação, no qual esta dissertação está inserida.

Um tema interessante de estudar diz respeito à fusão de identificadores, dado esta ser realizada através de diversas iterações do segundo *kernel*. Uma possível melhoria a esse aspecto seria desenvolver uma estrutura de dados *Disjoint-Set Forests* em GPGPU de forma a apenas necessitar de uma única invocação ao segundo *kernel*.

Uma outra possível solução para este aspecto consiste em definir uma união hierárquica através da memória local eliminando assim as latências associadas aos acessos à memória global.

Um outro tema relevante consiste em permitir a detecção de objectos consoante a sua cor inicial, não se encontrando limitado a imagens binárias. Este aspecto poderia ser importante para detectar regiões ambíguas, ou seja, que através da micro tomografia não foi possível definir se são objectos ou matriz.

Dada a decomposição dos blocos ser realizada segundo o eixo dos z, caso a imagem seja em 2D, não pode exceder a memória do GPGPU. Um possível tema de estudo seria efectuar divisões eficientes em x ou em y, dado que estas não resultam em blocos contíguos em memória.

Um outro possível tema de estudo futuro consiste em efectuar todo o processamento em GPGPU. Embora os blocos de dados não possam ser colocados em simultâneo na memória do GPGPU, actualmente existe uma técnica designada de *out-of-core*, que consiste em utilizar a memória RAM e também o disco para criar uma extensão à memória do GPGPU, como descrito no artigo [49].

Bibliografia

[1] HM Alnuweiri and VK Prasanna. Parallel architectures and algorithms for image component labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(10), 1992.

[2] AMD. R700-family instruction set architecture. *AMD whitepaper*, March 2009.

[3] AMD. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. *AMD White Paper*, March 2010.

[4] A. Arevalo, R. M. Matinata, E. Pandian, M. Peri, K. Ruby, F. Thomas, and C. Almond. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM Redbooks publication, 2008.

[5] A.R. Brodtkorb, C. Dyken, T.R. Hagen, J.M. Hjelmervik, and O.O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, pages 777–786. ACM, 2004.

[7] T. Cadavez. Análise de imagens tomográficas: visualização e paralelização de processamento. Master's thesis, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2008.

[8] C. Cascaval, S. Chatterjee, H. Franke, K.J. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54(5):5, 2010.

[9] T.H. Cormen. *Introduction to algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001.

[10] P.N. Glaskowsky. NVIDIAs Fermi: The First Complete GPU Computing Architecture. *nVidia whitepaper*, pages 1–26, 2009.

[11] M. Gschwind, D. Erb, S. Manning, and M. Nutter. An open source environment for Cell broadband engine system software. *Computer*, 40(6):37–47, 2007.

[12] Cyrus Harrison, Hank Childs, and Kelly P. Gaither. Data-parallel mesh connected components labeling and analysis. In Torsten Kuhlen, Renato Pajarola, and Kun Zhou, editors, *EGPGV*, pages 131–140. Eurographics Association, 2011.

[13] S. Hauck and A. DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2008.

[14] KA Hawick, A. Leist, and DP Playne. Parallel graph component labelling with gpus and cuda. *Parallel Computing*, 2010.

[15] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[16] W.W. Hwu. *Gpu Computing Gems*. Applications of GPU Computing Series. Elsevier Science & Technology, 2011.

[17] IBM. IBM BladeCenter QS22. available at: http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/ (visited 2011-01-07).

[18] Intel. Intel Sandy Bridge . available at: http://www.intel.com/technology/architecture-silicon/2ndgen/index.htm (visited 2011-01-26), 2011.

[19] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010.

[20] D. Kirk, W.M.W. Hwu, and W. Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann. Morgan Kaufmann Publishers, 2010.

[21] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, et al. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA Information Processing Techniques Office, Washington, DC*, page 278, 2008.

[22] G. Lupton and D. Thulin. Accelerating HPC Using GPU's. *HP whitepaper*, June 2008.

[23] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Transactions on Graphics (TOG)*, 23(3):787–795, 2004.

[24] M.D. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *GSPx Multicore Applications Conference*, 2006.

[25] M. Miller, C. Hansen, and C. Johnson. Simulation steering with SCIRun in a distributed environment. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, pages 366–376, 1998.

[26] G. E. Moore. Cramming more components onto integrated circuits Electron. *Electronics Magazine*, 38:114–117, April 1965.

[27] A. Munshi et al. The OpenCL specification version 1.1. *Khronos OpenCL Working Group*, 2010.

[28] NVIDIA. CUDA CUBLAS Library. *nVidia whitepaper*, June 2007.

[29] NVIDIA. Compute Unified Device Architecture, Programming Guide, version 2.0. *nVidia whitepaper*, 2008.

[30] NVIDIA. NVIDIA GeForce GTX 200 GPU architectural overview. *nVidia whitepaper*, May 2008.

[31] NVIDIA. CUDA CUFFT Library. *nVidia whitepaper*, February 2010.

[32] nVidia. Opencl best practices guide. available at: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Best_Practices_Guide.pdf (visited 2011-08-20), 2011.

[33] P. Paiva. Paralelização de operações de processamento de dados tomográficas usando o processador cell be. Master's thesis, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2011.

[34] M. Papakipos. The PeakStream platform: High-Productivity software development for multi-core processors. available at: http://download.microsoft.com/download/d/f/6/df6accd5-4bf2-4984-8285-f4f23b7b1f37/WinHEC2007_PeakStream.doc (visited 2011-01-26), 2007.

[35] Steven G. Parker and Christopher R. Johnson. SCIRun: a scientific programming environment for computational steering. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[36] P. Quaresma. Processamento paralelo aplicado a um problema de engenharia de materiais. Master's thesis, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2007.

[37] E. Rhodes. *ASIC Basics: Black & White Edition*. Lulu.com, 2008.

[38] R.J. Rost. *OpenGL (R) Shading Language*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004.

[39] M. Scarpino. *Programming the Cell processor: for games, graphics, and computation*. Prentice Hall, 2009.

[40] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[41] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106. Eurographics Association, 2007.

[42] K. Skaugen. Petascale to ExascaleExtending ExascaleExtending Intel's HPC Commitment. available at: `http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf` (visited 2011-01-26), 2010.

[43] J. E. Smith, N.P. Jouppi, and S. Palacharla. Complexity-effective superscalar processors. In *ISCA*, page 206. ACM Press, 1997.

[44] P. Sundararajan. High Performance Computing Using FPGAs. *Xilinx whitepaper*, September 2010.

[45] K. Suzuki, I. Horiba, and N. Sugie. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89(1):1–23, 2003.

[46] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335. ACM, 2006.

[47] Ryan Taylor and Xiaoming Li. Software-based branch predication for amd gpus. *SIGARCH Comput. Archit. News*, 38:66–72, January 2011.

[48] A. Velhinho. *Fundição centrífuga de compósitos alumínio/sic com gradiente funcional de propriedades: Processamento e caracterização*. PhD thesis, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2003.

[49] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33:209–271, June 2001.

[50] R. Woods and J. McAllister. *FPGA-based implementation of signal processing systems*. John Wiley & Sons, 2008.

[51] K. Wu, E. Otoo, and K. Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Analysis & Applications*, 12(2):117–135, 2009.