Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Paralelizace geometrických výpočtů pro analýzu biomolekul

Plzeň 2016 Václav Löffelmann

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16. června 2016

Václav Löffelmann

Abstract

This work examines parallel implementation of exploring the molecular surface. The new implementation is in OpenCL language and improves current implementation by 60% for largest datasets. Initial part sums up a theory about Voronoi diagrams and its dual structure. Following parts targeting difference between current and new - parallel - implementation. At the end are presented detailed results and a comparison of both implementations.

Abstrakt

Bakalářská práce se zabývá paralelním výpočtem molekulárního povrchu v programovacím jazyku OpenCL. Nová implementace dosahuje až 60% zrychlení pro nejobjemnější výpočty. V úvodní části je probrána nezbytná teorie Voronoi diagramů a jejich duální struktury. Následuje shrnutí současné implementace a popis nové, paralelní. V závěru nechybí detailní porovnání výsledků obou implementací napříč různými testovacími daty.

Poděkování

Na tomto místě bych chtěl poděkovat panu Mgr. Martinu Maňákovi za nepřeberné množství podnětů k vylepšení práce a odborného vedení v celém jejím průběhu.

Obsah

1 Úvod **1**

2 Voronoi diagramy **2**
2.1 Aditivně vážené Voronoi diagramy . . . 3
2.2 Voronoi diagramy a Delaunayova triangulace . . . 4
2.3 Algoritmy pro konstrukci Voronoi diagramů . . . 4
2.3.1 Algoritmus trasování hran . . . 5

3 Paralelní výpočty na současném hardwaru **7**
3.1 CPU/APU . . . 7
3.2 Grafické karty . . . 7
3.3 Ostatní výpočetní koprocesory . . . 8
3.4 Programové nástroje . . . 8

4 Jazyk OpenCL **9**
4.1 Výpočetní model . . . 9
4.2 Hierarchie paměti . . . 10
4.3 Některá omezení jazyka . . . 11

5 Výpočet molekulárního povrchu **12**
5.1 Van der Waals model . . . 12
5.2 Geometrický popis molekulárního povrchu . . . 13
5.2.1 Sférické pláty . . . 13
5.2.2 Sférické trojúhelníky . . . 13
5.2.3 Anuloidy . . . 14
5.2.4 Toroidální pláty . . . 14

6 Výpočty v biomolekulách vhodné pro paralelizaci **15**
6.1 Sestavení Voronoi diagramu a Delaunayovi triangulace . . . 15
6.2 Nalezení nejužších míst mezi atomy v molekule . . . 15
6.3 Algoritmus pro hledání geometrických primitiv molekulárního povrchu . . . 15
6.3.1 Současný algoritmus implementovaný na CPU . . . 15
6.3.2 Detekce kolizí geometrických primitiv . . . 16

7 Paralelní implementace v OpenCL **17**
7.1 Hledání sférických trojúhelníků . . . 17
7.1.1 Vstup algoritmu . . . 17
7.1.2 Výpočet . . . 18
7.1.3 Výstup . . . 18
7.2 Hledání sférických plátů . . . 18
7.3 Hledání torů . . . 19
7.4 Ořezávací roviny sférických trojúhelníků . . . 19

7.4.1 Paralelní prefix sum 19
7.4.2 Vstup algoritmu 19
7.4.3 Výpočet 21
7.5 Spuštění programu 22
7.5.1 Minimální a doporučené požadavky 22
7.5.2 Parametry programu 22
7.5.3 Závislosti 23
7.5.4 Sestavení programu 23
7.5.5 Vizualizace výsledků 24

8 Porovnání implementací **25**
8.1 Molekuly použité v testech 25
8.2 Výsledky měření současné implementace 26
8.3 Výsledky měřené nové implementace na CPU 26
8.4 Výsledky měřené nové implementace na GPU 26
8.5 Diskuse výsledků 28

9 Možnosti dalšího rozšiřování **30**

10 Závěr **31**

1 Úvod

S rozvojem biochemie vznikla potřeba důkladné analýzy makromolekul, které se skládají až z několik stovek tisíc atomů. Výpočet některých vlastností, například molekulárního povrchu, důležité pro výzkum interakcí mezi jednotlivými molekulami. Další zkoumanou strukturou jsou dutinky, které vznikají uvnitř molekul. Informace o těchto dutinkách jsou klíčové pro výzkum nových materiálů, léků a nebo nových vlastností proteinů. Pro některé prostorové výpočty se využívají Voronoi diagramy, které slouží k popisu prostorových vztahů mezi atomy v molekule. Tyto diagramy definují rozdělení prostoru tak, že každý bod prostoru přiřadí nejbližšímu zadanému bodu - generátoru. Některé části výpočtů molekulárních vlastností lze dobře paralelizovat.

Cílem této práce je najít tyto části algoritmů a implementovat je takovým způsobem, aby bylo možné využít masivního paralelizmu, který nabízejí grafické karty a nebo jiné výpočetní koprocesory. Hlavním problémem, na který se práce zaměří bude výpočet molekulárního povrchu. Tento povrch popíšeme jako množiny sférických trojúhelníků a toroiádlních i sférických plátů, které by vytvořila sférická sonda zvoleného poloměru při průzkumu molekuly.

Ve druhé sekci zadefinujeme Voronoi diagramy a základní pojmy z této oblasti. Dále bude obecná definice rozšířená o váhu generátoru a tím vzniklé vážené Voronoi diagramy. Následuje vztah Voronoi diagramů a Delaunayovi triangulace. Zkoumání těchto struktur bude uskutečněno z pohledu analytických výpočtů v biomolekulách. Třetí sekce nastíní problematiku paralelních výpočtů na současném hardwaru, vyzdvihne přednosti výpočtů na grafických kartách a upozorní na problémy spojené s vývojem programů pro ně. Dále si zadefinujeme geometrická primitiva která tvoří molekulární povrch. Následující sekce bude věnována částem algoritmů, které se používají k analýze biomolekul a které budou vybrány pro experimentální implementaci pro výpočty v paralelním prostředí, resp. na grafických kartách. Krátce také představíme jazyk OpenCL, používaný pro paralelní výpočty.

V sedmé kapitole vysvětlíme implementace realizovaných algoritmů a v krátkosti je popíšeme. Nebude chybět ani uživatelský manuál k vytvořenému programu.

Příští kapitolu věnujeme měření výkonnosti původní a nové implementace. Novou implementaci měříme jak na grafické kartě, tak i na procesoru. Všechna měření porovnáme a zhodnotíme, kdy se vyplatí výpočty provádět novým programem. Výsledky ukazují, že pro zkoumání velkých molekul za použití sondy malého poloměru je nová implementace v OpenCL až dvojnásobně rychlejší.

Vzhledem ke složitosti problematiky si práce neklade za cíl věnovat se všem detailům, které souvisí s podrobnou geometrickou analýzou biomolekuly.

2 Voronoi diagramy

Účelem Voronoi diagramu je rozdělení roviny, resp. prostoru, které přiřazuje část prostoru vždy právě jednomu, nejbližšímu, bodu. Tyto diagramy lze použít při řešení různorodých problémů. Například uveďme geografické rozdělení oblastí pro určení spádových oblastí. Dalším problémem, kde může Voronoi diagram pomoci je problém nalezení nejbližšího sousedního bodu nebo několika nejbližších sousedů. Podobným problémem může následně být vytvoření minimální kostry grafu. V rámci analýzy molekul Voronoi diagramy používáme při sestavování triangulace, hledání nejbližších sousedů. A Voronoi vrcholy zkoumáme jako body, kde je největší oblast mezi atomy. Tyto body jsou pro nás významné při hledání dutinek v molekulách.

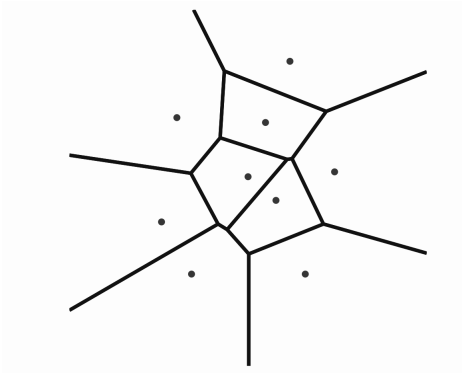
Voronoi diagram se skládá z množiny *Voronoi regionů* R_i. Regiony obecně definujeme rovnicí (1) [7, 24].

$$R_i = \{\vec{x} : d(\vec{p_i}, \vec{x}) \leq d(\vec{p_j}, \vec{x})\} \tag{1}$$

Kde $\vec{p_i}$ značí zadané body v prostoru (též značené jako generátory), pro které definujeme Voronoi diagram a $i, j \in \{1, 2, ..., n\}$, kde n je počet generátorů. Funkce d značí vzdálenost dvou bodů. Pro určení vzdálenosti můžeme obecně zvolit libovolnou normu, např. eukleidovskou nebo manhattanskou vzdálenost. Matematický zápis tedy definuje pro každý generátor prostor, kde každý bod tohoto prostoru je danému generátoru vzdálený maximálně tak, jak je vzdálený libovolnému jinému generátoru.

Výsledný Voronoi diagram ve 2D ukazuje obrázek 1.

V $\mathbb{R}^2$ dále definujme *Voronoi hranu*. Voronoi hrana je úsečka nebo polopřímka, která leží přesně na hranici mezi Voronoi oblastmi, je tedy ekvidistantní k příslušným generátorům. V místě, kde se setkávají Voronoi hrany, vzniká Voronoi *vrchol*, který je rovněž ekvidistantní k náležícím generátorům. Tento bod je společným krajním bodem Voronoi hran, které se v daném místě setkávají. V $\mathbb{R}^3$ máme místo Voronoi hran celé Voronoi stěny, které definují rozhraní mezi jednotlivými regiony.



Obrázek 1: Voronoi diagram ve 2D [12].

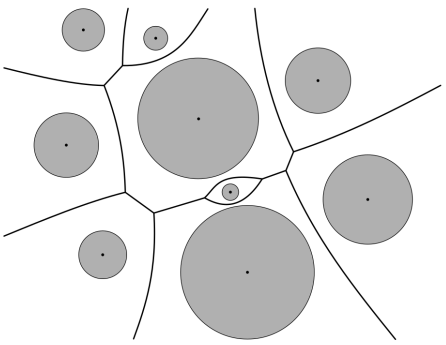
Zaveďme ještě pojem *sousední Voronoi region*. Dva Voronoi regiony považujeme za sousední právě tehdy, když sdílí alespoň jednu Voronoi hranu resp. stěnu.

