Visualisierungsinstitut der Universität Stuttgart
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3199

Analyse der Oberflächenstruktur von Proteinen

Christoph Schulz

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Thomas Ertl
Betreuer:	Dipl.-Inf. Michael Krone
begonnen am:	8. Juni 2011
beendet am:	8. Dezember 2011
CR-Klassifikation:	I.3.3, I.3.5, I.3.7, I.4.6, I.4.8, J.3

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Diese Diplomarbeit befasst sich mit dem Thema *Analyse der Oberflächenstruktur von Proteinen* auf moderner Grafikhardware. Das Ziel der Diplomarbeit ist es die technische Machbarkeit des interaktiven Umgangs mit zeitabhängigen Datensätzen aus Molekulardynamik-Simulationen zu zeigen.

Das Wort „Protein“ ist aus dem griechischen *proteis* (deutsch: grundlegend) abgeleitet und bezeichnet einen der Grundbausteine des Lebens. Proteine erfüllen viele verschiedene Aufgaben, wie beispielsweise das Regeln, Steuern und Katalysieren von Reaktionen. Sie dienen als Bausteine, Strukturelemente und Träger von Reaktionspartnern in unserem Körper. Ohne Proteine gäbe es kein Leben, wie wir es kennen. Wir Menschen benötigen Modelle, die auf unsere kognitiven Fähigkeiten zugeschnitten sind, um Proteine begreifen zu können. Proteinoberflächen sind Modelle, die visuell betrachtet werden können, Aussagen über die räumliche Ausdehnung ermöglichen und Rückschlüsse auf Vorgänge im Umfeld und im Inneren eines Proteins zulassen.

1.1 Aufgabenstellung

Im Rahmen des laufenden Sonderforschungsbereichs 716 der Universität Stuttgart beschäftigt sich der Teilbereich D4 mit der Visualisierung von Molekulardynamik-Simulationen, insbesondere mit der Extraktion und Darstellung komplexer Eigenschaften von Protein-Lösungsmittel-Systemen. In vorausgegangenen Arbeiten am Visualisierungsinstitut wurde unter anderem ein Algorithmus zur Berechnung einer volumetrischen Repräsentation von Molekülen entwickelt. Aus diesen Volumen können mittels Ray Casting (bzw. Ray Marching) Isoflächen extrahiert und dargestellt werden. Diese Technik wurde in einer vorangegangen Arbeit verwendet, um Hohlräume in Proteinen zu erkennen und mittels Segmentierung des Volumens zu extrahieren [KFR+11]. Basierend auf der volumetrischen Repräsentation des Moleküls soll nun eine triangulierte Oberfläche erstellt und anschließend für folgende Analyseschritte verwendet werden:

1. Das Finden zusammenhängender Komponenten, um die Außenhülle von Hohlräumen unterscheiden zu können,
2. die Korrelation der Komponenten über mehrere Zeitschritte und
3. die Klassifizierung von Vorgängen wie Amalgamieren und Aufspalten.

Die Ergebnisse müssen in geeigneter Form grafisch dargestellt werden.

1.2 Arbeitsthesen

Folgende Überlegungen wurden vor der Arbeit angestellt, um abzuschätzen, mit welchen Ergebnissen gerechnet werden kann.

Interessante Oberflächenänderungen sind schwer zu entdecken Proteinoberflächen sind sehr komplex und Oberflächenänderungen wenig intuitiv. Beim Betrachten des Proteins kann leicht etwas Wichtiges verdeckt werden. Damit Oberflächenänderungen nicht unentdeckt bleiben, muss der Betrachter auf interessante Bereiche aufmerksam gemacht werden.

Mit einem Dreiecksnetz arbeiten bringt Vorteile Algorithmen für Dreiecksnetze sind sehr verbreitet und ausgereift. Außerdem könnte eine Oberflächenbeschreibung aus Dreiecken weniger Speicher benötigen als eine Volumenbeschreibung.

GPU-Programmierung unterscheidet sich von CPU-Programmierung GPUs unterscheiden sich in ihrer Architektur deutlich von CPUs (mehr Kerne, andere Speicherhierarchie). Bei der Wahl der Algorithmen und Optimierungen muss daher die Hardware entsprechend berücksichtigt werden.

1.3 Zeitplan

Für den Zeitraum der Diplomarbeit wurde ein Zeitplan aufgestellt. Eine Technologieeinarbeitungsphase entfällt, da im Rahmen der Themensuche bereits eine erste Auseinandersetzung mit der Technologie stattfand. Der Zeitplan (siehe Abbildung 1.1) ist in folgende Phasen unterteilt:

Recherche Zu Beginn sollen bisherige und thematisch verwandte Veröffentlichungen auf diesem Gebiet gesucht und beurteilt sowie eine inhaltliche Struktur der Diplomarbeit erstellt und mögliche Lösungswege skizziert werden.

Iteration 1 In einer ersten Iteration wird zunächst eine zusammenhangslose Oberfläche aus Dreiecken mittels CUDA berechnet und ohne grafische Ausgabe in das Plug-in MegaMol Protein (siehe Seite 31) integriert.

Iteration 2 In einer zweiten Iteration werden eine grafische Ausgabe implementiert und Proteinanimationen unterstützt.

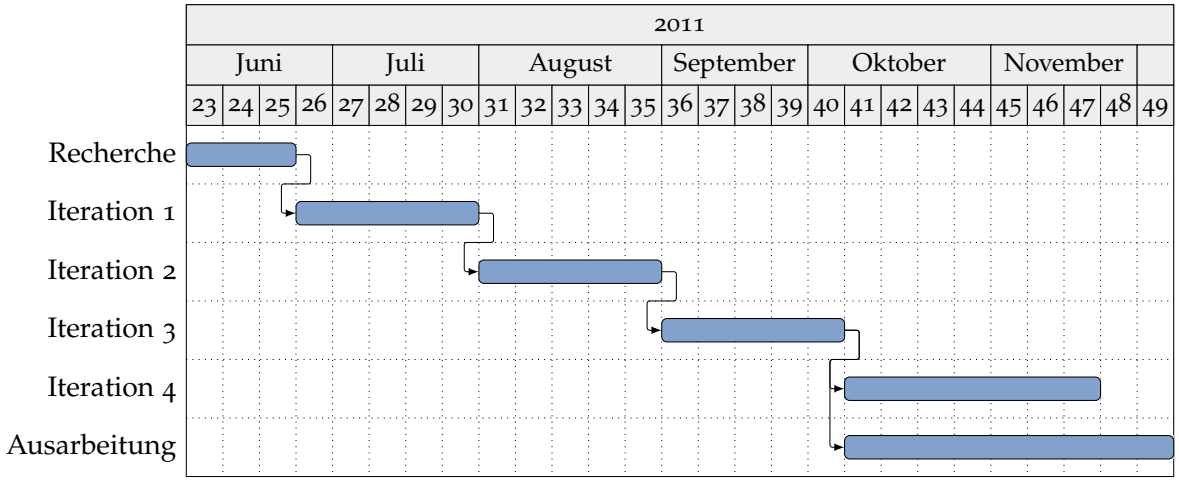


Abbildung 1.1: Gantt-Diagramm des Zeitplans.

Iteration 3 In einer dritten Iteration werden zusammenhängende Komponenten erkannt und zeitlich korreliert.

Iteration 4 In einer vierten Iteration werden zu erkennende Vorgänge grafisch hervorgehoben und Daten für diese Ausarbeitung gesammelt.

Ausarbeitung Abschließend wird diese Ausarbeitung erstellt. Ein bis zwei Wochen sind als Puffer für eine abschließende Kontrolle und Korrekturen eingeplant.

2 Grundlagen

In diesem Kapitel werden die Grundlagen zu dieser Arbeit behandelt. Hier werden die später als bekannt vorausgesetzten Begriffe wie Protein, Objekterkennung, Parallelrechner und die verwendeten Technologien beschrieben.

2.1 Proteine

Im deutschen Sprachraum sind Proteine bekannt als Eiweiße. Der in der Fachliteratur gängige Oberbegriff ist Protein. Proteine sind kettenartige, aus Aminosäuren aufgebaute Moleküle. Das kleinste bekannte Protein heißt Chignolin [SHI05] und besteht aus gerade einmal zehn Aminosäuren. Das größte bekannte Protein heißt Titin [tit11]. Es befindet sich in unseren Muskeln und besteht - je nach Isoform - aus über 30.000 Aminosäuren. Proteine haben häufig ein aktives Zentrum, das mit einem Substrat interagiert. Das Substrat wird dem aktiven Zentrum durch Tunnel, Taschen und Einschlüsse zugeführt.

2.1.1 Bedeutung

Proteine erfüllen unterschiedliche Funktionen: Manche Proteine dienen schlicht als Baustoff oder sind Träger von Reaktionspartnern. Andere dienen als Bindungspartner, um die Anwesenheit von Stoffen anzuzeigen oder als Katalysator für chemische Reaktionen. Das Anzeigen von Stoffen wird in diversen Regelkreisen im Körper eingesetzt, um Vorgänge wie den Transport von Stoffen in die Zelle zu hemmen oder zu fördern. In der Medizin wird diese Anwendung zur Diagnose verwendet. Proteine, die chemische Reaktionen katalysieren, nennt man Enzyme. Ohne Enzyme wären viele Reaktionen bei Körpertemperatur nicht möglich.

2.1.2 Abhängigkeiten

Proteine sind stark zeit- und umgebungsabhängig. Es spielt eine große Rolle, ob ein Protein angelagert oder gelöst ist, welche Fremdatome es einlagert und in welcher Konfiguration es sich bei einer bestimmten Umgebungstemperatur befindet. All diese Faktoren haben einen Einfluss auf die Funktion eines Proteins. Ein bekannteres Beispiel ist die Bleivergiftung: Bei einer Bleivergiftung werden unter anderem die an der Blutbildung beteiligten Enzyme durch die Anwesenheit von Bleiionen gehemmt, was bei entsprechender Dosis zu Blutarmut, schlecht heilenden Wunden und zum Tod führt.

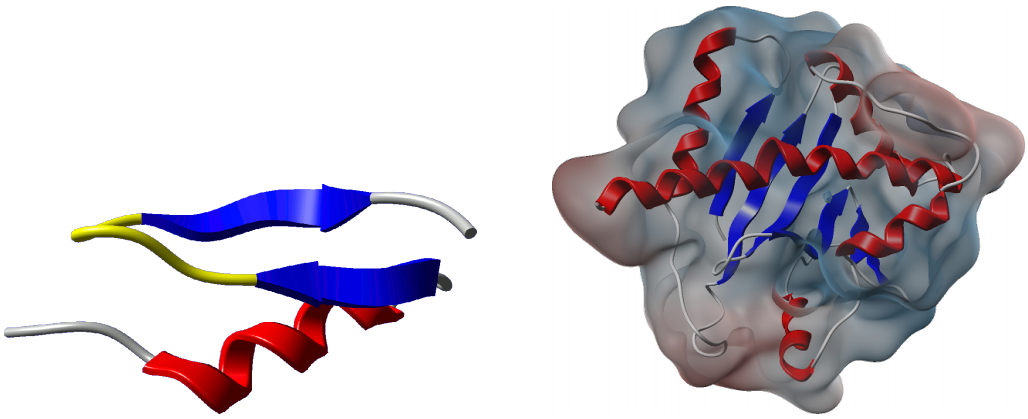


Abbildung 2.1: Sekundärstrukturelemente (links [KBE08]) α-Helix (rot) und β-Faltblatt (blau) sowie die Tertiärstruktur (rechts [KFR+11]) eines Proteins.

2.1.3 Strukturebenen

Im Jahr 1952 wurde von Kaj Ulrik Linderstrøm-Lang [LL52] eine Betrachtung von Proteinen auf den folgenden vier Strukturebenen (vergleiche Abbildung 2.1) vorgeschlagen:

Primärstruktur Die Primärstruktur beschreibt die Sequenz der Aminosäuren, aus der ein Protein aufgebaut ist. Aus der Primärstruktur werden alle weiteren Strukturebenen abgeleitet.

Sekundärstruktur Die Sekundärstruktur beschreibt in der Primärstruktur auftretende Muster, wie *α-Helix*, *β-Faltblatt* und *β-Schleifen*.

Tertiärstruktur Die Tertiärsturktur beschreibt die räumliche Anordnung, also auch die Oberfläche des Proteins.

Quartärstruktur Die Quartärstruktur beschreibt einen Proteinkomplex, also das Zusammenspiel mehrerer Proteine.

2.2 Objekterkennung

Objekterkennung lässt sich als Pipeline untergliedern, wie in Abbildung 2.2 dargestellt. Ziel der Objekterkennung ist es, aus (Bild-)Daten verwertbare Informationen in Form von Aussagen zu gewinnen. In einem ersten Schritt werden die Daten vorverarbeitet, um Messfehler oder Rauschen zu entfernen. Bekannte Beispiele für die Vorverarbeitung sind das

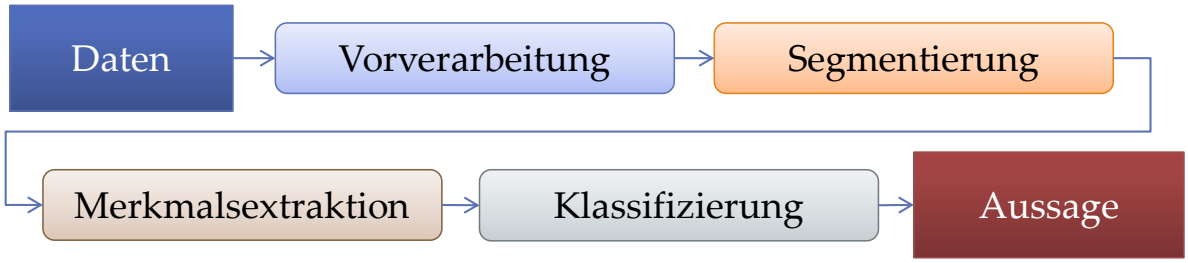


Abbildung 2.2: Objekterkennungspipeline.

Entfernen defekter Pixel mit Hilfe eines Median-Filters oder das Stabilisieren von aus der Bewegung heraus aufgenommenen Bildern. Anschließend werden die Daten mit Hilfe eines Homogenitätskriteriums segmentiert, was die Menge der Daten reduziert und so nachfolgende Arbeitsschritte vereinfacht. Ein typisches Beispiel für die Segmentierung ist das Unterscheiden von Papier und Text bei der Texterkennung. Abschließend werden Merkmale aus den Segmenten extrahiert und klassifiziert, sodass eine oder mehrere Aussagen getroffen werden können. Bei der Texterkennung werden Merkmale wie beispielsweise Schriftgröße sowie Schriftart extrahiert und zu einem Text, meist mit Hilfe eines Wörterbuches, klassifiziert. In der Praxis ist die Grenze zwischen Merkmalsextraktion und Klassifizierung oftmals fließend.

2.3 Parallelrechner

Software wird traditionell für sequenziell arbeitende Computer geschrieben. Lange dachte man, dass dies kein Problem sei, denn das von Gordon Moore formulierte und bisher gültige Mooresche Gesetz (1965/1975) sagt im Wesentlichen voraus, dass sich die Rechenleistung eines Computers alle zwei Jahre verdoppelt. Wie durch Asanovic et al. [ABC+06] berichtet, war bis kurz nach der Jahrtausendwende der Ursprung für eine Leistungsverdopplung eine höhere Taktrate oder die Verkleinerung von Schaltkreisen. Es wurden zwei, durch physikalische Randbedingungen bedingte Grenzen erreicht:

1. Taktverteilung: Man konnte einen Takt nicht mehr über den ganzen Chip verteilen, was die Chiparchitektur erheblich verkomplizierte.
2. Energiedichte: Man konnte so viele Transistoren auf einem Chip platzieren, dass die auf dem Chip verteilbare Energie nicht ausreichte, um diese zu schalten.

Um den Forderungen nach leistungsstärkeren Rechnern nachzukommen, hatte die Chipindustrie folgende Lösungen in Aussicht gestellt: Parallelrechner und Niedrig-Energie-Transistoren.

Parallele Programmierung war zu Beginn des Jahrtausends nicht neu, allerdings auch nicht sonderlich verbreitet. Klassische Programmiersprachen wie C/C++ sind auf sequenzielle Programmierung ausgelegt, weshalb auch parallelisierbare Probleme, begünstigt durch die

Programmiersprache, sequenziell formuliert wurden. Um von dem Paradigmenwechsel der Chipindustrie zu profitieren, muss in der Folge sequenziell arbeitende Software zu großen Teilen neu geschrieben werden. Nach dem Amdahlschen Gesetz (Formel 2.1) ist der zu erwartende Geschwindigkeitszuwachs S durch den sequenziellen Anteil $(1 - P)$ des Problems beschränkt, wobei N dem Grad der Parallelität (Anzahl der Prozessoren) entspricht.

$$\mathrm{S} = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

Vereinfacht ausgedrückt lässt sich ein Problem besonders gut parallel lösen, wenn die Teilprobleme möglichst unabhängig voneinander sind. Compiler und Programmiersprachen für parallel arbeitende Rechnerarchitekturen sind Gegenstand der Forschung. Nur wenige ‚parallele Hochsprachen', wie Erlang und Scala, haben es in die Industrie geschafft. Entsprechend benötigt der Programmierer mehr Wissen über die Problemstellung und Hardware, was die Entwicklung teurer und hardwarespezifischer gestaltet. Im nachfolgenden Abschnitt wird ein Überblick über die Klassifikation paralleler Rechnerarchitekturen vermittelt.

