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

Diplomová práce

Využití paralelismu pro analýzu biomolekul

Plzeň, 2014 Zuzana Majdišová

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracovala samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 8. května 2014

Zuzana Majdišová

Poděkování

Na tomto místě bych ráda poděkovala vedoucímu mé diplomové práce panu Mgr. Martinu Maňákovi za odborné vedení, vstřícnost, ochotné poskytnutí cenných rad a připomínek, které zásadním způsobem přispěly k dokončení této práce, poskytnuté konzultace a materiály. Poděkování patří také prof. Dr. Ing. Ivaně Kolingerové za její konkrétní a podnětné připomínky k textu diplomové práce, které zvýšily jeho kvalitu.

Abstrakt

Efektivní analýza biomolekul často vyžaduje popis prostorových vztahů mezi jednotlivými atomy pomocí aditivně váženého Voronoi diagramu, avšak pro velké molekuly a sekvence molekul je získání diagramu časově i paměťově náročné. Cílem práce je prozkoumat možnosti využití vyššího stupně paralelismu GPU k urychlení výpočtu diagramu, navrhnout a implementovat vhodné řešení pro velké molekuly a sekvence. Navržené řešení je založeno na algoritmu trasování hran, přičemž výpočet koncového vrcholu Voronoi hrany je počítán pomocí GPGPU.

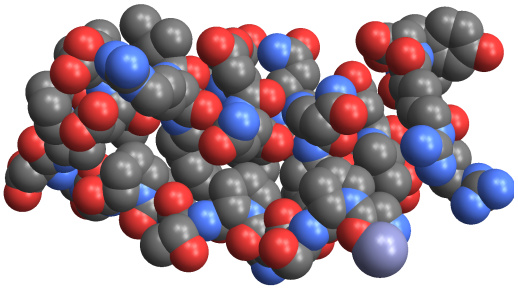
Abstract

Effective analysis of biomolecules often requires a description of the spatial relationships between individual atoms using the additively weighted Voronoi diagram. However obtaining such diagrams for large molecules and molecular sequences is a time and memory consuming task. The goal is to explore the possibility of using a higher degree of parallelism offered by GPU to accelerate the computation, and to design and implement appropriate solutions to large molecules and sequences. The proposed solution is based on the edge-tracing algorithm and computes the end vertex of Voronoi edge using GPGPU.

Obsah

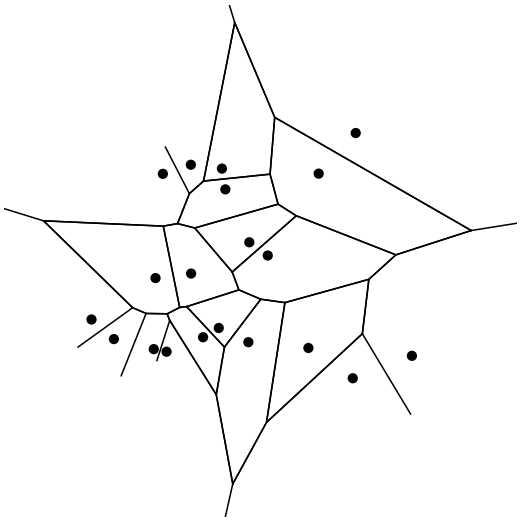
1 Úvod

Biomolekuly jsou chemické sloučeniny vyskytující se v živých organismech. Skládají se především z atomů uhlíku a vodíku. Příklad molekuly je znázorněn na Obrázku 1.1. V oblastech chemie, které se zabývají vývojem léků a molekulární enzymologií, je pak důležité provádět analýzu těchto biomolekul. Během analýzy biomolekul je potřeba popsat prostorové vztahy mezi jednotlivými atomy, k tomu se využívá aditivně vážený Voronoi diagram. Voronoi diagram se konkrétně používá k urychlení výpočtu molekulárního povrchu, výpočtu objemu, analýzy vnitřních dutin, hledání tunelů v molekule atd. Tunely v molekulách proteinů určují jejich zásadní vlastnosti, jako je například vyvolání chemické reakce nebo složitost modifikace molekuly. Analýza tunelů je využívána v proteinovém inženýrství, kdy pomáhá k lepšímu porozumnění konkrétních chemických reakcí a může být provedeno jejich následné ovlivnění. Chemická reakce je ovlivněna modifikací struktury proteinu tak, že je provedeno rozšíření nebo naopak uzavření tunelu v určitém místě a tím je tato reakce usnadněna či znemožněna.



Obrázek 1.1: Molekula s PDB ID 1PPT (PDB ID je identifikátor molekuly v Protein Data Bank)

Voronoi diagram byl poprvé charakterizován a definován teprve v roce 1908 ruským matematikem Georgijem Feodosjevičem Voronojem, přestože byla jeho myšlenka známá již mnohem dříve. Jedná se o způsob dekompozice metrického prostoru, přičemž je dělení prostoru dáno vzdálenostmi k dané diskrétní množině objektů. Voronoi diagram má širokou škálu uplatnění v mnoha vědeckých disciplínách včetně výpočetní geometrie, přičemž se používá v mnoha variantách. Nejznámější a nejvíce prostudovanou verzí je Voronoi diagram množiny bodů $VD(P)$ v prostoru, jehož dimenze je dva nebo vyšší. Příklad takového diagramu je na Obrázku 1.2.



Obrázek 1.2: Voronoi diagram pro množinu bodů v E^2

Existují však oblasti, ve kterých se pro řešení problému více hodí Voronoi diagram jiných geometrických tvarů než bodů. Takovou oblastí je i molekulární biologie, konkrétně analýza biomolekul, kdy se mnohem více jako reprezentace atomu hodí koule, a tudíž je pro nás mnohem významnější Euklidovský Voronoi diagram koulí $VD(B)$ v E^3, který je také označován jako aditivně vážený Voronoi diagram. Konstrukce tohoto diagramu je možná pomocí algoritmu sledování Voronoi hran, který byl navržen Kimem a kol. [7], a jehož časová složitost je $\mathcal{O}(n^2)$, kde n je počet atomů v molekule.

Z uvedené časové složitosti algoritmu sledování hran vyplývá, že pro velké molekuly nebo sekvence molekul je konstrukce diagramu časově náročná. To vede ke snaze urychlit konstrukci aditivně váženého Voronoi diagramu ať už úpravou samotného algoritmu trasování hran nebo jeho zparalelizováním.

Cílem této diplomové práce je prozkoumat možnosti využití GPGPU (technika umožňující obecný výpočet na GPU) k urychlení výpočtu diagramu a na základě toho navrhnout úpravu algoritmu sledování Voronoi hran. Dále má být provedeno rozšíření knihovny aw-Voronoi, která byla vyvinuta M. Maňákem a implementuje konstrukci Voronoi diagramu na základě algoritmu trasování hran a několika jeho modifikací, o implementaci navržené úpravy algoritmu pro účely výpočtu na GPGPU. Součástí diplomové práce bude taktéž porovnání navržené modifikace s dostupnými implementace z hlediska časové náročnosti a urychlení. Celou práci nakonec uzavřeme zhodnocením dosažených výsledků.

2 Aditivně vážený Voronoi diagram

V této kapitole se budeme zabývat aditivně váženým Voronoi diagramem v E^3, který je také znám pod názvem Euklidovský Voronoi diagram koulí. Uvedeme si zde jeho geometrické vlastnosti a topologii, krátce shrneme vlastnosti kvazitriangulace, což je struktura, která je k aditivně váženému Voronoi diagramu duální, a popíšeme si zakládní algoritmus konstrukce tohoto diagramu, kterým je algoritmus sledování hran. Výklad je uveden na základě článků [7], [8], [4] a [11].

2.1 Definice a terminologie

Definice 2.1: Nechť $B = \{b_1, b_2, \ldots, b_n\}$ je množina n koulí v E^3. Koule $b_i = (c_i, r_i)$ je generující koule Voronoi diagramu se středem $c_i = (x_i, y_i, z_i)$ a poloměrem $r_i \in \mathbb{R}$. Pak pro každou generující kouli $b_i \in B$ existuje odpovídající *Voronoi buňka* VR_i (Voronoi region), která je definována jako:

$$VR_i = \{p | \forall b_j \in B, j \neq i : \|p - c_i\| - r_i \leq \|p - c_j\| - r_j\}, \tag{2.1}$$

kde $\|p - c_i\|$ resp. $\|p - c_j\|$ je euklidovská vzdálenost mezi body p a c_i resp. p a c_j. A *aditivně vážený Voronoi diagram* pro B je pak definován jako množina Voronoi buněk:

$$VD(B) = \{VR_1, VR_2, \ldots, VR_n\}. \tag{2.2}$$

Při konstrukci Voronoi diagramu předpokládáme, že žádná z generujících koulí není celým svým objemem umístěna v jiné generující kouli, tím je zábráněno vzniku prázdné Voronoi buňky. Protínající se koule jsou však povoleny.

2.1.1 Geometrické a topologické vlastnosti

Voronoi buňky rozdělujeme na ohraničené a neohraničené. Buňka je neohraničená, pokud její generující koule leží na konvexní obálce množiny B, v ostatních případech se

jedná o buňku ohraničenou. Hranice Voronoi diagramu jsou v E^3 tvořeny následujícími primitivy:

Stěna Voronoi buňky (*Voronoi face*)

Stěna Voronoi buňky f je definována dvěma bezprostředně sousedícími generátory. Z geometrického hlediska se jedná o spojitou podmnožinu dvoudílného hyperboloidu.

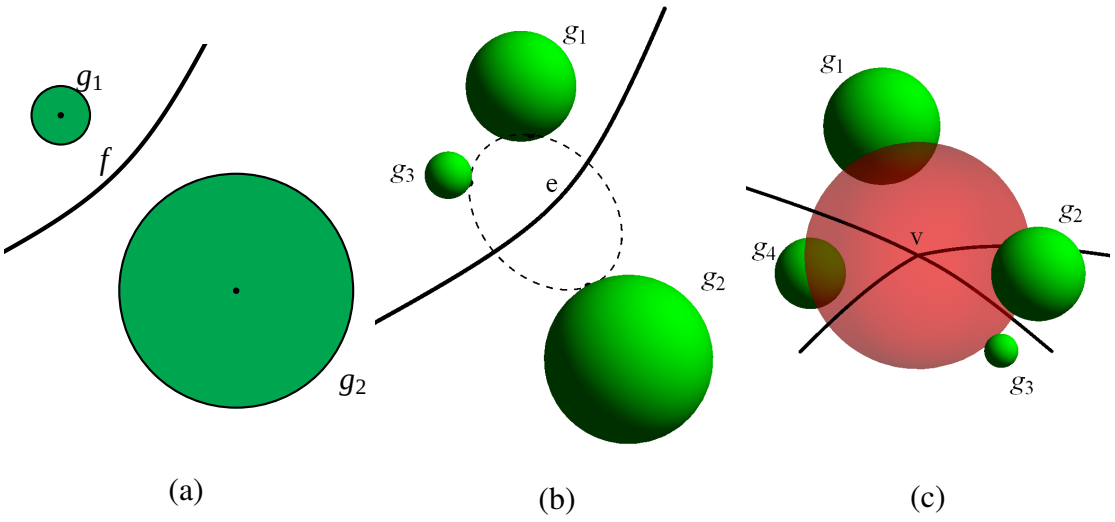
Voronoi hrana (*Voronoi edge*)

Voronoi hrana e je určena třemi sousedícími generátory. Z geometrického hlediska se jedné o kuželosečku, která je definována jako průnik dvou (sousedních) stěn Voronoi buňky.

Voronoi vrchol (*Voronoi vertex*)

Voronoi vrchol v je stanoven jako průsečík Voronoi hran. Každý Voronoi vrchol lze definovat pomocí čtveřice sousedících generátorů, přičemž se jedná o střed sféry, která je k těmto čtyřem generátorům tečná a neobsahuje ani neprotíná žádné jiné koule z množiny B vyjma těch generujících.

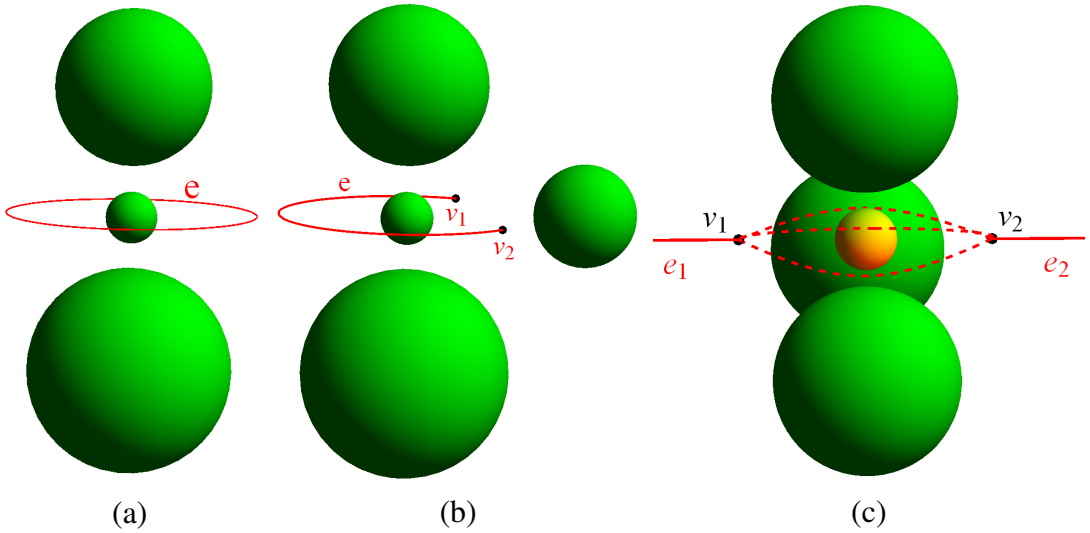
Všechny popsané elementy Voronoi diagramu včetně jejich generátorů jsou znázorněny na Obrázku 2.1. Abychom zabránili tomu, že budou primitiva produkována více koulemi než je nutné, předpokládáme, že jsou generátory umístěné v obecné pozici.



Obrázek 2.1: Základní prvky aditivně váženého Voronoi diagramu: (a) Stěna Voronoi buňky f definovaná dvěma generátory (g_1, g_2) v E^2; (b) Voronoi hrana e definovaná třemi generátory (g_1, g_2, g_3); (c) Voronoi vrchol v definovaný čtyřmi generátory (g_1, g_2, g_3, g_4)

Během konstrukce Voronoi diagramu koulí můžeme narazit na několik konfigurací, které jsou speciálními případy pouze pro Voronoi diagram koulí a nikdy nemohou vzniknout ve Voronoi diagramu bodů. Mezi tyto speciální konfigurace, které jsou

znázorněny na Obrázku 2.2, patří definování eliptické hrany, která není omezena vrcholem, dále situace, kdy tři generátory jednoznačně neurčují pouze jednu hranu nebo čtveřice generátorů negeneruje pouze jeden vrchol.

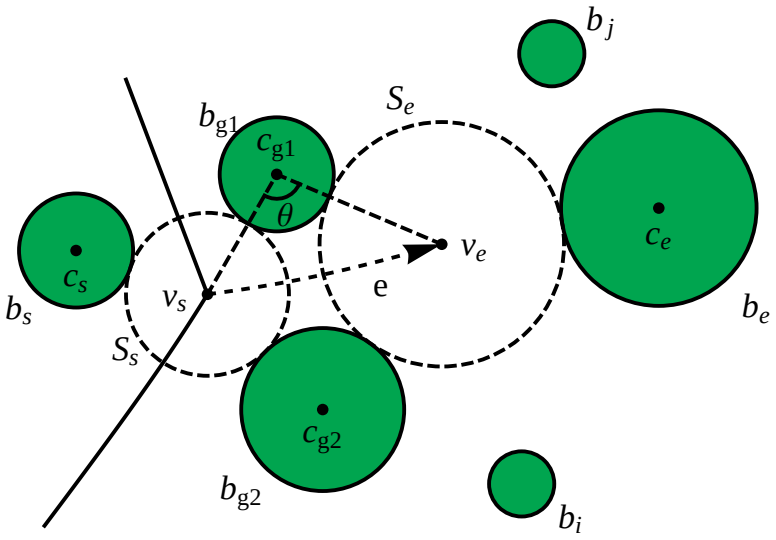


Obrázek 2.2: Konfigurace, které mohou nastat pro aditivně vážený Voronoi diagram: (a) Generátor malého průměru mezi dvěma generátory velkého průměru definují eliptickou hranu e; (b) Čtveřice generátorů definuje dva vrcholy $v_1, v_2 \in e$; (c) Jedna čtveřice generujících koulí definuje dva Voronoi vrcholy v_1 a v_2 a jedna trojice generátorů určuje dvě Voronoi hrany e_1 a e_2

2.2 Úhlová vzdálenost a orientace hrany

Daná hrana e je omezena maximálně dvěma vrcholy a je orientovaná od *počátečního vrcholu* v_s ke *koncovému vrcholu* v_e, přičemž tečná sféra odpovídající vrcholu v_s se nazývá *počáteční tečná sféra* S_s a sféra příslušná k vrcholu v_e se nazývá *koncová tečná sféra* S_e. Orientovaná hrana je vždy definována třemi sousedními generátory, které se nazývají gate koule (*gate balls*) b_{g1}, b_{g2} a b_{g3}. Tyto tři koule jsou společnými generátory pro oba vrcholy v_s a v_e. Zbývající koule generující počáteční, respektive koncový vrchol se označuje *počáteční koule* b_s, respektive *koncová koule* b_e. Na Obrázku 2.3 je zobrazena celá tato situace pro analogii v prostoru E^2, proto je hrana definována pouze dvěma generátory.

Abychom stanovili koncový vrchol v_e pro hranu e, musíme vypočítat tečnou sféru S_e, přičemž máme dány tři gate koule b_{g1}, b_{g2} a b_{g3} a počáteční kouli b_s. Pro určení tečné sféry S_e budeme testovat, zda může být tato sféra umístěna mezi čtveřicí generátorů skládající se z daných tří gate koulí b_{g1}, b_{g2} a b_{g3} a koule z množiny $K = B \setminus \{b_{g1}, b_{g2}, b_{g3}\}$, kterou označujeme jako *množinu kandidátů* pro hranu e. Poznamenejme, že i počáteční koule b_s je do množiny kandidátů zařazena.



Obrázek 2.3: Aditivně vážený Voronoi diagram v E^2 a související terminologie

Pro účely uvedeného testu používáme úhlovou vzdálenost (*angular distance*) θ. Nechť koule b_{g1} má nejmenší poloměr z daných tří gate koulí, pak úhlová vzdálenost θ bodu $p \in e$ z počátečního vrcholu v_s je úhel $\angle v_s c_{g1} p$, kde c_{g1} je střed gate koule b_{g1}. Úhlová vzdálenost nabývá hodnot z intervalu $(0, 2\pi)$, přičemž kladný směr je definován směrem hrany z počátečního vrcholu v_s. Pokud je spočítáno více úhlových vzdáleností pro různé koule, budeme pro stanovení koncového vrcholu hledat kouli, pro kterou je počítaný úhel minimální.

2.3 Kvazitriangulace

Duální reprezentací k aditivně váženému Voronoi diagramu v E^3 je *kvazitriangulace* $QT(B)$. Poznamenejme, že pro Voronoi diagram koulí, které mají všechny stejný poloměr, odpovídá jeho kvazitriangulace klasické Delaunay triangulaci.

Definice 2.2: Nechť B je množina generujících koulí v E^3. Pak kvazitriangulace pro množinu B je definována jako:

$$QT(B) = (V^Q, E^Q, F^Q, C^Q), \tag{2.3}$$

kde

$$\begin{aligned} V^Q &= \{v_1^Q, v_2^Q, \ldots, v_n^Q\}, \\ E^Q &= \{e_1^Q, e_2^Q, \ldots, e_n^Q\}, \\ F^Q &= \{f_1^Q, f_2^Q, \ldots, f_n^Q\}, \\ C^Q &= \{c_1^Q, c_2^Q, \ldots, c_n^Q\} \end{aligned}$$

označuje vrcholy, hrany, stěny buněk a buňky v kvazitriangulaci.

Podobně můžeme zadefinovat i aditivně vážený Voronoi diagram, tzn. platí $VD(B) = \{V^V, E^V, F^V, C^V\}$, kde V^V, E^V, F^V a C^V postupně označují Voronoi vrcholy, hrany, stěny buněk a buňky. Duální operátor $\mathcal{D}$ je poté definován následovně:

- $\mathcal{D} : c^V \Rightarrow v^Q$ je mapování Voronoi buňky $c^V \in C^V$ na vrchol kvazitriangulace $v^Q \in V^Q$, přičemž vrchol v^Q je střed p generující koule b, která definuje Voronoi buňku c^V.
- $\mathcal{D} : f^V \Rightarrow e^Q$ mapuje stěnu Voronoi buňky $f^V \in F^V$ na hranu kvazitriangulace $e^Q \in E^Q$, přičemž hrana e^Q je úsečka s krajními body v_i^Q a v_j^Q. Stěna Voronoi buňky f^V, která je k této hraně duální, je generována pomocí koulí b_i a b_j.
- $\mathcal{D} : e^V \Rightarrow f^Q$ je mapování Voronoi hrany $e^V \in E^V$ na stěnu buňky kvazitriangulace $f^Q \in F^Q$, přičemž stěna f^Q je z geometrického hlediska trojúhelník, jehož vrcholy jsou v_i^Q, v_j^Q a v_k^Q. Hrana Voronoi buňky e^V, která je k této stěně duální, je generována pomocí koulí b_i, b_j a b_k.
- $\mathcal{D} : v^V \Rightarrow c^Q$ mapuje Voronoi vrchol $v^V \in V^V$ na buňku kvazitriangulace $c^Q \in C^Q$, přičemž buňka c^Q topologicky představuje čtyřstěn, jehož vrcholy odpovídají čtveřici vrcholů v^Q. Každá dvojice v^Q definuje hranu e^Q a každá trojice v^Q určuje stěnu f^Q.

