



UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Geometric Representation and Detection Methods of Cavities on Protein Surfaces

Sérgio Emanuel Duarte Dias

Tese para obtenção do Grau de Doutor em
Engenharia Informática
(3º ciclo de estudos)

Orientador: Prof. Doutor Abel João Padrão Gomes

Covilhã, junho de 2015

To my family

Thesis prepared at Instituto de Telecomunicações and MediaLab, Universidade da Beira Interior, and submitted to Universidade da Beira Interior for public defense in doctoral exams.

Work financed by the Portuguese research council, Fundação para a Ciência e Tecnologia, through grant contract SFRH-BD-69829-2010 under the programme QREN POPH – Type 4.1 – Advanced Training, co-funded by the European Social Fund and by national funds from the Portuguese Ministry for Education and Science (Ministério da Educação e Ciência).

I would like to thank NVIDIA Corporation for its support, from which resulted the donation of the most recent professional graphics cards, as well as the access to its GPU Cluster used during my PhD programme at the University of Beira Interior.

I would also thank to the anonymous reviewers for their suggestions, who significantly contributed to improve the research work underlying my thesis here presented.



Acknowledgements

This thesis would not have been possible without the help of some people.

To my advisor, Professor Abel Gomes, I would like to express my utmost gratitude for his constant guidance, patience, encouragement, and support. Without his expertise, research insight, and invaluable help, this thesis would not have been possible.

Also I would like to acknowledge the financial support of FCT, University of Beira Interior in academic terms, and specially the NVIDIA Corporation for offering the wonderful graphics cards that allowed me to implement all the algorithms described in this thesis.

To my family, and especially to my parents, who incessantly and at all times have accompanied, helped and encouraged me to pursue this purpose in my academic journey, which culminated in this dissertation. For them, here it is the result of my effort, my love and gratitude. For them, I have the desire and promise to make every moment of my life the best.

To all of you "muito obrigado".

Resumo

Em geral, os organismos vivos são constituídos por células, enquanto que as células são compostas por moléculas. As moléculas desempenham um papel fundamental nos processos bioquímicos que sustentam a vida. As funções de uma molécula dependem não só da sua interacção com outras moléculas, mas também dos locais na sua superfície onde estas interacções têm lugar. Na verdade, essas interacções são a força motriz de quase todos os processos celulares.

As interacções entre moléculas ocorrem em regiões específicas das respectivas superfícies moleculares, comumente designados por locais de acoplamento (*binding sites*, do inglês). O desafio está em saber quais são os locais de acoplamento compatíveis entre moléculas. Na verdade, não basta que haja compatibilidade geométrica no acoplamento entre moléculas. É preciso também que haja compatibilidade físico-química nos locais de acoplamento entre moléculas.

A maioria dos (mas não todos) locais de acoplamento de uma molécula (por exemplo, uma proteína) correspondem a cavidades na sua superfície; inversamente, a maioria das (mas não todas) cavidades correspondem a locais de acoplamento. Esta tese aborda essencialmente algoritmos de deteção de cavidades em superfícies de proteínas. Isso significa que estamos principalmente interessados em métodos geométricos capazes de identificar cavidades em proteínas, enquanto (possíveis) locais de acoplamento preliminares para os seus ligandos (*ligands*, do inglês).

Determinar cavidades em proteínas tem sido um grande desafio em computação e modelação molecular, biologia computacional e química computacional. Isto explica-se pela forma das proteínas, a qual parece bastante imprevisível, face às muitas oscilações de forma decorrentes da combinação dos seus átomos. Estas pequenas características de forma da superfície de uma proteína são bastante elusivas, porque são muito pequenas quando comparadas com cavidades (enquanto possíveis locais de acoplamento). Isto significa que o conceito de curvatura (um descritor de forma local) não pode ser usado como uma ferramenta para detetar essas cavidades na superfície de proteínas. Consequentemente, há que utilizar descritores de forma não-local (ou zonal) se se quiser ter sucesso na determinação de tais cavidades.

Nessa linha de pensamento, esta tese explora a aplicação da teoria matemática de campos escalares, incluindo a sua topologia, como a pedra basilar para o desenvolvimento dos algoritmos de deteção de cavidades aqui descritos. Além disso, para efeitos de visualização gráfica, introduz-se um algoritmo de triangulação de superfícies moleculares em GPU.

Palavras-chave

Arquitetura de hardware gráfico
Algoritmo geométrico
Campo escalar
Cavidade em proteína
Computação Gráfica
CUDA
Descritor de forma
Geometria Computacional e Modelação de Objectos
Local de acoplamento
Multi-threading
OpenCL
Ponto Crítico
Processamento paralelo
Superfície de Blinn
Superfície do tipo Gaussiano
Superfície implícita
Superfície molecular
Triangulação

Abstract

Most living organisms are made up of cells, while cells are composed by molecules. Molecules play a fundamental role in biochemical processes that sustain life. The functions of a molecule depend not only on its interaction with other molecules, but also on the sites of its surface where such interactions take place. Indeed, these interactions are the driving force of almost all cellular processes.

Interactions between molecules occur on specific molecular surface regions, called binding sites. The challenge here is to know the compatible sites of two coupling molecules. The compatibility is only effective if there is physico-chemical compatibility, as well as geometric compatibility in respect to docking of shape between the interacting molecules.

Most (but not all) binding sites of a molecule (e.g., protein) correspond to cavities on its surface; conversely, most (but not all) cavities correspond to binding sites. This thesis essentially approaches cavity detection algorithms on protein surfaces. This means that we are primarily interested in geometric methods capable of identifying protein cavities as tentative binding sites for their ligands.

Finding protein cavities has been a major challenge in molecular graphics and modeling, computational biology, and computational chemistry. This is so because the shape of a protein usually looks very unpredictable, with many small downs and ups. These small shape features on the surface of a protein are rather illusive because they are too small when compared to cavities as tentative binding sites. This means that the concept of curvature (a local shape descriptor) cannot be used as a tool to detect those cavities. Thus, more enlarged shape descriptors have to be used to succeed in determining such cavities on the surface of proteins.

In this line of thought, this thesis explores the application of mathematical theory of scalar fields, including its topology, as the cornerstone for the development of cavity detection algorithms described herein. Furthermore, for the purpose of graphic visualisation, this thesis introduces a GPU-based triangulation algorithm for molecular surfaces.

Keywords

Computational Geometry and Object Modeling
Computer Graphics
Critical point
CUDA
Binding site
Blinn surfaces
Gaussian-like surface
Geometric Algorithm
Graphics Hardware Architecture
Implicit surface
Molecular surfaces
Multi-threading
OpenCL
Parallel Processing
Protein Cavity
Scalar field
Shape descriptor
Triangulation

Resumo Alargado

Este resumo alargado faz uma súmula, em Língua Portuguesa, do trabalho de investigação descrito nesta tese de doutoramento. Começa-se por fazer o enquadramento da tese. Depois, formula-se o problema que se pretende resolver com esta tese, colocando-se então a hipótese de investigação (*thesis statement*), bem como se avança com uma possível solução para o problema previamente formulado. O resumo termina com uma discussão breve das principais conclusões e a apresentação de algumas linhas de investigação futura.

Enquadramento da Tese

A maioria dos organismos vivos são compostos por células que, por sua vez, são constituídas por moléculas, cujas interações são fundamentais nos processos bioquímicos que dão suporte à vida. Essencialmente, estas moléculas interagem entre si, sendo que as suas interações estão na base de processos celulares como, por exemplo, a transcrição de DNA. Estas interações moleculares ocorrem em regiões específicas, designadas por locais de ligação (*binding sites*, do inglês), mas nem todos os locais de ligação de uma molécula são compatíveis com outra molécula. Este processo designa-se por ligação molecular e têm por objectivo localizar os locais na superfície de uma dada proteína onde algum ligante (*ligand*, do inglês) se ligará. Para que esta ligação se concretize é necessário que haja compatibilidade entre as duas moléculas tanto a nível geométrico, como a nível bioquímico.

Isso requer um profundo entendimento de como uma molécula se liga a uma região específica de outra molécula, e de como se pode utilizar essa informação para prever que tipo de moléculas se devem ligar a essa região específica. Como é do conhecimento geral, a forma de uma molécula desempenha um papel fundamental na função de reconhecimento biomolecular, em particular em interações não covalentes entre moléculas. Contudo, devido à complexidade estrutural das moléculas, a maioria destes algoritmos não é capaz de distinguir entre locais de ligação corretos e falsos em determinadas circunstâncias da simulação. Isto acontece porque os atuais descritores de forma não são capazes de descrever a forma de uma molécula, a não ser a nível local (e.g., cálculo da curvatura num dado ponto da superfície molecular) or global (e.g., cálculo da localização de ocos e túneis). Estes descritores são incapazes de detetar outras cavidades como é o caso de cavidades em forma de bolsa (*pockets*, do inglês) ou, mesmo cavidades mais abertas. Em suma, os métodos existentes não são capazes de descrever a forma de uma molécula sem ambiguidade.

Neste momento existem mais de 60.000 estruturas de moléculas (proteínas) já conhecidas e disponíveis na web, embora o número de conjuntos de dados (*datasets*) molec-

ulares disponíveis continue a aumentar. Estas estruturas moleculares são importantes não só para ajustar e validar os atuais e futuros métodos computacionais de detecção de cavidade em proteínas, mas também para detetar a localização de cavidades em moléculas para as quais não são conhecidas completamente as respetivas estruturas. Isto é particularmente importante em moléculas com um número elevado de átomos, i.e., moléculas com mais 0.5 milhão de átomos.

Assim, o principal foco deste trabalho é o de localizar cavidades (ou locais de ligação potenciais) na superfície de uma dada molécula. No entanto, ao contrário do que acontece com outros algoritmos, procurar-se-á que a detecção de cavidades seja feita sem ambiguidade. Usar-se-á para isso descritores intrínsecos de forma associados ao campo de densidade eletrónica de cada molécula.

Descrição do Problema

Existem três grandes famílias de algoritmos geométricos para a detecção de cavidades em superfícies moleculares: algoritmos baseados em grelha, algoritmos baseados em esferas, e algoritmos baseados em triangulações [KG07]. Em geral, estes algoritmos geométricos são muito exigentes em termos de cálculo matemático, em particular se o número de átomos vai além de alguns milhares. Esta necessidade de desempenho computacional agudiza-se quando se utiliza a voxelização do domínio onde jaz a molécula, uma vez que a complexidade computacional do algoritmo tende a ser cúbica, a menos que se tire proveito de meios de computação paralela, como é o caso da arquitetura CUDA das placas gráficas programáveis atuais.

Mas o principal desafio dos atuais algoritmos de detecção de cavidades em proteínas não é tanto a sua complexidade computacional, mas outrossim a sua incapacidade em discriminar entre resultados (i.e., cavidades) verdadeiros e falsos. Isto acontece, principalmente, porque os descritores de forma atualmente em uso na área da computação gráfica molecular, biologia computacional e bioinformática não são capazes de captar a forma zonal de uma molécula de maneira correta, ou, se se quiser, de maneira adequada ao problema. Consequentemente, não é exequível efetuar a segmentação de uma dada superfície molecular em função dos seus locais de ligação (ou cavidades).

Com esta tese doutoral pretende-se introduzir novas maneiras de compreender, representar, e analisar a forma de uma superfície molecular quer a nível local, quer a nível zonal e global. Para ser mais específico, pretende-se investigar a existência de descritores intrínsecos de forma que permitam resolver os problemas de ambiguidade na interpretação da forma molecular por parte dos atuais descritores de forma.

Na verdade, o grande problema colocado pela detecção de cavidades na superfície de uma proteína reside na aparente inexistência de uma teoria matemática para calcular a curvatura zonal e global de superfícies, visto que a curvatura pode ser calculada em cada ponto de uma superfície suave ou diferenciável, mas não em termos de suas zonas ou

no seu todo. De menor importância é a falta de desempenho dos algoritmos atualmente existentes, visto terem sido desenhados para computadores de cálculo sequencial.

Hipótese de Investigação

Nos trabalhos de investigação que conduziram a esta tese, procurou-se explorar novos descritores de forma para a deteção de cavidades em superfícies moleculares, principalmente aqueles sustentados em matemática. Pode até dizer-se que esta abordagem mais matematizada acabou por nos levar a uma nova categoria de métodos de deteção de cavidades. Na sua essência, esta categoria de métodos baseia-se na teoria dos campos escalares, em particular na topologia de campo escalares. Neste pressuposto, a hipótese de investigação (*thesis statement*, do inglês) que conduziu à elaboração da presente tese pode ler-se como se segue:

É possível detetar cavidades (i.e., potenciais locais de ligação) em superfícies moleculares de proteínas utilizando descritores de forma intrínsecos com base na teoria matemática dos campos escalares.

Em termos mais específicos, pretende-se investigar as relações entre os pontos críticos do campo escalar fora da superfície molecular e suas cavidades. É nossa intenção demonstrar que o cálculo dos pontos críticos permite identificar a localização exata de cavidades na superfície de qualquer proteína. Além disso, espera-se que este cálculo possa ser feito sem a utilização de qualquer grelha 3D, ou de qualquer revestimento à base de esferas, ou mesmo de qualquer triangulação.

Plano de Investigação

No decurso da investigação para provar a hipótese acima mencionada, houve que passar por uma série de etapas, como a seguir se descreve:

- *Superfícies Moleculares e Campos de Densidade Eletrónica.* Esta etapa visava, em primeiro lugar, encontrar uma formulação matemática adequada para superfícies moleculares, bem como uma formulação matemática apropriada para representar e modelar os campos de densidade eletrónica gerados por moléculas. Constatou-se rapidamente que a teoria dos campos escalares fornecia uma formulação matemática integradora daquelas duas formulações, ou seja, serve não só para representar a superfície de qualquer proteína, mas também o seu campo de densidade eletrónica.
- *Algoritmo de Triangulação Molecular.* Para que se pudesse visualizar as moléculas e suas cavidades em 3D, desenvolveu-se um algoritmo de triangulação de super-

fícies moleculares resultantes da soma de funções (quasi-) Gaussianas. Este algoritmo foi inspirado na formulação de Blinn [Bli82]. O requisito de processamento em tempo real levou-nos ao seu desenvolvimento em GPU, com o recurso à CUDA (*Compute Unified Device Architecture*).

- *Descritores Intrínsecos de Forma.* Um descritor intrínseco de forma é invariante a transformações geométricas, i.e., rotações e translações. Estes descritores são comumente usados em computação geométrica, mas não tanto assim em biologia computacional e química computacional. Exemplos destes descritores são as harmónicas esféricas, o operador de Laplace-Beltrami ou, ainda, a curvatura. No entanto, estes descritores de forma têm uma natureza pontual (i.e., ponto a ponto) localizada na superfície molecular, pelo que não levam em conta o espaço circundante da superfície molecular. Ou seja, estes descritores são inadequados para identificar cavidades na superfície molecular. Por isso, é nosso objetivo explorar a teoria dos campos escalares e a sua topologia no domínio (onde jaz a superfície molecular) no sentido de encontrar descritores intrínsecos de forma que identifiquem cavidades moleculares.
- *Algoritmos de Detecção de Cavidades Moleculares.* Em conformidade com as etapas anteriores, desenvolver-se-á algoritmos de deteção de cavidades moleculares com base em descritores intrínsecos de forma. O primeiro algoritmo utiliza duas superfícies implícitas de um conjunto de nível gerado pelo campo de densidade eletrónica da molécula. Esta técnica resolve o problema da ambiguidade dos métodos baseados em grelha (i.e., domínio voxelizado). O segundo algoritmo utiliza também o campo de densidade eletrónica da molécula, mas as cavidades são detetadas através da identificação dos pontos críticos (i.e., topologia) do referido campo de densidade eletrónica. Esses algoritmos também foram implementados em CUDA.
- *Elaboração Escrita da Tese.* A tese foi sendo escrita no decurso do programa de doutoramento, e, portanto, ao ritmo da publicação de artigos científicos. Daí que os capítulos centrais desta tese tenham dado lugar a artigos publicados, ou em fase de revisão, em revistas científicas e em anais de congressos.

Principais Contribuições

Levando-se em conta a hipótese de investigação (*thesis statement*, do inglês) mencionada acima, pode dizer-se que a principal contribuição do trabalho de investigação que conduziu à presente tese é a seguinte:

- É possível detetar cavidades na superfície de uma dada proteína com base na topologia do seu campo de densidade eletrónica, que é um caso particular de um campo escalar, independentemente da posição e da orientação da proteína.

Noutras palavras, é possível identificar sem ambiguidade as cavidades de uma proteína.

Como se verá mais à frente, tais cavidades correspondem a determinados pontos críticos do campo escalar que descreve quer a superfície molecular, quer o campo de densidade eletrónica gerado pela proteína. Entre outras contribuições, contam-se também as seguintes:

- Uma variante do algoritmo dos cubos marchantes (*marching cubes algorithm*) em GPU que permite efetuar a triangulação e a visualização de uma superfície molecular. Veja-se Capítulo 3 para mais detalhes.
- Um algoritmo que permite detetar cavidades em superfícies moleculares através da subtração Booleana dos interiores de duas superfícies Gaussianas que representam a mesma molécula. Veja-se Capítulo 4 para mais detalhes.
- Um descritor intrínseco de forma (baseado na topologia dos campos escalares) que permite identificar cavidades moleculares sem ambiguidade. Veja-se algoritmo descrito no Capítulo 5 para mais detalhes.

Estes três algoritmos sustentam-se em métodos baseados em grelha, ou seja, na voxelização do domínio. No entanto, é nossa convicção que é possível aplicar o último algoritmo sem recorrer à voxelização do domínio; em especial, através do cálculo de caminhos de minimização e maximização no domínio (cf. [Gom14] para mais detalhes). Esta é uma questão em aberto para trabalho futuro.

Publicações

No âmbito da investigação que conduziu à escrita desta tese de doutoramento, produziu-se os seguintes artigos, a maioria dos quais já está publicada em revistas e anais de conferências:

- Sérgio Dias and Abel Gomes. A Scalar Field Topology-Based Method for the Detection of Protein Cavities. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (submitted for publication).
- Sérgio Dias and Abel Gomes. GPU-Based Detection of Protein Cavities using Gaussian-like Implicit Surfaces. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (under 2nd revision).
- Sérgio Dias and Abel Gomes. Triangulating Gaussian-like Surfaces of Molecules with Millions of Atoms. Chapter in Walter Rocchia and Michela Spagnuolo (eds.), *Computational Electrostatics for Biological Applications*, Elsevier, Chapter 9, Springer-Verlag, pp.177-198, 2015.

- Sérgio Dias and Abel Gomes. Triangulating Molecular Surfaces over a LAN of GPU-Enabled Computers. *Parallel Computing*, Vol.42, pp.35-47, 2015.
- Sérgio Dias and Abel Gomes. Triangulating molecular surfaces on multiple GPUs. In Proceedings of the of the International Workshop on Parallelism in Bioinformatics (PBio'13), held as part of 20th European Conference on Message Passing Interface (EuroMPI'13), Madrid, Spain, September 15- 18, ACM Press, 2013.
- Sérgio Dias and Abel Gomes. Graphics processing unit-based triangulations of Blinn molecular surfaces. *Concurrency & Computation: Practice & Experience*, Vol.23, no.17, pp.2280-2291, 2011.
- Sérgio Dias and Abel Gomes. CUDA-based Triangulation of Convolution Molecular Surfaces. In Proceedings of the 5th International ACM Symposium on High Performance Distributed Computing (HPDC'10), Workshop on Emerging Computational Methods for the Life Sciences (ECMLS'10), Chicago, USA, June 21-25, ACM Press, 2010.
- Sérgio Dias and Abel Gomes. GPU-based Triangulations of the van der Waals Surface. In Proceedings of the IEEE International Conference on Bioinformatics & Biomedicine (BIBM'10), Hong Kong, December 18-21, IEEE Press, 2010.

Organização da Tese

Esta tese de doutoramento visa a introdução de novas formas de entendimento, representação, e deteção de cavidades em superfícies moleculares de proteínas. Neste contexto, a tese está estruturada da seguinte forma:

- *Capítulo 1:* Este capítulo apresenta o trabalho de investigação subjacente à presente tese de doutoramento. Em particular, dá-se conta das razões que estiveram na origem deste trabalho, o qual aborda a representação e a deteção de cavidades em proteínas, e suas aplicações em acoplamento de proteínas (*protein docking*, do inglês).
- *Capítulo 2:* Neste capítulo empreende-se a revisão da literatura no que concerne aos métodos de deteção de cavidades moleculares, com particular ênfase nos seus princípios, bem como nas suas potencialidades e limitações.
- *Capítulo 3:* Neste capítulo descreve-se um algoritmo de triangulação de superfícies moleculares que tira partido da distribuição de carga de processamento por várias GPUs. Este algoritmo foi desenvolvido para a visualização gráfica de moléculas e suas cavidades.
- *Capítulo 4:* Este capítulo descreve um algoritmo de deteção de cavidades em proteínas através da análise dos voxéis localizados entre duas iso-superfícies da

mesma proteína. Como se verá, este algoritmo resolve os problemas de ambiguidade inerente à categoria de algoritmos baseados em grelha. Além disso, é um dos primeiros algoritmos geométricos de detecção de cavidades em proteínas que foi desenhado para GPU.

- *Capítulo 5:* Este capítulo propõe um novo descritor de forma para reconhecer cavidades através da topologia (isto é, pontos críticos) do campo escalar que caracteriza o campo de densidade de elétrons de uma dada proteína. A abordagem baseia-se na análise dos pontos críticos deste campo escalar na parte exterior da superfície molecular, e não em qualquer ponto sobre esta mesma superfície.
- *Capítulo 6:* Este capítulo apresenta as principais conclusões do trabalho de investigação descrito nesta tese, não sem que importantes questões sejam colocadas para trabalho futuro.

Como nota marginal, refira-se que o público-alvo deste trabalho é não só a comunidade de computação gráfica e de computação geométrica, mas principalmente a comunidade da biologia computacional e bioinformática, para quem os algoritmos descritos nesta tese poderão ser particularmente úteis.

Principais Conclusões e Trabalho Futuro

O tema central desta tese de doutoramento é o da representação geométrica de proteínas e das suas cavidades, bem como a detecção destas últimas. Como se verá no decurso da tese, a hipótese de investigação colocada acima será validada, ou seja, é possível associar os pontos críticos do campo de densidade eletrónica gerados pela molécula às suas cavidades. Esta é, em nossa opinião, a maior contribuição deste trabalho de doutoramento.

Para se chegar ao objetivo subjacente à hipótese de investigação, o trabalho foi dividido em quatro etapas principais: revisão da literatura no que concerne à detecção de cavidades em moléculas, estudo de modelos matemáticos de superfícies moleculares e seus campos de densidade eletrónica, desenvolvimento de descritores intrínsecos de forma para detetar cavidades moleculares, e ainda a paralelização de todos os algoritmos desenvolvidos. Cada um destes passos resultou nalguma contribuição para a tese.

Nos trabalhos de doutoramento, desenvolveram-se vários algoritmos, entre os quais se contam aqueles descritos desta tese, nomeadamente:

- Algoritmo de triangulação de superfícies moleculares em GPU (veja-se Capítulo 3).
- Algoritmo geométrico de detecção de cavidades com recurso a duas iso-superfícies geradas pelo campo de densidade eletrónica da molécula (veja-se Capítulo 4).

- Algoritmo geométrico de detecção de cavidades com recurso à topologia (pontos críticos) do campo de densidade eletrónica da molécula (veja-se Capítulo 5).

Na conceção e desenvolvimento dos algoritmos de detecção de cavidades procurou-se, acima de tudo, utilizar descritores intrínsecos de forma para que assim se garantisse a invariância relativamente a transformações afins, i.e., translações e rotações. Procurou-se deste modo garantir que a ambiguidade dos métodos geométricos do estado-da-arte fosse erradicada. Ficou também demonstrado que, a não ser que se use os recursos computacionais das atuais GPUs ou outros recursos de computação paralela, torna-se difícil garantir resultados em tempo real, quer na visualização gráfica 3D, quer na detecção de cavidades em moléculas.

Como trabalho futuro, é nossa visão que será recomendável refinar o terceiro algoritmo acima indicado no sentido de permitir determinar os pontos críticos do campo de densidade eletrónica da molécula sem utilizar a voxelização do domínio. Outra linha de investigação será a de investigar o comportamento da topologia do campo de densidade eletrónica da molécula quando esta interage com outras moléculas. Outro aspeto importante é o da consolidação dos algoritmos desenvolvidos através da sua divulgação pública na forma de bibliotecas de código aberto.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Cavity Detection Methods: an Overview	2
1.3	Research Hypothesis	3
1.4	Research Plan	3
1.5	Contributions	5
1.6	Publications	6
1.7	Organization of the Thesis	6
2	Cavity Detection Methods for Proteins: a Survey	9
2.1	Introduction	9
2.2	Triangulation-Based Methods	11
2.3	Grid-based algorithms	13
2.3.1	Distance-Based Grid Algorithms	13
2.3.2	Visibility-Based Grid Algorithms	13
2.3.3	Depth-Based Grid Algorithms	15
2.4	Sphere-Based Algorithms	16
2.5	Discussion	18
2.6	Concluding Remarks	19
3	Triangulating Gaussian-like Molecular Surfaces	21
3.1	Introduction	21
3.2	Background	23
3.2.1	Molecular Surfaces	23
3.2.2	Marching Cubes' Triangulations	24
3.3	GPU-based Triangulation: Overview	25
3.4	Reading Atoms in CPU Side Memory from a PDB/VDB File	26
3.5	Computation of the Bounding Box	26
3.6	Voxelization of the Bounding Box	26
3.7	Slicing of the bounding box	27

3.8	GPU Memory Allocation	27
3.9	Launching OpenMP Threads to Invoke GPU CUDA kernels	28
3.10	Triangulation on CUDA devices	29
3.10.1	Computation of function values (1-st kernel)	29
3.10.2	Computation of voxel flags (2-nd kernel)	30
3.10.3	Computation of the number of triangulation vertices associated to each voxel (3-rd kernel)	31
3.10.4	Computation of the total number of triangulation vertices (4-th kernel)	31
3.10.5	Computation of the triangle vertices (5-th kernel)	31
3.10.6	Computation of normal vectors to the surface (6-th kernel)	32
3.10.7	Merging Partial Triangulations and Rendering	32
3.11	Optimization of CUDA Kernels	32
3.12	Results and Performance Evaluation	35
3.13	Concluding Remarks	39
4	GPU-Based Detection of Protein Cavities using Gaussian-like Implicit Surfaces	43
4.1	Introduction	43
4.2	Related Work	45
4.3	Background	47
4.4	Cavity Detection Algorithm	49
4.5	GPU Implementation	49
4.5.1	Reading Atomic Centers from a PDB File	50
4.5.2	Computation of the Bounding Box	50
4.5.3	Voxelization of the Bounding Box	50
4.5.4	Computation of the Scalar Field F	51
4.5.5	Collecting Cavity Voxels	51
4.5.6	Formation of Cavities	51
4.6	Molecular Triangulation	52
4.7	Results	53
4.7.1	Hardware/Software Setup	53
4.7.2	Time Performance	53

4.7.3	Memory Space Complexity	56
4.7.4	CUDA Code Optimization	56
4.7.5	Comparison with other Geometric Algorithms	58
4.7.5.1	Ground-Truth Dataset of Cavities	58
4.7.5.2	Benchmarking Methods	59
4.7.5.3	Benchmarking Metrics	59
4.8	Concluding Remarks	63
5	A Curvature-Based Cavity Detection Method	65
5.1	Introduction	65
5.2	Related Work	66
5.3	Theoretical Background	68
5.3.1	Scalar Fields	68
5.3.2	Gaussian Molecular Surface	69
5.3.3	Critical Points	69
5.4	Cavity Detection Algorithm	71
5.4.1	Overview	71
5.4.2	Voxelization	71
5.4.3	Evaluation of the Discrete Scalar Field	72
5.4.4	Computation of Critical Points	72
5.4.5	Computation of the Number of Critical Points	74
5.4.6	Clustering Critical Points into Cavities	74
5.4.7	Surface Triangulation	74
5.4.8	Rendering of Surface and its Pockets	74
5.5	Results	76
5.5.1	Hardware/Software	76
5.5.2	CUDA Code Optimization	77
5.5.3	Ground-Truth Molecule Dataset	79
5.5.4	Time Performance	79
5.5.5	Memory Space Performance	81
5.5.6	Accuracy Testing	81
5.5.6.1	Benchmarking Methods and Metrics	81

5.5.6.2 Benchmarking Metrics	82
5.6 Concluding Remarks	86
6 Conclusions and Future Work	87
6.1 The Revisited Research Plan	87
6.2 Final Conclusions	88
6.3 Future Work	88
Bibliografia	89

List of Figures

2.1	(a) Voronoi diagram of a molecule (i.e., set of spherical atoms); (b) convex hull of the atom centers, together with Delaunay triangulation; (c) alpha shape with triangles, edges, and vertices in black, where the empty triangles denote the existence of a cavity (taken and modified from [LWE98] [WPS07]).	11
2.2	(a) van der Waals surface in black, and inner blending surface as a connected arrangement of blue and black spherical patches; (b) inner blending mesh constructed from the atom centers and blending surface; (c) outer blending surface as a connected arrangement of red and black spherical patches; (d) outer blending mesh as the convex hull of atom centers (taken from Kim et al.[KCC ⁺ 08]).	12
2.3	Detecting cavities through POCKET: (a) in the x -direction; (b) in the y -direction.	14
2.4	Detecting cavities through LIGSITE: (a) in the -45° -direction; (b) in the $+45^\circ$ -direction.	15
2.5	Detecting cavities using the center of gravity of the molecule and successive nearest surface atoms (NSAs): (a) the first cavity; (b) the remaining four cavities (picture taken from [LJ06]).	16
2.6	Detecting a cavity through SURFNET: (a) Each probe sphere is placed at the midpoint of a pair of atoms (A, B); (b) but, if such probe sphere overlaps at least an atom (dashed spheres), its radius has to be reduced until it just has a tangential contact with the overlapped atom; (c) all probe spheres placed into cavity after considering all pairs of atoms; (d) the cavity surface enclosing all probe spheres of the cavity (pictures taken from [Las95]).	17
2.7	Detecting cavities through PASS: (a) coating the molecular surface with a first layer of probe spheres; (b) a second layer of probe spheres is, in this case, enough to find cavities on molecular surface (pictures taken from [WPS07], and inspired by [BS00]).	18
2.8	Detecting cavities through PHECOM: (a) Small probes are placed on the Van der Waals surface; (b) large probes are placed on the Van der Waals surface; (c) small probes that overlap with the large ones are removed (taken from [KG07]).	19
3.1	The distribution of the computation overhead by 2 CPU cores and 2 GPUs.	30

3.2	Practical complexity of GPU-based programs in terms of the number of atoms: (a) time complexity; (b) memory space complexity.	39
3.3	Examples of molecular surfaces displayed using GPU-based marching cubes algorithm: (a) PDB id: 39ME; (b) PDB id: 1M1C; (c) PDB id: 1OHG.	41
3.4	Examples of molecular surfaces displayed using GPU-based marching cubes algorithm: (d) PDB id: 1HTO, (e) PDB id: 1X9P, (f) PDB id: 2G34.	42
4.1	Molecular surfaces with the same blobiness $\beta = 0.4$ and distinct isovalues: (a) $c = 1.0$; (b) $c = 1.2$; (c) $c = 1.4$; (d) $c = 1.6$; (e) the previous four molecular surfaces (a)-(d) overlapping.	47
4.2	Molecular surfaces with the same isovalue $c = 1.0$ and distinct blobiness values: (a) $\beta = 0.34$; (b) $\beta = 0.32$; (c) $\beta = 0.30$; (d) $\beta = 0.28$; (e) the previous four molecular surfaces (a)-(d) overlapping.	47
4.3	Molecular visualization of the 110D molecule: (a) cavities are not depicted on the surface; (b) cavities are depicted on the surface as determined by our algorithm. The small spheres in blue indicate the location of cavities as calculated by MetaPocket [Hua09].	52
4.4	(a) Time performance of CPU algorithm with (in red) and without (in blue) surface triangulation; (b) Time performance of GPU algorithm with (in red) and without (in blue) surface triangulation.	53
4.5	(a) Memory space occupancy with respect to the number of atoms (n); (b) the number of voxels in memory with respect to the number of atoms (n).	54
4.6	Histograms of hit performance of GaussianFinder in relation to LIGSITE, PASS, SURFNET, POCASA, and fpocket: (left) numerical hit performance n_i in function of the dispersion of the number of cavities Δ_i ; (right) positional hit performance h_i in function of the dispersion of the number of cavities Δ_i	57
4.7	Cumulative cavity percentage of various detection methods in function of the distance d to ground-truth geometric centers: GaussianFinder, LIGSITE, PASS, SURFNET, POCASA, and fpocket for apo (left) and holo (right) structures.	60
4.8	Examples of 3 proteins and their cavities as detected by GaussianFinder (left) and LIGSITE (right): (a) PDB id: 1NEQ (13 cavities); (b) PDB id: 1SVR (10 cavities); (c) PDB id: 2NWD (15 cavities).	64
5.1	Critical points as red balls considering: (a) the voxels outside or intersecting the surface; (b) only the voxels intersecting the surface.	73

5.2	CriticalFinder (left) and LIGSITE (right) output the same number of cavities on the same locations for two proteins: (a) PDB id: 2gqv (20 cavities); (b) PDB id: 2nwd (18 cavities).	75
5.3	Time performance of the algorithm with (in red) and without (in blue) the triangulation and rendering steps: (a) CPU implementation; (b) GPU implementation.	78
5.4	(a) Memory space occupancy with respect to the number of atoms (n); (b) the number of voxels in memory with respect to the number of atoms (n).	78
5.5	Histograms of hit performance of CriticalFinder in relation to LIGSITE, PASS, SURFNET, POCASA, and fpocket: (left) numerical hit performance n_i in function of the dispersion of the number of cavities Δ_i ; (right) positional hit performance h_i in function of the dispersion of the number of cavities Δ_i	82
5.6	Cumulative cavity percentage of various detection methods in function of the distance d to ground-truth geometric centers: CriticalFinder, LIGSITE, PASS, SURFNET, POCASA, and fpocket for apo (left) and holo (right) structures.	85

List of Tables

3.1	Kepler GPU's memory and compute capabilities.	33
3.2	Performance data <i>before</i> optimizing the CUDA kernels.	34
3.3	Performance data <i>after</i> optimizing the CUDA kernels.	34
3.4	Time performance/memory occupancy for a number of molecular surfaces <i>before</i> optimization.	36
3.5	Time performance/memory occupancy for a number of molecular surfaces <i>after</i> optimization.	37
4.1	Kepler GPU's memory and compute capabilities.	55
4.2	Performance data <i>before</i> optimizing the CUDA kernels.	55
4.3	Performance data <i>after</i> optimizing the CUDA kernels.	55
4.4	Hit performance (n_i and h_i) of GaussianFinder with respect to LIGSITE, PASS, SURFNET, POCASA, and fpocket.	57
5.1	Kepler GPU's memory and compute capabilities.	76
5.2	Performance data <i>before</i> optimizing the five CUDA kernels (i.e., five steps of CriticalFinder).	76
5.3	Performance data <i>after</i> optimizing the CUDA kernels.	76
5.4	Hit performance (n_i and h_i) of CriticalFinder with respect to LIGSITE, PASS, SURFNET, POCASA, and fpocket.	82

List of Abbreviations

3D	Three dimensional
Å	Angstrom
ALU	Arithmetic logic units
API	Application programming Interface
APROPOS	Automatic Protein Pocket Search
ASPs	Active site points
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUDDP	CUDA Data Parallel Primitives Library
GB	Gigabytes
GPU	Graphical processor unit
HPC	High-performance computing
MB	Megabytes
MC	Marching Cubes
NBO	Normal Buffer Object
NSA	Nearest surface atom
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PDB	Protein Data Bank
PhD	Philosophiae doctor
SAS	Solvent-accessible surface
sec	Seconds
SES	Solvent-excluded surface
SM	Streaming Multiprocessors
STL	Standard Template Library
VBO	Vertex buffer object
VDB	Virus Data Bank
VDW	Van der Waals
VIPERdb	Virus Particle Explorer database

Chapter 1

Introduction

This chapter briefly describes the rationale behind the research carried out during the PhD programme, which fits in the scope of geometric computing and molecular graphics.

