Ricardo Alexandre do Rosário Ribeiro

Master of Science

Protein docking GPU acceleration

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Ludwig Krippahl, Full Professor, NOVA University of Lisbon

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

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2018

Resumo

Na área cientifica da Bio-Informática, o estudo das interações entre as proteinas tem sido objeto de interesse. No entanto conseguir determinar computacionalmente e com precisão como é que as mesmas se unem é dificil. Existem, no entanto diversos métodos e algoritmos para prever o ajuntamento das proteinas. Um desses métodos é o BiGGER, criado pelo prof. Ludwig Krippahl, prof. Nuno Palma do BIAL e outros professores integrados no projeto. Este algoritmo assume caracteristicas que lhe dão uma complexidade temporal inferior aos demais.

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 consequencia do aumento significativo de perfomance face à versão sequencial. Em caso de sucesso, a complexidade futura do algoritmo permitirá a adição de mais vantagens face aos seus concorrentes. Por consequencia 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 proteinas em qualquer máquina que tenha uma placa gráfica com as caracteristicas adequadas.

Palavras-chave: proteinas, docagem de proteinas,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 shoul 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 Introduction 1

2 Estado-da-arte 3

2.1 Docking . . . 3

2.1.1 Modelos de docagem em relação à rigidez da superficie . . . 3

2.1.2 Dinamica molecular . . . 4

2.1.3 Fast Fourier Transform . . . 4

2.1.4 O algoritmo BiGGER para docagem de proteinas . . . 5

2.1.5 BoGIE . . . 6

2.1.6 Geometric Hashing . . . 7

2.1.7 ZDOCK . . . 8

2.2 O uso de GPU como ferramenta auxiliar . . . 8

2.2.1 Arquitetura do GPU . . . 9

2.2.2 Modelo de programação . . . 10

2.2.3 Modelo de execução . . . 10

2.2.4 Optimizações . . . 11

2.2.5 APIs para computação acelerada . . . 11

2.3 Docking em GPU . . . 12

2.3.1 Megadock . . . 12

2.3.2 AutoDock . . . 13

3 Plano de trabalhos para a Elaboração da dissertação 15

3.1 Profiling . . . 15

3.2 Possibilidades de optimização . . . 15

3.3 Desafios 16
3.4 Plano de Trabalhos 17

Bibliografia **19**

Lista de Figuras

2.1 Representação em 2D das matrizes resultantes do segundo passo do BoGIE, as células preenchidas a tracejado diagonal correspondem à matriz de superficie, as com pontos correspondem à matriz *core* e a núvem com tracejado continuo representa o corte associado à esfera de van der waals com a proteina localizada ao centro [14]. 7
2.2 Fluxograma sobre o geometric hashing.Tirado de [2] 8
2.3 Esquema ilustrativo da diferença entre o hardware do CPU e do GPU, demonstrando também a diferença de cores entre os 2. Imagem da NVIDIA 9
2.4 Esquema da arquitetura do GPU. Neste caso a arquitetura corresponde ao Fermi GF100. [21] 10

3.1 Representação gráfica do ID geral de uma thread num array divido por blocos, em CUDA. Adaptações de [17] [22] 16

Lista de Tabelas

2.1 Tempos de cálculo das versões iniciais do Megadock em relação ao ZDOCK [13]. De notar que os testes foram realizados em 2014, pelo que actualmente em 2018, pelos mesmos motivos apontados 13

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, consectetuer 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, consectetuer 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

aaa	acornym aaa.
aab	acornym aab.
aba	acornym aba.
abbrev	abbreviation of a longer text.
AEU	adipiscing elit ut.
AFM	aenean faucibus morbi.
AMD	a magna donec.
ANP	ac nunc praesent.
ATG	amet tortor gravida.
AVF	adipiscing vitae felis.
bbb	acornym bbb.
CAS	curabitur auctor semper.
CDG	curabitur dictum gravida.
CEA	congue eu accumsan.
CIV	consectetuer id vulputate.
DIA	duis eget orci.
DNM	dolor nulla malesuada.
DNMC	duis nibh mi congue.
DRN	dignissim rutrum nam.
EII	est iaculis in.
ENE	et netus et.
EPA	eu pulvinar at.
ESQ	eleifend sagittis quis.
ESV	eget sem vel.
ETS	eu tellus sit.

