FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Refaktorizace systému ParaCell
Student:	Bc. Matěj Ulman
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

ParaCell[1] je multiplatformní program pro indexaci v práškové difrakci, napsaný v jazycích C/C++, využívající
technologie s OpenMP pro vícevláknové výpočty a CUDA pro výpočty na grafických čipech.
1. Seznamte se s danou problematikou a analyzujte aktuální stav aplikace.
2. Proveďte rešerši existujících nástrojů pro statickou analýzu kódu a refaktorizujte existující kód (standardizace jmenné konvence a formátování,
eliminace nedosažitelného kódu a redundantního kódu, ...). Zaměřte se na snadnost budoucího rozšíření aplikace.
3. Implementujte vybraný build systém pro automatizaci sestavování projektu. Přidejte možnost konfigurace sestavování v závislosti na zvolených technologiích
(např. možnost vypnout podporu CUDA).
4. Zobecněte existující CUDA výpočty pro možnou implementaci GPU výpočtů pomocí jiných rozhraní. Analyzujte existující hardwarově nezávislé API a jedno z nich implementujte pro podporu GPU výpočtů na hardwaru mimo CUDA ekosystém.
5. Navrhněte a zrealizujte pomocí vhodných technologií jednoduché GUI usnadňující tvorbu konfiguračních souborů.
6. Vytvořte jednotkové testy pokrývajících všechny funkce aplikace.
7. Výsledný kód zdokumentujte, a vytvořte uživatelskou příručku popisující použití aplikace.

[1] https://sourceforge.net/p/paracell/wiki/Home/

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 20. září 2022 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Refaktorizace systému ParaCell

Bc. Matěj Ulman

Katedra softwarového inženýrství
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

2. května 2023

Poděkování

Děkuji docentu Ivanu Šimečkovi za jeho ochotu a pomoc při vypracovávání této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 2. května 2023 .....

Odkaz na tuto práci

Ulman, Matěj. *Refaktorizace systému ParaCell*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Tato práce se zabývá refaktorizací a rozvojem stávající aplikace *ParaCell*, která slouží k indexaci výsledků v práškové difrakci. V práci je představena problematika a je analyzován aktuální stav aplikace, která je následně rozšířena a přepracována. Hlavními přínosy jsou navržení a implementace grafického rozhraní ve frameworku Qt, přepracování stávajícího CUDA kódu do alternativních technologií Vulkan a OpenCL a flexibilní integrace sestavovacího nástroje CMake. Výsledná aplikace je zdokumentována a podrobena automatickému a uživatelskému testování.

Klíčová slova prášková difrakce, Qt, CUDA, Vulkan, OpenCL, CMake

Abstract

This thesis deals with the refactorization and expansion of an existing application called *ParaCell*, used for indexation of powder diffraction records. The thesis introduces the problem domain, along with the analysis of the current application. The main improvements are the design and implementation of a graphical user interface, reworking existing CUDA code into alternative technologies (Vulkan, OpenCL) and a flexible integration of the CMake build system. The final application is documented and undergoes automatic and user testing.

Keywords powder diffraction, Qt, CUDA, Vulkan, OpenCL, CMake

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Hlavním bodem této práce je program *ParaCell*, který slouží k analýze krystalických látek z existujících naměřených dat metodou práškové difrakce. Tento princip původně vznikl v první polovině 20. století a mnoho dnes používaných metod jsou stále desítky let staré. Nicméně vzhledem k výpočetní náročnosti se stále jedná o problém, který i v aktuální době může benefitovat z moderních výpočetních možností, především z vysoké paralelizace. Program je poměrně rozsáhlý ve svých možnostech, implementující vícero různých výpočetních metod s podporou různých formátů dat. Na vývoji se v nějaké míře podílelo větší množství lidí (i v rámci některých dalších závěrečných pracích) což mělo negativní dopad na celkovou strukturu a konzistenci aplikace. Z toho důvodu vznikla tato práce, v rámci které bude program předělán pro jednodušší budoucí vývoj a jednodušší použití, spolu s rozšířením o některé nové funkce.

Osobní motivací pro tuto práci byla možnost získat zkušenosti s prací na rozsáhlejším neznámém projektu, který je třeba vzhledem k celkovému přepracování kompletně pochopit. Zároveň projekt nabízí množství oblastí, kde je možné uplatnit vlastní schopnosti a kreativitu, od netechnického návrhu uživatelského prostředí po reorganizaci a reimplementaci zdrojového kódu, s důrazem i na zachování výkonu. V neposlední řadě byl motivací zájem o naučení se více o výpočtech na grafických kartách, se kterými jsem měl dosud minimální zkušenosti.

Kapitola 1

Cíle práce

Cílem práce je seznámit se s aplikací a problémovou doménou a na základě provedené analýzy aplikaci přepracovat a rozšířit.

Úvod do problematiky krystalografie a práškové difrakce, které souvisí s aplikací, je zpracován v kapitole 2. Analýze současného stavu aplikace se věnuje sekce 3.1, spolu s popisem zdrojového kódu a s návrhy k refaktorizaci.

Pro jednodušší sestavování aplikace (podporu více operačních systémů, integraci s používanými technologiemi, možnost konfigurace) byla do projektu implementován sestavovací systém. Analýze dostupných možností se věnuje sekce 3.2.

Pro některé výpočty jsou v projektu používány grafické procesory pomocí technologie CUDA. V rámci této práce jsou grafické výpočty zobecněny a re-implementovány pomocí hardwarově nezávislých technologií. Představením a analýzou alternativních rozhraní se věnuje sekce 3.3, implementační detaily se pak nacházejí v sekci 5.2.

Aktuální aplikace nabízí pouze prostředí z příkazové řádky. Dalším cílem práce je vytvoření alternativního grafického prostředí pro usnadnění práce s používanými soubory a spouštění výpočtů. Analýze požadavků kladených na uživatelské prostředí se věnuje sekce 3.5, návrh prostředí je popsán v sekci 4.4 a detaily implementace v sekci 5.3.

Pro zlepšení kvality kódu byly implementovány některé nástroje pro statickou analýzu zdrojového kódu. Rešerše nástrojů byla provedena v sekci 3.6, detailům implementace spolu s přepracováním kódu se věnuje sekce 5.4.

Testování výsledné aplikace se věnuje kapitola 6. Pro hlavní výpočetní a grafickou část aplikace byly vytvořeny automatické testy. Dále pro ověření kvality grafického prostředí bylo rovněž provedeno uživatelské testování.

Kód z nových i původních částí aplikace byl zdokumentován, s popisem v sekci 5.5. Pro uživatelské použití byla vytvořena instalační B a uživatelská C příručka.

Kapitola 2

Představení problematiky

Program *ParaCell* slouží k analýze krystalické struktury látek pomocí dat získaných metodou práškové difrakce.

Pro vysvětlení, co a jak program dělá, je tato kapitola rozdělena do dvou hlavních částí: první je úvod do krystalografie, ve které budou vysvětleny základní pojmy a co přesně se program snaží řešit. V druhé části bude popsána metoda práškové difrakce, především *jak* aplikace funguje.

Pro lepší orientaci jsou hlavní symboly používané v této kapitole zapsány do tabulky 2.1. Tabulka slouží jako rychlá reference, všechny symboly jsou podrobněji představeny v samotném textu.

$\vec{a}$, $\vec{b}$, $\vec{c}$	Vektory primitivní buňky
a, b, c	Délky vektorů primitivní buňky
α, β, γ	Úhly mezi vektory primitivní buňky
h, k, l	Millerovy indexy, definující osnovu rovin pro primitivní buňku
d_{hkl}	Vzdálenost mezi rovinami osy definované indexy hkl
$\vec{a^*}$, $\vec{b^*}$, $\vec{c^*}$	Reciproké vektory zkonstruované z vektorů primitivní buňky
$\vec{d^*}_{hkl}$	Bod reciproké mřížky osnovy definované indexy hkl. Pro velikost vektoru platí $\lvert\vec{d^*}_{hkl}\rvert = \frac{1}{d_{hkl}}$

Tabulka 2.1: Symboly používané v této kapitole.

2.1 Úvod do krystalografie

Krystalografie je obecně vědní obor zabývající se krystalickými látkami. Krystaly jsou pevné látky, jejichž vnitřní struktura je pravidelně uspořádána (na rozdíl od amorfních, nahodile strukturovaných látek). Krystalická struktura má pak výrazný vliv na fyzikální vlastnosti látky, typickým příkladem je ani-

zotropie, kdy se vlastnosti látky mohou měnit v různých směrech. Informace v této sekci jsou čerpány ze zdrojů [1] a [2].

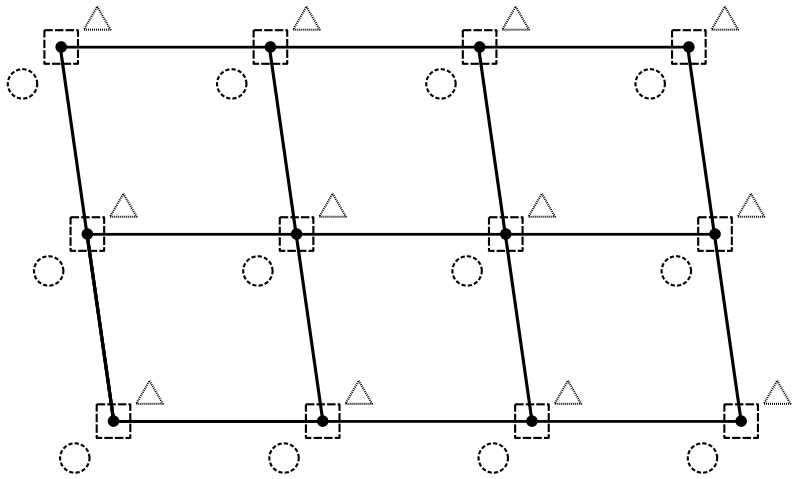
2.1.1 Krystalové mřížky

Krystalové mřížky slouží pro abstrakci pravidelné vnitřní struktury krystalů. Základní metodou pro vytvoření pravidelného vzoru je translační opakování, tedy že nějaký vzor (skupina atomů, molekul...) se pravidelně opakují ve všech směrech, s nějakých konstantním posunutím. Translační periodicitu je možné popsat pomocí *prostorové mřížky*, kde jsou opakující se vzory abstrahovány jako jednotlivé body (příklad je ukázán na obrázku 2.1).

