



Ricardo Alexandre do Rosário Ribeiro

Licenciado em Engenharia Informática

Protein docking GPU acceleration

Relatório intermédio para obtenção do Grau de Mestre em
Engenharia Informática

Orientador: Ludwig Krippahl, Professor Associado,
FCT-UNL

Co-orientador: Hervé Paulino, Professor Associado,
FCT-UNL



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2018

RESUMO

Na área científica da Bio-Informática, 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 ao CPU, em 2018, o que em termos de trabalho/tempo é ineficiente. Um desses métodos é o BiGGER, criado pelo prof. Ludwig Krippahl, prof. Nuno Palma e outros integrados no projeto. 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 na área da medicina, onde são desenvolvidas formas de proteger o Homem de doenças neuronais assim como desenvolve o desenho e concepção de drogas assistido por computador.

Para resolver a ineficiência referida, as ferramentas para docking foram optimizadas para usar o 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 o GPU da máquina em que corre o algoritmo para auxiliar o 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 open-chemera 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, Bio-Informática, BiGGER

ABSTRACT

The dissertation must contain two versions of the abstract, one in the same language as the main text, another in a different language. The package assumes that the two languages under consideration are always Portuguese and English.

The package will sort the abstracts in the appropriate order. This means that the first abstract will be in the same language as the main text, followed by the abstract in the other language, and then followed by the main text. For example, if the dissertation is written in Portuguese, first will come the summary in Portuguese and then in English, followed by the main text in Portuguese. If the dissertation is written in English, first will come the summary in English and then in Portuguese, followed by the main text in English.

The abstract should not exceed one page and should answer the following questions:

- What's the problem?
- Why is it interesting?
- What's the solution?
- What follows from the solution?

Keywords: Keywords (in English) ...

ÍNDICE

Lista de Figuras	ix
Lista de Tabelas	xi
Listagens	xiii
Glossário	xv
Siglas	xvii
1 Introdução	1
1.1 Enquadramento e motivação	1
1.2 Conceito de docking	2
1.3 Problema	3
1.4 Solução	4
1.5 Contribuições	4
1.6 Estrutura deste documento	4
2 Estado da arte	7
2.1 Docking	7
2.1.1 Conceitos de docking em relação à rigidez da superfície	7
2.1.2 Métodos usados em docking	8
2.1.3 Ferramentas para Docking	11
2.2 O GPU	13
2.2.1 Arquitectura e Modelo de Execução	13
2.2.2 Modelos Base de Programação	17
2.2.3 Optimizações	18
2.3 Docking em GPU	20
2.3.1 Megadock	20
2.3.2 PIPER	22
2.3.3 AutoDock	26
3 Plano de trabalhos para a Elaboração da dissertação	29
3.1 Trabalho a desenvolver	29

ÍNDICE

3.1.1	Ciclo de desenvolvimento	29
3.1.2	Profiling	30
3.1.3	Possibilidades de otimização	31
3.2	Metodologia de Avaliação	33
3.3	Plano de Trabalhos	33
	Bibliografia	35

LISTA DE FIGURAS

1.1	Representação gráfica do docking[7].	3
2.1	Fluxograma sobre o geometric hashing. Tirado de [9]	10
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 <i>core</i> e a núvem com tracejado contínuo representa o corte associado à esfera de van der waals com a proteína localizada ao centro [27].	11
2.3	Esquema do SM para o GV100[24].	14
2.4	Esquema da arquitetura do GPU, mais precisamente do Volta GV100[24]. Esta é a arquitetura mais recente que NVIDIA lançou no mercado.	15
2.5	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 possibilidade de ser tri-dimensional, o mesmo se aplica à grelha [23].	17
2.6	Etapas do processo de docking no Megadock. As etapas a cinzento foram aceleradas por GPU[31].	21
2.7	Esquema para o primeiro kernel introduzido na versão GPU2014 [18].	24
2.8	Esquema para o segundo <i>kernel</i> introduzido na versão GPU2014 [18].	24
2.9	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. [18]	25
2.10	Proporções de tempo gasto na execução para a versão CPU otimizada e a versão de 2009 que usa GPUs[18]	25
3.1	Diagrama do ciclo de desenvolvimento APOD[22].	30

LISTA DE TABELAS

2.1	Tabela de capacidades computacionais das arquiteturas NVIDIA mais recentes. Adaptado de [24].	16
-----	---	----

LISTAGENS

GLOSSÁRIO

bottleneck	Um <i>bottleneck</i> ..
complementaridade	A complementaridade..
correlação	A correlação..
dessolvatação	A dessolvatação.
hotspot	Um <i>hotspot</i> ..

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 Compiler.
PDB	Protein Database.
PDI	Protein-Drug Interaction.
PPI	Protein-Protein Interaction.
PSC	Pair-based Surface Complementarity.
SM	Stream Multiprocessor.
SP	Scalar Processor.

SIGLAS

XOR Exclusive Or.

INTRODUÇÃO

1.1 Enquadramento e motivação

As proteínas não funcionam de forma isolada, de acordo com [10], 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 benefício a proteção contra a doença de Alzheimer [8]. 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 [28] 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[14]. 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[27], a decorrer na fase de elaboração da dissertação, em que o GPU será utilizado 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 bio-informática e de informática. Bio-informática no sentido de envolver conceitos relacionados com o estudo das interações entre proteínas e informática devido à parte do uso do GPU para reduzir o tempo de execução do BiGGER.

