Identificação de Objetos em Imagens tomográficas através de GPGPUs

Bruno Preto Fernando Birra Pedro Medeiros
CITI, Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
bpreto@gmail.com, {fpb,pdm}@fct.unl.pt

Resumo

Neste artigo encontra-se a implementação e a avaliação de um novo algoritmo híbrido (CPU-GPU) para a identificação de estruturas conexas em volumes de dados tridimensionais. Este algoritmo explora o paralelismo tanto ao nível do CPU como dos GPGPUs. Contudo, o processamento é maioritariamente realizado em GPUs.
A motivação para este algoritmo resulta da sua utilização no contexto dum sistema mais vasto e orientado para a resolução de problemas de caracterização estrutural de materiais através de tomografia. Através da solução presente no artigo, será possível analisar a localização e a morfologia dos objetos presentes nos volumes tridimensionais.
A grande vantagem da utilização deste algoritmo deve-se a permitir tempos de execução bastante baixos, bem como à capacidade de processar grandes volumes de dados. Neste caso, as soluções parciais são calculadas de forma independente numa partição dos dados iniciais, sendo posteriormente integradas pelo CPU, usando uma abordagem que permite, ainda assim, explorar o paralelismo oferecido pelos múltiplos cores CPU.

Palavras-Chave

Algoritmos de processamento de imagens a três dimensões; Paralelização de aplicações; OpenCL; Tomografia; GPGPU.

1 INTRODUÇÃO

Os algoritmos do tipo *union—find* lidam com o problema de encontrar conjuntos disjuntos de objetos e as suas aplicações abrangem muitas áreas, tais como as redes de computadores, visão por computador e tomografia computorizada. Os objetos a identificar consistem em elementos conexos e as operações elementares neste tipo de algoritmos estão na base da sua nomenclatura. A operação *union* funde dois conjuntos disjuntos de elementos, interligando-os, enquanto a operação *find* descobre se dois dados elementos estão interligados.

Quando os objetos a identificar correspondem a pixéis de uma imagem ou amostras de um volume, sendo os dados de entrada substituídos pela etiqueta do objeto a que pertencem, os algoritmos designam-se por *Connected-component labeling*. No caso das imagens 2D ou 3D, a conetividade de cada elemento (pixel ou voxel) pode ser definida ao longo das faces, das arestas e dos vértices.

A Figura 1 apresenta um possível resultado aplicado a uma imagem 2D, sendo a conetividade dos pixéis definida ao longo das arestas.

Numa fase inicial, os dados provenientes das amostras são filtrados, sendo submetidos, na maior parte dos casos a processos de remoção de ruído, erosão, dilatação, etc. Um

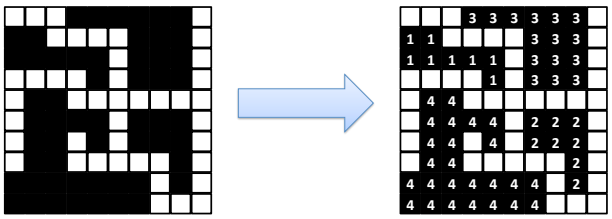


Figura 1: Algoritmo *Connected-component labeling*.

dos processos mais importantes nesse pré-processamento, é o da segmentação. É após a segmentação que se efetua a identificação de aglomerados conexos, para os quais cada amostra possui o mesmo valor após a segmentação.

Após a etiquetagem de cada objeto com um identificador único torna-se possível obter todos os pixéis ou voxéis do mesmo, para análise posterior, permitindo assim, não só visualizar esse mesmo objeto de forma isolada, mas também obter informação que permita ajudar à sua caracterização (área, volume, massa, fatores de forma, eixos prinicipais, etc.).

Este trabalho incide num módulo dum sistema mais vasto do tipo PSE (*Problem Solver Environment*), desenvolvido no topo do sistema de visualização de dados ci-

entíficos SCIRun [SCI12]. A componente em causa efetua a identificação de objetos em volumes de dados respeitantes a amostras de materiais compósitos obtidos por processos de microtomografia, contribuindo assim para uma posterior caracterização das amostras analisadas.