V kvazitriangulaci $QT(B)$ může nastat několik konfigurací, které způsobí výskyt anomálií. Vzniklé anomálie mohou být následující:

Multiplicita (*Multiplicity Anomaly*)

Dva čtyřstěny v kvazitriangulaci sdílejí dvě nebo více stěn. Tato anomálie je způsobena definicí dvou Voronoi vrcholů pomocí jedné čtveřice generátorů.

Singularita (*Singularity Anomaly*)

V kvazitriangulaci se protíná čtyřstěn a degenerovaný čtyřstěn, přičemž jejich průnikem vzniká hrana. Tato anomálie nastane, pokud se ve stěně Voronoi buňky nachází topologická díra.

Degenerativnost (*Degeneracy Anomaly*)

V kvazitriangulaci je definován trojúhelník, který není součástí žádného čtyřstěnu, tzn. tento trojúhelník je degenerovaný čtyřstěn. Tato anomálie je způsobena eliptickou hranou bez vrcholů v aditivně váženém Voronoi diagramu.

2.4 Matematický aparát

V této části si popíšeme výpočet vrcholů Voronoi diagramu, který je důležitý při konstrukci topologie. Dále si krátce shrneme i výpočet hran, kterému se však dá většinou vyhnout. Výpočty budou popsány na základě [7]. Výpočet stěn Voronoi buněk si nebudeme uvádět, jelikož není pro konstrukci Voronoi diagramu stěžejní. Rovnice popisující stěny Voronoi buněk jsou potřebné pouze v případě, že chceme tyto stěny zobrazit, počítat objemy či povrchy jednotlivých Voronoi buněk atd. V tomto případě je možné si výpočet nastudovat v [7] na stranách 1417 a 1418.

2.4.1 Voronoi vrcholy

Jak již bylo popsáno v Sekci 2.1.1, Voronoi vrchol v je střed prázdné sféry tečné ke čtyřem nejbližším generátorům. Předpokládejme, že b_1, b_2, b_3 a b_4 označují generující koule Voronoi vrcholu v, a tudíž k nim musíme vypočítat odpovídající tečnou sféru $S(v)$. Bez ztráty na obecnosti budeme předpokládat, že generátor b_4 má z generujících koulí nejmenší poloměr. Pak můžeme všechny generující koule zmenšit o tento poloměr, čímž se z generátoru b_4 stane bod. Navíc na koule aplikujeme transformaci tak, aby generátor b_4 odpovídal počátku souřadného systému. Pokud si jednotlivé generující koule popíšeme vztahem $b_i = (x_i, y_i, z_i, r_i)$, kde (x_i, y_i, z_i) je střed i-té koule a r_i je její poloměr, pak transformaci i-té koule včetně jejího zmenšení můžeme definovat následujícím vztahem:

$$b_i := b_i - b_4. \tag{2.4}$$

Tečnou sféru $S(v) = (x, y, z, r)$, kde (x, y, z) je její střed a r poloměr, vypočítáme na základě vyřešení následující soustavy rovnic:

$$(x - x_1)^2 + (y - y_1)^2 + (z - z_1)^2 = (r + r_1)^2 \tag{2.5a}$$
$$(x - x_2)^2 + (y - y_2)^2 + (z - z_2)^2 = (r + r_2)^2 \tag{2.5b}$$
$$(x - x_3)^2 + (y - y_3)^2 + (z - z_3)^2 = (r + r_3)^2 \tag{2.5c}$$
$$x^2 + y^2 + z^2 = r^2 \tag{2.5d}$$

Pokud nyní od rovnic (2.5a), (2.5b) a (2.5c) odečteme rovnici (2.5d), získáme systém tří lineárních rovnic o čtyřech neznámých x, y, z a r, který můžeme popsat následující

rozšířenou maticí soustavy:

$$\left(\begin{array}{cccc|c} x_1 & y_1 & z_1 & r_1 & \dfrac{x_1^2+y_1^2+z_1^2-r_1^2}{2} \\ x_2 & y_2 & z_2 & r_2 & \dfrac{x_2^2+y_2^2+z_2^2-r_2^2}{2} \\ x_3 & y_3 & z_3 & r_3 & \dfrac{x_3^2+y_3^2+z_3^2-r_3^2}{2} \end{array}\right). \tag{2.6}$$

Řešením této soustavy jsou výrazy o jedné neznámé, přičemž neznámou budeme volit tak, aby byla zachována numerická stabilita řešení, viz [10]. Získané výrazy dosadíme za příslušné proměnné do rovnice (2.5d), čímž dostaneme kvadratickou rovnici, kterou budeme řešit v oboru reálných čísel. Na získaná řešení aplikujeme zpětnou transformaci do původního souřadného systému a zmenšení poloměru, tzn. přičteme vektor $(x_4, y_4, z_4, -r_4)$. Budou nás zajímat pouze sféry, které budou mít po provedení této úpravy reálný nezáporný poloměr.

V závislosti na konfiguraci čtveřice generátorů můžeme získat žádnou, jednu, dvě nebo nekonečně tečných sfér.

2.4.2 Voronoi hrany

Voronoi hrana je definována třemi generujícími koulemi b_1, b_2 a b_3, přičemž musí platit, že body tvořící hranu e jsou stejně vzdálené od těchto tří koulí. Výpočet Voronoi hrany bychom proto mohli provést podobně jako výpočet Voronoi vrcholu, který byl uveden v Sekci 2.4.1.

Nicméně také víme, že Voronoi hrana je vždy kuželosečka, tudíž ji můžeme reprezentovat jako *racionální kvadratickou Bézierovu křivku*:

$$\beta(t) = \frac{w_0(1-t)^2 p_0 + 2w_1(1-t)t p_1 + w_2 t^2 p_2}{w_0(1-t)^2 + 2w_1(1-t)t + w_2 t^2}, t \in [0,1], \tag{2.7}$$

kde p_0, p_1 a p_2 jsou body řídícího polygonu a w_0, w_1 a w_2 jsou odpovídající váhy.

Abychom však mohli Voronoi hranu vyjádřit jako Bézierovu křivku ve tvaru (2.7), musíme znát pět parametrů, kterými jsou oba koncové body, tečné vektory v těchto bodech a jeden bod na křivce.

Koncové body jsou dány jako Voronoi vrcholy, kterými je hrana omezena, přičemž počáteční vrchol odpovídá bodu p_0 a koncový vrchol bodu p_2.

Tečné vektory ke křivce v bodech p_0 a p_2 určíme na základě Věty 2.1 a z ní vyplývajícího Důsledku 2.1, přičemž budeme předpokládat, že hrana e je definována třemi generátory

b_1, b_2 a b_3, pro jejichž poloměry platí $r_1 \leq r_2 \leq r_3$, v je Voronoi vrchol, ve kterém chceme definovat tečný vektor hrany, a vektory t_i jsou vyjádřeny vztahem:

$$t_i = \frac{c_i - v}{\|c_i - v\|}, i = 1, 2, 3,$$

kde c_i je střed generátoru b_i.

Věta 2.1: *Tečný vektor t k Voronoi hraně e v bodě v svírá s vektory t_1, t_2 a t_3 stejný úhel.*

Důkaz. Viz [7] str. 1416. □

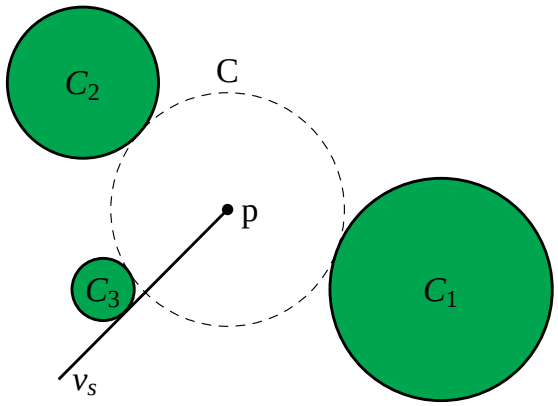
Důsledek 2.1: *Tečný vektor t v bodě v splňuje následující systém rovnic:*

$$t_i \cdot t = t_j \cdot t = t_k \cdot t, \tag{2.8}$$

kde $t \neq 0$.

Posledním parametrem, který potřebujeme pro stanovení rovnice Bézierovy křivky znát, je bod na křivce p, který může být nalezen následovně:

1. Promítneme koule b_1 b_2 a b_3, které generují hranu e, do roviny definované středy těchto koulí, tím získáme tři kružnice C_1, C_2 a C_3.
2. Stanovíme střed p kružnice C, která se dotýká kružnic C_1, C_2 a C_3 získaných v předchozím kroku (viz Obrázek 2.4).
3. Provedeme zpětnou transformaci středu kružnice C do prostoru.



Obrázek 2.4: Stanovení bodu ležícího na Voronoi hraně. C_1, C_2 a C_3 označují kružnice, které vzniknou průmětem generátorů b_1, b_2 a b_3 do roviny definované jejich středy, C je kružnice tečná ke kružnicím C_1, C_2 a C_3 a p je její střed.

Máme stanoveno pět potřebných parametrů, a tudíž můžeme reprezentovat Voronoi hranu pomocí Bézierovy křivky, předtím však musíme určit hodnoty vah w_0, w_1 a w_2.

Budeme předpokládat, že hrana bude reprezentována pomocí standardní formy racionální kvadratické Bézierovy křivky, tj. $w_0 = w_2 = 1$. Pro hodnotu w_1 pak platí následující rovnost:

$$w_1 = \frac{\tau_1}{2\sqrt{\tau_0 \tau_2}}, \tag{2.9}$$

kde τ_0, τ_1 a τ_2 jsou barycentrické souřadnice bodu p vzhledem k trojúhelníku $\triangle p_0 p_1 p_2$, přičemž bod p_1 je průsečík tečen ke křivce e, které procházejí bodem p_0 resp. p_2.

Při určování bodu p ležícího na křivce mohlo nastat, že byl určen bod na křivce orientované z koncového Voronoi vrcholu v_e do počátečního vrcholu v_s, tzn. stanovený bod neleží na Voronoi hraně, ale na jejím doplňku, tuto skutečnost odhalíme tak, že úhlová vzdálenost bodu p od počátečního vrcholu v_s je větší než úhlová vzdálenost koncového vrcholu v_e od v_s. V tomto případě použijeme váhu w_1 s opačným znaménkem.

2.5 Algoritmus sledování hran

Pomocí algoritmu sledování hran (*Edge tracing*) získáme Voronoi diagram, který bude reprezentován jako hranový graf $G = (V, E)$, kde V je množina Voronoi vrcholů a E označuje množinu Voronoi hran. Základní myšlenka tohoto algoritmu je velice jednoduchá. Algoritmus nejprve spočítá prázdnou tečnou sféru definovanou čtyřmi nejbližšími generátory, tím získáme Voronoi vrchol v_0. Po nalezení vrcholu v_0, dokážeme jednoduše určit čtyři hrany e_0, e_1, e_2 a e_3, které z tohoto vrcholu vycházejí, tzn. v_0 je jejich počáteční vrchol. Tyto hrany jsou poté dále trasovány k jejich koncovým vrcholům v pořadí, ve kterém byly vytvořeny. Nalezením nového vrcholu vzniknou další tři hrany, které z tohoto vrcholu vycházejí a budou zařazeny mezi hrany určené k trasování. Tento postup opakujeme, dokud nebude provedeno trasování všech hran k tomu určených. V dalších sekcích provedeme detailní popis tohoto algoritmus, který bude vyložen na základě článku [7].

2.5.1 Inicializace

Trasování hran je možné zahájit teprve, když nalezneme počáteční Voronoi vrchol v_0, který nám pomůže vytvořit první čtyři Voronoi hrany. To je dle uvedené literatury možné provést dvěma způsoby:

1. způsob - Brute Force

Pro každou čtveřici generujících koulí bude vypočtena tečná sféra a proveden test, zda je tato sféra prázdná či nikoliv. Pokud je tečná sféra prázdná, pak její střed představuje Voronoi vrchol v_0. Počet možných kombinací čtveřic generátorů

je $\mathcal{O}(n^4)$ a testů sféry na prázdnost je $n-4$, z toho plyne, že časová složitost nalezení počátečního vrcholu v_0 pomocí tohoto přístupu je v nejhorším případě $\mathcal{O}(n^5)$, a proto nelze tento postup doporučit.

2. způsob - Pomocí fiktivních vrcholů

V tomto postupu budeme předpokládat, že jsme do množiny generujících koulí přidali čtveřici fiktivních koulí, která je záměrně umístěna tak, aby pro ni existovala pouze jedna prázdná tečná sféra. Tím získáme Voronoi vrchol, který bude taktéž fiktivní. Z tohoto vrcholu pak spustíme trasování hran, které bude ukončeno ve chvíli, kdy získáme Voronoi vrchol, který bude definovaný pouze původní množinou generátorů. Tento vrchol pak zvolíme jako počáteční Voronoi vrchol v_0. Vrchol v_0 umožní začít konstruovat topologii Voronoi diagramu.

2.5.2 Trasování hran pomocí zásobníku

Když máme nalezen počáteční Voronoi vrchol v_0, můžeme definovat čtyři Voronoi hrany e_0, e_1, e_2 a e_3, jejichž počáteční vrchol je v_0, a přidat je do zásobníku hran. Jakmile jsou hrany uloženy do zásobníku, vyjmeme ze zásobníku jednu hranu e a budeme ji trasovat, tzn. vypočteme koncový bod v_e této hrany, čímž dokončíme její definici. Koncový vrchol v_e hrany e je definován třemi gate koulemi, které definují hranu, a jedním generátorem z množiny kandidátů, jejíž velikost je $n-3$.

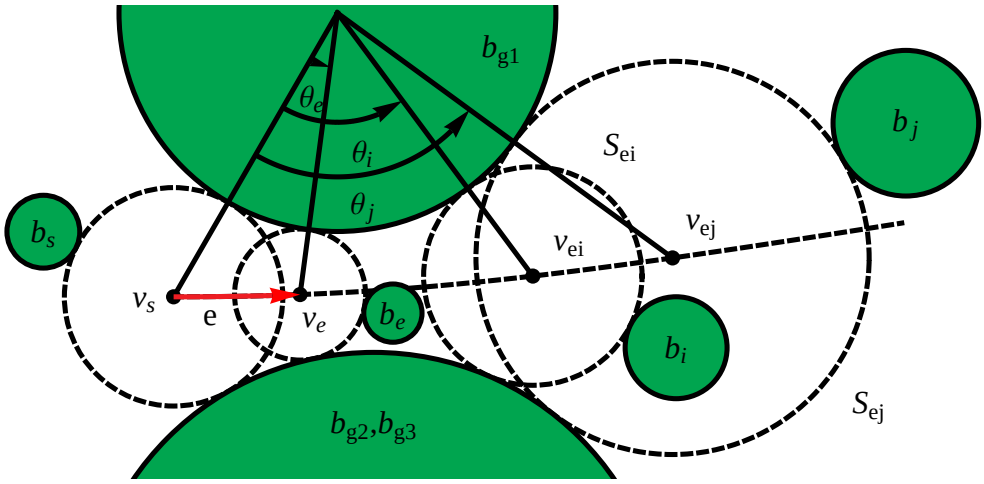
Předpokládejme, že koncový vrchol hrany vybrané ze zásobníku je Voronoi vrchol v_i, pak nám tento vrchol bude definovat další tři hrany e_{i1}, e_{i2} a e_{i3}, které z tohoto vrcholu vycházejí, a tudíž vrchol v_i je jejich počáteční vrchol. Některé z těchto hran mohou být redundantní. Také může nastat případ, že stanovený koncový vrchol je již vytvořen. Postup řešení pro tuto situaci bude uveden v Sekci 2.5.4. Nově vytvořené hrany opět uložíme do zásobníku hran.

Celý uvedený postup budeme opakovat, dokud nebude zásobník prázdný.

2.5.3 Výpočet koncového vrcholu

Předpokládáme, že máme hranu e vyjmutou ze zásobníku, přičemž je dán její počáteční Voronoi vrchol v_s, tři gate koule b_{g1}, b_{g2} a b_{g3} definující hranu e a generátor b_s, který společně s gate koulemi definuje prázdnou tečnou sféru S_s se středem ve vrcholu v_s. Naším cílem je vypočítat koncový vrchol v_e hrany e, což je ekvivalentní problému nalezení koncové koule b_e, která je jedním z generátorů koncového vrcholu v_e. Budeme tudíž určovat tečnou sféru S_{ei}, kterou generují uvedené tři gate koule a libovolná koule

b_i z množiny kandidátů $K = B \setminus \{b_{g1}, b_{g2}, b_{g3}\}$. Poté bude vybrán jiný kandidát b_j a vypočítáme tečnou kouli S_{ej}, která je definována generátorem b_j a třemi gate koulemi. Pokud generátor b_j protíná sféru S_{ei}, pak sféra S_{ei} není prázdná, a tudíž ji nahradíme sférou S_{ej}. Pokud generátor b_j sféru S_{ei} neprotíná, pak ze sfér S_{ei} a S_{ej} vybereme tu, jejíž střed má menší úhlovou vzdálenost θ vzhledem k startovnímu vrcholu v_s. Takto projdeme všechny generátory v množině kandidátů. Výsledkem bude, že dostaneme generátor b_e, který společně s gate koulemi definuje prázdnou tečnou sféru S_e, jejímž středem bude hledaný koncový Voronoi vrchol v_e. Uvedený postup odpovídá Algoritmu 2.1 a je pro názornost zachycen na Obrázku 2.5.



Obrázek 2.5: Hledání koncového Voronoi vrcholu v_e

Algoritmus 2.1: Hledání koncového vrcholu

Vstup: Startovní vrchol v_s; čtveřice generátorů b_{g1}, b_{g2}, b_{g3} a b_s; hrana e definována gate koulemi b_{g1}, b_{g2} a b_{g3}; množina kandidátů $K = B \setminus \{b_{g1}, b_{g2}, b_{g3}\}$

Výsledek: Prázdná tečná sféra S_e a generátor $b_e \in K$

1 $S_{e1} \leftarrow$ tečná sféra ke generátorům b_{g1}, b_{g2}, b_{g3} a $b_1 \in K$, pokud je více řešení, vybereme to s menší úhlovou vzdáleností
2 $S_e \leftarrow S_{e1}$
3 $b_e \leftarrow b_1$
4 **foreach** $b_i \in K \setminus \{b_1\}$ **do**
5 $S_{ei} \leftarrow$ tečná sféra ke generátorům b_{g1}, b_{g2}, b_{g3} a b_i
6 **if** $\theta_i < \theta_e$ **then**
7 $S_e \leftarrow S_{ei}$
8 $b_e \leftarrow b_i$
9 **return** (S_e, b_e)

Jelikož všechny koule v množině kandidátů K jsou procházeny pouze jednou, časová složitost nalezení koncového vrcholu je $\mathcal{O}(n)$. Časová složitost konstrukce hranového grafu reprezentujícího topologii Voronoi diagramu koulí je $\mathcal{O}(mn)$, kde m je počet Voronoi hran a n je počet Voronoi vrcholů.

2.5.4 Testování existence vrcholu

Trasováním hrany jsme vypočítali nový Voronoi vrchol. Nyní musíme otestovat, jestli nalezený vrchol nebyl vypočítán již dříve. Pokud bychom testem zjistili, že vrchol již existuje, nebudeme vytvářet nový vrchol, ale doplníme topologické informace již existujícímu vrcholu. Pro účely vyhledávání vrcholu máme definován slovník vypočtených vrcholů, který nazýváme VIDIC (*Vertex Index Dictionary*). Každý záznam umístěný do slovníku je definován pomocí čtveřice generátorů definujících Voronoi vrchol seřazených podle indexů a pointrem na definici vrcholu. Navíc musí být pro každý záznam definován jednoznačný klíč, kterým je buď orientovaná čtveřice indexů nebo pozice Voronoi vrcholu v prostoru či kombinace obou těchto možností. Pokud jako reprezentaci VIDIC použijeme hashovací tabulku, průměrná časová složitost testu na existenci vrcholu je $\mathcal{O}(1)$.

2.5.5 Algoritmus použitý v knihovně awVoronoi

Jedná se o modifikovanou verzi algoritmu sledování hran, která byla navržena a prezentována M. Maňákem. Zde se zaměříme na hlavní změny v původním algoritmu trasování hran, detailní popis modifikovaného algoritmu lze získat v [10].

Nalezení prvního Voronoi vrcholu

Tento postup vychází z toho, že nejprve najde nejbližší stěnu Voronoi buňky k zadanému generátoru ve smyslu aditivně vážené vzdálenosti. Dále najde nebližší hranu na této stěně a nakonec vyhledá nejbližší vrchol na nalezené hraně. Vzdálenost je měřena jako poloměr prázdné tečné sféry. Postup lze vyjádřit pomocí Algoritmu 2.2.

Problém nalezení MTS (minimální tečné sféry) v E^3 lze řešit jako $(n-1)$ dimenzionální problém, kde $n \leq 4$. Pro $n = 2$ řešíme problém nalezení nejbližšího souseda. Pro $n = 3$ můžeme problém převést do roviny definované středy tří generátorů a řešit jej jako problém nalezení kružnice dotýkající se třech jiných kružnic. A pro $n = 4$ řešíme problém nalezení tečné sféry dotýkající se čtveřice jiných sfér.