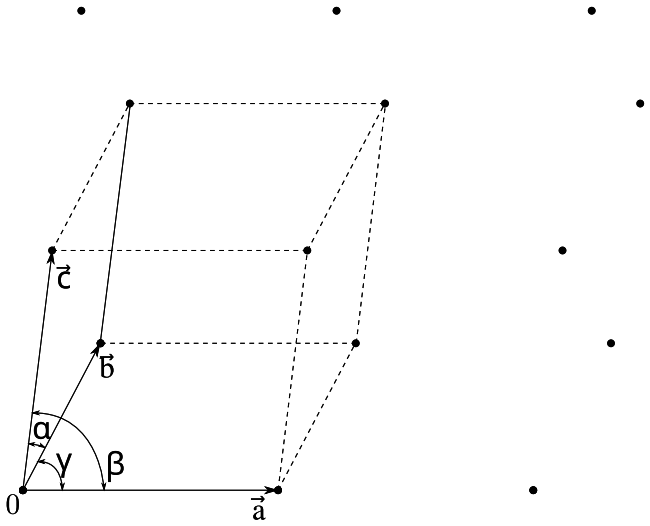
Pro vyjádření struktury krystalu se používají *Bravaisovy mřížky*. Ty je možné reprezentovat pomocí 3 vektorů ($\vec{a}$, $\vec{b}$, $\vec{c}$) takových, že každý bod mřížky (značen T) lze vyjádřit jako součet celočíselných násobků těchto vektorů (vektory tedy tvoří lineárně nezávislý soubor a každý bod mřížky lze vyjádřit jako jejich celočíselnou lineární kombinaci).

$$T = u \cdot \vec{a} + v \cdot \vec{b} + w \cdot \vec{c}$$

Rovnoběžnostěn definovaný těmito vektory se nazývá primitivní (nebo elementární) buňkou, vizualizovanou na obrázku [2.2]. Délky těchto vektorů ($a = |\vec{a}|$, $b = |\vec{b}|$, $c = |\vec{c}|$), spolu s úhly mezi nimi ($\alpha = \angle(\vec{b}, \vec{c})$, $\beta = \angle(\vec{a}, \vec{c})$, $\gamma = \angle(\vec{a}, \vec{b})$) tvoří *mřížkové parametry*. Přímky vytyčené jednotlivými vektory se označují jako osy buňky, jejich průsečík jako počátek (značen 0).



Obrázek 2.1: Příklad mřížky zobecněné na 2 dimenze. Skupiny tvarů (kruh, čtverec, trojúhelník) představují nějakou opakující se strukturu. Jednotlivé body mřížky pak tuto strukturu abstrahují. Převzato z [2, Obr. 2.3], vlastní zpracování.



Obrázek 2.2: Příklad prostorové mřížky s vyznačenou primitivní buňkou. Mimo bodů samotné buňky jsou viditelné i další body mřížky, které je možné popsat celočíselnou lineární kombinací vektorů. Převzato z [2, Obr. 2.8], vlastní zpracování.

Každou mřížku můžeme ekvivalentně popsat vícero (nekonečně) různými primitivními buňkami podle výběru vektorů, nicméně každá buňka jednoznačně definuje příslušnou krystalickou mřížku.

2.1.2 Krystalové soustavy

Podle souměrnosti se prostorové Bravaisovy mřížky dělí do 14 různých kategorií. Každá z nich zároveň spadá pod jednu ze 7 krystalových soustav, uvedených v tabulce 2.2. Ty kategorizují krystaly podle souměrnosti samotné krystalické struktury.

Do nejobecnější triklinické soustavy tak spadá každý krystal, zatímco o ostatních soustavách lze uvažovat jako o speciálních případech triklinické soustavy. Například znalost, že nějaký krystal patří do kubické soustavy, poskytuje výrazně více informací, než když víme, že je monoklinický. Do budoucna důležitý je fakt, že požadavky jednotlivých soustav lze vyjádřit podmínkami na jednotlivé mřížkové parametry.

2.1.3 Osnovy rovin

Pro každou mřížku se zvolenou buňkou můžeme definovat nekonečný počet *osnov rovin*. Každá osnova je množinou rovnoběžných rovin, kdy každé 2 sousední roviny mají od sebe stejnou vzdálenost, označovanou d.

Uvažujme osnovy takové, že jedna z rovin prochází počátkem, a další rovina nejblíže počátku vytýká na jednotlivých osách úseky délky $\frac{a}{h}$, $\frac{b}{k}$, $\frac{c}{h}$, kde h, k, l

Krystalová soustava	Omezení parametrů
Triklinická	-
Monoklinická	$\alpha = \gamma = 90°$
Ortorombická	$\alpha = \beta = \gamma = 90°$
Romboedrická	$a = b = c$ $\alpha = \beta = \gamma$
Tetragonální	$a = b$ $\alpha = \beta = \gamma = 90°$
Hexagonální	$a = b$ $\alpha = \beta = 90°$, $\gamma = 120°$
Kubická	$a = b = c$ $\alpha = \beta = 90°$

Tabulka 2.2: Přehled krystalových soustav spolu s kladenými podmínkami na mřížkové parametry.

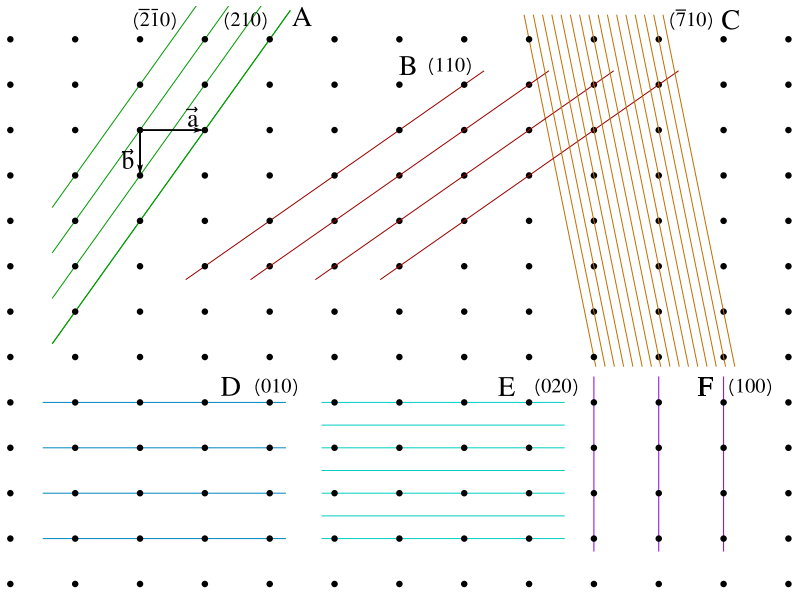
jsou celá čísla. Tuto trojici čísel nazýváme *Millerovými indexy* a značí se (hkl). Indexy lze interpretovat i tak, že jednotlivé roviny osnovy dělí vektory $\vec{a}$, $\vec{b}$, $\vec{c}$ na h, k, l stejně dlouhých částí. Trojice indexů jednoznačně určuje celou osnovu rovin (značení d_{hkl} tak do budoucna bude označovat mezirovinnou vzdálenost osnovy definované danými indexy). Příklad osnov je viditelný na obrázku 2.3.

V případě, že je rovina rovnoběžná s některou z os, se příslušný index definuje jako 0. Pokud rovina na nějaké ose vytýká záporný úsek (tedy pokud směr od počátku k průsečíku je opačný než směr příslušné osy), je daný index záporný, značeno například $\bar{h}$. Z požadavků na osnovu zároveň platí, že nejblíže počátku jsou 2 různé roviny, ze kterých vzejdou stejné indexy, ale s opačným znaménkem. Z toho vyplývá, že indexy (hkl) a $(\bar{h}\bar{k}\bar{l})$ popisují stejnou osnovu.

Pro libovolné indexy (hkl) můžeme získat novou osnovu přenásobením každého indexu stejným celým číslem n $(nh; nk; nl)$. Tato nová osnova bude obsahovat všechny roviny předchozí osnovy, pouze mezi každé dvě původně sousední roviny bude vloženo $n-1$ rovin nových. Z toho vyplývá, že vzdálenost mezi rovinami bude menší, vyjádřitelná vztahem $d_{nh;nk;nl} = \frac{d_{hkl}}{n}$.

2.1.4 Reciproká mřížka

Pro budoucí výpočty je důležitým (byť na první pohled značně abstraktním) konceptem takzvaná *reciproká mřížka*. Mějme nějakou primitivní buňku, definující příslušnou standardní mřížku (zvanou přímou). Dále mějme množinu



Obrázek 2.3: Příklad různých osnov rovin na dvojrozměrném řezu pro vybranou primitivní buňku. Vektor $\vec{c}$ je kolmý na nákresnu a všechny osnovy jsou s ním rovnoběžné, tedy index l je vždy 0. Na osnově A je viditelný ekvivalentní zápis pomocí negovaných indexů. Na osnovách D a E je znázorněno vynásobení indexů stejným číslem. Převzato z [1, obrázek B.8], vlastní zpracování.

osnov rovin definovanými všemi různými kombinacemi indexů (hkl). Každé takové osnově odpovídá 1 bod v reciproké mřížce, který leží na kolmici osnovy (= kolmici její libovolné roviny) procházející počátkem mřížky. Bod osnovy leží na této kolmici ve vzdálenost $\frac{1}{d}$ od počátku, v protisměru k osám mřížky. Reciproká mřížka je tvořena všemi takto zkonstruovatelnými body.

Ekvivalentně lze reciprokou mřížku zkonstruovat pomocí primitivní reciproké buňky, určenou vektory $\vec{a^*}$, $\vec{b^*}$ a $\vec{c^*}$. Ty jsou definovány jako vektory od počátku k reciprokým bodům osnov (100), (010) a (001). Vektor $\vec{a^*}$ je tedy kolmý na rovinu definovanou přímými vektory $\vec{b}$ a $\vec{c}$ a má velikost rovnou $\frac{1}{d_{100}}$. Ekvivalentně jsou definovány i ostatní dva vektory.

Pomocí nich je možné libovolný bod reciproké mřížky popsat následovně:

$$\vec{d^*}_{hkl} = h \cdot \vec{a^*} + k \cdot \vec{b^*} + l \cdot \vec{c^*}$$

Tedy, reciproký bod příslušící osnově s indexy (hkl) lze zkonstruovat lineární kombinací těchto vektorů.

2.2 Prášková difrakce

V předchozí sekci byly představeny základy krystalografie, spolu s možností, jak popsat krystalickou mřížku: nalezení příslušné primitivní buňky, která je specifikována svými 6 parametry.

Krystalická struktura některých látek může být natolik velká, že je jasně rozlišitelná pouhým okem. Na druhé straně jiné látky mohou obsahovat velké množství extrémně malých, různě orientovaných krystalů. V takovém případě se hovoří o prášku, s velikostí zrn v řádu mikrometrů. V této části bude představen způsob analýzy takto takových látek pomocí rentgenového záření. Použity budou zdroje [1], [3] a [4].