1.1 Motivation

Most organisms in the world are made up of cells, which in turn are composed by molecules that are fundamental in biochemical processes that sustain life. Essentially, molecules interact with other molecules, so these interactions are the basis for almost cellular processes (e.g., DNA transcription).

Interactions between molecules occur on specific molecular regions, also called binding sites. But, not all binding sites of a molecule are compatible with the binding sites of another molecule. For example, the challenge in molecular docking is to know the compatible sites of two coupling molecules in advance. Note that the compatibility between two molecules must occur at both geometric and chemical levels.

That requires a deep understanding of how a molecule binds to a specific surface region of another molecule, and how such information can be used to predict which molecules are supposed to bind to that region. Because of the structural variety and complexity of molecules, such an understanding seems to be a rather difficult task to accomplish.

Aware of these difficulties, the scientific community has used to computational resources to make this problem more tractable and timely. However, most of the underlying algorithms are not able to discriminate between correct results and false positives obtained in the simulation. In a way, this is so because the known shape descriptors are not able to output the (local and global) shape of a molecule uniquely, i.e., without ambiguities, not to say timely.

As known, shape plays a key role in bio-molecular recognition and function, in particular in the non-covalent interactions between molecules, as needed in applications such as drug design and protein engineering. Binding site recognition is the computational approach that allow us to locate or predict the location where cavities of a molecule are. The identification of protein binding sites has significant impact on understanding protein function. With the increasing number of protein datasets available across the web, predictive methods using geometric information only for protein interaction have drawn increasing interest. Currently, there are over 60 thousand protein 3D structures known, and this number is even increasing. One can use these data to predict the

binding sites in a protein structure of interest. Thus, the identification of those binding sites is often the first step to study protein functions and structure-based drug design.

The focus of this work is to predict the molecular binding sites, that is, to find regions on the protein surface that can bind to other molecules (i.e., ligands), exclusively using geometric algorithms. Such protein surface regions are here called *cavities*, which include pockets, voids, tunnels, and so forth. But, as mentioned above, not all protein cavities are ligandable. That is, cavities are *tentative* binding sites of proteins.

1.2 Cavity Detection Methods: an Overview

In general, the computational algorithms to find cavities on a molecule divide in four categories: geometry-based, energy-based, evolutionary-based, combined approaches [VGGR10]. However, in this thesis, we are only interested in geometry-based algorithms. These geometric-based algorithms divide in three categories [KG07]:

- Grid-based
- Sphere-based
- Triangulation-based

Grid-based algorithms consist in mapping a protein onto an axis-aligned 3D grid, after which we apply a scanning method to label each grid point as occupied or not by the protein, using then some geometric filtering method that outputs the non-occupied grid points that are deemed to be cavities or pockets. For example, Levitt and Banaszak determine which grid points lie in a pocket or a cavity by checking whether each of them is bracketed by two occupied points in either the x, y, or z directions [LB92]. Two significant drawbacks of grid-based methods is that they are not invariant to position and orientation of the molecule in 3D, i.e., the number of cavities and their locations may be distinct. The ambiguity of these methods becomes even more striking if one takes into account that it is not trivial to distinguish among all nodes outside the molecule those belonging to cavities. Besides, these methods are memory space-consuming because they have to hold the grid of nodes in memory.

The leading idea of *sphere-based algorithms* is to detect cavities, surface grooves and surface pockets by fitting probe spheres into them in some manner, defining then a surface around the resulting cluster of probe spheres. That is, these algorithms make usage of flood-filling procedure of probe spheres into cavities, grooves and pockets [Las95]. On average, sphere-based methods perform slower than grid-based methods in detecting of pockets, although they consume much less memory because it is not necessary to proceed to the voxelization of the 3D space surrounding the protein.

Triangulation-based algorithms usually focus on alpha shapes. An alpha shape is essentially a surface triangulation that is dual to the molecular surface in the sense that it

possesses the same number of pockets, concavities, and tunnels [EM94]. More specifically, an alpha shape can be defined as a subset of Delaunay tessellations of atom centers, where the removal of edges longer than the sum of the radii of two atoms allows to detect pockets. Examples of algorithms that fall in this category are CAST [BNL03] and APROPOS (Automatic Protein Pocket Search) [PFF96]. Alpha shape methods for prediction of protein binding sites are also time-consuming, at least for large molecules.

In general, the aforementioned geometric algorithms for the search of binding sites are very demanding in terms of mathematical computations, in particular when the number of atoms goes beyond some thousands. To keep the computational runtimes within reasonable limits, most algorithms only deal with small molecules. These circumstances get worse when we use 3D voxelizations of the space surrounding a molecule, as it the case of grid-based methods, since the computational complexity tends to be cubic, unless we take advantage of the parallel computing facilities (e.g., CUDA and OpenCL) of modern graphics cards.

1.3 Research Hypothesis

This thesis addresses a distinct approach to segment protein surfaces into clefts and knobs (i.e., concavities and convexities). We can even say that this approach leads to a new category of predicting methods for molecular cavities as tentative binding sites. Essentially, it is based on the theory of scalar fields, in particular scalar field topology. So the *thesis statement* behind the research described in this thesis can be stated as follows:

It is feasible to detect molecular cavities (as tentative binding sites) using shape descriptors based on the theory of scalar fields.

Particularly, we intend to investigate the relationships between critical points of the scalar field outside the molecular surface of the protein and its cavities. The computation of the critical points allows us to identify the exact location of pockets or clefts on the protein surface. Moreover, we believe that this can be done without using any 3D spatial grid or sphere-based coating of the molecular surface, or even any triangulation, but this has not been tried in the course of this thesis.

1.4 Research Plan

In the course of our research to prove the thesis statement mentioned above, we had to achieve a number of milestones, as described below:

- *Molecular Surfaces and Electron Density Fields of Molecules.* This step aimed at firstly finding a suited mathematical formulation for molecular surfaces. But,

we had also in mind an adequate mathematical formulation to describe the electron density field generated by the molecule. As described in this thesis, we use Gaussian-like scalar fields to model both the electron density field and the surface of each molecule. This two-in-one solution is advantageous in many respects, in particular in respect to finding the critical points of electron density field of each molecule. Additionally, molecular interactions in physiochemical processes mean that molecules are not static and, consequently, their surfaces are not static either. Molecular surfaces changes over time depending on the interactions a molecule establishes with others locally. In atomic terms, this means that atoms change their positions within a molecule, which adopts different conformations depending on various factors such as temperature, hydrophobic effects, etc. Unlike those models found in the literature, our mathematical model keeps the spherical atoms and their surface envelope together. In this way we are able to directly associate each surface point to a single atom. Such a surface envelope of a molecule has been defined as a Gaussian-like surface, which can be easily triangulated and rendered using the marching cubes algorithm [LC87].

- *Molecular Triangulation Algorithm.* In order to visualize the results of our cavity detection algorithms, we primarily needed to develop a triangulation algorithm for Gaussian-like molecular surfaces, which was inspired in Blinn's formulation [Bli82]. The leading idea was to tackle the problem of triangulating and rendering surfaces of molecules with at least 0.5 million atoms in real-time. This real-time requirement led us to develop triangulation algorithms on GPU, in particular using CUDA.
- *Intrinsic Shape Descriptors.* An intrinsic shape descriptor is one that is invariant to geometric transformations, typically rotations and translations. These shape descriptors should take into account the local shape of the surface surrounding a given point, and also be robust to noise and sampling errors. Shape descriptors should also have a meaningful comparison function, one that scales roughly linearly with perceived shape change and is robust to noise. Example of shape descriptors with such characteristic are spherical harmonics algorithms, Laplace-Beltrami descriptors or surface curvature. However, in general, shape descriptors operate on the molecular surface only, not taking into account the surrounding space of the molecular surface. With such domain-extended analysis it is possible to know the surface regions where the interactions are possible with other molecules. Thus, the analysis of protein cavities is of crucial importance to understand the biological processes in which proteins are involved in. Herein, we focus on the computational analysis of protein cavities, which is based on scalar field topology. This mathematical theory allows us to detect and recognize the location of different protein cavities, without spending much time and computational resources.
- *Protein Cavity Detection Algorithms.* According to the previous milestones, we

developed protein cavity detection algorithms based on intrinsic shape descriptors. The first algorithm uses two implicit surfaces of a level set generated by a Gaussian-like scalar field. This technique solves the ambiguity problem of grid-based methods used in protein cavity detection. The second algorithm uses the same Gaussian-like scalar field, but the cavities are detected using scalar field topology, i.e., the critical points of the such Gaussian-like scalar field. These algorithms were also implemented on GPU via CUDA.

- *Thesis Writing.* The thesis was being written in agreement with the course of the PhD programme, and also with the pace of publishing scientific papers. So, the core chapters of this thesis have given place to papers published or under revision in journals and conference proceedings.

1.5 Contributions

Taking into account the thesis statement mentioned above, it can be said that the main contribution of the research work behind this thesis is the following:

- It is possible to detect cavities on the protein surface using scalar field topology, independently of the position and orientation of the protein. In other words, it is possible to identify the protein cavities without ambiguities.

As seen further ahead, such cavities correspond to particular critical points of the scalar field that describes both the molecular surface and the electron density field generated by the protein. As secondary contributions, we would list the following:

- A fully multi-GPU-based multi-threaded variant of the marching cubes algorithm to triangulate and render molecular surfaces, with all the particularities of computing the molecular surface locally in the neighborhood of each atom.
- An algorithm for detecting cavities on molecular surfaces through the Boolean difference of the interiors of two Gaussian molecular surfaces that approximate the same molecule.
- An algorithm based on an intrinsic shape descriptor taken from the scalar field topology to identify the molecular cavities without ambiguities.

These algorithms sustain themselves on grid-based methods, i.e., on the voxelization of the domain. Nevertheless, we believe that it is possible to implement the latter algorithm without voxelizing the domain; in particular using path minimization and maximization in the domain (cf. [Gom14] for more details). This is an open issue for future work.

1.6 Publications

Within the scope of the research behind this thesis, we have produced the following articles, most of which already are published in journals and conference proceedings:

- Sérgio Dias and Abel Gomes. A Scalar Field Topology-Based Method for the Detection of Protein Cavities. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (submitted for publication).
- Sérgio Dias and Abel Gomes. GPU-Based Detection of Protein Cavities using Gaussian-like Implicit Surfaces. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (under 2nd revision).
- Sérgio Dias and Abel Gomes. Triangulating Gaussian-like Surfaces of Molecules with Millions of Atoms. Chapter in Walter Rocchia and Michela Spagnuolo (eds.), *Computational Electrostatics for Biological Applications*, Elsevier, Chapter 9, Springer-Verlag, pp.177-198, 2015.
- Sérgio Dias and Abel Gomes. Triangulating Molecular Surfaces over a LAN of GPU-Enabled Computers. *Parallel Computing*, Vol.42, pp.35-47, 2015.
- Sérgio Dias and Abel Gomes. Triangulating molecular surfaces on multiple GPUs. In Proceedings of the of the International Workshop on Parallelism in Bioinformatics (PBio'13), held as part of 20th European Conference on Message Passing Interface (EuroMPI'13), Madrid, Spain, September 15- 18, ACM Press, 2013.
- Sérgio Dias and Abel Gomes. Graphics processing unit-based triangulations of Blinn molecular surfaces. *Concurrency & Computation: Practice & Experience*, Vol.23, no.17, pp.2280-2291, 2011.
- Sérgio Dias and Abel Gomes. CUDA-based Triangulation of Convolution Molecular Surfaces. In Proceedings of the 5th International ACM Symposium on High Performance Distributed Computing (HPDC'10), Workshop on Emerging Computational Methods for the Life Sciences (ECMLS'10), Chicago, USA, June 21-25, ACM Press, 2010.
- Sérgio Dias and Abel Gomes. GPU-based Triangulations of the van der Waals Surface. In Proceedings of the IEEE International Conference on Bioinformatics & Biomedicine (BIBM'10), Hong Kong, December 18-21, IEEE Press, 2010.

1.7 Organization of the Thesis

This doctoral thesis aims at introducing new ways of understanding, representing, and detecting cavities on protein surfaces. In this context, the thesis is structured as follows:

- Chapter 1: This chapter introduces the research work underlying the present PhD thesis. Here the rationale behind the research done during the PhD programme is presented, specially in the area of geometric representation and recognition methods of protein cavities, as needed, for example, in protein docking.
- Chapter 2: This chapter carries out the review of the literature concerning cavity detection methods. Therein, it is presented the general ideas behind these methods, with a special emphasis on their leading ideas, advantages, and limitations.
- Chapter 3: In this chapter a multi GPU-based triangulation algorithm for molecular surfaces is described in detail. It is a fast, scalable, parallel triangulation algorithm for molecular surfaces that takes advantage of multicore processors of CPUs and GPUs of modern hardware architectures, where each CPU core works as the master of a single GPU, being the processing burden distributed over the CPU cores available in a single computer or a cluster. The algorithm is based on a parallel version of the marching cubes algorithm that triangulates molecular surfaces on multiple GPUs using CUDA and OpenMP. Besides it is carried out a study that compares a sequential version (CPU) to a parallel version (GPU) of well-know marching cubes (MC) algorithm to render molecular surfaces.
- Chapter 4: This chapter presents an algorithm that detects protein cavities through the analysis of the voxels located between two molecular surfaces. The novelty of the method lies in the fact that it makes usage of the scalar field of the protein in 3D space, from which we are able to define two distinct surfaces for the protein, being that the protein cavities are in the middle. Besides, as far as we know, this is the first geometric detection algorithm for protein cavity detection on GPU.
- Chapter 5: This chapter proposes a new molecular shape descriptor to recognize protein cavities through the topology (i.e., critical points) of the scalar field that features the electron density field of the protein. The approach is based on the study of the (normalized) Hessian matrix at points outside the molecular surfaces, not at points on the surface of the protein. Thus, the main novelty of the method relies in the use of the Morse theory as applicable to scalar fields. Furthermore, we compare our algorithm with others, in order to better assess the quality of the found cavities.
- Chapter 6: This chapter concludes this thesis, with important clues for future developments of this work.

As a marginal note, let us say that the target audience of this thesis is not only the community of computer graphics and geometric computing, but primarily the community of computational biology and bioinformatics, for whom we have designed the algorithms described herein.

Chapter 2

Cavity Detection Methods for Proteins: a Survey

The detection of protein cavities provides useful information about biological processes (e.g., protein docking). However to know the location of such regions it is necessary to collect information about the position, and the type of atoms that comprise a given molecule. Usually, this is done by X-ray crystallography, which is a technique based on the diffraction patterns of a X-ray irradiation in a crystal molecule. The result is then recorded in an appropriate plate that provides information about the position of each atom. Fortunately, for these molecules, their constituent atoms, and 3D atomic coordinates (x,y,z values) can be retrieved from a PDB file. With the three-dimensional structure of a protein, it is possible to find places where other molecules may bind. Such interactions generally happens in specific regions of the protein, called cavities, which usually correspond to pockets, clefts, inner cavities, or grooves on the surface of a given protein. In the literature, there are essentially three categories of computational algorithms to detect cavities on the protein surfaces: evolution-based, energy-based, and geometry-based. However, this chapter only surveys geometric algorithms.

2.1 Introduction

In 1894, Fischer conducted the initial studies on detection of protein cavities. From these studies, he concluded that the binding of a molecule to another is similar to the paradigm of inserting a key into a lock. In other words, this means that the affinity between two molecules exists if the shape of a molecule matches the shape of the other. However such model was considered very simplistic, because shape cannot be the only factor that influences the detection of protein cavities, since proteins are highly flexible and change shape over time. Generally, protein binding sites are specific large and deep clefts [LLST96]. However, protein shape can vary considerably, depending on the protein we have in hand. For example, the protein binding site of a ribonuclease is an extended rut, while the protein binding site of an endonuclease is a spherical cavity, and for an enzyme it is usually the largest cavity [LWE98].

In fact, a protein can bind many types of molecules, largely because of its non-negligible number of cavities. Many properties can be inferred from these molecular regions, helping us to understand how molecular interactions take place, as well as to be aware of important information on protein structures in the design of compounds, as usual in pharmaceutical and biotechnological domains. But, some of those protein zone targets are more receptive to bind with certain ligands than others. Those regions generally

have larger surface areas, and correspond to the concave, cleft or hole-shaped regions on a protein surface [KG07].

With molecular complexes involved in various molecular interactions, it is necessary to have adequate tools to characterize protein cavities as, for example, shape, size, and depth. But, a protein cavity only becomes a binding site if a number of factors like ambient temperature, pH, shape complementarity, electrostatics, hydrogen-bonding, and solvent interactions, combine gracefully. It is the conjugation of all of these factors that enable a ligand to identify the correct place to bind a given protein and induce its biological effect [HK06].

To make laboratory experiences easier it would be helpful to have computational methods capable of simulating such bio-chemical processes. However, computational methods capable of simulating such processes are extremely difficult to recreate. The difficulties behind that are related with the variety of admissible ligands, the variety of protein cavities, the protein shape variations themselves, and the physico-chemical factors that act on a cavity region.

It happens that such regions are usually located in pockets, clefts, inner cavities, and grooves on protein surfaces. Therefore, a better understanding of the process entangled in binding proteins requires the detection of cavities on the molecular surfaces. A computational estimate of the location of such protein regions, before initiating any experimental work in the drug discovery process, may be instrumental in the improvement of the design of new drugs. For that purpose, a number of algorithms for predicting and identifying protein cavities have been developed so far. Such algorithms can be divided into three major categories:

- *Evolutionary algorithms*: They rely on multiple sequence alignments to find the location of cavities on a given protein surface.
- *Energy-based algorithms*: In this case, cavities are detected through the calculation of the interaction energies between protein atoms and a small-molecule probe (e.g., Grid [Goo85], QSiteFinder [LJ05], and AutoLigand [HOG08]).
- *Geometric algorithms*: These algorithms are based on the analysis of geometric properties of a molecular surface to detect cavities (e.g., SURFNET [Las95], LIGSITE [HRB97], and PocketDepth [KC08]).

However, each method has its own drawbacks. For example, geometric methods relying on a grid are sensitive to protein position and grid spacing. Energy-based methods depend on their filtering procedures, force field parameterizations, and scoring functions. In turn, evolutionary-based methods depend on the quality of the alignment tool, and also on the number of available sequences. These problems show us that there is still a long way to go in this field, so that there is a need for further analysis of all the processes involved in the detection of binding sites of proteins [KG07]. This explains why the detection of molecular cavities still is a very active research area [HSAH⁺09].

2.2 Triangulation-Based Methods

The foundations of the triangulation-based methods lie in the field of computational geometry, in largely after the introduction of alpha shapes by Edelsbrunner and Mücke [EM94], in 1994. Edelsbrunner is a well-known computer scientist in the field of computational geometry. Interestingly, Edelsbrunner himself and colleagues [EFFL95] published a work on measuring proteins and voids in proteins in 1995.

In 1996, Peters et al. [PFF96] introduced new methods to detect molecular cavities through Voronoi diagrams, as in APROPOS program. The main objective was the development of an algorithm to identify cavities in proteins using only geometric criteria. In 1998, this technique was further developed by Liang et al. [LWE98], as illustrated in Fig. 2.1, which was called CAST. For that purpose, Liang and colleagues' algorithm firstly creates a Voronoi space decomposition from the atoms (atomic coordinates) of the molecule (Fig. 2.1(a)), from which one calculates the corresponding convex hull (i.e., Delaunay triangulation) (Fig. 2.1(b)), removing then the simplexes (e.g., triangles) that are not completely inside the molecule, resulting so in an alpha shape of the original molecule (Fig. 2.1(c)). The leading idea here is to get a triangulation with the same topological type as the original set of atoms that comprise the molecule, so that we can extract the cavities in a straightforward manner.

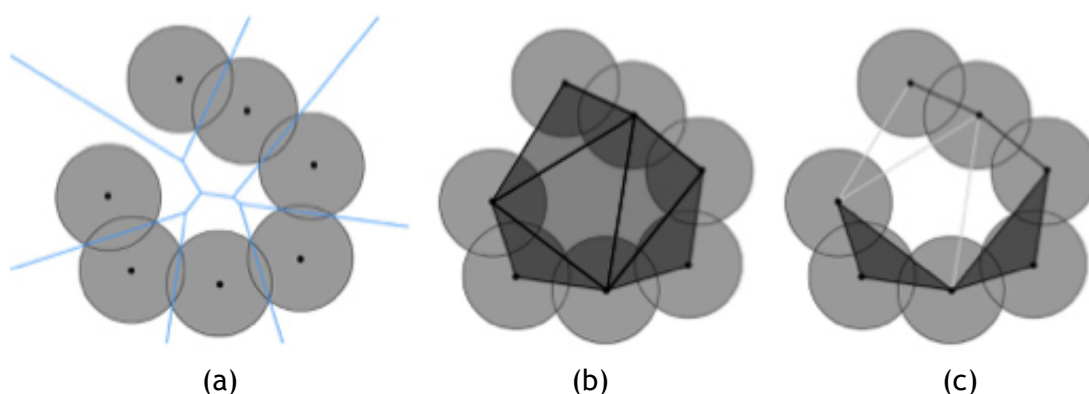


Figure 2.1: (a) Voronoi diagram of a molecule (i.e., set of spherical atoms); (b) convex hull of the atom centers, together with Delaunay triangulation; (c) alpha shape with triangles, edges, and vertices in black, where the empty triangles denote the existence of a cavity (taken and modified from [LWE98] [WPS07]).

Another algorithm based on the same steps as the CAST algorithm was developed by Yaffe et al. [YFW⁺08], and called MolAxis. The only difference is in the type of surface that is used to detect cavities; CAST uses the molecular surface, while MolAxis uses the van der Waals surface. Also, SplitPocket developed by Tseng et al. [TDCL09], in 2009, has the same steps of CAST [LWE98], but SplitPocket has the capability of splitting a cavity if necessary through the analysis of atomic structure. In 2013, Sridharamurthy et al. [SDP⁺13] introduced a similar algorithm to CAST, with the difference that one uses the concept of skin surface rather than the one of molecular surface.

In 2007, Xie et al. [XB07] proposed an algorithm similar to CAST (cf. Fig. 2.1), but they introduced a new shape descriptor, the geometric potential, in order to distinguish ligand and binding sites from non-ligand binding sites. Following the same line of research, Kim et al. [KCC⁺08] built up a blending mesh of triangles derived from a surface generated from blending atoms, as illustrated in Fig. 2.2. Then, they construct the convex hull from such a blending mesh. Cavities are found in places of the convex hull that are not occupied by the blending mesh. Interestingly, Kim et al. [KCC⁺08] use the Voronoi diagram of atoms, not the Voronoi diagram of atom centers, to easily calculate the molecular surface.

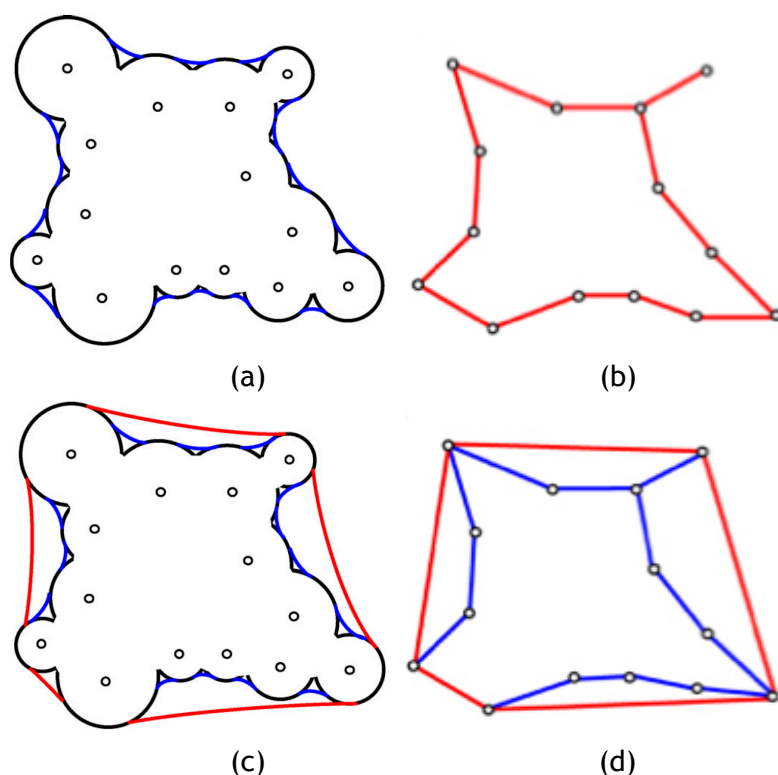


Figure 2.2: (a) van der Waals surface in black, and inner blending surface as a connected arrangement of blue and black spherical patches; (b) inner blending mesh constructed from the atom centers and blending surface; (c) outer blending surface as a connected arrangement of red and black spherical patches; (d) outer blending mesh as the convex hull of atom centers (taken from Kim et al. [KCC⁺08]).

Guilloux et al. [LGST09] introduced Fpocket, which builds upon on the concept of alpha spheres due to Liang et al. [LWE98]. For that purpose, one has to compute the Voronoi tessellation of atom centers, what is performed using the publicly available Qhull's source code on <http://www.qhull.org>, a well-known package that primarily calculates the convex hull of a set of points through the Quickhull algorithm. Recall that an alpha sphere is an external sphere that stands in contact with four boundary atoms of the molecule simultaneously, featuring thus the local curvature of molecular surface. Thus, cavities are located where we find alpha spheres; this obviously requires the use of some clustering of alpha spheres to form such cavities. This means that locating alpha spheres is equivalent to detect cavities on protein surfaces.

2.3 Grid-based algorithms

Grid-based algorithms essentially map any molecule onto an axis-aligned 3D grid, using then a specific geometric parameter to detect eventual cavities on the molecular surface. The most common geometric parameters are the following: distance, visibility, and depth.

2.3.1 Distance-Based Grid Algorithms

In respect to *distance-based grid algorithms*, let us mention FRODO, which is due to Voorintholt et al. [VKV⁺89], and is considered by many as the first cavity detection algorithm. This algorithm assigns a real value to every single grid point, which depends on whether such point is inside the molecule, between the van der Waals (vdW) surface and the solvent accessible surface (SAS), or beyond SAS. Such real value assigned to each grid point is produced by a real function that depends on the distance of such grid point to the nearest surface atom, so that we end up having a distance map associated to the grid. It is clear that cavities are located between the vdW surface and SAS, but truly speaking FRODO does not actually detect cavities [GT94], having it been designed only for the visualisation of SAS. In fact, as noted by Ho and Marshall [HM90], although FRODO is effective in finding regions where cavities are located, it is not that easy to isolate and define the extent of each specific cavity. To overcome this lack of specificity, Ho and Marshall [HM90] introduced the CAVITYSEARCH, which implements a search function to isolate and delineate each cavity of interest starting from a seed point, through a filling procedure of such cavity, thereby producing a cast of the cavity.

Another well-known distance-based grid algorithm is due to Zhang and Bajaj [ZB07]. More specifically, they use a signed distance function from the molecular surface to determine its pockets. The extraction of pockets can be performed for any closed compact molecular surface (e.g., van der Waals surface, Gaussian isosurface, and SAS) embedded in a regular grid, being the signed distance function to the surface evaluated for grid points far away and outside from the surface, as well as for grid points of cubic cells intersecting the surface. The cavities (e.g., pockets, tunnels, and voids) correspond to grid points outside the molecular surface where the signed distance function is positive.

2.3.2 Visibility-Based Grid Algorithms

In respect to *visibility-based grid algorithms*, also known as *scan-based grid algorithms*, they are all built upon the idea of line of sight or visibility; a line of sight is a scanning direction. This idea of scanning cavities along one or more directions was first proposed by Levitt and Banaszak [LB92]; their algorithm is known as POCKET. As a grid-based algorithm, this algorithm firstly maps the molecule onto an axis-aligned grid of equally-spaced points. The detection of cavities is carried out by scanning them along x , y , and

z axes. The x -axis scan is repeatedly done for all y and z values, starting on those grid points belonging to the leftmost plane of the 3D grid where x is minimum, i.e., $x = x_{min}$ (Fig. 2.3); analogous procedure applies to y -axis scans and z -axis scans. As usual, we produce a density map of the grid. Initially, all grid points are set with a density value of 0, but if a grid point is bracketed in an axis-aligned sequence of grid points bounded by the molecular surface itself, then it is set with the density value of 1. The cavities of the molecule are located in regions outside the molecule where the density equals 1. The bounds of each axis-aligned sequence of density-1 grid points are determined by checking if the centre of one of the atoms falls inside an imaginary sphere that moves in discrete steps along the scanning direction.

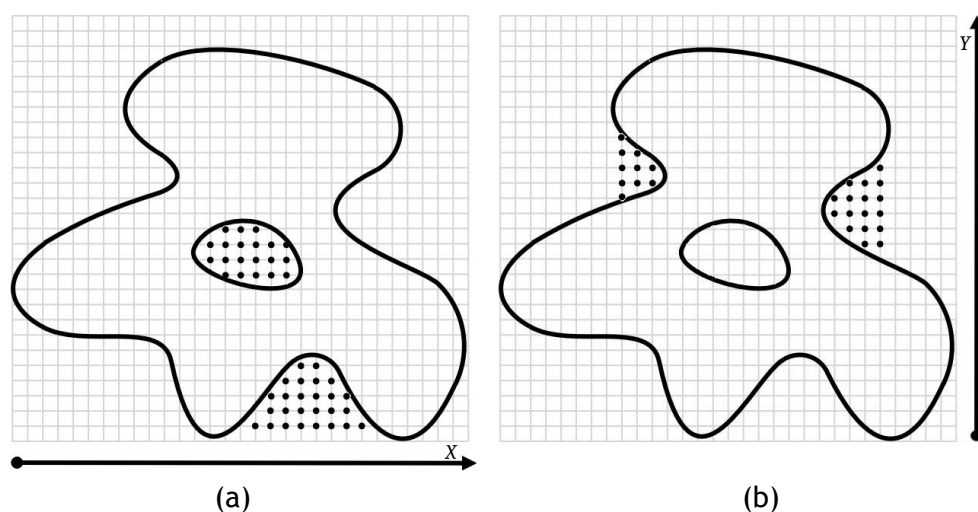


Figure 2.3: Detecting cavities through POCKET: (a) in the x -direction; (b) in the y -direction.

The main shortcoming of POCKET is that identifying pockets depends on the alignment of the protein in relation to the coordinate system of the grid [LJ06]. In order to mitigate this problem, Hendlich et al. [HRB97] developed a more sophisticated scanning method, which was implemented in LIGSITE. In addition to the three scans along x , y , and z , Hendlich and colleagues used more four scans with probes along the Cartesian cubic diagonals [LJ06], in a total of seven directions, in the attempt of making the identification of cavities less dependent on the orientation of the protein embedded in the 3D grid, as illustrated in Fig. 2.4. This seven of directions correspond to 14 oriented directions; for example, x direction corresponds to two oriented directions, x and $-x$. In practice, if we think in terms of grid cubes neighbouring a given grid cube, these 14 oriented directions are those defined by 14 out of 26 grid cubes surrounding a given cube. Li et al. [LTA⁺08] extended the number of scanning directions to those 26 oriented directions in VISGRID.

An algorithm similar to the POCKET and LIGSITE was developed by Weisel et al. [WPS07], and is called PocketPicker. The difference between PocketPicker and its predecessors is that scans are performed with a spherical probe along 30 directions equally distributed on a sphere [EA97]. Similarly, Tripathi and Kellogg developed VICE [TK10], which uses

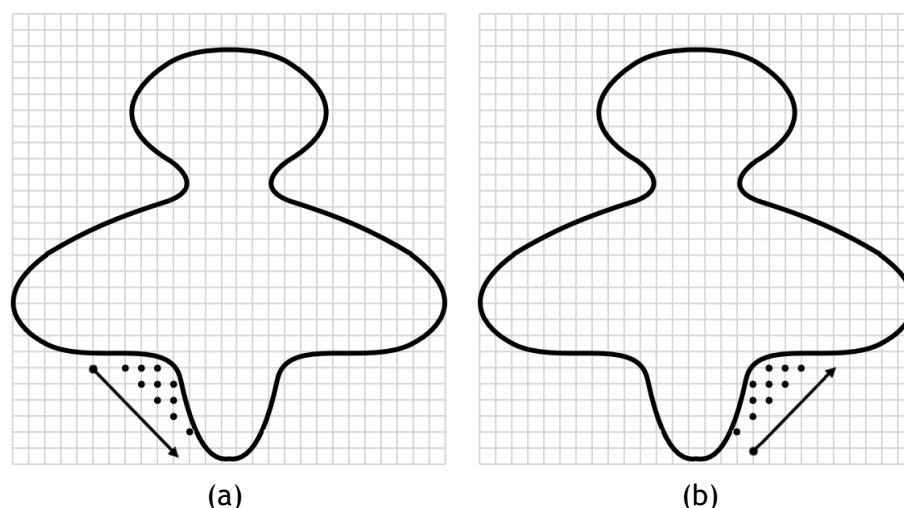


Figure 2.4: Detecting cavities through LIGSITE: (a) in the -45° -direction; (b) in the $+45^\circ$ -direction.

the concept of integer grid map instead of floating point grid map to define the scan directions through integer arithmetic vectors as a way of speeding up the computations of density map associated to the grid.

2.3.3 Depth-Based Grid Algorithms

In computational biology and chemistry, the depth is a measure of the buriedness of a protein atom, so that it is often defined as the distance between the atom center and the nearest water molecule on the protein surface [TBK⁺91]. The concept of depth has been also used in the cavity detection of proteins and other biomolecules, but not necessarily in the context of grid-based algorithms. For example, as far as we know, depth was firstly used by Del Carpio et al. [DTS93] to identify superficial cavities from the center of gravity of a given protein. After finding the nearest surface atom (NSA) from the center of gravity, a cavity is formed by clustering of the surface atoms that visible to the NSA. The process is repeated while there some concavity to detect on the molecular surface, as illustrated in Fig. 2.5.

In 2006, Coleman and Sharp [CS06] introduced a new concept of depth applied to grid cubes to where cavities are mapped; their cavity detection algorithm was termed Travel Depth. The cavities are located between the triangulated molecular surface and its convex hull, which is determined using any 3D convex hull algorithm (e.g., Quickhull). Then, each intermediate grid cube between the molecular surface and the convex hull is assigned a depth value that corresponds to the minimum path length needed to travel towards convex hull, in a way similar to what one does to calculate the shortest path in pathfinding (e.g., Dijkstra algorithm). Interestingly, such concept of depth allows them to organise cavities into sub-cavities in a hierarchical manner [CS10].

PocketDepth is another grid-based algorithm that takes advantage of the concept of depth and proximity of grid cubes to identify and gather them into cavities [KC08].

