



**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, Full Professor,  
NOVA University of Lisbon

Co-orientador: Hervé Paulino, Associate  
Professor, NOVA University of Lisbon



FACULDADE DE  
CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

**July, 2018**



## RESUMO

---

Na área científica da Bio-Informática, o estudo das interações entre as proteínas tem sido objeto de interesse. No entanto conseguir determinar computacionalmente e com precisão como é que as mesmas se unem é difícil. Existem, no entanto diversos métodos e algoritmos para prever o ajuntamento das proteínas. 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 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, docagem de proteínas, 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

<b>Lista de Figuras</b>	<b>ix</b>
<b>Lista de Tabelas</b>	<b>xi</b>
<b>Listagens</b>	<b>xiii</b>
<b>Glossário</b>	<b>xv</b>
<b>Siglas</b>	<b>xvii</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Enquadramento e motivação . . . . .	1
1.2 Formulação do problema . . . . .	1
1.2.1 Conceito de docking . . . . .	1
1.2.2 Tipos de Interações . . . . .	2
1.3 Objectivos a alcançar na elaboração da dissertação . . . . .	3
1.4 Estrutura deste documento . . . . .	3
<b>2 Estado da arte</b>	<b>5</b>
2.1 Docking . . . . .	5
2.1.1 Conceitos de docking em relação à rigidez da superfície . . . . .	5
2.1.2 Métodos usados em docking . . . . .	6
2.1.3 Ferramentas para Docking . . . . .	9
2.2 O GPU . . . . .	11
2.2.1 Arquitectura e Modelo de Execução . . . . .	11
2.2.2 Modelos Base de Programação . . . . .	15
2.2.3 Optimizações . . . . .	16
2.3 Docking em GPU . . . . .	18
2.3.1 Megadock . . . . .	18
2.3.2 PIPER . . . . .	20
2.3.3 AutoDock . . . . .	22
2.3.4 Dinâmica Molecular em GPUs . . . . .	23
<b>3 Plano de trabalhos para a Elaboração da dissertação</b>	<b>25</b>

## ÍNDICE

---

3.1	Ciclo de desenvolvimento . . . . .	25
3.1.1	Profiling . . . . .	26
3.1.2	Possibilidades de otimização . . . . .	27
3.2	Desafios . . . . .	28
3.3	Plano de Trabalhos . . . . .	28
	<b>Bibliografia</b>	<b>29</b>



## LISTA DE FIGURAS

1.1	Representação gráfica do docking[6]. . . . .	2
2.1	Fluxograma sobre o geometric hashing. Tirado de [8] . . . . .	8
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]. . . . .	9
2.3	Esquema do SM para o GV100[23]. . . . .	12
2.4	Esquema da arquitetura do GPU, mais precisamente do Volta GV100[23]. Esta é a arquitetura mais recente que NVIDIA lançou no mercado. . . . .	13
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 [22]. . . . .	15
2.6	Etapas do processo de docking no Megadock. As etapas a cinzento foram aceleradas por GPU[30]. . . . .	19
2.7	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. [17] . . . . .	21
2.8	Proporções de tempo gasto na execução para a versão CPU otimizada e a versão de 2009 que usa GPUs[17] . . . . .	22
3.1	Diagrama do ciclo de desenvolvimento APOD[21]. . . . .	26
3.2	Representação gráfica do ID geral de uma thread num array dividido por blocos, em CUDA. Adaptações de [29] [36] . . . . .	28



## LISTA DE TABELAS

2.1	Tabela de capacidades computacionais das arquiteturas NVIDIA mais recentes. Adaptado de [23]. . . . .	14
2.2	Tempos de cálculo das versões iniciais do Megadock em relação ao ZDOCK [25]. . . . .	20



## LISTAGENS



## GLOSSÁRIO

aliquam	tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris..
computer	An electronic device which is capable of receiving information (data) in a particular form and of performing a sequence of operations in accordance with a predetermined but variable set of procedural instructions (program) to produce a result in the form of information or signals..
cras viverra	metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat..
donec nonummy	pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo..
integer sapien	est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus..
lorem ipsum	dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris..
maecenas lacinia	nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem..
morbi ac	orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus..
morbi dolor	nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum..

nam lacus	libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi..
nam dui	ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo..
name arcu	libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo..
nulla malesuada	porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis..
sed lacinia	nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus..



## 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.
GPU	Graphics Processor Unit.
GSC	Grid-based Surface Complementarity.
IDE	Integrated Development Environment.
MPI	Message Passing Interface.
NVCC	NVIDIA CUDA Compiler.
PDI	Protein-Drug Interaction.
PPI	Protein-Protein Interaction.
PSC	Pair-based Surface Complementarity.
SM	Stream Multiprocessor.
SP	Scalar Processor.
XOR	Exclusive Or.



## INTRODUÇÃO

### 1.1 Enquadramento e motivação

Estudar as interações entre as proteínas tem garantido avanços na descoberta de formas de proteger o Homem 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-amiloide provocam um atraso na formação dos agregados do beta-amiloide, trazendo como benefício a proteção contra a doença de Alzheimer[7]. Estudar estas interações trouxe igualmente avanços consideráveis no desenho de drogas assistido por computador. Neste caso as interações a considerar são entre uma proteína e outra de dimensões menores.