2.1 Aditivně vážené Voronoi diagramy

Jestliže nebudeme uvažovat vstupní množinu jako množinu bodů, ale jako množinu sfér, musíme definici $\vec{p_i}$ rozšířit o poloměr, takzvanou váhu. Tuto váhu každému bodu přiřazuje funkce w. V $\mathbb{R}^3$ tedy definujeme $\vec{p_i} = (x_i, y_i, z_i)$. Náležitým způsobem také musíme upravit definici Voronoi oblastí, jak je znázorněno v rovnici (2) [24].

$$R_i = \{\vec{x} \in \mathbb{R}_3 : d(\vec{p_i}, \vec{x}) - w(i) \leq d(\vec{p_j}, \vec{x}) - w(j)\} \tag{2}$$

Stejně jako definice základního Voronoi diagramu jsou i zde uvažovány $i, j \in \{1, 2, ..., n\}$ a n jako počet generátorů s přidaným poloměrem - váhou. Změnou je odečtení váhy příslušného generátoru od vzdáleností bodů Voronoi regionu a generátoru. Voronoi hrana tedy již nebude ekvidistantní od středů generátorů, nýbrž bude ekvidistantní mezi sférami, jak je znázorněno na obrázku 2. Vzdálenost mezi sférami se pak nazývá *aditivní vážená vzdálenost.*



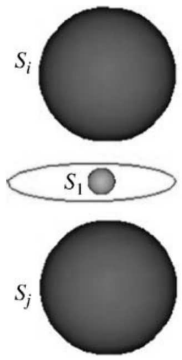
Obrázek 2: Aditivně vážený Voronoi diagram [12].

Z rozšíření definice plynou některé změny oproti neváženým Voronoi diagramům. Zatímco v obyčejném Voronoi diagramu jsou Voronoi hrany pouze přímkami, nebo jejich částmi, v aditivně váženém Voronoi diagramu je situace složitější. Obecně totiž Voronoi hrana ve váženém Voronoi diagramu může být jakákoliv kvadrika. Speciální případ, kdy voroni hrana je elipsou, je znázorněno na obrázku 3. Jak je na obrázku vidět, kruh vznikne tehdy, pokud se mezi dvěma sférami S_i a S_j nachází ještě jedna menší sféra S_1. Na obrázku 2 si lze také povšimnout rozdělení Voronoi hrany mezi dvěma sférami o velkém poloměru další sférou s menším poloměrem.

2.2 Voronoi diagramy a Delaunayova triangulace

S Voronoi diagramem úzce souvisí Delaunayova triangulace. Na obrázku 4 je znázorněn jejich vztah. Voronoi diagram je vyznačen plnými čarami a Delaunayovo triangulace čárkovaně. Dle [18], lze Delaunayova triangulace zadefinovat jako uskupení trojúhelníků, které vznikne propojením polohy generátorů ze sousedních Voronoi regionů, kde tyto trojúhelníky jsou konvexní obálkou zadaných bodů. Vzhledem ke vztahu Delaunayovo triangulace a Voronoi diagramů je nazveme duální strukturou.

Nutno podotknout, že o validní triangulaci se jedná pouze v případě, že neuvažujeme generátory s váhou. V případě použití aditivně vážených Voronoi diagramů hovoříme o takzvané kvazi triangulaci [15]. Mohou totiž nastat případy, kdy prvky triangulace sdílí více než jednu hranu. To porušuje definici triangulace, která nedovoluje mezi dvěma prvky triangulace sdílení více než jedné hrany v $\mathbb{R}^2$ resp. stěny v $\mathbb{R}^3$. Nicméně při využití v molekulárních výpočtech se nevalidní čtyřstěny v rámci triangulace vyskytují pouze výjimečně [15].



Obrázek 3: Případ Voronoi hrany jako kružnice [15]. Problém nazývaný Big brothers, kdy se mezi dvěma velkými atomy (S_i a S_j) nachází ještě jeden významně menší (S_1).

2.3 Algoritmy pro konstrukci Voronoi diagramů

Pro konstrukci Voronoi diagramů existuje několik algoritmů [4]. Jejich různé přístupy k problému dávají i různou časovou náročnost.

- **Inkrementální algoritmy.** Využívá se postupné konstrukce Voronoi diagramů, asymptotická časová složitost ve 2D je $O(n^2)$. Myšlenka byla poprvé představena v [10], poté dále rozšířena, například viz [11, 28, 23]. Tuto skupinu algoritmů použijeme v případě, že předem nevíme, z jakých generátorů budeme Voronoi diagram stavět a nebo generátory postupně přibývají.

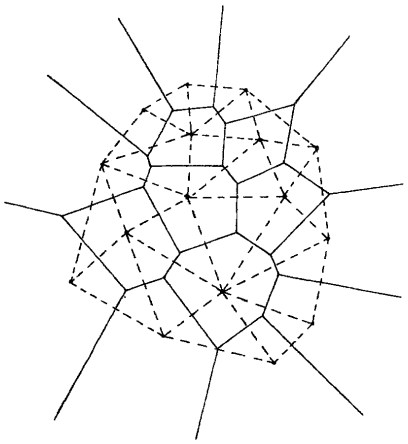
- **Rozděl a panuj.** Využívá se postupného rekurzivního dělení prostoru, sestavení Voronoi diagramu v podprostorech a následného spojování. Výsledná složitost celého algoritmu ve 2D je $O(n \log n)$ [27, 16].

- **Za pomoci konvexní obálky.** V případě konstrukce konvexní obálky v $d+1$ dimenzích, kterou pak můžeme použít ke konstrukci Voronoi diagramu [8]. Výhodou je nízká očekávaná časová náročnost. Ke konstrukci konvexní obálky lze například použít algoritmus QuickHull [5].

Pro aditivně vážené Voronoi diagramy však tyto postupy nefungují, a proto se používá algoritmus trasování hran.

2.3.1 Algoritmus trasování hran

Detailní popis algoritmu trasování hran (*Edge Tracing algorithm*) pro konstrukci váženého Voronoi diagramu v $\mathbb{R}^3$ je v [14]. Zde se omezíme pouze na stručný popis.



Obrázek 4: Delaunayova triangulace (čárkované čáry) a Voronoi diagramy (plné čáry) [18].

Na počátku algoritmus vypočte polohu prvního Voronoi vrcholu v_0. Tato poloha je určena jakožto střed sféry, která je vepsaná sférám čtyř generátorů a její vnitřek neprotíná žádný generátor. Je tedy prázdná. Na základě informací o Voronoi vrcholu v_0 a čtyřech generátorech jsou následně vypočteny Voronoi hrany e_0, e_1, e_2 a e_3, které vycházejí z vrcholu v_0. Objevené Voronoi hrany jsou poté umístěny na zásobník hran (*Edge-stack*). K nově nalezeným hranám se přidá i informace o vrcholu, kde mají počátek. V první iteraci je to tedy vrchol v_0. Následně jsou ze zásobníku odebírány hrany, kde pro každou odebranou hranu je vypočten vrchol (označme ho jako vrchol v_i), kde daná hrana končí. Tento vrchol lze dopočítat za pomoci sféry, která je tečná ke sférám, které definují danou hranu a jedné další sféry, která by mohla být generátorem pro hledaný Voronoi vrchol. Požadavkem, který specifikuje hledanou sféru je, aby tato sféra byla vepsána mezi 4 generátory, z toho 3 jsou sféry, které definují úzké hrdlo, a aby délka podél hrany od počátečního vrcholu ke středu dané sféry byla minimální. V praxi se místo délky křivky používá úhlová vzdálenost. Sféry, které definují Voronoi hranu, resp. úzké hrdlo na Voronoi hraně, se v anglické literatuře označují jako *gate balls.* Poloha vrcholu v_i bude odpovídat středu dopočtené koule.

Se znalostí vrcholu v_i a vstupní hrany můžeme dopočítat další tři hrany, které vystupují z tohoto vrcholu. Těmto novým hranám přiřadíme vrchol v_i jako jejich počáteční. A takto se algoritmus opakuje. Nově nalezené vrcholy jsou ukládány na zásobník a pro hrany vybrané ze zásobníku jsou zase dopočítány další vrcholy. Ve chvíli, kdy již v zásobníku nezbývá žádná další hrana, můžeme algoritmus ukončit, jelikož máme vytvořený celý Voronoi diagram.

Jestliže však ve výpočtech narazíme na již vypočítaný vrchol, je potřeba ho dohledat mezi vypočítanými vrcholy a vložit do zásobníku pouze ty vycházející hrany, které ještě nebyly vloženy. Pokud bychom tento krok vynechali, došlo by k zacyklení algoritmu. Již vypočtené vrcholy můžeme ukládat do slovníku a položky dohledávat v konstantním čase.

Otázkou zůstává způsob vybrání prvního Voronoi vrcholu v_0. Jestliže bychom uvažovali nalezení hrubou silou, musíme projít všechny kombinace čtveřic sfér,

k nim sestavit sféru, která se všech dotýká a následně zkontrolovat, zda-li je prázdná. Kontrola prázdnosti zase spočívá v prohledání všech ostatních sfér a otestování, zda-li se testovaná sféra nenachází uvnitř hledané sféry, tudíž, že sféra je neprázdná. V nejhorším případě má tento postup složitost $O(n^5)$. Takovéto situaci se chceme přirozeně vyhnout a proto se používá následující způsob hledání v_0.

K nalezení počátečního Voronoi vrcholu je více způsobů. Podle [14] nejdříve množinu generátorů rozšíříme o další čtyři sféry. Tyto sféry označíme jako fiktivní sféry. Fiktivní sféry volíme tak, aby měly právě jednu tečnou sféru. Střed této sféry definuje tzv. fiktivní Voronoi vrchol. Vzhledem k tomu, že fiktivní sféry jsou umístěny vedle zadaných sfér, můžeme začít prohledávání od fiktivního vrcholu a postupně se v hledání dostaneme do takového stavu, že najdeme vrchol, který již nebude definován za pomoci žádné fiktivní sféry. Takový vrchol pak budeme považovat za vrchol v_0, tedy jako počáteční vrchol.

Jestliže se zaměříme na způsob nalezení prvního vrcholu popsaný v [21], začneme se zvolením libovolného generátoru i. Dále nalezneme sféru j, jejíž vzdálenost povrchu od vzdálenosti povrchu sféry i je minimální. Předpoklady popsané v [21] zaručují existenci právě jedné takové sféry. Následně nalezneme mezi zbývajícími sférami sféru k, pro kterou platí, že vepsaná sféra ke sférám i, j, k má minimální poloměr. Taková sféra pak nutně náleží rovině, která prochází středy sfér i, j, k. Tyto tři sféry, i, j, k, tvoří *bottleneck* (nejužší místo mezi sférami). Při stanovování tečné sféry ke třem sférám vyjdeme z následujících rovnic [21].

$$(x - x_m)^2 + (y - y_m)^2 + (z - z_m)^2 = (R + R_m)^2,\ m = i, j, k \tag{3}$$

Kde (x_m, y_m, z_m, R_m) popisují zadané sféry a (x, y, z, R) popisuje hledanou tečnou sféru. Poslední sféru, tvořící počáteční Voronoi vrchol označme l. Abychom mezi zbývajícími nalezli sféru l, prohledáme zbývající sféry a stanovíme k nim tečnou sféru. Vybereme takovou sféru l, pro kterou bude platit, že tečná sféra mezi sférami i, j, k, l bude mít minimální poloměr. Minimální poloměr nám zajistí, že nalezneme právě Voronoi vrchol, jelikož hledáme řešení za pomocí sfér i, j, k, které definují hranu Voronoi vrcholu a její bottleneck. Jestliže řešení neexistuje, je možné, že hrana jde z nekonečna do nekonečna a nebo se jedná o elipsu bez vrcholu. Řešení těchto speciálních případů je nad rámec této práce, ale lze dohledat např. v [20]. Sféru l nalezneme vyřešením soustavy podobné rovnicím (3).

$$(x - x_m)^2 + (y - y_m)^2 + (z - z_m)^2 = (R + R_m)^2,\ m = i, j, k, l \tag{4}$$

I zde uvažujeme (x_m, y_m, z_m, R_m) jako zadané sféry a (x, y, z, R) jako hledanou tečnou sféru.