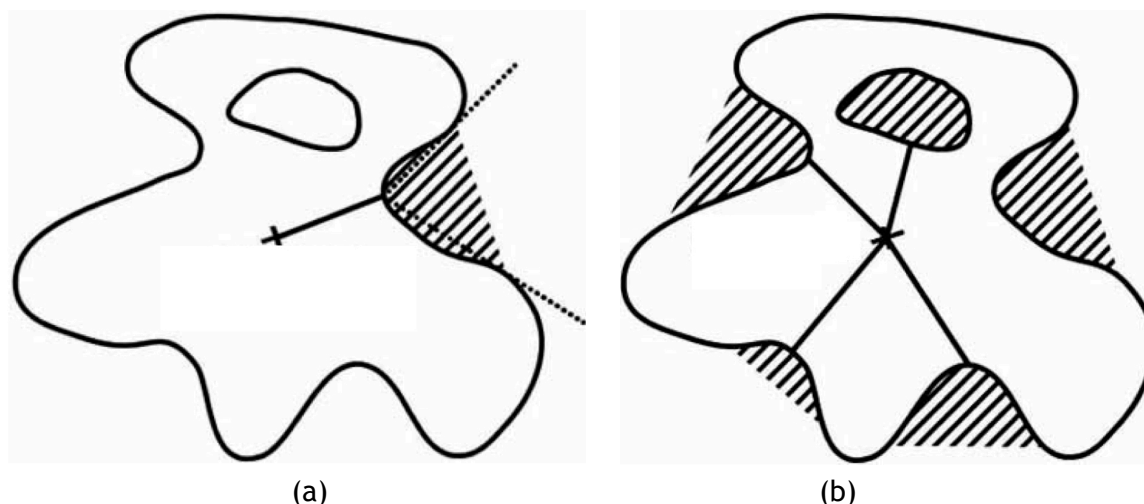


Figure 2.5: Detecting cavities using the center of gravity of the molecule and successive nearest surface atoms (NSAs): (a) the first cavity; (b) the remaining four cavities (picture taken from [LJ06]).

But, unlike Travel Depth, the depth is counted incrementally, rather than measured, from a grid cube to its neighbors. The ceiling of each cavity is found in a similar manner as in POCKET, i.e., it does not use any convex hull algorithm to find the outermost triangulated surface of the molecule.

2.4 Sphere-Based Algorithms

Sphere-based algorithms develop fundamentally around the concept of probe sphere. SURFNET seemingly is the first sphere-based algorithm developed by Laskowski [Las95]. The leading idea is to fill in cavities (e.g., indentations, surface grooves, and inner cavities or voids) with probe spheres of varying sizes. Note that it is not necessary to know in advance where the cavities are, because the probe spheres are fitted into the empty space between atoms. Basically, for every pair of atoms, we place a probe sphere centered at the midpoint of their atomic centers. Then the radius of the probe sphere is adjusted in order to guarantee that it does not overlap with any neighboring atoms, as illustrated in Fig. 2.6.

PASS is another sphere-based algorithm [BS00]. Cavity filling with probe spheres is carried out in layers, being sustained on three-point Connolly-like sphere geometry [Con83a], as shown in Fig. 2.7. A putative probe sphere has to survive to three filters to be considered as an effective sphere filling in a cavity [BS00]. The first filter determines that it cannot overlap with any atom belonging to the accretion substrate. The second filter determines that the probe cannot overlap any protein atoms. Finally, the third filter aims at ensuring that the probe is at some extent buried in the protein. The buriedness of a probe is determined by the number of protein atoms that lie within an empirical radius of 8 Å. Altogether, these filters determine the stopping condition of accretion of more spheres. Consequently, we end up having protein cavities filled with

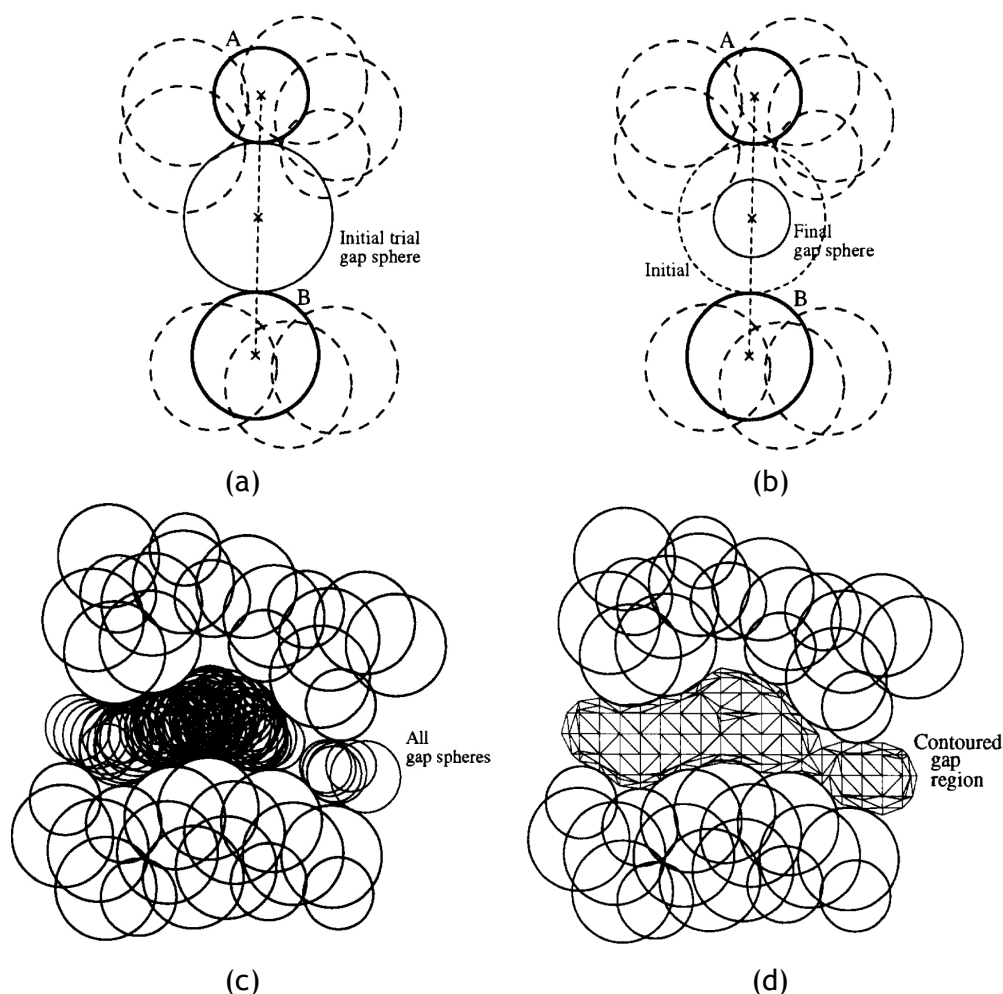


Figure 2.6: Detecting a cavity through SURFNET: (a) Each probe sphere is placed at the midpoint of a pair of atoms (A, B); (b) but, if such probe sphere overlaps at least an atom (dashed spheres), its radius has to be reduced until it just has a tangential contact with the overlapped atom; (c) all probe spheres placed into cavity after considering all pairs of atoms; (d) the cavity surface enclosing all probe spheres of the cavity (pictures taken from [Las95]).

a set of buried probe spheres, none of which sterically overlaps with the protein itself.

PHECOM is yet another sphere-based algorithm, and was developed by Kawabata and Go [KG07]. Similar to PASS, it also uses the three-point Connolly-like sphere geometry [Con83a] to coat the protein with a set of small probe spheres; the radius of each small probe sphere was set to 1.87 Å, which corresponds to the size of a single methyl group ($-\text{CH}_3$), as illustrated in Fig. 2.8(a). Additionally, PHECOM also produces a coating of the protein with large probe spheres, so that one removes small probe spheres that overlap with the large probe spheres, as shown in Fig. 2.8(b). Doing so, one considers that a cavity is an empty space into which a small probe sphere gets in, but not a large probe sphere; for example, this is shown in Fig. 2.8(c), where small probe spheres (in grey) overlap, which indicate where a cavity is located. Note that the probe spheres are allowed to overlap with each other, but not with protein atoms.

More recently the algorithm developed by Yu [YZTY10] is based in a imaginary sphere

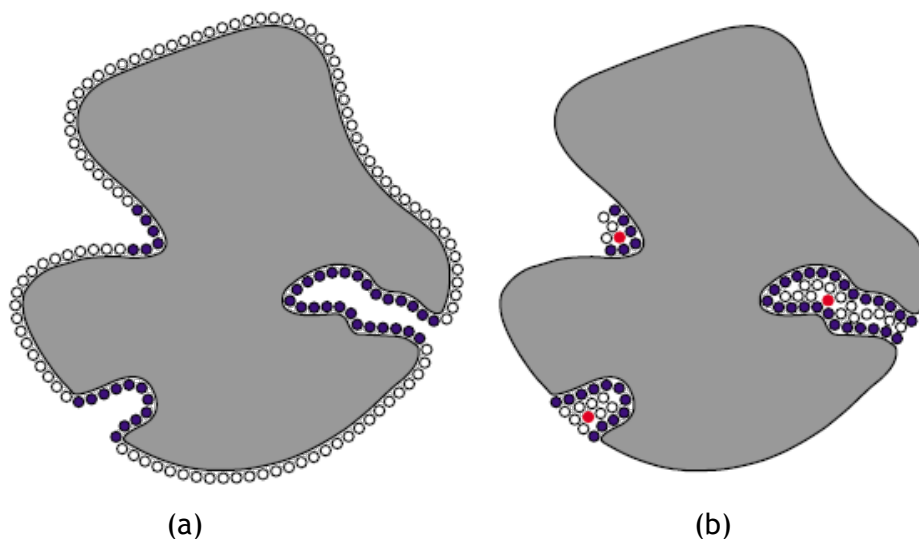


Figure 2.7: Detecting cavities through PASS: (a) coating the molecular surface with a first layer of probe spheres; (b) a second layer of probe spheres is, in this case, enough to find cavities on molecular surface (pictures taken from [WPS07], and inspired by [BS00]).

that rolls over protein surface. When the imaginary sphere rolls near the protein surface creates a probe surface based on the contact between the probe sphere and the protein surface. Finalized such process in the end, we have another surface that overlaps the molecular surface. However between both surface exist some voids that are defined has cavity regions.

2.5 Discussion

In this chapter surveying the state-of-the art in cavity detection on proteins, we have identified three major families of cavity detection algorithms: triangulation-based, grid-based, and sphere-based. None of them is without problems. Triangulation-based algorithms may miss some cavities, in particular narrow cavities. Grid-based algorithms suffer from ambiguity issues related with invariance to geometric transformations (e.g., rotations and translations) and delineation of cavity bounds. In regards to sphere-based algorithms, they utilize empirical sizes for probe spheres, therefrom resulting in difficulties in detecting and delineating cavities on proteins.

In the literature, we have noted the emergence of other two categories of algorithms that try to solve the aforementioned issues, namely: hybrid methods (mostly combining grid- and sphere-based approaches) and, flimsy, surface methods taking advantage of geometric properties and shape descriptors of smooth surfaces in differential geometry, like gradient, normal vector, and so forth. Surface methods do not use neither Voronoi decompositions, nor grids, nor probe spheres, and are potentially faster in their computations to find protein cavities.

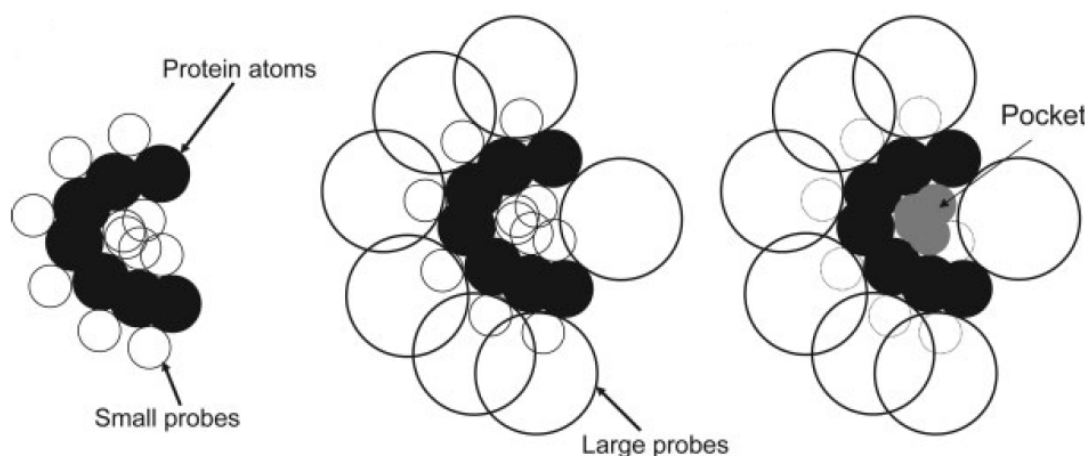


Figure 2.8: Detecting cavities through PHECOM: (a) Small probes are placed on the Van der Waals surface; (b) large probes are placed on the Van der Waals surface; (c) small probes that overlap with the large ones are removed (taken from [KG07]).

This thesis just introduces two surface-based algorithms to detect protein cavities. At the beginning of the research that is behind this thesis, there were not any surface-based algorithm of this sort in the literature.

2.6 Concluding Remarks

In this chapter, it was presented the key concepts for identifying, and characterizing a protein cavities in terms of geometrical algorithms methods. Also the discussion of some recent advances as well as limitations of the methods were also reviewed. Along the years various algorithms have tried to resolve the problem of detecting cavity region using only a geometric criteria. However the main issue in these algorithms is to define what is a cavity region, and which characteristics that region must have to be considered a possible site to detect protein cavities. Besides the problems presented by most of the algorithms, generally they are quick to detect cavity regions for small molecules, and easy to use.

Chapter 3

Triangulating Gaussian-like Molecular Surfaces

Triangulating molecular surfaces is an important requirement in computational biology and bioinformatics not only to visualize molecules and molecular complexes on computer screen, but also to measure their areas and volumes, as well as to infer useful information about interactions between molecules of a molecular complex. Triangulation algorithms for molecular surfaces are very expensive in terms of memory space and time performance, in particular when the number of atoms goes up to millions. Even so, this chapter aims to show that it is possible to triangulate and render molecules with a number of atoms in the order of magnitude of millions of atoms using a single desktop computer.

3.1 Introduction

Attempts to represent molecules on computer dates back to Levinthal's work [Lev66] in 1966, who used the stick model (i.e., a line segment for each bond) to display a 3D structure of a molecule on computer screen. However, as noted by Levinthal, the location of a molecular surface seems to be more important than the location of the bonds when there is a need to understand the interactions between molecules. Molecular surfaces are particularly helpful in the study of the structure and the function of proteins and nucleic acids, since molecular surfaces play an important role in the binding of macromolecules [XZ09]. For example, drug-nucleic acid interactions require a geometric representation of molecular surfaces.

It happens that a geometric representation for molecular surfaces is necessary not only to analyze interactions between molecules on computer, but also to render those molecules on computer screen for visual inspection purposes. It is clear that we can apply rendering techniques used in computer graphics to visualize molecular surfaces, namely ray tracing [MDG⁺10] and triangulation-based techniques [GVJ⁺09]. Note that triangulating a molecular surface starts from the principle that such a surface is analytic, i.e., it is well-behaved and without pathologies [KP02].

Several algorithms have been developed to triangulate and render molecular surfaces. In general terms, triangulation algorithms for molecular surfaces divide into three categories: continuation [AG03], space partitioning [BBB⁺97], and combinatorics [AE96, EM94, KCS⁺10]. However, for molecules with a large number of atoms (i.e., with thousands of atoms), most triangulation algorithms are costly in terms of memory storage and time consumption, staying thus far from real-time performance, in particular when

one uses non-parallel implementations of those algorithms.

Parallel algorithms for the triangulation of surfaces are more commonly found in the category of space partitioning algorithms, being the marching cubes (MC) algorithm the most popular of them [LC87] [NY06] [DCD⁺13]. However, these algorithms tumble in research fields other than molecular graphics and visualization. In the literature, we find parallel implementations of MCs based on CPUs and GPUs. Examples of parallel CPU-based implementations are those due to Mackerras [Mac92], Hansen [HH92], Sulatycke [SG02], Zhang [ZN04] and Wang [WJV07]. Parallel GPU-based implementations are mostly built upon shaders, with the drawback of the core of most algorithms does not run totally on GPU (cf. Geiss [Gei07], Uralsky [Ura06], Johansson [JC06]).

In respect to OpenCL-based implementations of MCs, they are scarce in the literature; two examples of these implementations are those due to Dias and Gomes [DG11] and Yi Peng et al. [PCY14], but only the first was designed to triangulate and render molecular surfaces. Finally, we have found a small number of CUDA based implementations of MCs in the literature, namely those due to Dias and Gomes [DBG10] [DG11] [DG13], Agostino et al. [DDG⁺12], and Petrescu et al. [PMMA11], but the latter was not designed to triangulate and rendering molecular surfaces.

The first MC algorithm designed to triangulate Gaussian molecular surfaces entirely on the GPU using CUDA was introduced by Dias and Gomes [DBG10] [DG11], but was limited to the usage of a single GPU, which was later extended to run on six GPUs within a single machine having 3 Nvidia GTX590 graphics cards (two 1.5GB GPU's each) [DG13]. With this setup, it was made possible to triangulate and render molecules having up to 60 thousands of atoms.

This chapter specifically addresses the problem of triangulating molecules owning up to one million atoms on a single computer equipped with one Nvidia Tesla K20 (5GB memory, 2496 CUDA cores) and one Nvidia Quadro K5000 (4GB memory, 1536 CUDA cores). The triangulation is accomplished using a parallel version of the well-known marching cubes algorithm, which takes advantage of the multicore processing of a single Intel Core i7 computer and of the OpenMP-CUDA technologies to distribute memory and processing load over those two graphics cards slotted inside a single computer.

The remainder of this chapter is organized as follows. Section 3.2 addresses the mathematical model underlying the Gaussian-like molecular surfaces, as well as the essentials of the MC algorithm for the triangulation of those surfaces. Section 3.3 provides an overview of our GPU-based triangulation algorithm that is based on marching cubes. Sections 3.4-3.10 detail the steps of our GPU-based triangulation algorithm. Section 3.11 describes the process of optimization of CUDA kernels. Section 3.12 carries out a comparative performance analysis of four implementations of MCs, namely: a CPU-based program, a multi-threaded CPU-based program, a GPU-based program using only one GPU, and a GPU-based program using two GPUs. Finally, Section 3.13 summarizes the main results produced by those programs, and points out directions for future work.

3.2 Background

3.2.1 Molecular Surfaces

There are various mathematical models for molecular surfaces, namely: van der Waals (VDW) surface, Lee-Richards surface, Connolly surface, blobby surface, and Gaussian surface. The VDW surface is described as the boundary of the union of (solid) balls that represent all atoms of a molecule. Lee-Richards surface, also known as the solvent-accessible surface (SAS), is an inflated VDW surface [LR71], which is obtained adding up the value of (water) solvent sphere radius, which is 1.4\AA , to the value of the VDW radius of each atom, approximately. Connolly surface [Con83b], also known as the solvent-excluded surface (SES), consists of two parts: contact surface and reentrant surface [Ric77]. The contact surface is made up by spherical patches of atoms that enter in contact with the solvent sphere; they are thus convex patches of the molecular surface. The reentrant surface consists of patches of the solvent sphere delineated by the contact points with two or more atoms simultaneously, which are of two types: saddle toroidal patches and concave spherical patches [Con83c].

The main problem with the previous three mathematical formulations is that they do not produce *smooth* molecular surfaces [DCC⁺13]. VDW and Lee-Richards surfaces obviously originate singular arcs resulting from the intersection between atomic balls. The Connolly surface is not smooth either, because its curvature may change suddenly from one patch to another [VH97], being also relatively easy to find redundant self-intersecting patches [Con83c]. On the other hand, in order to carry out energy computations and molecular simulations, we need to have smooth (i.e., infinitely differentiable) molecular surfaces as, for example, the blobby surfaces proposed by Blinn [Bli82].

The blobby surface is the result from summing up local analytic functions that describe the electron density field intensities of atoms [Bli82]. Each atom has its own electrical field, which is described by a Gaussian function that decays with the distance. Nevertheless, other Gaussian-like functions can be used to represent the behavior of the electrical field of an atom, namely: Wyvill function [WMW86] and reciprocal quadratic distance function [DBG10]. Taking this into consideration, and for the purposes of this chapter, we use the reciprocal quadratic distance function to describe the electron density field of each atom, which is given by:

$$f_i(x, y, z) = \frac{C}{d_i^2} \quad (3.1)$$

where $d_i = \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2}$ is the distance from the center (x_i, y_i, z_i) of the atom i to a generic point $(x, y, z) \in \mathbb{R}^3$, and $C \in]0, 1]$ stands for the smoothness or blobiness parameter. By summing all these distance functions associated to all atoms

we get the electron density field of the entire molecule as follows:

$$F(x, y, z) = \sum_{i=0}^{n-1} f_i(x, y, z) \quad (3.2)$$

where n is the total number of atoms of the molecule. The field intensity function $F : \mathbb{R}^3 \rightarrow \mathbb{R}$ given by Eq. (3.2) corresponds to a level set in the product space $\mathbb{R}^3 \times \mathbb{R}$. This level set feature a number of molecular surfaces that satisfy the equation $F(x, y, z) = T$, where T is the isovalue (also called threshold). That is, different level set values of $T \in \mathbb{R}$ produce distinct molecular surfaces in the domain \mathbb{R}^3 of the function F [BS91], much like the nesting Matryoshka doll. Note that molecular surfaces of this sort are essentially implicit surfaces; hence, points outside the surface satisfy the condition $F > T$, while points inside the surface satisfy $F < T$, and $F = T$ is valid for points on the surface. Besides, F is smooth because results from the summation of smooth blending functions f_i .

3.2.2 Marching Cubes' Triangulations

Marching cubes' triangulations were introduced by Lorensen and Cline [LC87], who used them to extract triangulated isosurfaces from volumetric medical datasets (e.g., the triangulated surface of a skull). Basically, extracting an isosurface was made possible because it is an operation to find the 3D pixels (sometimes called voxels) in a 3D volumetric dataset (i.e., stack of imaging slices) having intensities that match the isovalue associated to a specific anatomic organ (cf. Hounsfield scale [Fee10]).

Unlike the original marching cubes algorithm, the algorithm described in this chapter operates on an analytic function F that represents a molecular surface. This means that we must compute the value of F (cf. Eq. (3.2)) at every single corner of all cubes that constitute the boxed domain. These cubes are here called voxels. When the calculation of the intensity field F is accomplished for the eight corners of a voxel, we have to activate the corresponding 8-bit flag (1 bit per corner), in order to retrieve the configuration of the triangulated surface patch inside each voxel from a pre-defined lookup table. This is done by setting the corresponding flag bit to 1 if $F \geq T$ at a corner of a voxel, or else is set to 0. Such corner classification allows to know if a voxel is either above (corner bit takes on the value 1) or below (corner bit takes on the value 0) the isovalue T . That means that inside each voxel we may have 1 out of 256 possible configurations or triangulations represented in the aforementioned lookup table.

More specifically, the 8-bit flag associated to each voxel is used to retrieve the triangulation of the surface inside the voxel from a lookup table which is a two-dimensional array with 256×16 elements. In this lookup table, each one of the 256 entries (or configurations) consists of 16 elements, where each element represents an edge of a voxel that is intersected by the isosurface. Also, every single triple of elements (edges) of each entry refers to surface-intersecting edges for which we have to determine the

corresponding vertices of a triangle. For example assuming that we have a voxel with the flag $00001100_2 = 12_{10}$, that corresponds the following entry in the lookup table:

3, 10, 1, 11, 10, 3,-1,-1,-1,-1,-1,-1,-1,-1,-1

which represents a voxel within which we find two triangles, being the first triangle defined by three surface points located in the edges numbered as 3, 10, 1, and the second triangle is formed from surface points in the edges numbered as 11, 10, and 3; the element -1 is the terminator of each table entry, that is, the element that indicates the end of each entry. Taking into account that we have always this terminator element in each entry, we can conclude that the number of triangles inside a given voxel is 5 maximum.

Found the edges that cross the surface, we proceed to the computation of the surface points that result from the intersection between the isosurface and the edges of each voxel. The generation of these surface points is done by linear interpolation of the function values on the extremities of each voxel edge. Then, for each voxel, one proceeds to the generation of the corresponding set of triangles. The final result of this voxel triangulations is a triangular mesh that approximates the isosurface.

3.3 GPU-based Triangulation: Overview

The GPU-based triangulation proposed in this chapter takes advantage of the multi-core Intel Core i7 CPU and the multi-threaded OpenMP API (Application programming Interface); more specifically, we use OpenMP to distribute computation workload by two cores of the i7 CPU, each one of each will be associated to a GPU lying in a programmable graphics card (Fig. 3.1). The first GPU is the one of an Nvidia Quadro K5000, while the second GPU is part of an Nvidia Tesla K20.

In general terms, our algorithm can be described as follows:

1. Reading of atoms of a given molecule in CPU side memory.
2. Computation of the bounding box enclosing the input molecule.
3. Voxelization of the bounding box.
4. Slicing of the voxelized bounding box into 2 sub-boxes.
5. Memory allocation on each graphics card (GPU side) for atoms, sub-boxes, marching cubes lookup tables, and other supplementary data.
6. Launching of two CPU core threads, one per GPU.
7. GPU triangulation of the molecular surface inside each sub-box.

8. Junction of the sliced triangulations of the molecular surface into a VBO (vertex buffer object) on Quadro K5000.
9. Rendering of the triangulation existing in the VBO.

Each step of the previous algorithm is detailed in the next sections. The first six steps are operations triggered on CPU side. But, the fifth step involves CUDA memory allocation operations on GPU side, that is, `cudaMalloc()` calls, while the sixth step launches the OpenMP threads on CPU cores. The last three steps exclusively operate on GPU side. The next sections detail each of these steps.

3.4 Reading Atoms in CPU Side Memory from a PDB/VDB File

This is the first step of the algorithm. This step reads the centers $c_i = (x_i, y_i, z_i)$, with $i = 1, \dots, n$, of the atoms of a given molecule from a PDB (Protein Data Bank) file, which are then stored in three one-dimensional arrays, named CXARRAY, CYARRAY, and CZARRAY, each one for each sort of coordinates, namely x , y , and z coordinates of atom centers, respectively. In case of molecules with several chains, we use a VDB file. This type of file format is essentially the PDB format in the Virus Particle Explorer database (VIPERdb) coordinate system. VIPERdb is a database for icosahedral virus capsid structures (<http://viperdb.scripps.edu/>).

3.5 Computation of the Bounding Box

The second step of the algorithm consists in computing the axis-aligned bounding box that encloses the entire molecule. This is done sorting the values of CXARRAY, CYARRAY and CZARRAY in an increasing manner. This way, the bottom front left corner of the box is represented by the triple $(x_{min}, y_{min}, z_{min})$, whose values are the 0-th elements of those three arrays. Similarly, the top back right corner of the box is the triple $(x_{max}, y_{max}, z_{max})$, whose values correspond to the $(n - 1)$ -th elements of those arrays. To make sure that the molecule is completely inside the bounding box, we have to subtract the value $10 \times \Delta$ to the minimum values $(x_{min}, y_{min}, z_{min})$, as well to add the same value $10 \times \Delta$ to the maximum values $(x_{max}, y_{max}, z_{max})$ that delimit the bounding box, being Δ the size of each voxel.

3.6 Voxelization of the Bounding Box

After finding the axis-aligned bounding box that encloses a given molecule, and given the voxel length Δ , we can easily determine the number of $N = I \times J \times K$ of voxels inside

the bounding box, where I , J , K stand for the number of voxels in the x , y , z directions, respectively. It is clear that we do not need to explicitly partition the bounding box into voxels, because we only need the values of Δ , I , J , and K to compute the triangulation of the molecule inside the bounding box. In fact, it is enough to calculate and store the coordinates of the bottom front left corner (i.e., 0-th corner) of each voxel into an array named VXLARRAY. This means that each voxel is represented in VXLARRAY by its 0-th corner.

3.7 Slicing of the bounding box

This step also takes place on CPU side. Partitioning the bounding box into two sub-boxes aims at dividing the computation burden by two available high-performance computing devices, namely the Nvidia Kepler Tesla K20 and the Nvidia Quadro K5000. Note that the partitioning of the bounding box into two equally-sized sub-boxes (i.e., slices) is done in the z direction by its median value, so that VOXELA splits into smaller arrays, VOXELA0 and VOXELA1.

3.8 GPU Memory Allocation

Before starting the computations concerning the triangulation of a given molecular surface on GPU devices, we need to transfer specific data from CPU side to GPU side. These data divides into two categories: common data and exclusive data.

Common data are data that are transferred to both GPU devices; for example, the lookup tables of the marching cubes algorithm, the array of atom centers, and the (empty) array of N function intensities, where N stands for the number of 0-th vertices featuring voxels. More specifically, we need to transfer the following data:

- *Lookup table of triangulations.* This is the first lookup table of the marching cubes algorithm (see sub-section 3.2.2). This lookup table (named LUTR) has 256 entries which correspond to 256 possible triangulations of the surface inside a voxel. This table is copied into the texture memory of each CUDA device (graphics card).
- *Lookup table of triangulation vertices.* This lookup table (named LUTVV) is similar to the LUTR, but accounts for the number of vertices in each one of 256 possible triangulations inside a voxel. This table is also copied into the texture memory of each CUDA device.
- *Arrays of atom centers.* These are the CXARRAY, CYARRAY and CZARRAY arrays that hold the x , y , and z coordinates of atomic centers of a molecule. These arrays are copied into the global memory of each CUDA device.

- *Array of function values.* This array, called FARRAY, has N values initialized to 0, a function value per each 0-th corner of each voxel. Supposedly, the value that the function F (see Eq. (3.2)) takes on the 0-th corner of each voxel is calculated later on during the execution of a specific kernel on GPU side.

Exclusive data is data that is tied to each sub-box of voxels exclusively. This data is dispatched to the global memory of each CUDA device, and includes the following arrays:

- *Array of voxels.* This array is called VOXELA and is created during the slicing of the bounding box (see section 3.7). This array holds the 0-th corners of all voxels of each sub-box. In practice, we use three uni-dimensional arrays instead of a single three-dimensional array, that is, we use three arrays for the x, y and z coordinates of 0-th corners of voxels; they are called VOXELAX, VOXELAY, and VOXELAZ, respectively. These arrays are specific to each sub-box and are copied into the corresponding CUDA device.
- *Array of voxel flags.* This array is called FLAGA and holds the 8-flags associated to voxels belonging to each sub-box, as usual in the MC algorithm. When it is transferred to its CUDA device, it carries all flags initialized to zero. Thus, this array is supposed to be updated later in the CUDA kernel that calculates the 8-bit flag associated to each voxel. Recall that this 8-bit flag is the key that is necessary to retrieve the triangulation inside each voxel and the corresponding number of vertices from the lookup tables LUTR and LUTVV, respectively.
- *Array of number of voxel vertices.* This array is called NARRAY and, similarly to FLAGA, has size $m = N/2$, where m is the number of voxels of each sub-box. This array is also initialized to 0 before transferring it to the corresponding CUDA device, being it necessary to accommodate the number of triangulation vertices for each voxel. This number of vertices is also supposed to be determined later by one of the kernels running on each CUDA device.

These arrays are allocated in the GPU side memory between the 4-th and 5-th kernels because their sizes will be only known after the 4-th kernel. After allocating memory and copying the necessary data (e.g., MC lookup tables), we are ready to launch OpenMP threads on CPU cores and to invoke CUDA kernels on GPU to carry out the triangulation of a given molecular surface.

3.9 Launching OpenMP Threads to Invoke GPU CUDA kernels

The distribution of the computation burden is done using the multi-threaded OpenMP library [CJvdP07]. This allows us to have a setup that maps each CPU core (a single

thread) to a single CUDA device, that is, each thread `tid` is associated to the function `cudaSetDevice()`, as shown in the following code snippet:

```
#pragma omp parallel shared(n_verts)
{
    int tid = omp_get_thread_num();
    switch (tid)
    {
        case 0 :
            cutilSafeCall(cudaSetDevice(0));
            // 0-th sub-box computations on Quadro K5000
            ...
        case 1:
            cutilSafeCall(cudaSetDevice(1));
            // 1-st sub-box computations on Tesla K20
            ...
    }
}
```

Therefore, the CUDA device i is associated to the OpenMP thread i , with $i = 0, 1$. It is clear that this setup is scalable, but it is limited by the number of graphics card slots available in a single computer. Note that the computations taking place on each CUDA device refer to the triangulation of the molecular surface in each sub-box.

3.10 Triangulation on CUDA devices

After transferring data to the GPU side and launching CPU OpenMP threads associated to CUDA devices, we are ready to start with the triangulation of the molecular surface within both sub-boxes simultaneously. As illustrated in Fig. 3.1, this triangulation on each GPU involves the execution of 6 CUDA kernels as described below.

3.10.1 Computation of function values (1-st kernel)

This kernel computes the value of F at the 0-th corner of every single voxel of its associated sub-box. Note that this computation is performed in a per-atom basis, instead of in a per-voxel basis, because the number of voxels usually exceeds the number of atoms (cf. Table 3.4). Thus, using per-atom computations of F speeds up the first kernel on GPU side.

For this purpose, and taking into consideration that the value of Gaussian function f_i associated to each atom decays with the distance to the center of atom i , we proceed to the calculation of f_i at each 0-th corner of the sub-box of $15 \times 15 \times 15$ voxels that surrounds such an atom i . It is clear that f_i at a given 0-th corner is then added up

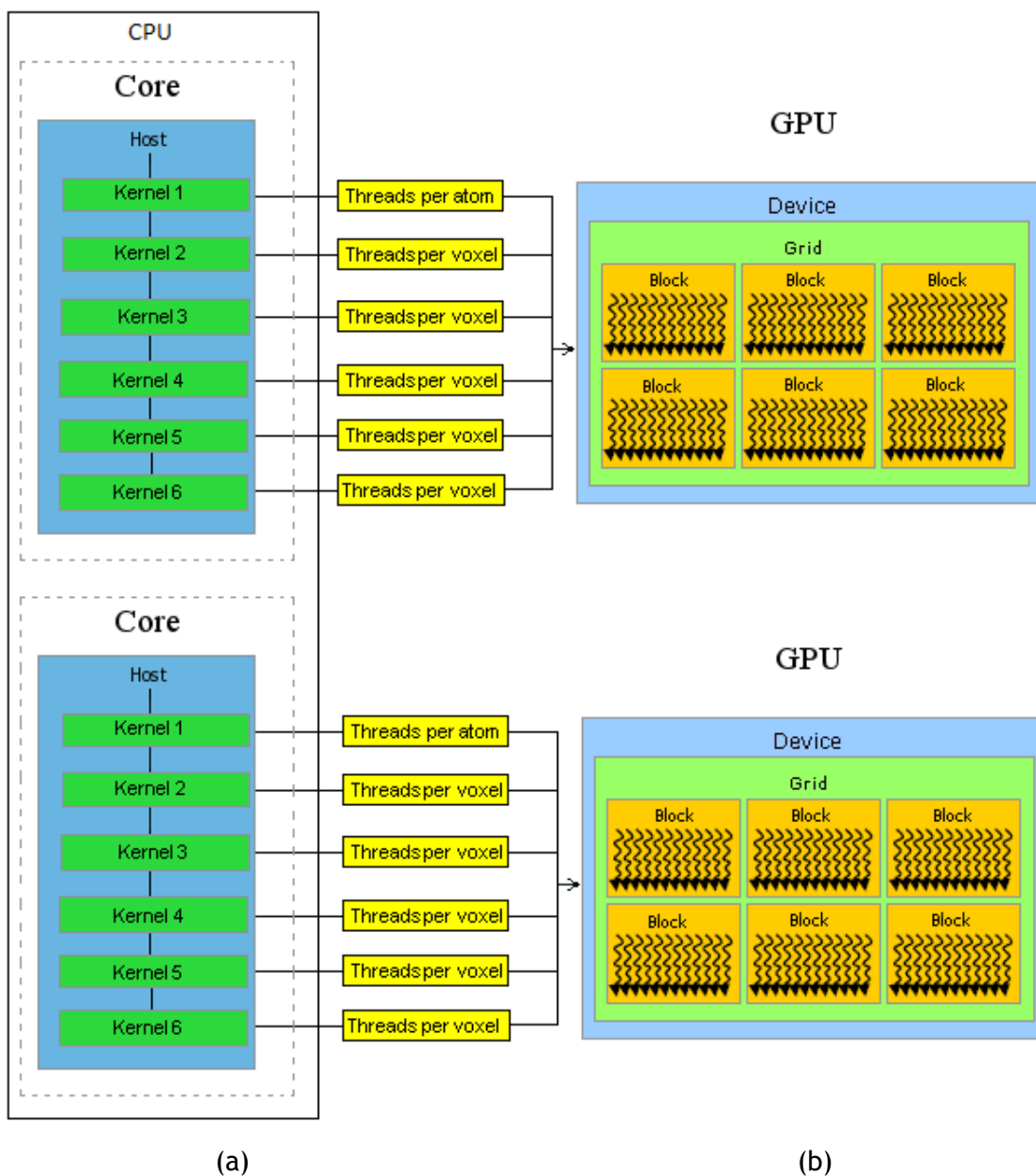


Figure 3.1: The distribution of the computation overhead by 2 CPU cores and 2 GPUs.

to the current value F at such a 0-th corner. The parallelization of this computation is done by running a single CUDA thread for each atom, so we end up having n CUDA threads for n atoms running simultaneously.