FUP	fringilla ultrices phasellus.
LID	lorem ipsum dolor.
LNE	libero nonummy eget.
LUB	leo ultrices bibendum.
LVU	lectus vestibulum urna.
MAC	mollis ac nulla.
MFA	malesuada fames ac.
MNA	mauris nam arcu.
MTS	morbi tristique senectus.
NDV	nulla donec varius.
NPH	neque pellentesque habitant.
OER	orci eget risus.
PEV	purus elit vestibulum.
PIS	placerat integer sapien.
PQV	pretium quis viverra.
SAO	sit amet orci.
SNE	sem nulla et.
STC	sit amet consectetuer.
TEM	turpis egestas mauris.
ULC	ut leo cras.
UPA	ut placerat ac.
VAE	vehicula augue eu.
VMR	viverra metus rhoncus.
xpto	and extension of a xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto xpto.

Capítulo 1

Introduction

Capítulo 2

Estado-da-arte

2.1 Docking

2.1.1 Modelos de docagem em relação à rigidez da superficie

Com o passar do tempo cada vez mais algoritmos e respetivas adaptações para simular a docagem dos complexos de proteinas têm surgido, adoptando modelos que formulam hipóteses em relação às caracteristicas dos elementos envolventes.

Um algoritmo pode ser classificado em função da forma que trata a rigidez da superficie dos pares de proteinas 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 docagem: flexivel, semi-flexivel e rigida.

2.1.1.1 Docagem Flexivel

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

2.1.1.2 Docagem Semi-Flexivel

Um dos corpos é considerado rigido e o outro não, em situações normais este tipo de algoritmos trata o ligando como se fosse a proteina flexivel, 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 flexiveis.

2.1.1.3 Docagem Rigida

O par é considerado como sendo rigido na sua integridade, sendo também considerado que na docagem entre os dois corpos uma das proteinas 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 superficies de ambos como rigidos, considera-se que ocorrem variações na superficie 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 superficies de ambos os elementos e traduzir para uma base de dados de docagem [4].

2.1.2 Dinamica molecular

2.1.3 Fast Fourier Transform

Algoritmo baseado em previsões geométricas sobre o complexo formado após a interação entre as proteinas, representando o método a partir do qual ocorreram adaptações que levaram origem à criação do BiGGER [14]. É utilizado em muitos softwares especificos para *protein docking* (FTDock, 3D-Dock,GRAMM, ZDOCK,..)[2].

A origem deste método remonta ao artigo de Katchalski Katzir et al [5] onde se considera que ambos os pares são corpos rigidos, o que no âmbito do ponto 2.1.1 pode-se assumir como docagem rigida.

À semelhança do BiGGER, este método também recorre a uma estimativa da complementaridade entre as superficies dos pares que compõem a PPI, apenas existe como requisito o conhecimento da estrutura tri-dimensional destes.

A metodologia com que são determinadas as possibilidades consiste, de forma resumida, nos passos:

1. Considerando duas moléculas a e b, correspondentes ao receptor e ao ligando, é determinada uma projeção das coordenadas atómicas de ambos para uma grelha tri-dimensional (N^3).

   São também determinadas duas funções discretas sobre as coordenadas, em que cada ponto da grelha assume o valor 0 ou 1, se a coordenada corresponde a uma posição interna à molécula e 0 caso contrário.

   O limite para atribuição dos valores está definido por um raio r associado a uma nuvem de van der waals, que assume uma curvatura semelhante à que se pode determinar na figura 2.4.

