Ricardo Alexandre do Rosário Ribeiro

Licenciado em Engenharia Informática

Protein docking GPU acceleration

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador:	Ludwig Krippahl, Professor Auxiliar, FCT-UNL
Co-orientador:	Hervé Paulino, Professor Auxiliar, FCT-UNL

Júri

Presidente:	Name of the committee chairperson
Arguentes:	Name of a raporteur
	Name of another raporteur
Vogais:	Another member of the committee
	Yet another member of the committee

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2019

Protein docking GPU acceleration

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

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Lorem ipsum.

Agradecimentos

The acknowledgements. You are free to write this section at your own will. However, usually it starts with the institutional acknowledgements (adviser, institution, grants, workmates, ...) and then comes the personal acknowledgements (friends, family, ...).

Resumo

Na área científica da Bioinformática, determinar com precisão o complexo formado pela interação entre um par de proteínas é computacionalmente difícil. Existem métodos e algoritmos para simular a fusão de um par de proteínas, que demoram horas para executar a simulação recorrendo apenas à CPU, sendo em termos de trabalho/tempo ineficiente. Um desses métodos é o BiGGER, desenvolvido pelo prof. Nuno Palma et al. Este algoritmo assume características que lhe dão uma complexidade temporal inferior aos demais, pelo que os tempos de execução do BiGGER são menores do que a maior parte dos algoritmos e respetivos programas de docking. O estudo das interações entre as proteínas tem aplicações medicinais, onde são desenvolvidas formas de proteger o Homem de doenças neuronais assim como traz desenvolvimentos na área do desenho e concepção de drogas assistido por computador.

Para resolver a ineficiência referida, as ferramentas para docking foram optimizadas para usar a GPU como auxiliar na execução das simulações, reduzindo o tempo de execução de horas para minutos ou até mesmo segundos. O presente documento aborda uma proposição para a paralelização do algoritmo BiGGER. A implementação será feita recorrendo a técnicas de computação acelerada i.e. utilizar a GPU da máquina em que corre o algoritmo para auxiliar a CPU na computação que é necessária. Tendo mais recursos à disposição, é esperado que o tempo de execução do BiGGER baixe drasticamente por consequência do aumento significativo de performance face à versão sequencial. Em caso de sucesso, a complexidade futura do algoritmo permitirá a adição de mais vantagens face aos seus concorrentes. Por consequência deste aumento de performance, uma proposta de valor para quem pretenda utilizar o BiGGER será ter uma ferramenta de trabalho eficiente no estudo das interações entre as proteínas em qualquer máquina que tenha uma placa gráfica com as características adequadas.

Palavras-chave: proteínas, docking, computação acelerada,GPU, Bioinformática, BiGGER

Abstract

In Bioinformatics, finding the complex resulting from an interaction between two pairs of proteins is computationally demanding. There are methods and algorithms to simulate the binding between two proteins, however, the computation related to docking has many extensive and repeating steps. Thus, the execution of the simulation if the program is only using the CPU can last for hours, which is very inefficient. One of the methods used is BiGGER, created by prof. Nuno Palma and others. This algorithm has features that give it a lower time complexity compared to others, therefore execution times in BiGGER can be lower than most of the algorithms for docking purpose. Studying protein interactions has medical aplications, contributing to the development of ways to protect, diagnose and heal mankind from neuronal diseases. It also contributes to computer assisted drug design and development. To improve the execution time of docking programs, these were optimized for GPU execution, reducing the execution time from hours to minutes or even seconds. This document presents an aproach to the implementation of optimizations to BiGGER, running it with GPU. This implementation is to be done via high performance computing techniques, so that the machine's GPU assists the CPU on parallelizing those required computations. By having more resources at disposal, it should be expected that the execution time of BiGGER is reduced due to the improvement of BiGGER performance in relation to the sequential version. If the implementations succeed, there will be additional advantages for BiGGER in relation to other algorithms. Thus, an value proposition for those who intend to use BiGGER as a method for efficiently study interactions between proteins in a machine that has adequate hardware.

Keywords: proteins, docking , high performance computing, GPU, Bioinformatics, BiGGER

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Glossário

assessment	Termo inglês para estudar ou analisar e tomar conclusões sobre uma dada matéria.
bottleneck	Zona de código de um dado programa onde é verificado que atrasa a execução deste.
complementaridade	Principio para descrever como é que duas entidades conseguem se unir.
conformação	Sinónimo de ajustamento.
core	Palavra de origem inglesa para definir a região central de uma entidade.
correlação	Relação geométrica entre duas posições num plano.
cristalografia de raios X	Método cientifico usado para elucidação das estruturas das proteínas. A cristalização de proteínas foi descoberta acidentalmente no século XIX (1840), por Hunfeld, antes da descoberta histórica dos raios X por parte de Wilhelm Röntgen (1895). A experiência de Hunfeld envolveu a extração de uma hemoglobina do sangue de uma minhoca, verificando por observação em microscópio .
deprecated	Da lingua inglesa para caracterizar algo que é usável mas foi considerado como obsoleto, não devendo ser utilizado.
hotspot	Zona de código de um dado programa onde a proporção de instruções executadas é maior ou onde a fração de tempo de execução é maior.
surface	Palavra de origem inglesa para definir a região de superficie de uma entidade.

Siglas

API	Application Programming Interface.
APOD	Assess, Parallelize, Optimize, Deploy.
BiGGER	Bimolecular complex Generation with Global Evaluation and Ranking.
BoGIE	Boolean Geometric Interaction Evaluation.
CPU	Central Processor Unit.
CUDA	Compute Unified Device Architecture.
FFT	Fast Fourier Transform.
GPC	GPU Processing Cluster.
GPU	Graphics Processor Unit.
GSC	Grid-based Surface Complementarity.
IDE	Integrated Development Environment.
MPI	Message Passing Interface.
NVCC	NVIDIA CUDA Complier.
PDB	Protein Database.
PDI	Protein-Drug Interaction.
PPI	Protein-Protein Interaction.
PSC	Pair-based Surface Complementarity.
SM	Stream Multiprocessor.
SP	Scalar Processor.

XOR Exclusive Or.

Capítulo 1

Introdução

1.1 Enquadramento e motivação

As proteínas não funcionam de forma isolada, de acordo com Gonzalez e Kahn (2012) [13], estas interagem não só com outras proteínas como também com outros tipos de moléculas, como por exemplo ADN ou moléculas constituintes das drogas. Desta forma os mecanismos que determinam o estado de saúde de um organismo são controlados pelas interações entre proteínas. Por sua vez o estudo destas interações tem garantido avanços na elucidação das formas moleculares associadas às doenças, trazendo avanços na proteção, diagnóstico e tratamento de doenças consideradas incuráveis. Um exemplo a considerar foi em 2018, um investigador português ter descoberto que a interação entre as proteínas S100B e beta-amilóide provocam um atraso na formação dos agregados do beta-amilóide, trazendo como beneficio a proteção contra a doença de Alzheimer [10]. Para além de avanços no estudo das doenças, trouxe também avanços consideráveis no desenho de drogas assistido por computador, permitindo a concepção de novas variantes de produtos farmacêuticos.

Segundo Pierce, Hourai e Weng (2011) [33] a maioria dos complexos de proteínas ainda não foram adicionados à base de dados sobre as proteínas (PDB), que contém apenas os complexos descobertos por cristalografia de raios X. Pelo que existe a possibilidade de usar técnicas de computação para docking na elucidação de estruturas que não constem na PDB, adicionando-as a esta.

No entanto este estudo é computacionalmente pesado, o procedimento envolve uma fase de pesquisa exaustiva sobre o conjunto total de estruturas possíveis para o complexo de proteínas final, a partir de um número elevado de rotações e conformações. O número de possibilidades cresce exponencialmente com o tamanho dos elementos do par[17]. Apesar de ser um processo computacionalmente pesado, este está dividido em

etapas que são boas candidatas para execução em paralelo. Neste contexto, as unidades de processamento gráfico (GPUs) apresentam-se como uma boa solução para aumentar o desempenho da computação associada ao docking entre proteínas, sendo uma solução com custos financeiramente viáveis.

O presente documento aborda a preparação para a futura implementação de acelerações ao algoritmo BiGGER [32], a decorrer na fase de elaboração da dissertação, em que a GPU será útilizada para a paralelização das zonas de código onde a execução do BiGGER passa mais tempo, melhorando os tempos de execução do algoritmo. O tema desta preparação está enquadrado nas áreas de bioinformática e de informática. Bioinformática no sentido de envolver conceitos relacionados com o estudo das interações entre proteínas e informática devido à componente da tese em que são desenvolvidas as optimizações e respetivas análises de desempenho.

1.2 Conceito de docking

De acordo com Halperin et al (2002) [17], docking pode ser visto como um conjunto de passos computacionais a desenvolver para determinar o melhor encaixe entre duas moléculas, sendo elas o receptor e o ligando como está ilustrado gráficamente na figura 1.1. Existem duas vertentes de docking, o docking acoplado (*bound docking*) e docking não-acoplado (*unbound docking*). Segundo Vakser (2014) [41], o docking acoplado é feito com a separação das proteínas de um complexo, voltando a juntar ambas por procedimentos computacionais. Por sua vez no docking não-acoplado, também conhecido como docking predictivo, o complexo final é obtido por estruturas isoladas. Em termos de computação não existem diferenças, no entanto no docking acoplado é mais fácil obter melhores resultados, pois não estão envolvidas alterações de conformação nas estruturas, pelo que estas irão encaixar de forma correta.

No entanto a versão não-acoplada é mais útilizada, pois as previsões sobre complexos formados por estruturas isoladas garantem útilidade cientifica. O problema associado ao docking consiste em duas fases: a primeira engloba fazer uma pesquisa sistemática e filtrar as estruturas de proteínas candidatas. A segunda fase consiste em avaliar os candidatos encontrados na parte anterior de forma a encontrar os corretos[37].

Docking de proteínas por sua vez consiste em prever a estrutura tridimensional do complexo de proteínas através das coordenadas atómicas do ligando e do receptor, consistindo nas duas fases anteriormente referidas. Em ambas, a superfície das proteínas pode ser considerada como rígida, ou seja, no encaixe entre as estruturas não existem sobreposições de elementos em ambas. O receptor fica estático e ao ligando são aplicadas rotações e translações, sendo determinados os complexos candidatos. O passo final da primeira fase corresponde à filtragem de candidatos, avaliando estes através de uma função de score. Nesta fase, a função de score pode incluir como parâmetros a complementaridade de superfície ou de forma [5]. Ambas as complementaridades são determinadas por uma abstração de ambas as proteínas numa grelha tridimensional, determinando para cada

célula da grelha se existe correspondência com uma coordenada atómica da proteína respetiva. A segunda fase consiste na atribuição de pontuação aos candidatos resultantes da fase anterior, através de uma função de score com combinações de parâmetros que envolvem contactos residuais, eletroestática até dessolvatação. A gama de parâmetros tem a ver com as características biológicas do par candidato. Esta fase permite averiguar de que forma o par candidato é correspondido com o par real [1].

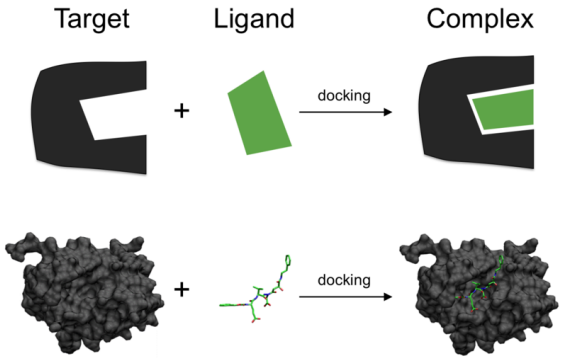

Figura 1.1: Representação gráfica do docking[8].

1.3 Problema

Como foi referido na secção 1.1, a execução em GPU apresenta-se como uma solução que permite um aumento considerável de desempenho na paralelização de etapas constituintes do processo de docking de proteínas.

a GPU (*Graphical Processor Unit*) consiste na unidade de processamento gráfico existente na placa gráfica instalada em qualquer computador, sendo especializada em processamento gráfico, mais precisamente renderização de gráficos 3D. No entanto a GPU também é adequado para processamentos alternativos, que, tal como a rederização de gráficos 3D, são igualmente intensivos demais para a CPU. Na actualidade os GPUs oferecem suporte para interfaces de programação e linguagem de alto nível, sendo possível a quem recorra à execução de um programa na GPU alcançar valores de *speedup* superiores em relação a uma implementação para CPU optimizada, em situações em que o problema a resolver pelo programa envolva computações extensas e repetidas. O uso da GPU para este tipo de processamentos é referido como *General-purpose computation on graphics processing units* (GPGPU) [14]. Exemplos de aplicações podem variar deste cálculo financeiro até aplicações bioinformáticas, como é o caso do docking. É possível encontrar um panorama detalhado sobre as aplicações da computação de alta performance, sobre forma de catálogo [15]. Existem programas implementados para GPU que recorrem a métodos de correção de matrizes por Fast-Fourier Transform. Neste caso uma das opções a adoptar para optimizações consiste em recorrer a bibliotecas externas especializadas em acelerar as etapas relacionadas com FFT, sendo uma das possibilidades a biblioteca

cuFFT, da NVIDIA. É necessário incluir esta optimização pois a maior parte do tempo de execução (mais de 70%) de um docking simulado num programa que use FFT é gasto nestas etapas [21] [36], o que faz com que a performance destes programas dependa da eficiência da biblioteca. No artigo sobre o trabalho do Megadock em GPUs [36], é referido que as versões antigas do cuFFT têm problemas de sincronização. Por sua vez o BiGGER, por não recorrer ao FFT na correlação das grelhas, não necessitará de tais bibliotecas para a versão acelerada por GPU, pelo que a performance desta versão será dependente apenas da implementação em si. Os programas que usam FFT têm ainda como limitação a largura das grelhas tridimensionais necessitar de ser potência de 2 [32], enquanto que para o BiGGER as matrizes devem ter a largura da molécula, pelo que as matrizes deste último são de menores dimensões de espaço e memória.