3.10.2 Computation of voxel flags (2-nd kernel)

The second CUDA kernel calculates the 8-bit flags of the voxels of its associated sub-box. Each flag bit indicates the position of each voxel corner in relation to the molecular surface. A flag bit takes on the value 1 if $F \geq T$, where $T = 0.8\text{\AA}$ is the threshold; otherwise, it remains with the initial value of 0. Performing this computation for all of the m voxels on each CUDA device, we end up having an one-dimensional array of m

flags (named FLAGA) on each CUDA device. It is clear that this computation is performed in a per-voxel basis on each CUDA device, i.e., a CUDA thread per voxel.

3.10.3 Computation of the number of triangulation vertices associated to each voxel (3-rd kernel)

This computation is also performed in a per-voxel basis on each CUDA device. The 3-rd kernel aims to compute the number of triangulation vertices associated to each voxel. This computation is performed using the 8-bit flag associated to such voxel, which was calculated in the 2-nd kernel described above. Using this 8-bit flag as a key for the LUTVV lookup table, we retrieve the number of vertices associated to each voxel, being this number stored into the previously allocated NVARRAY array.

3.10.4 Computation of the total number of triangulation vertices (4-th kernel)

This 4-th kernel also operates in a per-voxel basis on each CUDA device. Calculated the number of triangulation vertices associated to each voxel in the 3-rd kernel, we are ready to calculate the total number of vertices, three vertices per triangle. This task is done using the parallel prefix sum `cuDppScan` [HSO07] on NVARRAY, which outputs the total number of triangulation vertices for the molecular surface within each sub-box on GPU side. This sum operation is thus another GPU kernel. After running this 4-th kernel, we are at position of allocating memory on GPU side for the mesh triangles that will approximate the molecular surface, because now we know how many triangles will make part of the mesh.

Basically, we allocate a VBO (Vertex Buffer Object) array on Quadro K5000 (first device) that is organized in triples of vertices, a triple per triangle, well as an NBO (Normal Buffer Object) array of the same size as the VBO array to store the vertex normals also on the same graphics card. Also, we allocate memory for pseudo-VBO and pseudo-NBO arrays on Tesla K20 (second device) to temporarily store triangulation vertices and their normals of the surface inside the first sub-box, before copying those arrays directly to Quadro K5000. These two pseudo arrays are not neither VBO nor NBO array because the Tesla K20 is not prepared for graphics output, so that they need to be transferred to a VBO and a NBO on the Quadro K5000.

3.10.5 Computation of the triangle vertices (5-th kernel)

Also in a per-voxel basis, and after allocating memory for vertices (or triangles) of the surface mesh, we calculate the triangulation vertices associated to each voxel by finding the intersection points between the molecular surface and the edges of such voxel, what is done using linear interpolation. These triangulation vertices are then stored in a VBO array previously allocated on Quadro K5000 graphics card. Note that

the voxel edges that intersect the surface are retrieved from the LUTR lookup table using the 8-bit flag associated to a given voxel.

3.10.6 Computation of normal vectors to the surface (6-th kernel)

For graphics output of the molecular surface, we have to calculate the vectors that are normal to the surface at the triangulation vertices within each sub-box. This is performed by calculating the normal vector at each vertex of the VBO array, which is given by the gradient vector

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right) \quad (3.3)$$

that is, a triple of partial derivatives of the function F that analytically describes the molecular surface. Then all the normals are saved in the NBO array.

3.10.7 Merging Partial Triangulations and Rendering

Multi-GPUs systems are able to carry out massive computations on GPUs, but we should be careful to prevent eventual bottlenecks in the transfer of data from CPU side memory to GPU side memory, and vice-versa. These bottlenecks are particularly common in mid-range Nvidia graphics cards like GTX series 500 and 600. To avoid this backdrop, Nvidia launched the unified Nvidia Maximus technology for high-range graphics cards like Quadro K5000 and Tesla K20 to deliver dedicated floating-point horsepower for interactive design graphics as well as computational rendering.

This GPU compute technology allows for running jobs on each GPU, making sure that massive computations go to Tesla K20 and, eventually, to Quadro K5000, while all graphics calls go to Quadro K5000 exclusively. This means that when vertex buffer object (VBO) arrays are created to store vertex and/or vertex normals, the primitive calls are recognized as a graphical calls, and are thus automatically dispatched and processed on Quadro K5000. Therefore, the programmer just has to take care of the allocation of VBO arrays, that the unified Nvidia Maximus driver does the rest in respect to automatically transfer data from Tesla to Quadro for graphics output.

3.11 Optimization of CUDA Kernels

As mentioned above, we used the following two compute devices: one Nvidia Tesla K20 (exclusively for calculations) and one Nvidia Quadro K5000 (for calculations and graphics output). The memory and compute capabilities of these two devices are listed in Table 3.1. Both devices follow the Kepler architecture; hence, the letter 'K'. This explains

why the maximum number of active blocks (16), warps (64), threads (2048) scheduled simultaneously per multiprocessor are the same for both devices. The main difference between them lies in the memory size and in the number of streaming multiprocessors. But, the number of cores per multiprocessor is exactly the same and is equal to 192; in fact, for the Nvidia Tesla K20 we have $2,496/13=192$ cores per multiprocessor, while for Nvidia Quadro K5000 we have $1,536/8=192$ cores per multiprocessor. Recall that a kernel is a function executed on GPU as a grid of blocks of warps of threads; a block consists of 32 warps, while a warp consists of 32 threads, what results in at most 1,024 threads per block (cf. Table 3.1).

Device Type	Tesla K20	Quadro K5000
Memory Size (GB)	5.0	4.0
Streaming Multiprocessors (SM)	13	8
CUDA Cores	2,496	1,536
Maximum Number of Active Blocks per SM	16	16
Maximum Number of Active Warps per SM	64	64
Maximum Number of Active Threads per SM	2,048	2,048
Maximum Number of Threads per Block	1,024	1,024
Maximum Number of Warps per Block	32	32
Maximum Number of Registers per Thread	255	64
Maximum Number of Registers per Block	65,536	65,536

Table 3.1: Kepler GPU's memory and compute capabilities.

The optimization of the kernels depends on the architecture, type and capabilities of device that one intends to use. In our case, we have two devices with the same Kepler architecture, but different capabilities in terms of memory and multiprocessing (cf. Table 3.1). The compute capability of the Nvidia Tesla K20 is 3.5, while it is 3.0 for the Nvidia Quadro K5000. The compute capability determines how the CUDA code (kernels) is compiled using `nvcc`; more specifically, the flag `-arch` serves to specify which compute capability is about being used. After compiling and linking the code into an executable code, we are ready to run the program on CPU side, from where the kernels are called to be executed on the GPU side (see Fig. 3.1).

Optimizing code assumes that we perform a preliminary analysis of performance of the kernels. This is done using an Nvidia profiler called Nsight (<http://www.nvidia.com/object/nsight.html>). The performance data outputted by Nsight before optimizing CUDA code is shown in Table 3.2. For this purpose, we run the program and the Nsight profiler several times for the following molecules: 110D (120 atoms), 1FFY (9,010 atoms), and 1HTO (97,872 atoms) (cf. Table 3.4). After optimizing the CUDA code of the kernels, we obtain the figures shown in Table 3.3.

In order to take advantage of the massive computation power of Nvidia GPUs, we have to optimize the CUDA kernels described above in terms of streaming multiprocessor occupancy. More occupancy of multiprocessors of a CUDA device (e.g., Nvidia Tesla K20) means more efficiency and speed in computations taking place on such a device. In fact, the most important optimization profiling parameter is the streaming multiprocessor

# kernel	Active Blocks	Active Warps	Active Threads	Threads per Block	Warps per Block	Registers per Thread	Registers per Block	Multiprocessor Occupancy
1st kernel	16	16	16	160	16	84	768	25%
2nd kernel	7	20	1280	160	10	11	560	53%
3rd kernel	5	20	768	160	10	18	180	53%
4th kernel	8	20	768	160	10	19	609	53%
5th kernel	4	30	768	160	10	82	680	40%
6th kernel	6	20	1920	160	10	74	407	53%

Table 3.2: Performance data *before* optimizing the CUDA kernels.

# kernel	Active Blocks	Active Warps	Active Threads	Threads per Block	Warps per Block	Registers per Thread	Registers per Block	Multiprocessor Occupancy
1st kernel	16	32	1024	192	32	71	3587	62%
2nd kernel	12	60	1920	192	5	11	2560	93%
3rd kernel	12	60	1920	192	5	8	1280	93%
4th kernel	12	60	1920	192	5	9	2560	93%
5th kernel	10	40	1280	192	5	68	7680	70%
6th kernel	12	60	1920	192	5	24	3840	93%

Table 3.3: Performance data *after* optimizing the CUDA kernels.

occupancy of each kernel because it indicates whether a kernel is running properly or not. This parameter provides us the ratio of the number of running threads to the theoretical maximum number of threads that are supposed to run on the GPU. In other words, a kernel with low streaming multiprocessor occupancy means that its code is inefficient, so that it needs to be optimized.

The first and most important optimization step is changing the number of threads per block in CUDA code. Nsight tool suggested that the right number of threads per block is 192 out of 1024, which is the maximum number of threads per block in the Kepler architecture (cf. Table 3.1). Then, we changed this number from 160 (cf. Table 3.2) to 192 (cf. Table 3.3) in our CUDA program. As a consequence, the multiprocessor occupancy increased for most kernels, but it was more noticeable for the 4th and 6th kernels. Note that the thread block size must be a multiple of 32 so that, depending on the graphics card on which the code is supposed to run, it usually varies between 64 and 256.

In order to further improve the streaming multiprocessor occupancy of the kernels, we proceeded as follows:

- *To increase the number of active warps.* This is done by reducing, as much as possible, the number of conditional statements (i.e., if-else and switch statements) and the number of looping statements (i.e., for and while statements) in the CUDA kernels. The existence of this sort of statements in the 1st and 5th kernels, even after removing many of them in the preliminary code of the kernels, explain why they are less efficient than other kernels (cf. Table 3.3).
- *To decrease the number of registers per thread.* For this purpose, we have to

decrease, as much as possible, the number of mathematical operations carried out by a kernel. Also, using the CUDA mathematical operators helps to reduce the number of registers per thread because they are already optimized for CUDA code. For example, the optimized 5th kernel uses the parallel prefix sum (scan) of CUDA to calculate the sum of the number of vertices of the triangulation existing in an array of N entries, an entry per voxel. Note that the multiprocessor occupancy of the 1st and 5th kernels did not increase that much after the code optimization because the number of registers per thread for the 1st and 5th kernels did not decrease significantly either (cf. Table 3.2 and 3.3). But, their practical multiprocessor occupancy is higher than the theoretical counterpart; for example, the theoretical multiprocessor occupancy of the 1st kernel is 50% because the number of active threads is 1024, i.e., half the maximum number of active threads per multiprocessor, but in practice the multiprocessor occupancy is 62%. This is explained by the less number of registers we are using per kernel.

- *To fix unbalanced workloads.* There are two ways of dealing with this problem. The first solution is to use more blocks per kernel grid. This helps to distribute the work across the blocks in a more balanced way. The second solution consists in using a growing number of sub-kernels that is proportional to the number of atoms of the molecule. This is particularly useful for large molecules. Here the idea is to split the kernel grid into more blocks in order to distribute the work in a more balanced way, avoiding this way getting the final result on a single block, warp, or thread.

Looking at Table 3.3, we see that the number of registers per thread concerning the 1st and 5th kernels exceeds the maximum number of registers per thread (64) of the Quadro K5000 (cf. Table 3.1). However, this does not prevent us to use this device for GPU massive computations, although at the expense of a little less performance. Here what happens is that when the thread data are too big to fit in the multiprocessor registers, the data are spilled out to local memory (L1/L2 cache) or even to the global memory if needed.

3.12 Results and Performance Evaluation

The algorithm performance tests were carried out in a Nvidia Maximus system with a Windows 7 PC (64 bit version) powered by an Intel Core i7 4820K CPU (8 cores), 3.70Ghz clock, 32GB RAM, and with two graphics cards (Nvidia Quadro K5000 and Nvidia Tesla K20). Testing described in this chapter were done within Microsoft Visual Studio 2010, with Intel C compiler, and with the CUDA version 5.5. For that purpose, we designed and implemented four C language programs for the same algorithm as follows:

- *CPU-based serial program.* This program does not use any multi-threading facili-

ID	# Chains	# Atoms per Chain	# Atoms	CPU Time (sec)	8-CPU Time (sec)	GPU Time (sec)	2 GPUs Time (sec)	# Vertices	# Voxels	Total Memory (MB)
110D	0	0	120	0.062	0.060	0.045	0.028	4890	27440	214.004
200D	0	0	259	0.343	0.335	0.138	0.045	9558	42670	206.422
4pti	0	0	454	0.421	0.415	0.172	0.076	15156	66864	209.591
2OT5	0	0	545	0.453	0.448	0.422	0.056	18882	74281	209.122
1hh0	0	0	691	0.530	0.521	0.386	0.074	20736	100714	210.170
1hgv	0	0	691	0.562	0.514	0.438	0.078	21648	118657	209.437
1hgz	0	0	691	0.515	0.507	0.518	0.075	20856	99045	210.447
2INS	0	0	781	0.624	0.537	0.560	0.089	21348	79425	208.230
1QL2	0	0	966	0.858	0.821	0.474	0.128	42918	243984	216.328
1NEQ	0	0	1187	0.733	0.714	0.698	0.147	26058	92751	217.029
1A63	0	0	2065	1.482	1.458	0.754	0.248	37146	123024	211.732
3GME	0	0	3695	4.265	4.159	1.234	0.374	70896	335891	219.854
1BIJ	0	0	4384	5.180	5.127	1.398	0.468	76236	363888	220.432
1G50	0	0	5884	9.297	8.867	1.429	0.567	111534	635438	230.063
3SGZ	0	0	7912	17.254	16.458	1.576	0.695	159120	966016	240.740
1Z7X	0	0	8928	15.553	14.745	1.849	0.943	123924	538560	228.593
1FFY	0	0	9010	19.874	17.956	2.023	0.872	159222	922300	242.546
1AF6	0	0	10052	19.219	17.330	2.134	0.898	144114	522144	228.640
1A8R	0	0	26400	161.569	155.365	5.678	2.238	500796	4103958	376.417
1GTP	0	0	34740	208.791	198.478	7.356	4.896	389214	2069879	292.777
3RG2	0	0	44567	368.348	361.951	7.732	5.178	664128	4389665	398.882
3UOQ	0	0	55711	457.081	438.750	9.788	5.876	655200	5225634	415.352
1OTZ	0	0	68620	790.360	777.987	11.754	7.674	953712	4716425	424.709
1IR2	0	0	77088	587.965	580.892	11.890	8.963	637020	3593035	346.877
1HTO	0	0	97872	1071.862	1068.715	14.389	9.459	970002	5256893	431.869
1X9P	60	3677	220620	6368.641	6270.489	49.531	38.435	2412948	15687750	783.547
39ME	6	80664	483984	52641.150	49254.074	185.626	87.835	8638758	74973805	2755.435
1M1C	60	10302	618120	50090.580	48951.438	196.635	98.364	6449232	47964279	1897.838
1OHG	60	15070	904200	113294.348	110564.365	439.637	158.448	9785134	69732156	4545.113
1HTQ	10	90695	906950	114896.547	112697.478	445.836	169.827	9794567	69842317	4689.471

Table 3.4: Time performance/memory occupancy for a number of molecular surfaces *before* optimization.

ties of the multicore i7 CPU. No resources of the GPU are used at all.

- *8-core CPU-based parallel program.* This program uses multi-threading facilities of the 8-core i7 CPU. No resources of the GPU are used at all either.
- *A 1-GPU-based parallel program.* This program does not take advantage of CPU multi-threading, but leverages the computation power provided by the Nvidia Tesla K20 via CUDA API multi-threading. The Nvidia Quadro K5000 is only used for graphics output of molecular surfaces.
- *A 2-GPU-based parallel program.* In this case, we use two i7 CPU cores that run two OpenMPI threads, each one of which is associated to a distinct graphics card (Fig. 3.1). The 0-th thread on CPU side calls CUDA kernels on Nvidia Quadro K5000, while the 1-st thread on CPU side calls CUDA kernels on Nvidia Tesla K20. In this case, the Nvidia Quadro K5000 is used not only for graphics output, but also for massive computations as a second GPU device.

In our experiments, we considered a dataset with several tens of molecules, including those listed in Table 3.4 (also in Table 3.5). We considered molecules without and with

ID	# Chains	# Atoms per Chain	# Atoms	CPU Time (sec)	8-CPU Time (sec)	GPU Time (sec)	2 GPUs Time (sec)	# Vertices	# Voxels	Total Memory (MB)
110D	0	0	120	0.062	0.060	0.029	0.016	4890	27440	214.004
200d	0	0	259	0.343	0.335	0.098	0.031	9558	42670	206.422
4pti	0	0	454	0.421	0.415	0.172	0.063	15156	66864	209.591
20T5	0	0	545	0.453	0.448	0.219	0.047	18882	74281	209.122
1hh0	0	0	691	0.530	0.521	0.506	0.062	20736	100714	210.170
1hgv	0	0	691	0.562	0.514	0.558	0.063	21648	118657	209.437
1hgz	0	0	691	0.515	0.507	0.450	0.063	20856	99045	210.447
2INS	0	0	781	0.624	0.537	0.582	0.078	21348	79425	208.230
1QL2	0	0	966	0.858	0.821	0.665	0.109	42918	243984	216.328
1NEQ	0	0	1187	0.733	0.714	0.603	0.125	26058	92751	217.029
1A63	0	0	2065	1.482	1.458	0.791	0.203	37146	123024	211.732
3GME	0	0	3695	4.265	4.159	1.178	0.357	70896	335891	219.854
1BIJ	0	0	4384	5.180	5.127	1.211	0.421	76236	363888	220.432
1G50	0	0	5884	9.297	8.867	1.373	0.500	111534	635438	230.063
3SGZ	0	0	7912	17.254	16.458	1.467	0.686	159120	966016	240.740
1Z7X	0	0	8928	15.553	14.745	1.728	0.889	123924	538560	228.593
1FFY	0	0	9010	19.874	17.956	1.801	0.858	159222	922300	242.546
1AF6	0	0	10052	19.219	17.330	2.028	0.874	144114	522144	228.640
1A8R	0	0	26400	161.569	155.365	4.158	1.997	500796	4103958	376.417
1GTP	0	0	34740	208.791	198.478	6.282	3.697	389214	2069879	292.777
3RG2	0	0	44567	368.348	361.951	6.616	4.275	664128	4389665	398.882
3UOQ	0	0	55711	457.081	438.750	7.889	4.119	655200	5225634	415.352
1OTZ	0	0	68620	790.360	777.987	10.655	6.927	953712	4716425	424.709
1IR2	0	0	77088	587.965	580.892	10.491	7.800	637020	3593035	346.877
1HTO	0	0	97872	1071.862	1068.715	13.949	8.860	970002	5256893	431.869
1X9P	60	3677	220620	6368.641	6270.489	38.892	26.707	2412948	15687750	783.547
39ME	6	80664	483984	52641.150	49254.074	162.405	75.698	8638758	74973805	2755.435
1M1C	60	10302	618120	50090.580	48951.438	173.496	87.247	6449232	47964279	1897.838
1OHG	60	15070	904200	113294.348	110564.365	426.413	125.356	9785134	69732156	4545.113
1HTQ	10	90695	906950	114896.547	112697.478	431.589	130.745	9794567	69842317	4689.471

Table 3.5: Time performance/memory occupancy for a number of molecular surfaces *after* optimization.

chains. Those molecules without chains were acquired as PDB files (<http://www.pdb.org>), while the molecules with chains (we only considered capsids) were obtained as VDB files (<http://viperdbs.scripps.edu>). The last five molecules listed in Table 3.4 (also in Table 3.5) are capsids. Thus, the input of each program is a PDB file (or a VDB file) that describes a specific molecule as needed, for example, to read in the centers of its atoms (cf. the first step of the algorithm in Section 3.4). Table 3.4 shows the time performance for each one of the four implementations (columns 5-8) mentioned above, as well as the total occupied memory space for each molecule (last column), before any optimizing the CUDA kernels, while Table 3.5 lists the results after optimizing the CUDA kernels, as described in Section 3.11.

A preliminary analysis of the results shown in Table 3.5 allows us to observe the following:

- *CPU-based serial program.* As expected, the time performance of this non-multi-threaded CPU program is poor when compared to GPU-based programs, and this is more evident when the number of atoms is high. For example, the molecule 1OHG (cf. penultimate row in Table 3.5) with about 1 million atoms takes 31.5 hours

approximately to get rendered on computer screen. This can be explained by the lack of enough arithmetic logic units (ALU) inside the CPU to perform massive arithmetic calculations.

- *8-core CPU-based parallel program.* As in the previous program, no GPU computations were used in this program either. We only used the OpenMP multi-threading capabilities to take advantage of the 8 cores of the i7 CPU. However, after comparing the time performance results listed in the fifth and sixth columns of Table 3.5, we observe that there is not significant performance gain in using multi-threading, multi-core resources in relation to the serial program. This can be explained by the fact that we are using the same ALUs of the CPU. This means that CPU multi-threading is useful to distribute workload, but not to digest the workload.
- *A 1-GPU-based parallel program.* This program takes advantage of the CUDA GPU multi-threading, but not of the CPU multi-threading. The general-purpose GPU device is an Nvidia Tesla K20, which is used for triangulation-related computations. The Nvidia Quadro K5000 is only used for graphics output of molecular surfaces. This program or setup is capable of triangulating and rendering 1 million molecules like the 1OHG in 7 minutes (or 430 seconds) approximately, nearly exhausting the 5GB memory space of the of Tesla K20 GPU device (cf. Table 3.5).
- *A 2-GPU-based parallel program.* In this case, we use both CPU and GPU multi-threading to take advantage of the multi-core CPU and multi-core GPU streaming multiprocessors of two graphics cards, one Nvidia Tesla K20 and one Nvidia Quadro K5000. That is, both GPU devices work as general-purpose GPUs for computations concerning the MC triangulations, although the Nvidia Quadro K5000 is also used for graphics output. As shown in Fig. 3.1, each GPU device is tied to a single CPU core. This way, it is possible to triangulate and render the above mentioned 1-million molecule (identifier 1OHG) in 2.08 minutes (or 125 seconds) approximately. However, now the 4.44GB of GPU memory needed to triangulate the surface of the 1OHG molecule is divided into two halves, one per GPU device. This means that this setup makes it possible to render molecules up to 2 millions atoms approximately, at least theoretically.

Besides, the performance results concerning timing and memory space occupancy listed in Table 3.5 are depicted in Fig. 3.2 as complexity curves. In fact, the time complexity graphs shown in Fig. 3.2(a) suggest that the time complexity of the 1-GPU setup is super-linear, while that one of the 2-GPU setup is sub-linear. On the other hand, taking into consideration the graphs shown in Fig. 3.2(b), it seems that the memory space complexity is super-linear for both setups in terms of the number of atoms. Six of the biggest molecular surfaces listed in Table 3.5 are shown in Fig. 3.3. and Fig. 3.4.

Note that the most important problem with marching cubes algorithms is the amount of memory needed to triangulate the resulting surface. As shown in Table 3.2(b), more atoms means more occupied memory for vertices and voxels. For example, the bounding

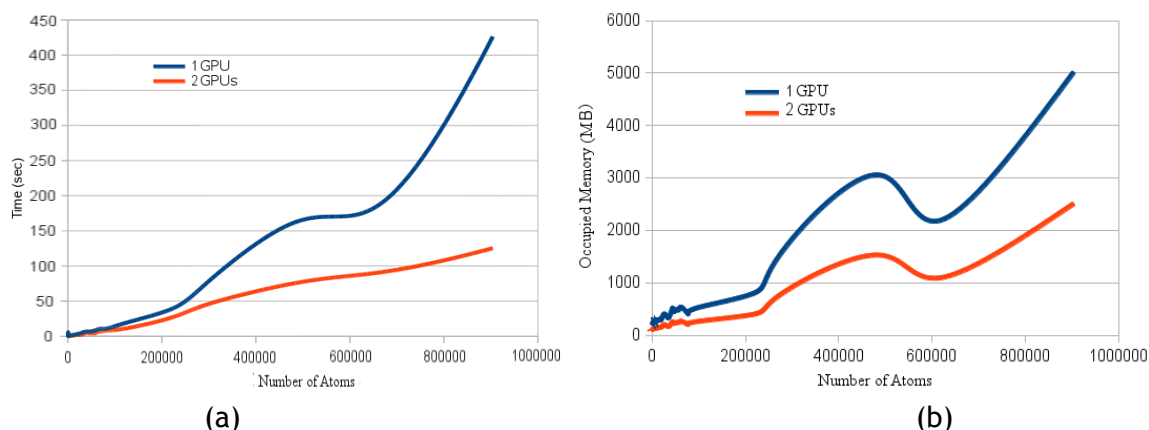


Figure 3.2: Practical complexity of GPU-based programs in terms of the number of atoms: (a) time complexity; (b) memory space complexity.

box enclosing the 1HTQ molecule (last row of Table 3.5) consists of about 70 million voxels. A way of mitigating this problem is using a voxel length Δ that increases with the number of atoms –what we did in practice–, but this issue needs further research. In fact, when using a large value of Δ to triangulate molecules with a small number of atoms (dozens hundreds or even a few thousands), the surface may be not correctly triangulated from the topological point of view. This happens when the Δ is greater than the van der Waals radius of some atoms, what is in conformity with the sampling Nyquist theorem. In fact, sampling a 3D spherical atom is equivalent to sampling two orthogonal 2D circles. Taking into account that a 2D circle can be decomposed into two semicircles, which can be shifted in a way that the endpoint of the top semicircle coincides with the start point of the bottom semicircle. The result is a semicircular wave that looks a sinusoidal wave. Therefore, in order to guarantee the correct triangulation of a smooth Gaussian-like molecular surface, the value of Δ , that indirectly stands for the sampling frequency, must be less than or equal to the van der Waals radius 1.2 Å of hydrogen atom, which is the smallest atom we find in nature. However, and in order to guarantee that the triangulations are smooth, we use the value $\Delta = 0.3$ Å for small molecules and the value $\Delta = 1.2$ Å for large molecules.

3.13 Concluding Remarks

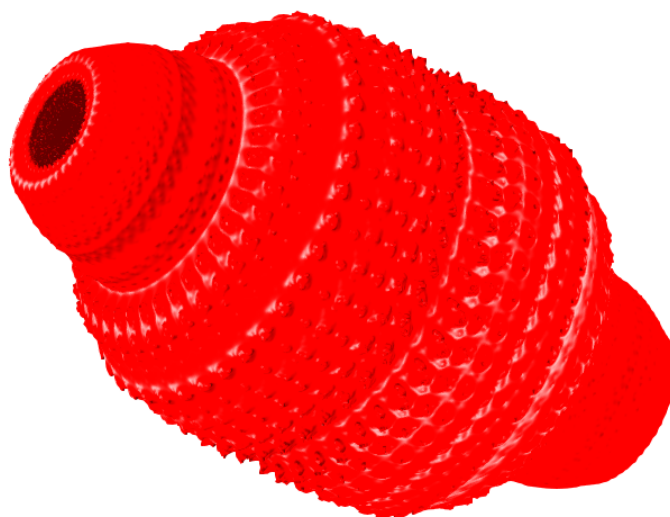
Triangulating and rendering small molecules (i.e., molecules with a small number of atoms) is feasible on commodity computers, but it is not that simple for big molecules having millions of atoms. This chapter has introduced a parallel computing-based marching cubes algorithm that allows for rendering big molecules on a desktop computer equipped with two high-end graphics cards, a Nvidia Quadro K5000 (for triangulation computations and rendering) and a Nvidia Tesla K20 (only for triangulation computations). This makes us to think of being possible to render molecules having 5 to 6 millions atoms on the same computer provided that we had more two Nvidia Tesla K20

graphics cards slotted in mother board. Also, we hope in the near future to design and implement a scalable parallel triangulation algorithm using a cluster of GPUs over a LAN (local area network) in order to process and render very big molecules (i.e. with many millions atoms) in real-time.

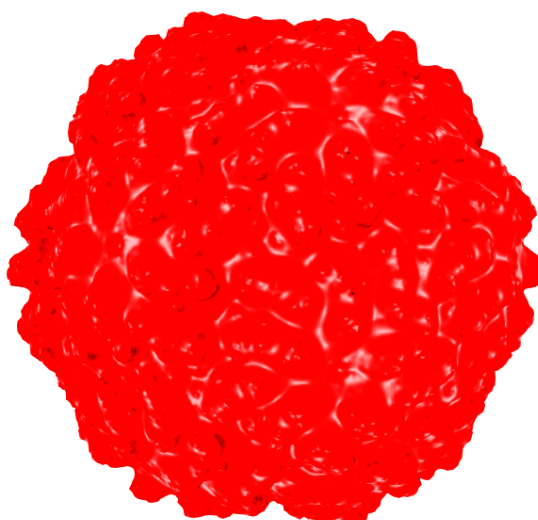
Related Publications

The triangulation algorithm described in this chapter is a follow-up of the algorithm described in the following book chapter:

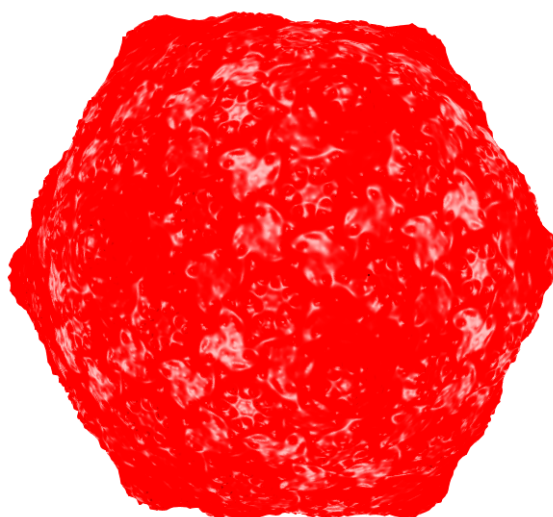
Sérgio Dias and Abel Gomes. Triangulating Gaussian-like Surfaces of Molecules with Millions of Atoms. Chapter in Walter Rocchia and Michela Spagnuolo (eds.), *Computational Electrostatics for Biological Applications*, Elsevier, Chapter 9, Springer-Verlag, pp.177-198, 2015.



(a)

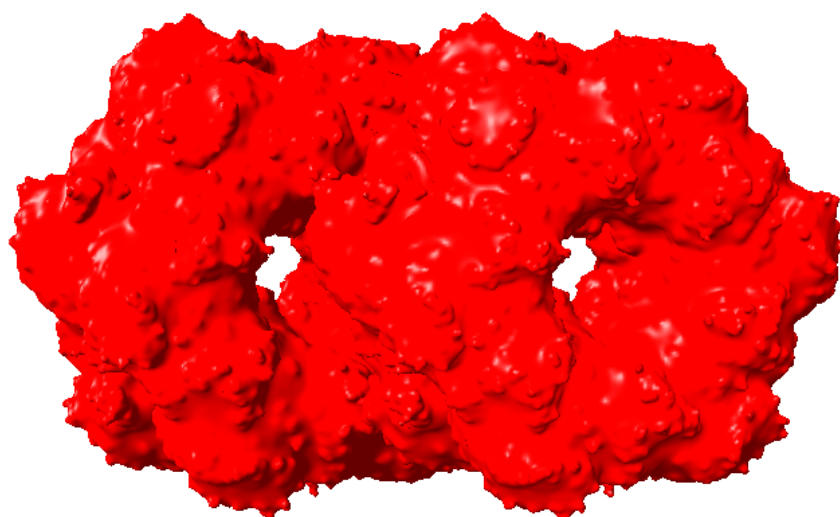


(b)

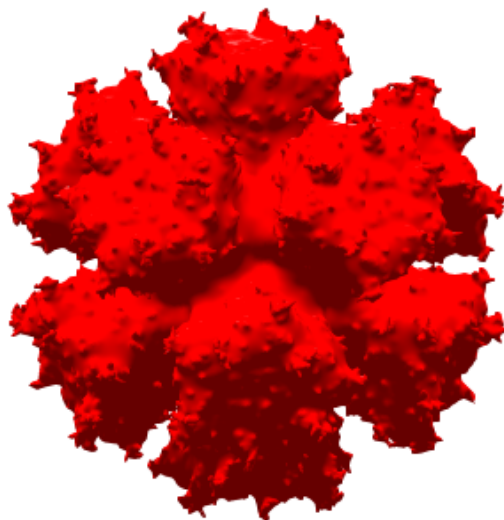


(c)

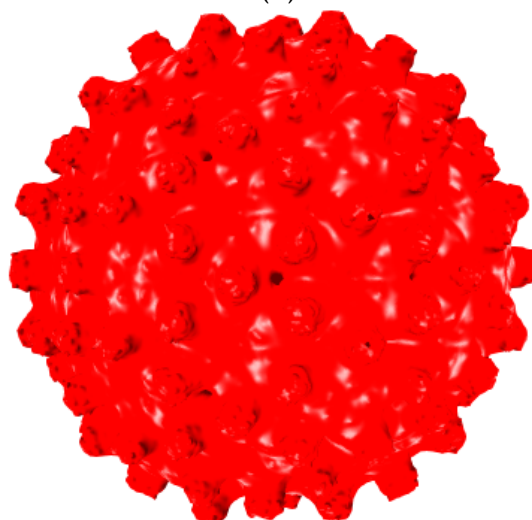
Figure 3.3: Examples of molecular surfaces displayed using GPU-based marching cubes algorithm: (a) PDB id: 39ME; (b) PDB id: 1M1C; (c) PDB id: 1OHG.



(a)



(b)



(c)

Figure 3.4: Examples of molecular surfaces displayed using GPU-based marching cubes algorithm: (d) PDB id: 1HTO, (e) PDB id: 1X9P, (f) PDB id: 2G34.