Algoritmus má asymptotickou složitost $O(mn)$, kde m je počet Voronoi hran a n je počet generátorů.

3 Paralelní výpočty na současném hardwaru

Na následujících řádkách nastíníme problematiku paralelních výpočtů na současném grafických kartách i procesorech a představíme jeden ze základních moderních přístupů - datový paralelizmus.

3.1 CPU/APU

Současným trendem je navyšování počtu jader v procesoru, spíše než zvyšováním výkonu jednoho jádra. Dnešní serverové procesory vyšší třídy, rodiny x86 mají až 18 fyzických výpočetních jader[1]. To znamená, že pokud bychom prováděli výpočty pouze v jednom vlákně, využili bychom, teoreticky, pouze jednu osmnáctinu výkonu. Dá se předpokládat, že trend navyšování jader bude i nadále pokračovat a frekvence jader, resp. jejich výpočetní výkon se bude zvyšovat pouze minoritně. Je zde tedy velký potenciál pro algoritmy, které dokáží tento paralelizmus využít.

Nicméně vícero jader není jediný paralelizmus, který současné procesory nabízí. V průběhu časů byly instrukční sady procesorů rozšířeny o tzv. SIMD (Single instruction, multiple data) instrukce. SIMD instrukce dovolují provádět operace nad vektory jako jedinou instrukci. Samozřejmě za předpokladu, že vektory se vejdou do vyhrazených registrů, které jsou v nových procesorech například 256 bitové. Dále se ušetří režije na kopírování dat z paměti do registrů, jelikož procesory podporují i načítání celých vektorů z hlavní paměti do registrů.

Pokud tedy například chceme vynásobit mezi sebou dvě skupiny čísel o velikosti 32 bitů, můžeme násobit až 8 čísel z každé skupiny současně, v jediné instrukci. Tato funkcionalita se využívá jako optimalizace například při práci s poli. Takové optimalizaci, nejčastěji prováděnou přímo překladačem, se říká vektorizace.

3.2 Grafické karty

Grafické karty kromě vektorových instrukcí nabízejí ještě větší paralelizmus, co se týká počtu výpočetních jader. Současné grafické karty jsou schopné spouštět až několik tisíc operací zároveň. To přináší velice vysoký teoretický výpočetní výkon, ale klade dodatečné nároky na zvolené algoritmy. Například karta *AMD/ATI RADEON R9 295X2* obsahuje 5632 streamových procesorů. Celkový teoretický výkon karty činí úctyhodných 11.5 TFLOPS (trilionu operací za sekundu v pohyblivé desetinné čárce čísel s jednoduchou přesností).

Grafické karty tedy mají vysoký výpočetní výkon, ale ne vždy se může vyplatit výpočty delegovat na grafickou kartu. Vzhledem k latenci, která je mezi hlavní pamětí a výpočetní jednotkou v grafickém čipu může být, pro menší objemy dat, výhodnější výpočty provést na CPU.

[1] https://ark.intel.com/products/85766/Intel-Xeon-Processor-E5-4669-v3

3.3 Ostatní výpočetní koprocesory

Mezi ostatní výpočetní výpočetní koprocesory můžeme zařadit FPGA (Field-programmable gate array), což jsou specializované výpočetní jednotky, které mají programovatelnou vnitřní strukturu a tedy i vysoký stupeň optimalizace pro konkrétní účel. Některé FPGA podporují OpenCL a lze je tedy využít jako obecné koprocesory. Dalším koprocesorem, který je přímo specializovaný na paralelní výpočty je například Intel Xeon Phi.

3.4 Programové nástroje

Aby softwarový vývojáři byly odstíněny od konkrétní implementace hardwaru, byly vyvinuty aplikační programovací rozhraní (API) pro programování v paralelním prostředí. Některé API se zabývají přímo grafickými výpočty, jako například OpenGL. Některé API se zase specializují na výpočty, například OpenCL. OpenCL je podporované napříč grafickými kartami, některými výpočetními koprocesory, ale i samotnými procesory. OpenCL hlavně nabízí jednoduchý způsob, jak využívat efektivní SIMD instrukce a další vektorové operace. Experimentální implementace algoritmů v bakalářské práci budou právě za pomoci OpenCL API.

Při návrhu paralelních algoritmů je potřeba dbát na sdílení a přístupu k paměti a dále pak na konzistenci, která závisí na pořadí prováděných operací. Obecně je potřeba synchronizaci minimalizovat, jelikož přináší velkou režii. Také se využívá rozdělení algoritmu do několika fází, které běží odděleně a synchronizace je prováděna jen mezi fázemi, viz např. MapReduce.

4 Jazyk OpenCL

OpenCL je programovací jazyk vycházející ze standartu C99, ze kterého přebírá základní syntaxi. Ta je dále doplněna zejména o vektorové datové typy a operace s nimi, rozšířené datové prostory a naopak omezena některými restrikcemi, vycházejícími z hardwarového návrhu cílové platformy. OpenCL definuje jednoduchý způsob, jakým spustit paralelní kód nad paralelní architekturou za využití nativních lehkých vláken. Vstupním bodem programu, který takto spouštíme jsou funkce označované *kernely*.

Termínem *host* nebo *hostitel* se v OpenCL terminologii označuje program, který má na starosti spouštění OpenCL kódu na příslušném zařízení. Cílovým zařízením může být například CPU, GPU nebo APU (Accelerated Processing Unit - kombinace GPU a CPU na jednom čipu), což je právě výhodou OpenCL. Dovoluje spouštět kód napříč různými zařízeními.

V současné době je poslední specifikací verze 2.1. Nicméně poslední podporovanou verzí napříč všemi významnými výrobci grafických karet je verze 1.2. S tímto ohledem je program práce napsaný pro verzi 1.2.

4.1 Výpočetní model

OpenCL kód se volá pouze skrze kernely. Kernel představuje paralelně spustitelný kód. Kolikrát se tento kód má spustit určujeme velikostí tzv. *globální a lokální pracovní s*kupiny. Globální pracovní skupina určuje celkový počet spuštění kernelu programu. Lokální pracovní skupina určuje kolikrát se má kernel spustit současně. Velikosti obou skupin jsou omezeny hardwarovým návrhem daného zařízení. Výhodou lokání pracovní skupiny je, že zařízení může spustit všechny prvky z této skupiny současně. V rámci lokální skupiny můžeme využívat rychlejší synchronizace a rychlou sdílenou paměť. Uvnitř kernelu pak můžeme zjistit index v globálním i lokální pracovní skupině. To nám pomůže určit data, která má zpracovat konkrétní kernel. Jedná se o přímou aplikaci teorie datového paralelismu.

Další funkcionalitou, kterou nám pracovní skupiny nabízejí je specifikovat více dimenzí těchto skupin. Můžeme tedy například obrázek zpracovávat pixel po pixelu při zachování adresování pomocí 2D souřadnic.

Data kernelu předáváme jako ukazatel do *globální oblasti paměti*. Tuto paměť inicializujeme na straně hostitelského programu, který spouští OpenCL kód. Obsah paměti je následně překopírován na GPU a nebo ji dokonce můžeme ponechat v hlavní paměti počítače, ale musíme počítat se zvýšenou latencí při každém přístupu. Pomocí příslušných funkcí můžeme také využít namapování adresního prostoru výpočetního zařízení do virtuální paměti programu a tím vytvořit sdílenou paměť.

4.2 Hierarchie paměti

Vzhledem k cílení OpenCL na aplikace, kde je výkon klíčový, nechává na programátorovi rozdělení dat do oblastí, které odpovídají paměťovým oblastem grafických karet. Díky správnému použití jednotlivých oblastí můžeme dosáhnout vysoké propustnosti našich programů. V programech můžeme využívat následující oblasti.

- **Globální paměť** - jedná se o oblast, do které mohou přistupovat všechny kernely. Pro synchronizaci paralelního přístupu do této paměti slouží množina atomických funkcí, které zajišťují konzistenci operací mezi kernely. Jedná se například o atomické inkrementy, logické funkce a komparátory.
- **Lokální paměť** - tato oblast je sdílená napříč všemi kernely v pracovní skupině. Přístup do této paměti je rychlejší než do globální díky kolokaci s výpočetní jednotkou. Nad lokální pamětí je umožněna synchronizace pomocí atomických operací, stejně jako nad globální pamětí, avšak s benefitem lokality paměti a z toho vyplývajícím rychlostním benefitem.
- **Konstantní paměť** - oblast paměti je dle specifikace alokovaná v globální paměti, ale je označena jako konstantní, tedy inicializace je povolená pouze v době překladu. Veškerá data sdílená mezi kernely v době překladu musí být specifikována v této oblasti. Jelikož se jedná o data přístupná pouze pro čtení, zařízení je může cachovat v různých úrovních. Využívanou optimalizací také je nakopírování využívaných dat přímo do registrů výpočetní jednotky.
- **Privátní paměť** - vyhrazený prostor pouze pro kernel. Ukládají se sem proměnné uvnitř funkcí. Výhodou je rychlý přístup do této paměti, avšak velikost je závislá na použitém hardwaru.

Vidíme, že nám OpenCL dává více možností, jak přesně můžeme data uchovávat. Naší snahou je dostat se používanými daty co nejblíže k výpočetní jednotce. Měli bychom tedy, pokud to okolnosti dovolují, nejdříve využívat privátní, konstantní a lokální paměť. Nicméně ani globální paměti se nelze vyhnout, ale minimalizace přístupů do tohoto segmentu je výrazná optimalizace.

4.3 Některá omezení jazyka

OpenCL nepodporuje dynamické alokace paměti. Při běhu kernelu již není jakkoli možné alokovat další paměť. Paměť musí být alokována buď v době překladu a nebo alokována na hostu a kernelu předána ukazatelem. V důsledku toho jsou zakázané i pole variabilní délky. Toto omezení je poměrně přísné. Za předpokladu, že si nechceme psát vlastní správu *haldy*, vylučuje velkou množinu algoritmů a běžně používaných datových struktur. Například konkurenční CUDA tímto omezením netrpí.