1.4 Solução

Como solução é pretendido implementar uma nova versão do algoritmo BiGGER, em que serão aplicadas optimizações ao código original, de forma a que este possa ser executado na GPU. A versão futura do BiGGER terá ganhos de speedup consideráveis em relação à versão actual, que neste momento recorre apenas à CPU. Como tal será necessário, inicialmente, identificar as zonas de código do BiGGER que demoram mais tempo e para cada uma destas definir soluções, com base em exemplos existentes, para optimizar o tempo de execução das zonas identificadas. Após o trabalho referido anteriormente estar concluido, as optimizações definidas serão implementadas por uma linguagem de programação específica, passando as zonas de código a serem executadas de forma paralelizada na GPU.

1.5 Contribuições

No final da tese, são esperadas contribuições na identificação das regiões de código do BiGGER que necessitam de paralelização, executando as mesmas na GPU, assim como as respetivas implementações e análise ao desempenho. O código poderá desenvolvido de forma a que possam ser aplicadas manutenções para versões futuras. O produto final será uma versão do BiGGER, com as etapas presentes na versão sequencial, porém aceleradas por GPU, sendo esta a primeira implementação em GPU do BiGGER, que promete assumir uma eficiência superior face às soluções actuais com optimizações relacionadas e que utilizam FFT. Será também feita uma análise de desempenho e exactidão de resultados de docking da versão optimizada, sendo está análise confrontada com os resultados de ambas versão sequencial do BiGGER e programas de docking optimizados para execução em GPU e que usam FFT.

1.6 Estrutura deste documento

O presente documento de preparação assume três capítulos:

1. Introdução
2. Estado da arte
3. Plano de trabalho

O capítulo 1 consiste na introdução sobre os conceitos de docking a ter em conta assim como uma formulação do problema computacional associado ao mesmo.

No capítulo 2 está presente uma visão geral em relação aos diversos métodos e ferramentas de docking, inclusive o BiGGER, indicando as diferenças em relação a este último. Estas ferramentas surgiram antes de ser considerado a GPU para acelerar o docking. Neste capítulo também está incluida uma secção em que são abordados os conceitos associados à GPU e respetiva programação, assim como o que é que existe em termos de software específico para docking com acelerações em GPU relacionado com o BiGGER. Da mesma forma estão presentes os detalhes de implementação e utilidade para as optimizações a desenvolver para o BiGGER. Também são abordados duas APIs para programação em GPU: CUDA e OpenCL.

No capítulo 3 é descrito o ciclo de desenvolvimento a seguir que é específico para desenvolvimento de optimizações para execução em GPU, sendo abordadas as etapas do ciclo e os esforços específicos para cada etapa, relativamente à tese. São ainda identificadas as possibilidades de paralelização nas etapas do BiGGER em relação às optimizações apresentadas no capítulo 2. Este capítulo termina com a definição do plano de trabalhos para o decorrer da dissertação.

Capítulo 2

Estado da arte

2.1 Docking

2.1.1 Conceitos de docking em relação à rigidez da superfície

Ao longo dos anos cada vez mais algoritmos e respetivas adaptações para simular o docking dos complexos de proteínas têm surgido.

Um algoritmo pode ser classificado em função da forma como é tratada a rigidez da superfície dos pares de proteínas a juntar ou até mesmo pelo modelo matemático que seguem, como por exemplo se aplicam a Fast Fourier Transform ou não. Serão apresentadas nas próximas subsecções 3 modelos de docking: flexível, semi-flexível e rígida. A maior parte dos algoritmos, no entanto, adoptaram os dois últimos modelos. Desta forma é comum os algoritmos de docking entre proteínas seguirem o modelo rígido enquanto que os restantes são adoptados por algoritmos de docking proteína-ligando.

2.1.1.1 Docking flexível

Em que ambos os complexos receptor e ligando são considerados como sendo corpos flexíveis e adaptáveis sendo, no entanto, a mesma flexibilidade interpretada pelo algoritmo de forma simplificada ou limitada e por consequência pode-se aplicar um modelo através de simulações de docking.

2.1.1.2 Docking semi-flexível

Um dos elementos do par é considerado rígido e o outro não. Normalmente este tipo de algoritmos trata o ligando como o elemento flexível, já que este é mais pequeno do que o receptor, tendo assim uma maior probabilidade de mudar a sua forma, outra justificação a considerar tem a ver com os custos de computação serem mais baixos do que se

considerarmos os receptores como flexíveis. Como referido anteriormente, este modelo é considerado apenas em docking proteína-ligando, no âmbito de desenho e desenvolvimento de drogas.

2.1.1.3 Docking rígido

O par é considerado como sendo rígido na sua integridade, sendo também considerado que no docking entre os dois corpos uma das proteínas irá acabar por penetrar a outra o que leva a que se tenha de adaptar o conjunto de soluções para o problema em seis dimensões de liberdade, 3 para a rotação e 3 para a translação [41]. Apesar de se considerarem as superfícies de ambos como rígidos, de forma a facilitar o problema foi considerada uma abordagem em que é considerada a ocorrência de variações na superfície molecular do par, permitindo a ocorrência de sobreposições entre átomos de ambas as proteínas. Esta abordagem tem o nome de docking macio (*soft-docking*).

2.1.2 Métodos usados em docking

Existem cinco métodos principais relacionados com o docking que são mencionados no presente documento: Fast Fourier Transform (FFT) [19], O algoritmo BoGIE [32], Hashing Geométrico [11], correlações de Fourier esféricas polares [35] e algoritmos genéticos de Lamarck [24]. Destes, serão apresentados nesta subsecção os três primeiros. O FFT é usado em docking sobre a forma de uma adaptação, sendo uma das formas mais conhecidas para transpor a superfície da proteína para uma grelha tridimensional. No entanto, este método é menos eficiente do que o algoritmo BoGIE, sendo este último usado no BiGGER para a mesma função. A técnica de hashing geométrico substitui a grelha tridimensional usada nos métodos anteriores por uma hash table. A complexidade temporal desta técnica situa-se entre a complexidade temporal do FFT e a do BiGGER, provando ser uma alternativa computacionalmente eficiente. Sobre as correlações de Fourier esféricas polares, foi estudado que são utilizadas no programa HEX [35]. No entanto estas correlações, à semelhança dos algoritmos genéticos, não estão relacionados com o funcionamento do BiGGER, pelo que não serão abordados.

2.1.2.1 Transformada Rápida de Fourier em docking

Existe uma adaptação da Transformada de Fourier para docking entre proteínas, com o nome transformada rápida de Fourier (FFT). Esta adaptação é usada em ferramentas de docking que consideram a complementaridade de superfície entre os pares de proteínas como função de score para determinar os candidatos à fase 2 do docking. Mais precisamente na criação das grelhas geométricas para comparação com as coordenadas atómicas do receptor. Dois exemplos de ferramentas abordadas nesta secção são o FTDock e o ZDOCK que são específicos para docking entre proteínas. A origem do uso do FFT no docking remonta ao artigo de Katchalski Katzir et al [19] onde é considerado que

ambos os pares são corpos rígidos. Também foi com base no estudo deste artigo que as adaptações que levaram ao BiGGER foram efectuadas [32]. A metodologia com que são determinadas as possibilidades consiste, de forma resumida, nos passos:

1. Determinação da região de fronteira, em que é determinada uma função para suportar os pontos de fonteira, tendo para cada nó da matriz as atribuições possíveis: 1 para correspondência a coordenadas atómicas localizadas na fronteira, ρ a coordenadas internas e 0 a externas. Esta função é denominada função discreta de Fourier (DFT).

2. Determinação da função de correlação associada às orientações entre as duas moléculas, sendo considerado que a molécula *a* está fixa enquanto *b* pode ter orientações variadas. Sobre um eixo *xyz* os ângulos que a orientação do ligando pode formar variam entre $360x360x180\Delta^3$, sendo Δ o intervalo de amostragem rotacional.

Em termos de complexidade temporal, executar estes passos tem um custo de ordem $O(N^3 * log_2(N^3))$, sendo N o número de nós presentes na grelha [20].

Considerou-se fazer um estudo deste método importante pois tal como foi explicitado no principio desta sub-subsecção, foi a partir do trabalho de Katchalski Katzir e colegas sobre o FFT que foi estudada a abordagem para o BiGGER [20].

Tendo em conta que os passos aqui descritos e as fases do BiGGER descritas na secção 2.1.3.1 são muito semelhantes, para se perceber como paralelizar o BiGGER é necessário entender como este funciona, e por consequência, como funciona o FFT.

Da mesma forma o FFT foi estudado para que se possa justificar onde é que o BiGGER é mais forte do que as ferramentas que usam FFT, no momento de descrição de resultados obtidos na fase da elaboração.

2.1.2.2 Hashing Geométrico

Este método é shape-explicit, em que as superfícies de ambos os elementos do par são quantificadas e representadas por valores binários nas superfícies *core* e *surface*.

A metodologia deste método divide-se em dois passos: Pré-processamento e Reconhecimento [11]. A fase de pré-processamento consiste em identificar os pontos críticos na superfície do ligando e a partir destes definir frames de coordenadas locais. Sobre estas frames, serão feitas indexações com base nos pontos críticos vizinhos a um selecionado. Os índices serão usados numa hash table que contem as coordenadas locais da frame corrente (o processo é iterativo). Repete-se o procedimento para o elemento receptor. Com as coordenadas locais de ambos determinadas, procede-se para a fase de reconhecimento, em que se usa as coordenadas locais do receptor para confrontar uma correspondência entre as coordenadas do ligando, através da hash table. Se houver demasiadas correspondências, existe uma grande possibilidade de as curvaturas de superfície serem semelhantes, e é feita uma verificação extra com esse âmbito [37]. Na figura 2.1, pode-se consultar uma sintetização sobre as etapas que o método desempenha. A principal vantagem deste método

em relação aos outros é substituir todos os passos que os outros métodos apresentados executam por uma verificação numa hash table, o que introduz eficiência de computação.

A complexidade deste método é $O(N^3)$, sendo N o número de pontos críticos a considerar. Os tempos de execução são baixos, sendo na ordem dos minutos independentemente da complexidade da previsão do docking [17]. No entanto a complexidade temporal é superior à do BiGGER ($O(N^{2,8})$), pelo que em teoria este último assume tempos de execução ainda menores.

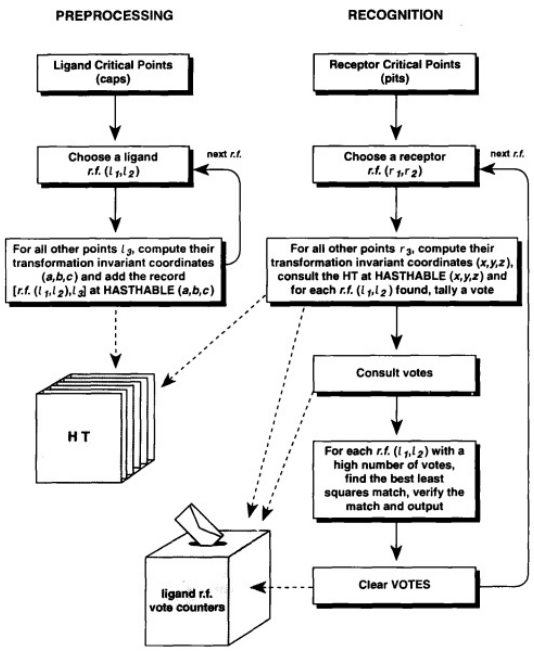

Figura 2.1: Fluxograma sobre o geometric hashing. Tirado de [11]

2.1.2.3 BoGIE

Acrónimo para *Boolean Geometric Interaction Evaluation*[20] [32], este algoritmo encontra-se incorporado no BiGGER e assume o papel de filtrar as poses candidatas através das superfícies de contacto entre os pares. A filtragem é feita através da avaliação do contacto entre ambas as superfícies dos elementos do par. Na adaptação do pseudo-código do BoGIE presente na ilustração 1, é possível verificar que existem dois processos principais a considerar, sendo o primeiro a definição de uma matriz tridimensional de booleanos em que cada posição da matriz representa uma parcela da forma que o complexo assume.

Um nó da matriz referida anteriormente assume valor 1, se a célula corresponde a uma parcela da proteína cujo centro se encontra a uma distância tridimensional designada por esfera de Van der Waals de qualquer outro átomo pertencente a outra proteína e nulo, se a mesma corresponder a frações do complexo que são consideradas como externas.

O segundo passo gera duas matrizes de valores booleanos semelhantes às anteriores para cada um dos elementos do par: a matriz de superfície (*surface matrix*) e a matriz central (*core matrix*) tal como está ilustrado graficamente na figura 2.2.

As células que ocupam a matriz de superfície são as que na matriz inicial do passo anterior assumiram valor 1 mas tinham vizinhos com valor 0, ou seja, pretendem-se os pontos de fronteira.