1.2 Conceito de docking

De acordo com Halperin et al (2002) [14], 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 graficamente na figura 1.1. Existem duas vertentes de docking, o docking acoplado (*bound docking*) e docking não-acoplado (*unbound docking*). Segundo Vakser (2014) [36], 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 utilizada, pois as previsões sobre complexos formados por estruturas isoladas garantem utilidade científica. 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[32].

Docking de proteínas por sua vez consiste em prever a estrutura tri-dimensional do complexo de proteínas através das coordenadas atômicas 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 parametros a complementaridade de superfície ou de forma. Ambas as complementaridades são determinadas por uma abstração de ambas as proteínas numa grelha tri-dimensional, determinando para cada célula da grelha se existe correspondência com uma coordenada atômica 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 parametros 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].

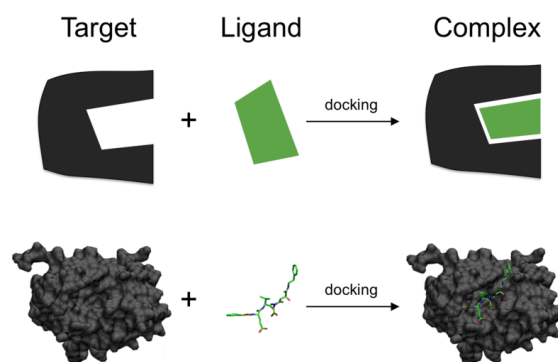


Figura 1.1: Representação gráfica do docking[7].

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.

O 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 o GPU também é adequado para processamentos alternativos, que, tal como a renderização de gráficos 3D, são igualmente intensivos demais para o 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 no GPU alcançar valores de *speedup* superiores em relação a uma implementação para CPU otimizada, em situações em que o problema a resolver pelo programa envolva computações extensas e repetidas. O uso do GPU para este tipo de processamentos é referido como *General-purpose computation on graphics processing units* (GPGPU)[11]. Exemplos de aplicações podem variar deste cálculo financeiro até aplicações bio-informáticas, como é o caso do docking. É possível encontrar um panorama detalhado sobre as aplicações da computação de alta performance, sobre forma de catálogo[12]. 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 num programa que use FFT é gasto nestas etapas[18] [31], o que faz com que a performance destes programas dependa da eficiência

da biblioteca. No artigo sobre o trabalho do Megadock em GPUs[31], 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 tri-dimensionais necessitar de ser potência de 2 [27], enquanto que para o BiGGER as matrizes devem ter a largura da molécula, pelo que as matrizes deste ultimo são de menores dimensões de espaço e memória.

1.4 Solução

Como solução é pretendido implementar a aceleração do algoritmo BiGGER, executando este no GPU. Esta aceleração terá de trazer à versão futura do BiGGER ganhos de speedup consideráveis em relação à versão actual que neste momento recorre apenas ao CPU. Como tal será necessário, inicialmente, identificar as zonas de código do BiGGER que demoram mais tempo e para cada uma destas definir abstrações, com base em exemplos existentes, para otimizar o tempo de execução da zona. Após estarem definidas as zonas de código que mais afectam o tempo de execução na versão sequencial e respectivas optimizações, estas últimas serão implementadas por uma linguagem de programação específica, passando as zonas de código a serem executadas de forma paralelizada no 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 no GPU, assim como as respectivas implementações e análise ao desempenho. O código poderá desenvolvido de forma a que possam ser aplicadas manutenções. 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 de um algoritmo de docking que utiliza métodos mais eficientes para determinar a interação entre proteínas do que o uso de 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 é feita a uma introdução sobre os conceitos de docking a ter em conta assim como uma formulação do problema computacional associado ao mesmo. No capítulo 2 é feita 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 o GPU para acelerar o docking. Neste capítulo também está incluído uma secção em que são abordados os conceitos associados ao 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. Como foi implementada essa aceleração e de que forma é que é útil para as optimizações a desenvolver para o BiGGER. Também são abordados duas APIs para programação em GPU: CUDA e OpenCL. Por fim 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 é terminado com a definição do plano de trabalhos para o decorrer da dissertação.

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 respectivas adaptações para simular o docking dos complexos de proteínas têm surgido, adoptando modelos que formulam hipóteses em relação às características dos elementos envolventes.

Um algoritmo pode ser classificado em função da forma que trata a rigidez da superfície dos pares de proteínas a juntar ou até mesmo pelo modelo matemático que os algoritmos 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.

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 corpos é considerado rígido e o outro não, em situações normais este tipo de algoritmos trata o ligando como se fosse a proteína flexível, já que este é mais pequeno do que o receptor, tendo assim uma maior probabilidade de mudar a sua forma, outra justificação tem a ver com os custos de computação serem mais baixos do que se considerarmos os receptores como flexíveis.

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 [36]. Apesar de se considerarem as superfícies de ambos como rígidos, considera-se que ocorrem variações na superfície permitindo que haja penetrações inter-moleculares.