Očekávaná časová složitost uvedeného algoritmu je $\mathcal{O}(n)$ (pro optimalizovanou verzi $\mathcal{O}(1)$) a časová složitost v nejhorším případě je $\mathcal{O}(n^2)$. Nejhorší časovou složitost

dostaneme, pokud má většina vrcholů neohraničenou nejbližší stěnu, a nebo pokud je nalezená nejbližší hrana neohraničená či eliptická.

Algoritmus 2.2: Hledání nejbližšího vrcholu

Vstup: Množina generujících sfér B

Výsledek: Voronoi vrchol definovaný jako $(b_{g1}, b_{g2}, b_{g3}, b_{g4}, S_v)$, kde b_{g1}, b_{g2}, b_{g3} a b_{g4} jsou jeho generátory a S_v je prázdná tečná sféra vrcholu

```
// Pro některou buňku
```

for $b_{g1} \in B$ **do**

```
    // Nalezení nejbližší stěny Voronoi buňky
```

$b_{g2} \leftarrow b \in B \setminus \{b_{g1}\}$, který minimalizuje $MTS(b_{g1}, b)$;

```
    // Vyhledání nejbližší hrany na stěně
```

$b_{g3} \leftarrow b \in B \setminus \{b_{g1}, b_{g2}\}$, který minimalizuje $MTS(b_{g1}, b_{g2}, b)$;

```
    // Vyhledání nejbližšího vrcholu na hraně
```

$b_{g4} \leftarrow b \in B \setminus \{b_{g1}, b_{g2}, b_{g3}\}$, který minimalizuje $MTS(b_{g1}, b_{g2}, b_{g3}, b)$;

if b_{g1}, b_{g2}, b_{g3} *a* b_{g4} *definují vrchol* **then**

return $(b_{g1}, b_{g2}, b_{g3}, b_{g4}, S_v)$;

Žádný vrchol nebyl nalezen;

```
/* MTS(b_g1,...,b_gn) značí minimální tečnou sféru pro danou
množinu n generátorů, přičemž je minimalizován její
poloměr.                                                  */
```

Výpočet koncového vrcholu

Algoritmus 2.1 bude mít problémy, pokud narazí na neohraničenou hranu vedoucí do nekonečna. V tomto případě nesprávně stanoví koncový vrchol v_e, který bude uvedeným algoritmem umístěn na zakázanou část hrany. Řešením, jak se vyhnout tomuto problému, je rozšíření množiny kandidátů K o imaginární nekonečný generátor b_∞ a přidání následujícího testu na začátek Algoritmu 2.1:

if (b_{g1}, b_{g2}, b_{g3}) *definují eliptickou hranu* **then**
    $b_1 \leftarrow b \in K$
else
    $b_1 \leftarrow b_\infty$

Tečná sféra ke generátorům b_{g1}, b_{g2}, b_{g3} a b_∞ bude reprezentována rovinou a pro výpočet úhlové vzdálenosti θ bude používána normála této roviny.

3 GPU

GPU (grafický procesor, z anglického *Graphics Processing Unit*) je specializovaný řídící procesor umístěný na grafické kartě. V této kapitole uvedeme stručný popis architektury GPU a srovnáme ji s architekturou CPU. Dále se zaměříme na programování GPU a jazyky, které je možno k tomuto účelu použít.

3.1 Vývoj GPU

Vývoj prvních specializovaných grafických prostředků počítačů je datován do první poloviny 60. let 20. století, kdy se začaly počítače využívat pro zpracování vědeckých dat, které bylo potřeba srozumitelně zobrazit. První grafická karta pak byla uvedena na trh počátkem 80. let 20. století firmou IBM. Grafická karta v té době převážně pracovala s čarami a kružnicemi. Později přibyla možnost vykreslování obrázků ve 2D prostředí. Těchto vlastností bylo hlavně využito v herním průmyslu, který se tudíž stal hlavním iniciátorem vývoje grafických karet.

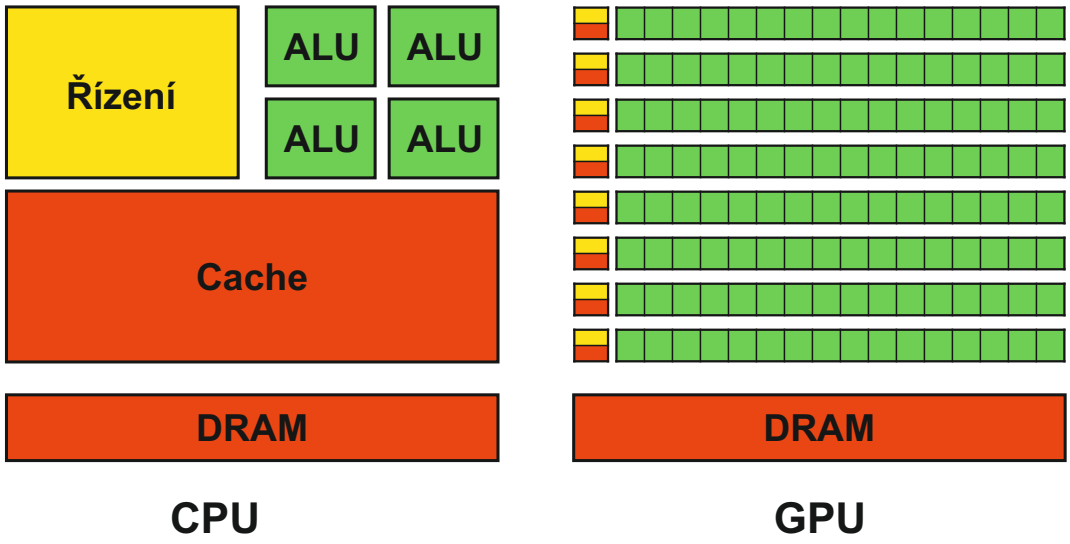
K dalšímu průlomu ve vývoji došlo v 90. letech, kdy se objevila první 3D grafika a s ní spojené grafické karty umožňující 3D zobrazení. Nicméně veškeré efekty byly pevně určeny v hardwaru grafické karty (tzn. fixní pipeline), a tudíž nebylo možné její využití pro jiné než grafické účely. Počátkem 21. století se objevily programovatelné shadery. To znamenalo, že grafická karta přestala být striktně specializovanou a začala být snaha o uplatnění jejího značného výkonu ve vědeckých odvětvích, simulacích atd.

3.2 CPU versus GPU

V prvé řadě je nutné si uvědomit, že CPU a GPU se vyvinuly každá pro jiný účel. Zatímco CPU slouží jako výpočetní jednotka, která zpracovává všechny instrukce vedoucí k požadovanému výsledku, GPU je specializovaný hardware pro renderování grafiky, z čehož plyne, že musí být schopna provést v co nejkratším čase výpočet pro co největší počet dat. Výsledkem je, že GPU je hardware, který poskytuje velké zvýšení výkonu při provádění určitých operací a má v hardwaru zabudovanou nativní podporu pro paralelismus. Naopak

CPU poskytuje uspokojivý výkon pro všechny operace, ale nemá takovou podporu pro paralelismus.

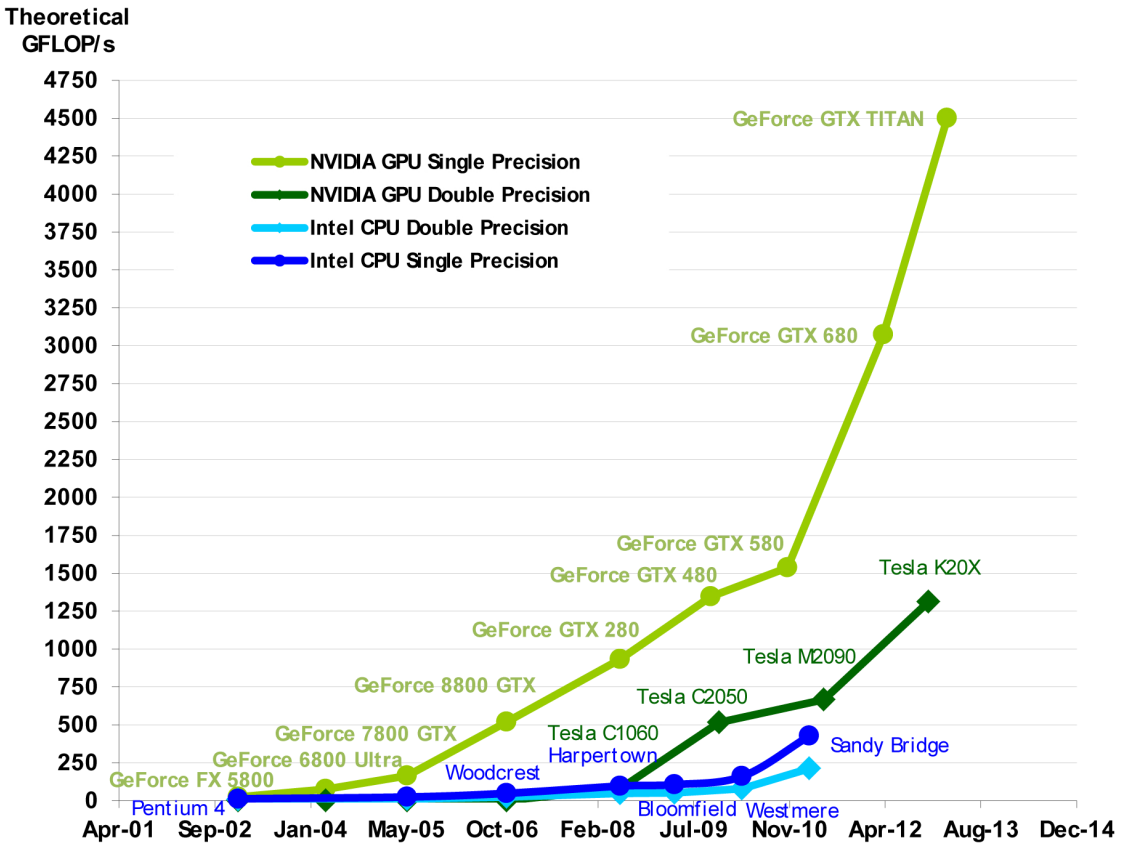
Nyní se zaměříme na hlavní rozdíly mezi GPU a CPU, viz Obrázek 3.1. Zatímco se CPU skládá jen z několika jader (v řádech jednotek až desítek), přičemž mohou zároveň běžet jedno až dvě vlákna na jednom jádře, GPU má několik stovek jader, na kterých může zároveň běžet tísíce vláken. Již z toho můžeme odvodit, že u CPU převládá kontrola a optimalizace běhu nad propustností dat, avšak u GPU je hlavní důraz kladen na propustnost dat. CPU a GPU se taktéž liší v přístupu spuštění instrukcí, jelikož pro CPU je typické spouštět instrukce mimo pořadí (*out-of-order*), ale GPU je spouští v pořadí (*in-order*). Dále se CPU a GPU liší z hlediska použitého paralelismu. Zatímco CPU využívá přístup MIMD (z anglického *Multiple Instruction Multiple Data* - více instrukcí, více dat), tj. na různých jádrech procesoru mohou běžet nezávisle na sobě různé úlohy, GPU využívá paralelismus typu SIMT (z anglického *Single Instruction Multiple Threads* - jedna instrukce, více vláken), tzn. stejná úloha je vykonávaná spoustou vláken zároveň na různých datech. Paralelismus typu SIMT je odvozen z modelu SIMD (z anglického *Single Instruction, Multiple Data* - jedna instrukce, více dat). Dalším podstatným rozdílem je množství cache paměti - CPU má k dispozici velké množství cache paměti a využívá ji k uložení často používaných dat z „pomalejších“ pamětí, GPU má pouze malou cache paměť, která je většinou určena pouze pro čtení.



Obrázek 3.1: Srovnání CPU a GPU z hlediska architektury

Celkově můžeme říci, že GPU je oproti CPU velmi výkonná jednotka, ale méně univerzální, jelikož kvůli zvýšení výkonu dochází k obětování cache paměti a řízení běhu. Pro názornost je na Obrázku 3.2 vidět porovnání maximálního počtu operací vykonaných za sekundu pro vybrané CPU a GPU. Další nevýhodou výpočtu na GPU je, že velkého

urychlení dosáhneme pouze tehdy, stačí-li nám provést výpočet pouze v jednoduché přesnosti.



Obrázek 3.2: Srovnání CPU a GPU podle počtu operací v plovoucí čárce za sekundu [13]

3.3 General Purpose GPU

General Purpose GPU (GPGPU) je technika, která umožňuje využít GPU pro řešení výpočtů, které nejsou grafického charakteru. Tato technika je založena na proudovém zpracování dat. GPGPU umožňuje využít velké množství výpočetných jader osazených na grafické kartě k efektivní paralelizaci algoritmů. V Kapitole 3.1 jsme uvedli, že grafickou kartu lze programovat pomocí shaderů. Shadery je sice možné využít i pro jiné než grafické výpočty, nicméně jejich použití pro jiné výpočty je omezené a může být velice komplikované. Z těchto důvodů bylo vyvinuto nové API umožňující využít výpočetní výkon grafických karet pro obecné výpočty způsobem podobným zpracování dat na CPU.

3.3.1 OpenCL

OpenCL (*Open Computing Language*) je otevřený standard umožňující paralelní programování heterogenních počítačových systémů a jedná se o první framework, který byl pro tyto systémy navržen. Prvotní návrh byl vytvořen společností Apple. Hlavním cílem bylo vytvořit takové rozhraní, které by dokázalo pracovat multiplatformně a využilo přitom maximální výkon a potenciál všech kompoment počítače. Později se vytvořila skupina Khronos Compute Working Group, která je v současné době zodpovědná za další vývoj OpenCL. První verze vyšla v prosinci 2008. Výhodou OpenCL je, že je podporováno téměř všemi firmami v oblasti počítačového hardwaru, a tudíž může běžet na kterémkoli GPU nebo provést fallback k CPU (tj. pokud není podporována GPU, pak je výpočet proveden na CPU). Tím se OpenCL stává široce rozšířeným systémem. OpenCL je založené na jazyku C konkrétně na jeho standardu C99.

3.3.2 Cuda

CUDA (*Compute Unified Device Architecture*) je univerzální architektura vyvíjená firmou NVidia určená pro paralelní výpočty, byla představena v roce 2006. Tato architektura podporuje heterogenní výpočty, při kterých mezi sebou spolupracují CPU a GPU. CUDA je výpočetní engine, který je snadno přístupný přes běžně používané a známé programovací standardy. Pro programování přímo pro GPU lze využít jazyk „C for CUDA“, který je založen na jazyce C. Nevýhodou CUDA je, že je hardwarově omezená, tzn. může běžet pouze na grafických kartách od firmy NVidia, přičemž musí být osazené GPU ze série G8x a novějších. Karty s GPU ze série G8x byly uvedeny na trh od roku 2006 a od roku 2007 měly přidanou podporu pro CUDA.

3.3.3 DirectCompute

Výpočetní shader je programovatelný shader, který umožňuje rozšířit použití rozhraní Microsoft DirectX 11 o možnost počítání obecných algoritmů. Technologie výpočetního shaderu je taktéž známá jako DirectCompute. Celý tento shader je založen na HLSL (*High Level Shading Language*). Nevýhodou tohoto API je, že běží pouze na Microsoft Windows od verze Vista. Další nevýhodou je, že DirectCompute funguje pouze na grafických kartách podporujících DirectX 10 a novější.

4 OpenCL

V předchozí kapitole jsme si uvedli stručný popis architektury GPU a provedli její srovnání s CPU architekturou. V této kapitole se důkladněji zaměříme na GPGPU za použití OpenCL. Otevřený standard OpenCL jsme zvolili proto, že umožňuje provádět obecné výpočty na většině GPU a dokonce i provést fallback k CPU. Abychom mohli OpenCL použít pro paralelizaci našeho algoritmu, je nutné nejprve pochopit jeho základní architekturu. Zdroji informací jsou specifikace OpenCL [6] a další zdroje [2] a [12].

4.1 Koncepce OpenCL

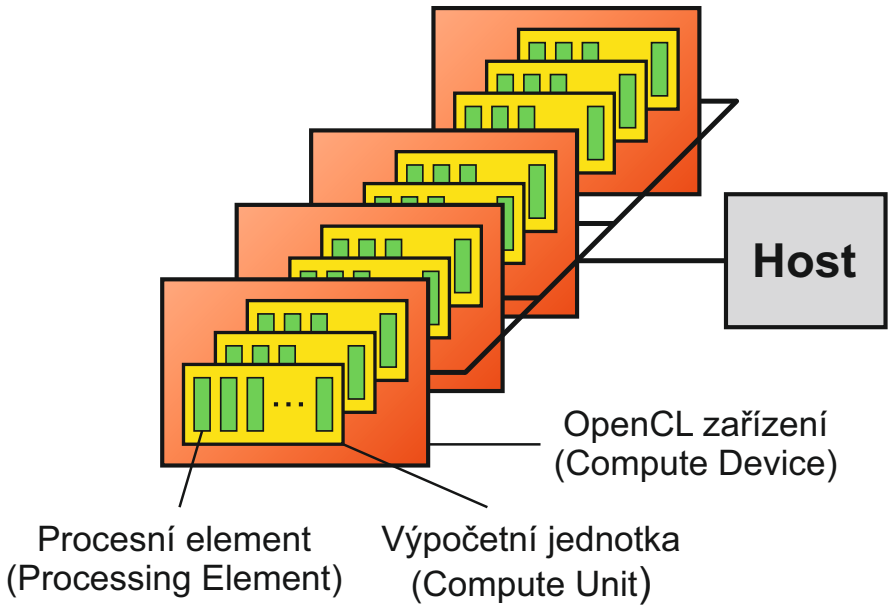
OpenCL lze použít v celé řadě aplikací. Provést zobecnění pro tyto aplikace by bylo obtížné, nicméně pro použití heterogenní platformy musíme vždy zachovat následující postup:

1. rozpoznání komponent tvořících heterogenní systém,
2. analýza architektury těchto komponent, aby mohl být software přizpůsoben specifickým vlastnostem daného hardwaru,
3. vytvoření bloků instrukcí (kernelů), které poběží na platformě,
4. nastavení paměťových objektů zapojených do výpočtu,
5. spuštění kernelů ve správném pořadí a na správných komponentách systému,
6. shromáždění konečných výsledků.

Tento postup je prováděn pomocí rozhraní uvnitř OpenCL a programovacího prostředí pro kernely. Celý problém můžeme rozdělit na 4 modely (model platformy, vykonávací model, paměťový model a programovací model).

4.2 Model platformy

Model platformy definuje vysokoúrovňovou reprezentaci jakékoli heterogenní platformy používané v OpenCL. Tento model je znázorněn na Obrázku 4.1. Platforma se skládá z hostitelského systému, který je připojen na jedno či více OpenCL zařízení (compute



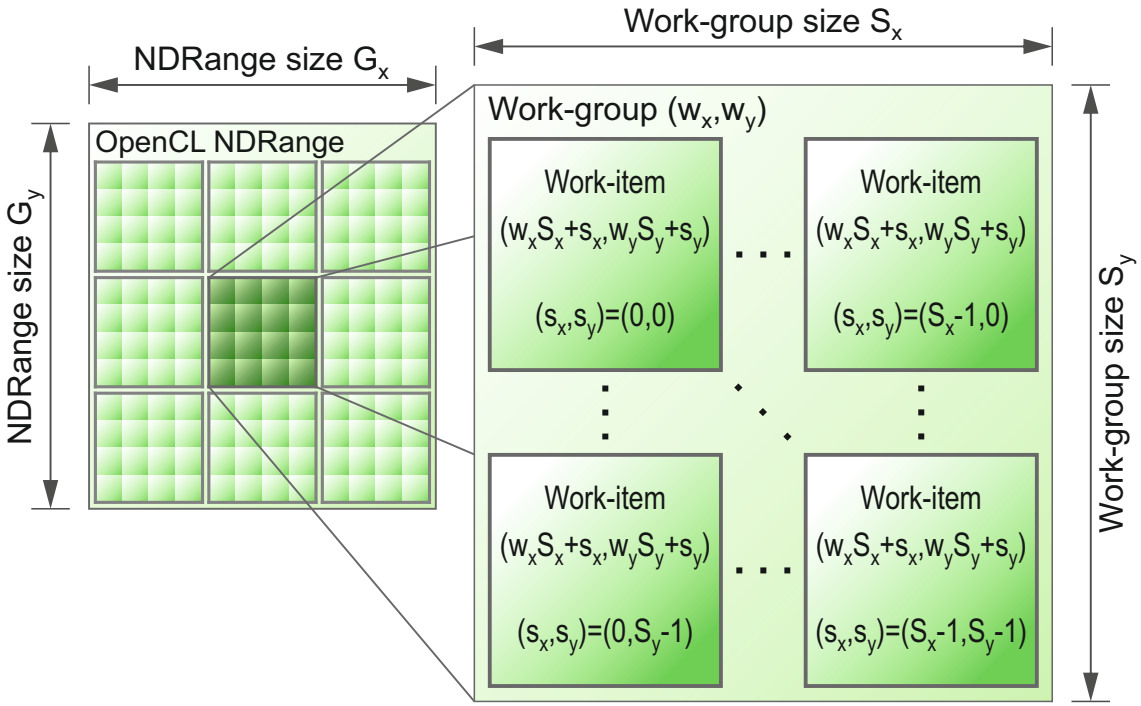
Obrázek 4.1: OpenCL model platformy

device), na kterém se spouští kernely. Hostitelský systém externě komunikuje s OpenCL programem pomocí I/O nebo s uživatelem. Každé OpenCL zařízení je dále rozděleno do několika výpočetních jednotek (compute unit), přičemž každá výpočetní jednotka obsahuje alespoň jeden procesní element (processing element). OpenCL zařízení může představovat např. GPU, vícejádrové CPU, DSP nebo další procesory podporující OpenCL.