Segundo [28] a maioria dos complexos de proteínas ainda não foram adicionados à base de dados sobre as proteínas (PDB), que contem apenas os complexos descobertos através cristalografia por raio 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.

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 melhorar a performance do BiGGER.

### 1.2 Formulação do problema

#### 1.2.1 Conceito de docking

De acordo com [13], 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 vinculado (*bounded docking*) aplica os passos computacionais referidos anteriormente na determinação do melhor encaixe através das estruturas vinculadas do par. Esta vertente de docking é computacionalmente mais simples do que o docking não-vinculado (*unbounded docking*), onde a computação é semelhante à do docking vinculado mas a determinação do melhor encaixe é obtida através das estruturas não-vinculadas. O docking não-vinculado é também conhecido como docking predictivo, sendo a abordagem mais pretendida porém computacionalmente mais complicada.

O problema associado ao docking consiste em duas partes: desenvolver uma função de score que consegue discriminar com a maior precisão as orientações do ligando optimais das que não optimais. A segunda é determinar um método de pesquisa global que consiga determinar qual das orientações corretas a melhor [32].

Docking de proteínas consiste em prever a estrutura tri-dimensional do complexo de proteínas através das coordenadas atómicas do ligando e do receptor, consistindo em duas fases. Na fase de pesquisa global, as proteínas são consideradas como sendo corpos rígidos, o receptor fica estático e ao ligando são aplicadas rotações e translações, sendo determinados os complexos candidatos. O passo final desta fase é avaliar os candidatos encontrados através da função de score baseada na complementaridade de superfície. A avaliação é feita por uma abstração da molécula numa grelha tri-dimensional, e determinando para cada célula da grelha, se existe correspondência com uma coordenada atómica da molécula. A segunda fase consiste na atribuição de pontuação aos candidatos resultantes da fase anterior, através de uma função de score com parametros como contactos residuais, eletroestática até dissolução. A gama de parametros 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].

### 1.2.2 Tipos de Interações

Existem dois tipos de interações a considerar: interações entre dois pares de proteínas (PPI) e interações entre uma proteína e uma molécula de dimensões pequenas (PDI).

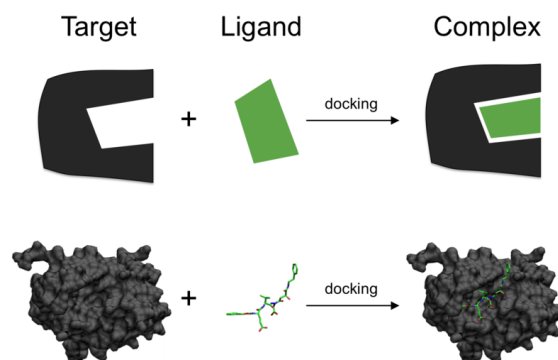


Figura 1.1: Representação gráfica do docking[6].

### 1.3 Objectivos a alcançar na elaboração da dissertação

Assumindo que a preparação tem condições para avançar para a fase de elaboração, o objetivo desta dissertação é implementar a aceleração do algoritmo BiGGER. Esta aceleração terá de trazer à versão futura do BiGGER ganhos de speedup consideráveis em relação à versão actual que apenas usa um *core* do CPU. O resultado final terá de ser uma versão do BiGGER que garanta uma alternativa mais eficiente em termos de computação e tempo de execução face aos restantes algoritmos de docking de proteínas.

### 1.4 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 é abordado 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 a aceleração do BiGGER. Também são abordados duas APIs para programação em GPU: CUDA e OpenCL. Por fim no capítulo 3, é descrito o estado de performance da versão actual do BiGGER, através dos resultados do profiling feito a este. Estes resultados permitem definir as zonas de código do programa que precisam de ser acelerados por GPU e por consequência o plano de trabalho para os acelerar. Aborda-se ainda duas possíveis ramificações sobre como implementar os mecanismos de aceleração ao BiGGER assim como os aspetos positivos e os negativos de ambas.



## ESTADO DA ARTE

### 2.1 Docking

#### 2.1.1 Conceitos de docking em relação à rigidez da superfície

Com o passar do tempo cada vez mais algoritmos e respetivas 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 abordadas nas próximas subsecções 3 modelos de docking: flexível, semi-flexível e rígido.

##### 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.

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

Este modelo tem a vantagem a ser o mais rápido dos 3 a considerar, sendo capaz de explorar as superfícies de ambos os elementos e traduzir para uma base de dados de docking [13].

## 2.1.2 Métodos usados em docking

### 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. três exemplos de ferramentas abordadas nesta secção são o FTDock, GRAMM e ZDOCK que são específicos para interações entre proteínas. A origem do uso do FFT no docking remonta ao artigo de Katchalski Katzir et al [15] 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 fronteirios: 1 neste passo é atribuído a coordenadas atómicas localizadas na fronteira,  $\rho$  associada a coordenadas internas e 0 a externas. A esta função é designada função discreta 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  $\Delta$  o intervalo de amostragem rotacional.