Na segunda matriz constam as células em que quer o seu valor quer o das suas células vizinhas assumem valor verdadeiro, correspondendo a posições em que o seu centro está próximo do centro do complexo ou podendo até mesmo coincidir.

Por fim a superfície molecular da proteína é determinada deslocando uma cópia da matriz inicial com a mesma em 26 possíveis direções. Para cada deslocação é feita uma operação lógica XOR (OU exclusivo) entre ambas as grelhas referidas. Desta forma a operação terá como output o valor 1 apenas nos pontos da fronteira, pois os valores entre as duas células são diferentes e falso se forem iguais. A avaliação do contacto de superfície entre as duas estruturas ocorre através do cálculo das sobreposições que ocorrem quando se juntam estas. No entanto é necessário atenuar o problema, considerando a rigidez molecular de ambas as proteínas como sendo macia e não rígida, tal como foi explicitado em 2.1.1.3. Como tal é necessário considerar que:

1. As poses candidatas em que sejam verificadas sobreposições entre células internas (*core*) de ambas as grelhas são descartadas por não corresponderem a uma solução realista, no âmbito do docking rígido.

2. Soluções em que ocorram sobreposições entre uma célula interna de uma grelha e outra de superfície são consideradas, favorecendo a atenuação anteriormente referida.

3. O que determina uma solução ser melhor do que outras é o número de sobreposições válidas entre as células das grelhas. É possível determinar a quantidade de células sobrepostas e válidas aplicando uma operação lógica AND entre as duas grelhas de superfície.

Sendo assim o primeiro passo contribui mais para a complexidade deste algoritmo do que o segundo, em que este último depende do output da matriz resultante do primeiro passo e apenas executa um conjunto de operações XOR, pelo que não é tão caro em termos de memória e tempo comparando com a medição para cada célula de uma distância.

```
Clear FirstGrid, SurfaceGrid, CoreGrid;
for f = 1 to number of Atoms do
    Fill FirstGrid cells closer to 1Å+Atoms[f];
end
for ShiftVector=(-1,-1,-1);(-1,-1,0) ... (1,1,1), except (0,0,0) do
    Copy FirstGrid to CopyGrid;
    Shift CopyGrid by ShiftVector;
    SurfaceGrid=SurfaceGrid OR (CopyGrid XOR FirstGrid);
end
SurfaceGrid=SurfaceGrid AND FirstGrid;
CoreGrid=FirstGrid AND NOT SurfaceGrid;
```

Algorithm 1: Pseudocódigo para o BoGIE. Este pseudocódigo inclui apenas a criação das grelhas tridimensionais [20].

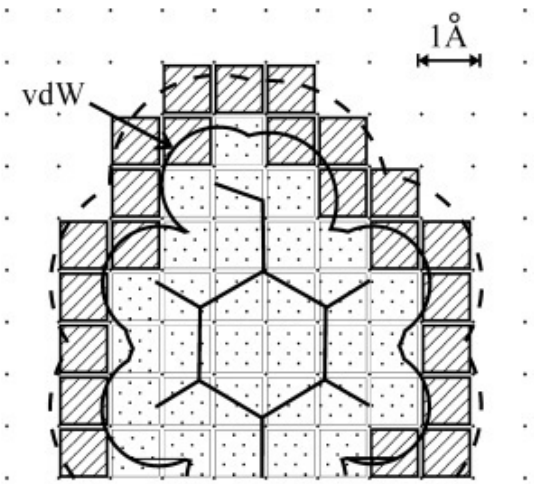

Figura 2.2: Representação em 2D das matrizes resultantes do segundo passo do BoGIE, as células preenchidas a tracejado diagonal correspondem à matriz de superfície, as com pontos correspondem à matriz *core* e a nuvem com tracejado continuo representa o corte associado à esfera de van der waals com a proteína localizada ao centro [32].

De notar, no entanto, que ambos os passos podem ser optimizados recorrendo à GPU, no capítulo 3 serão detalhadas possíveis abordagens à paralelização desta etapa do BiGGER, podendo trazer melhorias para além do uso do XOR.

2.1.3 Ferramentas para Docking

Antes de ser introduzido o termo GPGPU no desenvolvimento de software para docking entre proteínas, foram desenvolvidas várias ferramentas/aproximações para o estudo das interações entre proteínas. Exemplos de casos investigados incluem o BiGGER [32], o ZDOCK [5], o FTDOCK [12], o 3D-Dock [17], o GRAMM[37] e o HADDOCK[9] . No entanto o ZDOCK, FTDock, 3d-Dock e GRAMM são quatro exemplos de programas que

usam FFTs, sendo que apenas será apresentado o ZDOCK, já que este é uma ferramenta que foi referenciada em trabalhos de implementação de optimizações em GPU de programas para docking [29] [36]. Também é apresentado nesta subsecção o algoritmo BiGGER [32] como alternativa às aproximações que usam FFTs, pois como foi esclarecido no capítulo 1, é este o algoritmo que será paralelizado no decorrer da tese. Por fim o HADDOCK não será considerado pois esta aproximação considera a superfície das proteínas como sendo flexível, não estando relacionado com BIGGER, que em todo o processo considera a superfície destas como rígida.

2.1.3.1 BiGGER

O BiGGER[32] é um algoritmo para docking entre proteínas que considera a superfície destas como rígido, fazendo parte do software para docking Chemera.

Este algoritmo consiste em dois passos: o primeiro efectua uma redução de possíveis configurações resultantes de passos de translação e rotação população de cerca de 10^{15} configurações para uma amostra com poucos milhares de configurações corretas, através do algoritmo BoGIE relatado no ponto 2.1.2.3. Na figura 2.3 está ilustrado o processo de docking que o BiGGER segue.

A segunda fase do algoritmo consiste em aplicar metodologias de aprendizagem automática de modo a que se possa prever qual das configurações resultantes corresponde ao melhor ajuste entre os dois complexos, isto é, a que tem o score mais elevado.

Em termos de complexidade temporal, este algoritmo assume valores mais considerados como óptimos ($O(N^{2,8})$) do que os algoritmos que recorrem ao Fast Fourier Transform.

O motivo pelo qual das duas vertentes de algoritmos, o BiGGER assume-se com performance superior em termos de computação, deve-se ao facto de o BiGGER ter sido implementado com diversas optimizações face aos algoritmos FFT.

Sendo uma das optimizações o uso de uma heurística mais eficiente no passo da eliminação de possibilidades: descarta situações em que existem sobreposições entre *cores* ou até mesmo situações que não cumprem com os limites impostos nas restrições introduzidas .

O tempo de execução do algoritmo, segundo os autores do mesmo, estava situado entre as 2H e as 8H, dependendo do par de proteínas em contraste com o tempo de execução para FTT que ronda as 6H, numa máquina com uma CPU do ano de 2000 (Intel Pentium II 450 MHz dispõe apenas 1 core).

Segundo a lei de Moore, o número de transístores presentes numa CPU duplica a cada 2 anos, e por consequência a capacidade computacional, pelo que num computador em 2018 o tempo de execução do BiGGER provavelmente será menor, demorando entre 1H e 4H por exemplo.

for

Algorithm 2: Pseudo código do BiGGER.

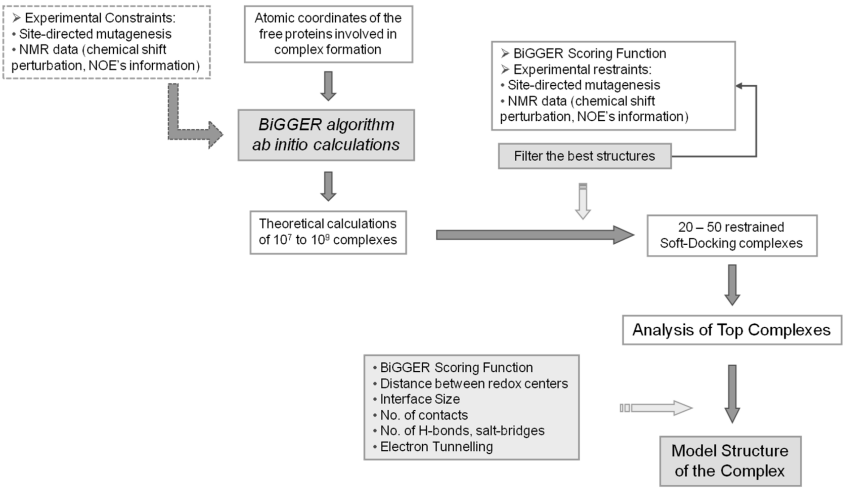

Figura 2.3: Diagrama sobre o processo de docking do BIGGER[2].

2.1.3.2 ZDOCK

O ZDOCK é um programa de docking entre pares de proteínas que usa FFT para optimizar os cálculos respetivos às características das proteínas complementaridade de formato, electrostática e dessolvatação (desolvation), sendo este o aspeto que faz com que o ZDOCK tenha uma performance positiva. O processo de funcionamento deste algoritmo foi abordado em [5]. Aborda a pesquisa de combinações na primeira fase do docking através de uma grelha de pontos. Ao contrário do BiGGER, que considera a superfície e o core, o ZDOCK considera apenas os pontos circundantes ao receptor. O número total de pontos desta grelha que correspondem a pontos do ligando contribuem para uma função de score especifica chamada GSC (Grid-based Shape Complementarity), tem ainda uma subtração que consiste na penalização de confronto entre os pares de proteínas [4]. Existe ainda a função de score PSC (Pair-based Shape Complementarity) que aplica o mesmo raciocinio para o GSC, inclusive a penalização, mas apenas considera os pares de átomos receptor-ligando que se encontram a uma dada distância. Existem quatro possibilidades para as funções de score: combinar as ditas características químicas das proteínas (GSC + dessolvatação + electrostática entre o par) numa só função, usar apenas PSC, combinar está ultima com a dessolvatação e substituir a primeira alternativa referida pelo PSC. Estas combinações são consideradas pois apenas usar GSC ou PSC não garante as soluções mais precisas, pelo que é necessário existir uma função de score a complementar uma das duas.

A diferença entre esta ferramenta e o BiGGER foca-se essencialmente na complexidade temporal: o ZDOCK por recorrer a FFT, requer a mesma complexidade temporal deste, tendo o BiGGER a complexidade temporal que garante a performance superior. No entanto, a grande diferença entre os dois foca-se no modo como é simulada o docking. O ZDOCK faz comparações entre os pontos da grelha, quer para determinar uma correspondência, quer para comparar a distância entre os elementos de um dado par de átomos

para determinar os valores da função de score. O BiGGER, por outro lado, apenas faz operações booleanas sobre as suas grelhas com os pontos de superfície.

2.2 A GPU

2.2.1 Arquitectura e Modelo de Execução

Tal como os CPUs, os GPUs também seguem uma arquitetura. Os conceitos essenciais das arquitecturas dos GPUs modernos são transversais aos diferentes modelos existentes, inclusive de diferentes marcas. No presente documento serão utilizadas como referência as arquitecturas dos GPUs NVIDIA, em particular a arquitetura mais recente [1] de nome Volta[28] que está presente nos GPUs de modelo Tesla V100 (figura 2.5). Esta nova arquitetura traz diversas optimizações de hardware e lógicas face às versões anteriores, eg. Pascal, Maxwel e Kepler, para desempenho em computações na área do *deep learning*. Também é uma arquitetura própria para acelerações relacionadas com aplicações que usam data-centers.

Em termos gerais as arquiteturas têm os seguintes elementos:

- **Streaming Multiprocessors** : Cada GPU tem uma quantidade variável de *streaming multi-processors* (SMs). Os SMs, por seu lado, são compostos por um conjunto de processadores escalares (SPs) que são também conhecidos como os *cores* da GPU. Os SMs Assumem a função de executar os *kernels* (sobre estes últimos é feita uma descrição detalhada na subsecção 2.2.1.2). Têm ciclos de relógio mais baixos, mas suportam paralelismo ao nível de instrução. As componentes dos SMs vão sendo melhoradas face aos SMs de arquiteturas anteriores. A destacar o número de registos que os SMs vão dispondo, a cache L1 e o número de *cores* de execução [42].Os SMs são agrupados em partições de hardware com tamanho igual denominados GPU processing clusters (GPCs) e o número de GPCs presentes num GPU depende da arquitetura.

  Na arquitetura Volta, cada um dos 84 SMs presentes, estão particionados em quatro blocos de processamento como se pode ver na figura 2.4. Cada um destes blocos é composto por 16 cores FP32, 8 cores FP64 e 16 cores INT32. Cada SM é capaz de executar no máximo 2048 threads. Foram aplicadas optimizações nos SMs para a versão Volta face a versões anteriores, mais precisamente a adição de *tensor cores*, que são componentes especiais para acelerar as operações associadas a redes neuronais. O GV100 encontra-se dividido em 6 GPCs, cada um destes GPCs contem 14 SMs.

- **Hierarquia de memória**: a GPU contém uma memória global, partilhada por todos os SMs. Esta memória global tem um quantidade de espaço que varia entre 12GB para a arquitetura Kepler e 16GB para a arquitetura Volta. Além disso, existem ainda dois níveis de cache a considerar. As caches L1 são usadas para melhorar

[1] O presente documento foi escrito no ano lectivo de 2018/2019.