Další běžně používanou konstrukcí je rekurze. Avšak i ta je v OpenCL zakázaná. Kvůli optimalizacím OpenCL, při překladu *rozepíše* volání funkcí (tzv. *inlining*). Program se tedy vykonává sekvenčně, bez volání funkcí. S rekurzí by tato optimalizace nebyla možná.

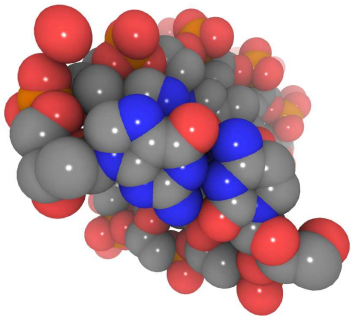
Přetypování ukazatelů mezi jednotlivými adresními prostory není povolené. Povolená nejsou ani vícerozměrná pole.

5 Výpočet molekulárního povrchu

Následující kapitola čtenáře seznámí s geometrickou strukturou molekuly tak, jak ji budeme využívat v našem modelu. Tento model rozebereme na jednotlivá geometrická primitiva, která bude následujícími algoritmy identifikovat.

5.1 Van der Waals model

Van der Waals model nám přináší způsob, jak si představit strukturu uvnitř molekuly či na jejím povrchu. Tento model znázorňuje každý atom jako sféru o určitém poloměru. Jednotlivé prvky (uhlík, vodík, ...) mají tabulkami stanovený tzv. Van der Waals poloměr. S informací o typů jednotlivých atomů a jejich polohách jsme tedy schopni sestavit model, který dále budeme pro analýzu využívat. Na obrázku 5 je takový model vizualizovaný na příkladu jednoduché molekuly. Typy atomů jsou vyznačené rozdílnými barvami.



Obrázek 5: Vizualizace Van der Waals modelu molekuly v programu CAVER Analyst [17].

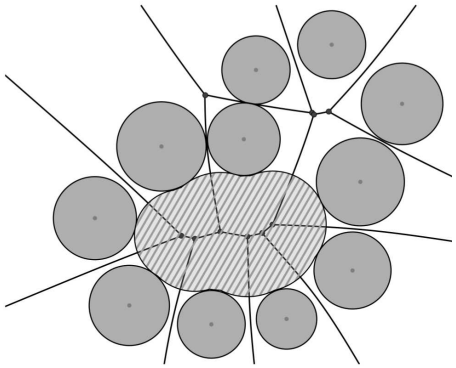
Volný prostor mezi atomy zkoumáme pomocí sondy. Jako sondu používáme abstrakci atomu o určitém poloměru. V praxi tento poloměr vychází z rozměrů určitých prvků, které jsou naším zájmem nebo dokonce celou další molekulou, která interaguje se zkoumanou molekulou. Zajímá nás prostor, kde se může, bez kolize s atomy molekuly, sonda pohybovat.

Zkoumáme jednak prostor vně i uvnitř molekuly (dutinky). Povrch, který je sondou vytvořen po označujeme Connolly povrch, kterému se někdy říká jednoduše molekulární povrch. Connolly povrch molekuly je tedy hranice sjednocení všech nekolizních sond pevně zvoleného poloměru. Se změnou poloměru sondy se samozřejmě mění i povrch molekuly a dutinky uvnitř. Názorně je dutinka šrafováním vyznačena na obrázku 6.

Pomocí průchodu Voronoi diagramu grafovým prohledáváním lze dutinky, kam se vejde sonda, detekovat a vypočítat polohy jednotlivých geometrických primitiv, které je tvoří. Stejný přístup aplikujeme i v případě, že hledáme čistě na povrchu molekuly.

5.2 Geometrický popis molekulárního povrchu

Molekulární povrch vizualizujeme spojením několika prostorových geometrických primitiv. Konkrétně se jedná o části kulové plochy - sférické pláty, tedy části jednotlivých atomů v molekule. Dále povrch popisujeme za pomoci anuloidů (torů), toroidální plátů a sférických trojúhelníků. Ty určují mezní prostor, kam se sonda dostane, při kontaktu s více atomy [9].



Obrázek 6: Znázornění dutinky uvnitř několika atomů [12].

5.2.1 Sférické pláty

Sférické pláty jsou součástí popisu povrchu právě tehdy když se sonda může pohybovat těsně po povrchu některého atomu, aniž by byla omezena nějakým jiným atomem. Na obrázku 8 jsou to tři kulové plochy, které znázorňují části povrchů atomů.

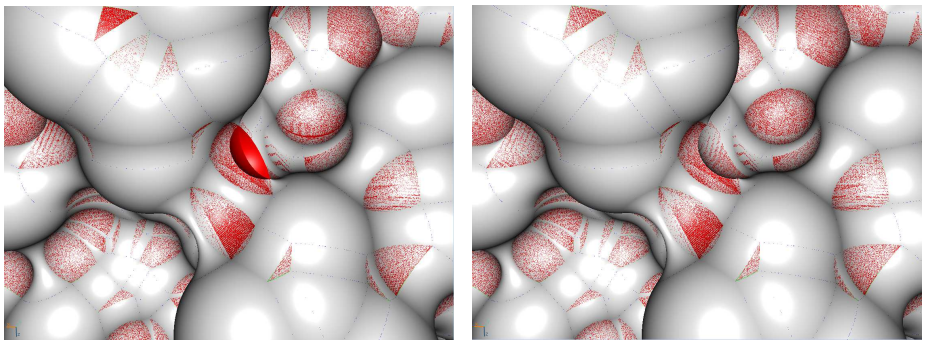
5.2.2 Sférické trojúhelníky

Sférické trojúhelníky vznikají v místech, kde se sonda „zastaví“ na překážce tvořenou třemi sférami. Jinak řečeno bottleneck mezi těmito atomy bude menší, než je poloměr sondy. Jelikož používáme sférickou sondu, její střed bude ekvidistantní vůči povrchům jednotlivých atomů a tedy bude ležet na hraně váženého Voronoi diagramu. Analyticky polohu sférického trojúhelníka zjistíme za pomocí rovnic (3).

Sférické trojúhelníky se mohou protínat a pro potřeby vizualizace povrchu je nutné oříznout každý sférický trojúhelník sférami ostatních trojúhelníků.

Obrázek 7 ukazuje případ, kdy je potřeba sférické trojúhelníky oříznout. Přibližně uprostřed se nachází sférické trojúhelníky, které se protínají. Tmavější červenou je znázorněn trojúhelník, který má střed dál od pozice kamery a světleji vykreslený trojúhelník, který ho protíná. Na obrázku vpravo už vidíme oba trojúhelníky oříznuté.

Na obrázku 8 si zeleného sférického trojúhelníku můžeme povšimnout ve středu mezi třemi znázorněnými atomy.



Obrázek 7: Porovnání neoříznutých (vlevo) a oříznutých (vpravo) sférických trojúhelníků. Vizualizováno v programu CAVER Analyst.

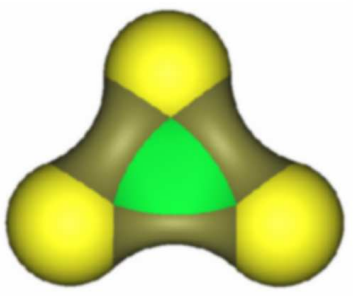
5.2.3 Anuloidy

Anuloidy neboli tory, resp. jejich vnitřní části, vznikají tehdy, když sonda nemůže proniknout mezi dvěma atomy, ale může prostor mezi atomy opsat při nepřetržitém kontaktu s nimi.

5.2.4 Toroidální pláty

Toroidální pláty vznikají za stejné podmínky jako anuloidy, avšak sonda neopíše celou kružnici po obvodu atomů ale pouze nějakou její část. V opsání celé části, aby povrch tvořil anuloid je sondě zabráněno nějakým jiným atomem. Toroidální pláty sdílejí krajní hranu s incidentními sférickými trojúhelníky.

Obrázek 8 je naznačuje mezi dvojicemi atomů které zároveň plynule navazují na sférický trojúhelník ve středu.



Obrázek 8: Sférické pláty (žlutě), sférický trojúhelník (zelená), toroidální pláty (hnědá) na příkladu molekuly se třemi atomy. Zdroj obrázku [25].

6 Výpočty v biomolekulách vhodné pro paralelizaci

Po shrnutí teoretického základu se od šesté kapitolu věnujeme již vlastnímu zkoumání výpočtů z pohledu paralelních výpočtů. V této kapitole identifikujeme výpočty, které lze přenést na grafickou kartu a dále stanovíme ty, které opravdu přeneseme. Kapitola popisuje i současný přístup, ze kterého budeme vycházet.

6.1 Sestavení Voronoi diagramu a Delaunayovi triangulace

Voronoi diagram je základním kamenem pro následnou analýzu biomolekul. Sestavení diagramu lze provést za použití paralelního zpracování, avšak to není cílem této práce. Například v publikace [19] se zabývá paralelním trasováním hran s implementací v OpenMP.

6.2 Nalezení nejužších míst mezi atomy v molekule

Pro jednotlivé Voronoi hrany je potřeba předpočítat velikost nejužšího místa. Nejužší místo v prostoru si můžeme představit jako největší možnou sféru, jejíž střed by se mohl nacházet ve všech místech Voronoi hrany bez kolize s okolními atomy. Tomuto úzkému hrdlu se v anglické literatuře říká bottleneck. Výpočet a uložení této informace nám pomůže v následujících fázích výpočtů - v průchodnosti sondy po jednotlivých hranách Voronoi diagramu.

Jelikož výpočet nejužších míst není v současné implementaci na CPU časově náročnou operací, v práci toto hledání není implementováno.

6.3 Algoritmus pro hledání geometrických primitiv molekulárního povrchu

6.3.1 Současný algoritmus implementovaný na CPU

Vyjděme z algoritmu popsaného ve [13]. Tento algoritmus očekává jako vstup aditivně vážený Voronoi diagram pro atomy zadané molekuly. U každého Voronoi vrcholu si uchováme jeho poloměr, což je minimální vzdálenost vrcholu od sfér všech čtyřech atomů, které ho definují. Následně jsou seřazeny Voronoi vrcholy od největšího k nejmenšímu podle poloměru. Dále algoritmus vypočítá bottlenecky na jednotlivých Voronoi hranách. Tato fáze se vykoná pouze jednou pro zadanou molekulu.