4.3 Vykonávací model

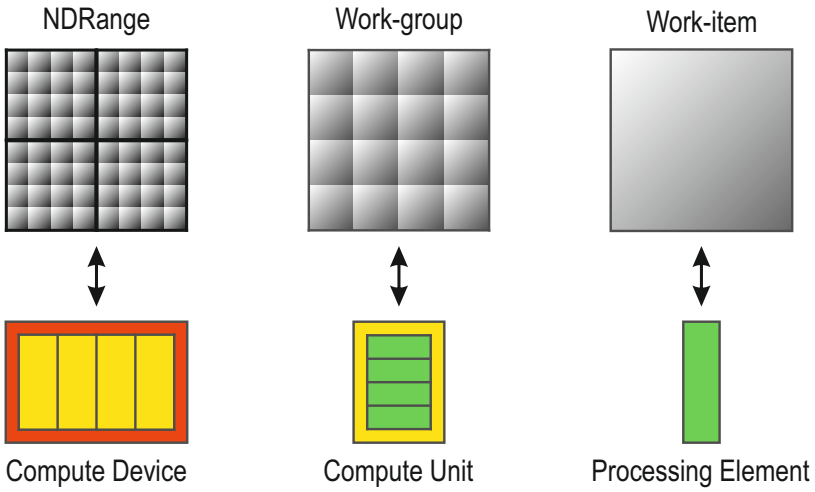
OpenCL program je rozdělen do dvou různých částí. První částí je hostitelský program běžící na hostitelském systému (dále jen aplikace). Druhou částí jsou kernely, což jsou programy vykonávané na OpenCL zařízení. Pomocí aplikace ovládáme jednotlivá OpenCL zařízení. Aplikace si na počátku musí zjistit informace o platformě a zařízeních, tyto infomace dále využije k vytvoření kontextu, což je prostředí, ze kterého se spouští jednotlivé kernely. Při spuštění kernelu dojde k definici indexového prostoru nazývaného NDRange. Indexový prostor je N-dimenzionální, přičemž N může nabývat hodnot 1, 2, a 3. Každý index v indexovém prostoru představuje jednu instanci kernelu nazývanou work-item (jedná se o jedno výpočetní vlákno, ve kterém probíhá kód kernelu). Každá work-item je jednoznačně definována pomocí *global ID*. Všechny work-items vykonávají stejný kód, ale na různých datech. Velikost a dimenze indexového prostoru je definována při spuštění kernelu.

Work-items jsou dále uspořádány do pracovních skupin nazývaných work-groups, ve kterých mohou vzájemně spolupracovat. Velikost work-group se definuje při spuštění kernelu a její dimenze odpovídá dimenzi NDRange. Vytvořením work-groups dojde k rozdělení celého indexového prostoru na menší části, ve kterých mohou být sdíleny prostředky, jako je například paměť. Navíc v rámci work-group je možné provést synchronizaci work-items. Také každá work-group je jednoznačně definována pomocí ID, které se nazývá *group ID*. V rámci skupiny pak mají work-items definované unikátní *local ID*. Identifikace work-items vzhledem k celému prostoru je pak možná buď pomocí *global ID* nebo pomocí součtu identifikátoru skupiny (*group ID*) a lokálního identifikátoru (*local ID*). Názorný příklad dvojrozměrného indexového prostoru je znázorněn na Obrázku 4.2.



Obrázek 4.2: Příklad dvojrozměrného NDRange zobrazující rozmístění work-items a work-groups (G značí globální velikost, S lokální velikost a w označuje group ID). Převzato z [6].

Na Obrázku 4.3 je názorně vidět, jak je softwarové provedení OpenCL namapované na hardware. Kód vykonávaný work-item je spouštěn na procesním elementu (processing element). Work-items se slučují, jak již bylo řečeno, do work-group a ta je zpracovávána na výpočetní jednotce (compute unit), do které se slučují procesní elementy. Kernel je spouštěn nad indexovým prostorem NDRange a pouze jeden kernel může být současně vykonáván na OpenCL zařízení (Compute Device).



Obrázek 4.3: Schéma přidělování práce mezi výpočetní prostředky

4.3.1 Kontext

Aplikace spuštěná na hostitelském systému definuje kontext, jenž představuje prostředí, v němž jsou k dispozici prostředky, kterými disponuje OpenCL zařízení. Prostřednictvím kontextu tak máme přístup k následujícím objektům OpenCL:

Zařízení (*Devices*)

Množina OpenCL zařízení použitých hostitelským systémem k vykonání našeho kódu.

Programy

Zdrojové kódy a spustitelné soubory, které implementují kernely, které se budou vykonávat. Vytvoření programu přímo ze zdrojových kódů provedeme příkazem `clCreateProgramWithSource`. Zkompilování kódu pak obstará příkaz `clBuildProgram`.

Kernely

Funkce OpenCL, které běží na jednotlivých OpenCL zařízení. Získáme je po vytvoření zkompilovaného programu pomocí příkazu:
`clCreateKernel(program, "JmenoKernelu", &chyba).`

Paměťové objekty (*Memory objects*)

Jedná se o množinu paměťových objektů viditelných pro OpenCL zařízení, které obsahují proměnné, s nimiž mohou pracovat jednotlivé kernely.

4.3.2 Příkazová fronta

Aplikace dále vytváří příkazovou frontu (*command-queue*), která zajišťuje vzájemnou komunikaci mezi hostitelským systémem a OpenCL zařízeními pomocí příkazů napsaných

v hostitelském systému. Tyto příkazy poté čekají v příkazové frontě, dokud se nespustí na OpenCL zařízení. Celkem existují tři typy příkazů:

Příkazy jádra (*Kernel execution commands*)
Příkazy, které spouští kernely, jenž mají být vykonány.

Paměťové příkazy (*Memory commands*)
Tyto příkazy zajišťují přenos dat mezi hostitelským systémem a OpenCL zařízeními. Jejich úkolem je přesun dat mezi paměťovými objekty nebo mapování paměťových objektů na zařízení.

Příkazy pro synchronizaci (*Synchronization commands*)
Synchronizují spouštěné příkazy a zajišťují tak správné pořadí vykonání.

Příkazy zařazené ve frontě jsou vykonány asynchronně, tzn. aplikace zařadí příkazy do fronty a pokračuje dál, přičemž kernely budou vykonány nezávisle na aplikaci. Pokud potřebujeme mít zajištěno, že kernel byl na zařízení dokončen před pokračováním aplikace, musíme toho dosáhnout buď synchronizací nebo nastavením blokování.

Příkazy zařazené ve frontě mohou být vykonány dvěma různými způsoby:

V pořadí (*In-order execution*)
Příkazy jsou do příkazové fronty přidávány v určitém pořadí a ve stejném pořadí jsou i vykonány. V tomto případě synchronizaci zajišťuje samotná fronta.

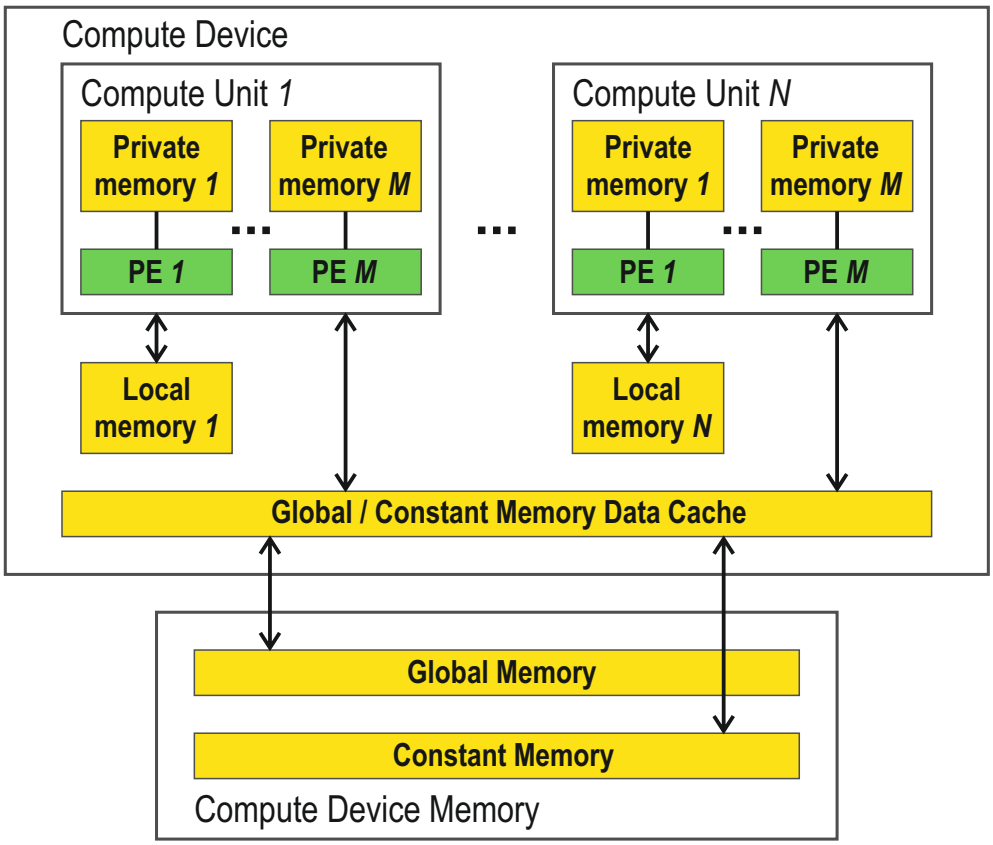
Mimo pořadí (*Out-of-order execution*)
Příkazy jsou do fronty zadávány v určitém pořadí, ale nemusí být v tomto pořadí i vykonány. Synchronizaci musí v tomto případě zajistit programátor pomocí synchronizačních příkazů.

4.4 Paměťový model

Paměťový model definuje jednotlivé paměti, vzájemné vztahy a komunikaci mezi nimi. Paměť je v OpenCL dělena na dvě části:

Paměť hostitelského systému
Tato část paměti je viditelná a dostupná pouze z hostitelského systému. OpenCL pouze definuje, jak bude paměť hostitelského systému komunikovat s OpenCL objekty a konstrukcemi.

Paměť OpenCL zařízení
Jedná se o část paměti, která je k dispozici kernelům vykonávaným na zařízení.



Obrázek 4.4: Paměťový model v OpenCL (PE = procesní element). Převzato z [12].

Paměť zařízení je dále rozdělena do čtyř typů paměti. Tyto paměti se liší v rychlosti přístupu a dostupnosti. Paměťový model OpenCL zařízení je zobrazen na Obrázku 4.4. Nyní si popíšeme jednotlivé typy:

Globální paměť (global memory)

Tento typ paměti umožňuje přístup pro všechny work-items ze všech work-groups bežících na OpenCL zařízení, které je zahrnuto v kontextu, pro čtení i zápis. Pokud to zařízení umožňuje, může být přístup nahrán do vyrovnávací paměti. Nevýhodou globální paměti je, že se jedná o nejpomalejší paměť a její výkon závisí na způsobu používání. Z tohoto důvodu se snažíme o omezení konkurenčního přístupu ke stejnému místu v paměti z několika různých work-items, které by mohly být vykonávány současně. Klíčové slovo pro její označení je `__global`.

Konstantní paměť (constant memory)

Jedná se o oblast globální paměti, která zůstává neměnná během vykonávání spuštěného kernelu. Work-items z ní mají právo pouze číst. Hardware má tudíž možnost optimalizovat konkurenční přístup k této paměti. Hostitelský systém

alokuje a inicializuje paměťové objekty, které jsou umístěny do konstantní paměti zařízení.

Lokální paměť (local memory)

Tato paměť je omezena pouze pro work-group, tzn. každá work-group má přidělenou svojí vlastní lokální paměť. Každá work-item v rámci dané work-group má právo pro čtení i zápis. Lokální paměť je rychlejší než globální. Klíčové slovo pro její označení je `__local`.

Privátní paměť (private memory)

Tato pamět je přístupná pouze pro konkrétní work-item a není přístupná pro ostatní work-items. Jedná se o nejrychlejší paměť. Její velikost není definována žádným standardem, ale závisí pouze na konkrétním hardwaru. Pokud nároky na privátní paměť překročí limit hardwaru, budou data uložena do globální paměti, což způsobí snížení výkonu.

Paměť hostitelského systému a paměť OpenCL zařízení jsou většinou na sobě nezávislé, proto musí být správou paměti umožněno sdílení dat mezi hostitelským systémem a výpočetním zařízením, tzn. data musí být explicitně přemístěna z paměti hostitelského systému do paměti zařízení a zpět. To je provedeno zařazením příkazu pro čtení/zápis do příkazové fronty. Tyto příkazy mohou být buď blokující nebo neblokující. Blokující přístup se liší tím, že čeká až bude příkaz vykonán, a teprve poté pokračuje dál v běhu programu. Při použití neblokujícího přenosu dat není žádná záruka, že data jsou validní (přenos nemusí být kompletní).

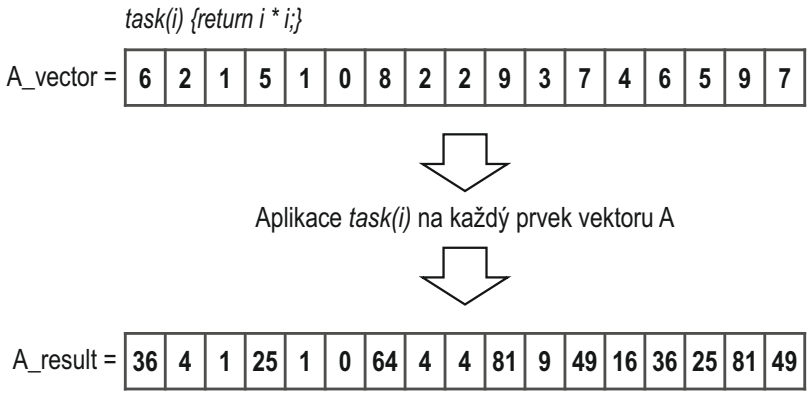
4.5 Programovací model

Každý programátor může naimplementovat stejný algoritmus jiným způsobem. Z tohoto důvodu musí být programovací model flexibilnější než hardwarově orientovaný přesně definovaný vykonávací model. OpenCL má definovány dva různé programovací modely, a to Datově paralelní programovací model (*Data-Parallel Programming Model*) a Úlohově paralelní programovací model (*Task-Parallel Programming Model*). Může být použita i jejich kombinace.

4.5.1 Datově paralelní programovací model

Datově paralelní programovací model se zaměřuje na datové struktury, jejichž složky mohou být měněny současně. V tomto modelu je tudíž výpočet definován jako sekvence

instrukcí, která je aplikována na všechny složky datové struktury. Indexový prostor vykonávacího modelu, jehož součástí jsou work-items, jednoznačně definuje, jak jsou work-items namapovány na složky datové struktury. V ideálním případě připadá jeden work-item na jednu složku dat, v OpenCL to však nemusí vždy platit. Jednoduchý příklad datového paralelismu je znánorněn na Obrázku 4.5.



Obrázek 4.5: Jednoduchý příklad datově paralelního programovacího modelu

4.5.2 Úlohově paralelní programovací model

Úlohově paralelní programovací model rozloží řešený problém na úlohy, které mohou běžet zároveň. Aby mohly být tyto úlohy vykonány, dojde k jejich namapování na procesní elementy. Jelikož se úlohy z hlediska výpočetních požadavků značně liší, je obtížné zajistit, aby všechny dokončily výpočet ve stejnou chvíli. OpenCL považuje za úlohu kernel, který se spustí jako jeden work-item.

5 Návrh algoritmu pro GPU

Tato kapitola se zabývá úpravou algoritmu sledování Voronoi hran tak, aby bylo docíleno urychlení konstrukce Voronoi diagramu, přičemž k tomu bude využit výpočet na GPU. Jak bylo uvedeno v Sekci 3.2, GPU má zabudovanou nativní podporu pro paralelizaci, a urychlení algoritmu pomocí GPU bude směřovat touto cestou. Aby bylo možné využít GPU výpočet efektivně a bylo skutečně dosaženo urychlení, musí být nejprve identifikovány výpočetně nejnáročnější části algoritmu a rozhodnuto o tom, zda je vhodné tyto části počítat pomocí GPU.

V této kapitole se zabýváme teoretickým principem řešení problému, přičemž pro lepší představivost je řešení uvedeno s využitím úhlové vzdálenosti. Během implementace algoritmu však nepočítáme přímo úhlovou vzdálenost, ale pro porovnávání využíváme normalizovaných směrových vektorů ramen úhlů.

5.1 Paralelizovatelné části

Modifikace algoritmu trasování hran, která umožňuje provedení části výpočtu na GPU, což vede k urychlení konstrukce Voronoi diagramu, vychází z formulace základní verze algoritmu, která je uvedena v Sekci 2.5.2. Výpočet koncového vrcholu pak probíhá pomocí postupu uvedeného v Sekci 2.5.3, přičemž je zahrnuta úprava popsaná v Sekci 2.5.5.

Nejdříve však bylo nutné stanovit, které části algoritmu jsou nejnáročnější, a rozhodnout, zda jsou paralelizovatelné na GPU. Výpočetně náročné je bezpochyby hledání koncových vrcholů jednotlivých hran uložených v zásobníku, dokud není zásobník prázdný. Pokud by byla paralelizace vykonávaná na CPU pomocí vláken, pak je paralelní běh celé této části algoritmu možný a v knihovně awVoronoi je již implementován. Tato diplomová práce si však klade za cíl provést paralelizaci pomocí výpočtu na GPU a ta by byla pro celý tento úsek algoritmu nemožná. Důvody, proč není vhodné či možné realizovat výpočet celé této části na GPU, jsou následující:

- Při paralelizaci uvedeného úseku algoritmu by bylo zapotřebí sdílet mezi hostitelským systémem a OpenCL zařízením různé struktury, které reprezentují hranový graf a jeho komponenty. Dále by musel být sdílen také zásobník, do nějž se uklá-

dají hrany, které mají být trasovány, protože by musel být alokován v globální paměti, aby k němu měly přístup všechny work-items. Sdílení dat mezi hostitelem a OpenCL zařízením však z hlediska datových typů poskytuje omezené možnosti. Přímo je umožněno sdílet pouze primitivní datové typy, buffery a obrázky. Různé datové struktury by bylo možné sdílet pouze pomocí byte bufferu, ale nebylo by zaručeno, že program zůstane multiplatformní, a proto je doporučeno nesdílet datové struktury. Toto omezení by způsobovalo pro uvedenou paralelizaci problém hlavně kvůli potřebě sdílení zásobníku.

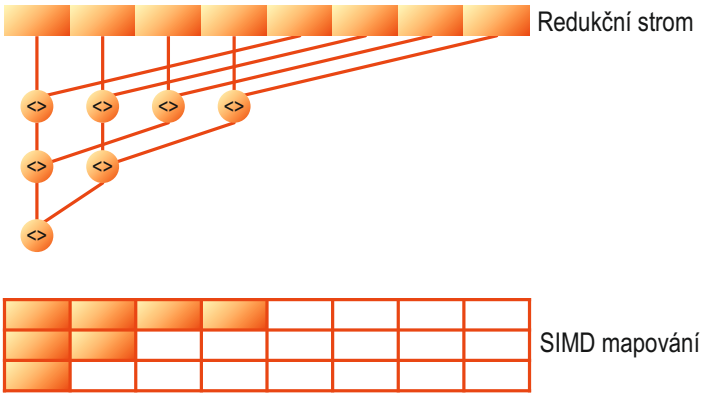
- Další příčina, která znemožní paralelizaci uvedeného úseku algoritmu, je, že velikost sdílené paměti mezi hostitelským systémem a OpenCL zařízením by měla být známa již v době překladu. GPU výpočty také neumožňují dynamickou alokaci paměti.
- Při provádění GPU výpočtu musí být předem známa velikost řešeného problému, což nejsme schopni zajistit, protože předem nevíme, jaké množství Voronoi hran bude umístěno do zásobníku k trasování. Navíc by bylo velmi problematické zajistit synchronizaci při práci se zásobníkem a nemožné uspat work-item, dokud do zásobníku nebudou vloženy nová data.

V předchozím odstavci bylo vysvětleno, že není možné paralelizovat na GPU celé sestavování hranového grafu, nicméně to nevylučuje možnost počítat na GPU některý z dílčích úkolů, který je výpočetně náročný. Tímto úkolem je stanovení čtvrtého generátoru definujícího koncový Voronoi vrchol hrany, kdy jsou procházeny všechny generátory umístěné v množině kandidátů. V tomto případě není potřeba sdílet žádná data, která se nedají reprezentovat pomocí primitivních datových typů, je předem známá velikost řešeného problému a není problém se stanovením velikosti sdílené paměti předem.

Při hledání koncového vrcholu dochází k zredukování množiny generátorů pomocí porovnávání úhlových vzdáleností na jeden generátor definující tento koncový vrchol. V GPU výpočtu se pro tyto účely využívá tzv. *redukční operace*, jejíž průběh je znázorněn na Obrázku 5.1, při níž je potřeba provádět synchronizaci dat pomocí bariéry po každé iteraci.