2.1.2 Métodos usados em docking

Existem cinco métodos principais relacionados com o docking: Fast Fourier Transform (FFT) [16], O algoritmo BoGIE [27], Hashing Geométrico[9], correlações de Fourier esféricas polares[30] e algoritmos genéticos de Lamarck. 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 do geometric hashing situa-se entre a complexidade temporal do FFT e a do BiGGER, provando ser uma técnica computacionalmente eficiente. Sobre as correlações de Fourier esféricas polares, foi estudado que são utilizadas no programa HEX[30]. 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 de Fourier em docking

Existe uma adaptação da Transformada de Fourier para docking entre proteínas. 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 [16] onde se considera que ambos os pares são corpos rígidos. Também foi com base no estudo deste artigo que as adaptações que levaram ao BiGGER foram efectuadas[27]. A metodologia com que são determinadas as possibilidades consiste, de forma resumida, nos passos:

1. Determinação da região de fronteira, em que a função discreta determinada no ponto anterior é estendida para suportar os pontos fronteiros: 1 neste passo é atribuído a coordenadas atómicas localizadas na fronteira, ρ associada a coordenadas internas e 0 a externas. A esta função é designada função distreta de Fourier (DFT).

2. É determinada a 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 $360 \times 360 \times 180 \Delta^3$, sendo Δ o intervalo de amostragem rotacional.

Em termos de complexidade temporal, executar estes passos assume uma ordem de complexidade $O(N^3 * \log_2(N^3))$, sendo *N* o número de nós presentes na grelha [17].

Considerou-se fazer um estudo deste método importante pois tal como foi explicitado no princípio desta secção foi a partir do trabalho de Katchalski Katzir e colegas sobre o FFT que foi estudada a abordagem para o BiGGER[17].

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.

Outra razão para se ter feito um estudo sobre esta técnica aborda poder justificar onde é que o BiGGER é mais forte do que as ferramentas que usam FFT aquando na descrição de resultados obtidos na fase da elaboração.

2.1.2.2 Hashing Geométrico

Este método é shape-explicit- 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 [9]. 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 estes 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 do 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 [32]. 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 [14]. 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.

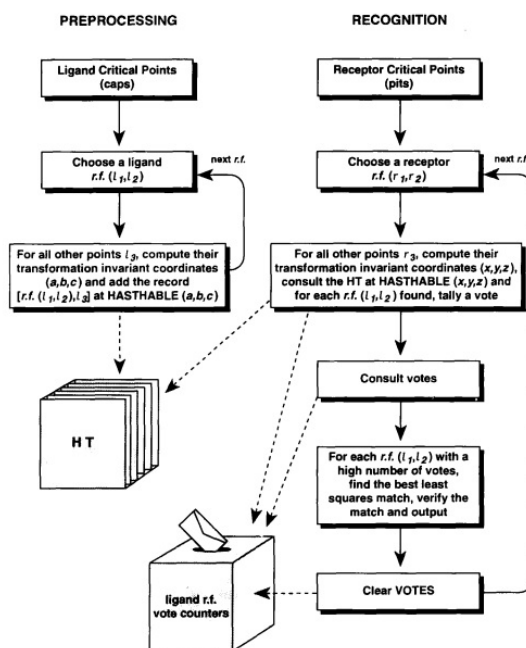


Figura 2.1: Fluxograma sobre o geometric hashing. Tirado de [9]

2.1.2.3 BoGIE

Acrónimo para *Boolean Geometric Interaction Evaluation*[17] [27], é um método de pesquisa em grelha utilizado na primeira fase do BiGGER, que é referido no ponto 2.1.3.1, mais precisamente na amostragem da população total de configurações possíveis para milhares.

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 assume valor 1, se a célula corresponde a uma parcela da proteína cujo centro se encontra a uma distancia tri-dimensional, designada por esfera de Van der Waals, de qualquer outro átomo pertencente a outra proteína, e o valor 0, 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.

Os elementos celulares que ocupam a matriz de superfície são aqueles que na matriz inicial do passo anterior assumiram valor 1 mas tinham vizinhos com valor 0, ou seja, pretende-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, o que corresponde a posições em que o seu centro está próximo do centro do complexo proteico ou podendo até mesmo coincidir.

A forma de garantir que se consegue obter a superfície molecular da proteína é através da operação lógica XOR (OU exclusivo), que terá como output 1 apenas nos pontos da

fronteira, pois é aqui que o resultado do XOR associado aos dois pontos selecionados dá o valor verdadeiro, já que os valores entre as duas células são diferentes e falso se forem iguais.

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

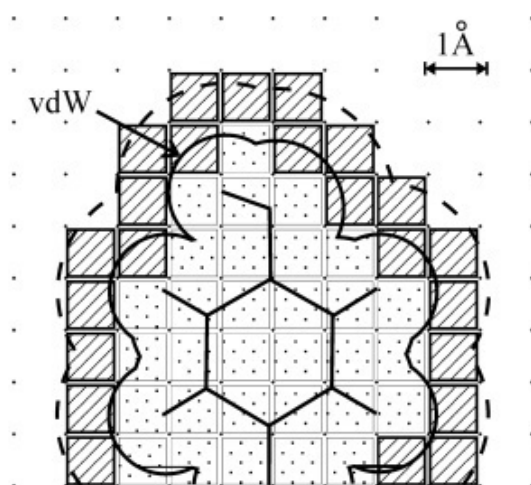


Figura 2.2: Representação em 2D das matrizes resultantes do segundo passo do BoGIE, as células preenchidas a tracejado diagonal correspondem à matriz de superfície, as com pontos correspondem à matriz *core* e a nuvem com tracejado contínuo representa o corte associado à esfera de van der waals com a proteína localizada ao centro [27].