Poté zvolíme poloměr sondy a začneme prohledávat graf Voronoi vrcholů, s tím, že se pohybujeme po jednotlivých Voronoi hranách. Vyhledáváme pouze ve Voronoi vrcholech, které mají alespoň tak velký poloměr, jako má sonda. Do sousedního Voronoi vrcholu můžeme vstoupit pouze v případě, že bottleneck na dané Voronoi hraně je vetší nebo roven poloměru sondy. Grafové prohledávání začneme od začátku seřazených Voronoi vrcholů, tedy od toho, kam se vejde největší sféra. Jestliže se zastavíme při grafovém prohledávání, tedy není

kam rozšířit již nalezenou dutinku, pokračujeme od Voronoi vrcholu s největším poloměrem, který jsme ještě nenavštívili. Jestliže si Voronoi vrcholy propojené Voronoi hranamy představíme jako graf, pak každý podgraf, který jsme prohledáváním identifikovali, je dutinkou. Speciálním případem je takzvaná vnější dutinka, kdy je dutinka propojená s prostorem mimo molekulu. Prohledávání končí, jestliže ze seznamu vybereme vrchol, kam již nelze sonda bezkolizně se sousedními atomy umístit.

Algoritmus tedy najde všechny geometrická primitiva, ze kterých se skládá molekulární povrch (i dutinky uvnitř molekuly). Například sféry definující sférické trojúhelníky se nachází v místě, kde se o bottleneck zastavilo grafové prohledávání. Sférické pláty jsou tvořeny atomy, které tvoří navštívené Voronoi vrcholy.

6.3.2 Detekce kolizí geometrických primitiv

Při skládání povrchu z jednotlivých geometrických primitiv, může nastat situace, kdy některé sférické trojúhelníky kolidují. Je tedy potřeba stanovit ořezávací roviny pro každou takovou kolizi. Porovnání kolizí každého sférického trojúhelníku navzájem způsobí kvadratickou časovou složitost vzhledem k počtu sférických trojúhelníků. Současný algoritmus proto využívá prostorové hashování [22]. Postup spočívá v tom, že prostor, ve kterém se molekula nachází je rozdělen na větší množství krychliček. Hashovací funkce nám určí, do které krychličky bude patřit zadaná sféra definující sférický trojúhelník. Nejdůležitějším aspektem toho hashování je co možná nejrovnoměrnější rozdělení napříč všemi krychličkami. Kolize ve smyslu promítnutí dvou různých prostorů do jedné krychličky je povolená.

Následně vybereme pouze podprostor z těchto krychliček, kde mohou nastat kolize. Poté zkontrolujeme kolize pouze v tomto omezeném prostoru.

Implementace toho algoritmu v OpenCL bude popsána v následující kapitole.

7 Paralelní implementace v OpenCL

V této kapitole popíšeme paralelní implementaci v OpenCL. Jazyk OpenCL byl zvolen zadavatelem, jakožto rozšířený a platformně nezávislý programovací jazyk umožňující využít paralelismu, který nabízejí moderní grafické karty.

Na straně hostitele se o načítání datových struktur a spouštění OpenCL kódu stará program napsaný v Javě. Java byla vybraná z důvodu kompatibility s ostatními dostupnými programy a knihovnami.

7.1 Hledání sférických trojúhelníků

Nejprve je za pomoci knihovny awVoronoi[1] sestavena množina tetrahedronů popisující Voronoi diagram a spočteny bottlenecky. Následně jsou tyto struktury transformovány na strukturu sdílenou s OpenCL. Tato struktura obsahuje pozici a poloměr Voronoi vrcholu. Dále pak pozice, poloměry a identifikátory atomů definujících tento Voronoi vrchol.

7.1.1 Vstup algoritmu

Hostitelský program v Javě komunikuje s OpenCL částí za pomoci předávání bufferů s daty, nad kterými se má spustit kernel. Skutečnost, že hostitelská část, jež alokuje struktury, je napsána v Javě, nám dovoluje vynechat nulování alokovaných bufferů. To provede JVM automaticky za nás i přes to, že se jedná o off-heap paměť. Kvůli značné režii při spouštění kernelu a kopírování obsahu bufferů do paměti grafické karty neslouží kernel na hledání sférických trojúhelníků pouze k jednomu účelu. Kernel na výpočet sférických trojúhelníků určuje také, jaké atomy se mají vykreslit resp. ty atomy, které tvoří viditelné sférické pláty. Algoritmus hledání sférických trojúhelníků tyto atomy snadno v průběhu detekuje a proto nemá smysl prohledávání spouštět vícekrát. Dále tento kernel stanovuje tory, které se mají vykreslit. Vstupem pro tento kernel tedy nejsou pouze struktury potřebné pro výpočet sférických trojúhelníků, ale i ostatní struktury, jejichž význam bude vysvětlen dále. Pro úplnost uveďme celý seznam vstupů kernelu.

- Počet Voronoi vrcholů
- Pole struktur Voronoi vrcholů a jejich definujících atomů
- Poloměr sondy
- Alokované číslo pro výstup - počet sférických trojúhelníků, číslo sem bude zapsané z kódu kernelu
- Alokovaná struktura pro výstup - struktura se sférickými trojúhelníky
- Alokovaná a inicializovaná mapa atomů určující zda-li se má atom vykreslit. Binární stav je reprezentován hodnotami 0 a 1
- Alokovaná struktura pro výstup toroidálních plátů

7.1.2 Výpočet

Vstupní pole Voronoi vrcholů je rovnoměrně rozděleno mezi jednotlivé výpočetní vlákna. Rozdílně, oproti původní implementaci, vstupní pole se neřadí. Toto řazení by se buď muselo udělat na straně hosta - kde by se nevyužil paralelizmus, který máme k dispozici, a nebo na straně OpenCL, takže by se na zařízení musely stejně kopírovat všechna data. Filtrování (viz dále) je nicméně hodně levná operace. Paralelní výběr prvků z pole je otázka jednotek milisekund pro pole o velikosti statisíců položek. Každé vlákno iteruje skrze svoji část tohoto pole a hledá Voronoi vrchol s poloměrem větším, než je poloměr sondy. Tím identifikujeme místa, kam se vejde sonda, což jsou místa, kde se může vyskytovat i sférický trojúhelník. Následně iterujeme všechny čtyři bottlenecky daného Voronoi vrcholu a testujeme, zda-li je sonda větší než bottleneck. Pokud ano, sonda se při průchodu touto Voronoi hranou zastaví o tři atomy definující bottleneck. Vznikne tím hledaný sférický trojúhelník.

Následuje určení středu sféry definující sférický trojúhelník. Ta je nalezena jako řešení rovnice (4), popsané ve druhé kapitole. Rovnice nám může dát dvě řešení - existují dvě tečné sféry, každá na opačné straně bottlenecku. Jestliže proložíme středy atomů definujících bottleneck rovinou, můžeme určit, na které straně roviny se má právě hledaná sféra sférického trojúhelníka nacházet. Bude se nacházet na stejné straně, jako je Voronoi vrchol, ze kterého jsme počítali sférický trojúhelník. Časová složitost výpočtu je $O(n)$ vzhledem k počtu Voronoi vrcholů na vstupu.

7.1.3 Výstup

Kromě středu sféry definující sférický trojúhelník do výstupní struktury sférických trojúhelníků uložíme také normalizované vektory na atomy definující bottleneck a jejich identifikátory. Vypočtené vektory určují vrcholy trojúhelníka. Každý ze tří vrcholů se spočítá tak, že ke středu sféry sférického trojúhelníka se přičte příslušný vektor vynásobený poloměrem sondy. Poloměr ukládat nemusíme, ten je definovaný poloměrem sondy.

7.2 Hledání sférických plátů

V předchozích odstavcích je naznačeno, že sférické pláty jsou hledány podobným algoritmem jako hledání sférických trojúhelníků. Při onom hledání totiž sférické pláty, resp. viditelné atomy, identifikujeme jednoduše jakožto atomy, které definují Voronoi vrcholy s poloměrem alespoň tak velkým jako je poloměr sondy.

Pro zaznamenání skutečnosti, že atom chceme vykreslovat, uděláme jednoduché nastavení hodnoty příslušného indexu v poli sfér atomů. Nutno podotknout, že implementačně se nejedná o pole booleovských hodnot, jelikož v OpenCL není velikost booleanu definovaná, ale je závislá na použité platformě.

7.3 Hledání torů

Při iteraci pole struktur Voronoi vrcholů na přítomnost toru testujeme každou dvojici definující tento vrchol. Zkontrolujeme, zda-li vzdálenost jejich středů, snížená o součet jejich poloměrů, je menší, než průměr sondy. Jestliže zjistíme, že tato vzdálenost je menší, víme, že se na daném místě torus nachází.

Před samotným výpočtem vzdálenosti je implementována drobná optimalizace, která zabraňuje dvojnásobnému výpočtu tím, že porovná hodnoty indexů atomů. Porovnává pouze takovou uspořádanou dvojici, kde index prvního prvku je větší než index druhého.

7.4 Ořezávací roviny sférických trojúhelníků

Zbývá popsat kernel pro stanovení ořezávacích rovin sférických trojúhelníků. V logice programu se vyskytuje paralelní prefix sum, který je zde vysvětlen. Dále se už zaměříme na vlastní algoritmus.

7.4.1 Paralelní prefix sum

V následujících výpočtech je použit algoritmus paralelního výpočtu tzv. prefixového součtu [6]. Tento součet použijeme v případě, že máme pole čísel, a chceme ke každému indexu pole přiřadit sumu všech předcházejících hodnot. Obyčejný sekvenční algoritmus by iteroval polem, sčítal jednotlivá čísla a mezivýsledky ukládal. Na grafické kartě by ale tento postup byl značně neefektivní a proto se k tomuto účelu používá paralelní varianta algoritmu.

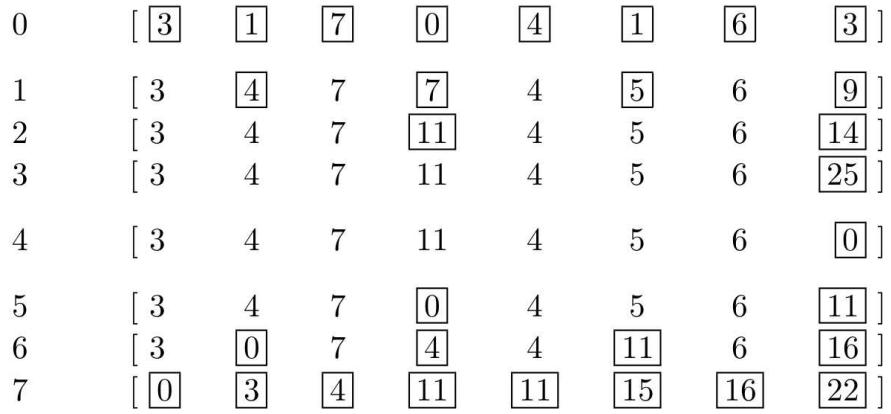
Nechť vstupní pole obsahuje čísla $x_0, x_1, \ldots, x_i$ (na obrázku 9 řádek označený 0). V prvním kroku vypočítáme součty z_i vždy po dvojicích ze vstupů x_i následujícím způsobem: $z_0 = x_0 + x_1$, $z_1 = x_2 + x_3$, ... Tento krok je na obrázku znázorněn řádkem označeným 1. Dále postupujeme rekurzivně např. prvky $w_0, \ldots, w_i$, kde $w_0 = z_0 + z_1$, ... Postup opakujeme dokud můžeme rekurzivně po dvojicích sčítat. V této části stavíme strukturu připomínající obrácený binární strom. Čísla indexů označují vždy $n-tý$ prvek na řádku, a nemají svojí přímou příslušnost k pořadí v původním vstupním poli.