Chapter 4

GPU-Based Detection of Protein Cavities using Gaussian-like Implicit Surfaces

Cavities on protein surfaces play a key role in bio-molecular recognition and function, particularly in the protein-ligand interactions, as usual in drug discovery and design. Typically, grid-based methods discretize the molecule's domain with a grid of nodes in order to find cavities as aggregates of grid nodes outside the molecule, under the condition that such cavities are bracketed by nodes on the molecule surface along a number of directions (not necessarily aligned with coordinate axes). Therefore, grid-based methods are sensitive to scanning directions, a problem we call cavity ground-and-walls ambiguity. This means that grid-based methods depend on the position and orientation of the protein embedded in the gridded domain. In addition, it is not an easy task to distinguish grid nodes belonging to protein cavities, amongst all grid nodes outside the protein, a problem known as cavity ceiling ambiguity. We solve these ambiguity problem using two implicit isosurfaces of the protein, the protein surface itself (called inner isosurface) that excludes all its interior nodes from any cavity, and the outer isosurface that excludes its exterior nodes from any cavity too. Summing up, the cavities are formed from nodes located between these two isosurfaces.

4.1 Introduction

Proteins participate in many cell processes that are essential to sustain life. However, they cannot live on their own without their constant interactions with other molecules. Proteins interact with other entities in the cell, namely bigger entities like acid nucleic molecules (e.g., DNA) and with smaller entities like nucleotides, peptides, catalytic substrates, and man-made chemicals. Thus, such interactions can possess a number of flavors, namely: protein-ligand, protein-protein, protein-DNA, and so forth. This also means that the binding sites on the surface of a given protein are potentially manifold.

In this chapter, we approach protein-ligand interactions, i.e., interactions between proteins and small molecule ligands. In general terms, we are interested in detecting and estimating the protein sites to where ligands supposedly bind. But, above all, we primarily aim to detect and estimate cavity locations on protein surfaces, because such cavities likely correspond to binding sites. Therefore, protein cavities are here seen as geometric shapes of possible binding sites for ligands. However, as noted in [KMLJ07], shape complementarity is a necessary but not sufficient requirement for ligand binding. The ligand binding to a protein only becomes effective if they enjoy the further property

of physicochemical complementarity, which has to do with, for example, electrostatic, hydrogen-bonding, hydrophobic, and solvent-mediated interactions between the protein and the ligand. This means that not all cavities work as binding sites; conversely, we know that not all binding sites correspond to cavities.

The shape of binding cavities range from small concave invaginations, to deep curved or linear clefts, and hole-shaped regions on a protein surface [CS10] [KG07]. Nevertheless, the size of the binding cavity is not necessarily related to the size of the ligand it binds [KMLJ07] [NH06]. For example, the average size of a drug-binding cavity ranges in [600-900] Å (cf. [SHF⁺06] [KG07] [PSM⁺10]), while the size of a drug-like ligand is about 400 Å. It is worthy noting that drug molecules are often designed to be as small as possible to improve their bioavailability. Moreover, it has been observed that ligands (drugs, in particular) usually bind into the largest and/or deepest concavity on the protein surface [NH06]. On average, such cavity might be three times as large as the ligand, what shows us the difficulty in defining the boundaries of what really is a binding site.

This chapter introduces a novel geometry-based algorithm to identify (or predict) cavities (i.e., tentative binding sites) on the surface of proteins, here called GaussianFinder. Geometry-based algorithms divide into three categories: grid-based, sphere-based, and triangulation-based [LJ06]. GaussianFinder is a grid-based method. Its novelty comes from the fact that it eliminates both ground-and-walls ambiguity and ceiling ambiguity in the building up of cavities as aggregates of grid nodes located outside the protein surface. For that purpose, our method uses two scalar fields of the protein in 3D space, which define two distinct surfaces for the protein, being that the cavities are in between. Additionally, at our best knowledge, this chapter describes one of the first geometric algorithms for detecting protein cavities on GPU. Besides, we used the LigASite dataset of binding sites [DLW08], as well as the corresponding apo and holo proteins, as the ground truth to evaluate the performance and efficiency of GaussianFinder in finding and locating the protein cavities on protein surfaces. In testing GaussianFinder, we also used five benchmark methods (i.e., LIGSITE [HRB97], PASS [BS00], SURFNET [Las95], POCASA [YZTY10], and fpocket [LGST09]) for comparison sake.

The chapter is organized as follows. Section 4.2 briefly approaches the related work existing in the literature. Section 4.3 introduces the scalar field theory behind our algorithm, including the corresponding analytical descriptions for molecular surfaces. Section 4.4 outlines our algorithm. Section 4.5 details how the algorithm has been implemented on GPU. Section 4.6 reviews the GPU-based triangulation of protein surfaces and cavities. Section 4.7 presents the results produced by our method. Section 4.8 concludes the chapter with hints for future work.

4.2 Related Work

The computational algorithms to identify cavities on a molecular surface can be divided in four categories: geometry-based, energy-based, evolutionary-based, combined approaches [VGGR10]. However, in this chapter, we are only interested in geometry-based algorithms. These geometric-based algorithms divide in four sub-categories [KG07]:

- Grid-based
- Sphere-based
- Triangulation-based

Grid-based algorithms map a protein onto an axis-aligned 3D grid, after which we apply a scanning method to label each grid point as occupied or not by the protein, using then some geometric filtering method that outputs the non-occupied grid points (i.e., grid points outside the protein) that are deemed to be cavities. The main problem is then to isolate grid points that belong to cavities from those that do not. Usually, a different grid-point clustering criterion embodies a new geometric method for detecting cavities on the protein surface.

Most of these methods use a visibility criterion, which basically indicates the blocked directions (and non-blocked directions), from any point on the protein surface, being that the protein surface itself plays the role of the occluder. For example, Levitt and Banaszak [LB92] determine which grid points lie in a cavity by checking whether each of them is bracketed by two occupied grid points in either the x, y, or z directions. See Delaney [Del92], Masuya and Doi [MD95], Venkatachalam et al. [VJOW03], Laurie and Jackson [LJ05], Coleman and Sharp [CS06], Weisel et al. [WPS07], Zhang and Bajaj [ZB07], Bock et al. [BGG08], Kalidas and Chandra [KC08], Li et al. [LTA⁺08], La et al. [LERV⁺09], Tripathi and Kellogg [TK10], Krone et al. [KFR⁺11] [KRS⁺13], and Lee and Bargiela [LB12] for further details about different (or similar) visibility geometric criteria to detect cavities through grid-based methods. A significant drawback of grid-based methods is the ambiguity that results from the need of distinguishing between grid nodes (outside the molecule) that belong to cavities from those that do not. Another drawback is that the voxelization (i.e., axis-aligned 3D grid) of the domain entails a lot of memory consumption, even when only the nodes (or corners) of voxels are used, not the voxels themselves.

The leading idea of *sphere-based methods* is to detect cavities (i.e., interior cavities, surface grooves and surface pockets) by fitting probe spheres of different radii into them. If a probe sphere overlaps any atom, the sphere radius decreases in order to ensure that the probe does not overlap with any other atom or any other probe sphere. As Laskowski noted in [Las95], sphere-based algorithms make usage of flood-filling procedure of probe spheres into inner cavities, grooves and pockets. On average, sphere-based methods perform slower than grid-based methods in detecting of cavities, al-

though they consume much less memory because it is not necessary to proceed to the voxelization of the 3D space surrounding the protein. Examples of sphere-based methods are those due to Kuntz et al. [KBO⁺82], Ruppert et al. [RWJ97], Brady and Stouten [BS00], Nayal and Honig[NH06], Kawabata and Go [KG07], Dong and Yang [XZ09], and Yu et al. [YZTY10].

In *triangulation-based methods*, the leading idea is to identify cavity regions from alpha shape methods and Voronoi diagrams. An alpha shape of a molecule is a triangulation that uniquely decomposes the space occupied by its atoms [BAM⁺14], with the advantage of capturing the shape of the molecule itself [EM94]. Depending on the alpha radius, a molecule may have an infinite number of alpha-shapes. Alpha shapes constitute a generalization of the convex hull of a point set [Ede98], which is nothing than an alpha shape of infinite radius. When the alpha radius gradually decreases, the alpha shape decreases and the cavities (i.e., voids, pockets, and depressions) become apparent. Consequently, some simplexes of the initial triangulation end up being empty, denoting the existence of those cavities. Thus, an alpha shape essentially is a surface triangulation that is dual to the molecular surface, in the sense that it possesses the same number of pockets, concavities, and tunnels [EM94]. Examples of algorithms that fall into this category are those due to Peters et al. [PFF96] (APROPOS), Binkowski et al. [BNL03] (CASTp), Xie and Bourne [XB07], Kim et al. [KCC⁺08], Guilloux et al. [LGST09], Tseng et al. [TDCL09], Sridharamurthy et al. [SDP⁺13], and Lindow et al. [LBH11]. Alpha shape methods (and Voronoi-based methods) for prediction of protein cavities are also time-consuming, at least for large molecules.

Summing up, geometry-based methods locate protein cavities from the outer atoms of the protein, in general using a 3D grid-based, sphere-based or tessellation-based techniques. Geometry-based methods comprise the majority of available software; examples include CavitySearch [HM90], POCKET [LB92], fpocket [LGST09], MDPOCKET [SBCLB11], SURFNET [Las95], LIGSITE [HRB97], CAST [LWE98], PASS [BS00], and DogSite [VGGR10]. It is worthy noting that geometry-based methods used in protein cavity detection assume that a cavity is a depression of the protein surface, so it has a specific depth.

In general, the aforementioned geometric algorithms for the search of protein cavities are very demanding in terms of mathematical computations, in particular when the number of atoms goes beyond some thousands. To keep computational run-times within reasonable limits, most algorithms only deal with small molecules. These circumstances get worse when one uses 3D voxelizations of the space surrounding a molecule, as in the case of grid-based methods, since the computational complexity tends to be cubic, unless we take advantage of the parallel computing facilities (e.g., CUDA and OpenCL) of modern graphics cards.

This chapter addresses a distinct method, called GaussianFinder, to identify and calculate protein cavities as clusters of voxels. It combines a grid-based approach with two protein surfaces, called inner and outer surfaces, as an approach to find cavities

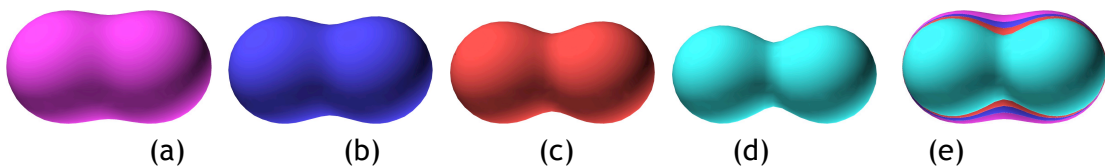


Figure 4.1: Molecular surfaces with the same blobiness $\beta = 0.4$ and distinct isovalues: (a) $c = 1.0$; (b) $c = 1.2$; (c) $c = 1.4$; (d) $c = 1.6$; (e) the previous four molecular surfaces (a)-(d) overlapping.

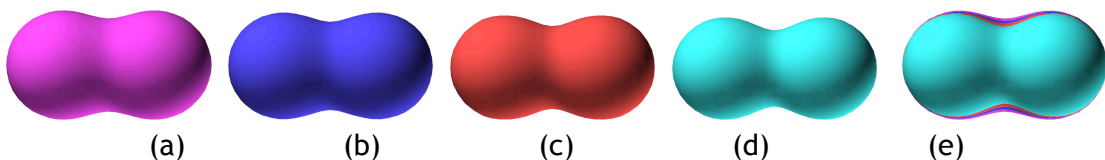


Figure 4.2: Molecular surfaces with the same isovalue $c = 1.0$ and distinct blobiness values: (a) $\beta = 0.34$; (b) $\beta = 0.32$; (c) $\beta = 0.30$; (d) $\beta = 0.28$; (e) the previous four molecular surfaces (a)-(d) overlapping.

as clusters of voxels located between those two surfaces. As shown further ahead, this solves the two ambiguity problems mentioned above, i.e., the problems faced in the delineation of the limits of protein cavities. Unlike other grid-based methods, GaussianFinder does not use any visibility criterion to determine cavities on molecular surface. Furthermore, GaussianFinder likely is one of the first *geometric* cavity detection algorithms to run entirely on GPU. Seemingly, Krone et al. [KFR⁺11] and Parulek et al. [PTRV12] also have used GPU computing to find cavities (as geometric features) on molecular surfaces, though a few GPU-based binding site detection methods have been proposed in the literature, as those due to Sukhwani and Herboldt [SH10], Wang et al. [WYCS13], and McIntosh-Smith et al. [MSPSI14].

4.3 Background

At our best knowledge, this chapter describes one of the first algorithms for detecting cavities on the surface of a protein using a scalar field in a pre-defined domain $D \subset \mathbb{R}^3$ (see also Parulek et al. [PTRV12]). More specifically, D is an axis-aligned bounding box that encloses a given protein. Intuitively, a *scalar field* ties a scalar value to every point in a space. Taking into consideration that the number of points of D is uncountable, we have to discretize D into a number of cubes, here called voxels, so that we only need to calculate the value of the scalar field at each corner of every single voxel. In practice, we only compute the value of the scalar field at the 0-th corner of each voxel, because its remaining seven corners are 0-th corners of its adjacent voxels. It is clear that we are here assuming that the labelling of the corners of all voxels is always done in the same manner.

A scalar field in \mathbb{R}^3 is generated by a real trivariate function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, that is, f is defined at every single point of \mathbb{R}^3 . In this chapter, we use a Gaussian-like function f_i

to model the scalar field generated by each atom i , which is given by

$$f_i(x, y, z) = \frac{\beta}{r_i} \quad (4.1)$$

where $\beta \in]0, 1]$ stands for the smoothness/blobiness parameter and $r_i = (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2$ is the squared distance from the center (x_i, y_i, z_i) , of the atom i to a generic point $(x, y, z) \in \mathbb{R}^3$. Therefore, the kernel of f_i is unbounded in \mathbb{R}^3 , though it decays with the distance to the center of atom i . It is worthy noting that other Gaussian-like functions can be used to model the electron density of each atom, namely those due to McLain [McL74], Blinn [Bli82], Wendland [Wen95], and so forth. The reader is referred to [GVJ⁺09] for further details.

Summing up the scalar fields of all atoms of a molecule results in a scalar field F of the molecule as follows

$$F(x, y, z) = \sum_{i=0}^{n-1} f_i(x, y, z) \quad (4.2)$$

where n is the total number of atoms of the molecule. Note that the Gaussian-like scalar field F allows us to define a family of molecular isosurfaces in \mathbb{R}^3 for the same set of atoms in the following manner:

$$F(x, y, z) - c = 0 \quad (4.3)$$

where $c \in \mathbb{R}$ stands for a varying real isovalue, one per isosurface [ZXB06] [GVJ⁺09]. This set of isosurfaces is also known as level set [OF03]. Note that these molecular isosurfaces are smooth because they result from the summation of smooth Gaussian-like functions associated to their atoms. Besides, each function f_i represents the decaying behavior of the electron density field of each atom with the distance to its center. In Fig. 4.1, we have four isosurfaces of the same molecule (two atoms) generated from 4 distinct isovalues, but with the same blobiness ($\beta = 0.4$). These isosurfaces are overlapping in Fig. 4.1(d) for comparison sake.

Alternatively, we can generate distinct molecular surfaces for the same molecule by varying the blobiness value β in Eq. (4.1), while the isovalue remains unchanged. This is illustrated in Fig. 4.2. As observed, given an isosurface, we can get a more dilated isosurface either by decreasing the isovalue or by increasing the value of blobiness. As explained in the next section, our algorithm uses two isosurfaces. So, we have opted by choosing c -varying isosurfaces instead of β -varying isosurfaces, because this way we have not to recalculate the value of F given by Eq. (4.2) for the second isosurface.

4.4 Cavity Detection Algorithm

The leading idea of our cavity detection algorithm is to use two isosurfaces, called inner and outer surfaces, defined by

$$F_{in} = F - c_{in} = 0 \quad (4.4)$$

and

$$F_{out} = F - c_{out} = 0 \quad (4.5)$$

for the same molecule in the domain $D \in \mathbb{R}^3$, where $c_{in} = 1.0$ and $c_{out} = 1.4$ are the corresponding isovalues, though we use the same blobiness value $\beta = 0.35$ for computing F (cf. Eq. (4.2)). Assuming that the domain D has been previously voxelized, we state that the cavities of a molecule are the connected sets of voxels that belong to the interior of the outer surface, but not to the interior of the inner surface.

Henceforth, we can outline an algorithm to detect the protein cavities as follows:

1. Read atomic centers of a protein from a PDB file.
2. Calculate the axis-aligned bounding box $D \in \mathbb{R}^3$ that encloses the molecule.
3. Discretize D into voxels.
4. Calculate the value of the scalar field F at every single corner of each voxel.
5. Collect the voxels in the interior of outer surface that do not belong to the interior of the inner surface.
6. Separate the collected voxel into clusters of connected voxels.

In the next section, we detail how this algorithm was implemented on GPU using CUDA (Compute Unified Device Architecture) [Coo12]. With the exception of the first step, each step of the algorithm corresponds to a CUDA kernel, in a total of 5 CUDA kernels.

4.5 GPU Implementation

The algorithm outlined above was implemented on GPU using CUDA, and involves the following steps:

Algorithm 1 CavityFormation

Input: H (hash table of cavity voxels)**Output:** C (hash table of cavities)

```
1  $k \leftarrow -1$  ; // initialize index  $k$  for  $C$ 
2  $n \leftarrow \text{sizeof}(H)$  ; // number of voxels belonging to cavities
3  $C \leftarrow \emptyset$  ; // create empty hash table  $C$  of voxel clusters
4 for  $i \leftarrow 0$  to  $n - 1$  do
5   if  $N[26] \geq 0$  for entry  $H(i)$  then
6     continue ; // go to the next voxel  $i$ 
7   end
8    $k \leftarrow k + 1$  ; // define index  $k$  for cavity  $C$ 
9    $N[26] \leftarrow k$  for entry  $H(i)$  ; // assign  $k$ -th cavity  $k$ 
10  Create dynamic array of voxels for  $C(k)$  ; // create voxel array for  $k$ -th cavity
11  Insert key of  $H(i)$  into  $C(k)$  ; // add initial key to voxel array
12   $m \leftarrow \text{sizeof}(C(k))$  ; // current number of voxels in cavity
13  for  $j \leftarrow 0$  to  $m - 1$  do
14    Insert neighbors of voxel  $j$  into  $C(k)$  ; // add neighbor voxels to cavity
15     $m \leftarrow \text{sizeof}(C(k))$  ; // current number of voxels in cavity
16  end
17 end
```

4.5.1 Reading Atomic Centers from a PDB File

The first step of the program is to acquire the atomic centers of a given molecule. For that purpose, we use the PDB file (<http://www.rcsb.org>) that describes the atoms and the structure of a particular molecule. This reading operation is accomplished on CPU side. The array of atomic centers (i.e., triples of coordinates x , y , and z) allocated in memory is later transferred to GPU memory using the usual CUDA function `cudaMemcpy`.

4.5.2 Computation of the Bounding Box

The computation of the bounding box $D \in \mathbb{R}^3$ that encloses the molecule takes place on GPU side. This step involves the computation of both minimum and maximum of the coordinates x , y , and z of atomic centers, that is, the triples $\mathbf{p} = (x_{MIN}, y_{MIN}, z_{MIN})$ and $\mathbf{q} = (x_{MAX}, y_{MAX}, z_{MAX})$. These coordinates are then updated such that $\mathbf{p} = \mathbf{p} - 2r$ and $\mathbf{q} = \mathbf{q} + 2r$, where r is the maximum radius of all atoms belonging to the molecule, in order to guarantee that the molecule stays inside the bounding box.

4.5.3 Voxelization of the Bounding Box

The voxelization of the bounding box D consists in partitioning D into a number of equally sized voxels (i.e., cubes); more specifically, we use cubes with side length $l = 0.3 \text{ \AA}$. This value is one third of the van der Waals radius of the smallest atom, i.e., the hydrogen atom. As explained by Dias and Gomes [DG15], and in conformity with the

Nyquist theorem, this guarantees that the isosurfaces will be, topologically speaking, correctly sampled and reconstructed. Considering that the voxels are all axis-aligned, it thus suffices using only the 0-th corner of each voxel to represent it, because the remaining seven corners of a voxel are 0-th corners of its adjacent voxels. This means that an array of homologous 0-th corners representing the voxels is allocated on GPU side; this array is named V . The location of each 0-th corner is also calculated on the GPU side.

4.5.4 Computation of the Scalar Field F

This kernel launches N (i.e., the size of array V) threads, one per 0-th corner, in order to calculate the value of F (cf. Eq. (4.2)) at each corner in V . These values of F are stored in an N -sized GPU array, also called F . It is clear that, before running this CUDA kernel on GPU, it is first necessary to allocate memory for the array F on GPU. As mentioned above, in the computation of the function F at each corner, we used the blobiness value of $\beta = 0.35$.

4.5.5 Collecting Cavity Voxels

Cavity voxels are those located between the inner and outer surfaces. That is, our voxels of interest are those whose corners satisfy the following conditions: $F > c_{in}$ and $F < c_{out}$. Recall that each voxel is represented by its 0-th corner or vertex in the array V .

These cavity voxels are collected from the array of vertices V into the hash table H on GPU, which consists of a series of pairs $\langle id, N[i] \rangle$, where id stands for the identifier of each cavity voxel, and $N[i]$ ($i = 0, \dots, 26$) is an array of 27 integer identifiers; the first 26 identifiers concern the 26 voxels that are adjacent to the voxel id , being the 27th identifier used as the identifier of the cluster to which the voxel id belongs. Initially, the elements of $N[i]$ are set to -1. Note that we use the Thrust library (<http://thrust.github.io/>) to deal with the hash table H on GPU.

4.5.6 Formation of Cavities

Having found the cavity voxels, it is necessary to partition this set H of voxels into subsets C_i ($i = 0, \dots, k$), where each subset of voxels represents a distinct cavity. This is done using a *voxel growing* technique.

The voxel growing procedure described in Alg. 1 starts with the retrieval of an arbitrary voxel from the original set of voxels H , appending than it to the first empty subset of voxels C_0 . This first voxel is the seed voxel of the first cavity C_0 . The cavity C_0 then grows from its seed voxel to adjacent voxels. Note that, a voxel has a 26-connected neighborhood; so, all cavity voxels in this neighborhood are appended to C_0 . Then, we

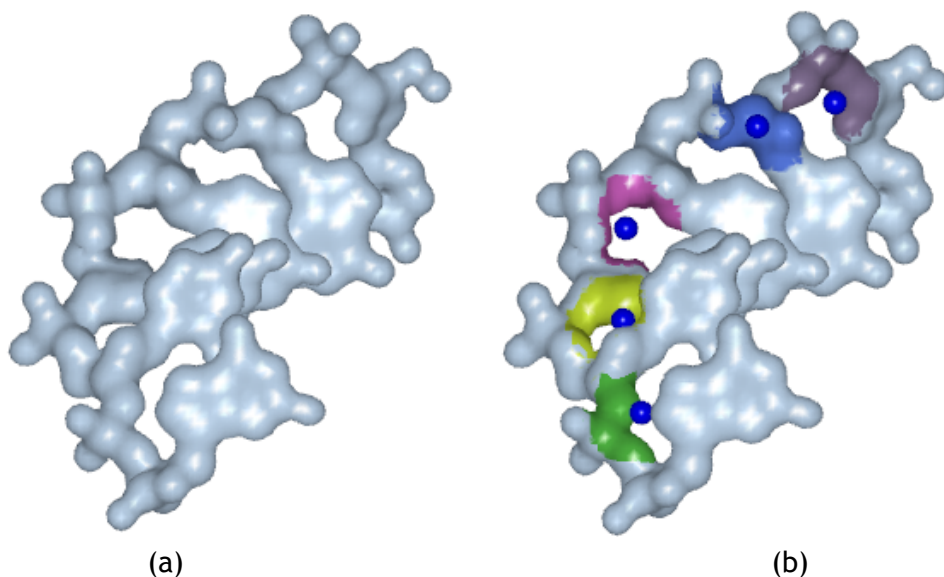


Figure 4.3: Molecular visualization of the 110D molecule: (a) cavities are not depicted on the surface; (b) cavities are depicted on the surface as determined by our algorithm. The small spheres in blue indicate the location of cavities as calculated by MetaPocket [Hua09].

repeat the appending operation of neighboring voxels for each voxel of C_0 in a manner that there is no duplicates of voxels in C_0 . C_0 stops growing when there is no more voxels in H to append to it. This procedure is repeated while there is some remaining voxel in H , i.e., while there another cavity C_k to build up.

In terms of GPU programming, the set $C = \{C_k\}$ of cavities is encoded as another Thrust hash table, being k the key used to access the Thrust dynamic array of voxels associated to a cavity C_k . Each dynamic array works as a cluster of voxels that belong to the same cavity. This voxel array is dynamic because of the growing nature of the formation of the cavity.

In fact, each dynamic array featuring a cavity is generated on-the-fly while we iterate on the tuples of the hash table H (cf. line 11 in Alg. 1). The inner 'for' loop (lines 13-16) of Alg. 1 builds up the voxel cluster of each cavity k , being the neighbors of each voxel inserted into C_k without repetitions (line 14). It is clear that after adding neighbors to C_k , we have to recalculate the size of C_k (line 15).

4.6 Molecular Triangulation

The graphics visualization of each protein requires the triangulation of the Gaussian-like molecular surface defined by $F_{in}(x, y, z) - c = 0$. This triangulation is carried out entirely on GPU side using the variant of the marching cubes algorithm introduced by Dias and Gomes [DBG10], as illustrated in Fig. 4.3(a). Fig. 4.3(b) shows the cavities on the surface of the 110D molecule depicted in distinct colors, and where the small balls in blue represent the locations of those cavities as determined by MetaPocket 2.0 [Hua09]. In this case, we see that there is a match between the locations of cavities

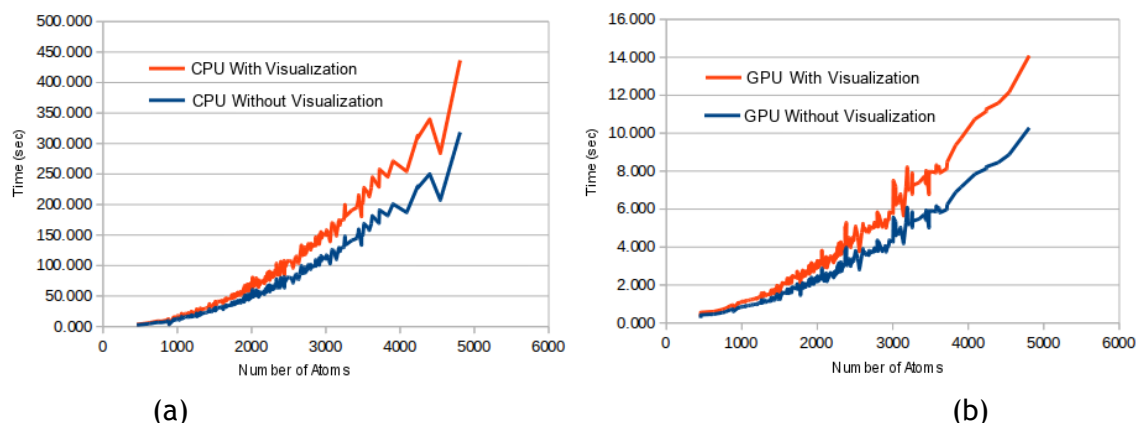


Figure 4.4: (a) Time performance of CPU algorithm with (in red) and without (in blue) surface triangulation; (b) Time performance of GPU algorithm with (in red) and without (in blue) surface triangulation.

calculated by our algorithm and those determined by MetaPocket package.

The envelope of each cavity can be also defined as the Gaussian-like surface that encloses the corresponding subset of voxels. For this purpose, we place imaginary atoms of radius $r = 1.25l$ at the centers of the voxels that are located at the frontier of the cavity, where l stands for the voxel length. Note that a frontier voxel of a cavity satisfies the condition $N[i] = -1$ for at least one of its neighbors in the cavity, whereas an interior voxel satisfies the condition $N[i] \geq 0$ for all of its neighbors.

4.7 Results

4.7.1 Hardware/Software Setup

Testing were accomplished using a desktop computer running the Linux Fedora 20 operating system, and equipped with a Intel-Core I7-4820k 3.70 Ghz Processor, 32GB RAM, 1 Nvidia Tesla K20, and 1 Nvidia Quadro K5000. Most computations to detect cavities of proteins and other molecules took place on a Nvidia Tesla K20. Also, all the computations needed to triangulate surfaces of molecules and their cavities were performed on the same Nvidia Tesla K20. The Nvidia Quadro K5000 was only used for graphics output.

4.7.2 Time Performance

In order to evaluate the time performance of our algorithm, we proceeded as follows:

- *Molecule Dataset.* We used the dataset of 2604 proteins taken from the LigASite database (<http://ligasite.org/>), where 816 are APO structures, and 1788 HOLO structures [DLW08]. The atomic structure of each one of these molecules was

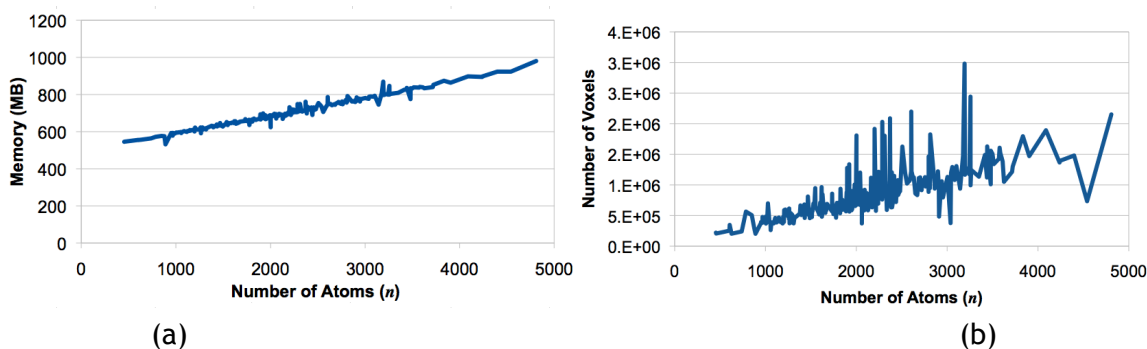


Figure 4.5: (a) Memory space occupancy with respect to the number of atoms (n); (b) the number of voxels in memory with respect to the number of atoms (n).

retrieved from PDB Data Bank as a PDB file. The molecule with the biggest number of atoms is the LUF4 molecule, which has 4808 atoms.

- *Multi-threaded CPU Program.* We used a CPU-based implementation of the algorithm taking advantage of multi-threading of 8 CPU cores. This program was written in C/C++ using the Standard Template Library (STL). For the sake of rendering of molecules and their cavities, we also used a multi-threaded CPU-based implementation of the marching cubes algorithm.
- *Parallel GPU Program.* The GPU-based implementation of our algorithm was written in C/C++ together with CUDA. We also used the Thrust library for manipulation of hash tables and dynamic arrays on GPU side. Thrust can be considered as the counterpart library of the C++ Standard Template Library (STL) for GPUs, which is available at <http://thrust.github.io/>. Triangulating and rendering surfaces of molecules and cavities on GPU were performed using the GPU-based implementation of the marching cubes algorithm by Dias and Gomes [DBG10].

The time performance of our cavity detection algorithm on CPU and GPU is shown in Fig. 4.4(a) and (b), respectively, in blue. The overall time performance of cavity detection algorithm plus surface triangulation algorithm is shown in red. It is not surprising that GPU-based implementation performs better than CPU-based implementation. From the analysis of the data graphically represented in Fig. 4.4, we observed the following:

- *CPU-based detection of cavities* (Fig. 4.4(a) in blue). The CPU-based algorithm for identifying cavities has a time performance given by the expression

$$t = 8.26 \times 10^{-6} \times n^{2.04} \quad (4.6)$$

- *CPU-based detection of cavities plus molecular surface triangulation* (Fig. 4.4(a) in red). The previous algorithm together with the triangulation algorithm has the

time performance given by

$$t = 9.74 \times 10^{-6} \times n^{2.06} \quad (4.7)$$

- *GPU-based detection of cavities* (Fig. 4.4(b) in blue). The time performance of the GPU-based algorithm for identifying cavities satisfies the following expression

$$t = 1.89 \times 10^{-5} \times n^{1.542} \quad (4.8)$$

- *GPU-based detection of cavities plus molecular surface triangulation* (Fig. 4.4(b) in red). The time performance of the previous algorithm together with the triangulation algorithm satisfies the following expression

$$t = 1.99 \times 10^{-5} \times n^{1.57} \quad (4.9)$$

Device Type	Tesla K20
Memory Size (GB)	5.0
Streaming Multiprocessors (SM)	13
CUDA Cores	2,496
Maximum Number of Active Blocks per SM	16
Maximum Number of Active Warps per SM	64
Maximum Number of Active Threads per SM	2,048
Maximum Number of Threads per Block	1,024
Maximum Number of Warps per Block	32
Maximum Number of Registers per Thread	255
Maximum Number of Registers per Block	65,536

Table 4.1: Kepler GPU's memory and compute capabilities.

# kernel	Active Blocks	Active Warps	Active Threads	Threads per Block	Warps per Block	Registers per Thread	Registers per Block	Multiprocessor Occupancy
1st kernel	16	32	1024	64	2	124	589	50%
2nd kernel	12	12	768	64	2	110	660	38%
3rd kernel	16	8	512	64	2	100	768	25%
4th kernel	6	16	384	64	2	150	545	19%
5th kernel	4	8	256	64	2	185	458	13%

Table 4.2: Performance data *before* optimizing the CUDA kernels.

# kernel	Active Blocks	Active Warps	Active Threads	Threads per Block	Warps per Block	Registers per Thread	Registers per Block	Multiprocessor Occupancy
1st kernel	10	60	1920	192	6	30	4510	94%
2nd kernel	10	60	1920	192	6	33	4665	94%
3rd kernel	10	60	1920	192	6	25	3978	94%
4th kernel	8	48	1536	192	6	40	3214	75%
5th kernel	8	40	1287	192	6	45	2786	63%

Table 4.3: Performance data *after* optimizing the CUDA kernels.

Eqs. (4.6)-(4.9) were obtained by curve fitting [Arl94]. Thus, the experimental time complexity of our method is quadratic on CPU (cf. Eqs. (4.6)-(4.7)), while it is super-linear on GPU (cf. Eqs. (4.8)-(4.9)). For example, finding the cavities of a molecule with 3000 atoms takes 102.41 seconds approximately on CPU, whereas it takes about 4.35 seconds on GPU, i.e., a speedup of 23.54x. When the triangulation of the molecular surface is also taken into account in the time counting, it takes 141.72 seconds on CPU and about 5.73 seconds on GPU, therefrom results a speedup of 24.73x.

For the entire set of molecules, the multi-threaded CPU-based solution (without triangulation) takes 163,020 seconds (about 45 hours), while the GPU-based solution (without triangulation) takes 10,800 seconds (about 3 hours) to determine all the cavities of all molecules altogether, resulting in an average speedup of 15x. Taking into consideration the extra time needed to triangulate and rendering of all molecules, those times increase up to 222,840 (about 62 hours) and 14,400 seconds (about 4 hours) on CPU and GPU, respectively, what features an average speedup of 16x. These times are end-to-end runtimes, i.e., times needed to run the 6 steps (7 if triangulation is included) of the algorithm outlined in the end of Section 4.4.

4.7.3 Memory Space Complexity

As expected, the algorithm consumes a lot a GPU memory in order to allocate space for voxels, which are usually in great number. For example, as shown in Fig. 4.5(b), there is a molecule in the dataset that requires the partitioning of its bounding box into 3 million voxels, even knowing that the biggest molecule of the dataset has less than 5,000 atoms. Interestingly, as shown in Fig. 4.5(a), the memory space complexity of the algorithm is essentially linear in respect to the number of atoms.