2.2.1 Braggova rovnice

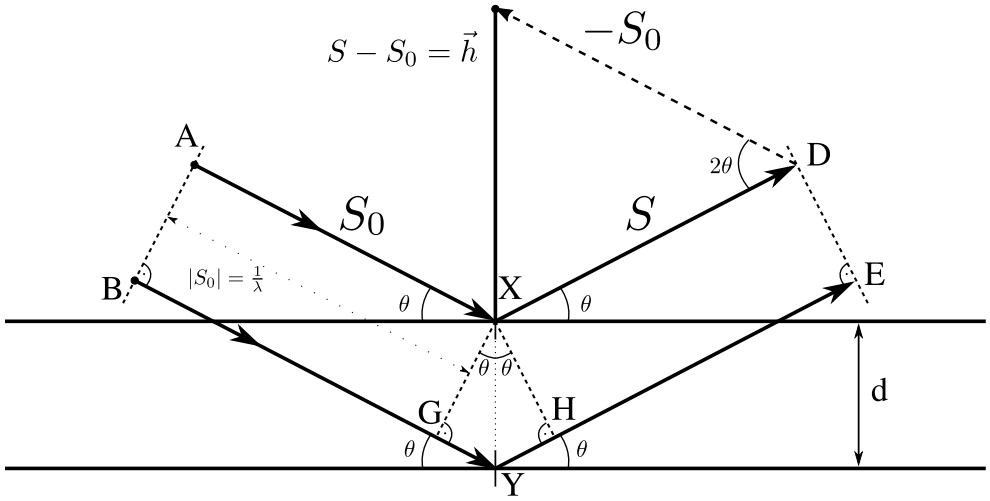
Pro jednoduchost budeme uvažovat ideální, nekonečný krystal. Základní princip spočívá ve vysílání paprsků rentgenového záření na daný krystal. Při průchodu záření látkou se každý atom stane *vysílačem* elektromagnetického záření se stejnou vlnovou délkou (předpokládáme elastický rozptyl, kdy nedojde ke ztrátě energie). Vysílané záření z jednotlivých atomů spolu interferuje a v ideálním případech, kdy mají obě vlnění stejný fázový posun, dojde ke konstruktivní interferenci (zvýšení amplitudy). Samotný pojem *difrakce* označuje konstruktivně se skládající vlnění, která se při průchodu překážkou (zde krystalem) ohýbají od svého původního směru.

Na zařízení měřícím amplitudy odražených paprsků (obecně *difraktometr*) je možné změřit směry, ve kterých je intenzita nejvyšší a ve kterých tedy došlo ke konstruktivnímu skládání. Vzhledem k pravidelnosti krystalu můžeme, pokud známe jeho strukturu, tyto směry vypočítat. Na druhé straně ale můžeme z naměřených výsledků rovněž zpětně vypočítat parametry krystalové mřížky.

Pro ilustraci, kdy ke skládání dojde, budeme používat krystalickou mřížku a zjednodušeně budeme předpokládat odraz od bodů této mřížky. Pokud bychom měli pouze jednu rovinu bodů mřížky, dojde ke konstruktivnímu skládání, pokud se úhel odrazu rovná úhlu dopadu. V případě trojrozměrné struktury je nicméně nutné zjistit případ, kdy dojde ke skládání s body ležícími na různých paralelních rovinách. K tomu nám poslouží dříve zavedená osnova rovin (předpokládáme, že na každé rovině osnovy leží body krystalické mřížky).

Příklad je viditelný na obrázku 2.4, kde jsou zvýrazněny 2 paralelní paprsky, které se odrazí od bodů X a Y ležících na mřížce v sousedních rovinách[1]. Předpokládáme, že oba paprsky jsou v bodech A a B ve stejné fázi, a cílem je zajistit, aby byly ve fázi i odražené paprsky v bodech D a E. K tomu použijeme vlnovou délku, λ, která značí vzdálenost uraženou vlněním během

[1] Daný příklad předpokládá, že body v sousedních rovinách leží „pod sebou“ (přímka jimi vedená je kolmá na osnovu). Aplikací dalších trigonometrických pravidel se však dá ukázat, že pravidlo platí i v případě, že jsou body mezi sebou v rovinách posunuty.



Obrázek 2.4: Příklad Braggovy rovnice. Převzato z [1, Obr. B.5] a [3, Fig. 1.6], vlastní zpracování.

jedné periody. Druhý paprsek oproti prvnímu navíc urazí úseky $\overrightarrow{GY}$ a $\overrightarrow{YH}$. Aby byla zajištěna stejná fáze, musí druhý paprsek mezi tímto úsekem urazit dráhu rovnou nějakému celočíselnému násobku své vlnové délky. Musí tedy platit $n \cdot \lambda = |\overrightarrow{GY}| + |\overrightarrow{YH}|$. Vzdálenost mezi rovinami je dána hodnotou d. Pokud délky vektorů vyjádříme pomocí funkce sinus, dostaneme *Braggovu rovnici*:

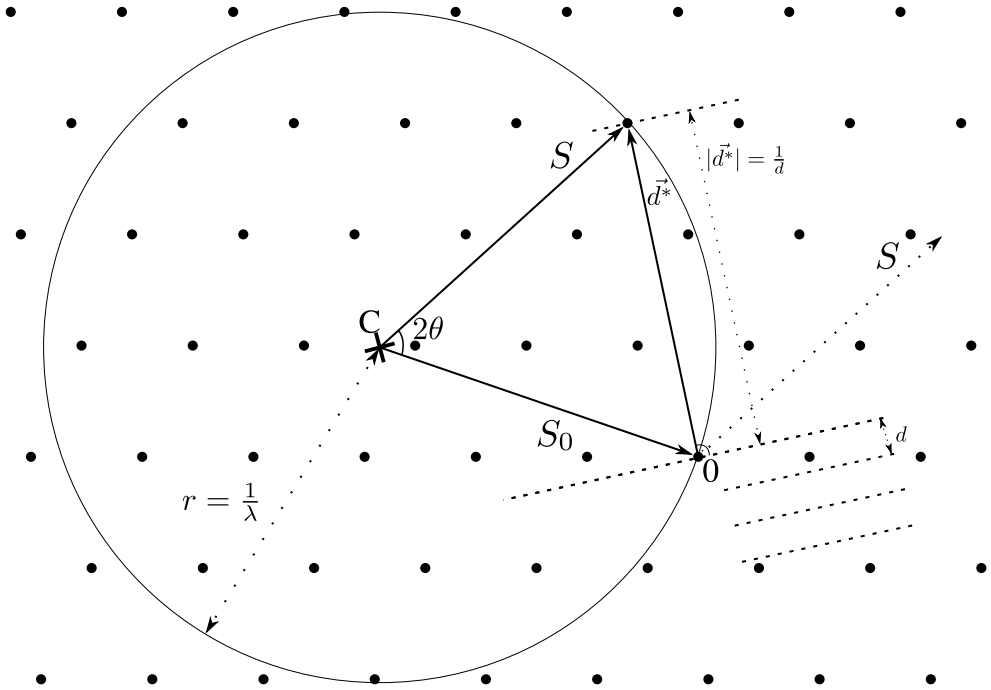
$$n \cdot \lambda = 2d \cdot sin(\theta) \tag{2.1}$$

Pro praktické použití převedeme Braggovu rovnice na tvar, který ji prováže s dříve zavedenou reciprokou mřížkou. Na přechozím obrázku je viditelný vektor h, definovaný jako rozdíl příchozího vektoru S_0 od odraženého vektoru S. Vektor h je potom kolmý na odrazovou rovinu a platí $sin(\theta) = \frac{0.5|h|}{|S_0|}$, z čehož plyne $2sin(\theta) = \frac{|h|}{|S_0|}$. Z Braggovy rovnice potom platí $\frac{n \cdot \lambda}{d} = 2sin\theta = \frac{|h|}{|S_0|}$. Pokud definujeme velikost vektoru S_0 jako $\frac{1}{\lambda}$, rovnice se redukuje na $|h| = \frac{n}{d}$.

V předchozí části jsme pro každou osnovu rovin s indexy (hkl) definovaly odpovídající bod v reciproké mřížce jako vektor $\vec{d^*}_{hkl}$ kolmý na rovinu s délkou $d^*_{hkl} = \frac{1}{d_{hkl}}$. Pokud tedy dojde k odrazu podle roviny (hkl), podle předchozí rovnice platí $h = n \cdot \vec{d^*}_{hkl}$.

Pravou stranu rovnice můžeme zjednodušit přenásobení faktorem n. Jak bylo zmíněno při definici osnov rovin, pokud každý z indexů přenásobíme celým číslem n, získáme osnovu stejně směrovanou, s mezirovinnou vzdáleností vydělenou n. Pro reciproký bod takto přenásobené osnovy platí:

$$|\vec{d^*}_{nh;nk;nl}| = \frac{1}{d_{nh;nk;nl}} = \frac{n}{d_{hkl}} = n \cdot |\vec{d^*}_{hkl}|$$



Obrázek 2.5: Ewaldova konstrukce. Převzato z [1, Obr. B.9] a [3, Figure 1.8], vlastní zpracování.

Ze stejného směru obou osnov potom platí $\vec{d^*}_{nh;nk;nl} = n \cdot \vec{d^*}_{hkl}$. Rovnici vektoru h tedy můžeme přepsat jako $h = \vec{d^*}_{nh;nk;nl}$.

Zjednodušeně tedy můžeme říci, že ke konstruktivní interferenci podle roviny indexované (hkl) dojde tehdy, pokud námi definovaný vektor h odpovídá reciprokému bodu této roviny.

Tuto vlastnost ilustruje takzvaná *Ewaldova konstrukce* (*Ewaldova koule*), naznačená na obrázku 2.5. Vezměme nějakou krystalickou mřížku s počátkem v bodě 0 a vektor rentgenového záření S_0 dopadající na tento počátek. Pokud druhý konec vektoru označíme písmenem C, podle dřívějšího zavedení velikosti vektoru platí $|S_0| = \frac{1}{\lambda}$. *Ewaldova koule* je nyní zkonstruována z bodu C s poloměrem $\frac{1}{\lambda}$ (na obrázku zjednodušeno jako kružnice). Koule tedy prochází počátkem 0. Pokud z počátku zkonstruujeme reciprokou mřížku (všechny její body), je možné, že koule bude procházet některými z reciprokých bodů. Označíme jeden takový bod jako $\vec{d^*}$ a zavedeme vektor S jako $\vec{d^*} - C$. Pro bod platí:

$$\vec{d^*} = \vec{d^*} - 0 + C - C = (\vec{d^*} - C) - (0 - C) = S - S_0$$

Vektor S_0 reprezentuje příchozí paprsek, vektor S odrazový paprsek při odrazu od osnovy definované bodem $\vec{d^*}$.