Figura 2.4: Esquema do SM para o GV100[28].

a latência das operações globais de escrita e leitura e como especificado no ponto anterior, estas fazem parte dos SMs. Existe ainda uma cache partilhada L2 para complementar a presença das L1. A cache L2 é uma cache de escrita/leitura com uma politica de substituição *write-back*. Esta cache responde a pedidos de instruções load, store assim como instruções atómicas de ambos SM e respetivas caches L1, preenchendo de forma igual as respetivas caches [25]. A partir da arquitetura Volta a cache L1 e a memória partilhada de cada SM passam a estar juntas, o que traz beneficios para a L1 como o aumento da capacidade de memória/SM em 7 vezes a capacidade da arquitetura Pascal, a diminuição da latência de acesso e o aumento da banda-larga[28]. Também existirá uma nova cache de instruções L0 em cada um dos blocos do GV100, melhorando a eficiência face ao uso de buffers de instruções dos SMs anteriores.

2.2.1.1 Capacidade de computação de uma arquitetura

Todas as arquiteturas NVIDIA têm o conceito de capacidade de computação rótulado a um valor (tabela 2.1). Este valor determina as funcionalidades permitidas pelo hardware respetivo, assim como as melhorias nas componentes de hardware face a arquiteturas anteriores. O número de registos presentes na GPU, o número máximo de threads em cada

Figura 2.5: Esquema da arquitetura da GPU, mais precisamente do Volta GV100[28]. Esta é a arquitetura mais recente que NVIDIA lançou no mercado.

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Cap. Computação	3.5	5.2	6.0	7.0

Tabela 2.1: Tabela de capacidades computacionais das arquiteturas NVIDIA mais recentes. Adaptado de [28].

SM e a granularidade de alocação dos registos variam com as diferentes capacidades de computação [26]. O valor da capacidade de computação pode ser usado pelas aplicações em tempo de execução para determinar o que a GPU presente na máquina dispõe em termos de funcionalidades e instruções nativas. Este valor é também décimal, sendo a casa das unidades respetiva ao número de revisão maior e o das décimas ao número de revisão menor. Se uma arquitetura tem um valor de capacidade de computação superior a uma segunda, por ser mais recente, quer dizer a primeira arquitetura tem as funcionalidades e características de hardware da segunda, com mais umas adições novas, assim como a primeira consegue resolver os problemas endereçados na segunda. O que faz com que um programa que tenha sido implementado em referência a uma arquitetura Kepler, com capacidade computacional 3.5, possa ser compativel para execução num GPU com a arquitetura Volta, com capacidade computacional 7.0.

2.2.1.2 Modelo de execução

O modelo de execução em GPU inclui o conceito de computação heterógenea, em que temos dois conjuntos de computações de carácter geral na GPU concorrentes a executar código: o conjunto *host* composto por CPUs e o *device*, composto por GPUs. Os dois

sistemas desempenham papeis diferentes. O *host* coordena as transferências de dados a manipular entre as duas entidades e a invocação dos *kernels*. Também gera a alocação de memória nos *devices*. Os *kernels* são funções programadas para executar um determinado número de vezes N em paralelo por N threads da GPU, tendo cada uma destas threads um ID único, que é acessível dentro do *kernel*. O executa os *kernels* e manipula os dados que o *host* alocou e transmitiu, retornando o resultado para o *host*. As *threads* da GPU são consideradas como *lightweight* pois são escalonadas em grupos conhecidos como *warps*[26][1]. No caso da GPU ter de ficar à espera de um desses grupos, este tem a possibilidade de avançar para outro, pelo que não é necessário haver o sistema de trocas presente na CPU. As *cores* da CPU minimizam a latência para um número muito reduzido de threads de cada vez, enquanto que as *cores* da GPU permitem a este gerir um número muito maior de threads mais ligeiras, maximizando o *throughput*. Uma computação que

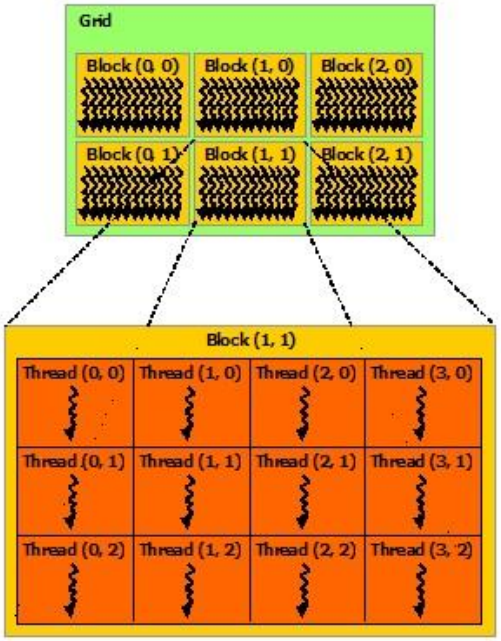

Figura 2.6: Ilustração de uma grelha de threadblocks, detalhando um elemento da grelha para ilustrar os pormenores de um bloco de threads. Neste caso o bloco é bi-dimensional, mas existe a possiblidade de ser tridimensional, o mesmo se aplica à grelha [27].

é executada pela GPU tem de ser estruturada numa grelha.

Cada um dos elementos da grelha é um bloco referido como *thread block* (na figura

[1] Tendo em conta a quantidade de *threads* individuais que têm de ser geridas e executadas de forma eficiente, é empregado pelos SMs uma arquitetura especifica para o efeito, denominada SIMT (*Single-Instruction Multiple-Thread*) [22]. É ainda permitido por parte de qualquer uma das arquiteturas referidas a criação, gestão, escalonamento e execução de *threads* concurrentes em grupos de 32 cada. Estes grupos denominam-se *warps*, podendo cada bloco de *threads* do CUDA ter 1 ou mais warps [25].

2.6 podemos consultar um exemplo de um *thread block*). A dimensão máxima associada ao tamanho de um *thread block* é dependente da arquitetura. No caso das arquiteturas mais recentes é 1024 threads. Numa situação em que é pretendido fazer computações numa estrutura de dados de tamanho superior a 1024 unidades, a mesma é repartida em partes de tamanho igual, sendo cada uma das partes é atribuida a um *thread block* para processamento.

2.2.2 Modelos Base de Programação

Em termos de programação em GPUs existem como APIs o CUDA (*Compute Unified Device Architecture*)[6] que foi implementado pela NVIDIA e o OpenCL(*Open computing language*)[31]. Ambos suportam a linguagem C/C++ apesar de poderem suportar outras linguagens, como por exemplo Free Pascal no caso do OpenCL. O CUDA apenas funciona com placas da NVIDIA enquanto que o OpenCL permite efectuar paralelizações em hardware de diferentes arquiteturas e tipos, desde CPUs a *clusters*. Sobre o CUDA existem bindings para outras linguagens, como por exemplo pyCUDA para a linguagem Python ou a biblioteca especializada com o cuFFT[7] para acelerar a técnica FFT. Por sua vez o modelo de programação em OpenCL oferece uma linguagem comum, interfaces de programação e abstrações em hardware, permitindo ao programador acelerar aplicações com computações paralelizadas por dados ou por tarefas, num ambiente de computação heterogénea [38].

O programador deve elaborar código em duas vertentes. Por um lado tem de programar as tarefas do lado do *host*, mais precisamente alocações de memória na GPU, eventuais transferências de dados entre a CPU e a GPU que vão ser manipulados na execução dos *kernels* e quando é que os estes são invocados. Sobre os *kernels* o programador tem ainda de especificar os parâmetros de execução (dimensão do bloco e número de threads a invocar no *kernel*), sendo que com estes parâmetros o CUDA/OpenCL determina a quantidade de threads a lançar. Por outro tem de implementar o código que cada thread da grelha deve executar, a correr dentro do *kernel* em que pretende fazer a respetiva paralelização. O código sequencial deve ser executado no *host* e o código a paralelizar no programa deve ser executado no *device*.

2.2.3 Optimizações

Um dos passos na programação de versões aceleradas em GPU para um programa consiste em aplicar um conjunto de possíveis optimizações à versão paralelizada do programa, de forma a optimizar a performance do programa para que se possa comparar o resultado final com as expetativas iniciais. Existem quatros aspetos a considerar quando pretendemos optimizar um programa recorrendo à GPU, com os respetivos detalhes de acordo com [26]:

- **Sobreposição de comunicação com computação** : É necessário sobrepor as computações do *host* e do device com a comunicação pois ter as duas sem sobreposição pode afectar a performance. Em CUDA tal pode ser feito através de *streams*. *Streams* são sequências de operações que são executadas no *device*, por uma dada ordem imposta pelo *host* podendo ser cópias de memória ou execuções de *kernels*. Apesar das operações numa *stream* terem de ser executadas pela ordem imposta pelo *host*, as operações entre *streams* podem ser interligadas, havendo sobreposição, e por consequência, possível a esconder a latência associada à transferência de dados entre o *host* e o *device*. Dependendo da arquitetura do device, é possível sobrepor a comunicação entre host e *device* e a computação respetiva à execução do *kernels*. O requisito é ambas as instruções serem de *streams* diferentes e não por omissão[2], caso contrário as instruções terão de ficar à espera de instruções anteriores no device terem acabado, sem poderem começar, o que impossibilita a sobreposição das instruções e esconder a latência de comunicação.

- **Taxa de ocupação da GPU** : É essencial, para a performance ser óptimo, manter os SMs da GPU o mais ocupados possível no decorrer da execução da aplicação, devendo existir uma distribuição de trabalho equilibrada entre os SMs. A aplicação final deve estar implementada de forma a que use os threads e respetivos blocos maximizando a útilização do hardware, evitando situações em que se deixa de impor a distribuição livre de trabalho entre os SMs. A taxa de ocupação determina o quão efectivamente a GPU se encontra ocupado, isto é, o número de *warps* que estão activos em relação ao número máximo de *warps* que a GPU consegue activar. É pretendido que esteja o mais próximo possível de um certo limite que depende da capacidade de computação da arquitetura da GPU [3]. Exceder este limite não traz melhorias de performance, no entanto se o código estiver longe de atingir este limite é garantido que a performance não vai ser óptimo. Para garantir a taxa de ocupação adequada à GPU, é possível optar por garantir que os *kernels* são executados ao mesmo tempo, o que é chamado de execução concorrente de *kernels*.

- **Optimizações de memória** : Em que são consideradas as memórias global e partilhada do device. Sobre a memória global, esta é acessivel via transações de memória de 32, 64 ou 128 bytes. Estas transações devem estar naturalmente alinhadas, o que implica apenas os segmentos de memória cujo primeiro endereço é um multiplo do tamanho do segmento, sendo este 32, 64 ou 128 bytes poderem ser escritos ou lidos pelas transações. Quando um warp executa uma instrução que pretende aceder à memória global, é feita a coalescência dos acessos à memoria por parte das threads do warp numa quantidade de transações de memória que depende do tamanho da

[2] Uma *stream* por omissão tem o seu *streamID* com o valor nulo. Pelo que o pretendido é uma *stream* cujo o seu id seja diferente de 0.

[3] Como foi descrito no ponto 2.2.1, a respeito da capacidade de computação de uma arquitetura, o respetivo valor está associado ao número de registos presentes em cada SM, que pode variar dependendo do valor da capacidade.

palavra acedida por cada thread e da distribuição dos endereços de memória pelas threads do warp. Quanto maior é o número de transações necessárias, maior é o número de palavras não usadas que são transferidos em adição às palavras acedidas pelas threads, o que tem como efeito a redução do *throughput* de instruções[27].

No caso da memória partilhada, esta tem uma latência de acessos menor do que a memória global, e largura de banda superior. A memória partilhada está dividia em módulos de memória com tamanho igual, chamados *banks*. Os *banks* podem ser acedidos de forma simultânea. Existe a possibilidade de ocorrer *bank conflicts* quando dois endereços de um pedido de acesso à memória correspondem ao mesmo *bank*. Tendo o acesso de ser serializado e o pedido dividido em sub-pedidos separados que são livres de conflito. Por consequência o throughput é reduzido em um factor que depende do número de divisões efectuadas.

- **Controlo de fluxos** : É muito importante evitar que ocorram divergências na execução de threads de um mesmo *warp*. Esta situação pode acontecer quando dentro do código de um *kernel* existem instruções de controlo de fluxos (eg. if, switch,while, do while e for) o que leva à redução do throughput de instruções devido ao facto de existirem threads dentro de um *warp* a divergir em caminhos de execução diferentes. No entanto podem existir situações em que o fluxo de controlo depende unicamente do thread ID, nessas situações é importante a escrita da condição de controlo de forma a atenuar o número de *warps* a divergir. Outra forma de garantir que não existem divergências é tornar fácil para o compilador [4] o uso de *branch predication* ou seja, o compilador desenrola os loops/condições impedindo divergências de *warps*. Apenas as instruções em que o predicado assume o valor verdadeiro em relação à condição de controlo são executadas.

2.3 Docking em GPU

Existem vários trabalhos que focam a paralelização de etapas constituintes do processo de docking na GPU[5]. Destes, são abordados nesta secção o Megadock, o PIPER e o AutoDock. O Megadock e o PIPER são os programas mais conhecidos cujo funcionamento é semelhante ao do BiGGER. O AutoDock, apesar de ser um programa cujo funcionamento é diferente do BiGGER, foi considerado pois foi desenvolvida uma paralelização à etapa de *scoring* [18] onde são discutidas duas abordagens em função da taxa de ocupação da GPU assim como é discutida a possibilidade sobre o uso da memória partilhada deste. O Megadock é um dos programas em que a aceleração aumentou drásticamente após o desenvolvimento da versão 4.0, demonstrando os beneficios de adaptar um programa para

[4] No caso do CUDA o compilador é o NVCC, no caso do OpenCL é o OpenCL Compiler.