De notar, no entanto, que ambos os passos podem ser otimizados recorrendo ao 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, o ZDOCK, o FTDock, o 3D-Dock, o GRAMM[32] e o HADDOCK. No entanto o ZDOCK, FTDock, 3d-Dock e GRAMM são quatro exemplos de programas que usam FFTs, sendo que apenas será apresentado o ZDOCK [4], já que este é uma ferramenta muito referenciada em trabalhos de implementação de otimizações em GPU de programas para docking. Também é apresentado nesta subsecção o algoritmo BiGGER [27] 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 [dominguez2003haddock] 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[27] é um algoritmo para docking entre proteínas que considera a superfície destas como rígido, fazendo parte do software para docking Chemera.

Este algoritmo permite 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.

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 optimais ($O(N^{2,8})$) 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 um CPU do ano de 2000 (Intel Pentium II 450 MHz dispõe apenas 1 core).

Segundo a lei de Moore, o número de transistors presentes num 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.

2.1.3.2 ZDOCK

O ZDOCK é um programa de docking entre pares de proteínas que usa FFT para otimizar os cálculos respetivos às características das proteínas complementaridade de formato, electroestática 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 [4]. 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 específica chamada GSC (Grid-based Shape Complementarity), tem ainda uma subtração que consiste na penalização de confronto entre os pares de proteínas [3]. Existe ainda a função de score PSC (Pair-based Shape Complementarity) que aplica o mesmo raciocínio 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 + eletrostática entre o par) numa só função, usar apenas PSC, combinar esta última 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 simulado 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 O GPU

2.2.1 Arquitectura e Modelo de Execução

Tal como os CPUs, os GPUs também seguem uma arquitetura. Os conceitos essenciais das arquiteturas dos GPUs modernos são transversais aos diferentes modelos existentes, inclusive de diferentes marcas. No presente documento serão utilizadas como referência as arquiteturas dos GPUs NVIDIA, em particular a arquitetura mais recente (em 2018) de nome Volta[24] que está presente nos GPUs de modelo Tesla V100 (figura 2.4). Esta nova arquitetura traz diversas optimizações de hardware e lógicas face às versões anteriores, eg. Pascal, Maxwell 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* do 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 dispor, a cache L1 e o número de *cores* de execução [37]. 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 2.3. 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.



Figura 2.3: Esquema do SM para o GV100[24].

- **Hierarquia de memória:** O 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 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 política de substituição *write-back*. Esta cache responde a pedidos de instruções load, store assim como instruções atômicas de ambos SM e respectivas caches L1, preenchendo de forma igual as respectivas caches [21]. A partir da arquitetura Volta a cache L1 e a memória partilhada de cada SM passam a estar juntas, o que traz benefícios 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[24]. 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.



Figura 2.4: Esquema da arquitetura do GPU, mais precisamente do Volta GV100[24]. Esta é a arquitetura mais recente que NVIDIA lançou no mercado.

2.2.1.1 Capacidade de computação de uma arquitetura

Todas as arquiteturas NVIDIA têm o conceito de capacidade de computação rotulado 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 no GPU, o número máximo de threads em cada SM e a granularidade de alocação dos registos variam com as diferentes capacidades de computação [22]. O valor da capacidade de computação pode ser usado pelas aplicações em tempo de execução para determinar o que o GPU presente na máquina dispõe em termos de funcionalidades e instruções nativas. Este valor é também decimal, 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

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 [24].

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 compatível 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 no GPU concorrentes a executar código: o conjunto *host* composto por CPUs e o *device*, composto por GPUs. Os dois sistemas desempenham papéis 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 do GPU, tendo cada uma destas threads um ID único, que é acessível dentro do *kernel*. O *host* executa os *kernels* e manipula os dados que o *host* alocou e transmitiu, retornando o resultado para o *host*. As *threads* do GPU são consideradas como *lightweight* pois são escalonadas em grupos conhecidos como *warps*[22]¹. No caso do 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 no CPU. As *cores* do CPU minimizam a latência para um número muito reduzido de threads de cada vez, enquanto que as *cores* do GPU permitem a este gerir um número muito maior de threads mais ligeiras, maximizando o *throughput*. Uma computação que é executada pelo GPU tem de ser estruturada numa grelha.

Cada um dos elementos da grelha é um bloco referido como *thread block* (na figura 2.5 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 atribuída a um *thread block* para processamento.

¹Tendo em conta a quantidade de *threads* individuais que têm de ser geridas e executadas de forma eficiente, é empregado pelos SMs uma arquitetura específica para o efeito, denominada SIMT (*Single-Instruction Multiple-Thread*) [19]. É ainda permitido por parte de qualquer uma das arquiteturas referidas a criação, gestão, escalonamento e execução de *threads* concorrentes em grupos de 32 cada. Estes grupos denominam-se *warps*, podendo cada bloco de *threads* do CUDA ter 1 ou mais *warps* [21].