Každý reciproký bod ležící na *Ewaldově kouli* tak určuje rovinu, podle které dojde k difrakci. Zároveň při rotaci krystalu (změně úhlu dopadu) dojde k rotaci reciproké mřížky a nové body se mohou stát difrakčními.

2.2.2 Srovnání krystalu s práškem

Doposud popisovaný způsob předpokládal chování na jediném krystalu. První pokusy s rentgenovým zářením skutečně používali celistvé krystaly[2].

Prášková difrakce nicméně neprovádí experimenty na jednotlivých krystalech, ale na látce s velkým množstvím malých, různě orientovaných krystalů. Při průchodu paprsku látkou tak efektivně může dojít k difrakci podle více krystalů najednou. Statisticky, při dostatečně velkém množství krystalů, je možné najednou zachytit všechny možné rotace a všechny možné roviny, podle kterých dojde k difrakci.

Nevýhodou oproti předchozímu záznamu je, že vzhledem ke všem možným rotacím jsou místo diskrétních bodů tvořeny soustředné kružnice. Tím dojde ke ztrátě informace: víme, pod jakými úhly dopadu došlo k difrakci, ale ztrácíme informaci o směru odraženého paprsku. Zároveň daný zkoumaný vzorek musí být dostatečně kvalitní: krystalů musí být dostatečný počet, aby byly reprezentované všechny možné orientace v podobném zastoupení.

2.2.3 Indexace

Pokud známe mřížku popisující krystal, je možné vypočítat úhly, pod kterými dojde ke konstruktivnímu skládání vlnění.

Indexace je proces opačný, tedy z experimentálně naměřených difrakčních dat se budeme snažit najít parametry dané buňky. V případě práškové difrakce máme k dispozici pouze úhly, pod kterými došlo k difrakci (upravené ve formě 2θ). Dále bude popsán postup, jak z naměřených úhlů vypočítat parametry primitivní buňky popisující mřížku krystalu.

Rovinu, podle které došlo k difrakci, můžeme označit Millerovými indexy (hkl), a vzdálenost mezi sousedními rovinami můžeme z Braggovy rovnice vypočítat $d_{hkl} = \frac{\lambda}{2sin\theta}$. Pro příslušný reciproký bod platí $|\vec{d^*}_{hkl}| = \frac{1}{d_{hkl}}$ (jak již bylo zmíněno).

Pro skalární součin vektoru se sebou samým platí, že se rovná kvadrátu své velikosti, tedy $(\vec{d^*}_{hkl})^2 = \frac{1}{(d_{hkl})^2}$. Zároveň podle vektorů primitivní reciproké buňky platí $\vec{d^*}_{hkl} = h \cdot \vec{a^*} + k \cdot \vec{b^*} + l \cdot \vec{c^*}$, po skalárním součinu:

$$(\vec{d^*}_{hkl})^2 = h^2\vec{a^*}^2 + k^2\vec{b^*}^2 + l^2\vec{c^*}^2 + \\ + 2hk \cdot \vec{a^*} \cdot \vec{b^*} \cdot cos(\gamma^*) + 2hl \cdot \vec{a^*} \cdot \vec{c^*} \cdot cos(\beta^*) + 2kl \cdot \vec{b^*} \cdot \vec{c^*} \cdot cos(\alpha^*)$$

Zde již máme závislost na všech parametrech reciproké buňky (ze kterých lze získat parametry přímé buňky). Dále se pro tento výpočet bude používat

[2] za první pokus v roce 1912 získal Max Theodor Felix von Laue Nobelovu cenu.

následující rovnice:

$$Q_{hkl} = h^2 A_{11} + k^2 A_{22} + l^2 A_{33} + hkA_{12} + hlA_{13} + klA_{23} \tag{2.2}$$

kde parametry A_{xy} pouze přeznačují výrazy z předchozí rovnice. Do budoucna se pro jednotlivé parametry složené z indexů bude používat značení p_{xy}, podle příslušnosti k parametrům A_{xy} (například $p_{11} = h^2$ pro $A_{11} = \vec{a^*}^2$).

Takovou rovnici je třeba sestavit pro každý naměřený úhel, podle kterého došlo k difrakci („peak“). Získáme tak soustavu rovnic. Zatímco parametry A_{xy} určují parametry samotné buňky a jsou pro každou rovnici stejné, pro každou z nich je třeba navíc najít individuální indexy hkl (odtud název indexace).

Vzhledem k experimentálnímu zdroji dat je náročné přesně určit jednotlivé parametry. Naměřené pozice difrakcí se mohou překrývat, samotné naměřené úhly mohou být mírně nepřesné. Pro data se povětšinou používá prvních 20 až 30 pozorovaných difrakcí, větší hodnoty úhlů jsou více citlivé na změny parametrů.

2.2.4 Figures of merit

Pro vyhodnocení, zdali nalezené kandidátní primitivní buňky dobře odpovídají naměřeným datům a která z nich buněk je nejlepší, se používají hodnotící funkce nazývané *figures of merit*.

První funkcí je *de Wolffovo kritérium*, značené M_{20}, která je definováno následovně:

$$M_{20} = \frac{Q_{20}}{2\epsilon N_{20}}$$

Kde Q_{20} je hodnota dvacáté indexované difrakce, ϵ je průměrný absolutní rozdíl mezi naměřenými a vypočtenými hodnotami Q u prvních 20 pozorovaných difrakcí a N_{20} značí počet různých vypočtených hodnot Q mezi prvními 20 pozorováními. Větší hodnota značí lépe odpovídající buňku, tedy nepřekvapivě je cílem minimalizovat ϵ a N_{20}. N_{20} slouží k preferenci buněk s menším objemem. Kritérium je silně svázáno s počtem 20 difrakcí, který byl zvolen jako kompromis z praxe.

Alternativou je *Smith/Snyder* index F_n, které počítá s libovolným počtem n difrakcí. Definován je následovně:

$$F_n = \frac{1}{|\delta \cdot 2\theta|} \cdot \frac{N}{N_p}$$

kde $(|\delta \cdot 2\theta|)$ je průměrný absolutní rozdíl mezi naměřenými a vypočtenými úhly 2θ.

2.2.5 Výpočetní metody

Z vytvořené soustavy rovnic není možné hledat řešení nějakým jednoznačným způsobem. V průběhu let vznikly různé heuristické metody, které přistupují odlišně k problému hledání s správných kombinací mřížkových parametrů a indexů. Metody se mohou výrazně lišit dobou běhů a kvalitou výsledků, rovněž mohou být různě závislé na kvalitě naměřených dat. Vzhledem k odlišnému přístupu každé z metod není možné některou z nich prohlásit objektivně za nejlepší. Konkrétní příklady jsou popsány v sekci 3.1.3, která se věnuje implementovaným metodám v programu.

Kapitola 3

Analýza

3.1 Stávající řešení

3.1.1 Workflow

Aplikace je používána skrze příkazovou řádku, kdy každé spuštění provádí jeden výpočet podle specifikovaných argumentů. Program očekává 3 argumenty: cestu ke konfiguračnímu souboru, cestu k souboru se vstupními daty a dobrovolně cestu k souboru pro výstupní záznamy.

Následuje popis jednotlivých fází výpočtu, znázorněných na 3.1 ve formátu UML activity diagramu ([5, kapitola 16]).

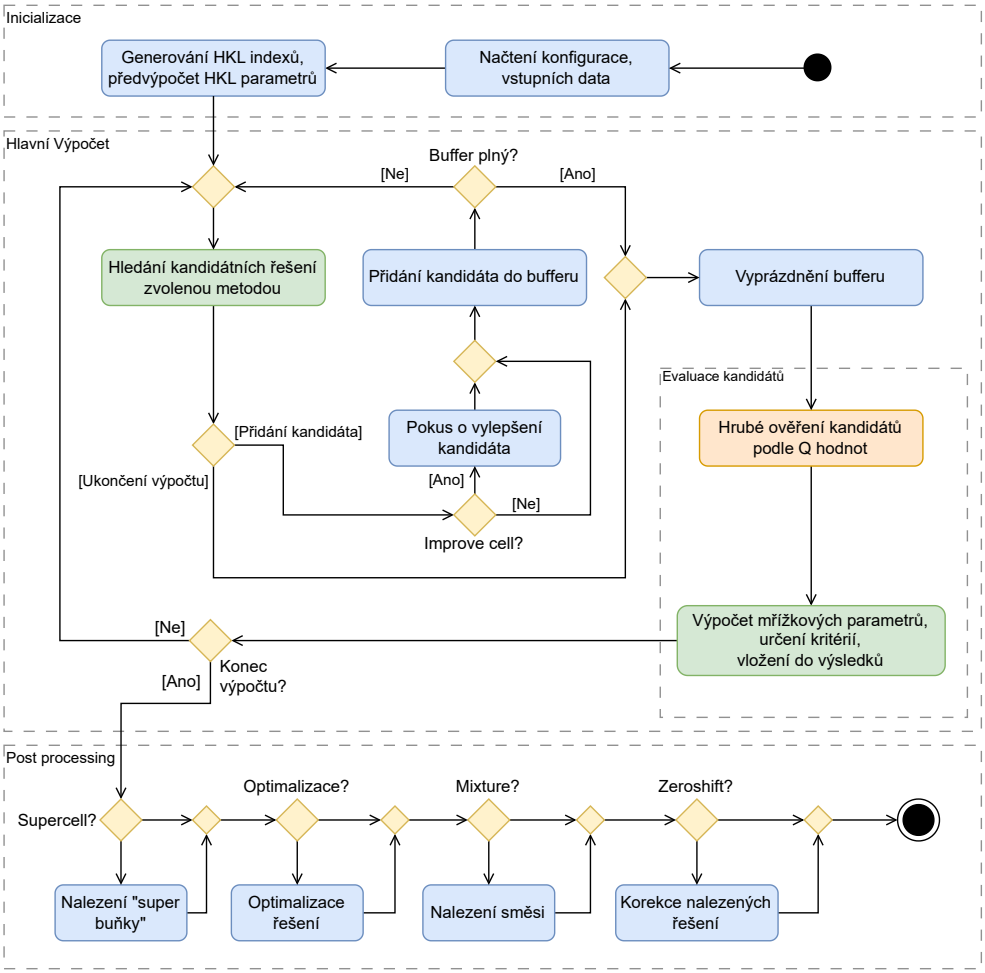
- V inicializační fázi jsou nejdříve z předaných argumentů načteny vstupní konfigurační a datové soubory. Konfigurační soubor kompletně specifikuje požadavky na výpočet, tedy není možné konfigurovat výpočet přes příkazovou řádku. Podrobně je formát konfiguračních a datových souborů popsán v další sekci.