2. Determinação da região de fronteira, em que a função discreta determinada no ponto anterior é estendida para suportar os pontos fronteiriços: 1 neste passo é atribuido a coordenadas atómicas localizadas na fronteira, uma variável associada

a coordenadas internas e 0 a externas. A esta função é designada função distreta de Fourier (DFT).

3. 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 $360x360x180\Delta^3$.

Em termos de complexidade temporal, executar estes passos assume uma ordem de complexidade $O(N^3 * log_2(N^3))$[6].

Considerou-se fazer um estudo deste método importante pois tal como foi explicitado no principio desta secção foi a partir do trabalho de Katchalski Katzir e colegas sobre o FFT que foi estudada a abordagem para o BiGGER[6].

Tendo em conta que os passos aqui descritos e as fases do BiGGER descritas na secção 2.1.4 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 o FFT aquando na descrição de resultados obtidos na fase da elaboração.

É esperado que o Open-chemera com o BiGGER paralelizado assuma uma perfomance superior a qualquer um dos casos estudados, que utiliza cuFFT .

2.1.4 O algoritmo BiGGER para docagem de proteinas

Este algoritmo (acrónimo para *Bimolecular complex Generation with Global Evaluation and Ranking*) [6] de acordo com a classificação referida no ponto 2.1.1 enquadra-se na doutrina de docagem rígida[14].

Permite resolver o problema de conseguir prever o ajuntamento dos complexos proteicos, consistindo essencialmente em dois passos: o primeiro efectua uma redução de possiveis 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 2.1.5.

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 espécies de algoritmos, o BiGGER assume-se com perfomance 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 [14].

O tempo de execução do algoritmo, segundo os autores do mesmo, estava situado entre as 2H e as 8H, dependendo do par de proteinas 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)[14].

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.

2.1.5 BoGIE

Acrónimo para *Boolean Geometric Interaction Evaluation*[6] [14], é um algoritmo de pesquisa em grelha utilizado na primeira fase do BiGGER, que é referido no ponto seguinte, mais precisamente na amostragem da população total de configurações possiveis 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 celula corresponde a uma parcela da proteina cujo centro se encontra a uma distancia tri-dimensional, designada por esfera de Van der Waals, de qualquer outro átomo pertencente a outra proteina, 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 superficie (*surface matrix*) e a matriz central (*core matrix*) tal como está ilustrado graficamente na figura 2.4.

Os elementos celulares que ocupam a matriz de superficie 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 superficie molecular da proteina 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 ultimo 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.

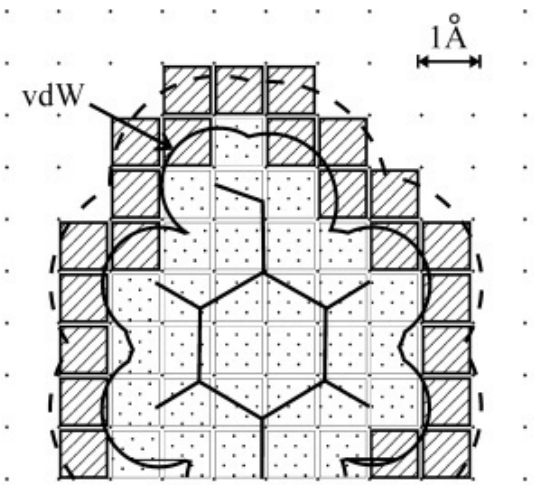

Figura 2.1: Representação em 2D das matrizes resultantes do segundo passo do BoGIE, as células preenchidas a tracejado diagonal correspondem à matriz de superficie, as com pontos correspondem à matriz *core* e a núvem com tracejado continuo representa o corte associado à esfera de van der waals com a proteina localizada ao centro [14].

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

2.1.6 Geometric Hashing

Este algoritmo é shape-explicit, ao contrário do BiGGER, que é surface-explicit - as superficies de ambos os elementos do par são quantificadas e representadas por valores binários nas superficies core e surface.