2.3.1 Flynnsche Taxonomie

Die Flynnschen Taxonomie ist eine Unterteilung von Rechnerarchitekturen anhand der Befehls- und Datenströme, was im Hinblick auf Hochsprachen und Compiler-Heuristiken interessant ist:

Single Instruction Single Data (SISD) bezeichnet eine klassische, sequenziell arbeitende Rechnerarchitektur. Dabei wird mit einem Befehl ein Datenstrom verarbeitet.

Single Instruction Multiple Data (SIMD) bezeichnet eine in der Praxis häufig vorkommende parallele Rechnerarchitektur. Dabei werden mit einem Befehl mehrere Datenströme parallel verarbeitet. Diese Art der Parallelität ist besonders leicht in Hardware zu implementieren, da lediglich der datenverarbeitende Teil vervielfältigt werden muss.

Multiple Instruction Single Data (MISD) ist eine exotische, redundante Form der Parallelität. Dabei wird mit mehreren Befehlen der gleiche Datenstrom verarbeitet.

Multiple Instruction Multiple Data (MIMD) ist eine häufig bei Supercomputern angewandte Rechnerarchitektur. Dabei werden mit mehreren Befehlen mehrere unterschiedliche Datenströme verarbeitet.

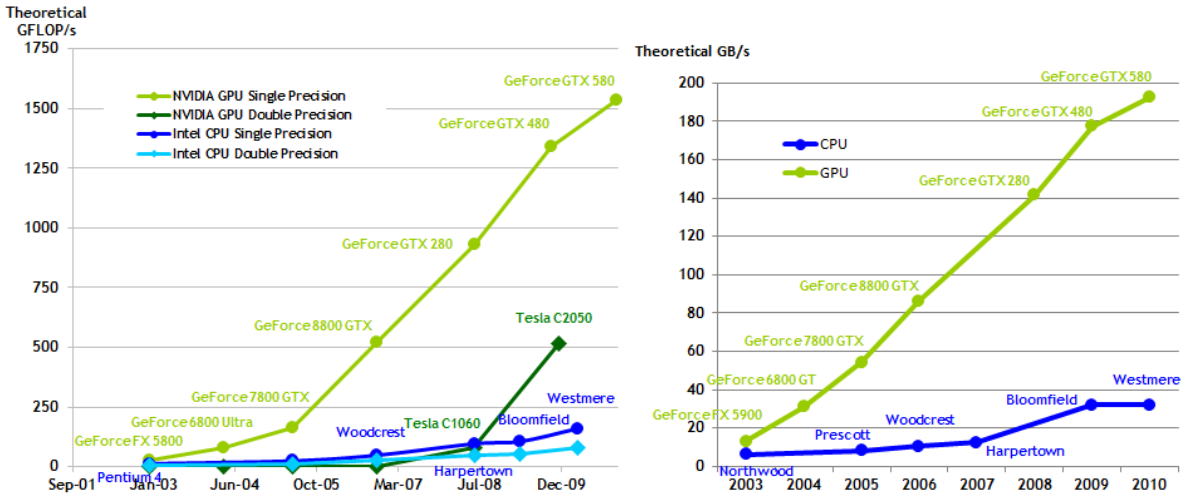


Abbildung 2.3: Entwicklungsverlauf von GPU und CPU anhand der Rechenoperationen und Speicherbandbreite [Nvi].

2.3.2 Speicherarchitektur

Die Flynnsche Taxonomie nimmt lediglich an, dass Datenströme zeitlich geordnet und zugänglich sind. Deshalb kann anhand der Speicherarchitektur weiter unterschieden werden:

Shared Memory bezeichnet eine Verteilung des Speichers, bei der jede Recheneinheit Zugriff auf den gesamten Speicher hat. Es existiert ein gemeinsamer Speicherbus. Man kann anhand der Zugriffszeiten weiter unterscheiden in Uniform Memory Access (UMA) und Non-Uniform Memory Access (NUMA).

Distributed Memory bezeichnet eine Verteilung des Speichers, bei der jede Recheneinheit nur auf den lokalen Speicher Zugriff hat. Es existiert kein gemeinsamer Speicherbus.

2.4 Graphics Processing Units

Graphics Processing Units (GPUs) waren ursprünglich Spezialprozessoren für die Film- und Computerspieleindustrie, die für das Rendering von Computergrafik entwickelt wurden. Computergrafik wurde in ihren Anfängen auf speziellen Supercomputern berechnet. Nach und nach wurden dann Grafikkarten entwickelt, die dieses Problem besser und günstiger lösen konnten. Abbildung 2.3 zeigt, dass moderne GPUs erheblich leistungsfähiger sind als entsprechende CPUs. Die Ursache hierfür liegt in den unterschiedlichen Entwurfszielen: CPUs unterstützen verschiedene Datentypen und sind auf sequenzielles Verarbeiten, Branching sowie Random-Memory-Access optimiert. GPUs hingegen sind dediziert für

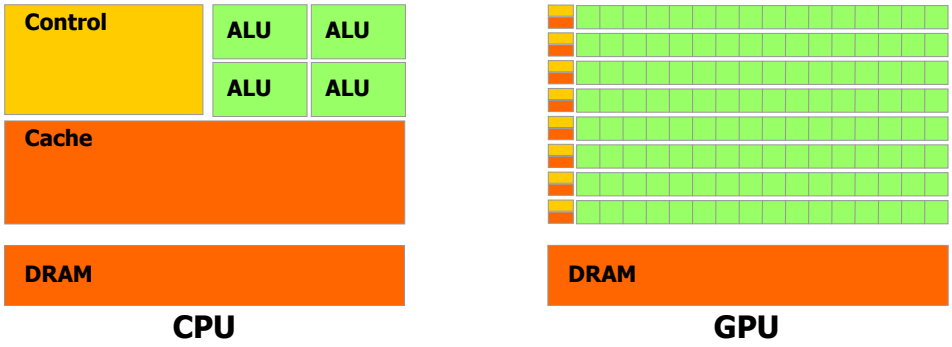


Abbildung 2.4: Größenverhältnisse der funktionalen Einheiten von GPU und CPU [Nvi].

Computergrafik entworfen, die zu großen Teilen parallel berechnet werden kann. Mit steigender Chipgröße wächst die Fehlproduktion und damit der Preis; also ist bei gleichem Preis die Anzahl an Transistoren pro Quadratzentimeter limitiert. Abbildung 2.4 zeigt die deutlichen Unterschiede im Entwurf: Transistoren, die bei einer CPU für Cachelogik und Kontrolllogik eingesetzt werden können, bei einer GPU für Arithmetisch-logische Einheiten (ALU) verwendet werden.

Die ersten Grafikkarten hatten eine sogenannte ‚Fixed Function Pipeline', bei der man nur einige Parameter für den Rendering-Vorgang festlegen konnte. Im Wettlauf um bessere Grafikeffekte, besonders in Computerspielen, beseitigten die Grafikkartenhersteller immer mehr Restriktionen was die Programmierbarkeit von GPUs betrifft. Durch die neu gewonnenen Freiheiten entstand letztlich ein neuer Markt, das sogenannte General Purpose Computation on Graphics Processing Units (GPGPU). Unter GPGPU versteht man das Lösen von nicht mit der Computergrafik verwandten Problemen mit Hilfe einer GPU. GPUs treten damit direkt in Konkurrenz zu klassischen Supercomputern. Allerdings sind GPUs deutlich günstiger (Massenprodukt) und verbrauchen erheblich weniger Energie (integrierter Schaltkreis) als ein Supercomputer. Das macht GPUs zu einem interessanten Werkzeug für Wissenschaft und Industrie.

2.4.1 NVIDIA GeForce

GeForce ist ein Markenname des Grafikkartenherstellers NVIDIA. Als Hardwareplattform wurde für diese Diplomarbeit die GeForce-400-Serie bzw. die GeForce-500-Serie vereinbart. Beide Serien basieren auf NVIDIAs Fermi-Architektur. Im Gegensatz zu älteren Serien unterstützt die Fermi-Architektur CUDA Compute Capability 2.0, die mehr atomare Operationen und Synchronisationsmechanismen bietet [Nvi].

In Tabelle 2.1 sind technische Merkmale der verwendeten GeForce-Karten aufgelistet. Bei der Fermi-Architektur werden mehrere skalare Streamprozessoren, namens CUDA Cores, zu Clustern, namens Streaming Multiprocessors, gruppiert.

	GeForce 480 GTX	GeForce 580 GTX
CUDA Compute Capability	2.0	2.0
Streaming Multiprocessor	15	16
CUDA Cores	480	512
Grafiktakt	700 MHz	772 MHz
Prozessortakt	1401 MHz	1544 MHz
Speicher	1536 MB	1536 MB
Speichertyp	GDDR5	GDDR5
Speichertakt	1848 MHz	2004 MHz
Speicherbandbreite	177,4 GB/s	192,4 GB/s
Speicherschnittstelle	384 Bit	384 Bit

Tabelle 2.1: Technische Daten der GeForce 480 und GeForce 580 [Cor]

2.4.2 NVIDIA CUDA

Bei NVIDIAs CUDA handelt es sich um eine Technik zur Programmierung von GPUs und ist ein Akronym für Compute Unified Device Architecture. CUDA ist, im Gegensatz zu den für Grafikanwendungen entworfenen Shadersprachen GLSL und HLSL, eine auf C/C++ aufbauende Programmiersprache. Viele Informationen in diesem Abschnitt wurden aus dem *NVIDIA CUDA C Programming Guide* [Nvi] entnommen.

Programmiermodell

Das CUDA Programmiermodell kann als Erweiterung von SIMD gesehen werden und wird auch als Single Program Multiple Data (SPMD) bezeichnet. Dabei werden mit einem Programm namens CUDA Kernel mehrere Datenströme verarbeitet. SPMD unterstützt, im Gegensatz zu SIMD, komplexe Berechnungen und Kontrollfluss. Ein Kernel enthält Befehle, die im Kontext eines CUDA Threads ausgeführt werden. Threads werden zu Blocks zusammengefasst und Blocks zu einem Grid (siehe Abbildung 2.5).

Diese Hierarchie ist für die Abbildung des Programms auf die Hardware wichtig. Jeder Thread wird parallel von einem CUDA Core verarbeitet, der in einem Streaming Multiprocessor mit anderen Streamprozessoren gebündelt ist. CUDA Cores teilen sich einen schnellen Speicher. Ein Block ist unabhängig von anderen Blocks. Die Ausführungsreihenfolge der Threads innerhalb eines Blocks kann nicht kontrolliert werden. Blocks werden nach Block-ID in aufsteigender Reihenfolge verarbeitet. Zugriffe mehrerer Threads auf gleiche Speicherbereiche können mittels atomarer Operationen und Threadsynchronisation koordiniert werden.

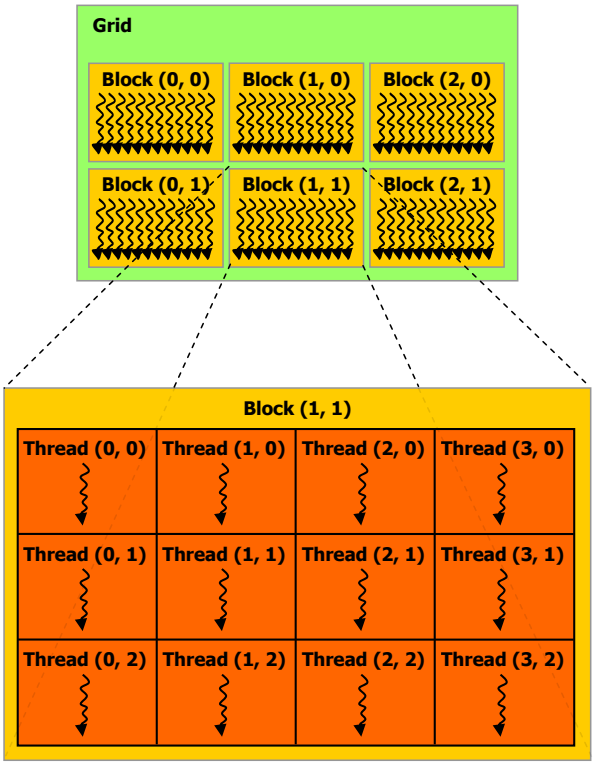


Abbildung 2.5: Hierarchie eines CUDA Grids [Nvi].

Speicher	Sichtbarkeit	Lebenszeit
Register	Thread	Kernel
Shared Memory	Block	Kernel
Constant Memory	Grid	Anwendung
Local Memory	Thread	Kernel
Global Memory	Grid	Anwendung

Tabelle 2.2: Speicherarten sortiert in aufsteigender Reihenfolge nach Zugriffszeit

Speicherarchitektur

Die CUDA Speicherarchitektur unterscheidet zwischen verschiedenen Speicherarten mit unterschiedlichen Zugriffszeiten (siehe Tabelle 2.2) und ist entsprechend als Cache Coherent NUMA (ccNUMA) einzuordnen. Register und Shared Memory sind on-chip und damit die schnelleren Speicherarten. Local, Constant und Global Memory sind off-chip und deshalb erheblich langsamer.

Die Bezeichnung Local Memory ist hier irreführend, da es sich nicht um schnellen lokalen Speicher handelt, sondern um langsamen globalen Speicher mit anderer Sichtbarkeit und Le-

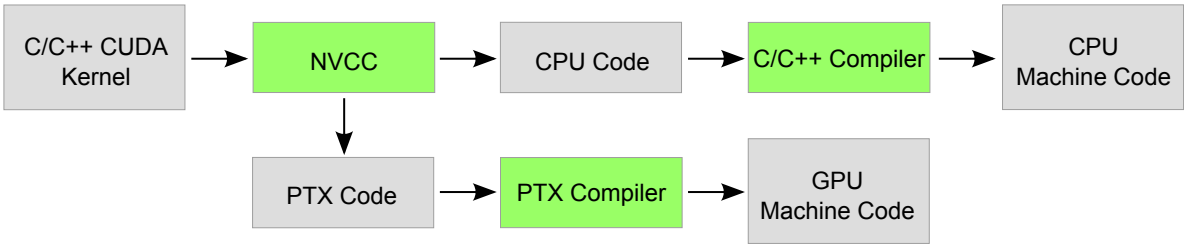


Abbildung 2.6: Übersetzung von C/C++-CUDA-Code mittels NVCC in Maschinen-Code.

benszeit. Local Memory wird verwendet, wenn der Register-Allocator des CUDA-Compilers nicht alle Variablen eines Kernels auf die verfügbaren Register abbilden kann. Dieser Flaschenhals kann durch Analyse des Assembler-Codes lokalisiert und beispielsweise durch Einsatz von Shared Memory beseitigt werden.

Aufgrund der geringen Anzahl an Speicherbänken im Verhältnis zu CUDA Cores sollte ein Thread, der Global Memory verwendet, mit möglichst wenigen Speicherbänken gleichzeitig interagieren. Ungünstige Nutzung des Global Memory führt zu sequenzieller Ausführung der Threads und dem Verlust der Cache Kohärenz.

Implizites Caching ist bei der Fermi-Architektur generisch implementiert, was bei älteren Architekturen nur für Lesezugriffe auf Texturen implementiert war. Explizites Caching kann mittels Shared Memory implementiert werden.

Werkzeuge

Der CUDA-Code wird, wie in Abbildung 2.6, mittels eines Compiler-Drivers namens NVIDIA C Compiler (NVCC) kompiliert. Dieser übersetzt zunächst den CUDA-Kernel in Parallel Thread Execution Assembler (PTX-Assembler) und normalen C/C++-Code. Anschließend startet NVCC einen PTX-Compiler, der GPU-spezifischen Maschinen-Code erzeugt und einen C/C++-Compiler, der CPU-spezifischen Maschinen-Code erzeugt. Beides wird anschließend zu einer ausführbaren Datei verlinkt. Der GPU-spezifische Maschinen-Code wird beim Ausführen der Anwendung durch den Grafikkartentreiber auf die Grafikkarte hochgeladen.

Das Werkzeug NVIDIA Parallel Nsight dient dem Debugging und Profiling von CUDA-Code auf der Grafikkarte. Für den Debugging-Modus werden zwei GPUs benötigt. Gleichzeitiges Debugging von CPU- und GPU-Code ist nicht möglich. Beide Einschränkungen schmälern den Anwenderkreis und den praktischen Nutzen. Der Profiling-Modus hat diese Einschränkungen nicht und liefert detaillierte Informationen über Auslastung und Laufzeiten. Bei der Suche nach einem Flaschenhals wird der Suchaufwand durch dieses Werkzeug merklich reduziert.