[5] Na investigação foram encontrados como programas de docking que podem ser executados em GPU o Megadock, o PIPER, o HEX, o AutoDock e o MolDock. Estes dois ultimos são para docking entre proteína-ligando.

executar em GPU. Os trabalhos sobre o Megadock [36] [29] abordam uma possibilidade para mapear etapas do funcionamento do BiGGER para a GPU.

Foi considerado o PIPER pois em termos históricos este foi um dos primeiros programas para docking a ser acelerado usando o CUDA[39]. A subsecção respetiva mostra ainda a importância de aplicar manutenção ao código acelerado de versões anteriores e o impacto da não aplicação de manutenções na performance quando se executa um programa com hardware mais recente, sendo desenvolvida uma versão mais recente [21].

2.3.1 Megadock

O Megadock 4.0[29] é um software de protein-protein docking de origem japonesa, com optimizações para executar recorrendo à GPU. Este programa usa a técnica de Fast Fourier Transform na sua correlação de matrizes.

Este programa é adequado para máquinas que têm muitos *cores* de GPU e CPU à disposição, características tipicas de supercomputadores. No entanto é possível útilizar o megadock em computadores pessoais, alterando a flag de compilação do programa para usar apenas a implementaçãa GPU. O funcionamento do Megadock 4.0 envolve a criação de um processo master que faz a aquisição de uma lista de pares de proteínas e distribui o docking dos pares para os workers presentes nos nós disponiveis. Estes, por sua vez, distribuem o trabalho de calcular a rotação do ligando em cada nó da lista, pelos diversos GPUs e CPUs do nó do cluster. A execução pelos GPUs de cada nó é feita por CUDA e pelos CPUs por OpenMP . Uma das vantagens que este protocolo assume é a tolerância a falhas pois o nó master consegue supervisionar os resultados dos jobs executados pelos workers, além disso é escalável com o número de elementos que compõem o cluster.

2.3.1.1 Execução em GPUs

As acelerações implementadas para o Megadock consistem em optimizações para 6 etapas. Na figura 2.7 ilustram-se as etapas no processo de docking no Megadock, foi aplicado um profile sobre o funcionamento do programa com apenas 1 core da CPU, registando os tempos de execução em cada etapa. Os resultados presentes em [36] indicam que as etapas P4 a P8 consomem a maioria do tempo de execução. Estas etapas constituem um ciclo em que se iteram as possibilidades de ângulos para a rotação do ligando. No caso da P4 as coordenadas do ligando são atualizadas de acordo com uma dada matriz de rotação e o processo respetivo é independente para cada átomo, sendo paralelizável. Esta etapa foi acelerada mapeando as coordenadas atómicas do ligando para a GPU. A segunda vertente da P4 consiste na voxelização do ligando, em que é feita uma afectação a uma posição da grelha em relação às coordenadas atómicas de um dado átomo do ligando. As coordenadas devem pertencer à região interna da curva de van der Waals. Este processo é também paralelizável em relação a cada átomo. Pelo que nesta vertente os átomos também são processados em paralelo e mapeados para a GPU, sendo cada átomo designado a uma *core* da GPU.

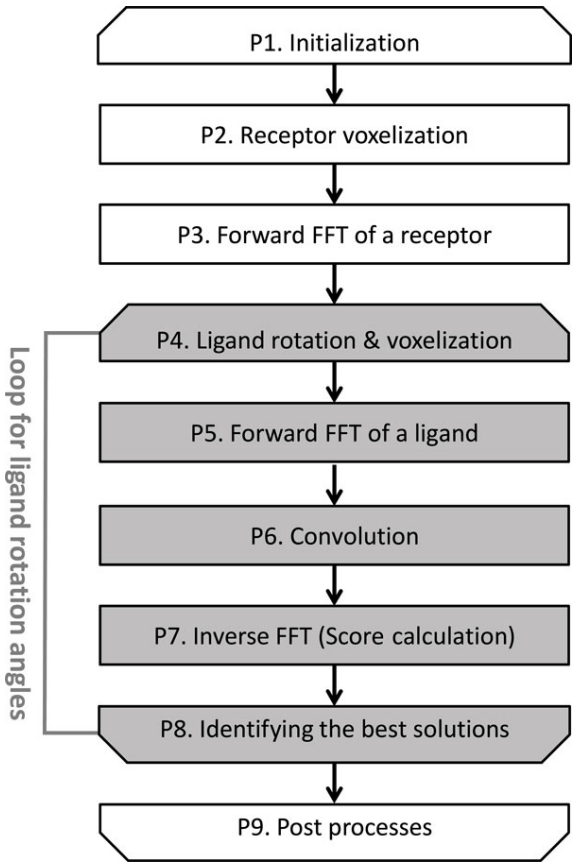

Figura 2.7: Etapas do processo de docking no Megadock. As etapas a cinzento foram aceleradas por GPU[36].

Para as etapas P5 e P7, em que é feita a FFT discreta do ligando e o cálculo da inversa do FFT deste, respetivamente, foi útilizada a biblioteca especializada para FFT, cuFFT. Na etapa P6, convolução, foi aplicado um mapeamento para a GPU. E na última etapa do ciclo, P8, em que os resultados têm uma dada pontuação, foi aplicada uma operação de redução na GPU. Sobre a comunicação *host-device*, é referido que apenas aconteceu uma vez a transferência de dados. O conteúdo da transferência inclui as coordenadas atómicas do ligando e a grelha do receptor com o FFT aplicado.

Em termos de performance e tempos de execução, os testes efectuados em 2014 revelam que a implementação MIC do Megadock 4.0 foi capaz de fazer em 3H, usando 420 nós, um caso de teste que requer 250.000 *dockings* que versões anteriores do Megadock levariam dias [29]. A implementaçãa GPU demonstrou ser 15.1 vezes mais rápida do que a versão que apenas usava um *core* da CPU.

2.3.1.2 Conclusões

Sobre a implementaçãa GPU do Megadock, em que foi usado CUDA, existem aspetos que podem ser aproveitados para ajudar na paralelização do BiGGER. Os mecanismos master-worker não são tão importantes pois o trabalho a desenvolver na elaboração da dissertação

não engloba programação em clusters. Ainda sobre [36] é revelado que a implementação MIC, apesar de ter menos custos de implementação, não é tão eficiente como a implementaçãa GPU na aceleração do FFT . As abstrações usadas na implementação dos *kernels*, no entanto, podem ser aproveitadas, mais precisamente a aplicação de uma operação de redução no passo do BiGGER que determina o ângulo de rotação óptimo para o ligando. Também podem ser aproveitados os mapeamentos para a GPU descritos. Sobre os custos de implementação, é referido no artigo sobre a implementaçãa GPU do Megadock [36] que no processo de mapear o docking para a GPU, foi necessário escrever várias funções *kernel*, assim como escrever instruções para facilitar a transferência de dados entre o *host* e o *device*. No total foram acrescentadas 1000 linhas de código às 7000 que o Megadock tinha originalmente, sendo necessário adicionar ficheiros de código-fonte e ramificações no código. Pelo que é esperado que desenvolver a paralelização do BiGGER mapeando etapas do mesmo para a GPU tenha um nível similar de custos de implementação.

2.3.2 PIPER

À semelhança do Megadock, o PIPER é um programa de protein-protein docking baseado em grelhas FFT para calcular a complementaridade de superfície. O PIPER introduziu o uso da energia de desolvação do par na função que avalia os complexos candidatos, complementando as funções de score respetivas à forma do complexo e a eletroestática. O fluxo de programação do PIPER consiste em duas fases: a fase *setup* envolve a leitura dos dados relacionados com as moléculas, que são passados como input em ficheiro, a computação dos passos relacionados com o receptor, e a criação das grelhas do ligando por correlação em FFT. Após o *setup* estar concluido são iteradas as possíveis rotações, em que as etapas referidas na figura 2.10 são aplicadas para cada rotação.

2.3.2.1 Execução em GPUs

A implementação de acelerações ao PIPER[39] por GPU ocorreu inicialmente em 2009. No entanto em 2014, a performance da versão PIPER GPU de 2009[6] foi confrontada com a versão CPU de 2014[7] através da execução de um profile à performance das duas versões. A versão CPU2014 introduziu optimizações no algoritmo original, mais precisamente foi alterada a bibilioteca que aplicava o FFT. Verificou-se que a versão CPU2014 conseguiu ter performance superior às acelerações introduzidas em 2009, com execuções útilizando um GPU de 2014 [21]. A proporção de tempo gasto no passo de correlação das matrizes é maior na versão CPU do que na GPU09 como se pode observar na figura 2.11. No entanto, as proporções para a filtragem, acumulação e cálculo do score assim como a atribuição de grelha e rotações é maior na versão que usa GPU do que na versão CPU. Pelo que foi desenvolvida em 2014 uma solução com acelerações em GPU que aborda a paralelização

[6] Esta versão será referida ao longo do texto coma GPU09
[7] Esta versão será referida ao longo do texto coma CPU2014

dos passos de filtragem, atribuição de grelhas e transferência de dados entre o *host* e o *device*[8]. Ambas as soluções foram desenvolvidas em CUDA.

Sobre o passo de filtragem e cálculo do score, para a versão de 2009 foi implementado um *kernel* para a filtragem e atribuição de score para cada um dos conjuntos de coeficientes que podem ser adicionados à função que determina a energia do par, para uma rotação. O caso de uso ideal consistia em útilizar 8 desses conjuntos ao mesmo tempo, e útilizar 1 SM para calcular o *score* óptimo de cada um dos conjuntos[9]. Esta optimização deixou de ser válida pois em 2014 uma das optimizações na versão CPU2014 do PIPER foi reduzir o número total de conjuntos de coeficientes a ser processados sendo que na versão GPU09 apenas um SM da GPU estava a ser utilizado. Passou a ser pretendido para o passo de filtragem encontrar o score óptimo para um conjunto de coeficientes no menor tempo possível, repetindo o mesmo procedimento para cada um dos restantes conjuntos.

A versão GPU2014 introduziu nestes dois passos a adição de dois *kernels* em alternativa a usar apenas um na versão GPU09 para os dois passos. Ambos os *kernels* partem o trabalho total em duas fases de forma a que a memória partilhada da GPU possa ser útilizada para acessos rápidos de memória assim como o trabalho possa ser distribuido por todos os SMs e são repetidos para cada um dos conjuntos.

O primeiro *kernel* particiona por todos os SMs desocupados os dados da grelha molecular (figura 2.8). Este *kernel* é lançado com um número de thread blocks que permita que cada SM fique ocupado com uma quantidade adequada de trabalho. Em cada um dos thread blocks, cada thread acede a uma parcela do output, calcula o score óptimo dentro do subconjunto e guarda o resultado num endereço de memória partilhada para o bloco correspondente. O acesso ao subconjunto segue as características apontadas na subsecção 2.2.3 em relação às optmizações de memória. O *kernel* é finalizado quando cada thread executada determina o score óptimo geral para cada bloco e guarda este na memória global.

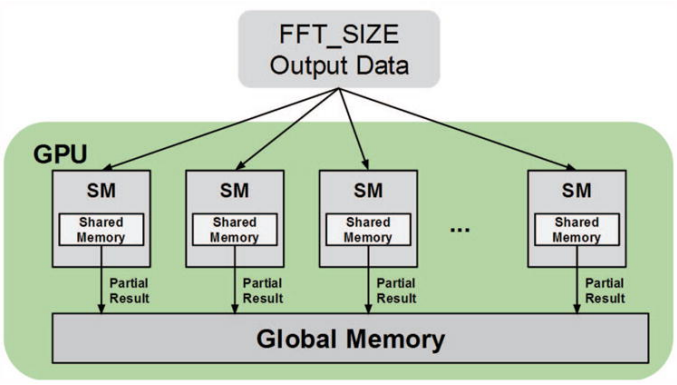

Figura 2.8: Esquema para o primeiro kernel introduzido na versão GPU2014 [21].

[8] Esta versão será referida ao longo do texto coma GPU2014

[9] A soluçãa GPU09 foi desenhada com a GPU Tesla C1060 em mente, esta arquitetura tem 30 SMs, ficando com 8 ocupados na execução do *kernel*, o que garante uma performance melhor do que uma versão do PIPER que apenas útilize a CPU.

O kernel anterior usa mais do que um thread block, por sua vez o segundo *kernel* apenas usa um bloco de threads (figura 2.9). O bloco a usar tem de ter o mesmo número de threads que o número de blocos que foram usados no passo anterior. Cada uma das threads do bloco compara dois scores e escreve o melhor dos dois na memória partilhada para o bloco, sendo feita uma sincronização para garantir que as threads operam no mesmo passo de iteração e sobre memória consistente. O score óptimo é determinado e guardado na memória global.

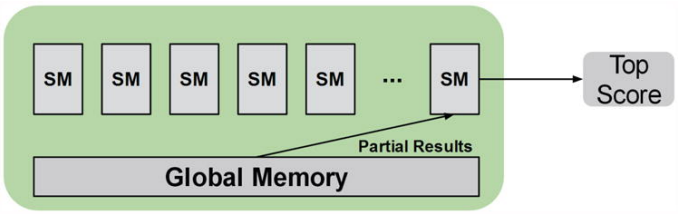

Figura 2.9: Esquema para o segundo *kernel* introduzido na versão GPU2014 [21].