Em termos de complexidade temporal, executar estes passos assume uma ordem de complexidade  $O(N^3 * \log_2(N^3))$ [16].

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[16].



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 [8]. 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 demais métodos executam por uma verificação numa hash table, o que introduz rapidez em termos de computação efectuada.

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 [13]. 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.

### 2.1.2.3 BoGIE

Acrónimo para *Boolean Geometric Interaction Evaluation*[16] [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.

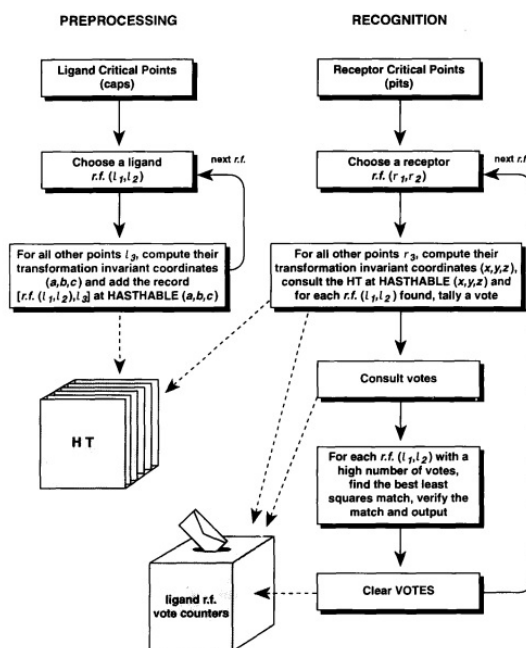


Figura 2.1: Fluxograma sobre o geometric hashing. Tirado de [8]

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.

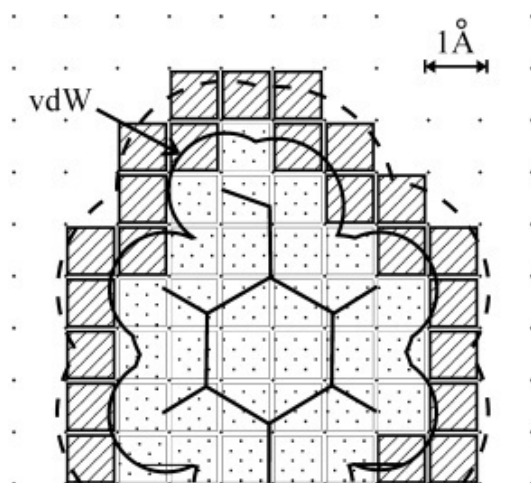


Figura 2.2: Representação em 2D das matrizes resultantes do segundo passo do BoGIE, as células preenchidas a tracejado diagonal correspondem à matriz de superfície, as com pontos correspondem à matriz *core* e a nuvem com tracejado contínuo representa o corte associado à esfera de van der Waals com a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

#### 2.1.3.1 BiGGER

O BiGGER[27] é um algoritmo para docking entre proteínas que considera a superfície destas como rígida. Faz parte do software para docking Open-Chemera.

Este algoritmo permite consiste em dois passos: o primeiro efectua uma redução de possíveis configurações resultantes de passos de translação e rotação numa magnitude de cerca de  $10^{15}$  configurações para poucos milhares das anteriores, 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 optimizar os cálculos respetivos às características das proteínas complementaridade de formato, electrostática e dissolução (desolvation), sendo este o aspeto que faz com que o ZDOCK tenha uma performance positiva. O processo de funcionamento deste algoritmo foi abordado em [3]. 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 [2]. 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 + dissolução + eletrostática entre o par) numa só função, usar apenas PSC, combinar está última com a dissolução 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 com as complementações referidas.

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 distancia 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.

### 2.1.3.3 FTDock

Esta ferramenta foi implementada em Perl e Fortran, apenas disponível para sistemas operativos UNIX [9]. Em termos de docking, usa a complementaridade de formato e eletroestática como função de score assim como a técnica de correlação de matrizes por FFT descrita no ponto 2.1.2.1, sendo uma ferramenta semelhante ao ZDOCK. No entanto para o FTDock desenvolveram-se melhorias em relação à versão do FFT introduzida em 1992. É considerado na função de score sobre a eletroestática do par um constrangimento adicional, de forma a aumentar a precisão do algoritmo na fase de pesquisa global. O tempo de execução para um docking completo (são completadas as duas fases) demorava seis horas para oito processadores em simultâneo.

## 2.2 O GPU

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. O uso do GPU para este tipo de processamentos é referido como *General-purpose computation on graphics processing units* (GPGPU)[10]. 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, em [11]. Algumas das aplicações presentes no catálogo referido são mencionadas na secção 2.3.

### 2.2.1 Arquitectura e Modelo de Execução

Tal como os CPUs, os GPUs também seguem uma arquitetura. Os aspetos essenciais não diferem entre arquiteturas, ou seja, o que está presente numa arquitetura de um GPU AMD também existe num GPU NVIDIA. No âmbito desta dissertação serão utilizados GPU de fabrico NVIDIA pelo que é com base nestes que serão descritos ambas arquitetura e funcionamento de um GPU em geral. No caso dos GPUs fabricados pela NVIDIA, a arquitetura mais recente ( em 2018) é a Volta[23] 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.