2 TRABALHO RELACIONADO

Atualmente, existem diversas soluções para a identificação de objetos, as quais podem ser agrupadas em dois grupos, as soluções puramente sequenciais, descritas em [Wu 09], e as soluções que exploram o paralelismo, utilizando *clusters* [Harrison 11], e GPGPUs [Hawick 10] [OS11].

2.1 Soluções sequenciais

As soluções que processam sequencialmente os volumes de dados, apresentadas em [Wu 09], 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 efetuam 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. [Suzuki 03], e efetua 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 efetuam 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 voxéis. Para efetuar a junção de identificadores, é utilizada a estrutura de dados *Disjoint-Set Forests*.

Estas soluções têm a grande desvantagem de proporcionarem tempos de resposta bastante elevados, e de apenas poderem ser aplicadas a conjuntos de dados limitados.

2.2 Soluções paralelas

Relativamente às soluções que exploram o paralelismo, estas permitem reduzir o tempo de resposta, decompondo o processamento em diversas tarefas, que podem ser processadas em paralelo por múltiplas unidades de processamento. O grande problema desta abordagem, reside na complexidade da divisão do processamento. Este problema não é passível de ser resolvido sem a necessidade de sincronismo, dado que os valores a atribuir a uma dada região, necessitam de estar coerentes com a sua vizinhança. No que se refere às soluções utilizando *clusters*, destaca-se a solução descrita em [Harrison 11], 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. 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 interatividade limitada.

No que diz respeito à solução CCL [OS11], esta consiste em efetuar todo o processamento no GPGPU, tirando partido da memória e da localização da informação.

Este algoritmo apresenta enormes vantagens face aos restantes, no que concerne à redução dos tempos de execução. Contudo, este apenas efetua o processamento de imagens bidimensionais, não permitindo assim a deteção dos objetos tridimensionais. Além disso, apenas processa imagens de pequenas dimensões.

De seguida, 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 interativo.

3 DESCRIÇÃO DA SOLUÇÃO

O presente algoritmo foi desenvolvido para ser executado em máquinas com um CPU, contendo um ou vários núcleos, auxiliados por um conjunto de um ou mais GPGPUs, sendo nestes últimos que se realiza a maioria do processamento.

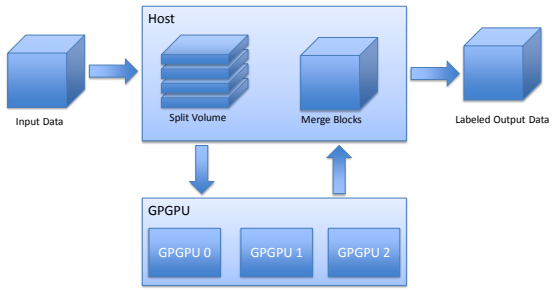


Figura 2: Arquitetura da solução.

Tal como se pode observar na Figura 2, esta solução recebe um volume de dados. o qual vai sendo dividido em vários blocos, posteriormente enviados e processados por um GPGPU duma pool de GPGPUs. Após o seu processamento, são enviados para o CPU, onde os resultados são consolidados no volume final de saída.

Este algoritmo explora o paralelismo oferecido pelos GPGPUs, visando a identificação de objetos em volumes de grande dimensão (aprox. 1 Giga amostras), em tempos de execução que permitam alguma interatividade.

Tal como é possível observar na Figura 3, este algoritmo encontra-se dividido nas seguintes quatro fases:

- decomposição do volume de dados em blocos;
- atribuição de identificadores iniciais aos voxéis de cada bloco (*kernel 1*);
- fusão de identificadores em subobjetos (*kernel 2*);

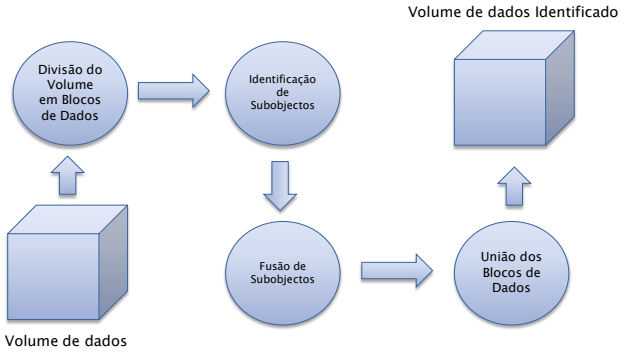


Figura 3: Fases do Algoritmo.