5.2 Problém s numerickou přesností a stabilitou

Knihovna awVoronoi, která konstruuje aditivně vážený Voronoi diagram, je primárně využívaná pro analýzu biomolekul, přičemž biomolekuly mají takovou strukturu dat, že je potřeba pro zajištění správnosti konstrukce Voronoi diagramu využívat výpočet v pohyblivé řádové čárce s dvojitou přesností. Použití dvojité přesnosti je velice důležité hlavně



Obrázek 5.1: Redukční strom a SIMD mapování

při výpočtu úhlové vzdálenosti pro účely porovnávání. Dvojitá přesnost je taktéž důležitá v některých výpočtech prázdné tečné sféry, aby byla zachována numerická stabilita řešení soustavy rovnic (2.5). Jelikož implementace navržené úpravy algoritmu sledování hran, která je urychlena pomocí výpočtu prováděného na GPU, je součástí knihovny awVoronoi, pak pro zajištění její správné funkčnosti musí být použita dvojitá přesnost hodnot v plovoucí řádové čárce.

Na druhou stranu při použití dvojité přesnosti ve výpočtech prováděných na GPU není zcela využito potenciálu GPU. Tato kontraproduktivnost je způsobena tím, že pouze relativně malé množství jader osazených na GPU umí pracovat s čísly v pohyblivé řádové čárce s dvojitou přesností. Ostatní jádra umožňují výpočet pouze v jednoduché přesnosti. Vysvětlení tohoto faktu je jednoduché, primární účel grafické karty byl vždy zaměřen na renderování grafiky, pro které je jednoduchá přesnost dostačující.

Z uvedeného plyne, že konstrukce diagramu musí být provedena tak, aby byla zaprvé zajištěna správnost řešení a za druhé byla při výpočtu na GPU co nejméně využívaná dvojitá přesnost. Tato podmínka vedla k tomu, že byl využit dvoufázový výpočet koncového vrcholu Voronoi hrany, konkrétně dvoufázové stanovení generátoru koncového vrcholu. Tyto fáze mají následující funkčnost:

1. fáze

Abychom našli koncový vrchol v_e Voronoi hrany e, musíme pro všechny kandidáty $b_i \in K$ spočítat v jednoduché přesnosti úhlovou vzdálenost θ_i středu prázdné tečné sféry S_{ei}, která se dotýká kandidáta b_i a třech generátorů b_{g1}, b_{g2} a b_{g3} definujících hranu e, vzhledem ke startovnímu vrcholu hrany v_s. Poté po dvojicích porovnáváme vypočtené úhlové vzdálenosti (přesný postup porovnávání bude popsán později). Jako generátor koncového vrcholu vybereme z porovnávané dvojice toho kandidáta, pro nějž je úhlová vzdálenost menší. Tzn. pokud

například porovnáváme vzdálenosti θ_j a θ_k a platí $\theta_j < \theta_k$, pak jako generátor koncového vrcholu vybereme kandidáta b_j. Takto pokračujeme v porovnávání, dokud nezískáme jediný generátor b_{ef}, jehož úhlová vzdálenost je minimální, a tudíž s největší pravděpodobností definuje koncový vrchol hrany. Když máme nalezeného kandidáta b_{ef} s minimální úhlovou vzdáleností θ_{ef}, projdeme ještě jednou všechny spočítané úhlové vzdálenosti a pro každou z nich testujeme, zda absolutní hodnota rozdílu této vzdálenosti a minimální úhlové vzdálenosti θ_{ef} je menší než definovaný práh. Je-li podmínka splněna, dojde k zařazení kandidáta odpovídajícího dané úhlové vzdálenosti do množiny podezřelých generátorů S. Tzn. pokud například testujeme vzdálenost θ_j a platí $|\theta_j - \theta_{ef}| < thres$, pak je kandidát b_j zařazen do množiny podezřelých generátorů S. Po skončení této fáze tudíž můžeme získat neprázdnou množinu podezřelých generátorů, na kterou je potřeba aplikovat druhou fázi výpočtu.

2. fáze

V druhé fázi výpočtu koncového vrcholu v_e hrany e počítáme pro všechny generátory b_i z množiny podezřelých kadidátů S úhlovou vzdálenost θ_i středu prázdné tečné sféry S_{ei}, která se dotýká generátoru b_i a třech generátorů b_{g1}, b_{g2} a b_{g3} definujících hranu e, vzhledem ke startovnímu vrcholu hrany v_s ve dvojité přesnosti. Poté opět po dvojicích porovnáváme vypočtené úhlové vzdálenosti a jako generátor koncového vrcholu vybereme z porovnávané dvojice kandidáta, pro nějž je úhlová vzdálenost menší. Takto pokračujeme v porovnávání dokud nezískáme jediný generátor b_{ed}, pro nějž je úhlová vzdálenost minimální.

Tímto dvoufázovým průchodem získáme maximálně dva generátory (z každé fáze jeden), které mohou definovat koncový vrchol. Ty již mezi sebou porovnáme v hostitelském systému, který je definován ve dvojité přesnosti.

Poznamenejme, že v pvní fázi se do výpočtu mohla zapojit většina jader osazených na GPU, čímž bylo plně využito jeho potenciálu. V druhé fázi pak bylo využito pouze malé množství jader, které umožňuje výpočet ve dvojité přesnosti, nicméně tomu odpovídala i velikost problému, které byla v první fázi značně zredukována.

5.3 Shrnutí algoritmu

V této sekci pro přehlednost shrneme celý algoritmus trasování hran pomocí GPU. Tzn. popíšeme zde algoritmus probíhající na hostitelském systému i jednotlivé kernely spouštěné na GPU.

5.3.1 Upravený algoritmus sledování hran

Trasování Voronoi hran probíhá z principiálního hlediska stejným způsobem, jako probíhá v algoritmu publikovaném Kimem a kol., který je uveden v Sekci 2.5.2. Avšak jednotlivé ucelené části algoritmu probíhají podle odlišného postupu: Nalezení prvního Voronoi vrcholu, který umožňuje začít trasovat hrany, je provedeno podle Algoritmu 2.2. Výpočet koncových vrcholů jednotlivých hran pak probíhá na GPU pomocí dvoufázového průchodu, jehož myšlenka byla nastíněna v předchozí Sekci 5.2 a jehož algoritmus je uveden v Sekci 5.3.2. Jelikož je používán výpočet prováděný na GPU, nesmí být opomenuto rozšířit původní postup trasování hran o připojení hostitelského systému k OpenCL zařízení. Pro přehlednost je trasování hran popsáno Algoritmem 5.1.

Algoritmus 5.1: Algoritmus sledování hran na hostitelském systému

Vstup: Množina generujících koulí B

Výsledek: Hranový graf $G = (V, E)$

1 inicializace OpenCL zařízení - připojení hostitelského systému
2 $v_0 \leftarrow$ vyhledání počátečního vrcholu
3 Vložení čtyř Voronoi hran e_0, e_1, e_2 a e_3, jejichž počáteční vrchol je v_0, do zásobníku `stack`
4 **while** `stack` *není prázdný* **do**
5 $e \leftarrow$`stack`.pop()
6 Výpočet koncového Voronoi vrcholu v_i hrany e pomocí prázdné tečné sféry, viz Algorimus 5.2. Pokud je vypočítaný vrchol již definován, použijeme postup uvedený v Sekci 2.5.4.
7 Nově vypočítaný vrchol definuje další tři hrany e_{i1}, e_{i2} a e_{i3}, které z něj vycházejí $\Rightarrow$ Vložíme tyto tři hrany, jejichž počáteční vrchol je v_i, do zásobníku.
8 **return** (V, E)

5.3.2 Nalezení generátoru koncového vrcholu

Výpočet koncového Voronoi vrcholu trasované Voronoi hrany probíhá na GPU. Výpočet provádíme ve dvou fázích, jelikož se jednalo o nejlepší kompromis mezi výpočtem ve dvojité přesnosti, která je v některých konfiguracích nutná pro správnost Voronoi diagramu, a využítím potenciálu grafického hardwaru. První fáze je provedena pro výpočet každého koncového vrcholu a je použita jednoduchá přesnost čísel v pohyblivé řádové čárce. Druhá fáze je provedena teprve tehdy, je-li množina podezřelých kandidátů,

která je generována v první fázi, neprázdná, a je použita dvojitá přesnost čísel v pohyblivé řádové čárce. V druhé fázi tudíž již není plně využito potenciálu grafického hardwaru.

Algoritmus 5.2: Výpočet koncového vrcholu na hostitelském systému

Vstup: Startovní vrchol v_s; čtveřice generátorů b_{g1}, b_{g2}, b_{g3} a b_s; hrana e definována gate koulemi b_{g1}, b_{g2} a b_{g3}; množina kandidátů $K = B \setminus \{b_{g1}, b_{g2}, b_{g3}\}$; imaginární nekonečný generátor b_∞

Výsledek: Prázdná tečná sféra S_e a generátor $b_e \in K$

1 $S_{e\infty} \leftarrow$ normála tečné roviny ke třem generátorům b_{g1}, b_{g2}, b_{g3} a b_∞
2 $S_{es} \leftarrow$ tečná sféra ke generátorům b_{g1}, b_{g2}, b_{g3} a $b_s \in K$, pokud je více řešení vybereme to s menší úhlovou vzdáleností
3 1. fáze výpočtu na GPU - Stanovení čtvrtého generátoru b_{ef} koncového vrcholu hrany e v jednoduché přesnosti, viz Algoritmus 5.4. Inicializační kandidát $b_{in} = b_\infty$. Zároveň získáme množinu podezřelých kandidátů S, viz Algoritmus 5.3.
4 $S_{ef} \leftarrow$ tečná sféra ke generátorům b_{g1}, b_{g2}, b_{g3} a $b_{ef} \in K$, pokud je více řešení vybereme to s menší úhlovou vzdáleností
5 **if** *S není prázdná množina* **then**
6 2. fáze výpočtu na GPU - Stanovení čtvrtého generátoru b_{ed} koncového vrcholu hrany e ve dvojité přesnosti, viz Algoritmus 5.5. Inicializační kandidát $b_{in} = b_\infty$.
7 $S_{ed} \leftarrow$ tečná sféra ke generátorům b_{g1}, b_{g2}, b_{g3} a $b_{ed} \in K$, pokud je více řešení vybereme to s menší úhlovou vzdáleností
8 $S_e \leftarrow S_{e\infty}$
9 $b_e \leftarrow b_\infty$
10 **foreach** $S_i \in \{S_{es}, S_{ef}, S_{ed}\}$ **do**
11 **if** $\theta_i < \theta_e$ **then**
12 $S_e \leftarrow S_i$
13 $b_e \leftarrow b_i$
14 **return** (S_e, b_e)

Při výpočtu koncového vrcholu, je na hostitelském systému nejprve stanovena tečná sféra dotýkající se generátorů b_{g1}, b_{g2} a b_{g3} definujících hranu e a imaginárního nekonečného generátoru b_∞, který způsobí, že tečná sféra degeneruje na rovinu. Imaginární nekonečný generátor a jemu odpovídající úhlová vzdálenost vzhledem k startovnímu vrcholu v_s jsou zároveň inicializačními hodnotami pro výpočet na GPU. Dále je z hostitelského systému taktéž proveden výpočet tečné sféry pro generátory, které definují

počáteční vrchol v_s, tato konfigurace musí být znovu počítána z toho důvodu, že čtveřice generátorů může definovat dva Voronoi vrcholy, což bylo uvedeno v Sekci 2.1.1. Poté hostitelský systém nasdílí data OpenCL zařízením a počká na výsledky výpočtů prováděných na GPU. Dále vypočte tečnou sféru ke generátorům hrany e a generátoru b_{ef} získanému z první fáze. Pokud navíc první fáze stanovila podezřelé generátory, je proveden výpočet tečné sféry i pro výsledek získaný během druhé fáze. Celkově má tedy hostitelský systém vypočtené 3-4 tečné sféry, které jsou kandidáty na koncový vrchol. Z nich vybere sféru s minimální úhlovou vzdáleností vzhledem k počátečnímu vrcholu v_s, jejíž střed je koncovým vrcholem Voronoi hrany. Celý postup je popsán Algoritmem 5.2.

1. fáze

Postup této fáze je podrobně popsán v Sekci 5.2. Hledání kandidáta s minimální úhlovou vzdáleností probíhající na jednotlivých GPU jádrech odpovídá Algoritmu 5.4.

Během vyhledávání kandidáta může dojít k potížím s numerickou stabilitou při výpočtech středů tečných sfér a při výpočtech jednotlivých úhlových vzdáleností. Z toho důvodu musí být stanovena množina podezřelých kandidátů S. Postup vyhledávání podezřelých kandidátů, který je prováděn na jednotlivých jádrech GPU, odpovídá Algoritmu 5.3 a potřebuje znát minimální úhlovou vzdálenost θ_{min}, která odpovídá úhlové vzdálenosti kandidáta b_{ef}, jenž byl stanoven pomocí Algoritmu 5.4.

Algoritmus 5.3: Algoritmus prováděný na jádrech GPU - 1.fáze - určení množiny podezřelých kandidátů

Vstup: Množina kandidátů $K = B \setminus \{b_{g1}, b_{g2}, b_{g3}, b_s\}$ o velikosti $n - 4$; pole úhlových vzdáleností Θ o velikosti $n - 4$ pro jednotlivé kandidáty z množiny K; minimální úhlová vzdálenost θ_{min}

Výsledek: Množina podezřelých kandidátů S

```
1 if i < n − 4 then
      // thres je konstanta udávající práh
2     if |Θ_i − θ_min| < thres then
          /* Operace add zde značí uložení podezřelého
          kadidáta na první volnou pozici v poli          */
3         S.add(b_i)
```

Algoritmus 5.4: Algoritmus prováděný na jádrech GPU - 1. fáze - hledání kandidáta

Vstup: Startovní vrchol v_s; trojice generátorů b_{g1}, b_{g2} a b_{g3} definujících hranu e; čtvrtý generátor b_s definující startovní vrchol v_s; množina kandidátů $K = B \setminus \{b_{g1}, b_{g2}, b_{g3}, b_s\}$ o velikosti $n - 4$; inicializační kandidát b_{in} a jeho úhlová vzdálenost θ_{in}; velikost problému $size$

Výsledek: Generátor b_{ef}, jehož úhlová vzdálenost je minimální; pole úhlových vzdáleností Θ pro jednotlivé generátory z množiny kandidátů

```
/* Velikost problému size je nejbližší vyšší mocnina
čísla 2 vzhledem k velikosti množiny kandidátů a udává
počet řešených úkolů v rámci problému. Každá work-item
zná index i řešeného úkolu                              */
```

if $i < n - 4$ **then**
  $b_i \in K$ `// i značí pozici prvku v množině K`
  $\theta_i \leftarrow$ úhlová vzdálenost středu tečné sféry S_{ei} ke generátorům b_{g1}, b_{g2}, b_{g3} a b_i vzhledem k vrcholu v_s
  $\Theta_i \leftarrow \theta_i$
else
  $b_i \leftarrow b_{in}$
  $\theta_i \leftarrow \theta_{in}$
Synchronizace
for *offset = size / 2* **downto** *1* **step** *offset / 2* **do**
  if $i <$ *offset* **then**
    if $\theta_{i+offset} < \theta_i$ **then**
      $b_i \leftarrow b_{i+offset}$
      $\theta_i \leftarrow \theta_{i+offset}$
  Synchronizace

```
// výsledný generátor bude uložen na pozici 0
```

2. fáze

Stejně jako postup první fáze je i postup druhé fáze popsán v Sekci 5.2. Výpočet prováděný na jednotlivých jádrech grafického hardwaru při druhé fázi odpovídá Algoritmu 5.5. Tato fáze je vykonávána jen v případě, že první fáze stanovila neprázdnou množinu podezřelých generátorů S.

Algoritmus 5.5: Algoritmus prováděný na jádrech GPU - 2. fáze - hledání kandidáta

Vstup: Startovní vrchol v_s; trojice generátorů b_{g1}, b_{g2} a b_{g3} definujících hranu e; čtvrtý generátor b_s definující startovní vrchol v_s; množina podezřelých kandidátů S o velikosti m; inicializační kandidát b_{in} a jeho úhlová vzdálenost θ_{in}; velikost problému $size$

Výsledek: Generátor b_{ed}, jehož úhlová vzdálenost je minimální

```
/* Velikost problému size je nejbližší vyšší mocnina
čísla 2 vzhledem k velikosti množiny podezřelých
kandidátů a udává počet řešených úkolů v rámci
problému. Každá work-item zná index i řešeného úkolu  */
```

1 **if** $i < m$ **then**
2 $b_i \in S$ `// i značí pozici prvku v množině` S
3 $\theta_i \leftarrow$ úhlová vzdálenost středu tečné sféry S_{ei} ke generátorům b_{g1}, b_{g2}, b_{g3} a b_i vzhledem k vrcholu v_s
4 **else**
5 $b_i \leftarrow b_{in}$
6 $\theta_i \leftarrow \theta_{in}$
7 Synchronizace
8 **for** *offset = size / 2* **downto** *1* **step** *offset / 2* **do**
9 **if** $i <$ *offset* **then**
10 **if** $\theta_{i+offset} < \theta_i$ **then**
11 $b_i \leftarrow b_{i+offset}$
12 $\theta_i \leftarrow \theta_{i+offset}$
13 Synchronizace

```
// výsledný generátor bude uložen na pozici 0
```

6 Další urychlení algoritmu

Pomocí úpravy algoritmu sledování hran umožňující výpočet Voronoi diagramu na GPU, která byla popsaná v Kapitole 5, bylo sice oproti standardní verzi dosaženo urychlení, nicméně pro velké molekuly je konstrukce diagramu tímto způsobem stále časově náročná. Proto se v této kapitole zaměříme na možnosti dalšího urychlení algoritmu sledování hran pomocí výpočtu na GPU.

6.1 Motivace

Při hledání koncového vrcholu v_e Voronoi hrany e pomocí algoritmu trasování hran dochází k procházení celé množiny kandidátů K, přičemž je potřeba pro každou generující kouli z této množiny vypočítat odpovídající směrový vektor reprezentující úhlovou vzdálenost vzhledem k počátečnímu vrcholu Voronoi hrany. Abychom však mohli vypočítat směrový vektor daného generátoru b_i, musí být vypočtena tečná sféra dotýkající se tohoto generátoru a tří generátorů b_{g1}, b_{g2} a b_{g3} definujících hranu, proto musí být pro každého kandidáta vyřešena soustava rovnic (2.5). Z uvedeného plyne, že stanovení směrového vektoru je výpočetně náročná operace.

6.2 Navržená úprava algoritmu pro GPU

Základní myšlenka urychlení algoritmu spočívá v tom, že při hledání koncového vrcholu Voronoi hrany nejprve zredukujeme množinu kandidátů K pro danou hranu pomocí jednoduššího testu než je porovnávání úhlové vzdálenosti. Tzn. pokud generátor z množiny K splňuje testovací podmínku, pak jej uložíme do zredukované množiny kandidátů R. Pokud po otestování všech generátorů z množiny K bude množina zredukovaných kandidátů R prázdná, provedeme přiřazení $R := K$. Následně již budeme provádět Algoritmus 5.2, přičemž vstupní množina kandidátů bude odpovídat množině R.

V další části této kapitoly se budeme zabývat stanovením testovací podmínky, pomocí které dokážeme zredukovat množinu kandidátů K a dosáhneme tím urychlení výpočtu.

6.3 Zmenšení prostoru pro vyhledávání generátoru koncového vrcholu hrany

Již dříve bylo uvedeno, že z geometrického hlediska je Voronoi hrana kuželosečka a je definována třemi generátory b_{g1}, b_{g2} a b_{g3}. Stanovení oblasti prostoru, ve které se zcela určitě nachází čtvrtý generátor koncového Voronoi vrcholu, pak závisí na tvaru této hrany, konkrétně musíme rozlišovat zda se jedná o hranu eliptickou či neeliptickou.

6.3.1 Neeliptická hrana

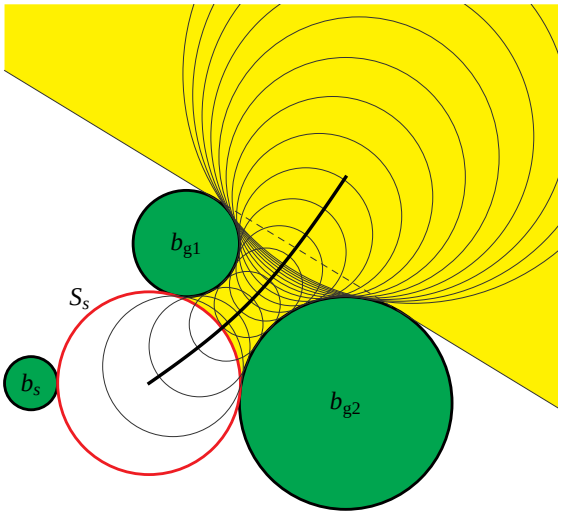
V případě, že je trasovaná hrana neeliptická, je nutno otestovat, zda některý z jejích generátorů b_{g1}, b_{g2} a b_{g3} není imaginární nekonečný generátor b_∞. Pokud by jedna z generujících koulí odpovídala tomuto generátoru, pak není možné provést zmenšení prostoru pro vyhledávání, a tudíž musí být porovnávána úhlová vzdálenost pro všechny generátory z množiny kandidátů K. V opačném případě je možné prostor omezit na základě určitých kritérií.