A metodologia deste método divide-se em dois passos: Pré-processamento e Reconhecimento [2]. A fase de pre-processamento consiste em identificar os pontos criticos na superficie do ligando e a partir destes definir frames de coordenandas locais. Sobre estes frames, serão feitas indexações com base nos pontos criticos vizinhos a um selecionado. Os indices 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 correspondencia entre as coordenadas do ligando, através da hash table. Se houver demasiadas correspondencias, existe uma grande possibilidade de as curvaturas de superficie serem semelhantes, e é feita uma verificação extra com esse ambito [18]. Na figura 2.2, 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 algoritmos anteriores

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 criticos a considerar. Os tempos de execução são baixos, sendo na ordem dos minutos independentemente da complexidade da previsão da docagem [4]. 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.

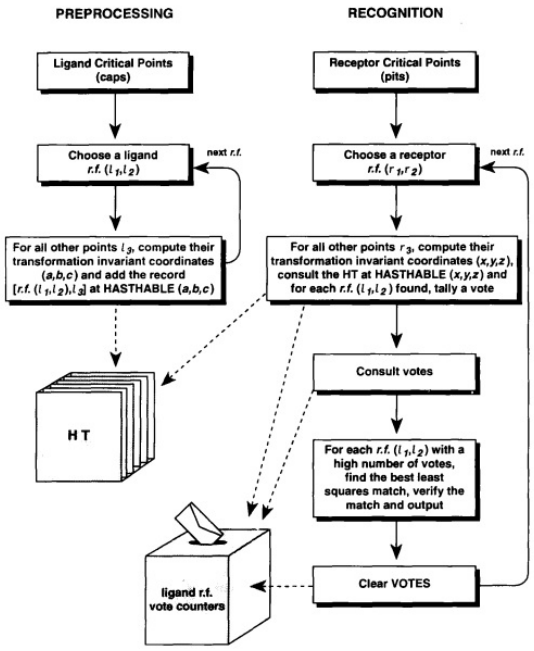

Figura 2.2: Fluxograma sobre o geometric hashing.Tirado de [2]

2.1.7 ZDOCK

2.2 O uso de GPU como ferramenta auxiliar

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 à renderização para o gaming, que são igualmente intensivos demais para o CPU. Exemplos de aplicações podem variar de cálculo financeiro até aplicações bio-informáticas, como é o caso do docking.

Consegue ser mais poderoso a executar instruções com custos de complexidade temporal elevado do que necessariamente o CPU, que tem um número de cores muito menor do que o GPU. [11]

Um programador que pretenda implementar uma paralelização para a versão sequêncial de um dado programa, recorrendo ao CPU, necessita de conhecimentos de programação com *threads* assim como gerir o acesso exclusivo a variáveis partilhadas entre as mesmas, o que pode ser feito por gestão de locks ou por instruções atómicas.

O resultado consiste num programa paralelizado mas com um speedup limitado comparando com uma solução que use aceleradores no GPU.

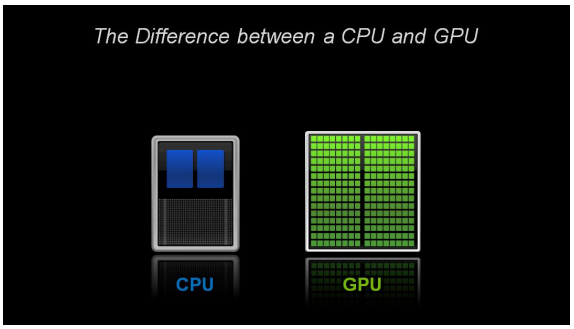

Figura 2.3: Esquema ilustrativo da diferença entre o hardware do CPU e do GPU, demonstrando também a diferença de cores entre os 2. Imagem da NVIDIA

2.2.1 Arquitetura do GPU

O GPU acaba por ser uma ferramenta mais potente para paralelizar programas já que a sua arquitetura é consistente em uma quantidade variável de streamming multi-processadores (*streaming multi-processors*) [15]. Que por si só têm um conjunto de processadores escalares (SPs) que são também conhecidos como os cores do GPU. É possivel ter até 512 cores no total, se a arquitetura for da espécie Fermi. [21]. Está presente ainda uma zona de memória global que pode variar entre MBs e GBs, e pode ser partilhada entre os diversos SPs.