- fusão dos subobjetos entre blocos distintos,

as quais serão analisadas de seguida.

3.1 Decomposição do volume em blocos

Dada a limitação no tamanho da memória dos GPGPUs, o algoritmo começa por dividir o conjunto de dados em blocos, que serão posteriormente processados isoladamente pelos GPGPUs. Cada GPGPU vai requerendo blocos assincronamente, duma fila de trabalho, à medida que termina o processamento do bloco a ele anteriormente atribuído. Esta estratégia permite maximizar a utilização dos dispositivos, visto que estes não ficam inativos, à espera que outras unidades terminem o seu processamento.

3.2 Atribuição inicial de identificadores

O objetivo desta fase é o de proceder a uma identificação prévia de possíveis objetos, focando a atenção em extremos dos mesmos. Por exemplo, efectuando uma atribuição inicial de identificadores aos voxéis que se situam nas extremidades dos objetos e propagando de seguida esses identificadores pelos vóxeis vizinhos, enquanto tal for possível.

Após a decomposição do volume de dados em blocos, cada voxel é mapeado num *thread*, que começa por poder atribuir identificadores únicos (recorrendo a contadores implementados com instruções atómicas), a um conjunto reduzido de voxéis, segundo a sua posição relativa nos objetos a que pertencem (p.ex. nos seus cantos). Esse conjunto de voxéis é calculado através da sua posição, caso um voxel possua o vizinho acima e à esquerda com a cor branca (vazio), então pede-se um identificador único novo para a sua cor, e o *thread* termina.

Quanto aos restantes *threads*, estes entram em estado de espera ativa, no qual vão validando os identificadores dos vizinhos. Assim que um vizinho possui um identificador válido, ou seja, diferente de branco ou preto, o voxel adota esse valor e termina. Esta operação tem como vantagem não precisar de memória auxiliar para a propagação dos identificadores, como os algoritmos sequencias, bem como permitir um elevado nível de paralelismo, visto que é gerado um *thread* por cada voxel.

Para minimizar o número de acessos à memória global, a propagação de identificadores é apenas realizada em memória local, tornado o processamento mais rápido.

O facto de se utilizar a memória local para efetuar a propagação dos identificadores, implica que sejam necessárias menos iterações, visto que esta tem dimensões bastante menores (usualmente utilizamos um layout de 8x8x8), que a memória global que contém as dimensões do bloco a processar. Como se pode observar na Figura 4, a propagação de um dado identificador é apenas realizada dentro de cada região local, ao invés de todo o bloco, encurtando assim o tempo da sua propagação.

Sendo o endereçamento dos *threads* igual ao dos voxéis na matriz, os acessos realizados à memória são coalescentes, tirando assim um maior partido da arquitetura dos dispositivos, no que à largura de banda da memória diz respeito.

3.3 Fusão de subobjetos

Como ilustrado na Figura 4, após a execução do primeiro *kernel*, todos os voxéis do volume de dados possuem regiões com identificadores atribuídos garantindo que, objetos disjuntos não partilham identificadores comuns e, para cada objeto há um excesso de identificadores, ainda a fundir num só. A tarefa dessa fusão, ao nível de cada bloco dos dados iniciais é da responsabilidade do *kernel 2*.

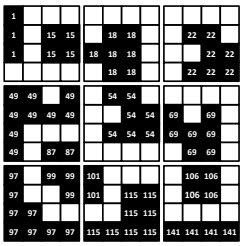


Figura 4: Volume de dados após a execução do primeiro *kernel*.

Este *kernel* é composto 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 (vetor de alterações), para que assim todos os voxéis com o valor antigo sejam alterados posteriormente.