3 Verwandte Arbeiten

In diesem Kapitel wird ein Überblick über themenverwandte Arbeiten, Verfahren und Werkzeuge gegeben. Für viele der im Folgenden vorgestellten Werkzeuge steht allerdings eine statische Betrachtung des ‚aktiven Teils' eines Proteins im Vordergrund.

Obwohl viele der hier vorgestellten Verfahren als interaktiv angepriesen werden, sind sie größtenteils im Kontext statischer Daten beschrieben. In wie weit die Verfahren im Einzelnen für zeitabhängige Daten geeignet sind, soll im Rahmen dieser Arbeit, nicht geprüft werden.

3.1 Voronoibasierte Verfahren

Die erste Gruppe von Verfahren nutzt Voronoi-Diagramme als Grundlage für die Suche nach intra-molekularen Pfaden. Voronoi-Diagramme geben eine durch Punkte definierte Partitionierung des Raumes an. Eine Partition wird auch Voronoi-Region genannt. Eine Voronoi-Region entspricht dem Schnitt der durch Formel 3.1 beschriebenen Ebenen zwischen den Punkten $p \in C$ und $q \in C \setminus \{p\}$, wobei p das Zentrum einer Voronoi-Region ist.

$$d(p,q) = \{x \in \mathbb{R}^N : |p - x| < |q - x|\} \tag{3.1}$$

Durch eine Erweiterung der Definition können Kugeln, wie durch die Van-der-Waals-Kräfte der Atome bestimmt, anstelle von Punkten verwendet werden. Das Voronoi-Diagramm wird als Graph für die Suche nach Pfaden innerhalb des Proteins interpretiert (vergleiche Abbildung 3.1). Bei einfachen Ansätzen, wie in Caver [BEB+11] und Mole [PKKO07] realisiert, wird ein Startpunkt für die Suche nach dem kürzesten Weg vom Inneren des Proteins zur Oberfläche durch den Benutzer festgelegt. In komfortableren Ansätzen, wie in *Voronoi-Based Extraction and Visualization of Molecular Paths* [LBH11] beschrieben, werden alle von außen zugänglichen Pfade in Betracht gezogen, sodass der Benutzer das Ergebnis durch Selektion und Filterung lediglich verfeinern muss.

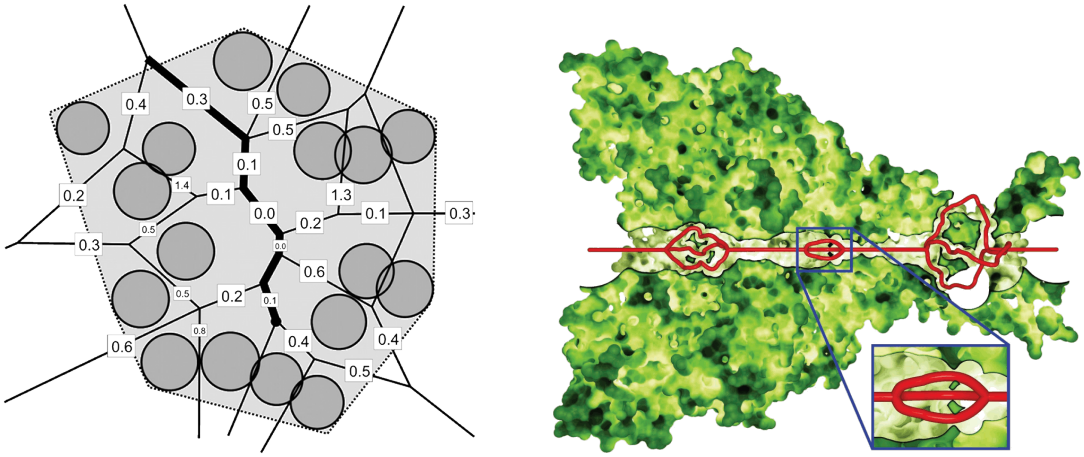


Abbildung 3.1: Voronoi-Diagramm als Graph aus den Van-der-Waals-Kräften der Atome (links [PKKO07]) und die Visualisierung eines Tunnel-Pfades (rechts [LBH11]).

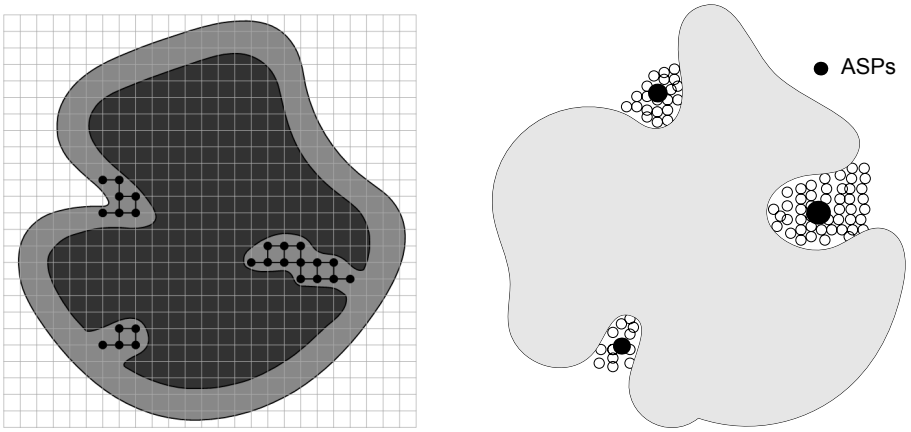


Abbildung 3.2: Auf Gitter (links [WPS07]) und auf Sonden (rechts [HS06]) basierende Verfahren. ASP steht für Active Site Point.

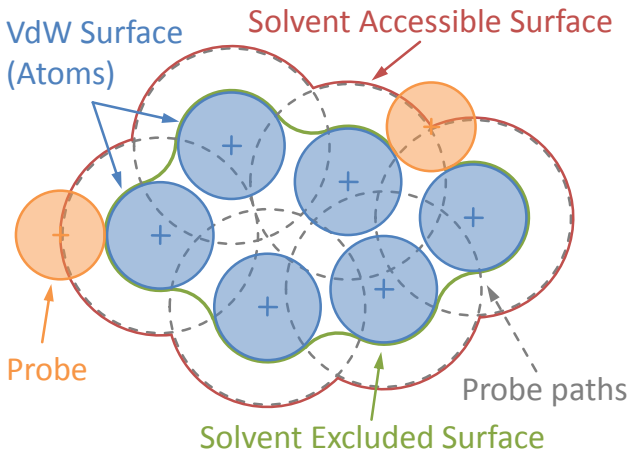


Abbildung 3.3: Schematische Darstellung der Oberflächendefinition [KGE11].

3.2 Sonden- und gitterbasierte Verfahren

Die zweite Gruppe von Verfahren nutzt Sonden oder Gitter für die Suche nach intramolekulare Taschen. Ein zentrales Problem dieser Verfahren ist die Oberflächendefinition. Abbildung 3.3 zeigt die in diesem Kontext gängigen Oberflächendefinitionen **Solvent Excluded Surface (SES)** und **Solvent Accessible Surface (SAS)**. Beispielsweise berechnet das Werkzeug McVol [TU10] die Oberfläche mit Hilfe eines Monte Carlo Verfahrens. In MegaMol [KFR+11] wird dieses Problem durch eine approximierte, SES-ähnliche Oberfläche, die sehr zügig durch Splatting berechnet werden kann, gelöst. Abbildung 3.2 zeigt den Unterschied zwischen einem auf Gitter und einem auf Sonden basierenden Verfahren:

Gitterbasierte Verfahren füllen Taschen anhand eines festen Gitters, um die Ausdehnung einer Tasche zu bestimmen. Dieser Ansatz hat den Nachteil, dass dünne, diagonal verlaufende Tunnel und Taschen bei niedriger Auflösung schlecht abgebildet werden. Bei hoher Auflösung steigt die Rechenzeit. Diese Verfahren sind in LIGSITE [HS06] und PocketPicker [WPS07] implementiert.

Sondenbasierte Verfahren lagern in einem ersten Schritt kugelförmige Sonden in atomähnlicher Größe an die Proteinoberfläche an. Sonden, die sich nicht in einer Tasche befinden, werden wieder entfernt. Anschließend werden an die übrigen Sonden weitere Sonden angelagert, bis die jeweiligen Taschen voll sind. Diese Verfahren sind in PASS [BS00] und Hollow [HG08] implementiert.

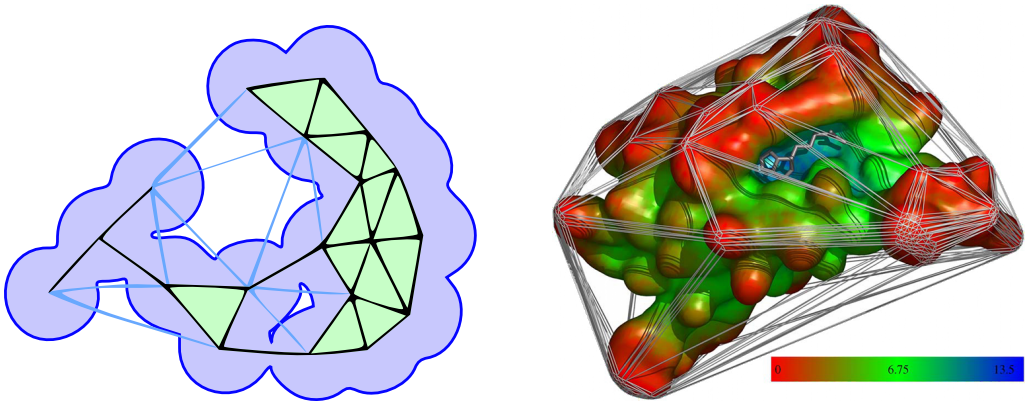


Abbildung 3.4: Proteintasche als Dreiecksnetz (links [LWE98]) und Einfärbung nach Tiefe anhand der konvexen Hülle (rechts [CS06]).

3.3 Verfahren auf Basis von Hüllendefinitionen

Die dritte Gruppe von Verfahren nutzt Hüllendefinitionen als Grundlage (siehe Abbildung 3.4). In CAST [LWE98] wird auf Basis von Alpha-Shapes mittels Delaunay-Triangulation ein Dreiecksnetz gebildet. Ein Alpha-Shape ist eine durch Kugeln definierte Hülle. Jede Kugel eines Alpha-Shapes hat den Radius α und darf Punkte (hier: Atome) lediglich berühren, aber nicht enthalten. Bei der Delaunay-Triangulation werden Punkte zu Dreiecken verbunden, wenn eine Kugel platziert werden kann, in der keine anderen Punkte liegen, sodass ein direkter Bezug zu Voronoi-Diagrammen besteht: Jeder Schnittpunkt zwischen Voronoi-Regionen entspricht dem Zentrum einer Kugel. In *Travel Depth, a New Shape Descriptor for Macromolecules: Application to Ligand Binding* [CS06] wird die konvexe Hülle verwendet, um eine Metrik für die Tiefe zu definieren. Das Interessante an dieser Metrik ist, dass das aktive Zentrum meist tief im Inneren eines Proteins liegt. Entsprechend liegt die Tiefe Null auf der konvexen Hülle. Die Berechnung der konvexen Hülle ist allerdings aufwendig.

4 Entwurf

In diesem Kapitel werden mögliche Lösungswege in Form von Entwurfsüberlegungen skizziert und über den Einsatz von Marching Cubes, Marching Tetrahedrons, Segmentierung und Korrelation reflektiert. Eine Beschreibung der realisierten Lösung befindet sich in Kapitel 5.

4.1 Marching Cubes

Marching Cubes ist ein von Lorensen und Cline [LC87] entwickeltes Verfahren zur Berechnung von polygonalen Isoflächen aus Volumendaten. Dazu wird in einem ersten Schritt ein Volumen in ein gleichmäßiges Gitter unterteilt. Ein Gitterelement wird, wie bei Marching Cubes üblich, Würfel genannt und besteht aus acht Eckpunkten. Jeder Eckpunkt eines Würfels wird anhand eines Schwellwerts als innenliegend oder außenliegend klassifiziert. Aus der Konfiguration des Würfels kann auf den Schnitt mit der Isofläche geschlossen werden. Jeder der acht Eckpunkte hat zwei mögliche Zustände, sodass es $2^8 = 256$ mögliche Konfigurationen gibt, die durch Ausnutzung von Symmetrien auf 15 Fälle (siehe Abbildung 4.1) vereinfacht werden können. Bereits kurz nach der Veröffentlichung des Algorithmus fiel auf, dass die Fälle 3, 4, 6, 7, 10, 12 und 13 unterschiedlich interpretiert werden können. Unterschiedliche Interpretation führt zu Mehrdeutigkeiten, was sich in Löchern im Polygonnetz äußert, wenn keine entsprechende Sonderfallbehandlung, wie durch Newman und Yi [NY06] in ihrer Zusammenstellung beschrieben, erfolgt.

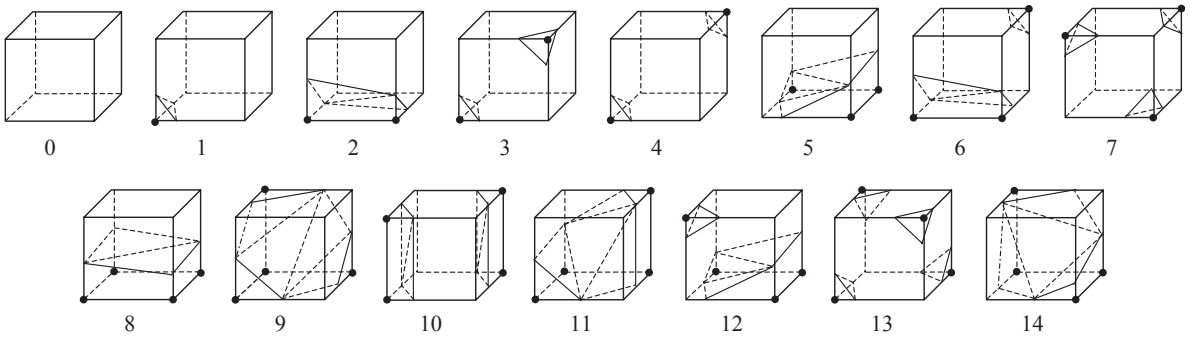
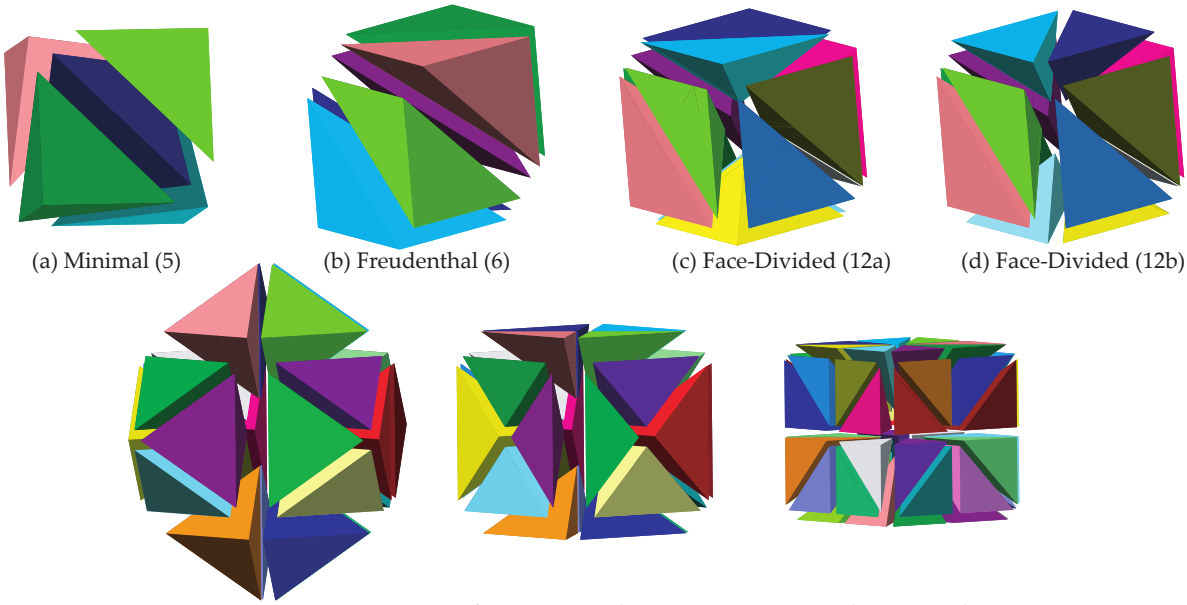


Abbildung 4.1: Grundlegende Topologien für Marching Cubes [NY06].



(a) Minimal (5) (b) Freudenthal (6) (c) Face-Divided (12a) (d) Face-Divided (12b)

(e) BCC (12 average) (f) Face-Centred (24) (g) Edge-Centred (48)

Abbildung 4.2: Unterteilungschemata für Marching Tetrahedrons [CMS06].