4.7.4 CUDA Code Optimization

As noted above, we only used a Nvidia Tesla K20 for running our cavity detection code on GPU. The Nvidia Quadro K5000 was exclusively used for rendering and visualization purposes. This means that the code optimization only took place on Tesla K20, whose memory and compute capabilities are listed in Table 4.1. This device supports up to 16 active blocks, 64 warps, 2048 threads scheduled simultaneously per multiprocessor, and 192 cores per multiprocessor ($2,496/13=192$) at maximum. Recall that a GPU kernel runs as a grid of blocks of warps of threads; more specifically, a block is composed by 32 warps, and a warp consists of 32 threads, i.e., 1024 threads per block at maximum (cf. Table 4.1).

The optimization of the kernels must take into consideration the architecture, type, and capabilities of GPU device. In the case of a Nvidia Tesla K20, the compute capability is known to be 3.5, which must be entered after the flag `-arch` when compiling CUDA code (kernels) through `nvcc`. We can then run the code on CPU side, being the kernels

Δ_i	LIGSITE		PASS		SURFNET		POCASA		fpocket		Mean Average	
	n_i	h_i	n_i	h_i	n_i	h_i	n_i	h_i	n_i	h_i	n_i	h_i
-4	37	41	53	61	73	84	96	117	108	127	73	86
-3	51	64	67	75	147	159	104	121	133	151	100	114
-2	126	137	178	196	188	204	201	247	234	248	185	206
-1	187	201	222	244	234	254	261	343	368	381	254	284
0	1776	1683	1577	1478	1347	1179	1244	925	859	743	1360	1201
1	196	214	187	201	215	251	283	359	356	377	247	280
2	124	135	175	184	167	213	195	220	255	262	183	202
3	76	81	84	93	137	167	137	165	164	176	119	136
4	31	48	61	72	96	93	83	107	127	139	79	91

Table 4.4: Hit performance (n_i and h_i) of GaussianFinder with respect to LIGSITE, PASS, SURFNET, POCASA, and fpocket.

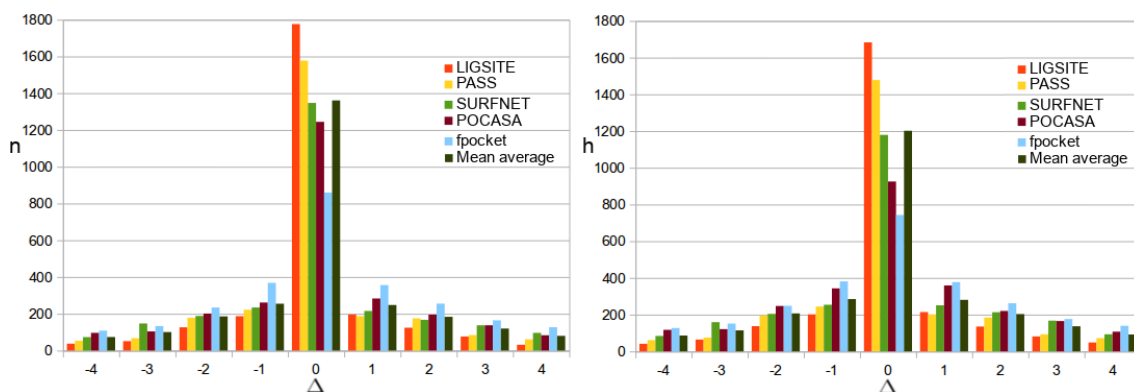


Figure 4.6: Histograms of hit performance of GaussianFinder in relation to LIGSITE, PASS, SURFNET, POCASA, and fpocket: (left) numerical hit performance n_i in function of the dispersion of the number of cavities Δ_i ; (right) positional hit performance h_i in function of the dispersion of the number of cavities Δ_i .

dispatched one after another to GPU side for execution.

Before optimizing kernel codes, we need to make their performance analysis using Nsight (<http://www.nvidia.com/object/nsight.html>). Table 4.2 shows the performance data measured by this Nvidia profiler before code optimization. In this preliminary analysis, we used three representative molecules with a varying number of atoms. This allowed us to optimize the CUDA kernel codes, according to the figures shown in Table 4.3.

The code optimization of each CUDA kernel has to do with its streaming multiprocessor occupancy, i.e., full occupancy means optimal performance of streaming multiprocessors in terms of efficiency and speed of the computations on CUDA device. Specifically, streaming multiprocessor occupancy can be defined as the ratio of the number of running threads to the maximum number of threads that are admissible to run on the GPU.

The optimization steps are as follows: (i) set the number of threads per block; (ii) increasing of the number of active warps; (iii) decreasing of the number of registers per thread; (iv) fixing unbalanced workloads. The first optimization step is the most important one, and consists in setting the number of threads per block for each CUDA kernel. According to Nsight, the number of threads per block should be 192, though

the maximum number of threads per block is 1024(cf. Table 4.1). So, the number of threads per block was changed from 160 (cf. Table 4.2) to 192 (cf. Table 4.3) in our CUDA program. Noticeably, as shown in Table 4.3, the multiprocessor occupancy increased for all kernels, in particular for the first three kernels. The last two kernels are not so efficient in terms of multiprocessor occupancy because the Thurst code we used for GPU hash tables and dynamic arrays is not so optimized as desirable. Recall that the thread block size must be a multiple of 32 in the range [64,256], being that this value is tied to the graphics card.

The second optimization step involves increasing the number of active warps. This is not done via Nsight. Instead, this optimization step has to do with changing our programming style. Increasing the number of active warps means reducing the number of 'if-else', 'switch', 'for', and 'while' statements to a minimum in the CUDA kernels. Doing so, we ended up having 6 warps per block (cf. sixth column in Table 4.3), instead of 2 warps per block (cf. sixth column in Table 4.2), with a significant increase of active warps (cf. third column in Tables 4.2 and 4.3).

The third optimization step consists in decreasing the number of registers per thread. This is not done via Nsight either. To achieve this goal for each kernel, it suffices to decrease its number of mathematical operations. Additionally, many of these operations can be replaced by CUDA mathematical operators, which are already optimized in terms of the minimal number of registers per thread. For example, we optimized the code of the first kernel in this manner, with 124 registers per thread before the optimization, and 30 after the optimization, as shown in seventh column of Tables 4.2 and 4.3.

The fourth optimization step refers to fix the effect of unbalanced workloads. In order to overcome this problem, we adopted the solution that consists in using a number of sub-kernels for each kernel. In the particular case of our algorithm, each kernel is a set of 20 sub-kernels, which are essentially identical in terms of CUDA code. The difference lies in the voxelized slices of the bounding box they operate on. This forces the distribution of the workload, preventing that the final result from being produced on a single block, warp, or thread.

4.7.5 Comparison with other Geometric Algorithms

In order to assess the efficiency and reliability of our cavity detection algorithm (i.e., GaussianFinder), we used the LigASite database of binding sites [DLW08], as well as five benchmarking cavity detection algorithms bundled with Metapocket [Hua09].

4.7.5.1 Ground-Truth Dataset of Cavities

We used LigASite as the ground-truth dataset of binding sites of proteins [DLW08]. LigASite consists of a dataset of 2604 proteins and their binding sites. More specifically,

LigASite includes 816 apo structures and 1788 holo structures of binding sites. An apo protein is a protein without ligands, while an holo protein is a protein-ligand complex. The corresponding PDB files were retrieved from PDB Data bank (www.rcsb.org). Note that the LigASite dataset is particularly adequate to benchmark cavity prediction methods because it consists of proteins with one unbound structure (i.e., without ligands); in addition, it also consists of at least one bound structure (i.e., with ligands) for each unbound structure, from which we easily come up with known binding sites, in order to make it possible to consider the structural changes of a protein upon binding of its ligands [DLW08].

4.7.5.2 Benchmarking Methods

We compared GaussianFinder with the following *geometric* algorithms wrapped in Metapocket [Hua09]:

- LIGSITE. It includes the grid-based method introduced by Hendlich et al. [HRB97].
- POCASA. It is essentially a grid-based method, called Roll, though it also uses a crust-like surface of probe spheres (see Yu et al. [YZTY10]).
- SURFNET. It includes the sphere-based method proposed by Laskowski [Las95].
- PASS. It includes the sphere-based method proposed by Brady et al. [BS00].
- fpocket. It includes a triangulation-based method based on a Voronoi tessellation and alpha spheres on the top of a convex hull algorithm (see Guilloux et al. [LGST09]).

The testing results are shown in Table 4.4, where $\Delta_i = C_i - c_i$ is the difference between the number C_i of cavities detected by GaussianFinder and the number c_i of cavities detected by each of those five methods.

4.7.5.3 Benchmarking Metrics

In our testing, we used the following metrics: numerical hit weighted deviation (σ), positional hit weighted deviation (δ), and cumulative cavity percentage (P).

Numerical Hit Weighted Deviation. The parameter n_i in Table 4.4 quantifies the numerical hits (i.e., number of proteins) for Δ_i i.e., number of proteins for which the difference between the number of cavities calculated by GaussianFilter and the number of cavities calculated by another method is given by Δ_i .

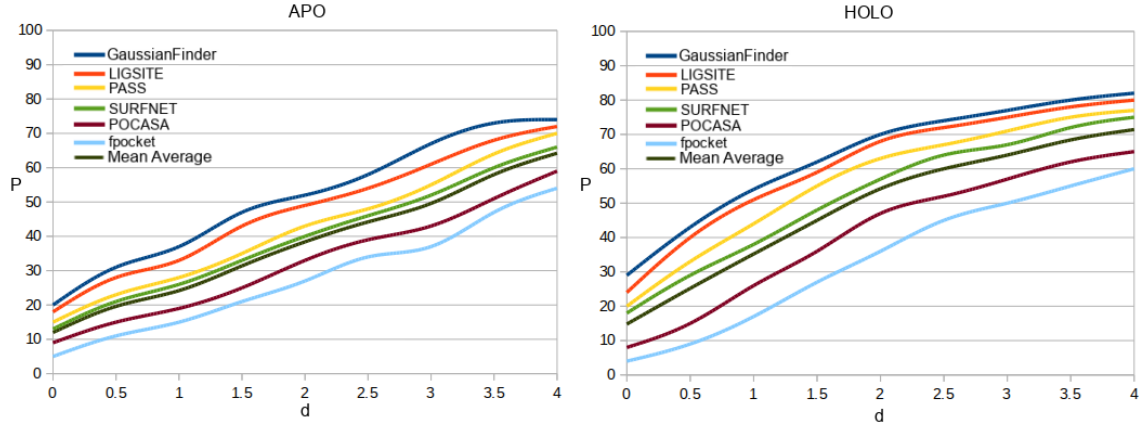


Figure 4.7: Cumulative cavity percentage of various detection methods in function of the distance d to ground-truth geometric centers: GaussianFinder, LIGSITE, PASS, SURFNET, POCASA, and fpocket for apo (left) and holo (right) structures.

By definition, the numerical hit weighted deviation is given by the following formula:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{k-1} n_i \Delta_i^2}{N}} \quad (4.10)$$

where k stands for the number of bins i , and $N = 2606$ is the number of proteins of the LigASite dataset. Thus, σ measures the numerical hit deviation of GaussianFinder in relation to each of those five methods listed in Table 4.4 as follows:

- LIGSITE: $\sigma = 1.384$, concerning hits in the range $\Delta_i \in [-1, 1]$, making a total of 2159 hits, that is, 82.91% of hits.
- PASS: $\sigma = 1.396$, which gathers hits in the range $\Delta_i \in [-1, 1]$, totalizing 1986 hits, that is, 76.27% of hits.
- SURFNET: $\sigma = 1.415$, comprising hits in the range $\Delta_i \in [-1, 1]$, making a total of 1796 hits, that is, 68.97% of hits.
- POCASA: $\sigma = 1.457$, gathering hits in the range $\Delta_i \in [-1, 1]$, totalizing 1788 hits, that is, 68.66% of hits.
- fpocket: $\sigma = 1.527$, covering hits in the range $\Delta_i \in [-1, 1]$, making a total of 1583 hits, that is, 60.79% of hits.

As illustrated in Fig. 4.6(left), these results show that the absolute maximum dispersion is $|\Delta| = 4$ in terms of the number of cavities found by GaussianFinder in respect to other five algorithms. For $|\Delta| = 1$, our method and LIGSITE match in 82.91% of hits, i.e., they compute the same number of cavities in 82.91% of the proteins; GaussianFinder and PASS match in 76.27% of proteins; GaussianFinder and SURFNET match in 68.97% of proteins; GaussianFinder and POCASA match in 68.66% of proteins; and, finally, Gaus-

sianFinder and fpocket find the same number of cavities for 60.79% of proteins.

Positional Hit Weighted Deviation. For comparison sake, we assumed that we have a positional hit (for a protein) exists if the cavity geometric centers of a protein found by the GaussianFinder are all within 4 Å in relatively to geometric centers calculated by each of those five methods. In this case, positional hit weighted deviation is calculated as follows:

$$\delta = \sqrt{\frac{\sum_{i=0}^{k-1} h_i \Delta_i^2}{N}} \quad (4.11)$$

where k denotes the number of bins i , h_i stands for the number of positional hits, and Δ_i the difference between the number of cavities found by GaussianFinder and any other detection method. From the values listed in Table 4.4 (cf. also Fig. 4.6(right)), we obtained the following:

- LIGSITE: $\delta = 1.274$, concerning hits in the range $\Delta_i \in [-1, 1]$, making a total of 2098 hits, that is, 80.56% of hits.
- PASS: $\delta = 1.467$, which gathers hits in the range $\Delta_i \in [-1, 1]$, totalizing 1923 hits, that is, 73.84% of hits.
- SURFNET: $\delta = 1.746$, comprising hits in the range $\Delta_i \in [-1, 1]$, making a total of 1684 hits, that is, 64.66% of hits.
- POCASA: $\delta = 1.830$, gathering hits in the range $\Delta_i \in [-1, 1]$, totalizing 1627 hits, that is, 62.48% of hits.
- fpocket: $\delta = 1.959$, covering hits in the range $\Delta_i \in [-1, 1]$, making a total of 1501 hits, that is, 57.64% of hits.

This means that, for example, GaussianFinder and LIGSITE match in 80.56% of hits for the absolute hit dispersion of $|\Delta| = 1$; that is, they compute the same cavities on the same locations (within 4 Å maximum) in 80.56% of the proteins, being that the number of proteins varies in 1 maximum. For the same absolute hit dispersion of 1, we have a match of 73.84% for PASS, 64.66% for SURFNET, 62.48% for POCASA, and 57.64% for fpocket.

Cumulative Cavity Percentage. In the computation of the positional hit weighted deviation (δ), we intended to know how GaussianFilter performed in identifying cavity locations in relation to five benchmark detection methods. This means that the geometric centers of cavities provided by those benchmark methods were used as the

ground truth to ascertain the efficiency of GaussianFinder in relation to such benchmark methods.

Now, we intend to assess the efficiency of GaussianFinder and benchmark methods in relation to the ground truth given by the actual location of the geometric centers of protein cavities. Every ground-truth geometric center of a cavity is previously calculated as the barycenter of the centers of the atoms associated to such a cavity, as expressed in the apo or holo structure of the corresponding binding site. We use the apo and holo structures of the protein binding sites that are part of LigASite dataset. These apo and holo structures are organized in separate PDB files, and work as benchmark structures for cavities on the surface of proteins. Recall that LigASite hosts 816 apo structures and 1788 holo structures of binding sites (i.e., cavities, in geometric terms), in a total of 2604 structures.

In order to determine the cumulative cavity percentage $C \in [0.0, 100.0]$ for each benchmark method, including GaussianFinder, we first have to know the geometric center of each protein cavity, as calculated by each benchmark method. Then, for each benchmark method, we determine the distance $d \in [0.0, 4.0]$ Å from the geometric center of each protein cavity to its ground-truth geometric center.

The results as a whole are shown in Fig. 4.7. Obviously, cavities located beyond $d = 4.0$ Å in relation to their ground-truth geometric centers are not accounted for, but they are only a few anyway. In respect to the 816 apo proteins, we obtained the following cumulative cavity percentages P in the range $d \in [0, 4]$: GaussianFinder (74%), LIGSITE (72%), PASS (70%), SURFNET (66%), POCASA (59%), and fpocket (54%). On the set of 1788 holo proteins (i.e., protein-ligand complexes), we ended up obtaining the following percentages: GaussianFinder (82%), LIGSITE (80%), PASS (77%), SURFNET (75%), POCASA (65%), and fpocket (60%). Fig. 4.8 shows three examples of molecules exhibiting their cavities as found through GaussianFinder (left) and LIGSITE (right).

Taking into consideration the results above, we can draw the following conclusions:

- GaussianFinder ranks first as a cavity detection method.
- Grid-based methods (GaussianFinder, LIGSITE, and PASS) seemingly perform better than sphere-based methods (SURFNET and POCASA), which in turn present better results than triangulation-based methods (fpocket).
- The benchmarking geometric methods above tend to find the same number of cavities (tentative binding sites) on the same sites on the surface of each protein.
- As expected, either benchmark method performs better for apo proteins than for holo proteins, simply because the number of cavities on apo proteins usually is higher than on holo proteins.

Note that we only considered benchmark geometric methods for the detection of cavities; hence, we focused on cavities, rather than on binding sites. However, we used

actual locations of binding sites as ground-truth for cavity locations.

4.8 Concluding Remarks

We have introduced a novel grid-based algorithm for identification of cavities on protein surfaces without using a visibility criterion. The leading idea of the method is to determine the grid nodes between those two Gaussian-like isosurfaces of each molecule, which are then aggregated into clusters of nodes featuring cavities. This avoids possible geometric ambiguities (concerning the limits of cavities) inherent to the use of grid-based methods to detect cavities of protein surface. Besides, as far as we know, this is the first geometric CUDA-based algorithm to detect cavities of proteins, though there are a few others that take advantage of shader programming on GPU. It is considerably fast, with the cavity detection stage completing in a matter of a few seconds on a GPU-based workstation equipped with a Nvidia Tesla K20 and a Nvidia Quadro K5000. In the near future, we intend to parallelize the cavity detection algorithms existing in the literature as a way to have a testbed to a more accurate comparison between algorithms in terms of performance.

Related Publications

The algorithm described in this chapter is a follow-up of the algorithm described in the following book chapter:

Sérgio Dias and Abel Gomes. GPU-Based Detection of Protein Cavities using Gaussian-like Implicit Surfaces. IEEE/ACM Transactions on Computational Biology and Bioinformatics (under 2nd revision).

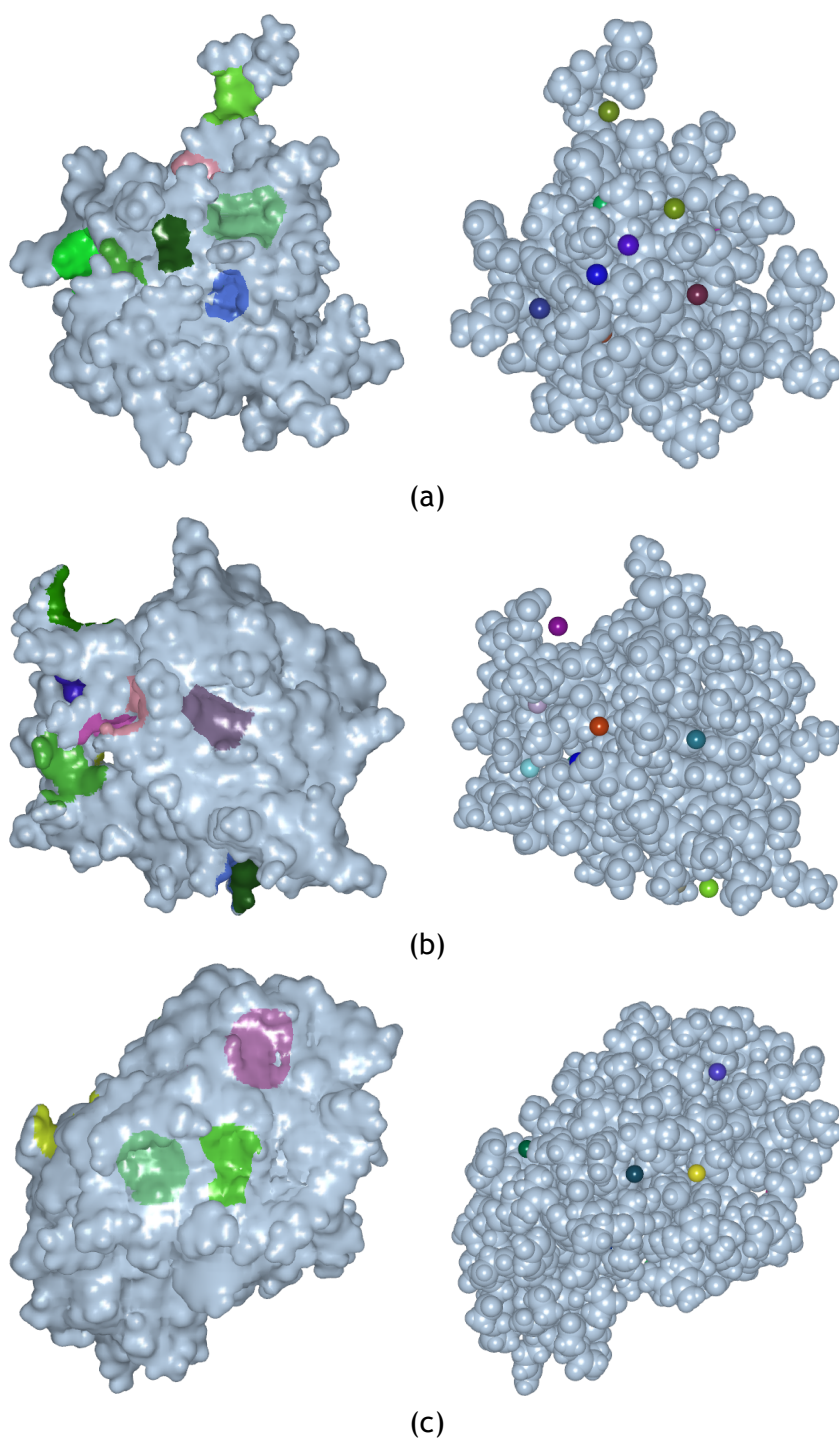


Figure 4.8: Examples of 3 proteins and their cavities as detected by GaussianFinder (left) and LIGSITE (right): (a) PDB id: 1NEQ (13 cavities); (b) PDB id: 1SVR (10 cavities); (c) PDB id: 2NWD (15 cavities).

Chapter 5

A Curvature-Based Cavity Detection Method

Molecular cavities are specific regions on the surface of a biomolecule (e.g., a protein) where another molecule (ligand) may bind. Usually, cavities correspond to voids, pockets, and depressions of the molecular surfaces. The location of such cavities is important to better understand protein functions, as needed in, for example, structure-based drug design. This chapter introduces a Morse theory-based method to detect cavities on the molecular surface. The method, called CriticalFinder, is different from other curvature-based methods found in the literature because it does not evaluate the curvature *on*, but *outside*, the surface of the molecule. In fact, what is really evaluated is the curvature of the electron density field (generated by the atoms) outside the molecule. To evaluate the performance of CriticalFinder, a comparative study with other five geometric methods (i.e., LIGSITE, PASS, SURFNET, POCASA, and fpocket) was carried out in order to better evaluate the success rate of the CriticalFinder method. In addition, it was also carried out a time performance analysis to both CPU/GPU implementations.

5.1 Introduction

Many biological processes in life sciences, in particular those involving drug interactions and protein docking, occur in water (solute). The interaction between water and a molecule can tell much information about the shape of a molecule, including the localization of its binding sites. As [Mez93] noted, this is of great importance to research in chemistry, biophysics, medicine, and nano-technology. A better interpretation and identification of such binding sites on a molecular surface can even enhance our insight in respect to the discovery process of new drugs. Hence, the identification of those binding sites is often the first step to study protein functions, as in the structure-based drug design.

However, many sorts of small molecules (i.e., ligands) can bind to a given protein, depending on the number of binding sites on its molecular surface. It happens that, as noted by [HSAH⁺09], checking whether a certain molecule binds a particular protein takes a lot of time in lab. In general, binding sites correspond to the concave, cleft or hole-shaped regions on a protein surface (cf. [KG07]), called cavities, though not all cavities end up being binding sites for small ligands. Taking this into consideration, the detection of binding sites can be simplified in computational terms by locating the cavities in a molecular surface.

So, in this chapter, we describe an algorithm to determine the cavities on the protein

surface as tentative binding sites for ligands. The leading idea of this algorithm is to use the notion of curvature of the scalar field—rather than the curvature of the molecular surface—to detect protein cavities. In fact, the novelty of the method is the detection of protein cavities through the calculation of the curvature of the scalar field outside the surface, rather than computing the curvature on the molecular surface. This is subtle because it marks the difference between locating cavities of molecule or not.

Also this method is the first of its class, here called class of curvature-based methods, to succeed in finding the downs of a molecular surface without using the surface directly. More specifically, our method relies on the Morse theory to identify possible cavities on the protein surface. It is clear that some research works have already tried to take advantage of the concept of curvature (see, for example, [NWB⁺06]), but the curvature is a local geometric measure that quantifies the bending of a surface at one of its points. On the contrary, our method measures the curvature in a larger extent of the surface, here called zonal curvature, instead of measuring the curvature at a surface point as usual [EKB02].

Let us now see how the remainder of this chapter is organized. Section 5.2 briefly approaches the related work existing in the literature. Section 5.3 briefly approaches the scalar field theory in the context our algorithm. Section 5.4 describes our algorithm in detail, as well as its implementation. Section 5.5 comes out with the most relevant results produced by our method, including its comparison with other well-know algorithms found in the literature. Section 5.6 draws the main conclusions about our algorithm, leaving important hints for future work.

5.2 Related Work

In the literature, we find two main families of methods to detect protein cavities: energy-based and geometry-based (cf. [NSG12]). *Energy-based methods* calculate the energy that results from the interaction between protein atoms and a small-molecule probe, whose value dictates the existence or not of a cavity. In turn, *geometry-based methods* aim at detecting solvent accessible regions of the protein surface only using geometric criteria, some of which will be approached further below. Interestingly, as noted [SSE⁺10], both families of methods perform quite well detecting around 95% of the known cavities. Additionally, the geometry-based methods are faster and more robust (against structural variations or missing atoms/residues in the input data concerning proteins) than the energy-based algorithms, particularly in a context of a large-scale prediction of potential binding cavities [SSE⁺10].

Even considering that the concept of cavity has been widely accepted in the research community, its geometrical definition is not straightforward ([KG07]). In fact, the shapes of cavities range from small spherical invaginations to deep curved or linear clefts in the protein ([CS10]). Besides, there is not necessarily a relation between the

size of the binding sites (and also their possible cavities) and the size of the ligand it binds ([NH06, KMLJ07]). On average, the size of a drug-binding cavity is in the range 600-900 Å (cf. [SHF⁺06, KG07, PSM⁺10]), while the size of a drug-like ligand is about 400 Å. Nevertheless, drug molecules are frequently designed as small as possible in order to ensure their bioavailability. Interestingly, researchers have observed that ligands (drugs, in particular) commonly bind into the largest and/or deepest concavity on the protein surface ([NH06]). Such cavity might be three times as large as the ligand, what shows us how hard it is to delineate the boundaries of what really is a protein cavity.

In a way, all geometric methods assume that a cavity is a depression of the protein surface, so it has a specific depth. In general, geometry-based methods can be divided into three main categories: grid-based, sphere-based, and triangulation-based ([PSM⁺10]). *Grid-based methods* use an axis-aligned 3D grid embedded in the domain $D \in \mathbb{R}^3$ that encloses a given molecule. The grid nodes are then classified as belonging to one of the following types: "outside", "inside", and "on" the protein. Then, those nodes that are not inside of the protein are geometrically clustered in order to form cavities. Usually, a different geometric clustering criterion embodies a new geometric method for detecting cavities on the protein surface. For example, the Levitt-Banaszak method (cf. [LB92]), implemented in POCKET program, makes use of line segments defined by consecutive outside grid nodes that are delimited by inside or on grid nodes, which is equivalent to determine those regions outside the protein that are delimited by the protein surface itself in a number of directions; those regions are where the solvent lies in. The reader is referred to [Del92], [MD95], [VJOW03], [LJ05] and [WPS07] for further details about different geometric criteria.

In *sphere-based methods*, the leading idea is to fill the concave regions of the protein surface with probe spheres of different radii, in a basis of layer-by-layer, until a cutting threshold is satisfied, as shown in [KBO⁺82], [Las95], [RWJ97], and [BS00]. If a probe sphere overlaps any atom, the sphere radius decreases in order to ensure that the probe does not overlap with any other atom. One of the first sphere-based methods to be implemented is due to [Las95] via SURFNET, but other similar programs have been proposed in the literature, as it is the case of PASS and PHECOM ([KG07]), and POCASA ([YZTY10]).

In *triangulation-based methods*, detecting protein cavities is typically accomplished using alpha shape methods and Voronoi diagrams, from which we are able to build up Delaunay triangulations of the atoms of a given molecule ([PFF96, LWE98]). An alpha shape of a molecule is nothing more than a triangulation that decomposes the space occupied by its atoms in a unique manner ([ASP⁺08]), with the particularity of additionally capturing the shape of the molecule itself ([BAM⁺14]). In general, a molecule has an infinite number of alpha shapes, with each alpha shape defined by a specific alpha radius. For example, the alpha shape with infinite radius coincides with the convex hull of such set of points. Therefore, the concept of alpha shapes generalizes the concept of the convex hull of a point set [EM94, EFL95, Ede98]. By gradually decreas-

ing the alpha radius, the alpha shapes monotonically retract, so that the cavities (i.e., voids, pockets, and depressions) become more and more apparent. These cavities correspond to empty simplexes of the initial triangulation, i.e., simplexes located outside the protein.

Curvature-based methods have not succeeded so far because the curvature analysis is performed locally on the molecular surface, i.e., for each point belonging to the molecular surface. The topographical properties of a surface are evaluated by measuring the two canonical curvatures at each surface point, which are featured by the eigenvalues of the Hessian matrix on each surface point ([EKB02]). Zachmann [ZHSB92] introduced the concept of global curvatures to describe larger surface regions involved in molecular recognition, where global curvatures are seen as average curvatures of the corresponding surface region. Similarly, [NWB⁺06] introduced a new method for segmentation of molecular surfaces using the Morse theory. However, the Morse theory is used for points of the surface in the attempt of segmenting the surface into regions that identify characteristic features.

On the contrary, we have applied curvature analysis to the scalar field (or function) that describes the surface, rather than the surface itself. Our argument is that cavities are in the vicinity of saddle points of the scalar field outside the molecular surface. Recall that in \mathbb{R}^3 , we also have other types of critical points, as it is the case of maxima (or sinks) and minima (or sources), but these critical points do not contribute to the detection of cavities on the molecular surface. What we propose in the present chapter is thus to use critical points outside the molecular surface to identify its cavities. Besides, seemingly, this chapter describes an algorithm for detecting cavities on GPU (graphics processing unit) using CUDA architecture, which speeds up its overall performance.

5.3 Theoretical Background

This chapter describes an algorithm for detecting cavities using scalar fields, and their critical points within an axis-aligned boxed domain $D \subset \mathbb{R}^3$ that encloses a given molecule.

5.3.1 Scalar Fields

A *scalar field* associates a scalar value to every single point in a given space. More specifically, a scalar field in \mathbb{R}^3 is defined by a real function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, such that every point $\mathbf{p} \in \mathbb{R}^3$ is assigned a scalar value in \mathbb{R} calculated by f . Following [Bli82], [GW96], [GP95], and [ZXB06], we use a Gaussian function f_i to model the scalar field generated by each atom i in D , which is given by

$$f_i(\mathbf{p}) = e^{B_i \left(\frac{\|\mathbf{p} - \mathbf{p}_i\|^2}{r_i^2} - 1 \right)} \quad (5.1)$$

where $\|\mathbf{p} - \mathbf{p}_i\|$ stands for the distance of an arbitrary point \mathbf{p} to the center \mathbf{p}_i of the i -th atom of the molecule, r_i is the radius of atom i , and B_i denotes the decay rate of each atom's Gaussian kernel. Note that B_i must be negative to guarantee that the scalar field f_i of the i -th atom goes to zero as $\|\mathbf{p} - \mathbf{p}_i\|$ goes to infinity. This means that the scalar field of each atom is stronger at its center than at any other point of D . Summing up the scalar fields of the n atoms of a molecule, we get the overall scalar field F of the molecule as follows:

$$F(\mathbf{p}) = \sum_{i=0}^{n-1} f_i(\mathbf{p}) \quad (5.2)$$

5.3.2 Gaussian Molecular Surface

According to [Bli82], the Gaussian scalar field F allows us to define a family of molecular isosurfaces in \mathbb{R}^3 for the same molecule (or set of atoms) in the following manner:

$$F(\mathbf{p}) - c = 0 \quad (5.3)$$

where $c \in \mathbb{R}$ stands for a varying real isovalue, one per isosurface. Note that these molecular isosurfaces given by Eq. (5.3) are smooth because they result from the summation of smooth Gaussian functions f_i (cf. Eq. (5.1)) associated to their atoms. Besides, each function f_i must represent the decaying behavior of the electron density field of the atom i with the distance from its center.

Besides, one can even generate distinct molecular surfaces for the same molecule by exclusively varying the blobbiness value B in Eq. (5.1), i.e., even keeping the isovalue unchanged. Indeed, as [Bli82] noted, a molecular surface with increasingly blobbiness tends to get more and more a sphere-like shape, so that any shape details of the original surface tend to not be preserved.

5.3.3 Critical Points

The function F given by Eq. (5.2) is smooth because its subsidiary Gaussian functions f_i are smooth. In differential geometry, a critical point of a smooth function is any point \mathbf{p} in its domain where the gradient vanishes, i.e., $\nabla F(\mathbf{p}) = \mathbf{0}$. Each critical point is classified with respect to the eigenvalues of the Hessian matrix of F :

$$H(\mathbf{p}) = \begin{bmatrix} H_{00} & H_{01} & H_{02} \\ H_{10} & H_{11} & H_{12} \\ H_{20} & H_{21} & H_{22} \end{bmatrix} \quad (5.4)$$

where $H_{ij} = \frac{\partial^2 F}{\partial u^i \partial v^j}$, with $i, j = 0, 1, 2$ and $u^i, v^j = x, y, z$, stands for each one of the second partial derivatives of F at any point \mathbf{p} of the domain. The Hessian matrix H describes how fast the slope of a function changes, so it gives a measure of the function curvature across the domain, including the curvature of the surface defined by $F(\mathbf{p}) = 0$.

However, the Hessian matrix (H) is not invariant under rotations. To overcome this issue, the Hessian matrix must be normalized as follows:

$$N(\mathbf{p}) = \begin{bmatrix} N_{00} & N_{01} & N_{02} \\ N_{10} & N_{11} & N_{12} \\ N_{20} & N_{21} & N_{22} \end{bmatrix} \quad (5.5)$$

where each N_{ij} is given by the following expression:

$$N_{ij} = \frac{H_{ij} \|\nabla F\| - \frac{\nabla F_i (\nabla F \cdot H_i)}{\|\nabla F\|}}{\|\nabla F\|^2} \quad (5.6)$$

and ∇F_i stands for the i th component of the gradient vector, $H_i = [H_{i0}, H_{i1}, H_{i2}]^T$ is the transpose of the i th row of $H(\mathbf{p})$, and \cdot denotes the dot product of two vectors. From the normalized Hessian matrix N , we then calculate the principal curvatures of the implicit function F .