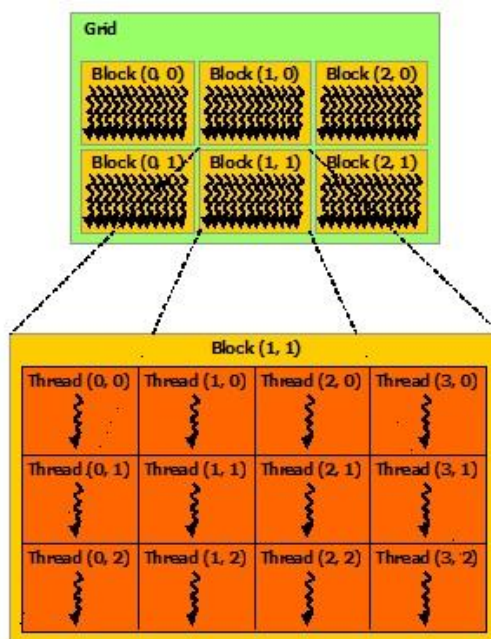


Figura 2.5: Ilustração de uma grelha de threadblocks, detalhando um elemento da grelha para ilustrar os pormenores de um bloco de threads. Neste caso o bloco é bi-dimensional, mas existe a possibilidade de ser tri-dimensional, o mesmo se aplica à grelha [23].

2.2.2 Modelos Base de Programação

Em termos de programação em GPUs existem como APIs o CUDA (*Compute Unified Device Architecture*)[5] que foi implementado pela NVIDIA e o OpenCL (*Open computing language*)[26]. 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[6] para acelerar a técnica FFT. [33]

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 no GPU, eventuais transferências de dados entre o CPU e o 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 parametros de execução (dimensão do bloco e número de threads a invocar no *kernel*), sendo que com estes parametros 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 expectativas iniciais. Existem quatro aspectos a considerar quando pretendemos optimizar um programa recorrendo ao GPU, com os respectivos detalhes de acordo com [22]:

- **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 respectiva à execução do *kernels*. O requisito é ambas as instruções serem de *streams* diferentes e não por omissão², 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 do GPU** : É essencial, para a performance ser optimal, manter os SMs do 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 respectivos blocos maximizando a utilização do hardware, evitando situações em que se deixa de impor a distribuição livre de trabalho entre os SMs. A taxa de ocupação determina o quanto efectivamente o GPU se encontra ocupado, isto é, o número de *warps* que estão activos em relação ao número máximo de *warps* que o GPU consegue activar. É pretendido que esteja o mais próximo possível de um certo limite que depende da capacidade de computação da arquitetura do GPU³. 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 optimal. Para garantir a taxa de ocupação

²Uma *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.

³Como 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.

adequada ao 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 acessível 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 múltiplo 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 memória por parte das threads do warp numa quantidade de transações de memória que depende do tamanho da 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[23].

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 dividida 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.

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

2.3 Docking em GPU

Existem vários trabalhos que focam a paralelização de etapas constituintes do processo de docking no GPU⁵. 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* [15] onde são discutidas duas abordagens em função da taxa de ocupação do 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 drasticamente após o desenvolvimento da versão 4.0, demonstrando os benefícios de adaptar um programa para executar em GPU. Os trabalhos sobre o Megadock [31] [25] abordam uma possibilidade para mapear etapas do funcionamento do BiGGER para o GPU.

Foi considerado o PIPER pois em termos históricos este foi um dos primeiros programas para docking a ser acelerado usando o CUDA[34]. 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 [18].

2.3.1 Megadock

O Megadock 4.0[25] é um software de protein-protein docking de origem japonesa, com optimizações para executar recorrendo ao 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 típicas de supercomputadores. No entanto é possível utilizar o megadock em computadores pessoais, alterando a flag de compilação do programa para usar apenas a implementação 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 disponíveis. 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 2.6 ilustram-se as etapas no processo de docking no Megadock, foi aplicado

⁵Na 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 últimos são para docking entre proteína-ligando.

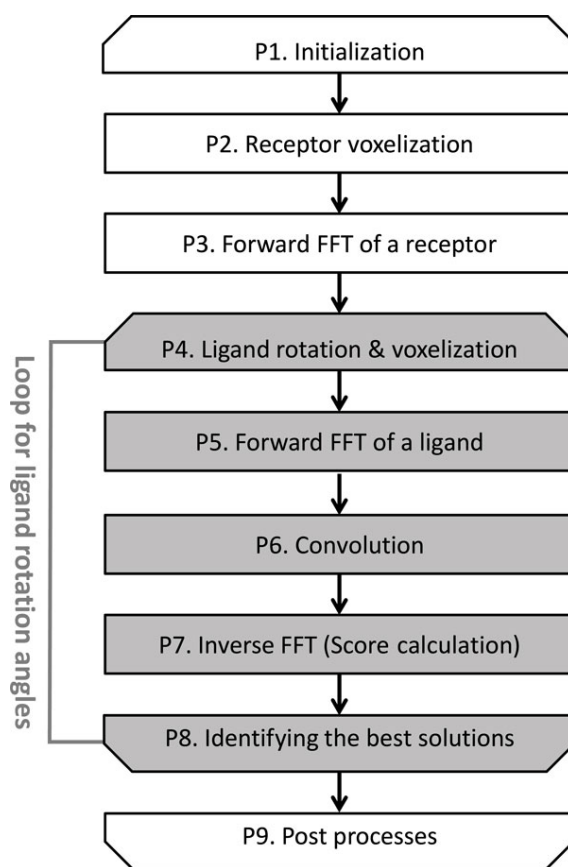


Figura 2.6: Etapas do processo de docking no Megadock. As etapas a cinzento foram aceleradas por GPU[31].

um profile sobre o funcionamento do programa com apenas 1 core do CPU, registrando os tempos de execução em cada etapa. Os resultados presentes em [31] 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 atômicas do ligando para o 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 atômicas 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 o GPU, sendo cada átomo designado a uma *core* do GPU.