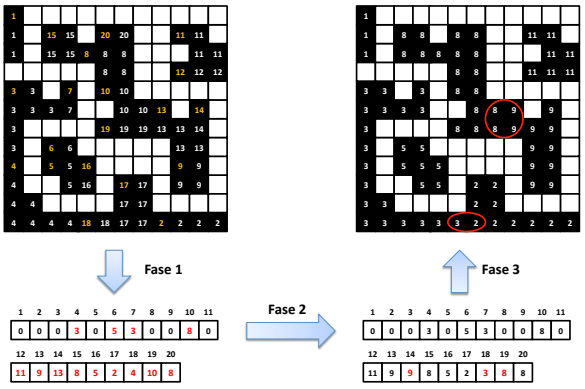
A segunda etapa diz respeito à otimização do vetor de alterações, a qual analisa a transitividade entre identificadores, de forma a encontrar o identificador raíz. Esta operação consiste em cada *thread* consultar a posição do vetor de alterações correspondente, e caso esta esteja marcada para ser modificada por um novo identificador, o *thread* consulta a posição correspondente a essa identificador. Caso essa nova posição também se encontre marcada, o *thread* volta a repetir a operação, até encontrar a raíz da transitividade, correspondendo a uma entrada no vetor apontando para ela própria. Assim, após esta operação, todos os voxéis são modificados a sua raíz da árvore de equivalências.

A aplicação desta técnica pode ser observada na Figura 5,

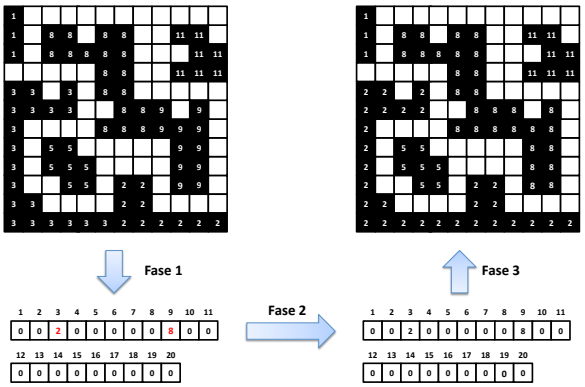
onde o identificador 14 está marcado para ser substituído pelo 13, encontrando-se este marcado para ser substituído pelo 9. Assim, a otimização realizada modifica para 9 o valor pelo qual o identificador 14 irá ser substituído. Caso contrário, seria necessária uma nova iteração para atribuir o identificador 9 a esta região.

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

As fases supracitadas são repetidas até que todos os subobjetos do bloco sejam unidos, ou seja, até que não exista nenhuma modificação a efetuar no vetor das alterações na primeira fase, tal como consta na Figura 5.



(a) Primeira iteração



(b) Segunda iteração

Figura 5: Exemplo de fusão de objetos em GPGPU.

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 objetos presentes em blocos distintos. Para realizar esta operação é utilizado um grafo, onde cada objeto 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.

Esta operação é realizada em CPU, utilizando um grafo, onde um nó representa um identificador e as arestas a relação entre identificadores. Para que a geração do grafo seja realizada de forma eficiente, 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 objeto, é realizada uma pesquisa em profundidade no grafo, que permite obter, de forma eficiente, listas de identificadores que relacionam os subobjetos do mesmo objeto.

5 OTIMIZAÇÕES

De forma a otimizar a solução foram implementadas alguma técnicas, adequadas à arquitetura utilizada [Nvidia 11], as quais visam maximizar a utilização dos GPGPUs, proporcionando tempos de resposta mais reduzidos.

Uma das otimizações realizadas diz respeito às transferências dos blocos de dados entre CPU e GPU. Estas, para que sejam realizadas de forma mais rápida, requerem que os *buffers* em memória RAM sejam criados com o tipo *page-locked*, permanecendo fixos em RAM durante a sua utilização. Esta característica permite que se consigam atingir taxas de transferência na ordem dos 5 GBps.

Uma outra otimização bastante relevante, diz respeito ao paralelismo entre transferências e execuções para o mesmo dispositivo. Para que esta situação seja possível, para além dos *buffers* em RAM terem que ser do tipo *page-locked*, devem ainda ser utilizadas duas filas de comandos (*queues*) por dispositivo.