Generalizando as arquiteturas têm os seguintes elementos:

Na arquitetura Volta, cada um dos 84 SMs presentes, estão partidos 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. O GV100 encontra-se dividido em 6 GPCs, cada um destes GPCs contem 14 SMs. 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.



- **Caches e memória:** Existem 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 respetivas caches L1, preenchendo de forma igual as respetivas caches [20]. 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[23]. 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[23]. 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 [21]. 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

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 [23].

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 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 Fermi (capacidade computacional 2.0) possa ser compatível para execução num GPU com a arquitetura Volta (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 dispositivos concorrentes a executar código: o conjunto *host* composta por CPUs e o *device*, composta por GPUs. Os dois sistemas são diferentes no papel desempenham. No caso do *host*, este coordena as transferências de dados a manipular entre as duas entidades e a invocação dos *kernels*. 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 *device* é responsável pela execução dos *kernels* e manipulação dos 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*[21]<sup>1</sup>. 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 com até 3 dimensões.

Cada um dos elementos da grelha é um bloco com a definição de *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 1024, pelo que cada *thread block* pode ter no máximo 1024 threads. Numa situação em que é pretendido fazer computações numa estrutura de

<sup>1</sup>Tendo 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*) [18]. 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* [20].



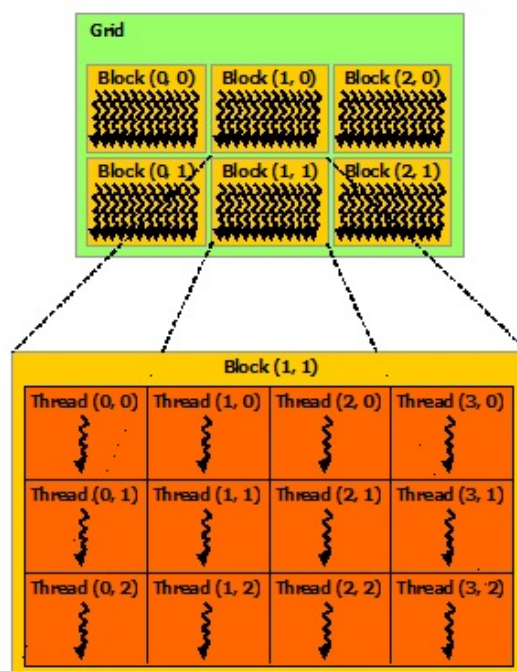


Figura 2.5: Ilustração de uma grade de threadblocks, detalhando um elemento da grade para ilustrar os pormenores de um bloco de threads. Neste caso o bloco é bi-dimensional, mas existe a possibilidade de ser tri-dimensional, o mesmo se aplica à grade [22].

dados de tamanho superior a 1024 unidades, a mesma é repartida em  $N/1024$  bocados, sendo  $N$  o tamanho da estrutura, e cada um desses bocados é atribuído 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*)[4] 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 do CUDA para acelerar a técnica FFT, cuFFT[5]. Quer para CUDA, quer para OpenCL, para além do que o programador necessita de programar em ambos host e device, existem dois elementos a considerar: *kernels* e *thread blocks*.

Tendo em conta estes dois elementos, 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 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 que é necessita de ser paralelizado de forma massiva pelo GPU, 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 quatros aspetos a considerar quando pretendemos optimizar um programa recorrendo ao GPU, com os respetivos detalhes de acordo com [21]:

- **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. 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 defeito<sup>2</sup>, 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 respetivos 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 quão 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

---

<sup>2</sup>Uma *stream* por defeito tem o seu *streamID* com o valor nulo. Pelo que o pretendido é uma *stream* cujo o seu id seja diferente de 0.

capacidade de computação da arquitetura do GPU <sup>3</sup>. 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 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[22].

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 entre 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 <sup>4</sup> o uso de *branch predication* ou seja, o compilador desenrola os loops/condições impedindo divergências de *warps*. Apenas

<sup>3</sup>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.

<sup>4</sup>No caso do CUDA o compilador é o NVCC, no caso do OpenCL é o OpenCL Compiler.

as instruções em que o predicado assume o valor verdadeiro em relação à condição de controlo são executadas.

## 2.3 Docking em GPU

Face à complexidade do problema de docking e os softwares específicos terem execuções muito extensas, por apenas utilizarem cores do CPU, foi necessário desenvolver os softwares de docking existentes para versões que recorram ao GPU para efectuar as computações necessárias e obter tempos de execução menos extensos. Desta forma torna-se possível o estudo das interações entre proteínas mais rápido e eficiente.

Os exemplos em baixo apontados estão divididos pelo tipo de interação entre proteínas, podendo ser PPI ou PDI.

### 2.3.1 Megadock

O Megadock é um software de protein-protein docking de origem japonesa, baseado em grelhas FFT. A versão 4.0[24] traz melhorias em relação ao Megadock 1.0 no sentido de ter implementações no âmbito da computação acelerada como é apontado em [25].