A arquitetura do GPU, conceptualmente, pode se dividir em duas componentes :

- **Streaming Multiprocessors** : responsáveis por executar os CUDA kernels. Funcionam como as *cores* do CPU mas com caracteristicas diferentes. Têm ciclos de relógio mais baixos, mas suportam paralelismo ao nível de instrução. A evoluição dos SMs tem sido notada desde o aparecimento de hardware capaz de executar CUDA, em 2006. Em que se vão melhorando as capacidades das suas componentes: A destacar a quantidade de registos que os SMs vão dispondo, a cache L1 e o número de *cores* de execução [20].

- **Caches e memória**: Em que 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 politica de substituição write-back . Esta cache responde

a pedidos de instruções load, store assim como instruções atómicas de ambos SM e respetivas caches L1, preenchendo de forma igual as respetivas caches [9] Os beneficios da cache L2 para o CUDA regem-se pelo aumento do armazenamento para instruções de load e store globais [21].

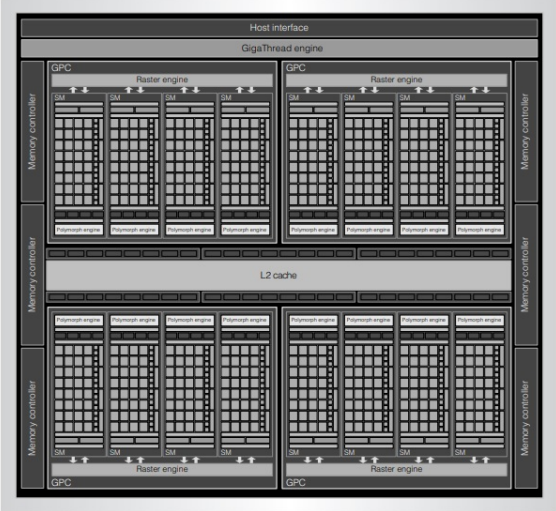

Figura 2.4: Esquema da arquitetura do GPU. Neste caso a arquitetura corresponde ao Fermi GF100. [21]

Tendo em conta a quantidade de threads individuais que têm de ser geridas e executadas em diferentes programas de forma eficiente, é empregado pelos SMs uma arquitetura especifica para o efeito, denominada SIMT (*Single-Instruction Multiple-Thread*). [7] Esta arquitetura permite a criação, gestão, escalonamento e execução de threads concurrentes em grupos de 32 cada. Estes grupos assumem o conceito de *warps*, podendo cada bloco de threads do CUDA ter 1 ou mais warps. [10]

Porém continuamos a necessitar do CPU para comunicar com o GPU, em que este último funciona como *device* e o CPU funciona como *host* numa interface de comunicação de dados entre os 2. Sendo assim é essencial usar o GPU como utensilio na paralelização de algoritmos que façam computações muito pesadas, como é o caso das etapas presentes nos algoritmos destacados nas secções anteriores.

2.2.2 Modelo de programação

[16]

2.2.3 Modelo de execução

[16]

2.2.4 Optimizações

De acordo com [16], existem três aspetos a considerar quando pretendemos optimizar um programa recorrendo a um GPU:

- **Floating Point throughput** : É dependente da percentagem de instruções de floating point incidentes. A performance aqui é optimal quando os streaming processors (SPs) estão totalmente ocupados o que é notado em aplicações com muitas threads à disposição, não precisando de muitas sincronizações entre elas, não afectando a banda larga da memoria global. O speedup do kernel pode ser melhorado neste contexto reduzindo o número de instruções que não fazem contributos para computação dos dados.