4.1.1 Marching Tetrahedrons

Für Marching Cubes wurde ein Patent erteilt, weshalb der Algorithmus lange nicht genutzt werden konnte, ohne Gebühren zu entrichten. Das Patent lief im Jahr 2005 aus. Deshalb wurde mit Marching Tetrahedrons eine Alternative entwickelt, die Tetraeder anstelle von Würfeln verwendet. Um ein kubisches Gitter in Tetraeder zu unterteilen, sind verschiedene Schemata denkbar (siehe Abbildung 4.2). Die Arbeit von Carr, Moller und Snoeyink [CMS06] zeigt, dass die entstehenden Artefakte, je nach Schema, sehr unterschiedlich ausfallen können. Die durch die Approximation entstehenden Artefakte, besonders bei geringer Auflösung, spielen für diese Arbeit keine Rolle, denn die Gitterauflösung kann maximal gewählt werden: Ein Gitterpunkt wird einem Voxel entsprechen.

Die am häufigsten verwendete Unterteilung für Marching Thedrahedrons ist das sechs Tetraeder-Schema in Abbildung 4.2 (b). Jeder der vier Eckpunkte eines Tetraeders hat zwei mögliche Zustände, sodass es $2^4 = 16$ mögliche Konfigurationen gibt, die bei geschickter Implementierung, wie durch Kipfer und Westermann [KW05] beschrieben, auf drei Fälle (siehe Abbildung 4.3) vereinfacht werden können.

4.1.2 Vergleich

Für Marching Cubes spricht, dass eine fertige Implementierung in CUDA vorhanden ist. Marching Cubes benötigt 12 Schnittberechnungen (eine für jede Würfelkante) und erzeugt

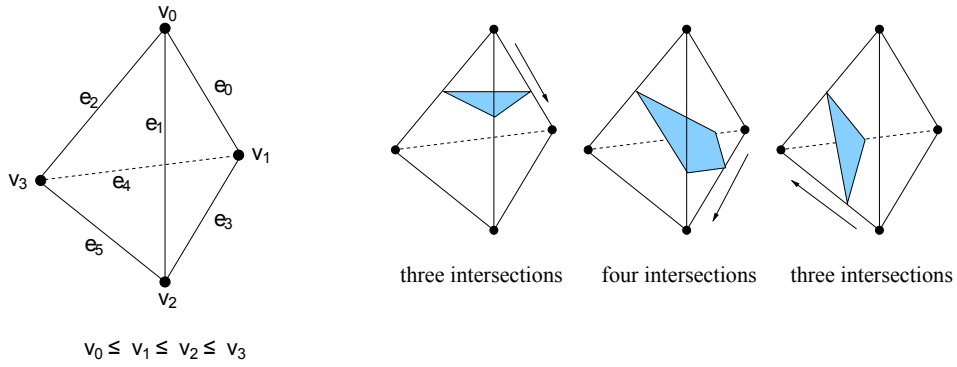


Abbildung 4.3: Grundlegende Topologien für Marching Tetrahedrons [KW05].

höchstens 4 Dreiecke pro Würfel. Gegen Marching Cubes spricht, dass Mehrdeutigkeiten gesondert behandelt werden müssen und Lookup-Tabellen für 256 Konfigurationen relativ groß ausfallen. Für Marching Thedrahedrons spricht, dass keine Mehrdeutigkeiten existieren, die Lookup-Tabellen für 16 Konfigurationen relativ klein ausfallen und der Algorithmus deshalb einfacher zu modifizieren ist. Marching Thedrahedrons benötigt 19 Schnittberechnungen - nicht jede der 6 Kanten eines Tetraeders ist relevant - und erzeugt höchstens 2 Dreiecke pro Tetraeder oder 12 Dreiecke pro Würfel. Gegen Marching Tetrahedrons spricht, dass keine fertige Implementierung in CUDA vorhanden ist. Obwohl mehr Dreiecke berechnet werden, ist das nicht unbedingt als Nachteil zu werten, denn jeder Tetraeder kann parallel berechnet werden.

Die Wahl fiel letztlich auf Marching Thedrahedrons, weil das Ermitteln der Nachbarschaft zwischen Tetraedern leichter zu implementieren und die Approximation genauer ist. Außerdem konnten Teile der fertigen Marching Cubes Implementierung für Marching Tetrahedrons wiederverwendet werden. Eine Beschreibung der Implementierung befindet sich in Kapitel 5 auf Seite 35.

4.2 Segmentierung

Die Segmentierung eines allgemeinen Dreiecksnetzes ist ein Problem aus der Graphentheorie. Bei Marching Thedrahedrons bilden sechs Tetraeder einen Würfel, die zusammen wiederum ein Gitter bilden, sodass eine Ähnlichkeit zu Voxeldaten besteht. Verfahren zur Bildsegmentierung arbeiten klassischerweise auf Pixelebene, können aber meist auf Voxel übertragen werden. Ein Exkurs in die Bildsegmentierung lohnt, weil GPGPU in der Bildverarbeitung weiter verbreitet ist als in der klassischen Graphentheorie. Außerdem werden möglicherweise übertragbare, zusätzliche Eigenschaften, wie Farbe oder Gradient verwendet, um das Ergebnis zu verbessern.

Große Teile des folgenden Abschnitts stammen aus Erfahrungen im Rahmen eines Studienprojekts und aus einem Abriss von O. Wirjadi [Wir07]. Für jede Verfahrensart folgt eine

kurze Beschreibung mit Informationen über die Beschaffenheit der berechneten Segmente, Eignung für dynamische Daten, veränderbare Parameter und die Parallelisierbarkeit.

4.2.1 Schwellwertverfahren

Bei Schwellwertverfahren wird anhand eines Schwellwerts s entschieden, ob ein Pixel p zu einem der Segmente s_0 oder s_1 gehört:

$$S(p) = \begin{cases} 0 & \text{falls } p < s \\ 1 & \text{falls } p \geq s \end{cases} \tag{4.1}$$

Pro Schwellwertverfahren sind leicht zu implementieren und parallel berechenbar.

Contra Proteine haben mit großer Wahrscheinlichkeit mehr als zwei Segmente. Schwellwertverfahren sind sehr anfällig gegenüber dynamischen Effekten, wie zeitabhängige Änderungen oder Rauschen, und das Ergebnis hängt von der Qualität des Schwellwerts ab.

4.2.2 Kantenerkennung

Bei der Kantenerkennung werden Flächen anhand von Kantenoperatoren, wie dem Sobel-Operator oder Laplace-Operator, getrennt. Ein Kantenoperator unterscheidet zwischen $\{s_{\text{Kante}}, s_{\text{NichtKante}}\}$ und wird häufig als Filter-Matrix dargestellt, deren Anwendung einer diskreten Faltung entspricht.

Pro Falls es bereits eine geeignete Filter-Matrix gibt, sind sie ebenfalls leicht zu implementieren und parallel berechenbar. Ein Operator, der auf der Oberflächenkrümmung basiert, könnte verwendet werden, um Taschen oder Tunneleingänge zu erkennen.

Contra Proteine haben mit großer Wahrscheinlichkeit mehr als zwei Segmente, sodass Kantenerkennung lediglich ergänzend eingesetzt werden könnte. Kantenoperatoren sind ebenfalls anfällig für dynamische Effekte. Das Herleiten einer eigenen Filter-Matrix oder Erfinden eines Kantenoperators ist mathematisch anspruchsvoll.

4.2.3 Regions- und formbasierte Verfahren

Bei regionsbasierten Verfahren wird anhand eines Prädikats entschieden, ob zwei Pixel dem gleichen Segment angehören. Ein solches Prädikat kann durch Region-Growing oder Region-Merging angewendet werden. Region-Growing bezeichnet einen Prozess, bei dem ein Startpixel, auch ‚Seed' genannt, ausgewählt und diese Region anhand ihrer anliegenden Pixel vergrößert wird, bis keine Veränderung mehr zu beobachten ist. Region-Merging funktioniert ohne Seed und vereinigt so lange Pixel zu Regionen, bis keine Veränderungen mehr zu beobachten sind. Formbasierte Verfahren unterscheiden sich von regionsbasierten Verfahren durch die Berücksichtigung der Ränder und benötigen immer einen Seed. Bekannte Vertreter dieser Verfahren sind Level-Set und Deformable Surfaces. Dieser Weg wurde mit *A new marching cubes algorithm for interactive level set with application to MR image segmentation* [FB10] bereits von Anderen beschritten

Pro Regions- und formbasierte Verfahren sind bis zu einem gewissen Grad parallel implementierbar. Die Anwendung eines Prädikats ist sehr allgemeingültig, also erweiterbar und veränderbar. Die Anfälligkeit für dynamische Effekte hängt stark vom verwendeten Prädikat ab.

Contra Seeds entsprechen einer Benutzereingabe, was einer Arbeitsthese widerspricht. Das Auflösen von Segmentreferenzen beim Zusammenführen großer Segmente ist kein triviales Problem.

4.2.4 Textur- und modellbasierte Verfahren

Texturbasierte Verfahren suchen nach Mustern, meist Oberflächenstrukturen. Modellbasierte Verfahren ordnen ein Pixel einem Segment anhand von Wissen über die Daten zu („Gras ist grün, Straße ist schwarz, Fahrbahnmarkierungen sind weiß").

Pro Die Einbringung von Wissen kann das Problem vereinfachen.

Contra Ein textur- oder modellbasierter Ansatz macht in diesem Kontext keinen Sinn, da lediglich bekannt ist, dass das Volumen Kugeln enthält.

4.2.5 Abwägung

Aufgrund der parallelen Verarbeitungsstrategie und beeinflusst von der Veröffentlichung *Analysis of a step-based watershed algorithm using CUDA* [VKALF10] erschienen regionsbasierte Verfahren geeignet. Es wird kein Startwert oder besonderes Wissen über die Daten benötigt; außerdem ist das Prädikat erweiterbar. Das Prädikat wird zunächst so definiert, dass

zusammenhängende Voxel ein Segment bilden. Als Erweiterung des Prädikats kommen beispielsweise Gradient, Krümmung und Voronoi-Region in Frage.

4.3 Korrelation über Zeit

In der Aufgabenstellung wird eine Korrelation der Dreiecksnetze über Zeit, also deren Bewegungen, gefordert. Die Ähnlichkeit zweier Funktionen x und m lässt sich wie in Formel 4.2 mit m^* als komplex-konjugiertes m definieren.

$$x \circ m = \int_{-\infty}^{\infty} x(t) m^*(t+\tau) \mathrm{d}t \tag{4.2}$$

Die Position eines Dreiecks kann als integrierte Geschwindigkeitsfunktion betrachtet werden. Mittels Korrelation kann für jedes Dreieck eine Aussage über die Ähnlichkeit zum vorherigen Zeitschritt getroffen werden. Die Korrelation $x \circ m$ lässt sich mit Hilfe der Fourier-Transformation (FT) wie in Formel 4.3 berechnen.

$$\begin{aligned} \mathrm{FT}(x \circ m) &= \mathrm{FT}(x)\mathrm{FT}^*(m) \\ x \circ m &= \mathrm{FT}^{-1}\left(\mathrm{FT}(x)\mathrm{FT}^*(m)\right) \end{aligned} \tag{4.3}$$

Die Fourier-Transformation bildet vom Ortsraum in den Frequenzraum ab. Die Ähnlichkeit kann durch filtern nach Extrema und Rücktransformation in den Ortsraum visualisiert werden.

Auch die schnelle Fourier-Transformation (FFT) kostet viel Rechenzeit, weshalb für diese Arbeit folgende Vereinfachung gemacht wird: Wenn man für jedes unabhängige Segment einen Schwerpunkt s_i aus den Eckpunkten eines Dreiecksnetzes v_j wie in Formel 4.4 bildet, bleiben wenige Werte übrig, die korreliert werden müssen.

$$s_i = \sum_{j=1}^{N_i} \frac{v_j}{N_i} \tag{4.4}$$

Der zeitliche Vorgänger eines Segments wäre demnach mit großer Wahrscheinlichkeit das Segment aus dem letzten Zeitschritt, dessen Schwerpunkt am nächsten liegt. Diese Heuristik verliert ihre Eindeutigkeit dann, wenn zwei Komponenten zwischen einzelnen Zeitschritten ‚Sprünge' in nächster Nähe machen, was Abbildung 4.4 verdeutlicht. Man kann allerdings davon ausgehen, dass dieser Fall in der Praxis nicht vorkommt: Animationsschritte sind in MegaMol zeitlich dicht gestaffelt und folgen einem physikalischen Gleichgewichtsmodell einer Molekulardynamik-Simulation. Ereignisse wie Amalgamieren und Aufspalten können anhand des zeitlichen Verlaufs klassifiziert werden.

Ein praktischer Vergleich zwischen einem Ansatz auf Basis der Fourier-Transformation und auf Basis von Schwerpunkten oder eine Kombination war aus zeitlichen Gründen nicht möglich.

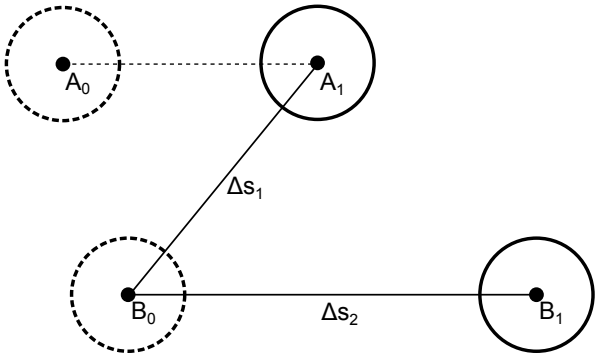


Abbildung 4.4: Fehlerfall der Korrelationsheuristik über Schwerpunkte: mit $\Delta_{s1} < \Delta_{s2}$ wird B_0 fälschlicherweise A_1 zugeordnet.

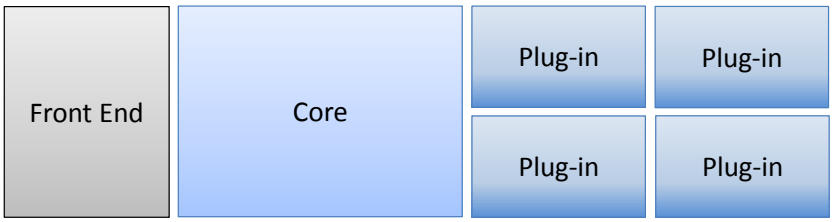


Abbildung 4.5: MegaMol Softwarearchitektur [Gro10].

4.4 MegaMol

Das MegaMol Framework wurde an der Universität Stuttgart im Rahmen des Sonderforschungsbereichs SFB 716 von Sebastian Grottel [Gro10] entwickelt und dient als gemeinsame Software-Plattform für die Visualisierungsprojekte des Sonderforschungsbereiches. MegaMol ist auf Partikeldaten und -rendering ausgelegt und kann mit Hilfe von einfachen XML- und Parameterdateien konfiguriert werden: In den XML-Dateien wird ein Scene Graph aus ‚Module-Knoten' und ‚Call-Kanten' definiert, der mit Hilfe einer Textdatei, die Schlüssel-Wert-Paare enthält, parametrisiert wird.

4.4.1 Softwarearchitektur

Die Softwarearchitektur des MegaMol Frameworks ist ein typisches Drei-Ebenen-Modell (siehe Abbildung 4.5):

Core Die Core-Komponente ist eine dynamische Bibliothek, die von allen anderen Komponenten verwendet wird. Core implementiert die Basisklassen `Module`, `Call` und einige Hilfsklassen unter anderem für den Umgang mit Konfigurationsdateien. Ein `Module` entspricht idealerweise einem Zweck, wie dem Laden einer Datei, durchführen einer Berechnung

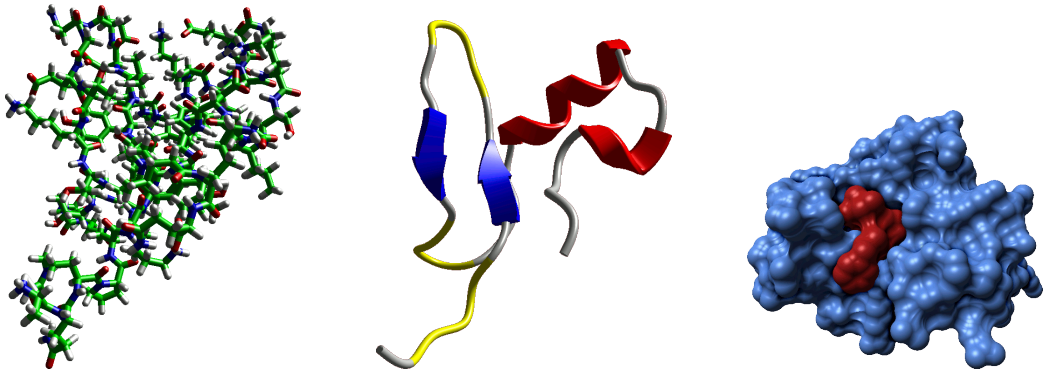


Abbildung 4.6: Ball&Stick-Darstellung (links [KBE08]) der Atome, Cartoon-Darstellung (mittig [KBE08]) von α-Helix und β-Faltblatt und Volumen-Darstellungen eines Solvent Excluded Surfaces (rechts [KFR+11]).

oder Darstellung eines Ergebnisses, sodass komplexes Verhalten durch Konfiguration hergestellt werden kann. Es wird sowohl das Push- als auch das Pull-Mantra im Umgang mit Arbeitsdaten unterstützt. Ein `Module` ist Eigentümer der Daten. Ein `Call` entspricht einem Aufruf einer `Module`-Instanz.