V následujícím kroku už nám zbývá pouze sečíst částečné součty do finální sekvence $y_0, y_1, ..., y_i$. Tu vyjádříme jako $y_0 = x_0, y_1 = z_0, y_2 = z_0 + x_2, y_3 = w_0$ a tak dále. Viz řádky 5, 6, 7 v obrázku 9.

Časová složitost algoritmu pak je $O(\log n)$. V implementaci byl paralelní prefix sum použit jako knihovna [2].

7.4.2 Vstup algoritmu

Vzhledem ke skutečnosti, že výpočet ořezávacích rovin sférických trojúhelníků, kde je pro další výpočty potřeba alokovat paměť (jak již bylo zmíněno, paměť lze alokovat pouze explicitním příkazem z hostitelského programu, nikoliv z kernelu), je potřeba tuto část výpočtu provést v dalším kernelu. Během výpočtu se bude používat prefix sum pro výpočet indexů sférických trojúhelníků v poli,



Obrázek 9: Ilustrace postupu paralelní prefix sumy. Čísla v prvním sloupci značí jednotlivé kroky algoritmu. První (nultý) řádek označuje vstup. V každém kroku sčítáme vždy čísla v rámečku z předchozího řádku. Zdroj obrázku [6].

které je mapované na mřížku pro prostorové hashování. Je proto potřeba alokovat nutné struktury pro výpočet této sumy. Definujme tedy vstup toho kernelu.

- Celkový počet sférických trojúhelníků
- Alokované číslo v globálním paměťovém prostoru pro zaznamenání celkového počtu ořezávacích rovin
- Pole struktur sférických trojúhelníků vytvořené v předchozím kernelu
- Poloměr sondy
- Alokované místo v globálním paměťovém prostoru pro spočítaný výsledek prefixové sumy
- Alokované místo v globálním paměťovém prostoru pro uchování informace, kolik sférických trojúhelníků patří do jedné buňky mřížky
- Alokované místo v globálním paměťovém prostoru pro pole definující příslušnost sférického trojúhelníku k buňkám mřížky
- Alokované místo v lokálním paměťovém prostoru pro dočasné výpočty prefixové sumy
- Alokované místo v globálním paměťovém prostoru pro výsledné ořezávací roviny

7.4.3 Výpočet

Algoritmus lze rozdělit do čtyř fází. V první fázi se spočítá, kolik sfér definujících sférické trojúhelníky náleží jedné buňce v mřížce prostorového hashování [22]. Každé výpočetní vlákno má stanovený svůj úsek sférických trojúhelníků, kterým určí index v mřížce. Funkce výpočtu indexu do mřížky byla inspirována knihovnou Boost [26].

Ve druhé části výpočtu použijeme prefix sum nad počty sférických trojúhelníků v jednotlivých buňkách mřížky. Tato suma nám pak určuje index do pole sférických trojúhelníků, kam ve třetí fázi sférické trojúhelníky opravdu zapíšeme. Resp. zapíšeme pouze náležité indexy do struktury sférických trojúhelníků.

Poslední fáze výpočtu iteruje znovu přes sférické trojúhelníky a vybírá z mřížky všechny buňky, které sousedí s buňkou, kam náleží aktuální sférický trojúhelník. Následně kontroluje kolize jednotlivých sférických trojúhelníků ležících v možném kolizním prostoru. Vybíráme všechny možné sousedící buňky v prostoru, tedy i ty, co sdílí s aktuální buňkou právě jeden bod. Celkem vybíráme $3 \times 3 \times 3$, tedy 27 buněk. Tím znatelně oříznemе množinu sférických trojúhelníků, pro kterou musíme navzájem hledat kolize. Řešení hrubou silou nepřipadá v úvahu pro molekuly s několika sta-tisíci sférických trojúhelníků.

Při samotné kontrole kolizí vypočteme vzdálenost mezi středy sfér definujících možné kolizní sférické trojúhelníky. Jestliže je polovina této vzdálenosti menší než poloměr sondy, víme, že dané dva sférické trojúhelníky kolidují. Musíme proto sestavit rovnici roviny, kde se má sféra oříznout. První tři parametry rovnice roviny určíme jako rozdíl polohy trojúhelníků. Pro druhý trojúhelník stačí otočit operandy odčítání. Poslední parametr rovnice této roviny stanovíme pomocí následující rovnice.

$$d = \frac{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}{2r} \tag{5}$$

Kde x_1, y_1, z_1 označuje polohu středu sféry definující první sférický trojúhelník. Podobně i x_2, y_2, z_2 značí polohu středu druhé sféry. Poloměr sondy je značen r.

Výsledné parametry rovnice ořezávací roviny pro první sférický trojúhelník jsou tedy $(x_1 - x_2, y_1 - y_2, z_1 - z_2, d)$. Obdobně pro druhý sférický trojúhelník $(x_2 - x_1, y_2 - y_1, z_2 - z_1, d)$.

Tento výsledek pro oba sférické trojúhelníky zapíšeme do výstupní datové struktury na následující volný index.

Samotné ořezávání je záležitost zobrazovací části v CAVER Analystu a provádí se až ve fragment shaderu na základě těchto parametrů roviny.

7.5 Spuštění programu

7.5.1 Minimální a doporučené požadavky

Program ke svému chodu potřebuje v první řadě běhové prostředí Javy. Minimální doporučená verze je 1.8.u40. Dále je vyžadováno prostředí pro spouštění OpenCL aplikací alespoň ve verzi kompatibilní se specifikací OpenCL verze 1.2. To lze, v případě grafické karty značky Nvidie nainstalovat přímo s ovladači[2]. Je možné nainstalovat i ovladače pro spouštění kódu na CPU. Pro procesory Intel dostupné například na následující adrese[3].

Pro zajištění chodu programu je potřeba minimálně 200 MB volné operační paměti a 130 MB volné paměti v grafické kartě. Pro výpočty nad velkými molekulami (velikost v řádu statisíců atomů) je potřeba 2,5 GB volné operační paměti a 600 MB volné paměti v grafické kartě.

7.5.2 Parametry programu

Program spustíme příkazem

```
java -jar ConnollySurfaceGPUComputation.jar
```

Jestliže vynecháme některé povinné příznaky, zobrazí se nám nápověda v následujícím výstupu.

```
This is program for computing Connolly Surface via OpenCL on GPU
No molecule path specified. Please specify -m argument
   -g N : GPU platfom id (default: 0)
   -m VAL : Specify path to molecule XML definition
   -o VAL : Output file (default: connolySurface.txt)
   -r N : Probe radius (default: 1.0)
   -t : Performance test. Multiple runs, multiple probe sizes
   -v N : Verbose - print GPU info (default: 1)
Invalid arguments. Exiting.
```

Vidíme přehledně vypsané parametry, které program přijímá. Uveďme zde jejich seznam s krátkým popisem.

- `-g` očekává číslo, kterým uživatel vybere OpenCL platformu, na které chce výpočet spustit. Výchozí hodnota je 0. Použije se tedy první nalezená OpenCL platforma. Jestliže má uživatel na počítači instalováno více platforem (například běhové prostředí pro procesor i pro grafickou kartu), může tímto parametrem vybrat některou z dalších platforem.

- `-m` je povinný parametr, kterým uživatel zadá absolutní cestu k souboru, kde je definována molekula. Přijímány jsou formáty `.xml` a `.xml.gz`. Vstup je následně přečten knihovnou awVoronoi. Formát, který přijímá tato

[2] https://www.nvidia.com/Download/index.aspx?lang=en-us
[3] https://software.intel.com/en-us/articles/opencl-drivers

knihovna je tedy závazný. Molekuly v tomto formátu lze stáhnout například ze stránek Proteinové databanky[4], tam jsou jednotlivé molekuly označené unikátním identifikátorem - např. 1CRN.

- `-o` tento parametr specifikuje cestu k výstupnímu souboru s vypočtenými daty. Výchozí hodnota ukazuje na soubor `connolySurface.txt` v aktuálním adresáři. Parametr je proto nepovinný, ale je doporučené ho nastavit.
- `-r` očekává desetinné číslo, kterým uživatel reprezentuje poloměr sondy, jež bude použita pro průzkum molekulárního povrchu. Tento rozměr se zadává v Ångströmech ($10^{-10}\,m$). Jestliže uživatel nezadá žádnou hodnotu, použije se výchozí hodnota 1.0 Å. Jednotky se nezadávají.
- `-t` tento přepínač zapíná testování rychlosti nad molekulou opakovanou iterací skrze vícero poloměrů sondy.
- `-v` nastavíme na 0, jestliže nechceme na standardní výstup vypisovat informace o použité OpenCL platformě. Výchozí hodnotou je 1, tedy zobrazování podrobných informací.

Kompletní příklad spuštění programu v Unixovém systému s povinnými i doporučenými příznaky je následující.

```
java -jar ConnollySurfaceGPUComputation.jar \
   -m /home/u/molecule/1crn.xml \
   -o /tmp/1crn_out.txt \
   -r 0.2
```

7.5.3 Závislosti

Implementace programu závisí nejen na knihovně awVoronoi, jak již bylo zmíněno, ale využívá také mapování na OpenCL z knihovny JOCL[3]. Dále je využita knihovna Args4j pro parsování parametrů z příkazové řádky.

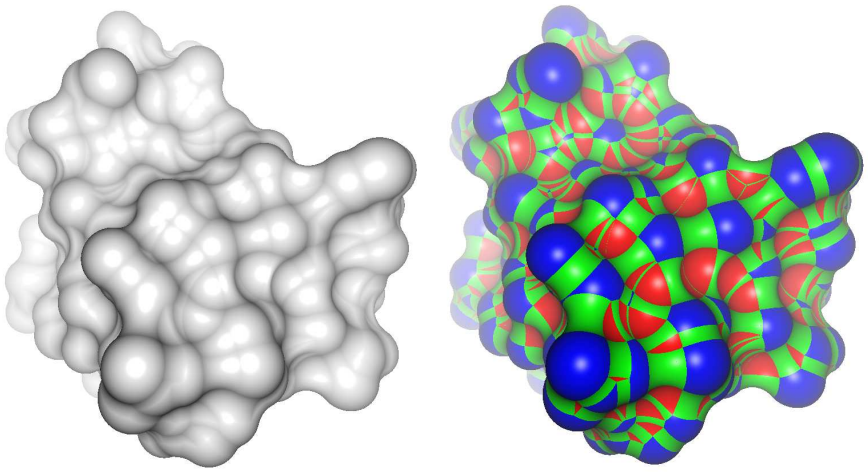
7.5.4 Sestavení programu

Program se sestavuje pomocí nástroje *Maven*. Doporučená minimální verze toho nástroje je 3.1. Pro instalaci knihovny awVoronoi i jejích závislostí do lokálního repozitáře můžeme použít například následující příkaz.

```
mvn install:install-file -Dfile=awvoronoi-run-1.0.3.jar
```

Vlastní sestavení aplikace provedeme příkazem `mvn package`, spuštěným v adresáři projektu. Tento příkaz se postará o stáhnutí ostatních závislostí z centrálního repozitáře a o následné vytvoření softwarového balíku. Ten najdeme v podadresáři `target`.