In differential geometry, the two principal curvatures determine the local shape of a point of the scalar field F , and the same applies to the surface $F(\mathbf{p}) = 0$. They measure how much the scalar field bends in different directions at that point. The first principal curvature is the maximum curvature (i.e. the rate of maximum bending of the scalar field), while the second principal curvature is the minimum curvature (i.e. the rate of minimum bending of the scalar field). The curvature at a scalar field point takes its maximum and minimum values along two directions that define two planes that are normal to the tangent plane, with the particularity that these two normal planes are perpendicular to one another.

The two principal curvatures at a given point of a scalar field are the first two eigenvalues of the normalized Hessian matrix at such point; the third eigenvalue is always zero since its eigenvector is normal to the scalar field (and thus collinear to the gradient vector ∇F). Note that the eigenvectors associated to principal curvatures at a given point \mathbf{p} of the domain are the principal directions defined in the plane tangent to the scalar field at \mathbf{p} . Hereupon, we can classify a critical point in the domain as follows: (i) a *maximum* if both eigenvalues are positive; (ii) a *minimum* if both eigenvalues are negative; (iii) a *saddle* if one eigenvalue is positive, while the other is negative. The saddle points of the scalar field outside the molecule indicate where surface cavities are.

5.4 Cavity Detection Algorithm

The CriticalFinder algorithm adheres to the category of curvature-based methods, although it takes advantage of the voxelization of the domain $D \in \mathbb{R}^3$. This voxelization is also needed to triangulate and render the molecular surface through the marching cubes (MC) algorithm, which is due to [LC87]. Nevertheless, we use the MC variant described in [DG11]. Both CriticalFinder and MC-based triangulation algorithms were designed and implemented to run on GPU via C++/CUDA.

5.4.1 Overview

In general terms, the algorithm can be described by the following steps:

- *Voxelization*: The domain D (i.e., an axis-aligned bounding box enclosing the molecule) is divided into a lattice of voxels or cubes.
- *Evaluation of the Discrete Scalar Field*: The scalar field F (cf. Eq. (5.2)) is evaluated at each grid node or voxel corner.
- *Computation of Critical Points*: Unlike other methods that try to find critical points on the molecular surface, we search for critical points in the set-theoretic complement of the molecule inside the domain (i.e., bounding box).
- *Computation of the Number of Critical Points*: This is necessary to carry out the clustering of critical points into cavities.
- *Clustering Critical Points into Cavities*: Clustering of critical points into cavities is necessary because the algorithm is able to find various critical points for the same cavity.
- *Surface Triangulation*: If the surface intersects a voxel, the surface patch inside such a voxel is triangulated according to one out of 256 possible MC triangulations.
- *Rendering of Surface and its Cavities*: This is the last step of the algorithm, and is essentially used for visualisation and rendering of both molecular surface and its cavities.

The first five steps constitute the cavity detection algorithm (i.e., CriticalFinder). The sixth step serves the purpose of triangulating the molecular surface via MC algorithm, while the seventh step is responsible for displaying the surface and its cavities on screen.

5.4.2 Voxelization

The algorithm starts by reading the centers of each atom of the molecule from the PDB file in a 1-dimensional array, named `CENTERA`, which is then transferred into the GPU global memory using the `cudaMemcpy` function.

The voxelization of the bounding box requires the preliminary calculation of its size. In this regard, the coordinates of the n atom centers are essential to determine the axis-aligned bounding box that encloses the molecule. We use the coordinates of each atom center to update the opposite corners $(x_{MIN}, y_{MIN}, z_{MIN})$ and $(x_{MAX}, y_{MAX}, z_{MAX})$ of the principal diagonal of the bounding box. Then, we determine the number $I = I \times J \times K$ of voxels or cubes with size of $\Delta = 0.3 \text{ \AA}$ that fit the bounding box, where I , J , and K stand for the number of voxels along x -, y -, and z -axis, respectively. All voxels are then stored in a 1-dimensional array of N elements, named `VOXELA`; more specifically, each array element stores the coordinate-minimum corner of each voxel, not the voxel itself, because the remaining seven corners can be easily inferred from such a coordinate-minimum corner and the voxel size Δ . Note that the computation of the number of voxels in the bounding box in the x -, y -, and z -axis directions is accomplished using a CUDA kernel, called `boundingBoxSizeCalculator`, while the computation of the coordinate-minimum corner of each voxel is carried out by another kernel on GPU, called `coordinateMinimumCornersCalculator`.

5.4.3 Evaluation of the Discrete Scalar Field

The `discreteScalarFieldEvaluator` CUDA kernel has the task of computing the value of the density field F on each voxel corner. However, before running this kernel on GPU, it is necessary to allocate memory on the GPU side for the 1-dimensional array that will hold the value of F at the coordinate-minimum corner of each voxel; this array is named `FUNCTIONA`, and has the same size as `VOXELA`.

The computation of F is carried out in an atom basis, i.e., one carries out the computation of the contribution f_i of the atom i to the value of F at each coordinate-minimum corner of the voxels belonging to a sub-box surrounding the atom i . Assuming that each voxel has a size of 0.3 \AA , and taking into account that the atom radius is about 1.0 \AA , then a sub-box with $12 \times 12 \times 12$ voxels guarantees a smooth transition of the molecular surface from one atom to another. The parallelization of this kernel is then accomplished by running a single thread per atom.

5.4.4 Computation of Critical Points

As said above, our algorithm calculates critical points outside the surface, not on the molecular surface, i.e., in the set-theoretic complement of the molecule inside the domain. The idea was inspired in the concept of critical regions of the domain, as proposed by [Gom14]. The critical point finding algorithm consists of the following steps:

- *Critical Point Detection*: Every single critical point is where the gradient (i.e., the first-order partial derivatives) vanishes in the domain. We relax this condition saying that the gradient annihilates approximately at every critical point. The computation of

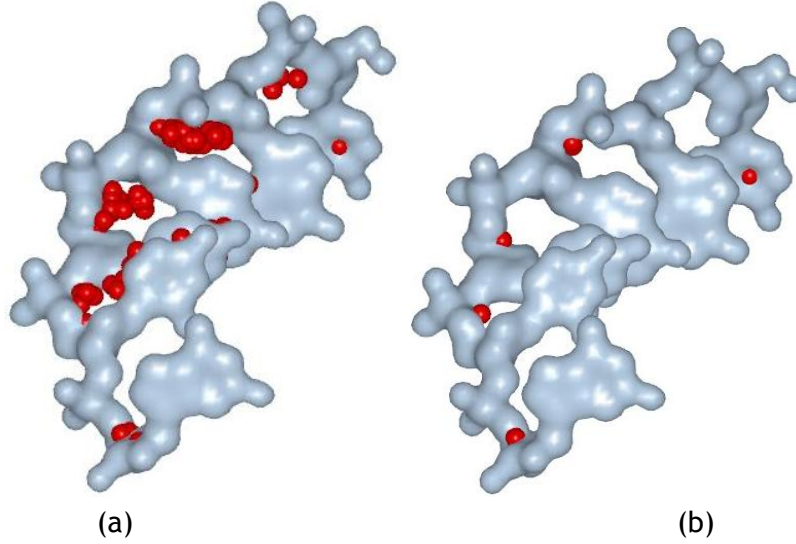


Figure 5.1: Critical points as red balls considering: (a) the voxels outside or intersecting the surface; (b) only the voxels intersecting the surface.

the partial derivatives for the coordinate-minimum corner of each voxel is carried out by the `CriticalPointDetector` kernel on GPU side. The (i, j, k) -index of those corners with nearly null gradient is stored in a 1-dimensional GPU array of size m , here called `DETECTA`.

- **Hessian Matrix Computation:** Before computing the Hessian matrix for each corner indexed in `DETECTA`, we need to allocate a GPU memory array for it, called `HESSIANA`. Then, by iterating on the `DETECTA` array, one calculates the second-order partial derivatives for each critical corner (cf. Eq. (5.4)), which are collected as a whole into the corresponding element of `HESSIANA`.

- **Normalized Hessian Matrix Computation:** The Hessian matrix of each critical corner in the `HESSIANA` array is normalized using Eq. (5.5), with the resulting matrix held in the `NHESSIANA` array. The first two normalized second-order partial derivatives are exactly the first two eigenvalues of the Hessian matrix, while the third eigenvalue vanishes following the normalization procedure.

- **Computation of Eigenvalues:** The `EigenCalculator` kernel computes the eigenvalues of the normalized Hessian matrix at each critical corner, which are held in an array named `EIGENVALUESA`.

- **Classification of Critical Corners:** Before classifying the critical corners into either saddle points (integer label `1') or maximum/minimum points (integer label `0'), we need to allocate GPU memory for another 1-dimensional array, called `CLASSIFICATIONA`, having the same size m as the `DETECTA` array. For each corner, we check whether its first two eigenvalues in the `HESSIANA` array have opposite signs or not. If so, and in conformity with the classification in the end of Section 5.3, the corresponding element of the `CLASSIFICATIONA` array is labelled with `1'; otherwise, it will be `0'.

The critical corners labelled with `1' are saddle points of the scalar field outside the

surface. These saddle points are depicted in Fig. 5.1 as red small balls: (a) considering all the voxels outside or intersecting the surface; (b) considering only the voxels intersecting the surface.

5.4.5 Computation of the Number of Critical Points

Computing the total number of critical points is done using the Parallel Prefix Sum (PPS) operation on the `CLASSIFICATION`. In CUDA, this is accomplished by calling the kernel function `cudppScan` that comes with CUDA.

5.4.6 Clustering Critical Points into Cavities

Clustering critical points into cavities is an algorithm that consists of the following steps:

- *Memory allocation of critical points on GPU:* Having found the critical points, we allocate a GPU array (called `CRITICALA`) for their 3D locations, setting then their corresponding x , y , and z coordinates.
- *Formation of cavities:* This corresponds to `clusteringCriticalPointsIntoCavities` CUDA kernel, which collects critical points into separate cavities. This operation starts with the critical points that are corners of voxels that intersect the molecular surface (Fig. 5.1(b)). Then, every other critical point is collected into a specific cavity if its distance to a critical point already in the cavity is less than 1 Å. It is clear that the center of each cavity is given by the average location of its critical points. As seen further ahead, these cavity centers are important to compare with those found by other software packages.

5.4.7 Surface Triangulation

In order to render the molecular surface on screen, we have developed a triangulation algorithm that entirely runs on GPU. In its essence, it is a variant of the marching cubes algorithm (cf. [LC87]), here used to triangulate molecular surfaces. Its particularities stem from the fact that it is an atom-centric triangulation algorithm for molecular surfaces, i.e., the computation of the value of the scalar field, at any point of the domain, is done in an atom basis. The reader is referred to [DBG10] and [DG11] for further details.

5.4.8 Rendering of Surface and its Pockets

Having terminated the triangulation, we need to calculate the normal vector at each vertex of the triangulation in order correctly shade the molecular surface. Considering

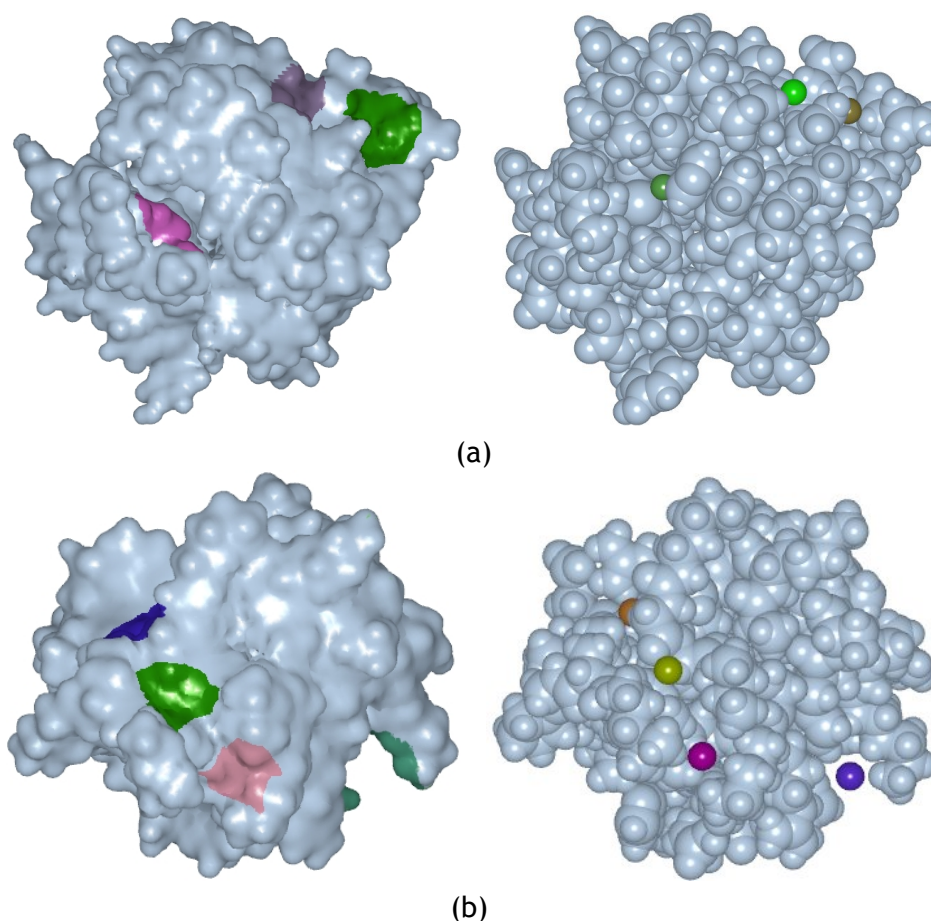


Figure 5.2: CriticalFinder (left) and LIGSITE (right) output the same number of cavities on the same locations for two proteins: (a) PDB id: 2gqv (20 cavities); (b) PDB id: 2nwd (18 cavities).

that each vertex is a point of the molecular surface, we only need to calculate the normal to the surface at such a vertex, which is given by the gradient vector ∇F (i.e., the first-order derivatives). These normals are held in a NBO (Normal Buffer Object) array, which has the same size as the array of vertices previously allocated in the triangulation step. The computation of these normals is carried out by another CUDA kernel, called `normalsCalculator`.

Besides, we use another kernel to shade the cavities on the molecular surface, what ends up being a sort of molecular surface segmentation, as shown in Fig. 5.2. For that, we calculate the distance of each triangulation vertex to each cavity center, checking then whether or not there is any obstacle in between, unless small balls concerning critical points. If so, we assign a cavity-specific color to that vertex on the surface.

5.5 Results

5.5.1 Hardware/Software

In testing, we used a desktop computer equipped with a Intel Core I7-4820k 3.70 Ghz Processor, 32GB RAM, and two graphics cards, namely one Nvidia Tesla K20 and one Nvidia Quadro K5000. The Nvidia Tesla K20 graphics card was used for general-purpose massive computations, while the Nvidia Quadro K5000 was used to render triangulated molecular surfaces and their cavities. In terms of software, the computer was running the Linux Fedora 20 operating system, with all the programming done in CUDA/C++. Both graphics cards (or devices) follow the Kepler architecture.

Device Type	Tesla K20
Memory Size (GB)	5.0
Streaming Multiprocessors (SM)	13
CUDA Cores	2,496
Maximum Number of Active Blocks per SM	16
Maximum Number of Active Warps per SM	64
Maximum Number of Active Threads per SM	2,048
Maximum Number of Threads per Block	1,024
Maximum Number of Warps per Block	32
Maximum Number of Registers per Thread	255
Maximum Number of Registers per Block	65,536

Table 5.1: Kepler GPU's memory and compute capabilities.

# kernel	Active Blocks	Active Warps	Active Threads	Threads per Block	Warps per Block	Registers per Thread	Registers per Block	Multiprocessor Occupancy
1st kernel	17	30	1041	64	2	128	589	56%
2nd kernel	11	15	788	64	2	108	660	40%
3rd kernel	15	10	542	64	2	102	768	29%
4th kernel	4	18	394	64	2	154	545	23%
5th kernel	2	11	276	64	2	194	458	21%

Table 5.2: Performance data *before* optimizing the five CUDA kernels (i.e., five steps of CriticalFinder).

# kernel	Active Blocks	Active Warps	Active Threads	Threads per Block	Warps per Block	Registers per Thread	Registers per Block	Multiprocessor Occupancy
1st kernel	9	58	1924	192	6	30	4420	96%
2nd kernel	9	58	1924	192	6	30	4765	96%
3rd kernel	9	58	1924	192	6	30	4584	96%
4th kernel	7	40	1644	192	6	35	3739	81%
5th kernel	7	35	1546	192	6	38	3106	78%

Table 5.3: Performance data *after* optimizing the CUDA kernels.

5.5.2 CUDA Code Optimization

The leading idea of our hardware/software setup was to run our cavity detection code on Nvidia Tesla K20 graphics card, while the Nvidia Quadro K5000 graphics card was only used for rendering and visualization of molecular surfaces and their cavities. Therefore, we needed only take care of the code optimization on Tesla K20.

The memory and compute capabilities of the Nvidia Tesla K20 graphics card appear listed in Table 5.1. Note that this device supports up to 16 active blocks, 64 warps, 2048 threads per multiprocessor simultaneously, being feasible to take advantage of 192 cores per multiprocessor ($2,496/13=192$) at maximum. In terms of CUDA, a kernel executes on the GPU device as a grid of blocks of warps of threads. As shown in Table 5.1, a block consists of 32 warps, whereas a warp is a set of 32 threads; consequently, we have 1024 threads per block at maximum.

As usual, optimizing codes of the CUDA kernels took into account the architecture, type, and capabilities of GPU device. As known, the compute capability of a Nvidia Tesla K20 graphics card is 3.5, which has to be entered in the command line that initiates the compilation of CUDA code (kernels) using `nvcc`. In order to optimize the code, we had to run it at least once. Then, we used a Nvidia profiler named Nsight (<http://www.nvidia.com/object/nsight.html>) to better analyze the performance data of the CUDA code on GPU device. The performance data of CUDA code before its optimization are shown in Table 5.2. In turn, after optimizing the CUDA kernel codes, we obtained the performance data listed in Table 5.3. Recall that the process of optimizing CUDA code essentially has to do with occupancy of multiprocessors at work, i.e., more multiprocessor occupancy means more efficiency in running CUDA kernels on GPU device. The optimization degree of CUDA code appears indicated on the last column of both Tables 5.2 and 5.3, where the streaming multiprocessor occupancy is expressed as the ratio of the number of running threads to the maximum number of threads on the GPU.

The code optimization involved the following steps: (i) increasing of the number of threads per block; (ii) increasing of the number of active warps; (iii) decreasing of the number of registers per thread; (iv) fixing the effect of unbalanced workloads. In respect to the first optimization step, we changed the number of threads per block from 160 (cf. Table 5.2) to 192 (cf. Table 5.3), which is the admissible maximum number of threads per block. This change is directly done on Nsight. Consequently, a brief glance at Tables 5.2 and 5.3 shows us that the multiprocessor occupancy increased for every single CUDA kernel, being that more noticeable for the first three kernels. The less efficiency of the last two kernels can be explained by the fact that the Thurst hash tables and dynamic arrays are not optimized at all.

The second optimization step is not done using Nsight. In order to increase the number of active warps, we have only to change our programming style. This is accomplished by reducing the number of 'if-else', 'switch', 'for', and 'while' statements to a minimum

in the CUDA code. This allowed us to increase the number of warps per block from 2 to 6, as shown in Tables 5.2 and 5.3 (cf. sixth column); consequently, the number of active warps also increased (cf. third column).

Similar to the second step, the third step does not make use of Nsight. In practice, the number of registers per thread can be reduced by decreasing the number of mathematical operations existing in the code of each kernel. Cleaning up the code in this manner resulted in a further increase in the code efficiency, as evidenced by the values shown in seventh column of Tables 5.2 and 5.3.

Finally, the fourth optimization step aims at guaranteeing that the workload is distributed in a balanced manner on the GPU device. The solution is to force the division of a kernel into a number of identical sub-kernels; in particular, we divided each kernel into 20 sub-kernels. Each sub-kernel operates on a distinct slice of voxels of the bounding box, forcing in this manner the workload distribution by several blocks, warps, and threads.

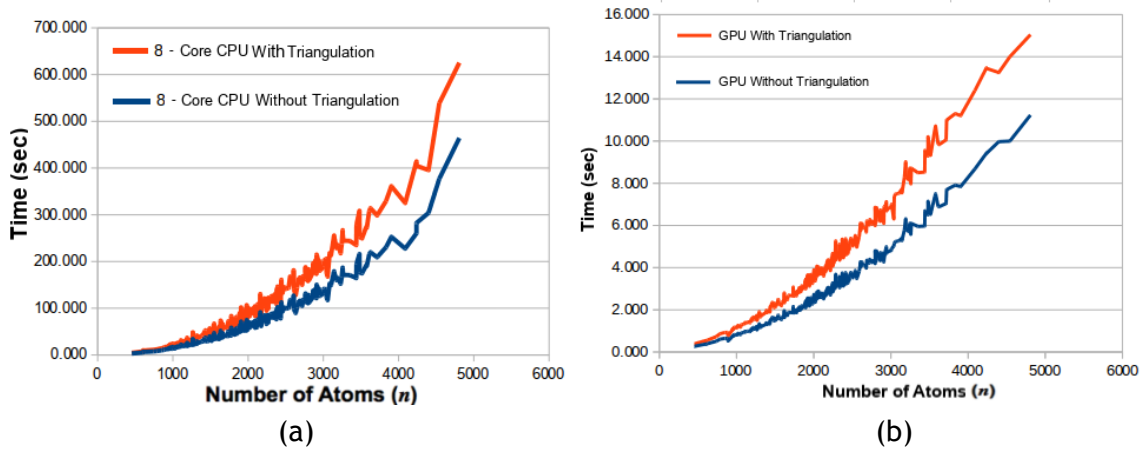


Figure 5.3: Time performance of the algorithm with (in red) and without (in blue) the triangulation and rendering steps: (a) CPU implementation; (b) GPU implementation.

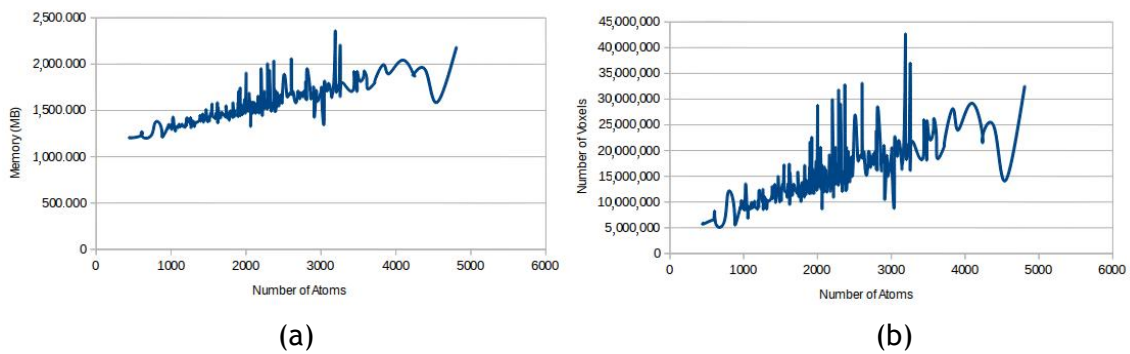


Figure 5.4: (a) Memory space occupancy with respect to the number of atoms (n); (b) the number of voxels in memory with respect to the number of atoms (n).

5.5.3 Ground-Thruth Molecule Dataset

As shown further ahead, the time performance and cavity detection accuracy of CriticalFinder were evaluated for the dataset of apo and holo proteins of the LigASite (<http://ligasite.org/>), which plays here the role of the ground-truth dataset of already known protein binding sites [DLW08]. Recall that LUF4 has 4808 atoms, and it is the biggest protein of the current LigASite dataset. For brevity, the atomic structure of each protein was retrieved from PDB Data Bank (www.rcsb.org).

LigASite comprises 2604 proteins and their binding sites. More specifically, this dataset consists of 816 apo structures and 1788 holo structures of binding sites. An apo protein is also known as a protein without ligands, whereas an holo protein is a protein with ligands (i.e., a protein-ligand complex).

It is worthy noting that LigASite ---as a ground-truth dataset of proteins and their binding sites--- is well suited to benchmark any cavity prediction method, because it includes not only unbounded proteins (i.e., apo proteins), but also the corresponding bounded proteins (i.e., holo proteins), what enables us to check the accuracy of any detection method in regards to the identification of protein cavities, as well as to consider the structural changes of a protein upon binding of its ligands [DLW08].

5.5.4 Time Performance

The time performance of CriticalFinder was evaluated with reference to two of its possible parallelized versions as follows:

- *Multi-threaded CPU Program.* This code takes advantage of 8 CPU cores via multi-threading. It was written in C/C++, together with resources provided by the Standard Template Library (STL). Likewise, for rendering of molecules and their binding sites, we used a multi-threaded CPU-based code of the marching cubes triangulation algorithm as implemented by Dias and Gomes [DBG10].
- *CUDA-based GPU Program.* The GPU-based implementation of CriticalFinder was written in C/C++, CUDA, and with the help of Thrust (i.e., GPU counterpart of STL), a GPU code library that includes hash tables and dynamic arrays on GPU side, which is available at <http://thrust.github.io/>. We also used the marching cubes variant due Dias and Gomes [DBG10] to triangulate and render protein surfaces and their binding sites on GPU.

The time performance results of the CriticalFinder (without triangulation) on CPU and GPU are depicted in Fig. 5.3(a) and (b), respectively, in blue. The results in red include the overhead incurred by the triangulation. After a more fine-grain analysis of data graphically represented in Fig. 5.3, and with the help of a curve fitting tool [Arl94], we noted the following:

- *CPU-based detection of pockets* (Fig. 5.3(a) in blue). The zigzag curve in blue features the CPU time performance of CriticalFinder in its course of identifying pockets (with the triangulation off), which is approximated by the following expression:

$$t = 1.32 \times 10^{-5} \times n^{2.02} \quad (5.7)$$

- *CPU-based detection of pockets plus molecular surface triangulation* (Fig. 5.3(a) in red). The zigzag curve in red represents the CPU time performance of CriticalFinder together with the triangulation algorithm, whose expression is given by

$$t = 1.89 \times 10^{-5} \times n^{2.02} \quad (5.8)$$

- *GPU-based detection of pockets* (Fig. 5.3(b) in blue). The time performance curve of the CriticalFinder on GPU approximately satisfies the following expression

$$t = 0.832 \times 10^{-5} \times n^{1.66} \quad (5.9)$$

- *GPU-based detection of pockets plus molecular surface triangulation* (Fig. 5.3(b) in red). In conjunction with the protein surface triangulation, CriticalFinder's time performance behaves in conformity with zigzag curve in red shown in Fig. 5.3(b), which approximately obeys to the following expression

$$t = 1.21 \times 10^{-5} \times n^{1.65} \quad (5.10)$$

As suggested previously, we used curve fitting to come up with Eqs. (5.7)-(5.10) [Arl94]. These equations show us that the experimental time complexity of CriticalFinder on CPU is quadratic (cf. Eqs. (5.7)-(5.8)), whereas its experimental time complexity on GPU is super-linear (cf. Eqs. (5.9)-(5.10)). For example, CriticalFinder takes about 139.431 seconds on CPU to find the pockets of a protein with 3000 atoms, but only 4.811 seconds on GPU, what amounts a speedup of 28.98x. Taking the protein surface triangulation taken into consideration, the running time obviously increases, such that it takes 199.639 seconds on CPU and 6.872 seconds on GPU, resulting in a speedup of 29.05x.

Considering all proteins of the dataset, the CPU-based CriticalFinder (without triangulation) lasts 360,028 seconds (about 100 hours), while the GPU-based CriticalFinder (without triangulation) takes 14,478 seconds (about 4 hours) to identify the pockets of all proteins, therefrom results an average speedup of 26x. If we also take into account the extra time incurred in triangulating and rendering all proteins, the running times increase to about 468,017 seconds (about 130 hours) and 18,037 seconds (about 5 hours) on CPU and GPU, respectively, i.e., an average speedup of 25x. Note that these times are end-to-end runtimes, i.e., times accounted for running the 5 steps (7 if triangulation

and rendering are included) of the CriticalFinder, as outlined in Section 5.4.1.

5.5.5 Memory Space Performance

Fig. 5.4 shows us that most memory is spent to allocate space for voxels. As shown in Fig. 5.4(a), there are molecules that require more than 2.0GB in memory for their voxels. However, this does not mean that such molecules are the biggest ones in terms of the number of atoms (cf. Fig. 5.4(b)). In terms of memory space complexity (i.e., the memory space occupied by voxels), the algorithm seems to be linear, on average, in regards to the number of atoms.

5.5.6 Accuracy Testing

Measuring the accuracy of CriticalFinder in identifying and locating pockets on protein surfaces certainly is more relevant than measuring its time performance. As aforementioned, we used the LigASite database of binding sites [DLW08] as the ground-truth in terms of the number and location of protein pockets.

5.5.6.1 Benchmarking Methods and Metrics

In addition, we used five cavity detection algorithms bundled with Metapocket [Hua09] as our benchmarking set of algorithms, namely:

- LIGSITE. It implements the grid-based algorithm proposed by Hendlich et al. [HRB97].
- POCASA. It implements a grid-based algorithm (called Roll), although it also makes usage of a crust-like surface of probe spheres (see Yu et al. [YZTY10] for further details).
- SURFNET. It implements the sphere-based algorithm proposed by Laskowski [Las95].
- PASS. It implements the sphere-based algorithm introduced by Brady et al. [BS00].
- fpocket. It implements a triangulation-based algorithm, i.e., a alpha shape algorithm (see Guilloux et al. [LGST09]).

In testing, we used the following metrics: numerical weighted hit deviation (σ), positional weighted hit deviation (δ), and cumulative cavity percentage (P).

The testing results are shown in Table 5.4, where $\Delta_i = C_i - c_i$ is the difference between the number C_i of cavities detected by CriticalFinder and the number c_i of cavities detected by each of those five methods.

Δ_i	LIGSITE		PASS		SURFNET		POCASA		fpocket		Mean Average	
	n_i	h_i	n_i	h_i	n_i	h_i	n_i	h_i	n_i	h_i	n_i	h_i
-4	41	70	88	97	100	108	126	139	145	154	100	113
-3	52	76	96	104	125	134	149	154	167	170	117	127
-2	65	84	102	121	136	148	157	161	173	182	126	139
-1	91	107	128	137	147	159	162	173	186	192	142	153
0	2048	1897	1768	1694	1590	1489	1411	1339	1250	1189	1613	1521
1	108	117	137	149	152	163	168	175	184	195	149	159
2	78	95	106	112	129	141	153	162	179	181	129	138
3	67	86	92	100	118	137	141	157	161	172	115	130
4	54	72	87	90	107	125	137	144	157	169	108	120

Table 5.4: Hit performance (n_i and h_i) of CriticalFinder with respect to LIGSITE, PASS, SURFNET, POCASA, and fpocket.

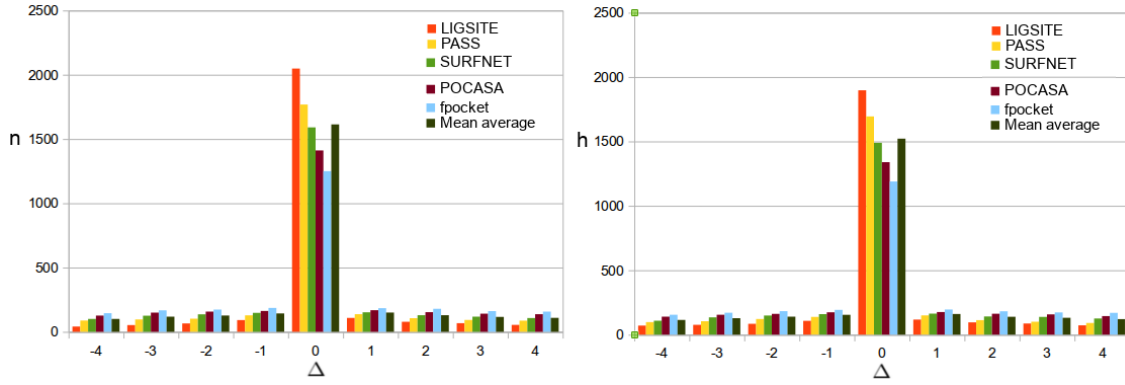


Figure 5.5: Histograms of hit performance of CriticalFinder in relation to LIGSITE, PASS, SURFNET, POCASA, and fpocket: (left) numerical hit performance n_i in function of the dispersion of the number of cavities Δ_i ; (right) positional hit performance h_i in function of the dispersion of the number of cavities Δ_i .

5.5.6.2 Benchmarking Metrics

The benchmarking metrics we used in our testing were the following: numerical hit weighted deviation (σ), positional hit weighted deviation (σ_p), and cumulative cavity percentage (P).

Numerical Hit Weighted Deviation. After calculating the number of cavities of all proteins of the LigASite, we noted that the difference between the number of cavities found by CriticalFinder and the number of cavities found by any other benchmarking algorithm was 4 maximum in absolute value for every single protein, i.e., the difference of cavities was within a range $[-4, 4]$, as shown in Fig. 5.5(left).

With this in mind, we decided to use the metric given by the numerical hit weighted deviation, which is as follows:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{k-1} n_i \Delta_i^2}{N}} \quad (5.11)$$

where n_i denotes the number of hits (i.e., number of proteins) for a specific Δ_i (i.e., the

difference between the number of cavities determined by two methods, CriticalFinder and any other method), k is the number of bins i (i.e., the number of distinct values of Δ_i), and $N = 2606$ stands for the total number of proteins of the LigASite dataset. That is, σ represents the numerical hit deviation of CriticalFinder in relation to each of those five methods listed in Table 5.4. For example, considering the three central bins ($k = 3$) in Fig. 5.5(left) concerning $\Delta_0 = -1$, $\Delta_1 = 0$, and $\Delta_2 = 1$, we have the following:

- LIGSITE: $\sigma = 1.13$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 2247 hits, i.e., 86.22% of hits.
- PASS: $\sigma = 1.46$, for hits in the range $\Delta_i \in [-1, 1]$, in a total 2033 hits, i.e., 78.01% of hits.
- SURFNET: $\sigma = 1.62$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 1889 hits, i.e., 72.48% of hits.
- POCASA: $\sigma = 1.79$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 1741 hits, i.e., 66.80% of hits.
- fpocket: $\sigma = 1.91$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 1620 hits, i.e., 62.16% of hits.

As shown in Fig. 5.5(left), the absolute maximum dispersion is $|\Delta| = 4$ in terms of the number of cavities found by CriticalFinder in relation to any other benchmarking algorithm. For example, in the particular case of $|\Delta| = 1$, CriticalFinder and LIGSITE attain 86.22% of hits, i.e., they find the same number of cavities, with a variation of 1 hit at maximum (i.e., within in the range $[-1, 1]$), in 86.22% of the proteins; CriticalFinder and PASS agree in 78.01% of hits; CriticalFinder and SURFNET match in 72.48% of hits; CriticalFinder and POCASA coincide in 66.80% of proteins, whereas CriticalFinder and fpocket match in 62.16% of proteins.