Při omezování prostoru pro vyhledávání generátoru koncového vrcholu neeliptické hrany můžeme brát na zřetel dva různé přístupy. Prvním přístupem je, že chceme prostor omezit tak, aby v něm byl generátor koncového vrcholu dané hrany vždy zcela určitě obsažen. Druhý přístup je, že pokud existuje alespoň jeden generátor z množiny kandidátů K, který má s definovaným podprostorem společný průnik, pak se v takto omezeném prostoru zcela určitě nachází generátor koncového vrcholu, v opačném případě musí být prohledána celá množina kandidátů. Tzn. při použití druhého přístupu nemáme zajištěnou existenci řešení v zredukovaném prostoru. Jelikož využití této práce je pro konstrukci aditivně váženého Voronoi diagramu biomolekul, o kterých víme, že se skládají z atomů (odpovídají generátorům), jenž jsou v prostoru ve většině případů umístěné blízko sebe, budeme v návrhu úpravy algoritmu pro GPU využívat přístupu druhého. Pro úplnost si zde však nastíníme i přístup první.

První přístup

V prvním přístupu vychází redukce prostoru pro vyhledávání generátoru koncového vrcholu z toho, že koncový vrchol v_e bude středem některé sféry z množiny sfér tečných ke třem generátorům b_{g1}, b_{g2} a b_{g3} definujícím hranu, přičemž středy těchto tečných sfér leží na orientované Voronoi hraně v intervalu $(v_s, v_\infty]$, kde v_s je počáteční vrchol hrany a v_∞ je vrchol hrany v nekonečnu. Tečná sféra S_∞ se středem v bodě v_∞ odpovídá rovině tečné ke třem generátorům hrany. Sjednocení uvedených tečný sfér, pak definuje podprostor, se

kterým má generátor koncového vrcholu neprázdný průnik. Pro představu je tato situace pro 2D analogii znázorněna na Obrázku 6.1 a celý přístup je podrobně popsán v článku [4].



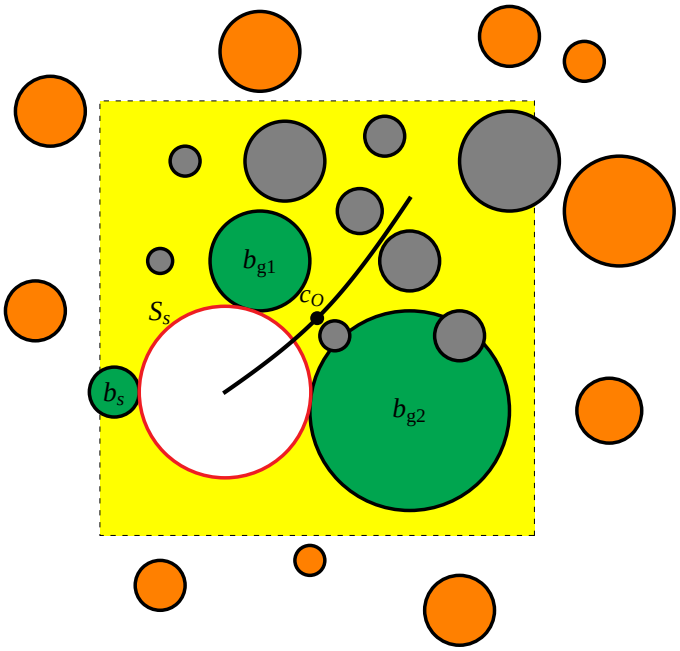
Obrázek 6.1: Znázornění tečných sfér ke gate koulím a oblasti, ve které bude umístěn koncový Voronoi vrchol neeliptické hrany (2D analogie)

Druhý přístup

Druhým přístupem, který budeme používat v návrhu urychlení algoritmu trasování hran pomocí výpočtu na GPU, dojde k omezení prostoru pomocí krychle, jejíž střed bude v bodě c_O a délka hrany bude a_O. Pro představu je na Obrázku 6.2 zobrazen takto omezený prostor pro 2D analogii.

Střed c_O omezující krychle umístíme do bodu Voronoi hrany, pro který platí, že je středem sféry tečné ke generátorům b_{g1}, b_{g2} a b_{g3} definujícím hranu, přičemž tato sféra má minimální poloměr. Střed tečné sféry s minimálním poloměrem odpovídá průsečíku Voronoi hrany s rovinou, která je určena středy tří generátorů hrany. Tento bod byl jako střed omezující krychle zvolen z toho důvodu, že neeeliptická Voronoi hrana je podle něj symetrická.

Předpokládejme, že b_{g1}, b_{g2} a b_{g3} jsou generátory neeliptické Voronoi hrany a generátor b_{g3} má z uvedených generátorů nejmenší poloměr r_{g3}. Pak bez ztráty na obecnosti můžeme zmenšit poloměr všech generátorů o hodnotu r_{g3}, čímž generátor b_{g3} přejde na bod, a aplikovat transformaci tak, aby generátor b_{g3} odpovídal počátku souřadného systému.



Obrázek 6.2: Znázornění podprostoru, ve kterém se bude vyhledávat koncový generátor koncového vrcholu (2D analogie). Šedé generátory představují zredukovanou množinu kandidátů, oranžové generátory patří mezi vyřazené kandidáty.

Bod $c_O = (x, y, z)$ můžeme nyní stanovit řešením následující soustavy rovnic:

$$n_1 x + n_2 y + n_3 z = 0 \tag{6.1a}$$

$$(x - x_{g1})^2 + (y - y_{g1})^2 + (z - z_{g1})^2 = (r + r_{g1})^2 \tag{6.1b}$$

$$(x - x_{g2})^2 + (y - y_{g2})^2 + (z - z_{g2})^2 = (r + r_{g2})^2 \tag{6.1c}$$

$$x^2 + y^2 + z^2 = r^2, \tag{6.1d}$$

kde $n = (n_1, n_2, n_3)$ je normálový vektor roviny středů generátorů hrany, a tudíž platí $n = c_{g1} \times c_{g2}$. Nakonec musíme pro takto vypočítaný bod provést zpětnou transformaci do původního systému rovnic. Druhou možností, jak vypočítat bod c_O, je nalézt extrém Voronoi neeliptické hrany. To lze provést stejným postupem jako při hledání extrémů eliptické hrany, který je uveden v Sekci 6.3.2, přičemž střed omezující krychle bude odpovídat extremálnímu bodu, jenž je definován jako střed koule s nezáporným poloměrem.

Délka hrany a_O omezující krychle by měla být v navržené úpravě algoritmu volena experimentálně tak, aby při vyhledávání koncového vrcholu hrany co nejméně docházelo k prohledávání celé množiny kandidátů K, ale na druhou stranu, aby nebyla zredukovaná množina kandidátů R moc velká. Navrhujeme začít na hodnotě dané vztahem:

$$a_O = 2\max\{\|c_{g1} - c_O\| + r_{g1}, \|c_{g2} - c_O\| + r_{g2}, \|c_{g3} - c_O\| + r_{g3}, \|v_s - c_O\| + r_s\}, \tag{6.2}$$

kde (c_{gi}, r_{gi}) jsou geometrie generátorů Voronoi hrany a (v_s, r_s) udává geometrii počáteční tečné sféry, a na základě výsledků pro různé molekuly tuto hodnotu upravit tak, aby byly splněny požadavky.

Testovací podmínka pro umístění generátoru $b_i = (x_i, y_i, z_i, r_i) \in K$ do zredukované množiny kandidátů R je:

$$\begin{aligned} |x_i - x_O| &\leq \frac{a_O}{2} + r_i \quad \wedge \\ |y_i - y_O| &\leq \frac{a_O}{2} + r_i \quad \wedge \\ |z_i - z_O| &\leq \frac{a_O}{2} + r_i. \end{aligned} \tag{6.3}$$

6.3.2 Eliptická hrana

Tečná sféra definující koncový vrchol eliptické Voronoi hrany bude ležet v oblasti prostoru, která má tvar koule. Aby bylo možné určit střed a poloměr této omezující koule, musí být nejprve stanovena rovnice popisující eliptickou Voronoi hranu. Matematický popis Voronoi hrany lze vypočítat podobně jako Voronoi vrchol, viz Sekce 2.4.1. Hrana je definována generátory b_{g1}, b_{g2} a b_{g3}, přičemž předpokládáme, že generátor b_{g3} má nejmenší poloměr r_{g3}. Pokud poloměr všech tří generátorů zmenšíme o hodnotu r_{g3} a aplikujeme na ně translaci, jejíž vektor posunutí je $-c_{g3}$, kde c_{g3} je střed generátoru b_{g3}, pak popis hrany získáme vyřešením následující soustavy tří rovnic o čtyřech neznámých:

$$(x - x_{g1})^2 + (y - y_{g1})^2 + (z - z_{g1})^2 = (r + r_{g1})^2 \tag{6.4a}$$

$$(x - x_{g2})^2 + (y - y_{g2})^2 + (z - z_{g2})^2 = (r + r_{g2})^2 \tag{6.4b}$$

$$x^2 + y^2 + z^2 = r^2, \tag{6.4c}$$

jejíž výsledek je možné formálně popsat rovností:

$$At^2 + 2Bts + Cs^2 + 2Dt + 2Es + F = 0, \tag{6.5}$$

kde A, B, C, D, E a F jsou koeficienty kuželosečky. Když máme stanovenu rovnici Voronoi hrany, potřebujeme vybrat z množiny sfér (viz Obrázek 6.3), které jsou tečné ke třem generátorům definujícím hranu a jejichž středy leží na dané hraně, sféru s minimálním resp. maximálním poloměrem. Středy těchto dvou sfér odpovídají extrémům na eliptické Voronoi hraně, které lze určit pomocí derivace implicitní funkce popisující tuto hranu. Tzn. budeme derivovat rovnici (6.5), přičemž předpokládáme, že s je funkce jedné proměnné

t, podle které budeme derivovat. Derivace této funkce pak je:

$$\frac{\mathrm{d}s}{\mathrm{d}t} = -\frac{At + Bs + D}{Bt + Cs + E} \tag{6.6}$$

a extrémy získáme vyřešením následující soustavy rovnic:

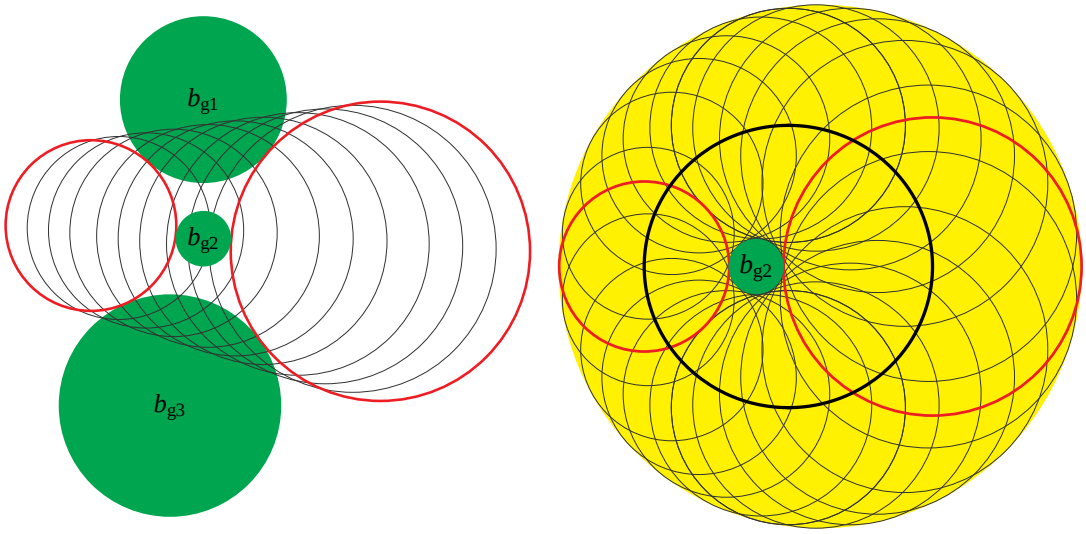
$$At + Bs + D = 0 \tag{6.7a}$$

$$At^2 + 2Bts + Cs^2 + 2Dt + 2Es + F = 0. \tag{6.7b}$$

Když máme stanoveny extrémy $m = (t_m, s_m)$ a $M = (t_M, s_M)$, určíme geometrii jim odpovídajících tečných sfér dosazením do vztahu:

$$S(t, s) = a + su + tv, \tag{6.8}$$

kde $S(t, s) = (c, r)$ označuje tečnou sféru se středem v bodě c a poloměrem r a vektory a, u a v definují rovinu hrany. Nakonec pro takto vypočítané sféry provedeme zpětnou transformaci do původního systému souřadnic a zmenšíme jejich poloměr o r_{g3}.



Obrázek 6.3: Znázornění tečných sfér ke gate koulím a oblasti, ve které bude umístěn koncový Voronoi vrchol eliptické hrany. Převzato z [4].

Nyní máme stanoveny tečné sféry s extrémními poloměry a můžeme tudíž určit geometrii koule $S_O = (c_O, r_O)$, která bude omezovat prostor pro vyhledání čtvrtého generátoru

b_{g4} koncového vrcholu. Střed a poloměr této omezující koule je dán následujícími vztahy:

$$c_O = \frac{1}{2}\left(c_{min} + c_{max} + \frac{(r_{max} - r_{min})(c_{max} - c_{min})}{\|c_{max} - c_{min}\|}\right) \tag{6.9}$$

$$r_O = \frac{1}{2}\left(r_{min} + r_{max} + \|c_{max} - c_{min}\|\right). \tag{6.10}$$

kde $S_{min} = (c_{min}, r_{min})$ resp. $S_{max} = (c_{max}, r_{max})$ je tečná sféra s minimálním resp. maximálním poloměrem.

Testovací podmínka pro umístění generátoru $b_i = (c_i, r_i) \in K$ do zredukované množiny kandidátů R je:

$$\|c_i - c_O\|^2 \leq (r_i + r_O)^2. \tag{6.11}$$

7 Implementace

Hostitelská část algoritmu sledování hran upraveného pro výpočet na GPU, který rozšiřuje knihovnu awVoronoi, je naimplementována v jazyce Java. Tento programovací jazyk byl zvolen proto, že v něm je provedena implementace celé knihovny awVoronoi. Pro implementaci výpočtu na GPU byl zvolen standard OpenCL, jelikož není podmíněn využitím hardwaru od konkrétního výrobce, a tudíž umožňuje provádět obecné výpočty na většině GPU. Propojení Javy s OpenCL umožňuje knihovna JOCL [1].

Implementace konstrukce Voronoi diagramu pomocí trasování hran, která využívá výpočet na GPU, je provedena na základě upraveného algoritmu popsaného v Sekci 5.3, konkrétně jsou využity Algoritmy 5.1 – 5.5. V této kapitole jsou uvedeny detaily, které je potřeba zahrnout do implementace, aby byla zajištěna správná funkčnost konstrukce Voronoi diagramu.

7.1 Hostitelská část

Hostitelská část programu je spouštěna na hostitelském systému (většinou CPU) a umožňuje správu OpenCL prostředí.

7.1.1 Paralelní výpočet pomocí OpenCL API

Provedení paralelního výpočtu pomocí OpenCL API je možno rozdělit do pěti částí, jejichž popisem se budeme nyní zabývat.

Inicializace

Během tohoto kroku jsou zjištěny informace o platformách a je provedeno vytvoření kontextu, ve kterém běží výpočty, a volba zařízení. Nejprve je zvolena platforma (viz Sekce 4.2), která je reprezentována objektem `cl_platform`, a jejíž inicializace je zajištěna funkcí `clGetPlatformIDs(...)`.

Dále je na základě získaných informací o platformě vytvořen kontext (viz Sekce 4.3.1), kterému jsou zároveň přiřazena všechna fyzická zařízení dostupná pro zvolenou

platformu. Kontext je v OpenCL referencován pomocí objektu `cl_context`, přičemž jeho inicializace včetně získání dostupných zařízení je provedena pomocí funkce `clCreateContextFromType(...)`. Každé OpenCL zařízení lze specifikovat jeho identifikátorem, který je reprezentován pomocí objektu `cl_device_id`. Pole těchto identifikátorů pro nalezená a použitá zařízení je získáno pomocí funkce `clGetContextInfo(...)`, přičemž parametr `param_name` je nastaven na hodnotu `CL_CONTEXT_DEVICES`.

Nakonec je pomocí funkce `clCreateCommandQueue(...)` vytvořena příkazová fronta (viz Sekce 4.3.2), která je reprezentována objektem `cl_command_queue`.

Tvorba kernelů

V tomto kroku se vytváří OpenCL programy, které se skládají z množiny kernelů a funkcí, které jsou z jednotlivých kernelů volané. Kernel začíná identifikátorem `__kernel` a jeho návratový typ musí být void. Zdrojový kód OpenCL programu je uložen v souboru, ze kterého musí být načten do pole stringů. Načtení souboru je zajištěno metodou `oclLoadProgSource(...)`, jež je součástí třídy `OpenCLProgramReader`. Z načteného zdrojového kódu se pomocí funkce `clCreateProgramWithSource(...)` vytvoří program, který je následně zkompilován funkcí `clBuildProgram(...)`. Dále je poskytnuta funkce `clGetProgramBuildInfo(...)`, která umožňuje zobrazit log s případnými chybami, které vznikly v průběhu kompilace. Kompilace probíhá vždy za běhu hostitelského programu.

Nakonec jsou po úspěšné kompilaci získány pomocí `cl_kernel` objektů vstupní body do programu. Objekt typu `cl_kernel` je vytvořen na základě funkce `clCreateKernel(...)`.

Alokace paměti

Paměť pro OpenCL zařízení je alokována pomocí funkce `clCreateBuffer(...)` a je reprezentována datovým typem `cl_mem`. Přístup k paměti je definován na základě parametru `flags`. Takto alokovaná paměť je využita za účelem sdílení dat mezi hostitelským systémem a OpenCL zařízeními.

Spuštění kernelů

Předtím než je možné kernel spustit, musí mu být nastaveny argumenty. K tomu je využita funkce `clSetKernelArg(...)`, která je volaná pro každý argument. Poté,

co jsou argumenty nastaveny, je možné zavolat vykonávání kernelu pomocí příkazu `clEnqueueNDRangeKernel(...)`.

Ukončení

Po skončení kernelu je potřeba přečíst výsledná data ze sdílené paměti, toho je dosaženo pomocí `clEnqueueReadBuffer(...)`. Nakonec jsou ještě uvolněny nepotřebné výpočetní prostředky a paměť pomocí `clRelease`.

7.1.2 Sdílení struktur

Z Algoritmů 5.4 a 5.5 je patrné, že potřebujeme sdílet strukturu, která definuje generující kouli, a strukturu reprezentující Voronoi vrchol.

Výpočet probíhající na GPU potřebuje pro úspěšné dokončení znát geometrii (tzn. střed a poloměr) generující koule a její jednoznačný identifikátor sloužící k vyhledávání příslušné koule v množině generátorů. Identifikátor je reprezentován pomocí datového typu `int` a geometrie pomocí vektorového datového typu `cl_float4` resp. `cl_double4` v závislosti na požadované přesnosti výpočtu, přičemž první tři složky vektoru představují souřadnice středu a čtvrtá složka udává poloměr generující koule.

Pro účely výpočtu na OpenCL zařízeních je třeba u Voronoi vrcholu znát geometrii odpovídající tečné sféry a množinu čtyř generátorů, které se této sféry dotýkají. Geometrie tečné sféry je reprezentována pomocí vektorového datového typu `cl_float4` resp. `cl_double4` opět v závislosti na požadované přesnosti. Čtveřice generátorů definující Voronoi vrchol je pak reprezentována pomocí vektorového datového typu `cl_int4`, ve kterém jsou uloženy identifikátory těchto generátorů, a bufferu, ve kterém jsou uloženy jejich geometrie.

7.1.3 Víceúrovňová redukce

Algoritmy 5.4 a 5.5, které jsou vykonávány na GPU, v sobě zahrnují redukční operaci, jejíž průběh je znázorněn na Obrázku 5.1 a bude podrobně popsán v Sekci 7.2. Redukční operaci je však možné provádět pouze v rámci work-group. Pokud tedy velikost množiny kandidátů na generátor koncového vrcholu je menší než maximální velikost work-group, můžeme provést redukční operaci tak, jak je posána v Sekci 7.2. Jestliže však dojde k rozdělení úkolu do více work-groups, je potřeba provést víceúrovňovou redukční operaci.

Předpokládejme, že jsou data rozdělena do p_1 work-groups. Princip víceúrovňové redukce pak spočívá v tom, že v každé work-group je nad částí přidělených dat aplikována

redukční operace, ze které získáme výsledek, jenž je uložen do nového pole. Pokud je toto pole větší než maximální velikost work-group, pak musí opět dojít k rozdělení dat do p_i pracovních skupin, kde i označuje prováděnou úroveň, a celý postup se opakuje tak dlouho, dokud neplatí $p_i = 1$, což je podmínka pro získání pouze jednoho výsledného generátoru.

Stanovení počtu úrovní, velikostí NDRange pro jednotlivé úrovně a počtu work-groups včetně jejich velikosti v jednotlivých úrovních je provedeno pomocí metody `initialize(pocet_kandidatu)` ze třídy `ReductionPass`.

7.2 Kernely

Kernely implementované pro algoritmus trasování hran mají za úkol nalézt generující kouli, která definuje tečnou sféru s minimální úhlovou vzdáleností vzhledem ke startovnímu Voronoi vrcholu počítané hrany. Tento úkol je ekvivalentní problému nalezení minima. Při hledání minima však narážíme na jeden z problémů paralelní implementace pomocí GPU, jelikož nemůžeme kvůli paralelnímu přístupu do paměti využít klasických metod. Tento problém je však možno řešit pomocí redukční operace.

```
barrier(CLK_LOCAL_MEM_FENCE);
for(int offset=get_local_size(0)>>1; offset>0; offset>>=1){
   if(lid<offset){     //lid = local id
      if(theta[lid+offset]<theta[lid]){
         theta[lid]=theta[lid+offset];
         candidate[lid]=candidate[lid+offset];
      }
   }
   barrier(CLK_LOCAL_MEM_FENCE);
}
```

Výpis 7.1: Kód redukční operace probíhající na jednotlivých work-item

Z uvedeného fragmentu kódu je patrné, že je při redukční operaci využívána polovina work-items, které porovnají levou polovinu pole úhlových vzdáleností s pravou polovinou a menší úhlové vzdálenosti uloží do levé poloviny pole `theta` a k nim příslušné generátory uloží do levé poloviny pole `candidate`. V dalším kroku se levá polovina opět rozdělí a celý proces se opakuje, dokud se do polí `theta` a `candidate` nepřesune na pozici s indexem 0 minimální úhlová vzdálenost a k ní příslušná generující koule.