- Před zahájením výpočtu je ještě vygenerována množina indexů hkl, ve které se budou hledat odpovídající trojice pro nalezené záznamy. Aplikace rovněž podporuje dobrovolnou specifikaci vstupního souboru s definicí indexů (které poté nejsou generovány).

- Pro každou vygenerovanou trojici indexů jsou následně předvypočítány příslušné parametry p_{xy} z rovnice 2.2 (h^2, k^2, ...).

- Po připravení indexů následuje hlavní výpočetní část, tedy snaha o nalezení trojic indexů pro každou pozorovanou difrakci ze vstupních dat a nalezení korektních parametrů krystalické mřížky.

  Hlavní výpočet se v závislosti na konfiguraci větví podle dvou hlavních podmínek: jednak podle specifikované metody a dále se každá metoda větví podle krystalové soustavy.



Obrázek 3.1: Průběh výpočtu v programu.

- Během výpočtu metoda vytváří kandidátní řešení, specifikovaná parametry A_{xy}, která vkládá do paměti se statickou velikostí („buffer“). Nalezení kandidáti mohou být ještě před vložením upraveni, ve snaze zlepšit jejich kvalitu (pokud to bylo vyžádáno v konfiguraci).

- Po zaplnění paměti (nebo po dokončení výpočtu, pokud v paměti ještě zbývají nezpracovaní kandidáti) proběhne fáze, kdy se pro každého kandidáta ověří, jestli dobře odpovídá všem naměřeným datům. Tato část výpočtu je výpočetně nejnáročnější: pro každého kandidáta se iteruje přes všechny vygenerované trojice hkl, pro kterou se vypočte hodnota Q z rovnice 2.2. Následně se iteruje přes všechny pozorované difrakce a porovnávají se naměřené a vypočtené hodnoty Q.

V ideálním případě se pro každou naměřenou pozici difrakce najde odpovídající trojice indexů. Kandidáti, kterým se nepodaří najít alespoň specifikovaný limit dobře odpovídajících difrakcí, jsou vyřazeni.

- Pro zbývající ověřené kandidáty jsou nalezené parametry A_{xy} převedeny do přímé mřížky a jsou vypočtena kritéria M_{20} a F_n. Řešení je poté přidáno do „databáze“, instance struktury, ve které je ukládán daný počet nejlepších řešení (podle vypočtených kritérií) a zároveň jsou eliminována duplikátní řešení. Každé z vláken obsahuje vlastní instanci, které jsou po dokončení výpočtu sjednoceny dohromady.

- Po dokončení hlavního výpočtu mohou podle konfigurace proběhnout ještě dodatečné úpravy, na diagramu zobrazené ve spodním rámečku.

  „Superbuňka“ (supercell) nějaké buňky popisuje stejnou krystalickou mřížku, ale s menším objemem (tedy, délky všech stran jsou vydělené stejným faktorem). Tato možnost se pokusí najít superbuňku u nejlepšího řešení.

  Optimalizace se u několika nejlepších řešení (počet je konfigurovatelný) snaží různými metodami vytvářet nová (ideálně kvalitnější) řešení.

  Dále může proběhnou pokus o nalezení 2 různých buněk, pokud byla vstupní data naměřena na směsi 2 rozdílných látek. Část vstupních dat tak odpovídá jedné látce, zatímco ostatní odpovídají látce druhé. V průběhu výpočtu jsou přidávány řešení odpovídající rozumné části dat a na konci se z nejlepších řešení pokusí vybrat pár, který by dohromady dobře odpovídal všem vstupům.

  Zeroshift hodnota specifikuje možnou odchylku naměřených úhlů od jejich skutečných hodnot. V konfiguračním souboru může být specifikována hodnota vyjadřující možné rozpětí. V takovém případě se program pokusí opravit nalezená řešení postupným upravováním vstupních úhlů, s cílem maximalizovat hodnotu F_n.

3.1.2 Konfigurační soubory

Konfigurační soubory mají jednoduchou podobu ve formátu *klíč = hodnota* na každé řádce. Rovněž je možné zapsat komentáře pomocí symbolu % (po zapsání je ignorován zbytek řádky).

V této části budou popsány hlavní konfigurovatelné klíče a co reprezentují. Mnoho z nich je nepovinných nebo relevantních pouze pro některé metody, pro rychlý přehled je celková specifikace dostupná v příloze D.

- DEVICE určuje, zdali má být použitý pouze hlavní procesor (CPU, výchozí hodnota), nebo jestli má být pro některé výpočty použit CUDA akcelerátor (GPU).

- METHOD specifikuje metodu, která má být použita pro výpočet, jejím textovým názvem. Program podporuje 5 různých metod (které budou popsány v další sekci) a některé metody navíc mají více verzí. Dohromady existuje 12 možností, definovaných ve specifikaci.
- SYSTEM udává krystalovou soustavu, ve které má být hledána cílová elementární buňka. Soustavy byly již dříve popsány v tabulce 2.2. Hodnota systému je zapsána číselně:

  0. Romboedrická
  1. Kubická
  2. Hexagonální
  3. Tetragonální
  4. Ortorombická
  5. Monoklinická
  6. Triklinická

  Výběr soustavy je jistým kompromisem mezi množinou výsledků a dobou běhu. Vyhledávání kubických mřížek je výrazně rychlejší, vzhledem k nutnosti nalézt pouze jeden parametr, ovšem pravděpodobnost, že reálná mřížka je skutečně kubická, je poměrně malá. Oproti tomu obecné monoklinické a triklinické mřížky budou odpovídat lépe, nicméně doba běhu je výrazně delší.

  Toto nastavení se rovněž používá pro specifikaci, zdali má být hledána směs, tedy 2 primitivní buňky pro různé látky. Vyhledávání lze povolit zdvojením čísla mřížky (např. místo *SYSTEM=6* napsat *SYSTEM=66*), obě buňky jsou vyhledávány ve stejné soustavě. Tento způsob nastavení je poměrně matoucí a do budoucna bude upraven do specifické nastavitelné hodnoty.
- Je možné omezit číselné rozsahy, ve kterých má být hledána buňka. Možné je nastavit podmínky na délku jednotlivých stran v angstromech (MINIM_A, MAXIM_A a analogicky pro strany b, c), na velikost úhlu ve stupních (MINIM_ANGLE, MAXIM_ANGLE zvolí společný rozsah pro všechny úhly) a na objem buňky (MINIM_V, MAXIM_V, rozsah musí dávat smysl vzhledem k ostatním omezením, aby vůbec mohla být nalezena buňka s daným objemem). Zadané rozsahy opět mají výrazný vliv na dobu běhu. Omezení na parametry, které nedávají smysl vzhledem ke zvoleném soustavě (např. omezení na úhel pro kubickou mřížku, která má všechny úhly pravé) jsou ignorovány.
- INPUT_ERR umožňuje specifikovat možnou malou chyby (ve stupních), která se zohlední v naměřených datech. Implicitně rovna 0, v takovém případě se použije chyba vypočítaná z poměru vstupní vlnové délky.

- LIMIT, RATIO a NON_INDEXED jsou vzájemně se vylučující nastavení umožňující omezit počet pozorovaných difrakcí ze vstupních dat, které budou brány v potaz. Jak bylo zmíněno, první pozorované difrakce s menšími úhly bývají přesnější a větší počet měření může mít negativní dopad na nalezení řešení, standardem je tedy se omezit na část naměřených dat.

  LIMIT s hodnotou přirozeného čísla přímo určuje počet použitých pozorování, ve výchozím nastavením 20.

  RATIO je hodnota v intervalu $(0; 1]$, která určí počet podle poměru k celkovému počtu naměřených dat. Například při hodnotě 0.5 stačí, aby kandidát dobře odpovídal polovině naměřených dat.

  NON_INDEXED funguje jako opak k LIMIT, specifikuje počet pozorování, které se nemusí správně oindexovat.

- THREADS umožňuje omezit počet vláken procesoru, která se použijí při výpočtu. Ve výchozí podobě se použijí všechna dostupná vlákna procesoru.

- DB_BEST_LIST nastavuje počet nejlepších řešení, která jsou při výpočtu uchovávána v každé instanci databáze (tedy pro každé vlákno). Standardně rovno 100.

- DB_DETAIL_LIST umožňuje omezit počet nejlepších řešení, u kterých jsou po výpočtu zapsány detailní výsledky. U všech nejlepších řešení jsou vypsány nalezené parametry, vypočtená kritéria a objem. Detailní výpis navíc pro každou pozorovanou difrakci obsahuje indexy rovin a hodnoty Q (vypočtené i naměřené).

- OPTIMIZE slouží k dodatečnému vylepšení výsledků po hlavní části výpočtu. Optimalizace probíhá na zvoleném počtu nejlepších řešení, z každého z nich jsou ve funkci vytvářeni různými způsoby další (potenciálně lepší) kandidáti. Výchozí hodnota 0 znamená, že optimalizace neprobíhá.

Následují některé binární možnosti, které je možné povolit nastavením hodnoty 1. Ve výchozím stavu jsou vypnuté:

- SUPERCELL, jestliže se má pro nejlepší řešení najít „super buňka“.

- IMPROVE_CELL povoluje mírné úpravy na všech řešeních (před jejich vložením do databáze), s cílem zlepšit jejich kvalitu.

Pro některé metody je možné specifikovat některé dodatečné hodnoty, ty jsou uvedeny ve specifikaci D a budou popsány v dále v následující sekci věnující se implementovaným metodám.

3.1.3 Podporované metody

Program implementuje 5 různých metod pro výpočet, které budou v této sekci představeny. Každá z nich odlišně generuje kandidáty, zbytek programu pak pro všechny metody funguje stejně. Metody povětšinou vychází z algoritmů publikovaných v různých vědeckých pracích. Kromě vlastní analýzy byly použity informace z článku [6].

GRID

Nejjednodušší metoda, která prohledává všechny kombinace parametrů v zadaných rozsazích. V konfiguračním souboru je možné nastavit počet kroků, které metoda vykoná pro každý parametr. Jednou z možností jsou nastavení změny, o kterou se parametr změní v každém kroku (*DELTA_LATTICE* pro změnu délek stran a *DELTA_ANGLE* pro změnu velikosti úhlů). Alternativně je možné specifikovat konkrétní počet kroků, které se mají pro parametr vykonat (tedy, ať už je interval jakýkoli, vždy se vykoná stejný počet kroků). Hodnoty *STEP1*, *STEP2* až *STEP6* nastavují počet kroků individuálně pro každý parametr, nebo je možné použít hodnotu STEPS, kterou použijí všechny parametry.