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 utilizada a biblioteca especializada para FFT, cuFFT. Na etapa P6, convolução, foi aplicado um mapeamento para o GPU. E na última etapa do ciclo, P8, em que os resultados têm uma dada pontuação, foi aplicada uma operação de redução no GPU. Sobre a comunicação *host-device*, é referido que apenas aconteceu uma

vez a transferência de dados. O conteúdo da transferência inclui as coordenadas atômicas 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 [25]. A implementação GPU demonstrou ser 15.1 vezes mais rápida do que a versão que apenas usava um *core* do CPU.

2.3.1.2 Conclusões

Sobre a implementação 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 [31] é revelado que a implementação MIC, apesar de ter menos custos de implementação, não é tão eficiente como a implementação 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 optimal para o ligando. Também podem ser aproveitados os mapeamentos para o GPU descritos. Sobre os custos de implementação, é referido em [31] que no processo de mapear o docking para o 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 o 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 desolvatação 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 concluído são iteradas as possíveis rotações, em que as etapas referidas na figura 2.9 são aplicadas para cada rotação.

2.3.2.1 Execução em GPUs

A implementação de acelerações ao PIPER[34] por GPU ocorreu inicialmente em 2009. No entanto em 2014, a performance da versão PIPER GPU de 2009⁶ foi confrontada com a versão CPU de 2014⁷ através da execução de um profile à performance das duas versões. A versão CPU2014 introduziu otimizações no algoritmo original, mais precisamente foi alterada a biblioteca que aplicava o FFT. Verificou-se que a versão CPU2014 conseguiu ter performance superior às acelerações introduzidas em 2009, com execuções utilizando um GPU de 2014 [18]. 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 2.10. 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 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 utilizar 8 desses conjuntos ao mesmo tempo, e utilizar 1 SM para calcular o *score* optimal de cada um dos conjuntos⁹. Esta otimização deixou de ser válida pois em 2014 uma das otimizações 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 do GPU estava a ser utilizado. Passou a ser pretendido para o passo de filtragem encontrar o *score* optimal 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 do GPU possa ser utilizada para acessos rápidos de memória assim como o trabalho possa ser distribuído 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 2.7). 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* optimal 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

⁶Esta versão será referida ao longo do texto como GPU09

⁷Esta versão será referida ao longo do texto como CPU2014

⁸Esta versão será referida ao longo do texto como GPU2014

⁹A solução GPU09 foi desenhada com o GPU Tesla C1060 em mente, esta arquitetura tem 30 SMs estando 8 ocupados com a execução do *kernel* o que garante uma performance melhor do que uma versão do PIPER que apenas utilize o CPU.

subsecção 2.2.3 em relação às optimizações de memória. O *kernel* é finalizado quando cada thread executada determina o score óptimo geral para cada bloco e guarda este na memória global.

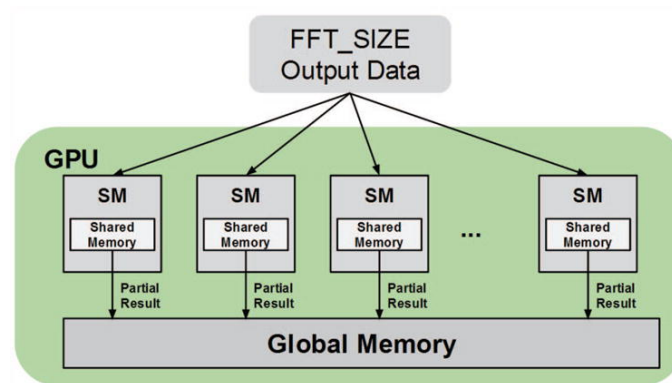


Figura 2.7: Esquema para o primeiro kernel introduzido na versão GPU2014 [18].

O kernel anterior usa mais do que um thread block, por sua vez o segundo *kernel* apenas usa um bloco de threads (figura 2.8). 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 top score é determinado e guardado na memória global.

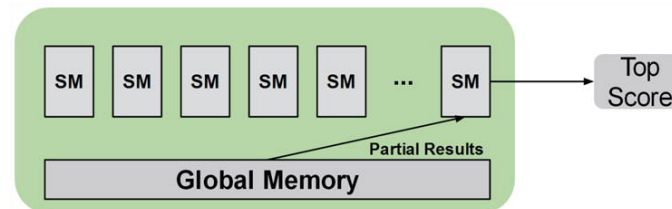


Figura 2.8: Esquema para o segundo *kernel* introduzido na versão GPU2014 [18].

Para além da adição dos dois *kernels* referidos anteriormente, o tempo em que o GPU não desempenha funções enquanto que a rotação e atribuições na grelha estão a ser processados pelo CPU foi eliminado. A computação da grelha do ligando passa a ser desempenhada pelo GPU, deixando de necessidade de a transferir entre a memória do *host* e a do 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 solução GPU09 que foram mantidas. Nas etapas em que a aplicação do FFT às grelhas é feita, a optimização original consistiu em aplicar a biblioteca 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 o GPU, novamente à semelhança da implementação para o Megadock.