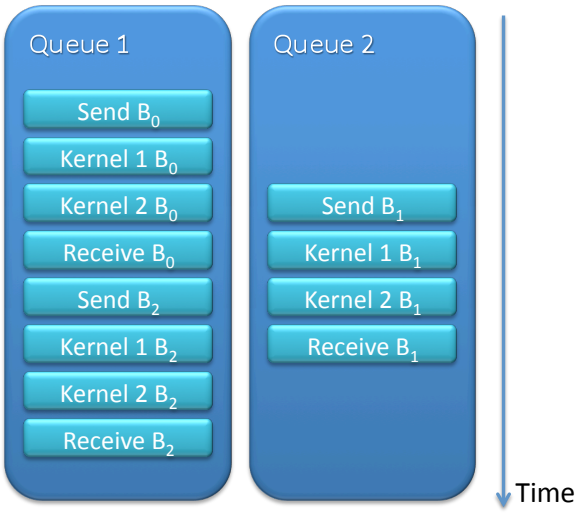


Figura 6: Ordem de envio dos comandos para o GPGPU.

Tal como se pode observar na Figura 6 , a solução desenvolvida atribui duas filas de comandos para um mesmo dispositivo, sendo as operações de transferência e execução dsitribuídas pelas filas de modo a permitir a execução de *kernels* em simultâneo com as transferências de dados. A partição dos comandos pelas duas filas depende do problema em concreto mas, através desta técnica, foi possível

reduzir avultadamente o tempo de execução, conseguindo ocultar a maior parte das transferências de memória com o dispositivo, ficando estas escondidas pelas execuções dos *kernels*.

Além das técnicas supracitadas, foram realizadas as seguintes otimizações, já naturais: os *threads* encontram-se agrupados em múltiplos dos *warps*, de forma a maximizar o número de *threads* ativos; os dados são mantidos na memória do GPU durante todas as fases, de forma a reduzir o número de transferências; os *buffers* são reutilizados, de forma a evitar operações de alocação de memória.

6 ANÁLISE DE RESULTADOS

Para analisar a solução foram realizados testes a diversos níveis, de forma a extrair as suas vantagens e limitações. A avaliação consistiu em analisar os tempos de resposta, assim como alguns aspetos inerentes à implementação. De seguida, encontra-se a descrição: do *hardware* e *software* utilizados; dos volumes de dados utilizados; dos resultados obtidos.

6.1 Hardware e Software utilizados

Para a realização dos testes, foi utilizada uma estação de trabalho que possui como *hardware* base: um processador Xeon E5504 (4-core); 12 Gbytes RAM; um GPU nVidia Quadro FX 3800 para a visualização e um nVidia c2050 (Fermi) para o processamento. Este acelerador, 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, efetuando um consumo de 238 W.

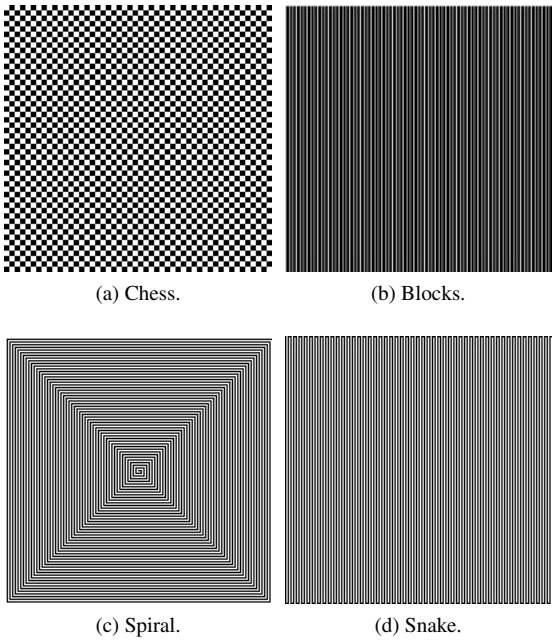
Relativamente ao software utilizado, este consistiu na plataforma OpenCL sobre o sistema operativo Linux na distribuição Ubuntu.