[4] http://www.rcsb.org



Obrázek 10: Srovnání výsledků pomocí vizualizace molekuly 1CRN programem CAVER Analyst. Vlevo byl výpočet proveden v programu CAVER Analyst. Vpravo byl výpočet proveden v OpenCL implementaci, kde jsou jednotlivá geometrická primitiva jsou zvýrazněna barvami.

7.5.5 Vizualizace výsledků

Pro vizualizaci výsledků byl využit program CAVER Analyst [17]. CAVER je program pro vizualizaci a analýzu vlastností molekul. Tento software je napsaný v jazyce Java a proto byla zvolena implementace této bakalářské práce v programovacím jazyku Java. Usnadní to případnou pozdější integraci.

CAVER Analyst umožňuje vizualizaci molekuly několika různými způsoby. Jedním z nich je zobrazení molekulárního povrchu, který by vytvořila sférická sonda pohybující se po těsně po povrchu molekuly. Výsledky programu v nové implementaci by se tedy měly shodovat. Za pomoci speciálního modulu byl zadavatelem software CAVER Analyst rozšířen o zobrazování dat vygenerovaných implementací zde představenou. Zobrazení umožňuje vykreslit oba výsledky přes sebe, čímž poskytuje jednoduchou možnost, jak vizuálně ověřit správnost nové implementace. Součástí práce tedy byl výpočet jednotlivých částí povrchu. Rendering byl realizován pomocí softwaru CAVER Analyst.

Na obrázku 10 vidíme molekulu s označením 1CRN, kde byl molekulární povrch vypočítán programem CAVER a pro porovnání je ukázán i stejný molekulární povrch, jak ho vypočítala nová implementace v OpenCL. Při tomto zobrazení jsou barvami odlišeny jednotlivá geometrická primitiva, ze kterých se molekulární povrch skládá. Modrou barvou jsou vyznačeny viditelné atomy, resp. jejich části - sférické pláty. Červenou barvou jsou vykresleny sférické trojúhelníky. A nakonec zeleně vidíme tory. Při vzájemném překrytí obou obrazů v programu CAVER zjistíme, že se výsledky shodují.

8 Porovnání implementací

V této kapitole se zaměříme na porovnání současné implementaci v programu CAVER, jež využívá jedno procesorové vlákno s novou implementací v OpenCL. Novou implementaci v OpenCL, vzniklou v rámci bakalářské práce, budeme testovat jak na grafické kartě, tak i na procesoru.

Obě implementace používají v principu stejný algoritmus, vycházejí tedy ze sestavené quasi-triangulace a dopředu vypočtených Voronoi vrcholů. Rozdílný je ovšem v použitých datových typech. Současná implementace používá dvojitou přesnost čísel v plovoucí desetinné čárce, kdežto nová implementace v OpenCL využívá pouze jednoduchou přesnost. Chyba, kterou s jednoduchou přesností desetinných čísel do programu zavedeme je vzhledem k účelu zanedbatelná.

Dalším rozdílem implementací je výpočet sférických a toroidálních plátů. Současná varianta vypočítává parametry obou struktur, kdežto nová varianta zjistí pouze informaci, zda plát nebo torus vykreslit.

Veškeré časy byly měřeny v Javě metodou `System.currentTimeMilis()`.

8.1 Molekuly použité v testech

Pro otestování programu byla vybrána sada molekul různých velikostí. Nejmenší molekula, na které probíhaly testy má pouhých 327 atomů. Naopak největší molekula, zvolená pro testování, má přes devadesát tisíc atomů. Tabulka 1 obsahuje výčet testovaných molekul spolu s počty zkoumaných atributů. Sloupec kód značí identifikátor molekuly v databázi uchovávající biologické makromolekulární struktury.

Kód	Atomů	Sférických troj.	Torů	Sférických plátů	Ořez. rovin
1CRN	327	650	2023	327	3544
1AF6	10050	23378	66983	10050	193756
1GKI	19536	47324	131955	19536	376236
1AON	58674	137960	382536	58674	973054
1JJ2	90418	184338	604364	90418	1225302

Tabulka 1: Použité molekuly pro výkonnostní testování implementací. Udávané počty jsou pro sondu o poloměru 1 Å. Poslední sloupec značí počet ořezávacích rovin sférických trojúhelníků.

Tabulka 2 udává počty hledaných geometrických primitiv na testovaných molekulách pro poloměr sondy $0,5$ Å. Za povšimnutí stojí, že se zvýšil hlavně počet sférických trojúhelníků. Kromě molekuly případu molekuly 1AON se tento počet zvedl o více než 60%. Počet torů je naopak ve všech případech menší, jelikož menší sonda již nepropojí tolik dvojic atomů. Počet sférických plátů zůstal nezměněn - jsou viditelné stále stejné atomy na povrchu molekuly. Ačkoliv se počet sférických trojúhelníků zvětšil, ořezávací roviny, tedy kolize, se vyskytují méně často. Počet atomů se při změně poloměru sondy samozřejmě nezmění, proto není uveden.

Kód	Sférických troj.	Torů	Sférických plátů	Ořez. rovin
1CRN	1104	1682	327	2648
1AF6	40764	52562	10050	115920
1GKI	71372	103766	19536	184014
1AON	194620	309961	58674	474200
1JJ2	337730	505424	90418	1154592

Tabulka 2: Použité molekuly pro výkonnostní testování implementací. Udávané počty jsou pro sondu o poloměru $0,5$ Å.

8.2 Výsledky měření současné implementace

Současná implementace, běžící v jednom vlákně, byla otestována na procesoru značky Intel model i7-4930K taktovaný na frekvenci 3.40 GHz. Tabulka 3 shrnuje výsledky měření. Časy se výhradně zvětšují s menším poloměrem sondy. To je způsobené tím, že menší sonda narazí na více geometrických primitiv. Měření současné implementace bylo pětkrát zopakováno a tabulka obsahuje průměrné hodnoty.

8.3 Výsledky měřené nové implementace na CPU

Na měření časů nové implementace v OpenCL, při výpočtech na CPU, byl využit totožný procesor jako pro měření současné implementace - Intel i7-4930K taktovaný na frekvenci 3.40 GHz. Každé měření bylo celkem třikrát zopakováno a v tabulce 3 jsou v příslušném sloupci uvedeny střední hodnoty. Do měření je započtený pouze čas výpočtů. Časy vytváření a kopírování potřebných datových struktur nejsou započteny.

8.4 Výsledky měřené nové implementace na GPU

Pro potřeby měření časů nové implementace v OpenCL byla použita grafická karta značky Nvidia model GeForce GTX 960M. Každé měření opět bylo celkem třikrát zopakováno a tabulka obsahuje pouze střední hodnoty. Do měření je započtený pouze čas výpočtů. Časy vytváření a kopírování potřebných datových struktur z grafické karty nejsou započteny.

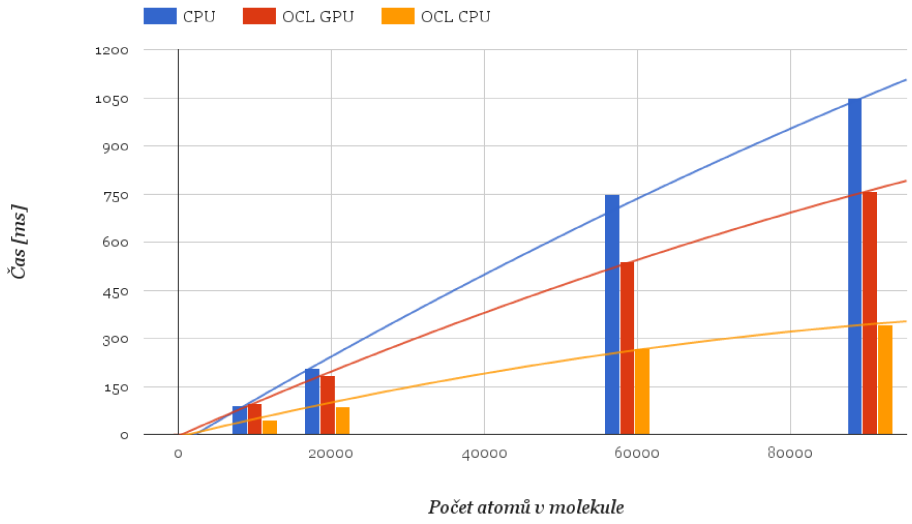
Kód	r [Å]	CPU [ms]	OCL GPU [ms]	Δ_1	OCL CPU[ms]	Δ_2
1CRN	0,1	4	3	0,75	1	0,25
	0,5	3	3	1,00	2	0,67
	1,0	2	3	1,50	1	0,50
	2,0	1	3	3,00	2	2,00
	5,0	1	4	4,00	1	1,00
1AF6	0,1	148	59	0,40	45	0,30
	0,5	180	105	0,58	56	0,31
	1,0	90	97	1,08	46	0,51
	2,0	32	56	1,75	20	0,63
	5,0	14	61	4,36	22	1,57
1GKI	0,1	327	116	0,35	86	0,26
	0,5	347	178	0,51	111	0,32
	1,0	206	184	0,89	88	0,43
	2,0	63	111	1,76	49	0,78
	5,0	19	82	4,32	29	1,53
1AON	0,1	1182	357	0,30	347	0,29
	0,5	1124	535	0,48	356	0,32
	1,0	750	540	0,72	267	0,36
	2,0	312	396	1,27	160	0,51
	5,0	95	308	3,24	124	1,31
1JJ2	0,1	2053	575	0,28	576	0,28
	0,5	2005	1017	0,51	670	0,33
	1,0	1049	759	0,72	341	0,33
	2,0	487	598	1,23	231	0,47
	5,0	96	365	3,80	123	1,28

Tabulka 3: Časy výpočtů konkrétních molekul. Sloupec r označuje poloměr sondy. Sloupec nadepsaný CPU obsahuje časy naměřené současnou implementací. Sloupce OCL CPU a OCL GPU označují časy naměřené novou implementací v OpenCL spuštěné na procesoru, resp. grafické kartě. Znakem Δ označujeme podíl časů vůči současné implementaci. Podíly jsou oříznuté na dvě desetinná místa.