- **Latência de memória global** : É possivel melhorar a latência da memória global em termos de acessos, criando um número de threads necessário para manter os SPs ocupados enquanto existem threads à espera de aceder à memória global. Este número depende da percentagem de acessos e outras instruções de grande latencia presente no programa. Quanto menor for esta percentagem, menos threads serão necessárias para atingir a ocupancia maxima dos SPs. Existe ainda outro factor que pode expor latencia de memória: O limite em registros e memória partilhada que os SMs dispõem podem impor uma limitação no número de threads que podem ser criados para esconder a latência.

- **Bandwidth da memória global** : Esta variável pode limitar o throughput do sistema e aumentar a quantidade de threads não resolve o problema. Igualmente se pode referir que diminuir a carga de pressão na bandwitdh da memória global implica recorrer a mais registos e memória partilhada dos SMs para reutilização de dados, o que limita o número de threads que podem correr em simultaneo. Pelo que é necessário um balanco entre as duas medidas referidas anteriormente.

2.2.5 APIs para computação acelerada

Em termos de programação em GPUs existe como API o CUDA (*Compute Unified Device Architecture*) que foi implementado pela NVIDIA

O CPU executa a parte sequêncial do programa enquanto que o GPU, com a sua quantidade numerosa de cores, executa a parte que requer computação mais intensiva como por exemplo multiplicação de matrizes (matmult), através de invocações especiais que se denominam *kernels*.[11]

De notar que é necessário fazer previamente as alocações respetivas de memória do host para o device, que são feitas por parte do CPU.

Um programador que queira utilizar CUDA para paralelizar o seu programa, deve transferir a toolkit respetiva ao sistema operativo em que pretende realizar o projeto, esta toolkit está disponivel na página da NVIDIA.

No entanto também existem mais APIs/frameworks para o efeito, como por OpenCL(*Open computing language*) que é adequado para programação paralela em Free pascal enquanto que o CUDA é mais utilizado para C/C++ ou até mesmo Python (pyCUDA).

Ambos são semelhantes no sentido de implementar o protocolo de comunicações entre o CPU e o GPU, e de permitir invocar kernels para este último computar a parte intensiva.

2.3 Docking em GPU

Já existem vários softwares e artigos dedicados à introdução de paralelizações ao FFT [15] o que muitos têm em comum é o facto de utilizarem CUDA, neste caso utilizam CuFFT. O CuFFT é uma versão própria do CUDA que fornece as implementações necessárias para que a perfomance do algoritmo seja aumentada em 10x, face às versões que utilizam o CPU para o processamento respetivo[1] .

2.3.1 Megadock

Um exemplo desta espécie de softwares especificos para docagem é o Megadock 4.0, este software é de origem japonesa, que melhora em relação ao Megadock 1.0 no sentido de ter implementações no ambito da computação acelerada [13].

Este programa é adequado para máquinas que têm muitos cores de GPU e CPU á disposição, caracteristicas tipicas de supercomputadores.

Foi implementado para GPU não só por CUDA como também por MPI (Message Passing Interface) e OpenMP. O MPI, tal como a sigla expecifica, é uma API de interface de comunicação de mensagens entre processos a correr nos nós de um cluster[3]. O Megadock 4.0 é um enhancement em relação à versão 1.0 no sentido de suportar o modelo master-worker. O funcionamento do Megadock 4.0 envolve a criação de um processo master que faz a aquisição de uma lista de pares de proteinas e cria jobs incidentes sobre a lista para os workers. 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 distribuição pelos GPUs de cada nó é feita por CUDA e pelos CPUs por OpenMP [12]. Uma das vantagens que este protocolo assume é a tolerancia de 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. Em termos de performance e tempos de execução, os testes efectuados em 2014 (tabela 2.1) 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 (1936) e no mesmo número para estruturas ligandas (14400). No caso do Megadock 4.0, o valor a considerar para o tempo de cálculo ronda as 5.71h para meio milhão de

	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 2.1: Tempos de cálculo das versões iniciais do Megadock em relação ao ZDOCK [13]. De notar que os testes foram realizados em 2014, pelo que actualmente em 2018, pelos mesmos motivos apontados

jobs e 11.51H para 1 milhão de jobs. Para este último teste foi utilizado a versão 2.3 do TSUBAME[12].