MGLS

Metoda funguje podobně jako GRID, tedy generováním parametrů mřížky ve specifikovaných intervalech podle diskrétních kroků. Oproti GRID metodě, která takto vytvořené kandidáty okamžitě předává k ověření, zkouší MGLS metoda pro každého kandidáta pro každou difrakci najít nejbližší odpovídající hodnoty indexů. Podle nich následně aproximuje kritérium F_n, ověří kvalitu kandidáta, a v případě nízké kvality jej může okamžitě vyřadit. Metoda umožňuje stejné nastavení v konfiguračních souborech jako metoda GRID.

TREOR

Metoda [7] se snaží najít kandidáty řešením maticové rovnice $M \cdot \vec{A} = \vec{Y}$, která se prakticky snaží vyřešit několik Q rovnic najednou. M je čtvercová matice, která po řádcích obsahuje indexové parametry hkl z původních rovnic, $\vec{A}$ je vektor obsahující postupně parametry A_{11}, až A_{23} a $\vec{Y}$ je vektor hodnot Q z naměřených dat.

Aplikace poté řeší rovnici $M^{-1} \cdot \vec{Y} = \vec{A}$, tedy vlastně získá požadované parametry z podmnožiny naměřených dat a později zkouší, jak dobře odpovídají zbytku. Aplikace zkouší vygenerovat několik matic M a pro každou z nich zkouší několik permutací naměřených dat (tedy, zkoušet přiřazovat indexy k jiné pozorované difrakci). Vzhledem k extrémnímu množství různých kombinací je generována pouze podmnožina.

DICHOTOMY

Metoda [8] funguje tak, že pro každý z hledaných parametrů rekurzivně půlí prohledávané rozsahy na menší podprostory (s až 6 dimenzemi v případě triklinické soustavy). Po rozdělení na dostatečně malé velikosti jsou kandidáti hledáni v rozsazích každého podprostoru, metoda však může pro každý podprostor ověřit, zdali v něm může řešení existovat, a případně jej vyřadí. Metoda je implementována v několika variantách: ve výchozí podobě je prohledáván prostor parametrů v přímé mřížce, potom však pro ověření kvality musí být dodatečně konvertovány do reciproké mřížky. Alternativně mohou být rovnou prohledávány a děleny prostory v reciprokých rozsazích (varianta DICHOTOMYR). Metoda rovněž podporuje vlastní vylepšení vytvořených kandidátů pomocí GPU výpočtů.

ITO

ITO metoda se nejprve snaží najít nadějné „zóny“, ve kterých by se mohlo nacházet řešení. Zóny jsou roviny takové, které mají jeden z indexů roven nule. Pro zóny je spočítána hodnota, určující jejich kvalitu (pro zónu se zvolí několik dvojic zbývajících indexů a zjistí se, jak dobře odpovídá naměřeným datům). Několik dostatečně kvalitních zón je následně prohledáváno společně pro nalezení kandidátních řešení – tento počet je v konfiguračním souboru možné změnit hodnotou *TOPZ*. Dále je možné nastavit hodnotu *Qe* jakožto zanedbatelný rozptyl při hodnocení zón (dvě zóny lišící se v hodnocení o méně jak *Qe* jsou brány jako jedna zóna).

3.1.4 Datové soubory

Datové soubory specifikují úhly, pod kterými byly naměřeny difrakce, spolu s dalšími informacemi. Aplikace rovněž podporuje 2 různé formáty, hlavní *.dat* formát a dodatečně formát *.dic*.

Příklad *.dat* vstupu je viditelný na 3.2. Řádky začínající * nebo *!* jsou ignorovány jako komentáře. Z prvního (nekomentovaného) řádku jsou načteny základní informace (vlnová délka, nepřesnost od 0). Následně je na každém řádku uvedena dvojice 2θ spolu s intenzitou (vzestupně podle hodnoty 2θ). Intenzita samotná není v programu používána.

Formát *.dic* vychází z programu *DICVOL91* [9]. Oproti prvnímu formátu umožňuje specifikaci většího množství parametrů, které mohou nahradit nastavení z konfiguračního souboru (např. krystalová soustava, omezení na parametry a objem buňky, ...). Celý formát je dostupný zde [10]. Hlavní data (hodnoty 2θ spolu s intenzitou) jsou po řádcích zapsána postupně jako v předešlém příkladu.

```
***   Cd3(OH)5(NO3)   J. Solid State Chem. 84, 58, 1990.  ***
! Wavelength, zeropoint and NGRID
1.54056 0.0 -3
! Couples of 2-theta and intensities
8.025  100.
11.925 100.
16.070 100.
18.340 100.
19.432 100.
23.970 100.
24.190 100.
25.794 100.
27.286 100.
27.530 100.
27.869 100.
28.716 100.
30.119 100.
30.733 100.
31.265 100.
32.447 100.
32.680 100.
33.686 100.
36.745 100.
36.967 100.
```

Obrázek 3.2: Příklad vstupního datového souboru.

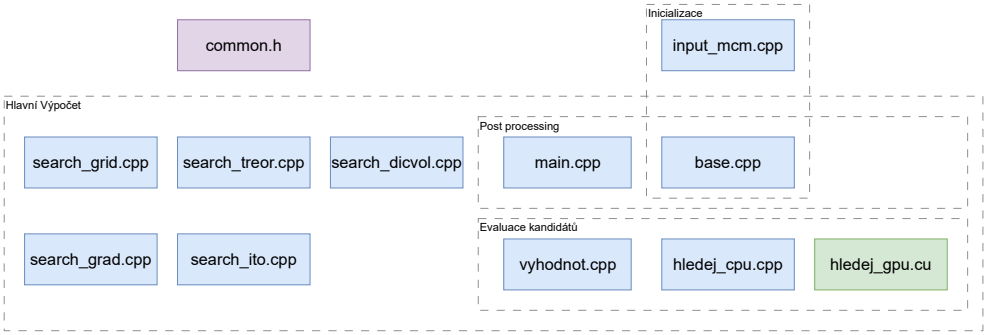
3.1.5 Aktuální kód

Součástí refaktorizace je celková analýza a úprava stávajícího kódu. Cílem je zajistit, aby kód napříč celou aplikací, mimo jiné, dodržoval konzistentní formátování, jmenné konvence a používal moderní možnosti jazyka C++. Celková úprava by měla zajistit lepší přehlednost a snazší budoucí rozvoj.

Pro základní přehled o struktuře kódu je na diagramu 3.3 zobrazena struktura zdrojových souborů. Pro ilustraci, jakou část výpočtu zajišťuje jaký soubor, byly jednotlivé soubory rozděleny do stejných skupin jako workflow diagram.

input_mcm.cpp zajišťuje načítání konfiguračních souborů, datových souborů a případné souborem definované *hkl* indexy.

base.cpp obsahuje různé funkce používané napříč projektem (matematické funkce, funkce pro výpis, vkládání do *databáze*). Rovněž implementuje generování *hkl* indexů spolu s předvypočítáním jejich parametrů a dále některé post processing funkce.



Obrázek 3.3: Přehled původních zdrojových souborů.

Každá z metod je implementovaná ve svém vlastním souboru (*search_dicvol*, *search_grid*, *search_grad* (MGLS), *search_treor*, *search_ito*).

vyhodnot.cpp zajišťuje zpracování kandidátů, od jejich přidávání do paměti po analýzu a vyhodnocení řešení. Pouze samotné hrubé vyhodnocení kandidátů je implementováno v souborech *hledej_cpu* a *hledej_gpu*, které implementují ověřovací logiku na CPU a GPU (CUDA).

common.h působí jako hlavní hlavičkový soubor pro celý projekt. V něm je deklarována většina hlavních datových struktur a funkcí.

Na aplikaci v minulosti pracovalo vícero lidí, což vedlo k celkové nekonzistenci kódu napříč různými částmi. Mezi hlavní problémy, které budou v rámci práce vyřešeny, jsou následující:

- Hlavičkový soubor *common.h* svou rozsáhlostí poměrně znepřehledňuje kód a výrazně zvyšuje dobu kompilace, vzhledem k nutnosti rekompilace téměř všech jednotek překladu při jakékoli změně hlavičkového souboru. Hlavní struktury by měli být vyhrazeny do vlastních souborů a deklarace funkcí přesunuty do hlavičkových souborů příslušných implementačních souborů.
- V kódu se často vyskytují konstrukty a konvence jazyka C. Projekt bude přepracován tak, aby lépe odpovídal standardu C++17.
- Pojmenovávání se značně liší napříč celým projektem, ať už základními konvencemi (snake_case vs CamelCase) nebo kombinace českého a anglického jazyka v některých částech. Kód bude upraven tak, aby byl celý v angličtině a co nejvíce konzistentní.
- Projekt obsahuje nepoužívaný kód, ať už v některých nepoužívaných souborech (*future.cu*), nebo v nepoužívaných zakomentovaných částech kódu.

Datové struktury

V této části budou představeny hlavní datové struktury používané napříč aplikací. Většina datových struktur je používáno ve stylu klasických struktur jazyka C, případné dodatečné chování pak zajišťují samostatně stojící funkce. Mezi hlavní struktury patří:

configuration zapouzdřuje všechny hodnoty načtené z konfiguračního souboru, spolu s některými dodatečnými proměnnými (logovací soubor, ladící proměnné pro profilování výpočetního času). Načtení dat ze souboru a jejich uložení do instance zajišťuje funkce *nacti_cfg*.

measured obsahuje načtená vstupní data (vlnová délka, zeroshift, dvojice úhlů 2θ s intenzitou), relevantní předvypočítané hodnoty (Q hodnoty ze všech naměřených dat a podle zadané vstupní chyby jsou určeny horní a dolní meze intervalů 2θ a Q hodnot, ve kterých se hledají řešení) a vygenerované trojice hkl indexů, spolu s předvypočítanými hodnotami p_{xy} z rovnice 2.2.

GPU_data primárně slouží pro dříve zmíněný *buffer* pro ukládání dat o vytvořených kandidátech. Dynamicky je alokována paměť pro parametry A_{xy} (*pole_al*), výstup z ověřovací funkce (*vystup1*, počet pozorovaných difrakcí, kterým kandidát s nějakou trojicí indexů dobře odpovídal) a pro přímé parametry buňky (*direct*, kromě 3 stran a 3 úhlů obsahuje i objem buňky).