Frontend MegaMol kann in andere Anwendungen eingebettet werden, weshalb Frontend-Komponenten getrennt zu betrachten sind. Im Fall von MegaMol Console ist das Frontend eine eigenständige Konsolenanwendung, die eine Konfiguration lädt und anzeigt.

Plug-ins Plug-ins sind dynamische Bibliotheken mit der Dateiendung `*.mmplg`. Plug-ins erweitern die Funktionalität von MegaMol zur Laufzeit um neue Modules und Calls.

4.4.2 Protein Plug-in

Das Protein Plug-in erweitert MegaMol um rund 50 Klassen, die von `Module` oder `Call` erben, für die Interaktion mit Proteindaten. Zu den vorhandenen Modulen gehören unter anderem ein `PDBLoader`, der `*.pdb` und `*.xtc` Dateien laden kann. PDB-Dateien enthalten eine ASCII-Beschreibung des Proteins. XTC-Dateien ergänzen PDB-Dateien um binär kodierte Trajektorien im GROMACS-Format. GROMACS ist eine Software für die Simulation von Molekulardynamik. Der Zugriff auf PDB- und XTX-Daten erfolgt mittels `MolecularDataCall`. Ein Großteil der übrigen Klassen implementieren Atom-, Cartoon- oder Volumen-Darstellungen (siehe Abbildung 4.6).

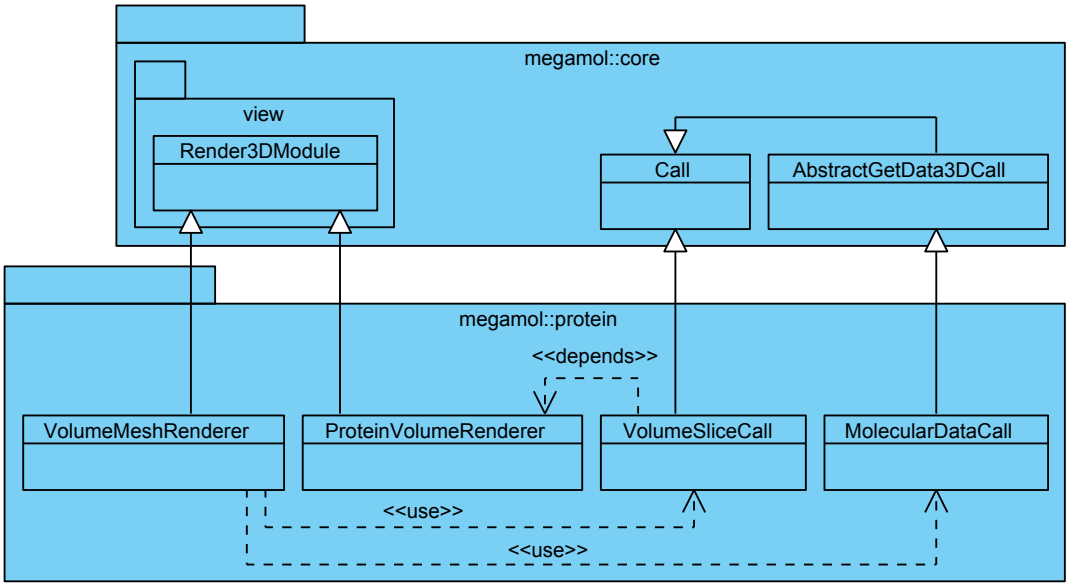


Abbildung 4.7: `VolumeMeshRenderer` als Klassendiagramm.

4.4.3 Neue Komponenten

Die Komponenten sind so entworfen, dass möglichst wenige Änderungen an bestehenden Klassen gemacht werden müssen, um das Merging in den Hauptentwicklungszweig von MegaMol Protein zu erleichtern.

Es wird an die drei Klassen `VolumeSliceCall`, `ProteinVolumeRenderer` und `MolecularDataCall` angeknüpft (siehe Abbildung 4.7). Das Splatting aller Atome eines Proteins in eine Volumen-Textur ist bereits in `ProteinVolumeRenderer` implementiert und ist über einen `VolumeSliceCall` zugänglich. `MolecularDataCall` wird für die Animationsdaten benötigt. Der `VolumeMeshRenderer` enthält die Routinen zur Berechnung des Dreieckesnetzes mittels Marching Thedrahedrons, Segmentierung, Korrelation und eine einfache grafische Ausgabe. Die Klasse ist so angelegt, dass die einzelnen Routinen möglichst unabhängig voneinander sind, um ein Refactoring zwecks Wiederverwendung zu erleichtern.

5 Implementierung

In diesem Kapitel wird die Implementierung erläutert. Die Beschreibung der Algorithmen und Datenstrukturen ist weitgehend von der Programmiersprache abstrahiert. Dieses Kapitel hat nicht den Anspruch einer Bauanleitung. Gängige Technologie-Idiome, wie das Bilden eines Index aus CUDA Thread- und Block-ID für den kohärenten Cache-Zugriff, werden nicht näher erläutert, weshalb Erfahrung mit C++ und CUDA empfohlen wird.

5.1 Überblick

Es kann davon ausgegangen werden, dass die Proteindaten bereits als Volumentextur verfügbar sind. Das Ergebnis wird im Anschluss grafisch aufbereitet. Es folgt zunächst eine stark vereinfachte Auflistung der einzelnen Arbeitsschritte, um einen Überblick zu vermitteln:

1. Marching Tetrahedrons
   a) ‚Aktive' zu ‚Arbeits'-Würfeln verdichten
      i. Würfel als aktiv oder inaktiv klassifizieren
      ii. Abbildung von Arbeits-Würfeln auf aktive Würfel erzeugen
   b) Tetraeder in einem aktiven Würfel klassifizieren
   c) Zusammenhangslose Dreiecke erzeugen
2. Komponenten segmentieren
   a) Alle Tetraeder aufsteigend nummerieren
   b) Lokales Minimum (Tetraeder-Nummer) in der direkten Nachbarschaft jedes Tetraeders finden
   c) Referenzpfade komprimieren
   d) Segmentnummer zuweisen
3. Komponenten über Zeit korrelieren
   a) Schwerpunkt einer Komponente berechnen
   b) Bezug zu vorherigem Zeitschritt herstellen
   c) Ereignisse klassifizieren

Index	0	1	2	3	4	5
Würfel-Aktivität	0	0	1	1	0	1
Würfel-Versatz	0	0	0	1	2	2
Arbeits-Würfel	2	3	5			

Tabelle 5.1: Beispiel für das Erzeugen einer Abbildung von ‚Arbeits'-Würfeln auf ‚aktive' Würfel

5.2 Marching Tetrahedrons

Die im Rahmen dieser Arbeit erstellte Implementierung von Marching Tetrahedrons basiert auf der Marching Cubes Implementierung, die dem CUDA Toolkit von NVIDIA beiliegt, einem Artikel über Marching Cubes sowie Marching Tetrahedrons von Paul Bourke [Bou94] und der Präsentation *Large-Scale Isosurfacing on a Distrubuted GPU Cluster* [BJN10]. Die Kenntnis der drei grundlegenden Topologien für Marching Tetrahedrons (siehe Abbildung 4.3, Seite 27) wird im Folgenden vorausgesetzt. Die acht Eckpunkte eines Würfels, der wiederum sechs Tetraeder enthält, entsprechen acht aneinander liegenden Voxeln, sodass die Auflösung maximal ist.

5.2.1 Verdichtung der Würfel

Zunächst wird nach ‚Aktivität' von Würfeln verdichtet. Ein Würfel wird als inaktiv betrachtet, wenn alle Eckpunkte des Würfels innen- oder außenliegend sind. Die Verdichtung reduziert die Datenmenge für nachfolgende Arbeitsschritte um 90-95% und sorgt gleichzeitig für eine gleichmäßigere Auslastung der Hardware, da praktisch jeder CUDA Thread anschließend Dreiecke generiert. Dazu werden in einem ersten Schritt alle Würfel parallel nach Aktivität klassifiziert. Anschließend wird der Versatz für eine Abbildung von ‚Arbeits-Würfel-Index' auf ‚Aktiver-Würfel-Index' durch eine Präfix-Summe aus der Thrust-Bibliothek [HB10] berechnet und die eigentliche Abbildung erzeugt. Tabelle 5.1 verdeutlicht die einzelnen Schritte an einem Beispiel. Die Auswirkungen auf das Leistungsverhalten durch diese Optimierung sind in Kapitel 6 dokumentiert.

5.2.2 Klassifizierung der Tetraeder

Die verbleibenden 5-10% der aktiven Würfel werden in sechs Tetraeder wie in Abbildung 5.1 anhand der Tabelle 5.2 aufgeteilt. Die Konfiguration $k(i)$ des i-ten Tetraeders in einem Würfel wird wie in Formel 5.1 anhand des Schwellwertes t berechnet und, wie für Marching Tetrahedrons üblich, als Bitfeld dargestellt.

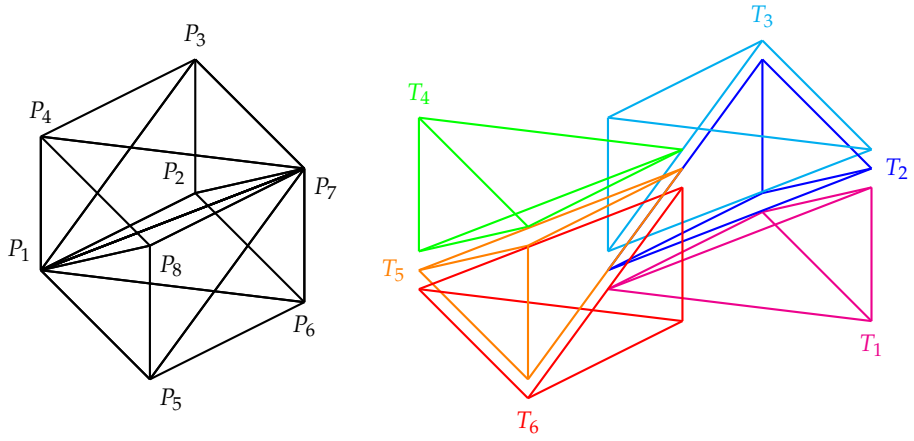


Abbildung 5.1: Aufteilung eines Würfels (links) in sechs Tetraeder (rechts)

	T_i^1	T_i^2	T_i^3	T_i^4
T_1	P_1	P_6	P_2	P_7
T_2	P_1	P_2	P_3	P_7
T_3	P_1	P_3	P_4	P_7
T_4	P_1	P_4	P_8	P_7
T_5	P_1	P_8	P_5	P_7
T_6	P_1	P_5	P_6	P_7

Tabelle 5.2: Aufteilung eines Würfels in sechs Tetraeder

$$c(i,j) = \begin{cases} 0 & \text{falls } f(\vec{T}_i^{j+1}) \leq t \\ 1 & \text{falls } f(\vec{T}_i^{j+1}) > t \end{cases} \tag{5.1}$$

$$k(i) = \sum_{j=0}^{3} c(i,j)2^j$$

Anhand der 16 möglichen Konfigurationen kann klassifiziert werden, ob ein Tetraeder null, drei oder sechs Eckpunkte erzeugt (vergleiche Tabelle 5.4). Der Versatz der Eckpunkte innerhalb des Dreiecksnetzes wird ebenfalls mit einer Präfix-Summe anhand der Klassifikation berechnet.

5.2.3 Berechnung der Dreiecke

Der für die Zieladresse nötige Versatz eines Dreiecks wurde im vorhergehenden Schritt berechnet, sodass für jeden Tetraeder unabhängig null bis zwei Dreiecke erzeugt werden

	E_1	E_2	E_3	E_4	E_5	E_6
Eckpunkt A der Tetraederkante	T_i^1	T_i^2	T_i^3	T_i^1	T_i^2	T_i^3
Eckpunkt B der Tetraederkante	T_i^2	T_i^3	T_i^1	T_i^4	T_i^4	T_i^4

Tabelle 5.3: Abbildung von Tetraederkanten auf Eckpunkte

Konfiguration	Kantenmaske	|Eckpunkte|	Tetraederkanten					
0000_2	$00\ 0000_2$	0						
0001_2	$00\ 1101_2$	3	E_1	E_4	E_3			
0010_2	$01\ 0011_2$	3	E_1	E_2	E_5			
0011_2	$01\ 1110_2$	6	E_2	E_5	E_3	E_3	E_5	E_4
0100_2	$10\ 0110_2$	3	E_2	E_3	E_6			
0101_2	$10\ 1011_2$	6	E_1	E_4	E_6	E_1	E_6	E_2
0110_2	$01\ 0011_2$	6	E_1	E_3	E_6	E_1	E_6	E_5
0111_2	$11\ 1000_2$	3	E_6	E_5	E_4			
1000_2	$11\ 1000_2$	3	E_4	E_5	E_6			
1001_2	$01\ 0011_2$	6	E_5	E_6	E_1	E_6	E_3	E_1
1010_2	$10\ 1011_2$	6	E_2	E_6	E_1	E_6	E_4	E_1
1011_2	$10\ 0110_2$	3	E_6	E_3	E_2			
1100_2	$01\ 1110_2$	6	E_4	E_5	E_3	E_3	E_5	E_2
1101_2	$01\ 0011_2$	3	E_5	E_2	E_1			
1110_2	$00\ 1101_2$	3	E_3	E_4	E_1			
1111_2	$00\ 0000_2$	0						

Tabelle 5.4: Für jede Tetraederkonfiguration: Kantenmaske, Anzahl der erzeugten Dreiecke und benötigte Kanten

können. Eine Kante eines Tetraeders trägt genau dann zu einem Dreieck bei, wenn einer der Eckpunkte einer Kante innen- und der andere außenliegend ist. Anhand der Konfiguration kann mit Hilfe einer Kantenmaske aus Tabelle 5.4 geprüft werden, ob eine Kante einen Eckpunkt erzeugt. Für jede zu betrachtende Kante wird anhand von Tabelle 5.3 zwischen zwei Eckpunkten des Tetraeders a und b anhand des Schwellwertes t, wie in Formel 5.2, linear interpoliert.

$$\delta = \frac{t - f(\vec{a})}{f(\vec{b}) - f(\vec{a})} \qquad \vec{v} = \vec{a} + \delta(\vec{b} - \vec{a}) \tag{5.2}$$

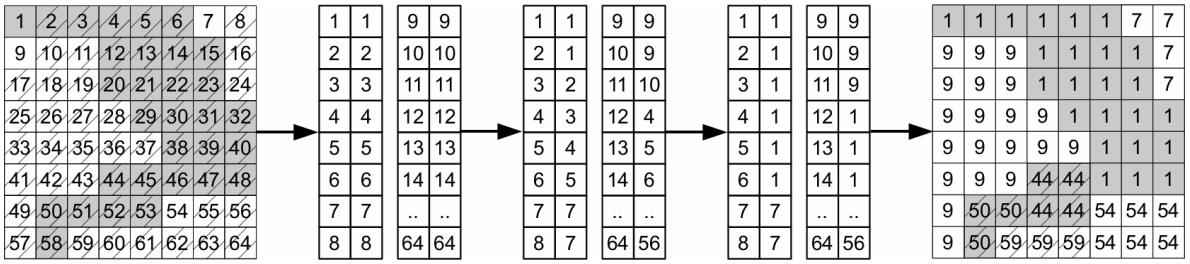


Abbildung 5.2: Eine Iteration der Label-Äquivalenz-Methode. Von links nach rechts: die anfängliche Beschriftung und Äquivalenzliste, die Liste nach der Scan-Phase, die Liste nach der Analyse-Phase und das Ergebnis der ersten Iteration [HLP10].

5.3 Segmentierung der Komponenten

Es stellte sich schnell heraus, dass ein naiver Ansatz, wie das Vergleichen aller Tetraeder gegeneinander, ungeeignet war, um zusammenhängende Komponenten zu finden, wenn man den Anspruch hat, mehr als 15-25 Bilder pro Sekunde zu berechnen: das Herunterladen einiger Megabyte von der Grafikkarte in den Hauptspeicher, dortiges Verarbeiten mit der CPU, um das Ergebnis wieder auf die Grafikkarte zu laden, benötigt über 100ms, was weniger als 10 Bildern pro Sekunde entspricht.