2.3.2 AutoDock

O AutoDock [8] consiste numa plataforma de software para auxilio na previsão da forma como pequenas moleculas, como é o caso das constituintes das drogas, se unem a receptores. Actualmente, existem duas versões do software: AutoDock 4 e Vina. O AutoDock 4 está divido 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 calculos previos para obter as grelhas que o autodock necessita para desempenhar as suas funções. O AutoDock Vina é diferente do 4 no sentido de não efectuar o cálculo das grelhas de forma prévia mas sim instantaneamente. Entre as optimizações efectuadas encontra-se o uso de um método quasi-Newton BFGS [19]

Capítulo 3

Plano de trabalhos para a Elaboração da dissertação

3.1 Profiling

3.2 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 dominios 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 3.3 é abordada uma paralelização à unidade linegrids.pas, que é a unidade onde é feito o cálculo das regiões de superficie e core dos pares assim como a determinação das grelhas para a superficie 3D dos mesmos, para o efeito de complementar a resolução do desafio distipulado na secção. As principais alterações serã, 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 3.1), dividida por blocos em que estes últimos serão constituidos 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.

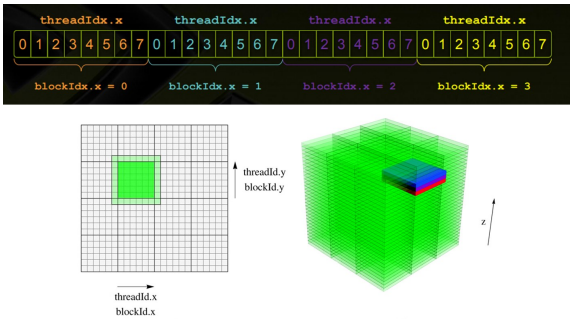

Figura 3.1: Representação gráfica do ID geral de uma thread num array divido por blocos, em CUDA. Adaptações de [17] [22]

Este esquema de indexação de threads também existe em OpenCL, por invocações próprias na sua sintaxe.

3.3 Desafios

Os desafios incidem-se sobre dois pontos:

- **Criação das grelhas tri-dimensionais** : Tal como foi elaborado no capitulo 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 proteina.

  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 possivel 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** :

3.4 Plano de Trabalhos

Bibliografia

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

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

[3] W. D. Gropp, W. Gropp, E. Lusk e A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.

[4] H. Inbal, M. Buyong, W. Haim e N. Ruth. "Principles of docking: An overview of search algorithms and a guide to scoring functions". Em: *Proteins: Structure, Function, and Bioinformatics* 47.4 (), pp. 409–443. DOI: 10.1002/prot.10115. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/prot.10115. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/prot.10115.

[5] 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.

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

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

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

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

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

[11] C. Nvidia. "Nvidia cuda c programming guide". Em: *Nvidia Corporation* 120.18 (2011), p. 8.

[12] 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. DOI: 10.1093/bioinformatics/btu532. eprint: /oup/backfile/content$_public/journal/bioinformatics/30/22/10.1093_bioinformatics_btu532/2/btu532.pdf. URL: http://dx.doi.org/10.1093/bioinformatics/btu532.

[13] 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.

[14] 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.

[15] 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.

[16] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk e W.-m. W. Hwu. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". Em: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM. 2008, pp. 73–82.

[17] J. Sainio. "CUDAEASY - a GPU accelerated cosmological lattice program". Em: *Computer Physics Communications* 181 (2010), pp. 906–912.

[18] 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.

[19] 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.

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

[21] C. M. Wittenbrink, E. Kilgariff e A. Prabhu. "Fermi GF100 GPU architecture". Em: *IEEE Micro* 31.2 (2011), pp. 50–59.

[22] C. Zeller. "Cuda c/c++ basics". Em: *NVIDIA Coporation* (2011).