Para além da adição dos dois *kernels* referidos anteriormente, o tempo em que a GPU não desempenha funções enquanto que a rotação e atribuições na grelha estão a ser processados pela CPU foi eliminado. A computação da grelha do ligando passa a ser desempenhada pela GPU, deixando de necessidade de a transferir entre a memória do *host* e a da GPU. Esta optimização tem como requisito um GPU com memória global suficiente para armazenar os arrays relacionados com as atribuições da grelha do ligando.

No entanto houve optimizações da soluçãa GPU09 que foram mantidas. Nas etapas em que a aplicação do FFT às grelhas é feita, a optimização original consistiu em aplicar a bibilioteca cuFFT à semelhança do que foi feito para o Megadock sobre os passos que envolvem FFT sobre as grelhas. As restantes etapas foram paralelizadas mapeando todo o processo para a GPU, novamente à semelhança da implementação para o Megadock.

2.3.2.2 Conclusões

Esta subsecção mostra que nem sempre uma versão de um programa com acelerações em GPU é superior a uma outra versão mais recente do mesmo programa que use a CPU com optimizações. Pelo que é necessário adaptar o código às funcionalidades que as arquiteturas GPU correntes suportam assim como às alterações que o programa original sofre. As optimizações que foram aplicadas ao PIPER em 2014 foram aplicadas à fase de *scoring*, mais precisamente às etapas de filtragem e cálculo dos *scores* considerados como óptimos. Pelo que é mostrado forma de optimizar a fase de *scoring* do BiGGER, implementando uma redução como foi confirmado no caso do Megadock para a mesma etapa.

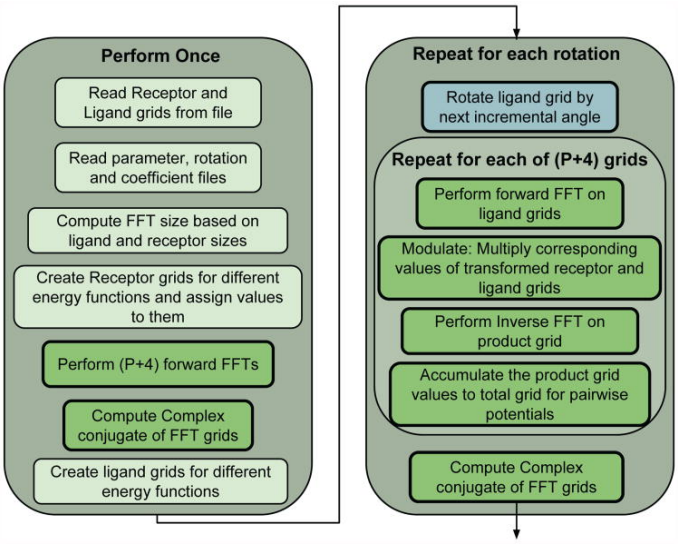

Figura 2.10: Etapas do processo de docking no PIPER. As etapas destacadas a verde escuro foram aceleradas por GPU em 2009 e a etapa a azul em 2014. [21]

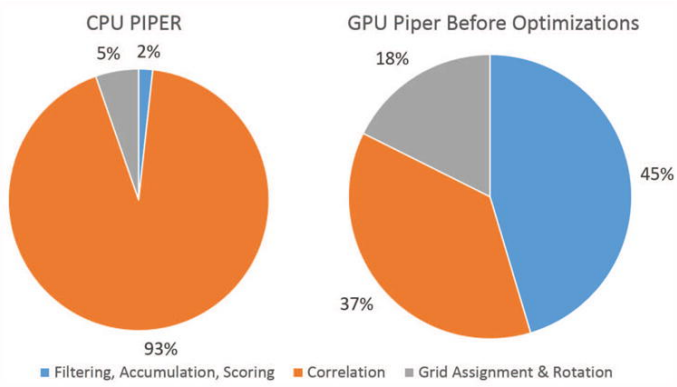

Figura 2.11: Proporções de tempo gasto na execução para a versão CPU optimizada e a versão de 2009 que usa GPUs[21] .

2.3.3 AutoDock

O AutoDock [23] é um programa diferente dos até agora indicados, não só por ser específico para interações entre proteína-droga[10] como também por útilizar o algoritmo genético de Lamarck para determinar a posição correta dos elementos do par, em alternativa ao uso da complementaridade de superfície como função de score. Pelo que descarta as operações binárias que o BiGGER usa e as correlações usando FFT que os outros programas usam. Actualmente, existem duas versões do software: AutoDock 4 e Vina[40]. O AutoDock 4 está dividido ainda em dois sub programas: *autodock* faz o docking do ligando com um conjunto de grelhas que fazem a descrição do complexo resultante. O segundo, *autogrid*, faz os cálculos prévios para obter as grelhas que o autodock necessita

[10]Uma droga é caracterizada por uma molécula de pequenas dimensões.

para desempenhar as suas funções. O AutoDock Vina é diferente do AutoDock 4 no sentido de não efectuar o cálculo das grelhas de forma prévia mas sim instantaneamente, de forma automática, não guardando as grelhas em disco.

2.3.3.1 Aceleração do AutoDock com a GPU

Em 2010 foi abordada, à semelhança do que foi feito para o Megadock assim como para o PIPER, uma possível paralelização do AutoDock útilizando a GPU [18]. Neste caso a API útilizada foi o CUDA, sendo desenvolvida a versão 4.2.1 do AutoDock, que permite um speedup ao algoritmo genético que é utilizado, em 2 vezes, sendo este speedup semelhante ao do Vina sobre o AudoDock 4. No entanto o Vina difere desta versão 4.2.1 no sentido de acelerar um algoritmo de pesquisa local que permite a redução das operações necessárias para chegar ao resultado final, enquanto que esta versão visa acelerar o algoritmo genético que o AutoDock útiliza para determinar a posição óptimo do ligando encaixado com a proteína. Mais precisamente a função de fitness do algoritmo genético, que avalia a energia dos individuos da população constituida pelo algoritmo genético, sendo esta energia individual a soma da energia inter-molecular dos átomos do receptor e os do ligando.

Os autores deste projeto de aceleração do AutoDock consideraram duas opções: a primeira seria dedicar cada thread da GPU ao cálculo da energia total de um individuo ao que eles designaram *PerThread* e a segunda dedicar um bloco de threads da GPU para determinar essa energia total (*PerBlock*). A primeira solução garante a saturação da GPU (taxa de ocupação alta) para milhares de individuos e a segunda oferece a mesma saturação para centenas de individuos. A solução optada foi uma variante do *PerBlock*, em que existem 5 *kernels* de escrita/leitura para cada passo do algoritmo genético (inicialização de coordenadas atómicas do individuo, aplicar a torção até ao cálculo da energia interna) para as coordenadas atómicas respetivas a um individuo. De forma a eliminar a possibilidade de existir overhead, foi optado lançar um *kernel* e guardar as coordenadas atómicas na memória partilhada. Por este factor, a solução foi renomeada para *PerBlockCached*.

2.3.3.2 Conclusões

Foi demonstrado na subsecção 2.3.1 uma possibilidade de solução para acelerar o BiGGER, em que é implementado um conjunto de *kernels* para as diversas etapas, inclusive uma redução para determinar a solução óptimo e mapeamentos de dados para a GPU. A solução para o AutoDock oferece uma forma de poder acelerar o cálculo da função de score do BiGGER útilizando a memória partilhada da GPU.

Capítulo 3

Plano de trabalhos para a Elaboração da dissertação

3.1 Trabalho a desenvolver

3.1.1 Ciclo de desenvolvimento

Tendo em conta que no decorrer da elaboração não será desenvolvido código de raiz, mas sim optimizar o código existente, através de CUDA ou OpenCL, os esforços a adoptar durante a fase de elaboração seguirão um ciclo de desenvolvimento próprio para programação em GPUs. Este ciclo denomina-se APOD (*Assess*, *Parallelize*, *Optimize*, *Deploy*) [26] e consiste em quatro fases (figura 3.1):

1. ***Assess*** : Onde é feito um *assessment* ao estado atual do programa, em termos de performance. Nesta fase são determinados os pontos do programa onde este passa mais tempo a executar e identificar os bottlenecks de instruções, através de profilers para confirmar as identificações efectuadas. No caso da dissertação, é feito o profiling do ficheiro bigger.lpi presente na pasta bigger da biblioteca open source para o software que usa o BiGGER, Open-chemera e determinadas as funções que este ficheiro chama onde a fracção de tempo de execução é maior (*hotspots*) assim como zonas de código que atrasam a execução do programa (*bottlenecks*).

2. ***Parallelize*** : Após o *assessment* referido anteriormente estar concluido, procede-se para a fase de implementação do código para paralelizar os pontos encontrados na fase anterior. De acordo com *Six ways to SAXPY*, de Mark Harris [16], existem três possibilidades para implementar acelerações: usar bibliotecas aceleradas, diretivas OpenACC ou recorrer a linguagens para programação em GPUs, como CUDA ou OpenCL. Na elaboração da dissertação é pretendido abordar a primeira e terceira

possibilidades, no caso da terceira, existe a possibilidade de usar OpenCL pois é suportado pelo IDE Lazarus.

3. ***Optimize*** : Nesta terceira fase é pretendido aumentar a performance da solução base, inicialmente esta ultima tem de ser determinada executando o programa com um dataset de tamanho adequado. Da mesma forma é pretendido recorrer às técnicas descritas na subsecção 2.2.3 para maximizar a performance assim como às abstrações de outros programas relatados na subsecção 2.3.

4. ***Deploy*** : A última fase do ciclo consiste em confrontar a performance obtida com as expectativas fundamentadas no início do ciclo. Se os resultados obtidos não corresponderem ao speedup potencial registado na fase inicial, é necessário voltar à fase *Assess*, recomeçando o ciclo.

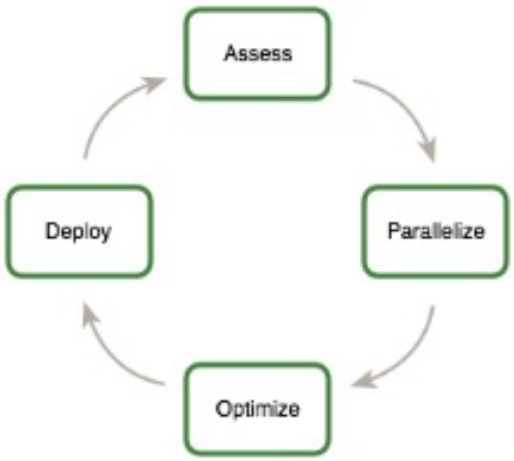

Figura 3.1: Diagrama do ciclo de desenvolvimento APOD[26].

3.1.2 Profiling

De forma a confirmar as assumpções sobre as zonas de código a optimizar para execução na GPU, será necessário fazer uma averiguação sobre o custo de cada etapa do mesmo em termos de número de chamadas e tempo total de computação, através de um profiler. Na tentativa de fazer esta actividade, foi verificado que no código fonte existem chamadas para funções que estão *deprecated* com uma versão actual do Free Pascal, tornado a compilação e o profiling com ferramentas actuais difícil. Neste contexto uma das primeiras tarefas a desempenhar será a actualização do código fonte seguido do profile à performance do BiGGER. O profiling será feito aos ficheiros relacionados com o funcionamento deste último, pois existem elementos do Open-chemera que não são importantes para o mesmo, como por exemplo o ficheiro *chemera.lpi* é o que é necessário compilar e executar no Lazarus para a parte gráfica do Open-Chemera, não influenciando a parte do BiGGER,

que é independente. Para executar a componente do BiGGER, é necessário passar como parâmetro um conjunto de ficheiros xml com a informação sobre os átomos das proteínas. No caso do Lazarus é possível recorrer a duas formas principais para fazer profiling a um programa [34]:

- gprof : Pode-se utilizar o gprof para fazer profiling em Lazarus, gerando um ficheiro de texto com os dados necessários para averiguar quais as funções do ficheiro bigger.lpi que podem representar oportunidades de paraleliação.
- LazProf: IDE de profiling para o Lazarus, que funciona complementada com o FPProfiler.

A abordagem LazProfiler requer uma instalação complexa mas é a alternativa cujo profiling mostra os resultados com qualidade superior, sendo necessário apenas ordenar na interface do programa a execução do programa. A alternativa é usar o gprof, que é suportado pelo compilador FreePascal pelo que não requer uma instalação com o nível de complexidade da do LazProfiler, no entanto os resultados que são obtidos não assumem a mesma qualidade deste último.

3.1.3 Possibilidades de optimização

A biblioteca *open-source* para docking de proteínas usada no BiGGER, Open-Chemera, encontra-se implementada em Free Pascal (97.6% do código total)[11]. O trabalho a realizar na elaboração é focado sobre o package *docking*, mais precisamente às unidades *bogie.pas* e *dockdomains.pas*. O *bogie.pas* consiste no módulo que trata a componente geométrica do docking e na unidade *dockdomains.pas* são determinados os dominios nos três eixos para a simulação da docagem geométrica.

O trabalho poderá abarcar a paralelização de unidades adicionais presentes no pacote *docking* o que só garante melhorias adicionais à performance da biblioteca Open-Chemera.

Face à possibilidade de não existir nenhuma versão do CUDA para programar paralelizações em Free Pascal, poderá ser optado por desenvolver as optimizações usando OpenCL, que é suportada pelo Lazarus. Este ultimo é o IDE a utilizar durante o trabalho de elaboração. É possível, no entanto, implementar uma biblioteca acelerada (ficheiro dll) para o BiGGER com CUDA.