Pro data je alokována paměť i na GPU, spolu s dalšími požadovanými neměnnými daty (hkl trojice, limity intervalu naměřených Q hodnot). V případě čistě CPU výpočtů je tato struktura stále využívána, ovšem bez alokací na GPU. Pro inicializaci a deinicializaci se používají funkce `CPUGPU_start`, `CPUGPU_end`, nebo `CPUGPU_start_dic`, `CPUGPU_end_dic` pro DICHOTOMY výpočty.

database ukládá určitý počet nejlepších řešení, spolu s určitými informacemi pro rozhodování (největší/nejmenší hodnota jednotlivých metrik). Uložená řešení jsou ve formě instancí struktury *item_best*: pro každé řešení jsou uloženy parametry přímé mřížky spolu s objemem, hodnoty kritérií F_n, M_{20} a parametry A_{xy} (např. při optimalizaci řešení se tyto parametry při řešení rovnice stále používají).

Pro sestavování projektu aktuální verze aplikace používá *Visual Studio* projekt, případně jednoduchý skript, který při každém spuštění překompiluje všechny zdrojové soubory. Jedním z cílů práce je implementace přenositelného systému pro automatizaci sestavování, který bude umožňovat konfiguraci sestavení. Výběrem systému se věnuje následující sekce 3.2.

CPU výpočty

Pro paralelní výpočty na procesoru projekt používá rozhraní *OpenMP* [11], které zjednodušuje paralelní programování pomocí knihovních funkcí a preprocesorových direktiv (ve formátu *#pragma omp*). Paralelizaci kódu pak zajišťuje kompilátor, který musí *OpenMP* rozhraní podporovat. Výhodou je snadnější implementace s lepší konfigurovatelností a větší bezpečností (oproti samostatné manipulaci s nízkoúrovňovými objekty pro vytváření a synchronizaci vláken většinou stačí použít několik jednořádkových direktiv).

GPU výpočty

Grafické výpočty se v aplikaci používají primárně pro hrubé ověřování vytvořených kandidátů (na diagramu 3.1 fáze zvýrazněná oranžově). Ač se může zdát, že většina aplikace stále závisí na CPU, ověřování kandidátů je natolik náročnou (a častou) operací, že vliv grafických výpočtů je velký (více popsáno v části 3.1.1).

Zároveň mohou být implementovány GPU výpočty na úrovni jednotlivých metod. To částečně podporuje pouze metoda DICHOTOMY, nabízející grafické výpočty pro monoklinické a triklinické systémy (které jsou nejnáročnější). Ověřování kandidátů na GPU nicméně umožňuje metodě „hloupě“ generovat rychlá řešení, které potom mohou být rychle vyřazeny.

Celkově podpora dalších API znamená reimplementaci 4 CUDA funkcí: *calcGPU* pro ověřování kandidátů a trojici *refineGPU*, *calcGPUDichMon* a *calcGPUDichTri* pro výpočty DICH metody.

3.2 Sestavovací systémy

Standardem pro jazyky C a C++ je technologie *CMake*. CMake samotný nezařizuje kompilaci, ale slouží jako generátor pro různé technologie, od jednoduchých *Makefile* souborů po projektové soubory pro pokročilá vývojová prostředí (např. již zmíněné *Microsoft Visual Studio*).

Alternativ existuje poměrně velké množství, jako například *Bazel* [12] od společnosti Google, *qmake* [13] od tvůrců frameworku *Qt* nebo *Gradle* [14], který je populární především v Java ekosystému. Přesto zůstává CMake zůstává nejpopulárnějším systémem (podle ankety ze zdroje [15]) a je bohatě podporován napříč průmyslem. I vzhledem k osobním zkušenostem se tak CMake jeví jako nejrozumnější volba.

CMake projekty jsou popsány pomocí kořenového souboru *CMakeLists.txt*, podle kterého jsou následně generovány soubory. V souboru je možné se odkazovat na další soubory, do kterých byla konfigurace rozdělena, případně se rekurzivně odkazovat na další CMake projekty definované vlastním *CMakeLists* souborem.

V nejjednodušší podobě CMake umožňuje specifikovat zdrojové soubory, definovat požadované výstupní spustitelné soubory nebo knihovny a nastavit možnosti pro kompilaci. Dále CMake umožňuje například:

Jednodušší integraci s externími technologiemi (např. CUDA), které může automaticky nalézt v systému, automaticky nalezne požadované knihovny nebo cesty k požadovaným souborům, které je pak snadné použít pro sestavování.

Definovat dodatečné procesy, nesouvisející přímo s kompilací (například generování nebo kopírování souborů). Ty lze navíc dobře provázat s jednotlivými částmi projektu (co se má provést před a co po kompilaci).

Detekovat aktuální operační systém nebo použitý kompilátor, a podle toho větvit konfiguraci.

Implementace CMake pro tento projekt je detailně popsána v části 5.1.

3.3 Porovnání GPGPU aplikačních rozhraní

Jak již bylo dříve zmíněno, aplikace používá CUDA framework pro urychlení paralelního ověřování kandidátů (a dodatečně pro výpočty v DICHOTOMY metodě). Zatímco v minulosti grafické karty podporovaly pouze specifické funkce pro grafické vykreslování, moderní architektury umožňují uživateli formulovat libovolné vlastní funkce i pro obecné výpočty, někdy označováno jako *GPGPU* („general purpose GPU“). Tento trend je možný vidět i na vývoji dlouho fungujících grafických API, například OpenGL.

Moderní grafické karty jsou výrazně vhodnější pro velmi paralelizovatelné výpočty než standardní výpočty přes CPU. Nevýhodou je obecně vyšší náročnost na samotné programování – zatímco pro standardní procesory existuje velké množství různých programovacích jazyků, pro GPU je výběr výrazně menší, vyžadují odlišný přístup k programování a je nutné propojení s hostujícím CPU programem.

3.3.1 CUDA

Framework CUDA [16] od společnosti Nvidia je velmi dominantní platformou od svého vydání v roce 2007. Hlavní výhodou je snadná integrabilita se standardním C/C++ kódem pro CPU pomocí speciálního kompilátoru. Funkce pro GPU mohou existovat s běžným kódem v jednom zdrojovém souboru a mohou být snadno volány z CPU kódu speciální syntaxí. Překlad a spouštění funguje prakticky stejně jako u čistě CPU programů.

Hlavní nevýhodou CUDA frameworku je limitace pouze na Nvidia hardware, což znemožňuje použití na ostatních fyzických platformách. To je také hlavním důvodem pro implementaci GPU kódu na alternativních rozhraních.

3.3.2 OpenCL

OpenCL [17] (Open Computing Language) je framework relativně podobný technologii CUDA. Jazyk pro psaní programů je rovněž podobný jazykům C/C++ a nabízí podobné možnosti. Použití je ovšem oproti CUDA složitější, část kódu musí být věnována konfiguraci, samotný GPU kód nemůže existovat spolu s CPU kódem a musí být načítán za běhu.

Standard je aktivně vyvíjen konsorciem Khronos, nicméně budoucí podpora ze strany průmyslu je poměrně nejistá, Například v systému macOS je framework „deprecated“ a podporována byla pouze verze 1.2 z roku 2011.

3.3.3 Vulkan

Oproti OpenCL je standard Vulkan [18] zaměřen primárně na grafické vykreslování. Vznikl jakožto následník OpenGL (Open Graphics Library), s cílem být více nízkoúrovňovým rozhraním, poskytujícím programátorovi větší kontrolu nad hardwarem. Zároveň však podporuje obecné, negrafické, výpočty na GPU. Přestože primární účel je použití pro dodatečné výpočty při vykreslování, je možné Vulkan použít jako čistě výpočetní platformu.

Oproti CUDA nebo OpenCL je však použití a programování výrazně složitější, vzhledem k nutnosti výrazně většího množství kódu pro inicializaci a obtížnosti psaní samotného GPU kódu. Interně Vulkan používá pro GPU kód stejně jako OpenCL SPIR-V mezikód, ovšem v případě Vulkanu se jedná o omezenější *shader* model, který nenabízí možnosti jako například ukazatele.

Vulkan je stejně jako OpenCL v aktivním vývoji pod stejným konsorciem, nicméně oproti OpenCL se těší výrazně větší popularitě a je dobře podporovaný na všech hardwarových platformách a operačních systémech.

3.3.4 Ostatní

Existuje mnoho dalších otevřených technologií, například společnosti AMD i Intel mají své vlastní platformy (AMD ROCm [19], Intel oneAPI [20]), které jsou oproti CUDA otevřené, nebo standard SYCL [21] (opět od Khronos konsorcia). Společnost Apple vyvíjí vlastní API *Metal* [22], které je podporováno na více hardwarových platformách, ale je limitováno na operační systémy Apple (macOS, iOS). Zajímavostí je Vulkan implementace na Apple zařízeních (zvaná *MoltenVK*), která interně používá právě Metal.

Ani jeden ze standardů však není dostatečně podporován napříč všemi výrobci, aby se jednalo o vhodnou volbu při snaze podporovat co nejvíce hardwarových a softwarových platforem.

Nejdříve byl pro implementaci zvolen Vulkan standard. Nicméně po implementaci, ačkoli funkční, byl vývoj shledán jako příliš náročný pro případné budoucí rozšíření. Dodatečně byl tedy GPU kód implementován i v technologii OpenCL. Funkční jsou tedy 3 různé implementace pro GPU výpočty. Detaily o implementaci každé z technologií jsou uvedeny v sekci 5.2.

3.4 Rozšíření dávkových souborů

V rámci přepracování kódu byl dodatečně vznesen požadavek na takzvané *dávkové* soubory, které jsou vzhledem k aktuální struktuře kódu obtížně implementovatelné.

Aplikace v aktuální podobě se vždy spouští pro jeden vybraný konfigurační soubor, kdy jsou prohledány vždy celé zadané rozsahy a v nich jsou nalezeny nejlepší řešení. Výpočet však v daných rozsazích vždy proběhne celý, což může zbytečně zdržovat, pokud již kvalitní kandidát byl nalezen. Zároveň však není jisté, že konfigurace nalezne vůbec nějaké rozumné řešení.

Dávkové soubory by měli být alternativou k individuálním konfiguračním souborům. Měli by umožňovat specifikaci kvalitativních požadavků na hledané řešení (hodnota kritérií F_{20} a M_{20} ze sekce 2.2.4) – jakmile výpočet najde vhodného kandidáta, sám se zastaví, což řeší první zmíněný problém. Zároveň umožňuje specifikace několika konfiguračních souborů, které budou postupně použity pro výpočet, dokud se nenajde dostatečně kvalitní řešení (odtud název dávkový soubor, výpočet je „dávkově“ spouštěn s různými konfiguracemi). To pak řeší druhý zmíněný problém, kdy se řešení hledá obtížněji.