6.2 Volumes de dados utilizados

Os volumes de dados utilizados, ilustrados na Figura 7, consistem em dados fictícios, que têm com principal função simular cenários extremos, mais exigentes do que os volumes reais, obtidos através de micro tomografia. O primeiro volume de dados, designado de *chess* consiste num xadrez tridimensional, que tem como finalidade criar um elevado número de objetos de pequenas dimensões. O segundo volume de dados, designado de *blocks*, consiste em diversos paralelepípedos que se estendem em toda a dimensão z, analisando assim o comportamento dos algoritmos em volumes com bastantes objetos de grandes dimensões. O terceiro volume, designado de *spiral*, consiste num único objeto que o ocupa por completo, sendo a sua forma em espiral, de modo a maximizar a sua dimensão. Relativamente ao quarto volume de dados, designado de *snake*, consiste num único objeto, que percorre todo o volume de dados, possuindo apenas a espessura de um voxel.

6.3 Resultados obtidos

No gráfico ilustrado na Figura 8 é possível visualizar o tempo de transferência dos blocos de dados em função do volume de dados e do tipo de transferência. Nesta análise



(a) Chess. (b) Blocks.

(c) Spiral. (d) Snake.

Figura 7: Volumes de dados.

quantitativa pode-se observar o tempo despendido no envio e na receção dos dados, podendo ver-se a sua variação em função da dimensão dos dados.

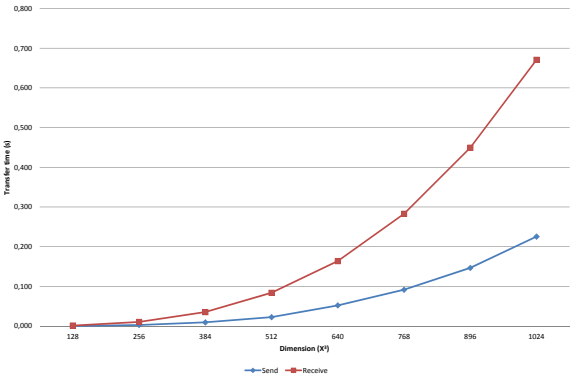


Figura 8: Tempos de transferência entre CPU e GPGPU.

Os tempos associados à emissão dos blocos para o GPGPU são inferiores à receção, visto que o volume de dados enviado é 4x menor. Este utiliza, para cada voxel, um byte, enquanto que o volume de dados recebido utiliza quatro bytes para representar cada identificador.

Relativamente ao processamento em GPGPU, através do gráfico da Figura 9, é possível observar o tempo de execução dos dois *kernels* em função da dimensão do volume de dados *chess*, ilustrado na figura 7a. Através deste gráfico é possível visualizar diversos aspetos. O primeiro diz respeito às diferenças entre o primeiro e o segundo *kernel*. O primeiro *kernel* começa por apresentar tempos superiores aos do segundo *kernel*, devido à complexidade do mesmo. Este primeiro *kernel*, como já foi referido, coloca diversos threads em espera ativa até que os identificadores

sejam propagados pelos vizinhos, ao contrário do segundo *kernel*, em que cada thread executa um processamento de complexidade constante.

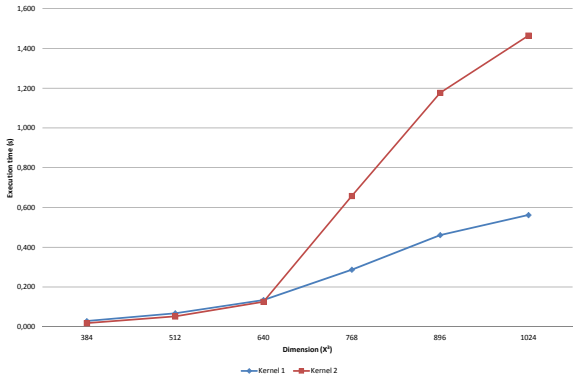


Figura 9: Tempos de processamento em GPGPU.

Através deste gráfico é possível verificar que quando a dimensão do volume de dados é superior a 640x640x640, o segundo *kernel* começa a exibir tempos superiores aos do primeiro. Esse facto deve-se à quantidade de iterações que é necessário executar para o segundo *kernel*, de modo a que os subobjetos sejam todos fundidos.

Um outro aspecto importante de analisar, é a fusão de blocos em CPU. Esta operação apenas é necessária quando o volume de dados é superior à memória do GPGPU. Contudo, o seu tempo de processamento é bastante baixo, visto que é uma operação realizada em paralelo, e apenas são validadas as fronteiras de cada bloco.