Os pontos de optimização focam-se nos seguintes:

- **Criação das grelhas tri-dimensionais** : Tal como foi mencionado na subsecção 2.1.3.1, um dos passos iniciais do BiGGER consiste na criação de duas grelhas tridimensionais de booleanos, para cada um dos elementos do par, que assumem valor 1 ou 0 se a posição respetiva na grelha corresponde a uma posição atómica da proteina. Neste contexto, é possível determinar a região de superficie de cada proteina,

[11] É possível obter a biblioteca e o código fonte do BiGGER através do repositório presente no Github [30].

sendo esta definida pelo conjunto de todos os nós da matriz que se encontram a 1. Na versão sequencial do BiGGER, a região de superficie das proteínas é determinada fazendo um *shift* da cópia da grelha uma casa para o lado em cada uma das 26 direções possíveis, aplicando uma operação lógica XOR entre os nós originais e os que foram deslocados. É possível acelerar este procedimento implementando um kernel em que é mapeada a grelha original na GPU e distribuindo cada uma das direções pelos SMs.

- **Pesquisa de sobreposições** : De forma a poder avaliar a correspondência entre as superficies dos pares, para cada rotação a ser tratada é aplicada uma operação de shift, à semelhança do passo anterior, mas desta vez com as matrizes originais de ambos os elementos do par. A determinação de sobreposições é feita determinando o número de sobreposições entre os nós de superficie de ambas as proteínas. Na versão sequencial, para determinar as sobreposições é necessário aplicar uma operação lógica AND entre os registos de memória que contêm as matrizes de superficie. Por sua vez situações em que existam sobreposições entre nós *core* são descartadas como candidatos, no entanto situações em que são sobrepostos um nó *core* com um nó de superficie não são. É feita uma rotação entre os pares de proteínas em um ângulo de 15º por omissão, repetindo os processos de criação das grelhas, translação por shift e correspondência de superficie até todas as possibilidades num espaço de 6 dimensões estarem cobertas. Este procedimento assume as mesmas condições do que a etapa do Megadock correspondente à rotação do ligando, neste o processo para cada átomo (representado como um nó da grelha do ligando) é independente, sendo mapeado para a GPU. Neste contexto é possível abordar uma optimização em que os nós da grelha são mapeados para a GPU. Cada um dos nós serão processados por um core da GPU e o AND passa a ser aplicado entre registos da GPU em vez de ser em registos da CPU.

- **Filtragem de Candidatos**: O BiGGER contem um passo de filtragem de candidatos, em que 10^9 possíveis contactos entre as proteínas são filtrados, sendo eliminados no final deste processo 99.999% destes. O processo consiste em usar uma combinação de critérios, sendo esta combinação composta inicialmente por complementaridade de superficie, seguida pelo número de contactos favoráveis entre pares de aminoácidos. A avaliação é determinada para cada candidato e esta é comparada com as avaliações presentes numa tabela com as 1000 melhores encontradas até ao momento. Se a avaliação mostrar que a solução é pior do que a do último elemento da tabela, a solução é descartada. Em caso de ser melhor, a solução é guardada na tabela e o último elemento da lista é descartado. Uma optimização a considerar para este passo será distribuir a computação relacionada com as avaliações da complementaridade de superficie e número de contactos em dois *kernels*. A tabela com os melhores candidatos poderá ser guardada em memória partilhada na GPU, pois é nesta memória que os acessos são mais rápidos do que na memória global. De notar

que nesta fase são feitos muitos acessos à tabela para alterar o conteúdo desta, pelo que é importante que estes sejam rápidos.

- **Scoring**: Na fase de scoring do BiGGER, são considerados quatro termos individuais na avaliação de cada solução encontrada: complementaridade de superficie, contactos entre ámino-acidos, electroestática e solvatação. Estes quatro termos são combinados numa função de scoring global, sendo esta última calculada para cada um dos candidatos, à semelhança do que é feito para o PIPER sobre os seus conjuntos de quoficientes na função de score. Poderá ser necessário aplicar uma optimização em que os procedimentos associados quatros termos são distribuidos por 4 SMs, pelo que esta optimização poderá ser semelhante à correspondente ao uso do segundo kernel no conjunto de optimizações para o PIPER em 2014.

3.2 Metodologia de Avaliação

No final de cada iteração do ciclo, serão feitas análises de performance ao BiGGER, onde serão avaliadas ambas redução de tempo de execução e exactidão dos resultados face à versão actual do programa. A exactidão poderá ser feita submetendo a versão optimizada do BiGGER aos testes aplicados à versão sequencial, procurando coincidencias nos resultados. Por sua vez a redução do tempo de execução poderá ser avaliada comparando os valores de speedup da solução desenvolvida em cada ciclo com o speedup máximo teórico. Segundo o guia de boas práticas da NVIDIA [26], os conceitos de escalabilidade forte e fraca são necessários para determinar o quanto um programa pode ser paralelizado. No entanto usar um ou outro depende das características do problema que o programa resolve. É recomendado considerar a escalabilidade forte quando o tamanho do problema é constante e o tempo de solução decresce à medida que o número de processadores aumenta. Em alternativa a escalabilidade fraca é considerada quando o tamanho do problema varia à medida que o número de processadores aumenta. No caso do BiGGER e da modelação das interações entre moléculas, o tamanho destas pode variar com a complexidade da molécula, mas o tamanho é constante no decorrer da execução do programa. Da mesma forma o tamanho das grelhas criadas inicialmente não é variável assim como o número de direções para onde a operação de shift entre as grelhas, pelo que é aplicável a escalabilidade forte a este problema. Neste contexto o speedup máximo teórico pode ser determinado após a fase *Assess* em cada ciclo estar concluida, recorrendo à lei de Amdahl [3], em que o speedup máximo teórico é dado pela fórmula $S = 1/(1 - P)$, sendo S o speedup máximo teórico que o programa pode alcançar e P a fração sequencial do código que pode ser paralelizado. O speedup corrente, por sua vez, pode ser determinado através da fórmula $S = t1/tp$, sendo $t1$ o tempo de execução da versão do BiGGER que usa apenas a CPU e tp o mesmo tempo para a versão que usa ambos CPU e GPU.

3.3 Plano de Trabalhos

O plano de trabalhos consistirá em quatro iterações do ciclo APOD, para cada um dos pontos de optimização apresentados na sub-secção 3.1.3, tendo cada um dos ciclos uma carga de horária total de 224 horas de trabalho nos dias úteis (tabela 3.1). Esta carga horária será repartida pelas quatro etapas do ciclo APOD, em que a primeira etapa do ciclo terá uma carga horária de 24 horas, a segunda e terceira etapas 80 e a quarta etapa, que é concorrente com um bloco dedicado à escrita do relatório, tem uma duração de 40 horas. Neste contexto o primeiro ciclo será iniciado quando o ano lectivo iniciar a 10 de Setembro de 2018, em que será feito o trabalho respetivo à fase *Assess*. O programa será submetido a um profiler e serão confirmados os pontos de optimização identificados na fase de preparação da tese. Após a conclusão da fase anterior, é dado início às fases *Parallelize* e *Optimize*, cada uma num bloco de actividade com uma duração de 20 dias úteis de trabalho para ambas as fases. Nestas duas fases serão implementadas as optimizações para os problemas de performance indicados na fase anterior. Na eventualidade de ocorrerem optimizações que ficaram pendentes, o esforço será transferido para os blocos *Parallelize* e *Optimize* da iteração seguinte. O passo final do ciclo, *Deploy*, será acompanhado com a actividade de escrita do relatório. A primeira iteração será terminada a 17 de Outubro de 2018.

A segunda iteração do ciclo será iniciada dia 18 de Outubro de 2018, onde serão repetidos os procedimentos do ciclo anterior para a segunda versão optimizada do algoritmo, e terminará dia 26 de Novembro de 2018. A terceira iteração terá inicio a 27 de Novembro de 2018, A quarta iteração do ciclo será iniciada a 4 de Janeiro de 2019, terminando em 11 de Fevereiro de 2019, o que garante uma margem de atrasos relacionados com as fases *Optimize* e *Parallelize* de todas as iterações de duas semanas até ao inicio do mês onde será procedida à entrega da dissertação. Na eventualidade de o trabalho de implementação não se encontrar a correr dentro do plano, o espaço de tempo entre o fim da quarta iteração e a data de entrega pode ser adaptado para integrar uma quinta iteração, a decorrer até dia 20 de Março de 2019. No entanto o esforço nesta iteração terá de ser complementado com a elaboração do relatório de dissertação. Se o contrário se verificar e o trabalho estiver dentro do prazo, após o final do quarto ciclo de desenvolvimento, será focada a elaboração do relatório de dissertação até à data de entrega deste, que segundo o calendário publicado para o ano lectivo de 2018-2019, é dia 25 de Março de 2019. A fase de elaboração do relatório terá uma carga horária total de 233 horas. Este plano consome na totalidade 1121 horas de trabalho (apenas dias úteis), garantindo aproximadamente 40 ECTS de esforço. No entanto como a elaboração da dissertação será acompanhada com uma unidade curricular pendente, reduzindo a contribuição para 34 ECTS, sendo superior ao 30 ECTS exigidos para a cadeira correspondente à fase de elaboração da tese.

	Inicio	Fim	Carga de trabalho (horas)
Iteração 1	**10/9/2018**	**17/10/2018**	**224**
Assess 1	10/9/2018	12/8/2018	24
Parallelize 1	13/9/2018	26/9/2018	80
Optimize 1	27/9/2018	10/10/2018	80
Deploy e Relatório 1	11/10/2018	17/10/2018	40
Iteração 2	**18/10/2018**	**26/11/2018**	**224**
Assess 2	18/10/2018	22/10/2018	24
Parallelize 2	23/10/2018	5/11/2018	80
Optimize 2	6/11/2018	19/11/2018	80
Deploy e Relatório 2	20/11/2018	26/11/2018	40
Iteração 3	**27/11/2018**	**3/1/2019**	**224**
Assess 3	27/11/2018	29/11/2018	24
Parallelize 3	30/11/2018	13/12/2018	80
Optimize 3	14/12/2018	27/12/2018	80
Deploy e Relatório 3	28/12/2018	31/12/2018	40
Iteração 4	**4/1/2019**	**11/2/2019**	**224**
Assess 4	4/1/2019	7/1/2019	24
Parallelize 4	8/1/2019	21/1/2019	80
Optimize 4	22/1/2019	4/2/2019	80
Deploy e Relatório 4	5/2/2019	11/2/2019	40
Elaboração do relatório	**12/2/2019**	**25/3/2019**	**233**

Tabela 3.1: Tabela cronograma do plano de trabalho.

3.4 Profiling na fase Inicial

Nesta fase inicial do desenvolvimento das optimizações,l foi feita uma análise de performance ao BiGGER na versão sequêncial conforme foi indicado no plano de trabalhos na fase de preparação. Como tal a biblioteca open-chemera foi submetida a um conjunto extenso de testes a variar parametros específicos como o nº de átomos do ligando/receptor, a sobreposição mínima das superfícies moleculares e o nº de rotações no docking como está indicado na tabela. A ferramenta usada para a actividade foi a única que funcionou de diversas alternativas que podem ser usadas para fazer profiling a um programa que se encontra em FreePascal - Valgrind/Callgrind [34]. Este programa devolve resultados que incluem o número total de chamadas efectuadas para uma dada função e duas métricas de avaliação de custo temporal: self cost e inclusive cost. Para efeitos de amostragem de resultados, foi considerada principalmente a métrica self cost pois este refere-se à fração de tempo de execução que é passado dentro da função, enquanto que o inclusive cost inclui as frações de tempo das funções chamadas pela função em questão. No caso deste programa, após a execução do profile é gerado um ficheiro especifico com os resultados que podem ser consultados através de um segundo programa chamado KCacheGrind.

Sólido geométrico	NátomosProbe	NátomosTarget	MinOverlap (Sobreposição minima)	Rotações
Cubo	216	512	0	100
Esfera	1728	3375	500	1000
-	6859	19683	1000	1500
-	19683	32678	2000	2000
-	-	-	200000	6000
-	-	-	-	15000

Tabela 3.2: Parametros de teste e variação.

3.4.1 Sobre os parametros de teste

3.4.1.1 Complexos de proteínas usados

Para que seja possível determinar o impacto do formato dos pares na performance do BiGGER, apenas foram considerados pares ligando-receptor que assumam formatos uniformes, sendo considerados apenas cubos e esferas para os testes. Adicionalmente, não foram testadas situações em que, por exemplo, o ligando tem forma cúbica e o receptor esferica, pois isto implica duplicar a quantidade de testes necessários. Em termos de número de átomos, não foram aplicados testes em que o número de átomos do receptor seja menor do que o número de átomos do ligando. Este cenário não corresponde a uma situação real assim como introduz esforços adicionais ao BiGGER que são desnecessários. Conforme foi referido no capítulo 2 o BiGGER resolve o problema de docking através de um conjunto de passos, entre eles a digitização de grelhas tridimensionais. É criada e digitizada uma vez a grelha do receptor e tantas vezes como rotações pedidas a grelha do ligando.

3.4.1.2 Sobreposição mínima e rotações

Foi pretendido estudar o impacto da variação da sobreposição minima na fase de scoring do algoritmo, sendo que esta foi incrementada de forma gradual entre o valor nulo e um valor excessivamente grande para uma situação real. Sobre as rotações, foi aplicado o mesmo procedimento, sendo que neste caso o número de rotações foi variado gradualmente entre 100 rotações, valor baixo para uma situação real e 6000/15000 que são usados habitualmente num docking.