Positional Hit Weighted Deviation. On the other hand, we were also interested in knowing if the benchmarking algorithms, including CriticalFinder, were able to detect same cavities about the same locations of proteins. For that purpose, we came across with the concept of positional hit weighted deviation (δ). When comparing CriticalFinder with any other benchmarking algorithm, it can be said that there is a positional hit (for a protein) if the geometric centers of the cavities of a protein, as determined by the CriticalFinder, are all within 4 Å of the corresponding geometric centers calculated by the other algorithm. In practice, the value of δ is calculated through the following formula:

$$\delta = \sqrt{\frac{\sum_{i=0}^{k-1} h_i \Delta_i^2}{N}} \quad (5.12)$$

where k stands for the number of bins i , h_i denotes the number of positional hits, while Δ_i is the difference between the number of cavities as determined by CriticalFinder and any other detection algorithm of the benchmark. For example, taking once again into account the three central bins ($k = 3$), now in Fig. 5.5(right), i.e., $\Delta_0 = -1$, $\Delta_1 = 0$, and $\Delta_2 = 1$, we obtained the following results:

- LIGSITE: $\delta = 1.339$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 2121 hits, i.e., 81.45% of hits.
- PASS: $\delta = 1.523$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 1980 hits, i.e., 76.03% of hits.
- SURFNET: $\delta = 1.713$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 1811 hits, i.e., 69.54% of hits.
- POCASA: $\delta = 1.855$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 1687 hits, i.e., 64.78% of hits.
- fpocket: $\delta = 1.968$, for hits in the range $\Delta_i \in [-1, 1]$, in a total of 1576 hits, i.e., 60.52% of hits.

Therefore, taking into consideration the absolute hit dispersion of $|\Delta| = 1$, we observe that CriticalFinder and LIGSITE they find the same cavities about the same locations (within 4 Å maximum) in 81.45% of hits (i.e., proteins). This percentage decreases to 76.03% in the case of PASS, being 69.54% for SURFNET, 64.78% for POCASA, and 60.52% for fpocket.

Cumulative Cavity Percentage. In the previous two metrics, we compared CriticalFinder with the remaining benchmarking detection algorithms; that is, the latter played the role of ground truth to evaluate the efficiency of CriticalFinder.

In the following, the ground truth concerning the number of cavities and their locations is provided by the apo and holo proteins existing in LigASite dataset of real binding sites. For that purpose, we previously computed the ground-truth geometric center of each cavity as the barycenter of the centers of the surface atoms associated to such cavity, after retrieving those surface atoms from the apo or holo structure of the respective binding site. Therefore, the apo and holo structures in LigASite correspond to separate PDB files, whose binding sites here work as benchmark structures for cavities on protein surfaces.

The cumulative cavity percentage $P \in [0.0, 100.0]$ can be defined as the ratio of number of detected cavities to the number of real cavities (i.e., ground-truth cavities) when the distance d (in Å) between their homologous geometric centers vary in the range $[0.0, 4.0]$. The results produced by the benchmarking detection algorithms relatively to the ground-truth cavities of apo and holo proteins are shown in Fig. 5.6. Obviously, cavities detected beyond $d = 4.0$ Å in relation to the geometric centers of the ground-truth

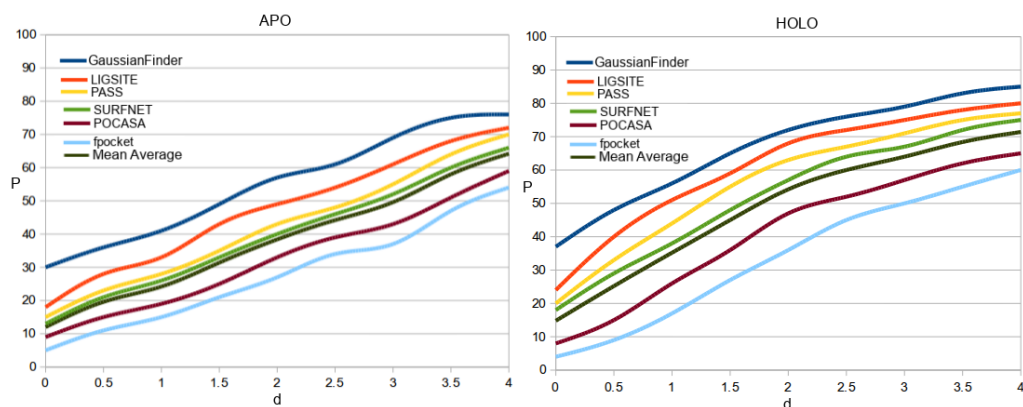


Figure 5.6: Cumulative cavity percentage of various detection methods in function of the distance d to ground-truth geometric centers: CriticalFinder, LIGSITE, PASS, SURFNET, POCASA, and fpocket for apo (left) and holo (right) structures.

cavities were not considered for comparison sake, but the amount of these far-away cavities is negligible. As far as apo proteins (i.e., unbounded proteins) are concerned, the cumulative cavity percentages P in the range $d \in [0, 4]$ were the following: CriticalFinder (76%), LIGSITE (72%), PASS (71%), SURFNET (66%), POCASA (59%), and fpocket (54%). In regards to holo proteins (i.e., protein-ligand complexes), the percentages were the following: CriticalFinder (85%), LIGSITE (80%), PASS (77%), SURFNET (75%), POCASA (64%), and fpocket (61%). In light of previous results, also expressed in Fig. 5.6, we conclude the following:

- CriticalFinder seemingly outperforms other cavity detection methods.
- The grid-based methods (CriticalFinder, LIGSITE, and PASS) seemingly are better than sphere-based methods (SURFNET and POCASA) in our testing; in turn, the sphere-based methods produce better results than triangulation-based methods (fpocket).
- The benchmarking geometric methods considered in our testing tend to detect the same number of cavities on the same locations on the protein surfaces.
- Any benchmark method performs better for holo proteins than for app proteins, simply because the number of cavities of holo proteins usually is less than the number of cavities of apo proteins.

Note that, in our testing, we only considered geometric detection methods for cavities (i.e., tentative binding sites), but we actually used actual locations of binding sites of proteins (via LigASite) as the ground-truth for location and number of cavity on protein surfaces.

5.6 Concluding Remarks

We have developed a novel algorithm for identification of cavities over a molecular surface using the theory of critical points. At our best knowledge, this is the first algorithm belonging to the class of curvature-based algorithms that successfully detects and delineates cavities as usual in other classes of algorithms. It is true that other algorithms have tried unsuccessfully to use the concept of curvature (via eigenvalues of the normalised Hessian matrix) to detect cavities on the surface. The reason why our algorithm works is that we are not trying to know the curvature *on* the surface, but the curvature of the function (or scalar field) *over* the surface. As shown above, the cavities of the molecular surface correspond to critical points of the scalar field outside the molecular surface.

Related Publications

The cavity detection algorithm described in this chapter was previously submitted for publication as a journal paper, as indicated below:

Sérgio Dias and Abel Gomes. A Scalar Field Topology-Based Method for the Detection of Protein Cavities. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (submitted for publication).

Chapter 6

Conclusions and Future Work

This thesis focuses on the geometric representation and detection of cavities on molecular surfaces of proteins. This chapter presents the main conclusions that have resulted from the research work described in this thesis. Furthermore, some issues will be pointed out for future work.

6.1 The Revisited Research Plan

Looking back to the traveled road of this doctoral programme, we see that the main milestones have been achieved, namely:

- *Molecular Surfaces and Electron Density Fields of Molecules.* Using the theory of scalar fields, we were able to come up with a unified mathematical formulation to approach both a geometric representation for molecular surfaces and a functional representation for electron density fields generated by the molecules themselves.
- *Molecular Triangulation Algorithm.* We were able to devise a number of GPU-based triangulation algorithms for distinct setups, including the one described in Chapter 3. This was of paramount importance to allow us to visualize and inspect molecules (and their cavities) as 3D geometric objects, as usual in computer graphics and geometric computing. We were even able to triangulate molecules with about 1 million atoms.
- *Intrinsic Shape Descriptors.* We were also able to advance intrinsic shape descriptors, i.e., those that are invariant to affine transformations, in order to capture the shape of molecules in terms of their cavities. This has originated a new class of cavity detection algorithms, as noted in Chapters 4-5.
- *Protein Cavity Detection Algorithms.* We were also able to design and implement cavity detection algorithms that take advantage of intrinsic shape descriptors. In particular, we have shown that there is a correspondence between critical points of the electron density field generated by a given molecule and its cavities.

Thus, it can be said that the research plan was fulfilled in general, including the publications in computer science journals and conference proceedings.

6.2 Final Conclusions

As seen above, this thesis presents a few contributions to the advance of the knowledge in molecular (computer) graphics, computational biology, computational chemistry, and bioinformatics. As explained in Chapter 5, the main contribution is the use of scalar field topology to determine cavities on molecular surfaces, but without using the surface itself. Nevertheless, as explained in Chapter 4, we also used isosurfaces to determine those cavities of molecules. It is clear that the algorithms described in Chapters 4-5 are the core of this thesis, though they are only the visible face of the contributions that led to the creation of a new category of cavity detection algorithms, i.e., those based on curvature. Recall that, in the past, curvature-based algorithms had not succeeded in detecting cavities on molecular surfaces. On the contrary, we have achieved such a goal using the topology of the electron density field, instead of the topology of the molecular surface itself. Of not less importance, it is the triangulation algorithm described in Chapter 3, which is a follow-up of the first triangulation algorithm to run entirely on GPU via CUDA (cf. [DBG10] for further details).

6.3 Future Work

As seen above, all the algorithms described in this thesis have made use of the voxelization of the domain. Nevertheless, in conformity with the work developed in [Gom14], it is plausible to find molecular cavities by means of minimization and maximization paths as usual in optimization problems. This is so because critical points are maxima, minima or saddles (i.e., maxima and minima at the same time).

Another line of research would be to track the topology of the electron density fields of two molecules (e.g., proteins) when they interact with one another; in particular, how the critical points mate with each other. This likely is of great importance in protein docking and protein engineering.

Yet another issue is related to transmutation of cavities in result of the interaction of molecules. Is that possible to know in advance when an open cavity (or pocket) becomes a void, and vice-versa, by looking at the topology of electron density fields of two interacting molecules?

Certainly, there will be many more questions to ask and respond, but truly speaking we have to stop at some point.

Bibliography

- [AE96] N. Akkiraju and H. Edelsbrunner. Triangulating the surface of a molecule. *Discrete Applied Mathematics*, 71(1-3):5-22, 1996. 21
- [AG03] E. Allgower and K. Georg. *Introduction to Numerical Continuation Methods*. SIAM's Classics in Applied Mathematics. SIAM Press, 2003. 21
- [Arl94] Sandra Arlinghaus. *Practical Handbook of Curve Fitting*. CRC Press, 1994. 56, 79, 80
- [ASP⁺08] L.-P. Albou, B. Schwarz, O. Poch, J. Wurtz, and D. Moras. Defining and characterizing protein surface using alpha shapes. *Proteins: Structure, Function, and Bioinformatics*, 76(1):1-12, 2008. 67
- [BAM⁺14] L. Benkaidali, F. André, B. Maouche, P. Siregar, M. Benyettou, F. Maurel, and M. Petitjean. Computing cavities, channels, pores and pockets in proteins from non spherical ligands models. *Bioinformatics*, 30(6):792-800, 2014. 46, 67
- [BBB⁺97] J. Bloomenthal, C. Bajaj, J. Blinn, M.-P. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, Inc., 1997. 21
- [BGG08] Mary Ellen Bock, Claudio Garutti, and Concettina Guerra. Cavity detection and matching for binding site recognition. *Theoretical Computer Science*, 408(2-3):151 - 162, 2008. 45
- [Bli82] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235-256, July 1982. xvi, 4, 23, 48, 68, 69
- [BNL03] T. Andrew Binkowski, Shapor Naghibzadeh, and Jie Liang. Castp: computed atlas of surface topography of proteins. *Nucleic Acids Res*, 31(13):3352-3355, 2003. 3, 46
- [BS91] Jules Bloomenthal and Ken Shoemake. Convolution surfaces. *Computer Graphics*, 25(4):251-256, 1991. 24
- [BS00] G. Patrick Brady and Pieter F. W. Stouten. Fast prediction and visualization of protein binding pockets with PASS. *Journal of Computer-Aided Molecular Design*, 14(4):383-401, may 2000. xxv, 16, 18, 44, 46, 59, 67, 81
- [CJvdP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007. 28
- [Con83a] M. Connolly. Analytical molecular surface calculation. *Journal of Applied Crystallography*, 16(5):548-558, October 1983. 16, 17

- [Con83b] M. Connolly. Analytical Molecular Surface Calculation. *Journal of Applied Crystallography*, 16(5):548-558, October 1983. 23
- [Con83c] M. Connolly. Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221(4612):709-713, August 1983. 23
- [Coo12] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing. Morgan Kaufmann, 2012. 49
- [CS06] Ryan G. Coleman and Kim A. Sharp. Travel depth, a new shape descriptor for macromolecules: Application to ligand binding. *Journal of Molecular Biology*, 362(3):441 - 458, 2006. 15, 45
- [CS10] R. Coleman and K. Sharp. Protein pockets: Inventory, shape, and comparison. *Journal of Chemical Information and Modeling*, 50(4):589-603, 2010. 15, 44, 66
- [DBG10] Sérgio Dias, Kuldeep Bora, and Abel Gomes. Cuda-based triangulations of convolution molecular surfaces. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 531-540, New York, NY, USA, 2010. ACM. 22, 23, 52, 54, 74, 79, 88
- [DCC⁺13] Sergio Decherchi, Jose; Colmenares, Chiara Eva Catalano, Michela Spagnuolo, Emil Alexov, and Walter Rocchia. Between algorithm and model: different molecular surface definitions for the Poisson-Boltzmann based electrostatic characterization of biomolecules in solution. *Communications in Computational Physics*, 13:61-89, 2013. 23
- [DCD⁺13] Daniele D'Agostino, Andrea Clematis, Sergio Decherchi, Walter Rocchia, Luciano Milanese, and Ivan Merelli. CUDA accelerated molecular surface generation. *Concurrency and Computation: Practice and Experience*, 2013. 22
- [DDG⁺12] Daniele D'Agostino, Sergio Decherchi, Antonella Galizia, José Colmenares, Alfonso Quarati, Walter Rocchia, and Andrea Clematis. CUDA Accelerated Blobby Molecular Surface Generation. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, volume 7203 of *Lecture Notes in Computer Science*, pages 347-356. Springer Berlin Heidelberg, 2012. 22
- [Del92] John S. Delaney. Finding and filling protein cavities using cellular logic operations. *Journal of Molecular Graphics*, 10(3):174 - 177, 1992. 45, 67
- [DG11] Sérgio E.D. Dias and Abel J.P. Gomes. Graphics processing unit-based triangulations of Blinn molecular surfaces. *Concurrency and Computation: Practice and Experience*, 23(17):2280-2291, 2011. 22, 71, 74

- [DG13] Sérgio Dias and Abel Gomes. Triangulating Molecular Surfaces on Multiple GPUs. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 181-186, New York, NY, USA, 2013. ACM. Available from: <http://doi.acm.org/10.1145/2488551.2488582>. 22
- [DG15] Sérgio Dias and Abel J.P. Gomes. Triangulating gaussian-like surfaces of molecules with millions of atoms. In Walter Rocchia and Michela Spagnuolo, editors, *Computational Electrostatics for Biological Applications*, pages 177-198. Springer International Publishing, 2015. 50
- [DLW08] B. H. Dessailly, M. F. Lensink, and S. J. Wodak. LigASite: a database of biologically relevant binding sites in proteins with known apo-structures. *Acid Nucleic Research*, 36:D667--673, 2008. 44, 53, 58, 59, 79, 81
- [DTS93] Carpio Del, Y Takahashi, and S Sasaki. A new approach to the automatic identification of candidates for ligand receptor sites in proteins: (i). search for pocket regions. *Journal of molecular graphics*, (11), 1993. 15
- [EA97] Saff E.B. and Kuijlaars A.B. Distributing many points on a sphere. *The Mathematical Intelligencer*, 19(1):5--11, 1997. 14
- [Ede98] H. Edelsbrunner. On the definition and the construction of pockets in macromolecules. *Discrete Applied Mathematics*, 88(1-3):83-102, November 1998. 46, 67
- [EFFL95] H. Edelsbrunner, M. Facello, Ping Fu, and Jie Liang. Measuring proteins and voids in proteins. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, HICSS '95, pages 256-, Washington, DC, USA, 1995. IEEE Computer Society. 11, 67
- [EKB02] T. Exner, M. Keil, and J. Brickmann. Pattern recognition strategies for molecular surfaces. i. pattern generation using fuzzy set theory. *Journal of Computational Chemistry*, 23(12):1176-1187, 2002. 66, 68
- [EM94] H. Edelsbrunner and E. P. Mucke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13:43-72, 1994. 3, 11, 21, 46, 67
- [Fee10] Timothy G. Feeman. *The Mathematics of Medical Imaging: A Beginner's Guide*. Undergraduate Texts in Mathematics and Technology. Springer, 2010. 24
- [Gei07] R. Geiss. Generating complex procedural terrains using the GPU. In H. Nguyen, editor, *GPU Gems 3*. Addison-Wesley Professional, New Jersey, USA, 2007. 22
- [Gom14] Abel J.P. Gomes. A continuation algorithm for planar implicit curves with singularities. *Computers & Graphics*, 38(1):365-373, 2014. xvii, 5, 72, 88

- [Goo85] P. J. Goodford. A computational procedure for determining energetically favorable binding sites on biologically important macromolecules. *Journal of Medicinal Chemistry*, 28(7):849-857, 1985. 10
- [GP95] J. A. Grant and B. T. Pickup. A gaussian description of molecular shape. *Journal of Physical Chemistry*, 99(11):3503-3510, 1995. 68
- [GT94] Kleywegt G.J. and Jones T.A. Detection, delineation, measurement and display of cavities in macromolecular structures. *Acta Crystallographica, Section D, Biological Crystallography*, 50, Parte 2:178--185, 1994. 13
- [GVJ⁺09] A.J.P. Gomes, I. Voiculescu, J. Jorge, B. Wyvill, and C. Galbraith. *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms*. Springer-Verlag, London, 2009. 21, 48
- [GW96] R. Gabdoulline and R. Wade. Analytically defined surfaces to analyze molecular interaction properties. *Journal of Molecular Graphics*, 14:341-353, 1996. 68
- [HH92] Charles D. Hansen and Paul Hinker. Massively parallel isosurface extraction. In *Proceedings of the 3rd conference on Visualization '92*, pages 77-83, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 22
- [HK06] Gerhard Muller Hugo Kubinyi. *Chemogenomics in Drug Discovery*. Wiley-VCH, 2006. 10
- [HM90] ChrisM.W. Ho and GarlandR. Marshall. Cavity search: An algorithm for the isolation and display of cavity-like binding regions. *Journal of Computer-Aided Molecular Design*, 4(4):337-354, 1990. 13, 46
- [HOG08] Rodney Harris, Arthur J. Olson, and David S. Goodsell. Automated prediction of ligand-binding sites in proteins. *Proteins: Structure, Function, and Bioinformatics*, 70(4):1506 - 1517, 2008. 10
- [HRB97] Manfred Hendlich, Friedrich Rippmann, and Gerhard Barnickel. Ligsite: automatic and efficient detection of potential small molecule-binding sites in proteins. *Journal of Molecular Graphics and Modelling*, 15(6):359 - 363, 1997. 10, 14, 44, 46, 59, 81
- [HSAH⁺09] Stefan Henrich, Outi M. H. Salo-Ahen, Bingding Huang, Friedrich F. Rippmann, Gabriele Cruciani, and Rebecca C. Wade. Computational approaches to identifying and characterizing protein binding sites for ligand design. *Journal of Molecular Recognition*, 23(2):209-219, 2009. 10, 65
- [HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum *Scan* with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison-Wesley Professional, Upper Saddle River, New Jersey, Aug 2007. 31

- [Hua09] Bingding Huang. Metapocket: A meta approach to improve protein ligand binding site prediction. *OMICS*, 13(4):325-330, 2009. xxvi, 52, 58, 59, 81
- [JC06] Gunnar Johansson and Hamish Carr. Accelerating marching cubes with graphics hardware. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '06*, Riverton, NJ, USA, 2006. IBM Corp. 22
- [KBO⁺82] I. D. Kuntz, J. M. Blaney, S. J. Oatley, R. Langridge, and T. E. Ferrin. A geometric approach to macromolecule-ligand interactions. *Journal of molecular biology*, 161(2):269-288, oct 1982. 46, 67
- [KC08] Yeturu Kalidas and Nagasuma Chandra. Pocketdepth: A new depth based algorithm for identification of ligand binding sites in proteins. *Journal of Structural Biology*, 161(1):31 - 42, 2008. 10, 15, 45
- [KCC⁺08] D. Kim, C. H. Cho, Y. Cho, J. Ryu, J. Bhak, and D. S. Kim. Pocket extraction on proteins via the Voronoi diagram of spheres. *Journal of molecular graphics & modelling*, 26(7):1104-1112, apr 2008. xxv, 12, 46
- [KCS⁺10] Deok-Soo Kim, Youngsong Cho, Kokichi Sugihara, Joonghyun Ryu, and Donguk Kim. Three-dimensional beta-shapes and beta-complexes via quasi-triangulation. *Computer-Aided Design*, 42(10):911-929, 2010. 21
- [KFR⁺11] M. Krone, M. Falk, S. Rehm, J. Pleiss, and T. Ertl. Interactive exploration of protein cavities. *Computer Graphics Forum*, 30(3):673--682, 2011. 45, 47
- [KG07] Takeshi Kawabata and Nobuhiro Go. Detection of pockets on protein surfaces using small and large probe spheres to find putative ligand binding sites. *Proteins: Structure, Function, and Bioinformatics*, 68(2):516-529, 2007. xiv, xxv, 2, 10, 17, 19, 44, 45, 46, 65, 66, 67
- [KMLJ07] A. Kahraman, R. Morris, R. Laskowski, and Thornton J. Shape variation in protein binding pockets and their ligands. *Journal of Molecular Biology*, 368(1):283-301, 2007. 43, 44, 67
- [KP02] Steven G. Krantz and Harold R. Parks. *A primer of real analytic functions*. Birkhäuser Advanced Texts. Birkhäuser Basel, 2 edition, 2002. 21
- [KRS⁺13] M. Krone, G. Reina, C. Schulz, T. Kulschewski, J. Pleiss, and T. Ertl. Interactive extraction and tracking of biomolecular surface features. *Computer Graphics Forum*, 32(3):331--340, 2013. 45
- [Las95] Roman A. Laskowski. Surfnet: A program for visualizing molecular surfaces, cavities, and intermolecular interactions. *Journal of Molecular Graphics*, 13(5):323 - 330, 1995. xxv, 2, 10, 16, 17, 44, 45, 46, 59, 67, 81

- [LB92] David G. Levitt and Leonard J. Banaszak. Pocket: A computer graphic method for identifying and displaying protein cavities and their surrounding amino acids. *Journal of Molecular Graphics*, 10(4):229 - 234, 1992. 2, 13, 45, 46, 67
- [LB12] Ling Wei Lee and Andrzej Bargiela. Prediction and modelling of ligand-binding sites using. In *ECMS 2012 Proceedings edited by: K. G. Troitzsch, M. Moehring, U. Lotzmann*, pages 96-102, May 2012. 45
- [LBH11] Norbert Lindow, Daniel Baum, and Hans-Christian Hege. Voronoi-based extraction and visualization of molecular paths. *IEEE Transactions on Visualization & Computer Graphics*, 17(12):2025--2034, 2011. 46
- [LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163-169, July 1987. 4, 22, 24, 71, 74
- [LERV⁺09] David La, Juan Esquivel-Rodríguez, Vishwesh Venkatraman, Bin Li, Lee Sael, Stephen Ueng, Steven Ahrendt, and Daisuke Kihara. 3D-SURFER: software for high-throughput protein surface comparison and analysis. *Bioinformatics*, 25(21):2843--2844, 2009. 45
- [Lev66] C. Levinthal. Molecular Model-building by Computer. *Scientific American*, 214(6):42-52, June 1966. 21
- [LGST09] Vincent Le Guilloux, Peter Schmidtke, and Pierre Tuffery. Fpocket: an open source platform for ligand pocket detection. *BMC bioinformatics*, 10(1):1-11, December 2009. 12, 44, 46, 59, 81
- [LJ05] Alasdair T. R. Laurie and Richard M. Jackson. Q-sitefinder: an energy-based method for the prediction of protein-ligand binding sites. *Bioinformatics*, 21(9):1908-1916, 2005. 10, 45, 67
- [LJ06] Alasdair T. Laurie and Richard M. Jackson. Methods for the prediction of protein-ligand binding sites for structure-based drug design and virtual ligand screening. *Current protein & peptide science*, 7(5):395-406, October 2006. xxv, 14, 16, 44
- [LLST96] Roman A Laskowski, NM Luscombe, Mark B Swindells, and Janet M Thornton. Protein clefts in molecular recognition and function. *Protein science: a publication of the Protein Society*, 5(12):2438, 1996. 9
- [LR71] B. Lee and F. Richards. The interpretation of protein structures: Estimation of static accessibility. *Journal of Molecular Biology*, 55(3):379-380, February 1971. 23
- [LTA⁺08] Bin Li, Srinivasan Turuvekere, Manish Agrawal, David La, Karthik Ramani, and Daisuke Kihara. Characterization of local geometry of protein surfaces

- with the visibility criterion. *Proteins: Structure, Function, and Bioinformatics*, 71(2):670-683, 2008. 14, 45
- [LWE98] Jie Liang, Clare Woodward, and Herbert Edelsbrunner. Anatomy of protein pockets and cavities: Measurement of binding site geometry and implications for ligand design. *Protein Science*, 7(9):1884-1897, 1998. xxv, 9, 11, 12, 46, 67
- [Mac92] Paul Mackerras. A Fast Parallel Marching-Cubes Implementation on the Fujitsu AP1000. Technical Report TR-CS-92-10, Department of Computer Science, The Australian National University, 1992. 22
- [McL74] D. McLain. Drawing contours from arbitrary data points. *The Computer Journal*, 17(4):318-324, 1974. 48
- [MD95] M. Masuya and J. Doi. Detection and geometric modeling of molecular surfaces and cavities using digital mathematical morphology operations. *J.Mol. Graph. Model*, 13:331-336, 1995. 45, 67
- [MDG⁺10] L. Marsalek, A.K. Dehof, I. Georgiev, H.-P. Lenhof, P. Slusallek, and A. Hildebrandt. Real-time ray tracing of complex molecular scenes. In *Proceedings of the 14th International Conference on Information Visualisation (IV'10)*, pages 239-245, London, UK, July 26-29, 2010. IEEE Press. 21
- [Mez93] P G Mezey. *Shape in Chemistry: An Introduction to Molecular Shape and Topology*. Wiley-VCH, August 1993. 65
- [MSPSI14] Simon McIntosh-Smith, James Price, Richard B. Sessions, and Amaury A. Ibarra. High performance in silico virtual drug screening on many-core processors. *International Journal of High Performance Computing Applications*, 29(2):119--134, 2014. 47
- [NH06] Murad Nayal and Barry Honig. On the nature of cavities on protein surfaces: Application to the identification of drug-binding sites. *Proteins*, 63(4):892-906, February 2006. 44, 46, 67
- [NSG12] Britta Nisius, Fan Sha, and Holger Gohlke. Structure-based computational analysis of protein binding sites for function and druggability prediction. *Journal of Biotechnology*, 159(3):123-134, 2012. 66
- [NWB⁺06] V. Natarajan, Y. Wang, P.-T. Bremer, V. Pascucci, and B. Hamann. Segmenting molecular surfaces. *Computer Aided Geometric Design*, 23(6):495-509, 2006. 66, 68
- [NY06] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854-879, oct 2006. 22

- [OF03] Stanley Osher and Ronald Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Applied Mathematical Sciences, Vol. 153. Springer-Verlag, 2003. 48
- [PCY14] Yi Peng, Li Chen, and Jun-Hai Yong. Importance-driven isosurface decimation for visualization of large simulation data based on opencl. *Computing in Science Engineering*, 16(1):24-32, Jan 2014. 22
- [PFF96] Klaus P. Peters, Jana Fauck, and Cornelius Frommel. The automatic search for ligand binding sites in proteins of known three-dimensional structure using only geometric criteria. *J. Mol. Biol*, 256:201-213, 1996. 3, 11, 46, 67
- [PMMA11] L. Petrescu, A. Morar, F. Moldoveanu, and V. Asavei. Real time reconstruction of volumes from very large datasets using CUDA. In *Proceedings of the 15th International Conference on System Theory, Control, and Computing (ICSTCC'11)*, pages 1-5, Sinaia, Romania, October 14-16, October 2011. IEEE Press. 22
- [PSM⁺10] Stéphanie Pérot, Olivier Sperandio, Maria A Miteva, Anne-Claude Camproux, Bruno O Villoutreix, et al. Druggable pockets and binding site centric chemical space: a paradigm shift in drug discovery. *Drug discovery today*, 15(15-16):656, 2010. 44, 67
- [PTRV12] J. Parulek, C. Turkay, N. Reuter, and I. Viola. Implicit surfaces for interactive graph based cavity analysis of molecular simulations. In *Proceedings of the 2012 IEEE Symposium on Biological Data Visualization (BioVis'2012)*, pages 115-122. IEEE Press, Oct 2012. 47
- [Ric77] F. Richards. Areas, Volumes, Packing, and Protein Structure. *Annual Review of Biophysics and Bioengineering*, 6(3):151-176, February 1977. 23
- [RWJ97] Jim Ruppert, Will Welch, and Ajay N. Jain. Automatic identification and representation of protein binding sites for molecular docking. *Protein Science*, 6(3):524-533, 1997. 46, 67
- [SBCLB11] Peter Schmidtke, Axel Bidon-Chanal, F. Javier Luque, and Xavier Barril. MDpocket: open-source cavity detection and characterization on molecular dynamics trajectories. *Bioinformatics*, 27(23):3276--3285, 2011. 46
- [SDP⁺13] Raghavendra Sridharamurthy, Harish Doraiswamy, Siddharth Patel, Raghavan Varadarajan, and Vijay Natarajan. Extraction of Robust Voids and Pockets in Proteins. In *Proceedings of the EuroVis - Short Papers*, pages 67-71, Leipzig, Germany, 2013. Eurographics Association. 11, 46
- [SG02] Peter D. Sulatycke and Kanad Ghose. Multithreaded isosurface rendering on smps using span-space buckets. In *Proceedings of the 2002 International Conference on Parallel Processing*, Washington, DC, USA, 2002. IEEE Computer Society. 22

- [SH10] Bharat Sukhwani and Martin C. Herbordt. Fast binding site mapping using GPUs and CUDA. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW'2010)*, volume 27, pages 1--8. IEEE Press, 2010. 47
- [SHF⁺06] R. Smith, L. Hu, J. Falkner, M. Benson, J. Nerothin, and H. Carlson. Exploring protein-ligand recognition with binding moad. *Journal of Molecular Graphics and Modeling*, 24(6):283-301, 2006. 44, 67
- [SSE⁺10] Peter Schmidtke, Catherine Souaille, Frédéric Estienne, Nicolas Baurin, and Romano T. Kroemer. Large-scale comparison of four binding site detection algorithms. *Journal of Chemical Information and Modeling*, 50(12):2191-2200, 2010. 66
- [TBK⁺91] Pedersen T.G., Sigurskjold B.W., Andersen K.V., Kjaer M., Poulsen F.M., Dobson C.M., and Redfield C. A nuclear magnetic resonance study of the hydrogen-exchange behaviour of lysozyme in crystals and solution. *Journal of Molecular Biology*, 218(2):413--426, 1991. 15
- [TDCL09] Yan Yuan Tseng, Craig Dupree, Z. Jeffrey Chen, and Wen-Hsiung Li. Split-pocket: identification of protein functional surfaces and characterization of their spatial patterns. *Nucleic Acids Research*, 37:W384-W389, 2009. 11, 46
- [TK10] Ashutosh Tripathi and Glen E. Kellogg. A novel and efficient tool for locating and characterizing protein cavities and binding sites. *Proteins: Structure, Function, and Bioinformatics*, 78(4):825-842, 2010. 14, 45
- [Ura06] Y. Uralsky. DX 10: Pratical metaballs and implicit surfaces. In *Game Developers Conference*, 2006. 22
- [VGGR10] Andrea Volkamer, Axel Griewel, Thomas Grombacher, and Matthias Rarey. Analyzing the Topology of Active Sites: On the Prediction of Pockets and Subpockets. *Journal of Chemical Information and Modeling*, 50(11):2041-2052, November 2010. 2, 45, 46
- [VH97] Y. Vorobjev and J. Hermans. SIMS: Computation of a Smooth Invariant Molecular Surface. *Biophysical Journal*, 73(2):722-732, 1997. 23
- [VJOW03] C.M. Venkatachalam, X. Jiang, T. Oldfield, and M. Waldman. Ligandfit: a novel method for the shape-directed rapid docking of ligands to protein active sites. *Journal of Molecular Graphics and Modelling*, 21(4):289 - 307, 2003. 45, 67
- [VKV⁺89] R Voorintholt, M T Kusters, G Vegter, G Vriend, and W G Hol. A very fast program for visualizing protein surfaces, channels and cavities. *J Mol Graph*, 7(4):243-5, 1989. 13

- [Wen95] H. Wendland. Piecewise polynomial, positive definite and compactly supported radial basis functions of minimal degree. *Advances in Computational Mathematics*, 4(1):389-396, 1995. 48
- [WJV07] Qin Wang, Joseph JaJa, and Amitabh Varshney. An efficient and scalable parallel algorithm for out-of-core isosurface extraction and rendering. *J. Parallel Distrib. Comput.*, 67(5):592-603, 2007. 22
- [WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2(4):227-234, February 1986. 23
- [WPS07] Martin Weisel, Ewgenij Proschak, and Gisbert Schneider. Pocketpicker: analysis of ligand binding-sites with shape descriptors. *Chemistry Central Journal*, 1(1):7, 2007. xxv, 11, 14, 18, 45, 67
- [WYCS13] Kai Wang, Yanzhi Yang, John D. Chodera, and Michael R. Shirts. Identifying ligand binding sites and poses using GPU-accelerated Hamiltonian replica exchange molecular dynamics. *Journal of Computer-Aided Molecular Design*, 27(12):989--1007, 2013. 47
- [XB07] Lei Xie and PhilipE Bourne. A robust and efficient algorithm for the shape description of protein structures and its application in predicting ligand binding sites. *BMC Bioinformatics*, 8(Suppl 4), 2007. 12, 46
- [XZ09] Dong Xu and Yang Zhang. Generating triangulated macromolecular surfaces by euclidean distance transform. *PLoS ONE*, 4(12), December 2009. 21, 46
- [YFW⁺08] Eitan Yaffe, Dan Fishelovitch, Haim J. Wolfson, Dan Halperin, and Ruth Nussinov. MolAxis: efficient and accurate identification of channels in macromolecules. *Proteins*, 73(1):72-86, oct 2008. 11
- [YZTY10] J. Yu, Y. Zhou, I. Tanaka, and M. Yao. Roll: a new algorithm for the detection of protein pockets and cavities with a rolling probe sphere. *Bioinformatics*, 26(1):46-52, 2010. 17, 44, 46, 59, 67, 81
- [ZB07] X. Zhang and C. Bajaj. Extraction, quantification and visualization of protein pockets. *Comput Syst Bioinformatics Conf*, 6, 2007. 13, 45
- [ZHSB92] Carl-Dieter Zachmann, Wolfgang Heiden, Micheal Schlenkrich, and Jürgen Brickmann. Topological analysis of complex molecular surfaces. *Journal of Computational Chemistry*, 13(1):76-84, 1992. 68
- [ZN04] Huijuan Zhang and T.S. Newman. Span space data structures for multi-threaded isosurfacing. In *SoutheastCon, 2004. Proceedings. IEEE*, pages 290-296, 2004. 22
- [ZXB06] Y. Zhang, G. Xu, and C. Bajaj. Quality meshing of implicit solvation models of biomolecular structures. *Computer Aided Geometric Design*, 23(6):510-530, 2006. 48, 68