Die im Folgenden beschriebene Segmentierung basiert auf der Arbeit *Parallel Graph Component Labelling with GPUs and CUDA* [HLP10] von Harwick et al.. Der Verständlichkeit halber wird der Algorithmus zunächst als Pixelbild anhand von Abbildung 5.2 beschrieben. Der Algorithmus ist iterativ und zerfällt in eine Scan-, Analysis- sowie Labeling-Phase. Alle Listen werden vorab mit einer aufsteigenden, eindeutigen Nummer initialisiert. In der Scan-Phase werden für alle Pixel die direkten Nachbarpixel auf Ähnlichkeit untersucht und das Minimum in eine Referenzliste geschrieben. In der Analysis-Phase werden die Referenzen komprimiert. Eine Referenz gilt als komprimiert, wenn sie auf eine Selbstreferenz zeigt oder eine Selbstreferenz ist. In der Labeling-Phase werden die Referenzen aufgelöst. Diese drei Schritte werden wiederholt, bis keine Änderungen mehr in der Scan-Phase festgestellt werden. Das Interessante an diesem Algorithmus ist, dass er für jedes Pixel parallel laufen kann und praktisch keine Synchronisation innerhalb der Phasen benötigt, wenn man von einem atomaren Minimum beim Schreiben in die Äquivalenzliste in der Scan-Phase absieht.

5.3.1 Tetraedernachbarschaft

Ein Pixel eines Bildes hat 8 direkte Nachbarn. Bei Marching Thedrahedrons gibt es für jeden Würfeleckpunkt 7 Nachbarwürfel, die jeweils 6 Tetraedern enthalten, die wiederum jeweils 4 Eckpunkte haben, sodass bis zu 168 Eckpunkte auf Gleichheit geprüft werden müssten. Aufgrund der Art der Implementierung ist die Identität jedes Eckpunkts eines jeden Dreiecks

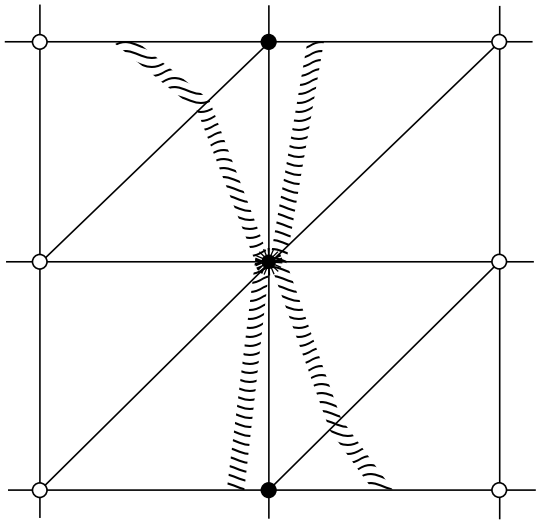


Abbildung 5.3: Grenzfall Identität vs. Wert: Die schraffierte Fläche entspricht der Oberfläche, weiß gefüllte Punkte liegen außerhalb, schwarz gefüllte Punkte liegen innerhalb des Volumens und der Wert des markierten Punktes entspricht dem Schwellwert

bekannt, sodass anhand der Voxelkoordiante eines Tetraeders in $O(1)$ auf einen Eckpunkt zugegriffen werden kann.

Die Anzahl der Lesezugriffe lässt sich deutlich reduzieren, indem lediglich die Kante eines Tetraeders untersucht wird. Wie bereits erwähnt, generiert eine Kante genau dann einen Eckpunkt, wenn einer der Kanteneckpunkte innenliegend und der andere außenliegend ist, sodass eine Nachbarschaft mit zwei Zugriffen behandelt werden kann:

1. Aktivität des Tetraeders prüfen
2. Auslesen des Bezeichners im Rahmen der Scan-Phase (Minimumsuche)

Anhand der in Abbildung 5.4 abgebildeten Nummerierung der Tetraederflächen und mit Tabelle 5.5 kann der zu prüfende Tetraeder bestimmt werden. Wichtig ist festzuhalten, dass es sich bei diesem Nachbarschaftskriterium nicht um eine Prüfung auf Gleichheit, sondern auf Identität handelt. Das Volumen könnte so gewählt werden, dass Tetraederflächen keine Nachbarn sind, obwohl es optisch und wertmäßig danach aussieht, wie in Abbildung 5.3 beispielhaft konstruiert. Dieser Fall tritt aufgrund der linearen Interpolation ein, wenn der Wert eines Voxels dem Schwellwert entspricht, sodass der Zähler zu Null ausgewertet wird (vergleiche Formel 5.2). Der Interpolator δ funktioniert hier als Gewicht.

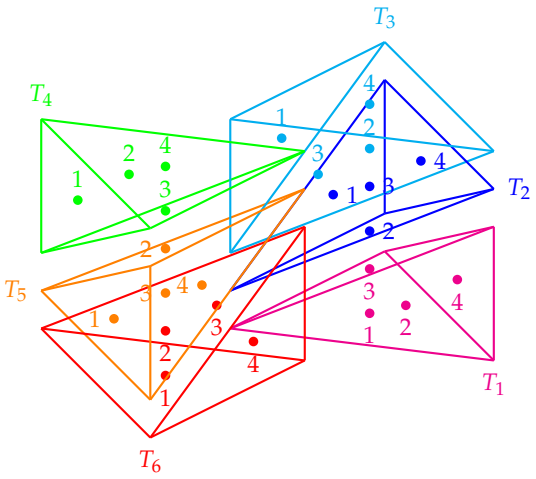


Abbildung 5.4: Nummerierung der Tetraederflächen T_iF_j

T_i	T_k für T_iF_1	T_k für T_iF_2	T_k für T_iF_3	T_k für T_iF_4
T_1	$T_3 \begin{pmatrix} 0 & -1 & 0 \end{pmatrix}^T$	$T_6 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_2 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_5 \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$
T_2	$T_6 \begin{pmatrix} 0 & 0 & -1 \end{pmatrix}^T$	$T_1 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_3 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_4 \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$
T_3	$T_5 \begin{pmatrix} 0 & 0 & -1 \end{pmatrix}^T$	$T_2 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_4 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_1 \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T$
T_4	$T_2 \begin{pmatrix} -1 & 0 & 0 \end{pmatrix}^T$	$T_3 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_5 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_6 \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T$
T_5	$T_1 \begin{pmatrix} -1 & 0 & 0 \end{pmatrix}^T$	$T_4 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_6 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_3 \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T$
T_6	$T_4 \begin{pmatrix} 0 & -1 & 0 \end{pmatrix}^T$	$T_5 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_1 \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$	$T_2 \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T$

Tabelle 5.5: Abbildung von Ursprungstetraederfläche T_iF_j nach Nachbartetraeder T_k mit Würfelgitterversatz

5.4 Korrelation und Klassifikation von Ereignissen

Die Schwerpunkte aller zusammenhängenden Komponenten werden mittels des Map-Reduce Entwurfsmusters berechnet. Dazu werden die Bezeichner der Tetraeder aus der vorhergehenden Segmentierung auf die Eckpunkte der Dreiecke verteilt (Mapping-Phase). Anschließend wird nach Komponenten sortiert, gezählt, summiert und der Quotient gebildet (Reduce-Phase).

Vor der Klassifikation wird zunächst für jeden Schwerpunkt eine nach Distanz sortierte Liste aller Schwerpunkt-Kandidaten aus dem vorherigen Zeitschritt erstellt. Durch Betrachten der Anzahl an Kandidaten k und Anzahl der Schwerpunkte s, die dem selben Kandidaten am nächsten liegen, können zeitliche Ereignisse wie folgt klassifiziert werden:

- $k = 0$: neue Komponente.
- $k = 1$ & $s = 1$: gleiche Komponente.
- $k = 1$ & $s > 1$: Aufspaltung in s Komponenten.
- $k > 1$: Amalgamierung zu einer Komponente.

5.5 Visualisierung

Die Visualisierung ist lediglich ein Sekundärziel der Aufgabenstellung und daher entsprechend einfach gehalten. Die für die Beleuchtung relevanten Vertex-Normalen entsprechen dem Gradient ∇f des Volumens und werden anhand der zentralen Differenz berechnet (siehe Formel 5.3). Zusammenhängende Komponenten erhalten eine über die Zeit möglichst gleichbleibende Farbe durch eine Zuordnungstabelle. Der Blick in das Innere des Proteins wird mittels Blending hergestellt.

$$\vec{n} = \nabla f = \begin{pmatrix} f(x-1,y,z) - f(x+1,y,z) \\ f(x,y-1,z) - f(x,y+1,z) \\ f(x,y,z-1) - f(x,y,z+1) \end{pmatrix} \tag{5.3}$$

5.6 Speicherverwaltung

Das Reservieren des Grafikspeichers benötigt bis zu einigen Millisekunden, was verhältnismäßig viel Zeit ist, im Vergleich zu den Laufzeiten einzelner CUDA Kernels. Eine Abschätzung des Speicherbedarfs durch den oberen Grenzwert ist nicht sinnvoll, denn für ein Volumen aus 128^3 Voxeln bei 6 Tetraedern pro Würfel, die jeweils bis zu 2 Dreiecke erzeugen, mit jeweils 3 Eckpunkten (`float4`), würden 1,2GB Grafikspeicher benötigt. Im Verlauf der Berechnungen werden einige Kopien benötigt und lediglich 5-10% der Voxel sind aktiv. Deshalb wurde eine adaptive Speicherverwaltungsstrategie implementiert, die so wenig und so selten wie möglich Speicher reserviert: Jeweils nach bekannt werden der Anzahl an aktiven Würfeln, Dreiecken und Schwerpunkten wird doppelt so viel Speicher wie benötigt reserviert. Der Speicher wird nur dann frei gegeben und erneut reserviert, wenn die bereits reservierte Menge nicht ausreicht. Je nach Datensatz werden mit dieser Strategie lediglich 200MB bis 300MB Grafikspeicher für die gesamte Anwendung benötigt.

6 Ergebnisse und Bewertung

In diesem Kapitel werden die Ergebnisse in Form von Erkenntnissen und Benchmarks diskutiert. Dabei wird ausführlich auf Teilergebnisse der realisierten Lösung eingegangen und falls möglich mit anderen Lösungen verglichen.

Alle Benchmarks wurden auf einem Intel Core i7 950 mit einer GeForce 580 GTX auf Windows 7 Professional 64-Bit durchgeführt. Alle Programme wurden mit Visual Studio 2010 Ultimate SP1, dem CUDA Toolkit 4.0.17 kompiliert. Vor der Durchführung der Benchmarks wurde darauf geachtet, dass das System bereits warm gelaufen war und nicht unter Last stand. Der verwendete Datensatz und die Volumengröße sind jeweils angegeben.

6.1 Bezug zu den Arbeitsthesen

Das im Rahmen dieser Arbeit realisierte Verfahren erkennt zeitliche Ereignisse auf Proteinoberflächen ohne Benutzereingabe unter der Voraussetzung, dass das vorgeschaltete Volumen-Splatting korrekt ist. Trägt der Benutzer unpassende Splatting-Parameter ein, ist die Oberflächendefinition möglicherweise verfälscht und somit irreführend. Die von der Arbeit mit Dreiecksnetzen erhofften Vorteile können nur teilweise bestätigt werden. Einerseits wurde durch die Identität der Tetraederkanten eine Optimierung ermöglicht. Andererseits benötigt das Dreiecksnetz bei maximaler Auflösung nahezu die gleiche Menge an Speicher wie das ursprüngliche Volumen. Abbildung 6.1 zeigt das Dreiecksnetz und das Referenzbild im Vergleich. Für Entwickler gibt es einige Hilfsdarstellungen (siehe Abbildung 6.2). Die Arbeit mit der GPU Technologie gestaltete sich erwartungsgemäß deutlich anders als die Arbeit mit CPU Technologien. Besonders zeitraubend war die Fehlersuche, wenn die CUDA Laufzeitumgebung lediglich „Unknown Error“ mitteilte oder der Computer unerwartet einen Neustart auslöste.

6.2 Wirkungen von Optimierungen

Eine Überraschung ist die Optimierbarkeit von CUDA Kernels. Im Folgenden wird die Auswirkung von Optimierungen bezogen auf Marching Tetrahedrons beschrieben (siehe Abbildung 6.3). Der Entwicklungsverlauf ist anhand von Subversion-Revisionen, die sequenziell und aufsteigend im Verlauf der Zeit nummeriert werden, datiert:

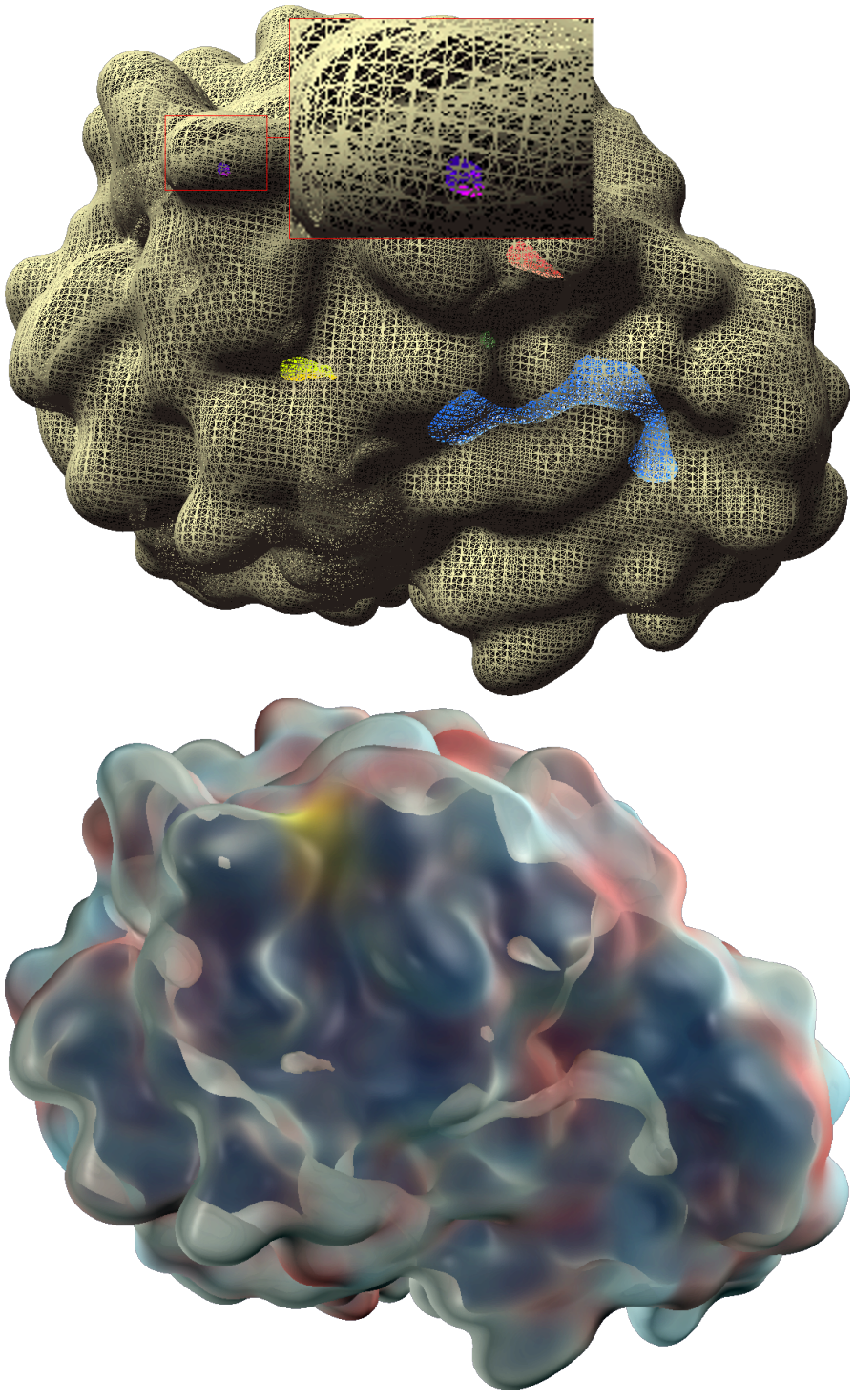


Abbildung 6.1: Drahtgittermodell (oben) und mittels Ray-Tracing erzeugtes Referenzbild (unten).