Dada a limitação da memória do GPGPU utilizado, bem como a limitação do OpenCL, relativa ao montante máximo de memória alocável para um *buffer*, foi necessário efetuar uma divisão do volume de dados em blocos de dados com a dimensão máxima de128 MB. Assim sendo, um volume de 1024x1024x1024, é dividido em oito blocos.

Através da Tabela 1, é possível verificar os tempos de execução obtidos para cada um dos volumes de dados, com dimensão de 1024x1024x1024. Na referida tabela pode-se observar o tempo despendido na criação e na pesquisa em profundidade

	Criação		Pesquisa		
Volume	$\bar{x}$ (ms)	σ (ms)	%	$\bar{x}$ (ms)	σ (ms)
Chess	556.17	3.21	11,35	239.78	0.50
Blocks	111.29	3.09	1,88	11.00	0.26
Spiral	67.14	0.99	0,91	0.07	0.00
Snake	8.49	0.04	0,13	0.06	0.00

Tabela 1: Tempo de processamento na união dos blocos.

A diferença nos tempos de execução dos diferentes volumes de dados, diz respeito à quantidade de identificadores presentes no volume, que fazem com que o grafo resultante do processamento em CPU seja maior ou menor. Esse facto é bastante importante para determinar o tempo de processamento necessário para a pesquisa em profundidade, realizada sequencialmente.

No que se refere ao tempo total dispendido pela solução, este pode ser visualizado no gráfico da Figura 10. Estes valores foram obtidos através da média dos tempos de execução de cada um dos volumes. Como se pode verificar, esta solução possui tempos de execução baixos mesmo quando se aumenta o volume de dados a processar.

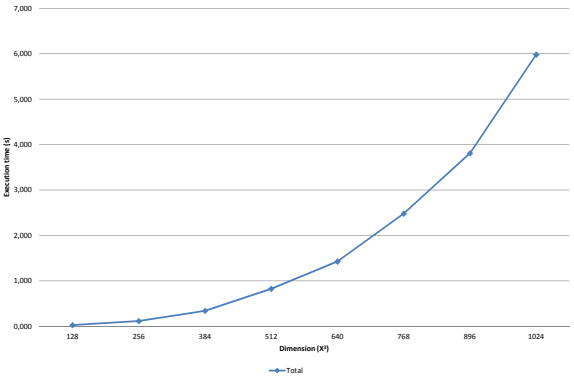


Figura 10: Tempos de execução da solução.

Para descrever cada componente da solução, o gráfico da Figura 11 apresenta a percentagem do tempo total de execução de cada componente. Uma vez mais, refere-se aos valores médios para os quatro volumes de teste. Como se pode verificar, o processamento predomina. Esse facto deve-se à permanência dos dados na memória do GPU enquanto são necessários, bem como às transferências do tipo *pinned*. Neste gráfico pode-se constatar que a componente realizada pelo CPU é bastante baixa.

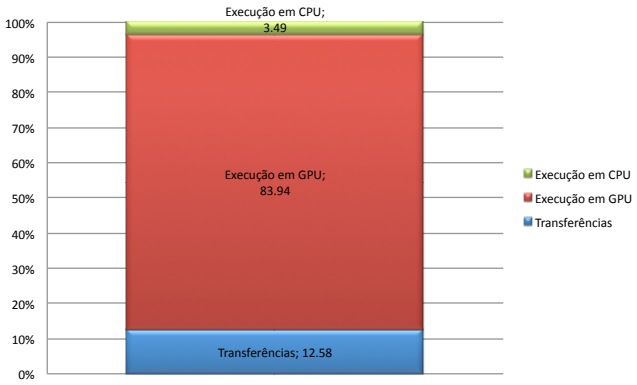


Figura 11: Decomposição do tempo total gasto pela solução.

Em suma, a solução permite o processamento de grandes volumes de dados com pequenos tempos de execução, permitindo a sua utilização em ambientes interativos.