Během redukční operace musí probíhat synchronizace, a to před jejím spuštěním a poté po každém kroku cyklu. Synchronizace work-items je v rámci work-group provedena

pomocí bariéry, přičemž každá work-item ve work-group se nejprve musí zastavit na bariéře, než je kterékoliv work-item povoleno pokračovat ve vykonávání kódu následujícího za bariérou. Poznamenejme, že v rámci OpenCL je umožněna synchronizace pouze uvnitř work-group, globální synchronizace není povolena.

7.2.1 Kernel pro 1. fázi výpočtu na GPU

Během první fáze výpočtu hledáme čtvrtý generátor koncového vrcholu Voronoi hrany, přičemž všechna data v plovoucí řádové čárce mají jednoduchou přesnost, tzn. jsou reprezentovány datovým typem `float`. Zdrojový kód OpenCL programu implementující první fázi výpočtu je uložen v souboru `Kernels.cl`. Tento program implementuje tři kernely. Kernel `computeNeighbor` je vykonáván jednotlivými work-items v první úrovni redukční operace. Tento kernel implementuje Algoritmus 5.4 a zpracovává celá pole generátorů. Kernel `computeNeighborNextPass` je pak vykonáván jednotlivými work-items ve zbylých úrovních víceúrovňové redukce. Druhý kernel se od kernelu `computeNeighbor` odlišuje tím, že vstupními daty je pouze pole, jenž obsahuje identifikátory generátorů vybraných v předchozí úrovni, a pole, v němž jsou uloženy úhlové vzdálenosti příslušné k vybraným generátorům, které byly vypočteny v první úrovni. Kernel `computeNeighborNextPass` tudíž již nepočítá úhlové vzdálenosti, ale pouze provádí výběr minimální úhlové vzdálenosti, přičemž postup odpovídá řádkům 10 – 15 v Algoritmu 5.4.

Součástí první fáze výpočtu je taktéž stanovení podezřelých generátorů pro druhou fázi. K tomu je určen kernel `computeSuspect`, který svou funkčností odpovídá Algoritmu 5.3. Pokud je odhalena podezřelá generující koule, pak je uložen její identifikátor do pole podezřelých generátorů `suspect`. Abychom co nejvíce zredukovali velikost problému pro druhou fázi výpočtu, je potřeba, aby byly identifikátory podezřelých generátorů ukládány souvisle od začátku pole, a aby byl stanoven jejich celkový počet. K tomu bychom však potřebovali provádět globální synchronizaci, která však v OpenCL není umožněna. Tento problém byl tudíž vyřešen použitím atomické funkce `atomic_inc(...)` na proměnnou reprezentující počet podezřelých generátorů, která provede inkrementaci uvedené proměnné a navrátí její původní hodnotu, která zároveň udává index prvku v poli `suspect`, do nějž má být uložen identifikátor podezřelého generátoru. Poznamenejme, že prahová hodnota $thres$, pomocí které se určuje, zda je generátor podezřelý, byla stanovena experimentálně.

7.2.2 Kernel pro 2. fázi výpočtu na GPU

Druhá fáze výpočtu na GPU stanoví čtvrtý generátor koncového vrcholu Voronoi hrany z množiny podezřelých vrcholů, přičemž jsou používány hodnoty v plovoucí řádové čárce s dvojitou přesností, tzn. datový typ `double`. Zdrojový kód OpenCL programu implementující druhou fázi výpočtu je uložen v souboru `KernelsDouble.cl`. V tomto programu jsou implementovány dva kernely `computeNeighbor` a `computeNeighborNextPass`, přičemž jejich využití je stejné jako v Sekci 7.2.1. Kernel `computeNeighbor` implementuje Algoritmus 5.5. Kernel `computeNeighborNextPass` pak již opět pouze vyhledává minimální úhlovou vzdálenost, přičemž tento postup odpovídá řádkům 7 – 14 v Algoritmu 5.5.

8 Experimenty a výsledky

V této kapitole jsou prezentovány výsledky, které byly získány měřením časové náročnosti výpočtu Voronoi diagramu a je provedeno jejich porovnání s dostupnými implementacemi. Dále jsou zde zaznamenány výsledky testu, který měl za úkol stanovit pro algoritmus trasování hran pomocí GPU, kolik koncových vrcholů Voronoi hrany bude správně vypočteno až v průběhu druhé fáze, tj. jak velká nepřesnost vzniká během výpočtu koncového vrcholu, pokud je pro čísla v plovoucí řádové čárce využita jednoduchá přesnost. Nakonec je zde uvedeno statistické srovnání odchylek výpočtu koncového vrcholu Voronoi hrany v jednoduché přesnosti vzhledem k výpočtu ve dvojité přesnosti.

8.1 Doba výpočtu

V této sekci se zaměříme na porovnání implementace navržené úpravy algoritmu umožňující výpočet na GPU s dostupnými implementacemi z hlediska časové náročnosti výpočtu Voronoi diagramu. Měření doby výpočtu pro různě velké molekuly bylo provedeno na počítači s 16 GB RAM, procesorem Intel Core i7-3770S CPU @ 3.10 GHz 3.10 GHz (4 jádra + HT $\Rightarrow$ 8 vláken), GPU NVIDIA GeForce GTX 660 Ti (1019 MHz, PCI Express 3.0, 7SM, 1344 CUDA Cores) osazeným 2 GB GDDR5 a 64-bitovým operačním systémem Windows 7.

Mezi dostupné implementace, s kterými je porovnáván navržený algoritmus pro GPU, patří:

Brute force
: provádí trasování Voronoi hran v základní podobě.

Paralelní brute force (parallel brute force)
: provádí trasování Voronoi hran, přičemž výpočet koncového vrcholu pro každou hranu je paralelizován pomocí vláken běžících na CPU.

Filtrovaný paralelní brute force (filtered parallel brute force)
: provádí trasování Voronoi hran, přičemž výpočet každého koncového vrcholu je opět paralelizován pomocí vláken běžících na CPU. Navíc jsou používány prostorové filtry, které omezí oblast prohledávání, a tím zajistí lepší výkonnost.

Delaunay

provádí trasování Voronoi hran, přičemž při výpočtu koncového vrcholu hrany používá Delaunay triangulaci středů generátorů a prostorové filtry, aby byl omezen prohledávaný prostor, a tím došlo k urychlení výpočtu.

Navíc pro některé z nich můžeme vybírat mezi dvěma trasovacími strategiemi:

Sekvenční (SEQUENTIAL)

Voronoi hrany, které mají být trasovány, jsou zpracovávány sériově.

Paralelní (PARALLEL)

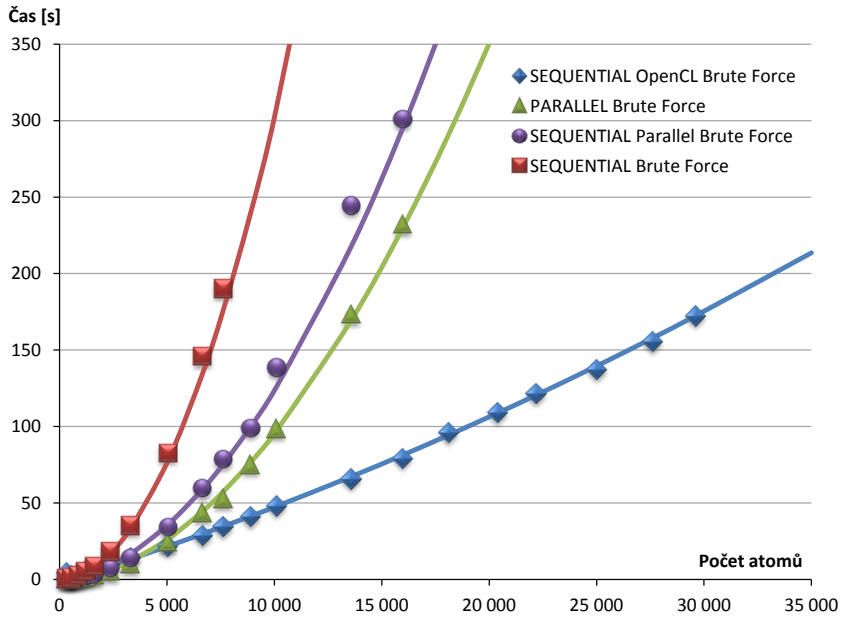
Voronoi hrany, které mají být trasovány, jsou zpracovávány paralelně pomocí vláken běžících na CPU.

Trasovací strategie		**SEKVENČNÍ**					**PARALELNÍ**	
PDB ID	**Počet atomů**	**BF [s]**	**PAR BF [s]**	**Filtr PAR BF [s]**	**OCL BF [s]**	**DT [s]**	**BF [s]**	**DT [s]**
2LKW	312	0,46	0,32	0,26	4,03	0,24	0,23	0,14
3U23	562	1,11	0,65	0,36	2,38	0,27	0,37	0,15
3PS8	859	2,52	1,16	0,54	3,45	0,39	0,75	0,21
2Y4Q	1 209	5,19	2,23	0,85	5,31	0,62	1,60	0,28
3UYO	1 614	8,51	3,22	1,06	6,67	0,60	2,57	0,31
1CQW	2 358	18,20	7,37	1,71	10,17	0,95	5,21	0,50
2Y36	3 294	35,05	14,01	2,67	14,12	1,39	10,18	0,68
4NU6	5 034	82,61	34,02	4,83	21,69	1,98	24,27	0,93
3B08	6 649	145,82	59,71	7,08	28,62	2,62	43,76	1,20
3WBH	7 612	190,18	78,46	8,57	34,43	3,00	52,69	1,36
4P12	8 888		99,18	10,73	40,90	3,64	75,02	1,65
4PWX	10 094		138,52	13,81	47,84	4,36	98,40	1,95
2OAU	13 573		244,45	22,82	65,55	6,17	173,42	2,65
4CV1	15 969		300,75	28,50	79,14	6,36	232,19	2,70
4MOJ	18 116			34,87	95,87	7,40		3,26
3J5M	20 394			42,68	108,91	8,21		3,52
3WMM	22 183			47,15	121,50	8,97		4,04
3ZHQ	25 015			71,71	137,26	10,48		4,85
4ATU	27 600			91,10	155,56			
3SQG	29 607			96,87	172,15	13,42		6,17

Tabulka 8.1: Tabulka s naměřenými časy pro jednotlivé verze a různé velikosti molekuly (BF - brute force, PAR - paralelní, Filtr - filtrovaný, OCL - OpenCL, DT - Delaunay)

Naměřené doby výpočtu Voronoi diagramu včetně předzpracování jsou uvedeny v Tabulce 8.1. Pro všechny dostupné implementace, až na verzi využívající Delaunay triangulaci, je doba trvání předzpracování zanedbatelná. Pro verzi využívající Delaunay triangulaci není čas strávený předzpracováním zanedbatelný proto, že během něj dochází k výpočtu Delaunay triangulace pro středy generátorů.

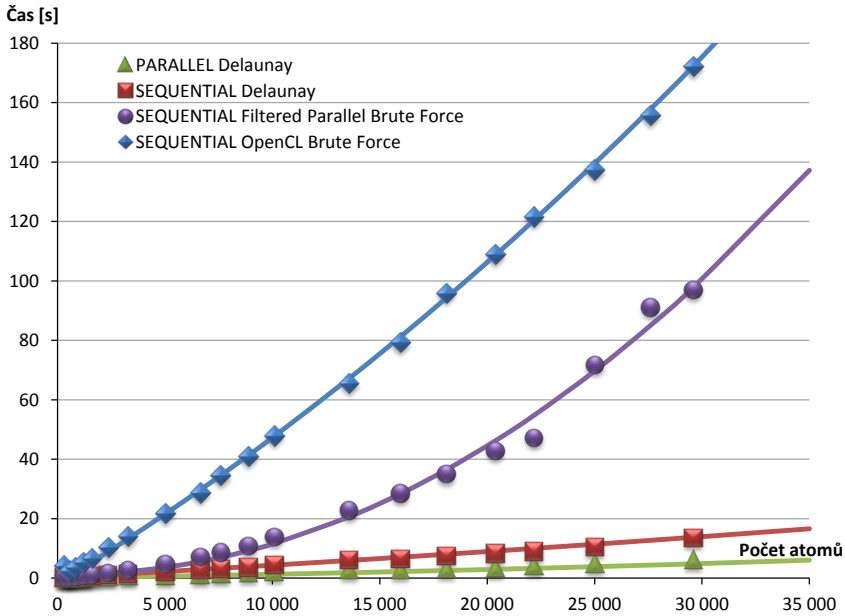
Na Obrázku 8.1 je znázorněn graf závislosti doby výpočtu Voronoi diagramu na počtu atomů v molekule. Tato závislost je zde vykreslena pro brute force implementaci výpočtu koncového vrcholu v kombinaci se sekvenční i paralelní trasovací strategií. Dále jsou do grafu vyneseny výsledky pro paralelní brute force výpočet koncových vrcholů a pro brute force výpočet koncových vrcholů probíhající na GPU pomocí OpenCL, obě tyto verze využívají sekvenční strategii zpracování hran určených k trasování. Z uvedeného grafu je partné, že největší urychlení výpočtu oproti standardní verzi (brute force implementace a sekvenční trasovací strategie) poskytuje implementace využívající výpočetních prostředků GPU. Dále můžeme z grafu vyvodit, že z hlediska časové náročnosti je lepší využít pro konstrukci Voronoi diagramu brute force výpočet koncového vrcholu v kombinaci s paralelní trasovací strategií než paralelní brute force výpočet koncového vrcholu a sekvenční trasovácí strategii.



Obrázek 8.1: Graf závislosti doby výpočtu Voronoi diagramu na počtu atomů v molekule pro GPU verzi algoritmu trasování hran v porovnání s dobou výpočtu pro základní verzi algoritmu (Brute Force) a jeho dvě paralelizace prováděné pomocí CPU

Obrázek 8.2 znázorňuje opět graf závislosti diagramu na počtu atomů v molekule, přičemž je zde porovnávána brute force implementace výpočtu koncového vrcholu pro-

bíhající na GPU vůči implementacím, které využívají pro výpočet koncového vrcholu trochu jiné techniky. Mezi tyto implementace patří filtrovaný paralelní brute force výpočet koncového vrcholu v kombinaci se sekvenční trasovací strategií a výpočet koncového vrcholu pomocí Delaunay triangulace v kombinaci se sekvenční i paralelní trasovací strategií. Z tohoto grafu je patrné, že z uvedených implementací je brute force výpočet koncového vrcholu na GPU nejpomalejší. Z časového hlediska je pro konstrukci Voronoi diagramu nejvýhodnější použít implementaci využívající Delaunay triangulaci pro stanovení koncového vrcholu a paralelní trasovací strategii. Na základě výsledků znázorněných na Obrázku 8.2 můžeme vyvodit, že využitím prostorových filtrů je množina kandidátů na generátor koncového vrcholu zredukována do takové míry, že urychlení, kterého bylo dosaženo pomocí brute force výpočtu koncového vrcholu na GPU, není v porovnání s filtrovaným paralelním brute force výpočtem uspokojivé. Situace by se mohla zlepšit pokud bychom při výpočtu, který je prováděn na GPU, omezili prostor, v němž se mohou nacházet kandidáti na generátor koncového vrcholu. Možností, jak prostor zredukovat, se zabývá Kapitola 6.



Obrázek 8.2: Graf závislosti doby výpočtu Voronoi diagramu na počtu atomů v molekule pro GPU verzi algoritmu trasování hran v porovnání s dobou výpočtu pro algoritmus trasování hran využívající Delaunay triangulaci a algoritmus provádějící předfiltrování dat

8.2 Jednoduchá vs. dvojitá přesnost

8.2.1 Chybovost jednoduché přesnosti při určování vrcholu

V této části budou uvedeny výsledky testu, který byl prováděn pro algoritmus trasování hran pomocí GPU. Úkolem tohoto testu bylo stanovení množství koncových vrcholů, které byly správně určeny až v průběhu druhé fáze výpočtu. Tzn. bylo testováno pro jaký podíl z celkového počtu určovaných koncových vrcholů došlo v důsledku použití jednoduché přesnosti čísel v plovoucí řádové čárce k nesprávnému určení generátoru koncového vrcholu. Tento test byl prováděn pro různě velké molekuly a jeho výsledky jsou uvedeny v Tabulce 8.2.

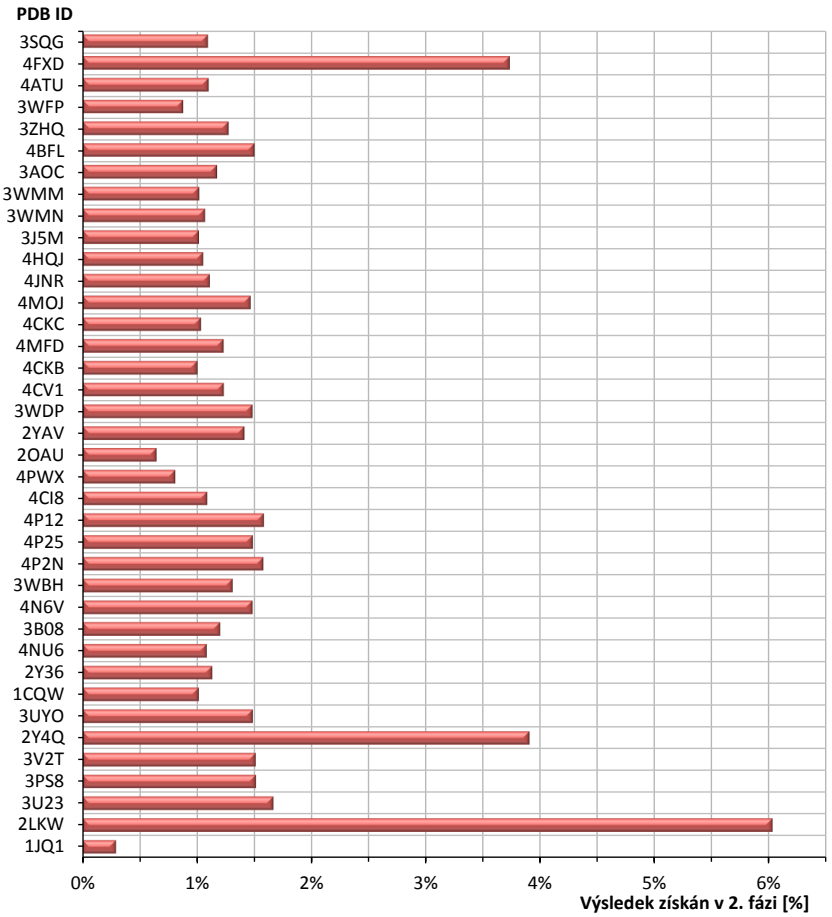
PDB ID	Počet atomů	Počet v_e	Počet záměn
1JQ1	137	1 696	5
2LKW	313	4 258	257
3U23	563	7 238	121
3PS8	860	11 126	169
3V2T	1 029	13 518	205
2Y4Q	1 210	17 288	676
3UYO	1 615	21 334	318
1CQW	2 359	31 544	321
2Y36	3 295	44 130	501
4NU6	5 035	67 990	739
3B08	6 650	89 558	1 079
4N6V	7 421	100 186	1 491
3WBH	7 613	103 492	1 360
4P2N	8 732	117 734	1 863
4P25	8 794	118 614	1 767
4P12	8 889	119 900	1 904
4CI8	9 861	133 280	1 456
4PWX	10 095	137 886	1 121
2OAU	13 574	183 432	1 192

PDB ID	Počet atomů	Počet v_e	Počet záměn
2YAV	14 803	201 224	2 853
3WDP	15 433	211 382	3 146
4CV1	15 970	217 966	2 698
4CKB	17 755	240 876	2 424
4MFD	17 795	241 770	2 985
4CKC	17 910	242 938	2 518
4MOJ	18 116	247 542	3 641
4JNR	19 321	263 974	2 944
4HQJ	20 360	278 280	2 937
3J5M	20 395	276 242	2 818
3WMN	21 859	298 462	3 202
3WMM	22 184	302 762	3 094
3AOC	23 323	319 454	3 762
4BFL	23 401	322 502	4 853
3ZHQ	25 016	342 994	4 386
3WFP	27 525	377 220	3 330
4ATU	27 601	379 070	4 185
4FXD	27 702	415 274	15 527
3SQG	29 608	406 534	4 469

Tabulka 8.2: Tabulka uvádějící celkový počet hledaných koncových vrcholů v_e a počet koncových vrcholů, které byly zaměněny za výsledek z druhé fáze výpočtu, pro různé molekuly

Pruhový graf zobrazený na Obrázku 8.3 pak znázorňuje kolik procent z celkového počtu určovaných koncových vrcholů je při použití jednoduché přesnosti stanoveno chybně.