Este programa é adequado para máquinas que têm muitas *cores* de GPU e CPU à disposição, características típicas de supercomputadores. 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 Optimizações para GPU

As acelerações implementadas para o Megadock consistem em 6 *kernels*. Na figura 2.6) ilustram-se as etapas no processo de docking no Megadock, foi aplicado um profile sobre o funcionamento do programa com apenas 1 core do CPU, registando os tempos de execução em cada etapa. Os resultados presentes em [30] 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 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 atómicas de um dado átomo do ligando. As

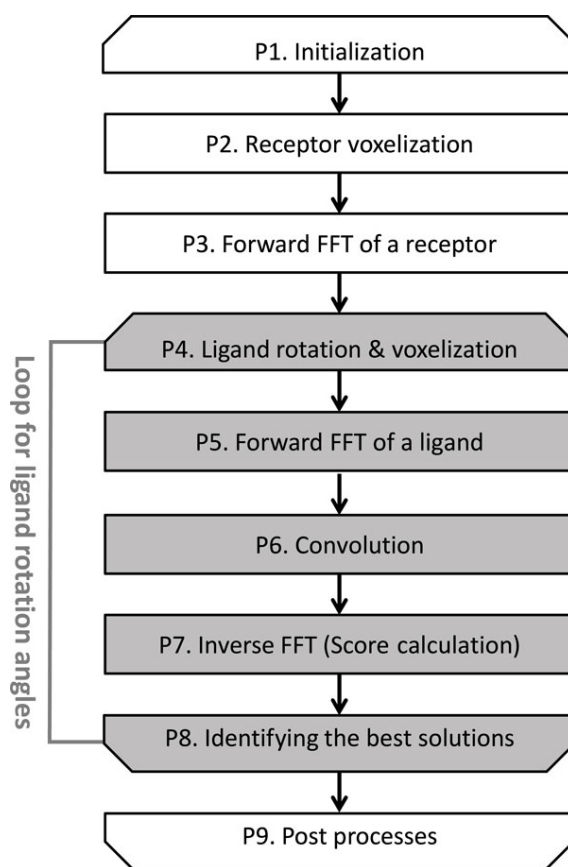


Figura 2.6: Etapas do processo de docking no Megadock. As etapas a cinzento foram aceleradas por GPU[30].

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 existe a aplicação do FFT ao ligando e o cálculo da inversa do FFT, respetivamente, foi utilizada a biblioteca acelerada 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. Em termos de comunicação host-device, é referido que apenas aconteceu uma vez a transferência de dados. O conteúdo destes 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 (tabela 2.2) pelos autores do programa revelam a sua rapidez. O Megadock foi capaz de determinar em minutos um caso de teste que o ZDOCK determinou em duas horas. Os resultados referidos foram obtidos por execução no supercomputador disponível no Instituto de Tecnologia de Tóquio, TSUBAME 2.0. A dimensão amostral consiste no número total de combinações de estruturas receptoras e no número total de combinações para estruturas ligandas. No caso do Megadock 4.0, o valor a considerar para o tempo de cálculo ronda

	Megadock 1.0	Megadock 2.0	Megadock 2.1	ZDOCK 3.0
Tempo de cálculo (min)	13.3	14.7	16.6	124.6

Tabela 2.2: Tempos de cálculo das versões iniciais do Megadock em relação ao ZDOCK [25].

as 5.71h para meio milhão de jobs e 11.51H para 1 milhão de jobs. Para este último teste foi utilizado a versão 2.3 do TSUBAME[24].

### 2.3.1.2 Conclusões

Sobre esta implementação da parte do Megadock que utiliza o GPU, em 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. As implementações 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.

### 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 pela primeira vez o uso da energia de desolvacão 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 (figura 2.7) 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 criar as grelhas do ligando. Após o *setup* estar concluído são iteradas as possíveis rotações, em que as etapas referidas na figura são aplicadas para cada rotação.

#### 2.3.2.1 Optimizações para GPU

A implementação de acelerações ao PIPER[33] por GPU ocorreu inicialmente em 2009, no entanto, em 2014 a performance da versão PIPER GPU de 2009 foi confrontada com a versão CPU de 2014. Esta versão introduziu optimizações no algoritmo original, mais precisamente foi alterada a biblioteca que aplicava o FFT. Verificou-se que a versão CPU 2014 conseguiu ter performance superior às acelerações introduzidas em 2009, com execuções utilizando um GPU de 2014 [17]. A proporção de tempo gasto no passo de correlação das matrizes é maior na versão CPU do que na GPU 2009 como se pode observar na figura 2.8. 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. Sobre o passo de filtragem e cálculo do score, a versão de 2009 usava 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 optimal consistia em utilizar 8 desses conjuntos ao mesmo tempo, e utilizar 8 SMs para calcular o score optimal de cada um dos conjuntos<sup>5</sup>. 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 para um, sendo apenas um SM do GPU utilizado, passando a ser pretendido encontrar o score optimal para o conjunto de coeficientes no menor tempo possível, de forma repetida. A versão GPU2014 introduziu nestes dois passos dois kernels em substituição de um na versão GPU2009, os dois 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.