8.5 Diskuse výsledků

Na obrázku 11 je k dispozici graf s porovnáním současné CPU implementace a nové implementace v OpenCL, s časy změřenými na zmíněném procesoru a grafické kartě. Pro vstupní data grafu byla vybrána sonda o poloměru 1 Å. Z grafu je patrné, že pro molekuly s více než dvaceti tisíci atomy se již vyplatí výpočet přenést na OpenCL, jelikož nabízí minimálně 10% zrychlení v případě výpočtů na GPU a minimálně 50% zrychlení na CPU, pokud uvažujeme i sondy menší než 1 Å. Pro molekuly s více jak šedesáti tisíci atomy pak výpočty na GPU vykazují přibližně 30% zrychlení a na CPU více než 60%. Pro větší poloměr sondy, jak ukazuje tabulka 3, nemusí být přesun výhodný. Nicméně kdyby se přenesly všechny výpočty na novou implementaci, spouštěnou na CPU, ovlivní se tím negativně jen výpočty, které trvají na obou platformách relativně krátkou dobu. V absolutních číslech se jedná o maximálně 30 ms za použití sondy o poloměru 5 Å. Takové rozdíly časů nejsou pro uživatele ani postřehnutelné. Negativní vliv by tyto malé odchylky mohly mít v případech simulací v čase, kdy molekuly mění svoji strukturu a je potřeba série výpočtů napříč simulací.

Čísla měření ukazují, že požadovaného výsledku zrychlení se podařilo dosáhnout v případech, kde se zrychlení nejvíce projeví. Tedy v případech velkých molekul a malého poloměru sondy. U velkých molekul se zvoleným poloměrem sondy menším než 1 Å se výpočty srazily na méně než polovinu. V případě velkých molekul a poloměru sondy 1 Å se podařila doba výpočtů srazit na přibližně třetinu.



Obrázek 11: Časy měření současné implementace (CPU) a nové implementace (OCL CPU a OCL GPU) zanesené v grafu, vztaženo k počtu atomům v molekule. Naměřené časy jsou pro sondu o poloměru 1 Å . Časy jsou proloženy kvadratickou regresní funkcí.

Tyto výsledky lze považovat ze uspokojivé i když je pravděpodobné, že v případě dalšího (nemalého) úsilí, lze nová implementace ještě vylepšit.

V případech použití sondy o větším poloměru se nepovedlo současné implementaci konkurovat. Režie spouštění OpenCL kódu je větší, než možná úspora. Spuštění na procesoru tímto neduhem tolik netrpí, ale při spuštění na grafické kartě je poznat, že přenos dat do paměti grafické karty je náročná operace, zpomalující začátek vlastního výpočtu.

Zajímavé také je, že OpenCL implementace spuštěná na dostatečně výkonném CPU je rychlejší, než výpočty spuštěné na grafické kartě. A to i pro větší molekuly, resp. množství výpočtů. Domnívám se, že to je způsobené nezbytným větvením v programu, na které nejsou současné grafické karty vybaveny tak dobře jako současné procesory. Současné procesory velice přesně odhadují jakou větví se program vydá a proto nemusí docházet k častému zpoždění výpočtů při čekání na data po rozvětvení programu. Současné grafické karty největší propustnost dodávají v případě úplné vektorizace, naprosté minimalizace skoků, zredukované synchronizaci i přístupu do globálního paměťového rozsahu.

9 Možnosti dalšího rozšiřování

Následujícím krokem by logicky mohlo být integrování OpenCL implementace do CAVERu. Integrace zpočátku může fungovat jako rozšíření modulu pro výpočet molekulárního povrchu s možností zadat výpočty na grafickou kartu.

Jelikož se implementace v OpenCL výkonnostně osvědčila, je možné pokračovat s převodem i dalších částí, které jsou výpočetně náročné. Ideálním kandidátem na zrychlení je algoritmus sestavování quasi-triangulace. Současná implementace quasi-triangulaci sestavuje pro velké molekuly více než deset sekund. Pro uživatele by bylo přívětivé, kdyby se tento čas výrazně srazil.

Kromě implementace quasi-triangulace je možné na grafickou kartu také přenést i výpočty bottlenecků. Jestliže bude na grafické kartě k dispozici triangulace, ušetří se čas na přesuny mezi hlavní pamětí počítače a pamětí grafické karty. Bottlenecky i triangulace pak budou potřebné i pro výpočet molekulárního povrchu a dutinek.

10 Závěr

Tato práce teoreticky pokryla definici a vlastnosti aditivně vážených i obyčejných Voronoi diagramů. Dále byl teoreticky rozebrán vztah Voronoi diagramů a duální triangulace. Práce se dále zabývá popisem vybraných algoritmů pro sestrojení Voronoi diagramů resp. quasi-triangulace. Na tomto nezbytném teoretickém základu se zakládají základní geometrické výpočty pro analýzu biomolekul. Pro zpracování této teoretické části jsem čerpal znalosti z referenční literatury poskytnuté vedoucím práce.

Práce neopomíná ani letmý přehled současného vztahu hardwaru a paralelních výpočtů. Zdůvodňuje potřebu paralelních algoritmů, které v budoucnosti budou mít ještě větší význam než dnes. Konkrétně byly představeny základní principy programování v OpenCL.

Dále byly rozebrány části výpočtů při analýze biomolekul, které by bylo vhodné implementovat paralelně. Následně práce detailně rozebírá jejich novou implementaci v jazyce OpenCL. Tato implementace splnila rychlostní očekávání pro analýzu velkých molekul pří průzkumu pomocí sondy s malým poloměrem. Pro tuto sadu vstupů je nová implementace opravdu znatelně rychlejší. Více než dvou sekundové časy se podařilo zredukovat pod 700 ms, při spuštění na stejném procesoru. Výhodou nové implementace je možnost výpočty přenést na grafickou kartu a tím ulehčit procesoru, který mezitím může počítat jiné úlohy spojené s analýzou biomolekul.

V průběhu realizace práce jsem se seznámil se základy zpracování prostorových výpočtů a osvojil si paradigmata programovacího jazyka OpenCL.

Literatura

[1] MANAK, M. Computing Dual Structure of 3D Additively Weighted Voronoi Diagram [software]. Červenec 2016. Verze 1.0.3. Dostupné z: http://awvoronoi.sf.net. Licence MIT.

[2] Clpp Library [software], Červen 2011, Verze 1.0b3, Dostupné z: https://code.google.com/archive/p/clpp/downloads. Licence MIT.

[3] JogAmp: High Performance Cross Platform Java Li- braries for 3D Graphics, Multimedia and Processing [software]. Dostupné z: http://jogamp.org/. 2015.

[4] AURENHAMMER, F. – KLEIN, R. Voronoi diagrams. *Handbook of computational geometry*. 2000, 5, s. 201–290.

[5] BARBER, C. B. – DOBKIN, D. P. – HUHDANPAA, H. The quickhull algorithm for convex hull. Technical report, Technical Report GCG53, The Geometry Center, MN, 1993. `http://dl.acm.org/citation.cfm?id=235821`.

[6] BLELLOCH, E. Prefix sums and their applications. 1990.

[7] BOISSONNAT, J.-D. – TEILLAUD, M. *Effective computational geometry for curves and surfaces*. Springer, 2007.

[8] BROWN, K. Q. *Geometric transforms for fast geometric algorithms*. PhD thesis, Carnegie-Mellon, 1979.

[9] CONNOLLY, M. L. Solvent-accessible surfaces of proteins and nucleic acids. *Science*. 1983, 221, 4612, s. 709–713.

[10] GREEN, P. J. – SIBSON, R. Computing Dirichlet tessellations in the plane. *The Computer Journal*. 1978, 21, 2, s. 168–173.

[11] GUIBAS, L. – STOLFI, J. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM transactions on graphics (TOG)*. 1985, 4, 2, s. 74–123.

[12] JIRKOVSKY, L. Výpočet a vizualizace dutin ve velkých modelech molekul. 2014.

[13] JIRKOVSKY, L. Finding Cavities in a Molecule. 2012. `http://www.cescg.org/CESCG-2012/proceedings/CESCG-2012.pdf`.

[14] KIM, D.-S. – CHO, Y. – KIM, D. Euclidean Voronoi diagram of 3D balls and its computation via tracing edges. *Computer-Aided Design*. 2005, 37, 13, s. 1412–1424.

[15] KIM, D.-S. et al. Quasi-triangulation and interworld data structure in three dimensions. *Computer-Aided Design*. 2006, 38, 7, s. 808–819.

[16] KOHOUT, J. *Delaunay triangulation in parallel and distributed environment.* PhD thesis, Západočeská univerzita v Plzni, 2005.

[17] KOZLIKOVA, B. et al. CAVER Analyst 1.0: graphic tool for interactive visualization and analysis of tunnels and channels in protein structures. *Bioinformatics.* 2014, s. btu364.

[18] LEE, D.-T. – SCHACHTER, B. J. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences.* 1980, 9, 3, s. 219–242.

[19] LINDOW, N. – BAUM, D. – HEGE, H.-C. Voronoi-based extraction and visualization of molecular paths. *Visualization and Computer Graphics, IEEE Transactions on.* 2011, 17, 12, s. 2025–2034.

[20] MANAK, M. – KOLINGEROVA, I. Extension of the edge tracing algorithm to disconnected Voronoi skeletons. *Information Processing Letters.* 2016, 116, 2, s. 85–92.

[21] MEDVEDEV, N. N. et al. An algorithm for three-dimensional Voronoi S-network. *Journal of computational chemistry.* 2006, 27, 14, s. 1676–1692.

[22] MIRTICH, B. Efficient Algorithms for Two-Phase Collision Detection. Technical Report TR97-23, MERL - Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, December 1997. Dostupné z: `http://www.merl.com/publications/TR97-23/`.

[23] OHYA, T. – IRI, M. – MUROTA, K. Improvements of the incremental method for the Voronoi diagram with computational comparison of various algorithms. *J. OPER. RES. SOC. JAPAN.* 1984, 27, 4, s. 306–336.

[24] OKABE, A. et al. *Spatial tessellations: concepts and applications of Voronoi diagrams.* 501. John Wiley & Sons, 2009.

[25] RYU, J. – PARK, R. – KIM, D.-S. Connolly surface on an atomic structure via Voronoi diagram of atoms. *Journal of Computer Science and Technology.* 2006, 21, 2, s. 255–260.

[26] SCHÄLING, B. *The boost C++ libraries.* Boris Schäling, 2011.

[27] SHAMOS, M. I. – HOEY, D. Closest-point problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, s. 151–162. IEEE, 1975.

[28] SUGIHARA, K. – IRI, M. Construction of the Voronoi diagram for one million generators in single-precision arithmetic. *Proceedings of the IEEE.* 1992, 80, 9, s. 1471–1484.