Z grafu je patrné, že pro většinu molekul je v jednoduché přesnosti nesprávně určeno méně než 2 % koncových vrcholů Voronoi hrany. Existují však molekuly, jejichž konfigurace atomů v prostoru způsobuje větší chybovost. Takové rozmístění atomů má například molekula s PDB ID 2LKW, pro níž bylo v jednoduché přesnosti nesprávně určeno 6 % koncových vrcholů. Celkem bylo pro všechny testované molekuly nesprávně stanoveno 1.34 % koncových vrcholů.



Obrázek 8.3: Graf znázorňující jak velký podíl z celkového počtu počítaných koncových vrcholů je správně stanoven až v průběhu druhé fáze výpočtu (tj. ve dvojité přesnosti čísel v plovoucí řádové čárce) pro různé molekuly

8.2.2 Odchylka mezi výsledky vypočtenými v jednoduché a dvojité přesnosti

V této části jsou uvedeny výsledky dalších testů, které byly provedeny pro algoritmus trasování hran pomocí GPU. Tentokrát se zaměříme na srovnání odchylek mezi hodnotami, které byly získány výpočtem koncových vrcholů Voronoi hran v jednoduché a dvojité přesnosti, přičemž tyto odchylky počítáme pouze pro koncové vrcholy, které byly správně

určeny až v druhé fázi. Tím získáme pro každou molekulu soubor dat, který následně statisticky zpracujeme. Konkrétně byly počítány následující odchylky:

Odchylka směrových vektorů

Pro nalezený koncový generátor byl vypočítán směrový vektor k odpovídajícímu koncovému vrcholu Voronoi hrany jak v jednoduché, tak v dvojité přesnosti a tyto dva vektory byly vůči sobě porovnány. Jejich odchylka byla stanovena jako velikost úhlu, který mezi sebou svírají.

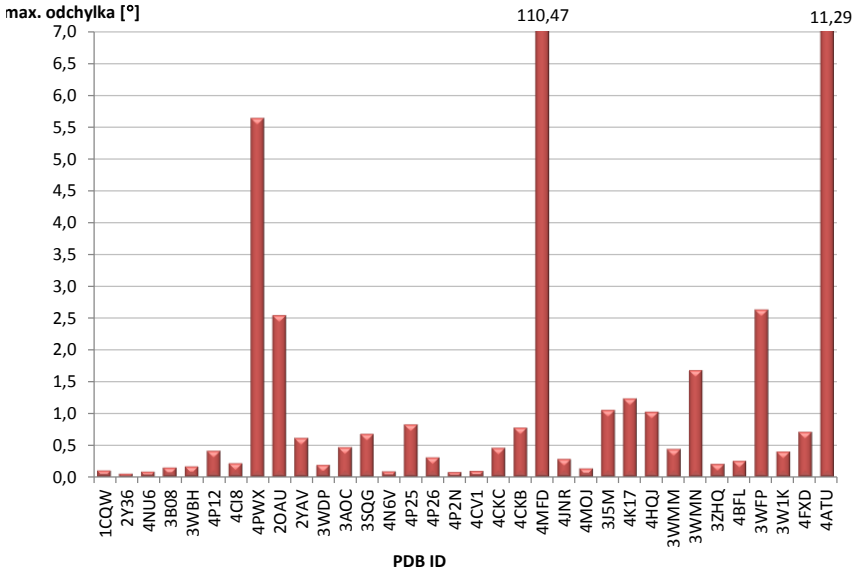
Odchylka středů

Pro nalezený koncový generátor byly vypočítány souřadnice odpovídajícího koncového vrcholu hrany jak v jednoduché, tak v dvojité přesnosti a byla stanovena vzdálenost mezi dvěma body, které tyto souřadnice reprezentují. Výsledná odchylka pak byla stanovena jako relativní hodnota této vzdálenosti vzhledem k poloměru prázdné tečné sféry se středem v koncovém vrcholu stanoveném ve dvojité přesnosti.

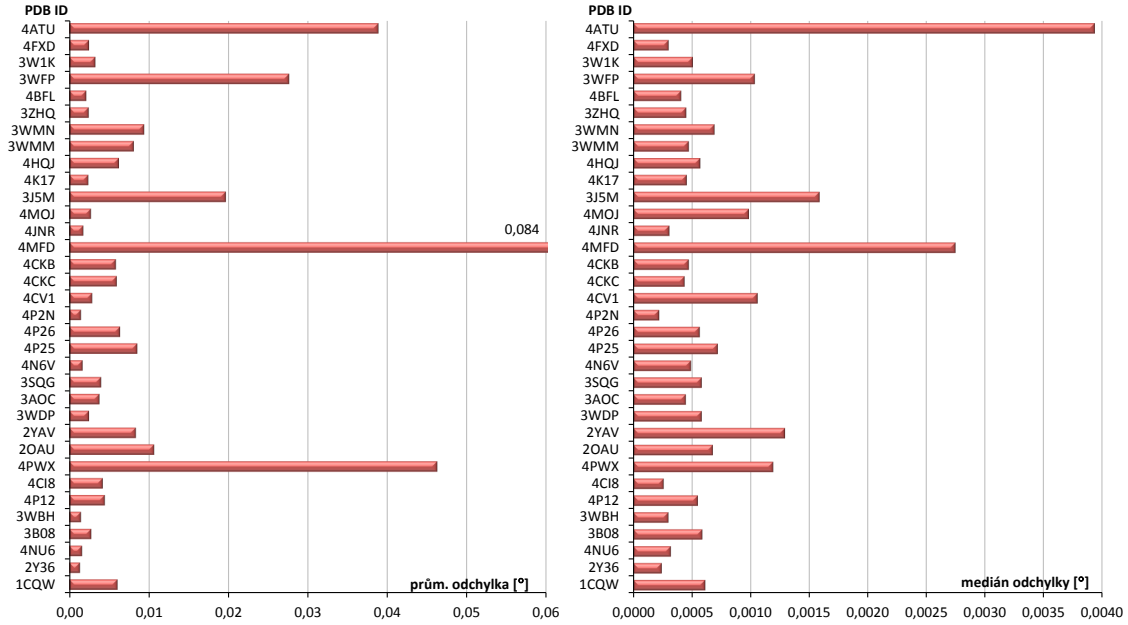
Výsledky pro odchylky směrových vektorů

Na Obrázku 8.4 je graf znázorňující maximální odchylku směrových vektorů stanovených pro koncový vrchol vypočtený v jednoduché resp. dvojité přesnosti, přičemž tato odchylka byla stanovena pro různé molekuly. Z grafu je patrné, že největší maximální odchylka směrových vektorů nastala pro molekulu s PDB ID 4MFD, ve které se uvedené směrové vektory lišily o více než $110°$. Vznik tak značné odchylky mezi směrovými vektory si vysvětlujeme tím, že pro čtveřici generátorů definující koncový vrchol Voronoi hrany existují minimálně dvě tečné sféry, přičemž v důsledku numerické nepřesnosti způsobené výpočtem v jednoduché přesnosti byl jako koncový vrchol, jenž je nejblíže ke startovnímu vrcholu, vybrán střed jiné tečné sféry než té, jejíž střed byl vybrán v přesnosti dvojité. Tato situace pravděpodobně nastala i v jiných molekulách, např. molekula s PDB ID 4ATU nebo 4PWX, pro něž není maximální odchylka směrových vektorů až tak markantní, nicméně na to, aby byla způsobena pouze nepřesností vzniklou použitím jednoduché přesnosti při výpočtu tečné sféry totožné s tou, která byla vybrána ve dvojité přesnosti, je moc velká.

Na Obrázku 8.5 vlevo je pak graf, který zobrazuje průměrnou odchylku směrových vektorů určených pro koncový vrchol vypočtený v jednoduché, resp. dvojité přesnosti. Průměrná odchylka byla opět stanovena pro různé molekuly. Bohužel průměrná odchylka poskytuje velice zkreslené výsledky pro molekuly, které obsahují několik extrémních hodnot. Z toho důvodu vykresluje graf vpravo na Obrázku 8.5 medián odchylky daných směrových vektorů pro různé molekuly, pomocí kterého získáme pro molekuly s několika extrémními hodnotami lepší představu o úrovni hodnot odchylek. Z tohoto grafu je patrné, že odchylka směrových vektorů pro testovanou množinu dat je řádově v tisícinách stupně.



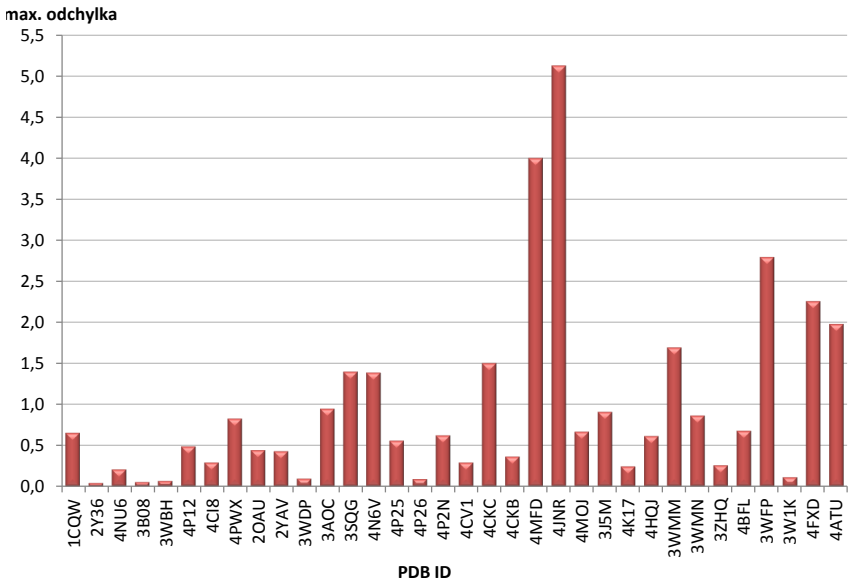
Obrázek 8.4: Maximální odchylka směrového vektoru koncového vrcholu vypočteného v jednoduché přesnosti od směrového vektoru koncového vrcholu stanoveného v přesnosti dvojité, přičemž koncové vrcholy jsou definovány stejnou čtveřicí generátorů, pro různé molekuly. Odchylka je stanovena pomocí úhlu, který mezi sebou uvedená dvojice vektorů svírá.



Obrázek 8.5: Odchylka směrového vektoru koncového vrcholu vypočteného v jednoduché přesnosti od směrového vektoru koncového vrcholu stanoveného v přesnosti dvojité, přičemž koncové vrcholy jsou definovány stejnou čtveřicí generátorů, pro různé molekuly. Odchylka je stanovena pomocí úhlu, který mezi sebou uvedené dva vektory svírají. Na levém grafu je znázorněna průměrná odchylka, graf vpravo zachycuje medián odchylky.

Výsledky pro odchylky středů

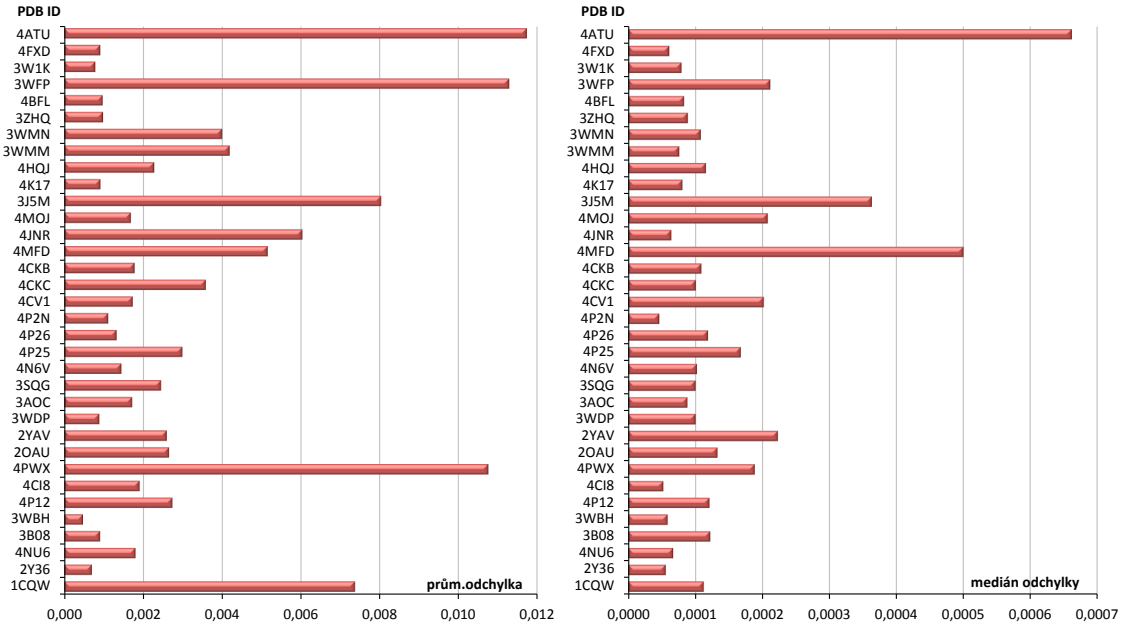
V této části jsou uvedeny výsledky získané při výpočtu odchylky středů, která byla stanovena pro nalezený koncový generátor jako relativní hodnota vzdálenosti odpovídajících koncových vrcholů hrany vypočtených v jednoduché, resp. dvojité přesnosti vzhledem k poloměru prázdné tečné sféry, jejíž střed odpovídá koncovému vrcholu vypočtenému ve dvojité přesnosti. Na Obrázku 8.6 je graf znázorňující maximální odchylku středů pro různé molekuly. Z grafu je patrné, že pro molekulu s PDB ID 4MFD, pro níž byla stanovena extrémní odchylka směrových vektorů, je i maximální odchylka středů velká, přičemž vysvětlení této skutečnosti bude stejné jako pro odchylku směrových vektorů. Z uvedeného grafu je však také zřejmé, že pro molekulu s PDB ID 4JNR je maximální odchylka středů značná, nicméně maximální odchylka směrových vektorů této molekuly svou velikostí nenaznačuje, že by došlo k výběru různých tečných sfér při výpočtech v jednoduché a dvojité přesnosti. Vznik této odchylky si tudíž vysvětlujeme tak, že poloměr tečné sféry, k němuž je odchylka relativně vztažená, byl natolik malý, že se může jednat o nepřesnost způsobenou použitím jednoduché přesnosti při výpočtu tečné sféry totožné s tou, která byla vybrána ve dvojité přesnosti. Uvedená situace pravděpodobně nastala i pro další molekuly.



Obrázek 8.6: Maximální odchylka mezi pozicí koncového vrcholu stanoveného ve dvojité přesnosti a pozicí koncového vrcholu stanoveného v přesnosti jednoduché, přičemž koncové vrcholy jsou definovány stejnou čtveřicí generátorů, pro různě velké molekuly. Vzdálenost mezi těmito body je stanovena relativně vzhledem k poloměru odpovídající tečné sféry.

Vlevo na Obrázku 8.7 je znázorněn graf zobrazující průměrné odchylky středů pro

různé molekuly, přičemž jsou tyto hodnoty pro většinu molekul opět zkresleny v důsledku výskytu několika extrémních hodnot odchylek. Proto je na Obrázku 8.7 vpravo uveden graf znázorňující medián odchylek středů pro různé molekuly, aby bylo možné si vytvořit lepší představu o rozsahu hodnot většiny odchylek. Z tohoto grafu lze vyčíst, že se odchylky středů pohybují řádově v desetitisících.



Obrázek 8.7: Odchylka mezi pozicí koncového vrcholu stanoveného ve dvojité přesnosti a pozicí koncového vrcholu stanoveného v přesnosti jednoduché, přičemž koncové vrcholy jsou definovány stejnou čtveřicí generátorů, pro různě velké molekuly. Vzdálenost mezi těmito body je stanovena relativně vzhledem k poloměru odpovídající tečné sféry. Na grafu vlevo jsou vykresleny průměrné odchylky a graf vpravo znázorňuje medián odchylek.

9 Závěr

Prozkoumali jsme možnosti využití GPGPU k urychlení výpočtu aditivně váženého Voronoi diagramu. Na základě tohoto průzkumu jsme navrhli a implementovali algoritmus pro konstrukci aditivně váženého Voronoi diagramu založený na algoritmu trasování hran a využívající paralelizaci na GPU. Návrhem tohoto algoritmu se zabývá Kapitola 5, přičemž algoritmus probíhá ve dvou fázích, kdy první fáze je počítána v jednoduché přesnosti a druhá ve dvojité přesnosti čísel v plovoucí řádové čárce. Dvě fáze byly využity proto, že pro zajištění správnosti konstrukce Voronoi diagramu potřebujeme využívat výpočet hodnot ve dvojité přesnosti a zároveň chceme plně využít potenciálu GPU, čehož bude docíleno pouze v případě výpočtu v přesnosti jednoduché.

Dále jsme provedli porovnání navrženého algoritmu s dostupnými implementacemi, které jsou součástí knihovny awVoronoi, z hlediska časové náročnosti pro různě velké molekuly proteinů. Výsledkem tohoto porovnání je, že navržený algoritmus dokáže konkurovat algoritmu trasování hran v základní podobě nebo jeho modifikaci, při které dochází k paralelizaci algoritmu sledování hran na CPU. Nicméně při srovnání s implementacemi, které pro urychlení konstrukce Voronoi diagramu využívají prostorových filtrů či Delaunay triangulace středů atomů, poskytuje navržený algoritmus neuspokojivé urychlení. Z toho důvodu byla v této práci navržena další úprava algoritmu pro GPU, která by měla poskytnout větší urychlení. Tato úprava je založena na zredukování prostoru pro vyhledávání koncového vrcholu Voronoi hrany a podrobně je popsána v Kapitole 6.

Nakonec jsme pro navržený algoritmus pro GPU povedli několik experimentů. Na základě jejich výsledků jsme zjistili, že při konstrukci Voronoi diagramu pomocí výpočtu v jednoduché přesnosti čísel v plovoucí řádové čárce dochází průměrně pro 1.34% koncových vrcholů z jejich celkového počtu k chybnému určení čtvrtého generátoru, jenž tento vrchol definuje. V rámci molekuly pak může být chybně určeno více než 6% koncových vrcholů při výpočtu v jednoduché přesnosti. Tato chybovost je způsobena problémy s numerickou přesností, které mohou vést až k výběru nesprávné sféry odpovídající koncovému vrcholu hrany z množiny sfér tečných k jedné konkrétní čtveřici generátorů.

Nevýhodou navrženého algoritmu je, že neposkytuje uspokojivé urychlení konstrukce Voronoi diagramu. Tato situace by mohla být vylepšena zredukováním prostoru pro vy-

hledávání, které bylo navrženo v Kapitole 6. Další urychlení by také mohlo přinést přidání paralelizace zpracovávání hran ze zásobníku pomocí CPU, ta by však měla smysl pouze v případě, že bychom k tomu měli přizpůsobeno hardwarové vybavení, konkrétně bychom potřebovali více OpenCL zařízení, aby každé vlákno spuštěné na CPU mělo přiřazeno svoje vlastní. Tato podmínka značí, že optimální počet vláken spuštěných na CPU pro paralelizaci zpracovávaných hran by odpovídal počtu OpenCL zařízení.

Literatura

[1] JOCL. URL http://www.jocl.org/.

[2] *Introduction to OpenCL Programming*. Advanced Micro Devices, May 2010. URL http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/01/Introduction_to_OpenCL_Programming-Training_Guide-201005.pdf.

[3] *AMD Accelerated Parallel Processing: OpenCL Programming Guide*. Advanced Micro Devices, November 2013. URL http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf.

[4] Y. Cho, D. Kim, H. C. Lee, J. Y. Park, and D.-S. Kim. Reduction of the Search Space in the Edge-Tracing Algorithm for the Voronoi Diagram of 3D Balls. In *Computational Science and Its Applications - ICCSA 2006*, volume 3980 of *Lecture Notes in Computer Science*, pages 111–120, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-34070-6.

[5] M. Gavrilova. An Explicit Solution for Computing the Euclidean d-dimensional Voronoi Diagram of Spheres in a Floating-Point Arithmetic. In *Computational Science and Its Applications - ICCSA 2003*, volume 2669 of *Lecture Notes in Computer Science*, pages 827–835, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 978-3-540-40156-8.

[6] *The OpenCL Specification*. Khronos Group, March 2014. URL http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[7] D.-S. Kim, Y. Cho, and D. Kim. Euclidean Voronoi diagram of 3D balls and its computation via tracing edges. *Computer-Aided Design*, 37(13):1412–1424, 2005. ISSN 0010-4485.

[8] D.-S. Kim, D. Kim, Y. Cho, and K. Sugihara. Quasi-triangulation and interworld data structure in three dimensions. *Computer-Aided Design*, 38(7):808–819, 2006. ISSN 0010-4485.

[9] V. A. Luchnikov, N. N. Medvedev, and M. L. Gavrilova. The Voronoi-Delaunay Approach for Modeling the Packing of Balls in a Cylindrical Container. In *Computational Science - ICCS 2001*, volume 2073 of *Lecture Notes in Computer Science*, pages 748–752, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 978-3-540-42232-7.

[10] M. Maňák. Voronoi Diagrams for Spheres in E3. Master thesis, Charles University in Prague, 2008.

[11] N. N. Medvedev, V. P. Voloshin, V. A. Luchnikov, and M. L. Gavrilova. An Algorithm for Three-Dimensional Voronoi S-Network. *Journal of Computational Chemistry*, 27(14):1676–1692, 2006. ISSN 1096-987X.

[12] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley, 2011. ISBN 0-321-74964-2.

[13] NVidia. CUDA C Programming Guide, 2013. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/. [online, citováno 18. 3. 2014].

[14] M. Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning Publication, Shelter Island, 2011. ISBN 1-61729-017-3.