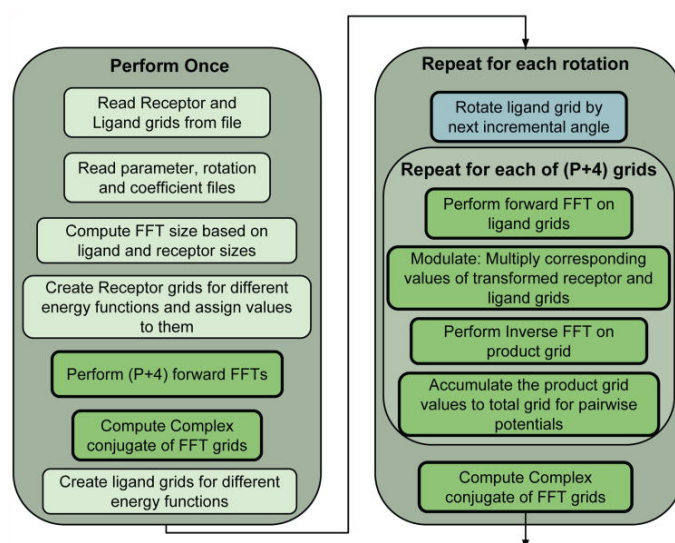


Figura 2.9: Etapas do processo de docking no PIPER. As etapas destacadas a verde escuro foram aceleradas por GPU em 2009 e a etapa a azul em 2014. [18]

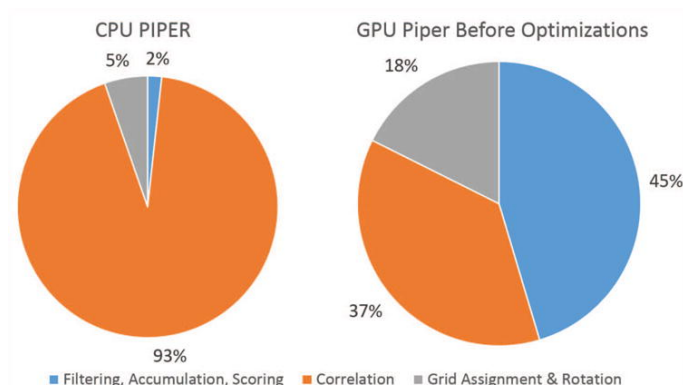


Figura 2.10: Proporções de tempo gasto na execução para a versão CPU otimizada e a versão de 2009 que usa GPUs[18].

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 o CPU com otimizações. 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 otimizações que foram aplicadas ao PIPER em 2014 foram aplicadas à fase de *scoring*, mais precisamente às etapas de filtragem e cálculo dos *scores* otimais. Pelo que é mostrado forma de otimizar a fase de *scoring* do BiGGER, implementando uma redução como foi confirmado no caso do Megadock para a mesma etapa.

2.3.3 AutoDock

O AutoDock [20] é um programa diferente dos até agora indicados, não só por ser específico para interações entre proteína-droga¹⁰ como também por utilizar 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[35]. 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 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 o GPU

Em 2010 foi abordada, à semelhança do que foi feito para o Megadock assim como para o PIPER, uma possível paralelização do AutoDock utilizando o GPU [15]. Neste caso a API utilizada 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 AutoDock 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 utiliza para determinar a posição optimal do ligando encaixado com a proteína. Mais precisamente a função de fitness do algoritmo genético, que avalia a energia dos indivíduos da população constituída 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 do GPU ao cálculo da energia total de um indivíduo ao que eles designaram *PerThread* e a segunda dedicar um bloco de threads do GPU para determinar essa energia total (*PerBlock*). A primeira solução garante a saturação do GPU (taxa de ocupação alta) para milhares de indivíduos e a segunda oferece a mesma saturação para centenas de indivíduos. 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 indivíduo, aplicar a torção até ao cálculo da energia interna) para as coordenadas atómicas respetivas a um indivíduo. 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*.

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

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 optimal e mapeamentos de dados para o GPU. A solução para o AutoDock oferece uma forma de poder acelerar o cálculo da função de score do BiGGER utilizando a memória partilhada do GPU.

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 otimizar 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*) [22] 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` do `Open-chemera` e determinadas as funções que este ficheiro chama onde a fracção de tempo de execução é maior.
2. **Parallelize** : Após o *assessment* referido anteriormente estar concluído, 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 [13], 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. E recorrer às técnicas descritas na subsecção 2.2.3 para maximizar a performance. Também se podem considerar 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. Em caso de não corresponder ao speedup potencial registado na fase inicial, é necessário voltar à fase *Assess*, recomeçando o ciclo.

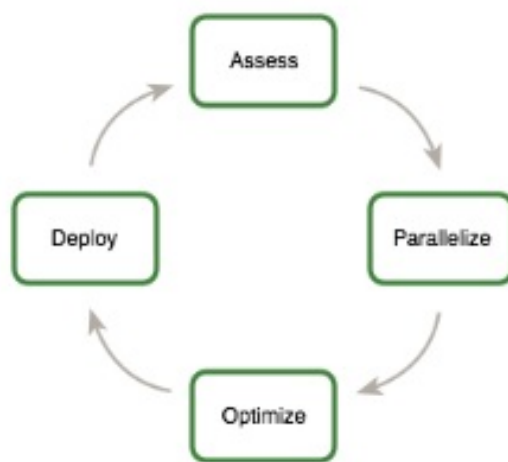


Figura 3.1: Diagrama do ciclo de desenvolvimento APOD[22].