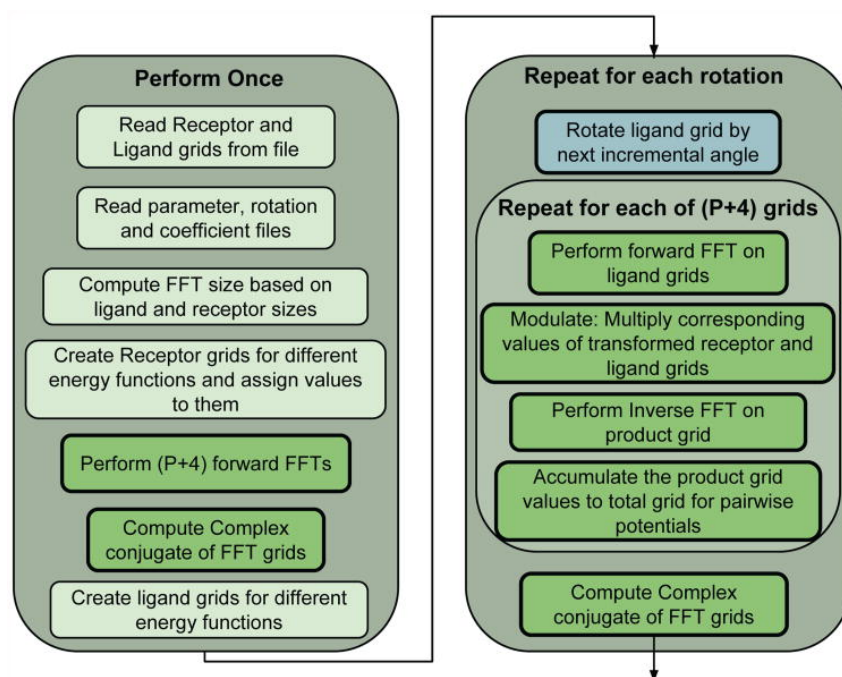


Figura 2.7: Etapas do processo de docking no PIPER. As etapas destacadas a verde escuro foram aceleradas por GPU em 2009 e a etapa a azul em 2014. [17]

### 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 do mesmo programa que use o CPU com optimizações.

<sup>5</sup>A solução 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.

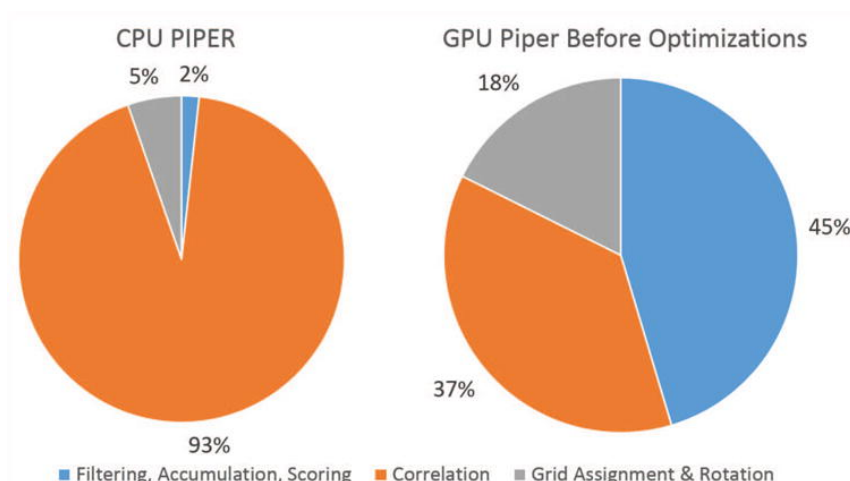


Figura 2.8: Proporções de tempo gasto na execução para a versão CPU otimizada e a versão de 2009 que usa GPUs[17].

Pelo que é necessário garantir ao longo do tempo que as otimizações são atualizadas com mecanismos que as arquiteturas correntes dispõem.

### 2.3.3 AutoDock

O AutoDock [19] é um programa diferente dos até agora indicados, não só por ser específico para interações entre proteína-droga<sup>6</sup> 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. O caso é considerado pois foi desenvolvida uma aceleração do AutoDock usando o GPU[14]. Actualmente, existem duas versões do software: AutoDock 4 e Vina. 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 [34] 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. Igualmente para o clustering dos resultados após o docking ter sido feito, que no Vina é feito sem mostrar ao utilizador detalhes desnecessários. Comparando com o AutoDock 4, em que o utilizador tem de introduzir um conjunto de parametros de input, para manipular o tamanho das estruturas de dados, que estão fixos em tempo de compilação e quanto maiores forem, mais espaço em memória podem ocupar assim como aumenta o tempo de computação do programa. O utilizador pode manipular estes valores ao seu critério no código fonte, tendo de recompilar o AutoDock 4, o que não é user-friendly. O Vina consegue ser superior neste aspeto adaptando-se ao input introduzido.

<sup>6</sup>Uma droga é caracterizada por uma molécula de pequenas dimensões.