3.4.2 Scripts desenvolvidos para a fase de profiling

Tendo em conta as variações de parametros referidas na tabela 3.2, o leitor fica sensibilizado para a extensão de cenários de teste possíveis de realizar, produto do nº de combinações possíveis. Para cada um dos pares de número de átomos (216 e 512 por exemplo), foram feitos 5 testes a variar o minOverlap e dentro de cada uma destas variações 6 testes a variar as rotações. No total foram realizados 120 testes, em que para cada um foram necessários aplicar passos manuais como:

1. A criação através do elemento dockprep do BiGGER do ficheiro de job (xml) para o docking, sendo este último completado pela execução do BiGGER.

2. Alteração dos campos necessários do ficheiro xml para o docking estar dentro do cenário pretendido.

3. Executar o teste por linha de comandos

4. Guardar os resultados numa folha excel

Todo este processo para cada um dos 120 testes fez com que o processo de realizar o profile ao BiGGER tenha demorado excessivamente. Como tal foram implementados 3 scripts que automatizam/ facilitam o procedimento de testar quer o programa inicial, quer as optimizações efectuadas. Adicionalmente têm o efeito de diminuir o erro humano na execução dos passos referidos. O primeiro script (dummyJobCreator) desenvolvido em python automatiza os dois primeiros passos do procedimento. O programa recebe como input um conjunto de argumentos que descrevem o cenário pretendido, criando como output um ficheiro xml de job com o mesmo formato dos criados pelo dockprep para a execução do BiGGER completar. Após a invocação do script os seguintes argumentos deverão ser indicados pela ordem referida:

1. probe;

2. sólido geometrico do probe (cubo/esfera);

3. número de átomos no probe no caso deste ser uma esfera ou o tamanho do lado se este for cubo;

4. raio de cada átomo presente no probe (recomendado ser 1.0 para o sólido ser uniforme);

5. target;

6. sólido geometrico do target (cubo/esfera);

7. numero de átomos no target no caso deste ser uma esfera ou o tamanho do lado se for cubo;

8. raio de cada átomo presente no target (recomendado ser 1.0 para o sólido ser uniforme);

9. número de rotações;

10. valor do minOverlap.

O segundo script é meramente um ficheiro com uma sequência de comandos para um terminal linux. O terceiro programa que similarmente ao primeiro foi desenvolvido em

python cria gráficos lineares com os valores de self cost obtidos pela execução do bigger com o valgrind/callgrind para o conjunto de funções mais problemáticas no open-chemera. O programa lê uma matriz com estes valores, gerando três agrupamentos de gráficos. O primeiro engloba apenas os gráficos em que foi variado apenas o número de átomos presentes no ligando e no receptor, um segundo onde os gráficos variam só a sobreposição e o restante onde é variada apenas a rotação. Exemplos de output do script podem ser observados nas figuras x, y, e z. Estes scripts automatizam a grande maioria do workflow necessário para aplicar o profilling. Falta no entanto automatizar a introdução/actualização dos valores de self cost na matriz de onde são gerados os gráficos, tarefa esta que até ao momento é difícil de concretizar.

3.4.3 Conclusões

Após a análise dos dados obtidos nesta fase inicial, foram tiradas algumas conclusões. Verificou-se que os problemas de performance estão localizados nas unidades partilhadas do open-chemera que este usa para resolver os dockings (pasta sharedUnits), mais precisamente as unidades geomhash.pas, na função isInnerPoint e geomutils.pas, na função de distancia euclidiana. Dos resultados do profiling, é possível inferir:

1. O tempo de execução gasto nas funções das unidades geomhash e geomutils ocupam a maioria ou até mesmo a totalidade do tempo gasto na fase de digitização. Sendo que este tempo cresce à medida que o número de rotações e o tamanho das estruturas aumenta. Neste contexto pode ser inferido que estes parametros têm peso na fase de digitização, que por sua vez ocupa a maior parte do tempo de execução do algoritmo. Pelo que é necessário paralelizar as funções isInnerPoint e distance;

2. Pelos gráficos em que é variado o minOverlap, é possível observar que a curva mantem-se constante para a maior parte das situações, variando apenas a curva respetiva à função addModel, em que para um minOverlap nulo os valores de self cost associados a esta função têm valores superiores aos restantes valores de minOverlap. Neste contexto a curva assume-se como descrescente em relação ao self-cost. Este comportamento da curva é justificável pois para valores de minOverlap baixo, o BiGGER pode passar uma parcela de tempo considerável a preencher uma lista de modelos solução com soluções de poses em que o valor de score associado à sobreposição é maior ou igual a 0 até preencher por completo a lista. Se a sobreposição minima for aumentada, a probabilidade do algoritmo encontrar soluções adequadas para reservar na lista baixa, sendo que a lista já não é preenchida e o self cost da função add model baixa, encontrando-se nulo em situações onde o minOverlap têm o valor máximo establecido para a bateria de testes. Como tal infere-se que o parametro sobreposição minima tem peso apenas para a fase de scoring do algoritmo e não para a fase de digitização. Por consequência a paralelização da função addModel passa a ser um objetivo secundário da dissertação;

3. Existe ainda a função setExtremes da unidade dockdomains, cujos valores de self cost registados não são tão consideráveis como os valores das funções referidas nos pontos anteriores. A paralelização desta função constitui, à semelhança da função addModel, um objetivo secundário. Sobre os valores de self cost, acrescenta-se que as curvas do setExtremes nos gráficos são constantes em todos os gráficos nos 3 agrupamentos, pelo que o comportamento desta função não é afectada por nenhum dos parametros. No entanto é possível notar a diferença dos valores de self cost nos testes em que foram usados complexos com formato cúbico para os com formato esférico. Deste modo conclui-se que é o formato dos complexos que afecta esta função.

4. Este último ponto diz respeito aos limites máximos de valores de self cost observados para as funções referidas. Para os cenários de teste mais exigentes verificou-se que o self cost do BiGGER (fração de tempo gasto na execução do bigger) encontrava-se a 96% e os restantes 4% dizem respeito às bibliotecas externas auxiliares, sendo que esta fracção de tempo não pode exceder ou alcançar os 100%. Com base neste facto e na análise dos gráficos gerados, é possível estimar a assimptota horizontal das curvas associadas às funções de distancia e isInnerPoint, que está situada entre os 40% e 45% para a primeira e 35% e 40% para a última. Esta inferência poderia ser comprovada introduzindo cenários de teste que excedem a exigência máximo considerada para esta fase (aumentar o tamanho dos pares e mais rotações, com overlap minimo nulo), certamente haveria um resultado em que a fração do BiGGER alcançasse os 99%. No entanto para os cenários actuais mais exigentes, o tempo de execução de cada um demorou 8 horas a executar, o que faz com que executar um teste ainda mais exigente possa demorar até 24H a terminar, não sendo compensável face à alternativa de estimar os limites.

Bibliografia

[1] R. Almeida, S. Dell'Acqua, L. Krippahl, J. Moura e S. Pauleta. "Predicting Protein-Protein Interactions Using BiGGER: Case Studies". Em: 21 (ago. de 2016), p. 1037.

[2] R. M. Almeida, S. Dell'Acqua, L. Krippahl, J. J. Moura e S. R. Pauleta. "Predicting protein-protein interactions using bigger: Case studies". Em: *Molecules* 21.8 (2016), p. 1037.

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

[4] R. Chen e Z. Weng. "Docking unbound proteins using shape complementarity, desolvation, and electrostatics". Em: *Proteins: Structure, Function, and Bioinformatics* 47.3 (2002), pp. 281–294.

[5] R. Chen, L. Li e Z. Weng. "ZDOCK: an initial-stage protein-docking algorithm". Em: *Proteins: Structure, Function, and Bioinformatics* 52.1 (2003), pp. 80–87.

[6] *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone.

[7] *cuFFT*. 2018. URL: https://developer.nvidia.com/cufft.

[8] *Docking (molecular)*. URL: https://en.wikipedia.org/wiki/Docking_(molecular).

[9] C. Dominguez, R. Boelens e A. M. Bonvin. "HADDOCK: a protein- protein docking approach based on biochemical or biophysical information". Em: *Journal of the American Chemical Society* 125.7 (2003), pp. 1731–1737.

[10] *Equipa liderada por português descobre proteína do cérebro que protege de Alzheimer*. URL: https://observador.pt/2018/06/29/ha-uma-proteina-do-cerebro-que-pode-proteger-contra-a-doenca-de-alzheimer/.

[11] D. Fischer, S. L. Lin, H. L. Wolfson e R. Nussinov. "A geometry-based suite of moleculardocking processes". Em: *Journal of Molecular Biology* 248.2 (1995), pp. 459–477.

[12] H. A. Gabb, R. M. Jackson e M. J. Sternberg. "Modelling protein docking using shape complementarity, electrostatics and biochemical information1". Em: *Journal of molecular biology* 272.1 (1997), pp. 106–120.

[13] M. W. Gonzalez e M. G. Kann. "Protein interactions and disease". Em: *PLoS computational biology* 8.12 (2012), e1002819.

[14] *GPGPU.org*. URL: http://gpgpu.org/about.

[15] *GPU Aplications Catalog*. URL: https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/.

[16] M. Harris. *Six Ways to SAXPY*. URL: https://devblogs.nvidia.com/six-ways-saxpy/.

[17] H. Inbal, M. Buyong, W. Haim e N. Ruth. *Principles of docking: An overview of search algorithms and a guide to scoring functions*. Vol. 47. 4, pp. 409–443. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/prot.10115.

[18] S Kannan e R Ganji. "Porting AutoDock to CUDA [J]. Evolutionary Computation (CEC)". Em: *2010 IEEE Congress on*, pp. 1–8.

[19] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. A. Friesem, C. Aflalo e I. A. Vakser. "Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques." Em: *Proceedings of the National Academy of Sciences* 89.6 (1992), 2195–2199. DOI: 10.1073/pnas.89.6.2195.

[20] L. Krippahl. *Integrating protein structural information*. 2003.

[21] R. Landaverde e M. C. Herbordt. "GPU optimizations for a production molecular docking code". Em: *... IEEE conference on high performance extreme computing. IEEE Conference on High Performance Extreme Computing*. Vol. 2014. NIH Public Access. 2014.

[22] E. Lindholm, J. Nickolls, S. Oberman e J. Montrym. "NVIDIA Tesla: A unified graphics and computing architecture". Em: *IEEE micro* 28.2 (2008).

[23] G. M. Morris. *AutoDock*. URL: http://autodock.scripps.edu/.

[24] G. M. Morris, D. S. Goodsell, R. S. Halliday, R. Huey, W. E. Hart, R. K. Belew e A. J. Olson. "Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function". Em: *Journal of computational chemistry* 19.14 (1998), pp. 1639–1662.

[25] J. Nickolls e W. J. Dally. "The GPU computing era". Em: *IEEE micro* 30.2 (2010).

[26] NVIDIA. *CUDA C BEST PRACTICES GUIDE*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[27] NVIDIA. *CUDA C PROGRAMMING GUIDE*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[28] NVIDIA. *NVIDIA Tesla V100 GPU architecture*. URL: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[29] M. Ohue, T. Shimoda, S. Suzuki, Y. Matsuzaki, T. Ishida e Y. Akiyama. "MEGADOCK 4.0: an ultra–high-performance protein–protein docking software for heterogeneous supercomputers". Em: *Bioinformatics* 30.22 (2014), pp. 3281–3283. URL: http://dx.doi.org/10.1093/bioinformatics/btu532.

[30] *Open Chemera repository*. URL: https://github.com/lkrippahl/Open-Chemera.

[31] *OpenCL*. URL: https://developer.nvidia.com/opencl.

[32] P. N. Palma, L. Krippahl, J. E. Wampler e J. Moura. "BIGGER: A new (soft) docking algorithm for predicting protein interactions". Em: 39 (jun. de 2000), pp. 372–84.

[33] B. G. Pierce, Y. Hourai e Z. Weng. "Accelerating protein docking in ZDOCK using an advanced 3D convolution library". Em: *PloS one* 6.9 (2011), e24657.

[34] *Profiling*. URL: http://wiki.lazarus.freepascal.org/Profiling.

[35] Ritchie, D. W., Venkatraman e Vishwesh. *Ultra-fast FFT protein docking on graphics processors | Bioinformatics | Oxford Academic*. 2010. URL: https://academic.oup.com/bioinformatics/article/26/19/2398/229220.

[36] T. Shimoda, S. Suzuki, M. Ohue, T. Ishida e Y. Akiyama. "Protein-protein docking on hardware accelerators: comparison of GPU and MIC architectures". Em: *BMC systems biology*. Vol. 9. 1. BioMed Central. 2015, S6.

[37] G. R. Smith e M. J. Sternberg. "Prediction of protein–protein interactions by docking methods". Em: *Current opinion in structural biology* 12.1 (2002), pp. 28–35.

[38] J. E. Stone, D. Gohara e G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". Em: *Computing in Science and Engineering* 12.3 (2010), pp. 66–73.

[39] B. Sukhwani e M. C. Herbordt. "GPU acceleration of a production molecular docking code". Em: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM. 2009, pp. 19–27.

[40] O. Trott e A. J. Olson. "AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading". Em: *Journal of computational chemistry* 31.2 (2010), pp. 455–461.

[41] I. A. Vakser. "Protein-protein docking: From interaction to interactome". Em: *Biophysical journal* 107.8 (2014), pp. 1785–1793.

[42] N. Wilt. *The CUDA handbook: a comprehensive guide to GPU programming*. Addison-Wesley, 2013.