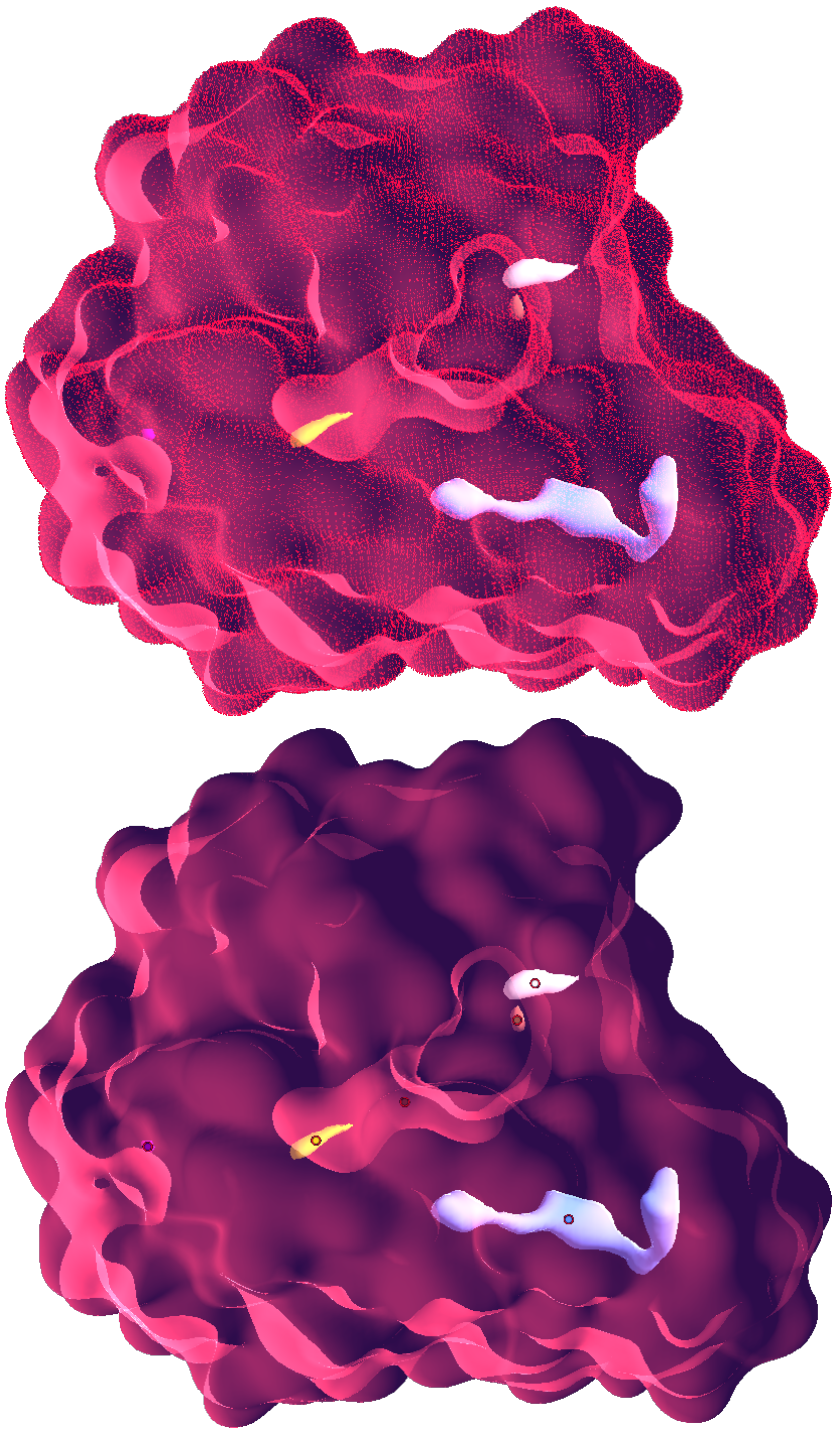


Abbildung 6.2: Hilfsvisualisierungen der Vertex-Normalen als ‚Büschel' auf der Oberfläche (oben) und alle Schwerpunkte der jeweiligen Komponenten als eingefärbte Kreise mit rotem Rand (unten).

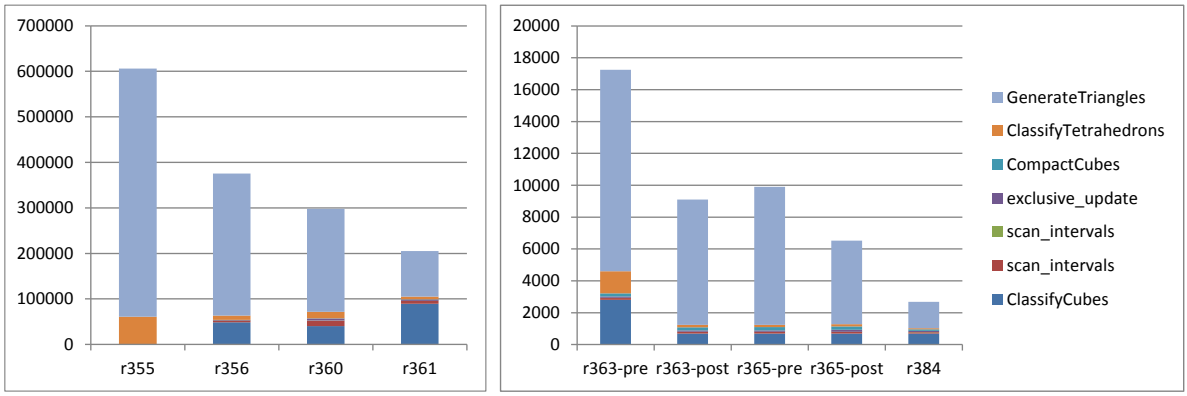


Abbildung 6.3: Durchschnittliche Laufzeiten (μs) der CUDA Kernels über den Entwicklungszeitraum in Revisionen: links Debug-Builds, rechts Release-Builds. `-pre` und `-post` stehen für ‚vor' und ‚nach' der Optimierung innerhalb einer Revision.

r355 → r356 Die hier erzielte Halbierung der Laufzeit wurde durch die in Kapitel 5 beschriebene Verdichtung erzielt.

r356 → r360 Als Basis diente ursprünglich eine Marching Cubes Implementierung, bei der sechs Tetraeder in einem Würfel von einem CUDA Thread berechnet werden - obwohl die Tetraeder in einem Würfel unabhängig voneinander sind. Deshalb wurde zunächst die Schleife parallelisiert, ohne das Speicherlayout zu verändern

r360 → r361 In diesem Arbeitsschritt wurde das Speicherlayout auf Tetraeder angepasst, sodass pro Tetraeder statt 8 Zugriffen auf die Würfeleckpunkte lediglich 4 Zugriffe auf die Tetraedereckpunkte benötigt wurden. Dieser und der vorhergehende Arbeitsschritt führten wieder zu einer Halbierung der Laufzeit.

r360 → r363-pre Die geplanten algorithmischen Optimierungen waren an diesem Punkt ausgeschöpft. Eine Umstellung von Debug- auf Release-Modus brachte weitere 15% Laufzeit.

r363-pre → r365-post Sehr gute Erfahrungen konnten bei der Umstellung von Local auf Shared Memory gemacht werden - insbesondere bei der Umstellung von Lookup-Tabellen. Die erneute Halbierung der Laufzeit wurde erzielt, indem der PTX-Assembler-Code auf `ld.local` und `st.local` durchsucht wurde und der entsprechende Code durch `__shared__` ergänzt wurde. In der Praxis ist das etwas komplizierter aufgrund des veränderten Gültigkeitsbereichs und der Synchronisation mehrerer Threads.

r363-post → r365-pre Der hier entstandene Laufzeitverlust resultierte aus einer Fehlerkorrektur: Bei der Berechnung der Normalen wurde nicht richtig interpoliert.

r365-post → r384 Die letzte Halbierung der Laufzeit ist zum Teil durch Optimierung der Konfiguration zu erklären, die allerdings weniger beisteuerte als erwartet, da die Größe des CUDA Grids bereits alle Multiprozessoren auslastete. Der größere Teil entstand durch Optimierungen auf Instruktionsebene, wie sie in *Chapter 5. Performance Guidelines* [Nvi] beschrieben sind: Entfernen von Branching, Unrolling von Schleifen, Vermeiden von Division und Modulo, Vermeiden von unnötigen Konvertierungen (`int` ↔ `float`, `singed` ↔ `unsigned`).

Möglicherweise ist eine weitere Halbierung erreichbar, durch Umstellen auf linearen Speicher oder Teilen gemeinsamer Punkte der Tetraeder. Diese Optimierungen wurden aus Zeitgründen nicht implementiert, da das Zeitfenster von 2,6ms für Interaktivität mehr als ausreichend ist.

6.3 Leistungsverhalten

Im Vordergrund der Aufgabenstellung stand die Machbarkeit, weniger die besonders elegante Lösung eines bekannten Problems. Ein direkter Gesamtvergleich ist deshalb nicht möglich, weshalb Teillösungen, falls möglich, verglichen werden. Zunächst wird ein Vergleich zwischen der Marching Tetrahedrons Implementierung dieser Arbeit und der Marching Cubes Implementierung von NVIDIA gemacht. Ein Vergleich der Ergebnisse der Segmentierung mit veröffentlichten Ergebnissen von Harwick et al. [HLP10], ist nicht sinnvoll wegen des veränderten Prädikats und der veränderten Hardwareplattform. Für die darauf folgenden Schritte gibt es keine Vergleichsgrundlage, sodass lediglich Laufzeiten angegeben werden können.

In Tabelle 6.1 und Abbildung 6.4 können, wie erwartet, deutliche Unterschiede zwischen Marching Cubes und Marching Tetrahedrons festgestellt werden. Eigentlich müsste der `ClassifyCubes`-Kernel von Marching Tetrahedrons geringfügig schneller sein als der von Marching Cubes, denn die Verdichtung benötigt einen Texturzugriff weniger und ist in PTX Assembler auch kürzer. Eine mögliche Erklärung liefert der folgenden Unterschied: Die Marching Cubes Implementierung verwendet `tex1Dfetch()`, wohingegen die Marching Tetrahedrons Implementierung `tex3D()` verwendet. Ersteres dient dem Zugriff auf linearen und Letzteres dem Zugriff auf normalen Texturspeicher. Der Unterschied ist bezüglich des Leistungsverhaltens nur dürftig dokumentiert. Für Marching Cubes wird das Volumen von der Festplatte in linearen Speicher geladen. Für Marching Tetrahedrons wird das Volumen mittels Splatting als OpenGL-Volumen-Textur erzeugt. Eine Funktion `tex3Dfetch()` gibt es nicht, weswegen der Speicher kopiert werden müsste, um `tex1Dfetch()` verwenden zu können, was ebenfalls Zeit kostet. Eine andere mögliche Erklärung wäre das Texturformat, da es sich bei der OpenGL-Textur um eine Half-Float-Textur handelt, die bei jedem Zugriff konvertiert werden muss. `scan_invervals()` und `exclusive_update()` aus Thrust sowie

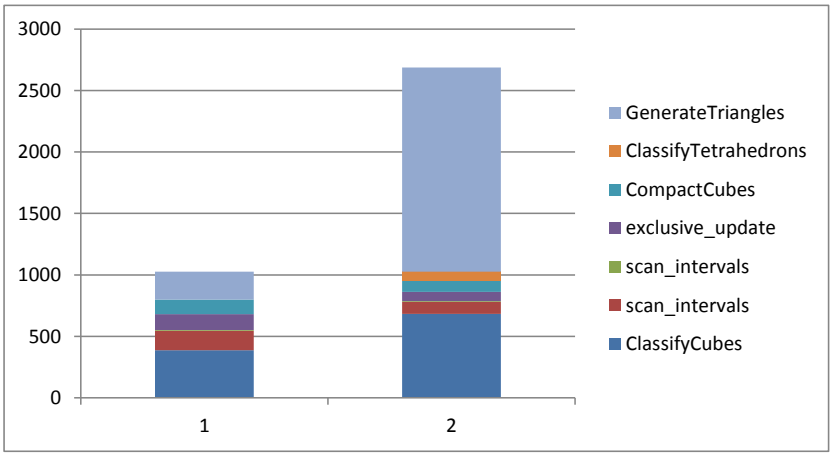


Abbildung 6.4: Vergleich zwischen Marching Tetrahedrons und Marching Cubes für ein 128^3 Volumen (Datensatz *2ve003*)

Kernel	Zeit für Marching Tetrahedrons (μs)	Zeit für Marching Cubes (μs)	Verhältnis (%)
ClassifyCubes	683,728	386,793	176,77
scan_intervals (1)	96,274	159,508	60,36
scan_intervals (2)	6,616	6,447	102,62
exclusive_update	75,215	129,163	58,23
CompactCubes	89,706	116,976	76,69
ClassifyTetrahedrons	75,575		
GenerateTriangles	1659,41	226,65	732,15
Summe	2686,524	1025,537	254,59

Tabelle 6.1: Vergleich zwischen Marching Tetrahedrons und Marching Cubes für ein 128^3 Volumen (Datensatz *2ve003*)

`CompactCubes()` weisen erwartungsgemäß ähnliche oder, aufgrund einer besseren Konfiguration, bessere Laufzeiten auf. `GenerateTriangles()` benötigt erwartungsgemäß mehr Zeit: Zum einen werden bei Marching Tetrahedrons mehr Dreiecke generiert, zum anderen approximiert die Marching Cubes Implementierung die Vertex-Normalen lediglich. Diese Approximation spart 6 Texturzugriffe pro Eckpunkt und resultiert in einer unsauberen Beleuchtung. Werden bei Marching Tetrahedrons nur Konstanten geschrieben und die Berechnung der Normalen deaktiviert, sinkt die durchschnittliche Laufzeit um $640,48\mu s$, was das Verhältnis auf $449,55\%$ verbessert.

Für die Messwerte des iterativen Tetraeder-Lablings in Tabelle 6.2 und der Schwerpunktberechnung Tabelle 6.3 ist wegen der fehlenden Vergleichsbasis kein Vergleich möglich. Die Schwerpunktberechnung nimmt unerwartet viel Zeit in Anspruch. Die Ur-

Kernel	Iteration	Zeit (μs)
Reset		14,463
Scan	1	183,421
Analysis	1	99,166
Labeling	1	28,287
Scan	2	185,469
Analysis	2	39,935
Labeling	2	30,848
Scan	3	180,38
Analysis	3	28,351
Labeling	3	28,416
Scan	4	179,868
Analysis	4	28,319
Labeling	4	28,448
Scan	5	179,996
	Summe	1235,367

Tabelle 6.2: Kernel-Timings des iterativen Tetraeder-Labelings für ein 128^3 Volumen (Datensatz *2ve003*)

	Zeit mit	
Kernel	Thrust 1.4 (μs)	Thrust 1.5 (μs)
CentroidMap	80,479	82,367
Sort & Reduce	4096,699	2732,432
CentroidFinalize	6,431	6,112
ColorizeByCentroid	114,686	115,87
Summe	4298,295	2936,781

Tabelle 6.3: Kernel-Timings der Schwerpunktberechnung für ein 128^3 Volumen (Datensatz *2ve003*)

sache hierfür ist in `thrust::sort_by_key()` und `thrust::reduce_by_key()` zu suchen. `thrust::reduce_by_key()` setzt sortierte Schlüssel voraus. `thrust::sort_by_key()` ist ein sehr komplexer Kernel, was besonders an der zum Kompilieren benötigten Zeit ($>$ 2 Minuten) spürbar ist. `thrust::reduce_by_key()` ist in Thrust 1.4 nicht direkt in CUDA implementiert. Dieser Makel wurde für Thrust 1.5 am 08.09.2011 (Revision *e8762d3fd466*, Issue 347) beseitigt. Version 1.5 wurde kurz vor Abgabe dieser Arbeit am 29.11.2011 veröffentlicht und halbiert die Laufzeit.

6.4 Praktischer Nutzen

Obwohl das Rendering nicht auf Leistung optimiert wurde, kann die in Abbildung 6.5 gezeigte Bilderserie mit 17-19 Bildern pro Sekunde animiert und mit 53-55 Bildern pro Sekunde statisch angezeigt werden. Ohne Volumen-Splatting und Rendering würden mit Thrust 1.5 insgesamt 6,86ms benötigt, was 145 Bildern pro Sekunde entspräche. In Werkzeugen wie GROMACS oder CAVER ist das Rendering ebenfalls interaktiv, allerdings benötigen sie für die Verarbeitung vor dem Rendering einige Sekunden. Der Fortschritt in der Handhabung von animierten Simulationsdaten ist entsprechend spürbar und offensichtlich.

Der momentane praktische Nutzen von MegaMol Protein für die Grundlagenforschung konnte im Rahmen eines Gespräches mit einem Anwender - vom Institut für Technische Biochemie der Universität Stuttgart - grob abgeschätzt werden. Im Folgenden sind Gesprächspunkte stichpunktartig beschrieben:

- MegaMol wird zur Interpretation von Simulationsergebnissen benutzt.
- Ein unbekannter Anteil der Proteine verschließt sich bei einer Interaktion mit dem Substrat. Für diese Proteine können zeitlich korrelierte Komponenten direkt dem ‚Weg des Substrats' entsprechen.
- Ein ebenfalls unbekannter Anteil der Proteine etabliert Tunnels zum aktiven Zentrum, die sich nie von der Oberfläche entkoppeln. Für die Beurteilung sind Größe, Radius und Auftrittsdauer relevant, was eine sinnvolle Erweiterung wäre.
- Es gibt Tunnel, wodurch ein Substrat das aktive Zentrum erreicht, obwohl der Tunnel eigentlich zu schmal ist. ‚Tunnel-Flexibilität' wäre eine interessante Erweiterung.
- Es ist bisher sehr schwierig, die Interaktion von Substrat und Protein zu beurteilen ohne beide gleichzeitig zu sehen. Ein sinnvoller Ansatz wäre das Filtern des Substrats nach Semantik oder Interaktion.
- Das Bobachten von langen oder schnellen Animationen erfordert eine hohe Aufmerksamkeit, weil Größenänderungen von Tunnels, Helices oder Faltblättern nicht hervorgehoben werden.
- Die Benutzbarkeit von MegaMol würde verbessert werden, falls die aufwändige Konfiguration über Textdateien entfällt.