7 CONCLUSÕES

Tal como já foi referido no presente artigo, a solução desenvolvida tem a capacidade de processar volumes de dados de dimensões elevadas, em curtos espaços de

tempo. Sendo assim uma mais-valia no que concerne à identificação de objetos. Através dos tempos de execução obtidos nesta solução, é possível concluir que esta se adequa a ambientes interativos, visto que até ao momento, a mesma era realizada *offline*.

A solução desenvolvida, além da importante aplicação que tem, no âmbito do projeto na qual foi desenvolvida, apresenta também diversas outras aplicações, como na área da saúde, para a deteção de massas em determinadas áreas do corpo humano, bem como na identificação de padrões em imagens.

Esta solução contém algumas limitações, as quais se encontram em estudo, para melhorar o desempenho do algoritmo. Uma dessas limitações diz respeito às interações necessárias realizar pelo *kernel 2*, as quais acrescem *overheads* no tempo total de execução. Uma outra limitação diz respeito à fusão dos blocos realizada em CPU. Embora este usualmente não cause grande impacto no tempo total da solução, em volumes com uma grande quantidade de objetos, os tempos de processamento desta fase começam a ser significativos.

Para além das limitações supracitadas, existem ainda alguns aspetos que se encontram em estudo para melhorar o algoritmo. Um desses aspetos diz respeito à utilização de múltiplos GPGPUs. Atualmente a solução já tem suporte para distribuir os blocos pelos múltiplos dispositivos, contudo, dada a falta de *hardware* com características semelhantes ao nVidia c2050, não foi possível analisar o desempenho da solução corretamente, bem como desenvolver otimizações para a combinação dos mesmos. Uma outra possível melhoria ao algoritmo, seria a possibilidade de identificar objetos provenientes de diversos intervalos de segmentação, os quais são representados com diferentes tons de cinzento.

8. AGRADECIMENTOS

Este trabalho foi parcialmente suportado por FCT/MCTES, projecto Ambiente de Resolução de Problemas para Caracterização Estrutural de Materiais por Tomografia - PTDC/EIA-EIA/102579/2008 e pelo CITI - PEst-OE/EEI/UI0527/2011.

9. REFERÊNCIAS

[Harrison 11] 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.

[Hawick 10] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Comput.*, 36(12):655–678, December 2010.

[Nvidia 11] C. Nvidia. Nvidia opencl programming guide, 2011.

[OS11] Bedrich Benes Ondrej Stava. Connected component labelling in CUDA. In W.W. Hwu, editor, *GPU Computing Gems Emerald Edition*, pages 569–581. Morgan Kaufmann, 2011.

[SCI12] Scirun: A scientific computing problem solving environment, scientific computing and imaging institute (sci). http://www.scirun.org, 2012.

[Suzuki 03] 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.

[Wu 09] K. Wu, E. Otoo, and K. Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Analysis & Applications*, 12(2):117–135, 2009.

A Apêndice

Algoritmo 1 Kernel 1.

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

Algoritmo 2 Kernel 2

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

__private Point3D $globalID \leftarrow$ globalAddress($threadId$);
__private Point3D $localID \leftarrow$ localAddress($threadId$);
__private unsigned int $oldValue \leftarrow sharedMem[localID.index]$;
__private unsigned int $newValue \leftarrow oldValue$;
__private unsigned int $newId \leftarrow BLACK$;
if $(globalID.x > 0)$ and $(localID.x = 0)$ **then**
  $newId \leftarrow image3D$[globalAddress($globalID.x - 1, globalID.y, globalID.z$)];
  if $(newId \neq WHITE)$ and $(newValue > newId)$ **then**
    $newValue \leftarrow newId$;
  end if
end if
(...) // Expansão para $x+1$, $y-1$, $y+1$, $z-1$ e $z+1$.
if $newValue \neq oldValue$ **then**
  $isDone \leftarrow false$;
  $changes[oldValue] \leftarrow newValue$;
end if
barrier(CLK_LOCAL_MEM_FENCE);
__private unsigned int $newId \leftarrow changes[globalID.index + 1]$;
while $(changes[newID] \neq WHITE)$ and $(change[newID] < newID)$ **do**
  $newID \leftarrow change[newID]$;
end while
if $(newId \neq WHITE)$ and $(currId < newId)$ **then**
  $image3D[globalID.index] \leftarrow newId$;
end if