O AutoDock Vina foi implementado segundo o paradigma de programação genérica assim como por programação orientada a objetos, na linguagem C++.

### 2.3.3.1 Aceleração do AutoDock com o GPU

Em 2010 foi abordada, à semelhança do que foi feito para o Megadock, uma possível paralelização do AutoDock utilizando o GPU. 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*.

### 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.

### 2.3.4 Dinâmica Molecular em GPUs

Para além das técnicas de docking descritas previamente, em que se recorre à complementaridade de superfície do par de proteínas, existe ainda a alternativa em usar a dinâmica molecular. Esta técnica é adequada para situações em que é pretendido analisar a termodinâmica e energia cinética resultantes da ligação e separação do ligando com o receptor[31]. A dinâmica molecular é um método de simulação de docking que considera



## PLANO DE TRABALHOS PARA A ELABORAÇÃO DA DISSERTAÇÃO

### 3.1 Ciclo de desenvolvimento

Tendo em conta que no decorrer da elaboração não será desenvolvido código de raiz, mas sim otimizar 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, com quatro fases. Este ciclo denomina-se APOD (*Assess, Parallelize, Optimize, Deploy*) [21] 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, fazendo um profile da aplicação para confirmar as identificações efectuadas. No caso da dissertação, é feito um profile do ficheiro bigger.lpi do open-chemera, como está descrito em 3.1.1.
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 [12], 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 em 2.2.3 para maximizar a performance. Também se podem considerar abstrações de outros programas relatados em 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.

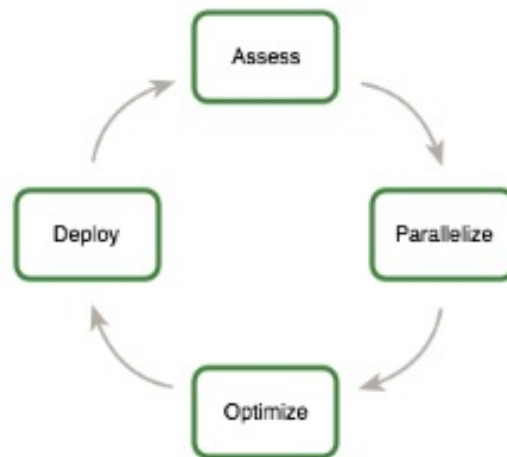


Figura 3.1: Diagrama do ciclo de desenvolvimento APOD[21].

Este ciclo de desenvolvimento será iterado o número de vezes necessário até a performance do BiGGER se enquadrar com os objetivos indicados no capítulo 1.

### 3.1.1 Profiling

Antes de se considerar opções sobre as possibilidades para efectuar a optimização do BiGGER, foi necessário fazer uma averiguação sobre o custo de cada etapa do mesmo em termos de número de chamadas e tempo total de computação. O objetivo é poder determinar zonas de código a paralelizar, sendo esta tarefa o acto de fazer profiling a um programa. O profiling será feito aos ficheiros relacionados com o funcionamento do BiGGER, pois existem elementos do open-chemera que não são importantes para o mesmo, como por exemplo o 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. No caso do Lazarus é possível recorrer a duas formas principais para fazer profiling a um programa:

- **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 FP-Profiler

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 não assumem a mesma qualidade do LazProfiler.

### 3.1.2 Possibilidades de optimização

O programa que serve de interface gráfica ao BiGGER, que se pode fazer a clonagem do repositório em <https://github.com/lkrippahl/Open-Chemera> encontra-se implementado em Free Pascal, 97.6% do código total, ao acordo com o que foi abordado previamente, o trabalho a realizar na elaboração incide sobre o package **docking** mais precisamente às unidades **bogie.pas** e **dockdomains.pas**. O bogie.pas consiste no módulo de docagem geométrica e na unidade dockdomains são determinados os domínios nos três eixos para a simulação da docagem geométrica.

O trabalho poderá abarcar a paralelização de mais unidades presentes no pacote o que só garante melhorias adicionais à performance do Open Chemera, por exemplo na secção 3.2 é abordada uma paralelização à unidade linegrids.pas, que é a unidade onde é feito o cálculo das regiões de superfície e core dos pares assim como a determinação das grelhas para a superfície 3D dos mesmos, para o efeito de complementar a resolução do desafio estipulado na secção. As principais alterações serão, no entanto, focadas nos dois referidos anteriormente.

Face à possibilidade de não existir nenhuma versão do CUDA para programar paralelizações em Free Pascal.

Põe-se de lado esta última alternativa em troca de uma outra que recorra à framework OpenCL, que é suportada pelo Lazarus (IDE a utilizar durante a fase de elaboração do tema).

Poderão vir a ser implementados para a paralelização das duas unidades, kernels que executam operações de mapeamento, em que o espaço de possibilidades a tratar na primeira fase do BiGGER poderá ser associado a uma estrutura de dados (figura 3.2), dividida por blocos em que estes últimos serão constituídos por casas indexadas pelo ID da thread correspondente. Pelo que a indexação geral estará associada a uma fórmula que envolva a posição da thread no bloco e o número de bloco. Para cada uma das posições o core do GPU executará a função que determina se a configuração é aceite ou não. Este esquema de indexação de threads também existe em OpenCL, por invocações próprias na sua sintaxe.