MegaMol ist nach aktuellen Stand in der Praxis einsetzbar.

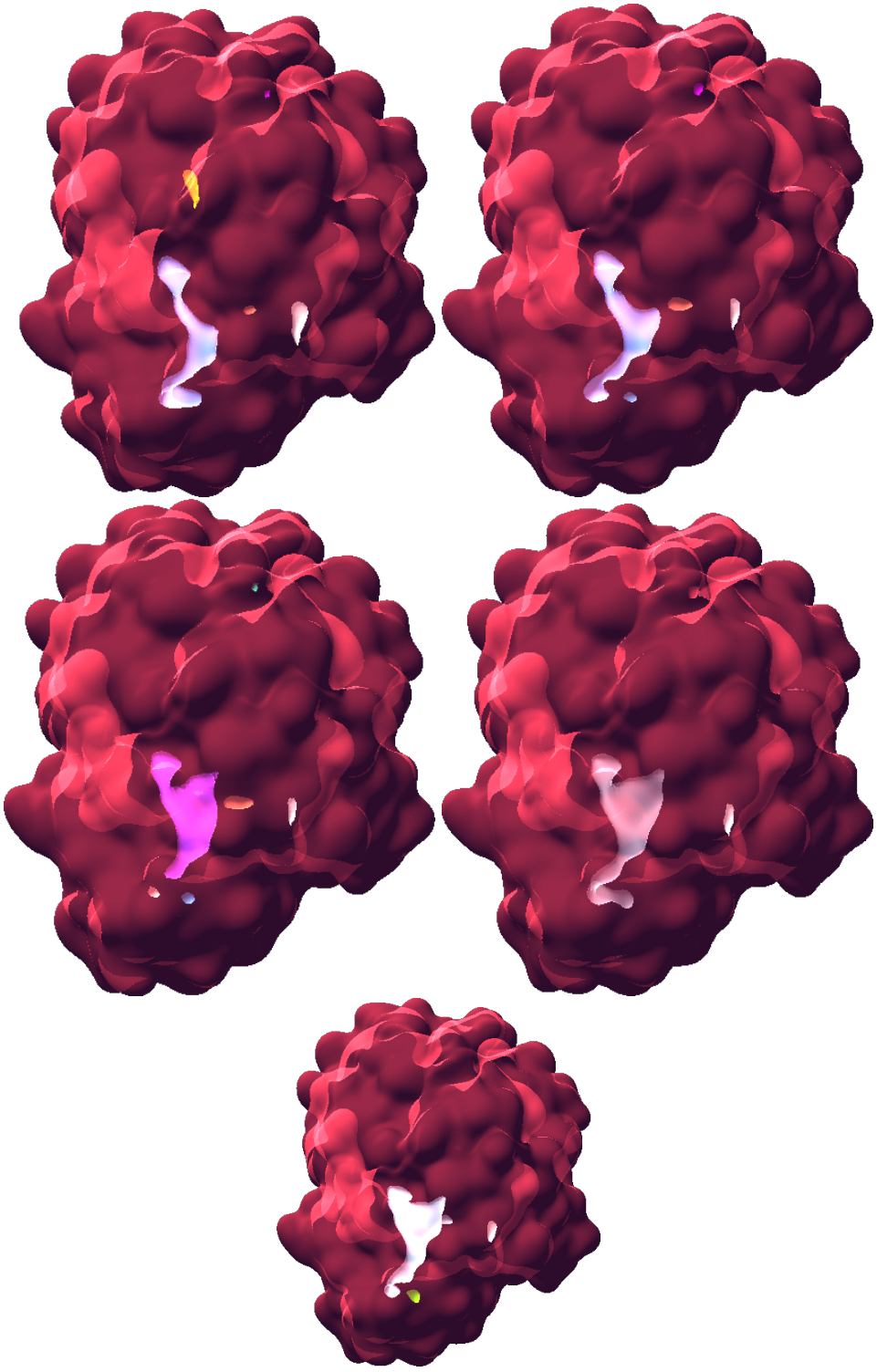


Abbildung 6.5: Von links nach rechts und oben nach unten: Einzelbilder einer Animation des Datensatzes *2ve003*, bei einer Volumenauflösung von 128^3 Voxeln, mit Blending aktiviert und nach Komponenten über Zeit korreliert eingefärbt.

7 Fazit

Im Rahmen dieser Arbeit wurde gezeigt, dass der interaktive Umgang mit Proteinoberflächenstrukturen aus zeitabhängigen Molekulardynamik-Simulationsdatensätzen auf moderner Grafikhardware möglich ist. Von Beginn der Arbeit an, erschien ein Ansatz über ein Dreiecksnetz vielversprechend. Allerdings war unklar, welche Algorithmen zum Ziel führen und ob das Leistungsverhalten zufriedenstellend sein würde, denn vorhandene Werkzeuge benötigten für solche Berechnungen mehrere Sekunden. Die Arbeit wurde in vier Iterationen unterteilt.

In der ersten Iteration wurden verschiedene Möglichkeiten untersucht, um ein Dreiecksnetz aus Volumendaten zu generieren. Nachdem sich herausstellte, dass der Marching-Tetrahedrons-Algorithmus geeignet und leicht zu erweitern ist, fand eine erste Auseinandersetzung mit CUDA und dem MegaMol-Framework statt. Die anfänglich fehlende, praktische Erfahrung mit CUDA und dem damit einhergehenden Remote-Debugging verlangsamte die Entwicklung die ersten Wochen merklich. Später konnte gut mit den entsprechenden Werkzeugen gearbeitet werden.

In der zweiten Iteration wurden eine einfache grafische Ausgabe und Unterstützung für animierte Datensätze implementiert. Die Laufzeit war anfangs nicht akzeptabel, weshalb einige Optimierungen eingepflegt wurden, deren Wirkung genauer in Kapitel 6 beschrieben ist.

In der dritten Iteration fand auf der Suche nach einem Verfahren zur Segmentierung der Dreiecke ein Exkurs in die Objekterkennung statt. Nachdem ein passendes Verfahren gefunden war, dauerte die Implementierung nicht lange. Ein Versuch zur zeitlichen Korrelation der Komponenten über Schwerpunkte mittels CPU schlug, aufgrund zu hoher Laufzeit, fehl. Der Versuch zeigte allerdings, dass die nachfolgende Klassifizierung zeitlicher Ereignisse wie gedacht funktionierte.

In der vierten und letzten Iteration wurde eine verbesserte Lösung für die zeitliche Korrelation implementiert. Außerdem wurde Zeit für Feinarbeit aufgewendet: Es wurde mit Blending und Beleuchtung experimentiert. Außerdem wurde die Visualisierung der Komponenten und Schwerpunkte verbessert.

Im Verlauf der Arbeit wuchs das Verständnis für das Problem und die CUDA-Technologie. Am Ende konnten alle Berechnungen in unter 7ms durchgeführt werden. Das persönliche Ziel, dass im Rahmen der Diplomarbeit ein Werkzeug (weiter-)entwickelt wird, mit dem andere Menschen arbeiten können, ist erfüllt: Es gibt einen praktischen Nutzen und es wurde ein Grundstein für weitere Arbeiten gelegt.

Ausblick

Die Interaktive Analyse von Molekulardynamik-Simulationen ist noch in einem frühen Entwicklungsstadium. In der Grundlagenforschung werden Molekulardynamik-Simulationen entwickelt, um ein besseres Verständnis von Proteinen zu entwickeln. Diese Art des konstruktivistischen Forschens und Lernens profitiert in hohem Maße von Interaktivität. Eine Weiterentwicklung der Oberflächenvisualisierung von MegaMol Protein erscheint deshalb sinnvoll. Im Folgenden sind einige entsprechende Erweiterungsmöglichkeiten aufgelistet:

- Erkennung von an die Außenhülle gekoppelten Taschen und Tunnel als eigene Komponenten.
- Entwicklung einer Visualisierung für Oberflächenflexibilität.
- Ausmessen von Tunnel und Taschen.
- Berechnung des Schwellwerts (Fehlerquelle) für Marching-Tetrahedrons.
- Visualisierung von Änderungen, um die Aufmerksamkeit des Benutzers zu lenken.

Die Simulationssoftware GROMACS bietet nur begrenzte Unterstützung für GPGPU, sodass für die eigentliche Simulation noch ein klassischer Supercomputer benötigt wird. Das interaktive ‚Zusammenstecken und Analysieren' von Molekulardynamik-Simulationen wird auf absehbare Zeit deswegen voraussichtlich nicht machbar sein. Eine Kooperation mit den GROMACS-Entwicklern könnte diesem Ziel dienlich sein.

Literaturverzeichnis

[ABC+06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. (Zitiert auf Seite 13)

[BEB+11] P. Beneš, C. Eva, K. Barbora, P. Antonín, S. Ondřej, B. Jan, Šustr Vilém, K. Martin, S. Tibor, G. Artur, Z. Matúš, B. Lada, M. Petr, D. Jiří, S. Jiří. CAVER 2.1. http://www.caver.cz, 2009-2011. (Zitiert auf Seite 21)

[BJN10] G. J. Byungil Jeong, Greg Abram, P. Navrátil. Large-Scale Isosurfacing on a Distrubuted GPU Cluster. SC 2010, NVIDIA GPU Computing Theater, 2010. (Zitiert auf Seite 36)

[Bou94] P. Bourke. Polygonising a scalar field, 1994. URL http://paulbourke.net/geometry/polygonise/. (Zitiert auf Seite 36)

[BS00] G. P. Brady, P. F. Stouten. Fast prediction and visualization of protein binding pockets with PASS. *Journal of computeraided molecular design*, 14(4):383–401, 2000. URL http://www.ncbi.nlm.nih.gov/pubmed/10815774. (Zitiert auf Seite 23)

[CMS06] H. Carr, T. Moller, J. Snoeyink. Artifacts Caused by Simplicial Subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 12:231–242, 2006. URL http://dx.doi.org/10.1109/TVCG.2006.22. (Zitiert auf Seite 26)

[Cor] N. Corporation. NVIDIA GeForce GTX 580 and NVIDIA GeForce GTX 580. URL http://www.nvidia.de/. (Zitiert auf Seite 17)

[CS06] R. G. Coleman, K. A. Sharp. Travel depth, a new shape descriptor for macromolecules: application to ligand binding. *J Mol Biol*, 362(3):441–458, 2006. URL http://dx.doi.org/10.1016/j.jmb.2006.07.022. (Zitiert auf Seite 24)

[FB10] D. Feltell, L. Bai. A new marching cubes algorithm for interactive level set with application to MR image segmentation. In *Proceedings of the 6th international conference on Advances in visual computing - Volume Part I*, ISVC'10, pp. 371–380. Springer-Verlag, Berlin, Heidelberg, 2010. URL http://dl.acm.org/citation.cfm?id=1939921.1939963. (Zitiert auf Seite 29)

[Gro10] S. Grottel. The MegaMol. Technical report, VISUS Universität Stuttgart, 2010. (Zitiert auf Seite 31)

[HB10] J. Hoberock, N. Bell. Thrust: A Parallel Template Library, 2010. URL http://www.meganewtons.com/. Version 1.3.0. (Zitiert auf Seite 36)

[HG08] B. Ho, F. Gruswitz. HOLLOW: Generating Accurate Representations of Channel and Interior Surfaces in Molecular Structures. *BMC Structural Biology*, 8(1):49, 2008. URL http://www.biomedcentral.com/1472-6807/8/49. (Zitiert auf Seite 23)

[HLP10] K. A. Hawick, A. Leist, D. P. Playne. Parallel graph component labelling with GPUs and CUDA. *Parallel Comput.*, 36:655–678, 2010. URL http://dx.doi.org/10.1016/j.parco.2010.07.002. (Zitiert auf den Seiten 39 und 47)

[HS06] B. Huang, M. Schroeder. LIGSITEcsc: predicting ligand binding sites using the Connolly surface and degree of conservation. *BMC Structural Biology*, 6(1):19, 2006. URL http://www.biomedcentral.com/1472-6807/6/19. (Zitiert auf den Seiten 22 und 23)

[KBE08] M. Krone, K. Bidmon, T. Ertl. GPU-based Visualisation of Protein Secondary Structure. In *TPCG'08*, pp. 115–122. 2008. (Zitiert auf den Seiten 12 und 32)

[KFR+11] M. Krone, M. Falk, S. Rehm, J. Pleiss, T. Ertl. Interactive Exploration of Protein Cavities. *Computer Graphics Forum*, 30(3), 2011. (Zitiert auf den Seiten 7, 12, 23 und 32)

[KGE11] M. Krone, S. Grottel, T. Ertl. Parallel Contour-Buildup Algorithm for the Molecular Surface. *Proceedings of IEEE Symposium on Biological Data Visualization (biovis'11)*, 2011. (Zitiert auf Seite 23)

[KW05] P. Kipfer, R. Westermann. GPU Construction and Transparent Rendering of Iso-Surfaces. In G. Greiner, J. Hornegger, H. Niemann, M. Stamminger, editors, *Proceedings Vision, Modeling and Visualization 2005*, pp. 241–248. IOS Press, infix, 2005. (Zitiert auf den Seiten 26 und 27)

[LBH11] N. Lindow, D. Baum, H.-C. Hege. Voronoi-Based Extraction and Visualization of Molecular Paths. *IEEE Transactions on Visualization and Computer Graphics*, 17:2025–2034, 2011. doi:http://doi.ieeecomputersociety.org/10.1109/TVCG.2011.259. (Zitiert auf den Seiten 21 und 22)

[LC87] W. E. Lorensen, H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169, 1987. URL http://doi.acm.org/10.1145/37402.37422. (Zitiert auf Seite 25)

[LL52] K. Linderstrøm-Lang. *Proteins and enzymes. (Lane medical lectures, 1951)....* Stanford university press, 1952. URL http://books.google.com/books?id=9z8BPAAACAAJ. (Zitiert auf Seite 12)

[LWE98] J. Liang, C. Woodward, H. Edelsbrunner. Anatomy of protein pockets and cavities: Measurement of binding site geometry and implications for ligand design. *Protein Science*, 7(9):1884–1897, 1998. URL http://dx.doi.org/10.1002/pro.5560070905. (Zitiert auf Seite 24)

[Nvi] Nvidia Corporation. *NVIDIA CUDA C Programming Guide*. (Zitiert auf den Seiten 15, 16, 17, 18 und 47)

[NY06] T. Newman, H. Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, 2006. URL http://linkinghub.elsevier.com/retrieve/pii/S0097849306001336. (Zitiert auf Seite 25)

[PKKO07] M. Petřek, P. Košinová, J. Koča, M. Otyepka. MOLE: A Voronoi Diagram-Based Explorer of Molecular Channels, Pores, and Tunnels. *Structure*, 15(11):1357–1363, 2007. URL http://dx.doi.org/10.1016/j.str.2007.10.007. (Zitiert auf den Seiten 21 und 22)

[SHI05] H. SHIN'YA. Discovery of chignolin, a "protein"consisting of only ten amino acids. *Tanpakushitsu kakusan koso. Protein, nucleic acid, enzyme*, 50:427–433, 2005. (Zitiert auf Seite 11)

[tit11] Q8WZ42, 2011. URL http://www.uniprot.org/uniprot/Q8WZ42. (Zitiert auf Seite 11)

[TU10] M. Till, G. Ullmann. McVol - A program for calculating protein volumes and identifying cavities by a Monte Carlo algorithm. *Journal of Molecular Modeling*, 16:419–429, 2010. URL http://dx.doi.org/10.1007/s00894-009-0541-y. 10.1007/s00894-009-0541-y. (Zitiert auf Seite 23)

[VKALF10] G. B. Vitor, A. Körbes, R. de Alencar Lotufo, J. V. Ferreira. Analysis of a Step-Based Watershed Algorithm Using CUDA. *IJNCR*, 1(4):16–28, 2010. URL http://dblp.uni-trier.de/db/journals/ijncr/ijncr1.html. (Zitiert auf Seite 29)

[Wir07] O. Wirjadi. Survey of 3D image segmentation methods. Technical Report 123, Fraunhofer-Institut für Techno- und Wirtschaftsmathematik, 2007. URL http://kluedo.ub.uni-kl.de/volltexte/2008/2213/pdf/bericht123.pdf. (Zitiert auf Seite 27)

[WPS07] M. Weisel, E. Proschak, G. Schneider. PocketPicker: analysis of ligand binding-sites with shape descriptors. *Chemistry Central Journal*, 1(1):7, 2007. URL http://journal.chemistrycentral.com/content/1/1/7. (Zitiert auf den Seiten 22 und 23)

Alle URLs wurden zuletzt am 01.12.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Christoph Schulz)