3.1.2 Profiling

De forma a confirmar as suposições sobre as zonas de código a otimizar para execução no 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 parametro 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 [29]:

- **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 paralelização.
- **LazProf**: IDE de profiling para o Lazarus, que funciona complementada com o FPPProfiler.

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). 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 *textbfdockdomains.pas* são determinados os domínios 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 último 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 tri-dimensionais 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 proteína. Neste contexto, é possível determinar a região de superfície de cada proteína, sendo esta definida pelo conjunto de todos os nós da matriz que se encontram a 1. Na versão sequencial do BiGGER, a região de superfície 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 no GPU e distribuindo cada uma das direções pelos SMs.

- **Pesquisa de sobreposições :** De forma a poder avaliar a correspondência entre as superfícies 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 superfície 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 superfície. 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 superfície não são. É feita uma rotação entre os pares de proteínas em um ângulo de 15° por omissão, repetindo os processos de criação das grelhas, translação por shift e correspondência de superfície 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 o GPU. Neste contexto é possível abordar uma optimização em que os nós da grelha são mapeados para o GPU. Cada um dos nós serão processados por um core do GPU e o AND passa a ser aplicado entre registos do GPU em vez de ser em registos do CPU.
- **Filtragem de Candidatos:** O BiGGER contém 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 superfície, seguida pelo número de contactos favoráveis entre pares de amino-ácidos. 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 superfície e número de contactos em dois kernels. A tabela com os melhores candidatos poderá ser guardada em memória partilhada no 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 superfície, contactos entre amino-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 coincidências 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 [22], 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 direcções 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 concluída, recorrendo à lei de Amdahl [2], 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 formula $S = t1/tp$, sendo $t1$ o tempo de execução da versão do BiGGER que usa apenas o CPU e tp o mesmo tempo para a versão que usa ambos CPU e GPU.

3.3 Plano de Trabalhos

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] 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.
- [3] 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.
- [4] 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.
- [5] CUDA Zone. URL: <https://developer.nvidia.com/cuda-zone>.
- [6] cuFFT. 2018. URL: <https://developer.nvidia.com/cufft>.
- [7] Docking (molecular). URL: [https://en.wikipedia.org/wiki/Docking_\(molecular\)](https://en.wikipedia.org/wiki/Docking_(molecular)).
- [8] *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-protoger-contr-a-doenca-de-alzheimer/>.
- [9] D. Fischer, S. L. Lin, H. L. Wolfson e R. Nussinov. “A geometry-based suite of molecular docking processes”. Em: *Journal of Molecular Biology* 248.2 (1995), pp. 459–477.
- [10] M. W. Gonzalez e M. G. Kann. “Protein interactions and disease”. Em: *PLoS computational biology* 8.12 (2012), e1002819.
- [11] GPGPU.org. URL: <http://gpgpu.org/about>.
- [12] GPU Applications Catalog. URL: <https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/>.
- [13] M. Harris. *Six Ways to SAXPY*. URL: <https://devblogs.nvidia.com/six-ways-saxpy/>.
- [14] 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>.

- [15] S Kannan e R Ganji. “Porting AutoDock to CUDA [J]. Evolutionary Computation (CEC)”. Em: *2010 IEEE Congress on*, pp. 1–8.
- [16] 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](https://doi.org/10.1073/pnas.89.6.2195).
- [17] L. Krippahl. *Integrating protein structural information*. 2003.
- [18] 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.
- [19] E. Lindholm, J. Nickolls, S. Oberman e J. Montrym. “NVIDIA Tesla: A unified graphics and computing architecture”. Em: *IEEE micro* 28.2 (2008).
- [20] G. M. Morris. *AutoDock*. URL: <http://autodock.scripps.edu/>.
- [21] J. Nickolls e W. J. Dally. “The GPU computing era”. Em: *IEEE micro* 30.2 (2010).
- [22] NVIDIA. *CUDA C BEST PRACTICES GUIDE*. URL: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide>.
- [23] NVIDIA. *CUDA C PROGRAMMING GUIDE*. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide>.
- [24] NVIDIA. *NVIDIA Tesla V100 GPU architecture*. URL: <http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>.
- [25] M. Ohue, T. Shimoda, S. Suzuki, Y. Matsuzaki, T. Ishida e Y. Akiyama. “MEGA-DOCK 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>.
- [26] *OpenCL*. URL: <https://developer.nvidia.com/opencl>.
- [27] 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.
- [28] 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.
- [29] *Profiling*. URL: <http://wiki.lazarus.freepascal.org/Profiling>.
- [30] 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>.
- [31] 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.

- [32] 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.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] I. A. Vakser. "Protein-protein docking: From interaction to interactome". Em: *Biophysical journal* 107.8 (2014), pp. 1785–1793.
- [37] N. Wilt. *The CUDA handbook: a comprehensive guide to GPU programming*. Addison-Wesley, 2013.