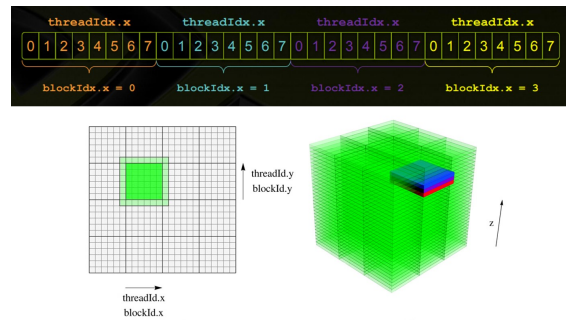


Figura 3.2: Representação gráfica do ID geral de uma thread num array dividido por blocos, em CUDA. Adaptações de [29] [36]

## 3.2 Desafios

Os desafios incidem-se sobre dois pontos:

- **Criação das grelhas tri-dimensionais** : Tal como foi elaborado no capítulo anterior, o passo inicial do BiGGER consiste na criação de grelhas tri-dimensionais, de booleanos, que assumem valor 1 ou 0 se a posição respetiva na grelha corresponde a uma posição atómica da proteína.

Analisando o código que se pode encontrar na unidade dockdomains.pas pode-se verificar que existem uma certa diversidade nas possibilidades de optimização.

O código presente nesta unidade contém um conjunto de *procedures* que executam cadeias de ciclos for/while, em teoria é possível paralelizar o dockdomains recorrendo a kernels que executam operações de map, de forma a reduzir a carga de computação.

Paralelizações adicionais poderão ser feitas na unit linegrids.pas que permitem trazer melhorias extra de performance, no entanto serão alterações complementares, segundo a documentação inicial do código fonte da unidade, os segmentos gerados por esta unidade são referenciados apenas ao eixo z, sendo então uma unidade auxiliar para o BiGGER para a indexação da matriz tri-dimensional por parte deste.

- **Pesquisa de sobreposições** :

*Pending Executar o profiling e apresentar resultados (1 Página com uma tabela)*

## 3.3 Plano de Trabalhos

*Gantt chart?*

## 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. 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.
- [3] 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.
- [4] CUDA Zone. URL: <https://developer.nvidia.com/cuda-zone>.
- [5] cuFFT. 2018. URL: <https://developer.nvidia.com/cufft>.
- [6] Docking (molecular). URL: [https://en.wikipedia.org/wiki/Docking\\_\(molecular\)](https://en.wikipedia.org/wiki/Docking_(molecular)).
- [7] 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-protger-contra-a-doenca-de-alzheimer/>.
- [8] 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.
- [9] 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.
- [10] GPGPU.org. URL: <http://gpgpu.org/about>.
- [11] GPU Applications Catalog. URL: <https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/>.
- [12] M. Harris. Six Ways to SAXPY. URL: <https://devblogs.nvidia.com/six-ways-saxpy/>.
- [13] 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>.
- [14] S Kannan e R Ganji. “Porting AutoDock to CUDA [J]. Evolutionary Computation (CEC)”. Em: *2010 IEEE Congress on*, pp. 1–8.

- [15] 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).
- [16] L. Krippahl. *Integrating protein structural information*. 2003.
- [17] 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.
- [18] E. Lindholm, J. Nickolls, S. Oberman e J. Montrym. "NVIDIA Tesla: A unified graphics and computing architecture". Em: *IEEE micro* 28.2 (2008).
- [19] G. M. Morris. *AutoDock*. URL: <http://autodock.scripps.edu/>.
- [20] J. Nickolls e W. J. Dally. "The GPU computing era". Em: *IEEE micro* 30.2 (2010).
- [21] NVIDIA. *CUDA C BEST PRACTICES GUIDE*. URL: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide>.
- [22] NVIDIA. *CUDA C PROGRAMMING GUIDE*. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide>.
- [23] NVIDIA. *NVIDIA Tesla V100 GPU architecture*. URL: <http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>.
- [24] 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>.
- [25] M. Ohue, Y. Matsuzaki, N. Uchikoga, T. Ishida e Y. Akiyama. "MEGADOCK: an all-to-all protein-protein interaction prediction system using tertiary structure data". Em: *Protein and peptide letters* 21.8 (2014), pp. 766–778.
- [26] *OpenCL*. URL: <https://developer.nvidia.com/opencv>.
- [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] J. Sainio. "CUDA-EASY - a GPU accelerated cosmological lattice program". Em: *Computer Physics Communications* 181 (2010), pp. 906–912.
- [30] 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.



- [31] P. Śledź e A. Caflisch. “Protein structure-based drug design: From docking to molecular dynamics”. Em: *Current opinion in structural biology* 48 (2018), pp. 93–102.
- [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] 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.
- [34] 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.
- [35] N. Wilt. *The CUDA handbook: a comprehensive guide to GPU programming*. Addison-Wesley, 2013.
- [36] C. Zeller. “Cuda c/c++ basics”. Em: *NVIDIA Coporation* (2011).