3.5 GUI

Původní aplikace nabízela pouze terminálové uživatelské rozhraní. V rámci této práce bude jako alternativní možnost implementováno plně grafické rozhraní. Tato část se věnuje požadavkům, které by rozhraní mělo splňovat.

3.5.1 Funkční požadavky

F1 Správa aplikačních souborů
V aplikaci bude uživatel moci pracovat s konfiguračními, dávkovými a datovými soubory (obecně *aplikačními* soubory) a to ve všech podporovaných formátech.

Načtené soubory všech různých typů bude aplikace schopna reprezentovat pomocí grafických prvků a bude schopná zobrazit informace o všech hodnotách ze standardní textové podoby. Hodnoty rovněž uživatel může upravovat a pozměněné soubory uložit.

Kromě existujících souborů bude uživatel moci vytvářet soubory nové.

F2 Úprava konfiguračních souborů
Pomocí GUI bude možné vytvářet nové konfigurační soubory a upravovat soubory existující. Automaticky bude kontrolována validita zadaných hodnot a aplikace bude vhodně reagovat na sémantické závislosti mezi různými hodnotami (například volba různých krystalických soustav zabrání/povolí zadání rozsahu prohledávaných úhlů).

F3 Úprava dávkových souborů
Podobně jako u konfiguračních souborů bude možné vytvářet a upravovat dávkovací soubory. U nich bude možné nastavit různé parametry výpočtu a zvolit jednotlivé konfigurační soubory. Aplikace bude ověřovat existenci a validitu zvolených konfiguračních souborů.

F4 Spouštění výpočtů
Z GUI bude možné spouštět samotný hlavní výpočet. Aplikace bude umožňovat graficky zvolit argumenty pro výpočet u kterých automaticky ověří jejich korektnost.

F5 Šablony konfiguračních souborů
Pro usnadnění výpočtů bude aplikace nabízet předem vytvořené konfigurační soubory („šablony"), které bude uživatel moci snadno použít pro výpočet, bez nutnosti vytváření a načítání vlastních souborů.

Kromě předpřipravených souborů bude uživateli umožněno mezi šablony přidat vlastní soubory, ze kterých si následně bude moci vybírat.

3.5.2 Nefunkční požadavky

N1 OS podpora
Aplikace bude *desktop* aplikací, ovládaná myší a klávesnicí. Požadovaná je podpora hlavních operačních systémů (Windows, macOS, Linux).

N2 Prevence chyb
Aplikace bude uživateli při úpravě libovolného typu souboru umožňovat navracení provedených úprav.

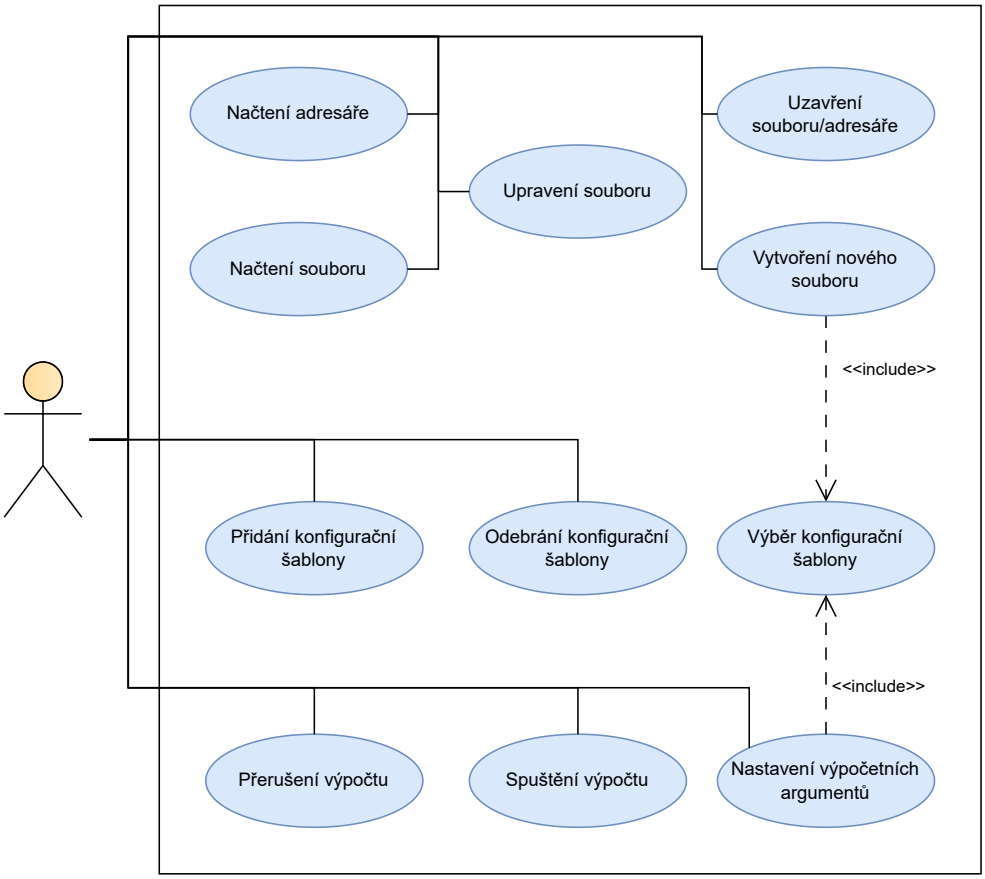
U všech nastavitelných hodnot souborů aplikace umožní zadání pouze korektních dat, nebo uživatele na případnou chybu upozorní. V případě, že se upravovaný soubor stane nevalidním, nebude možné jej uložit.

Uživatel při pokusu o ukončení aplikace bude upozorněn na případné neuložené změny.

3.5.3 Model případů užití

Dále bude podle požadavků na grafické rozhraní představen model případů užití. Jeho cílem je konkretizovat, jaké akce bude uživatel v aplikaci provádět a jakým způsobem budou naplněny funkční požadavky. Případy užití pak poslouží jako *specifikace*, co přesně má navrhované uživatelské prostředí umět. Model je zpracován jako diagram 3.4, společně s textovým popisem každého případu

1 Načtení souboru
Uživatel může v aplikaci načíst jednotlivé aplikační soubory. Po zvolení systémové cesty k souboru se aplikace pokusí soubor načíst a podle



Obrázek 3.4: Model případů užití.

obsahu automaticky zjistit jeho typ. Pokud soubor žádnému formátu neodpovídá, je uživatel upozorněn. Po úspěšném načtení je soubor přidán do seznamu načtených souborů, odkud může být otevřen k úpravám.

2 Načtení adresáře
Kromě jednotlivých souborů může uživatel načíst celou složku. Automaticky budou načteny všechny relevantní soubory a uživatel může pracovat se všemi z nich. Rovněž budou rekurzivně načteny všechny podsložky a jejich soubory. Otevřená složka je přidána do seznamu souborů, odkud uživatel může přistupovat k jejím souborů.

3 Vytvoření nového souboru
Uživatel si může od aplikace nechat vytvořit nový aplikační soubor. Nejdříve uživatel musí zvolit typ souboru a následně vybere systémovou cestu, kam je soubor okamžitě uložen.

V případě nového konfiguračního souboru musí uživatel ještě před zvolením umístění souboru vybrat konfigurační šablonu, která určí výchozí

obsah souboru. Nový datový soubor bude obsahovat předpřipravené hodnoty s komentářem, podle kterých může uživatel soubor upravit. Dávkový soubor bude vytvořen prázdný a před použitím musí uživatel přidat alespoň jednu konfiguraci.

4 Úprava aplikačních souborů
Ze seznamu načtených souborů může libovolný z nich uživatel otevřít pro úpravy. Zobrazená obrazovka pro provádění úprav se liší podle typu souboru. Po provedení úpravy je soubor označen jako modifikovaný a je graficky označen v seznamu. Pro každý soubor jsou uchovávány změny a je možné provádět krok zpět/vpřed pro navracení provedených změn. Při pokusu o uložení aplikace zkontroluje, zdali je soubor validní. Aplikace nedovolí uložení nevalidního souboru.

5 Uzavření souboru/adresáře
Kořenové složky a soubory (tedy ne podsoubory a podsložky nějakého adresáře) mohou být po načtení uživatelem odstraněny ze seznamu. Při pokusu o uzavření aplikace zkontroluje, zdali má soubor (nebo rekurzivně některý z podsouborů adresáře) neuložené změny, a případně uživatele upozorní a nabídne jejich uložení.

6 Výběr konfigurační šablony
Při výzvě pro zvolení šablony je uživateli zobrazen seznam vestavěných a uživatel přidaných konfigurací. Před zvolením si uživatel může pro každou šablonu nechat zobrazit její informační popis (např. pro jaké výpočty má být použita).

7 Přidání konfigurační šablony
Načtené konfigurační soubory ze seznamu může uživatel přidat mezi ostatní nabízené šablony. Ty se poté automaticky zobrazí v seznamu, když je uživatel vyzván k výběru.

8 Odebrání konfigurační šablony
Při výzvě o vybrání šablony může uživatel rovněž ze seznamu odebrat šablony, které sám přidal. Vestavěné šablony odebrány být nemohou.

9 Nastavení výpočetních argumentů
Před spuštěním výpočtu v aplikaci musí uživatel povinně zvolit konfigurační (nebo dávkový) soubor, soubor s naměřenými daty a dobrovolně soubor pro výpis řešení. Všechny soubory mohou být zvoleny pomocí systémové cesty k souboru. Konfigurační, dávkové a datové soubory je rovněž možné zvolit ze seznamu načtených souborů. Místo specifického konfiguračního souboru může uživatel rovněž vybrat konfigurační šablonu.

10 Spuštění výpočtu
Aplikace při pokusu o spuštění zkontroluje uživatelem nastavené argumenty a v případě korektního nastavení spustí výpočet. Aplikace automaticky zobrazuje výstup z výpočtu. V případě, že uživatel nenastavil některý z povinných argumentů, bude aplikací upozorněn.

11 Přerušení výpočtu
Běžící výpočet může být uživatelem dobrovolně přerušen.

3.5.4 Výběr GUI technologie

Vzhledem k úzké provázanosti GUI s hlavní aplikací dává smysl použít stejný programovací jazyk, bez nutnosti buď reimplementovat části kódu nebo řešit případné propojení. GUI tedy bude implementováno jako nativní C++ aplikace.

V této části budou představeny 2 hlavní multiplatformní frameworky pro tvorbu C++ grafických rozhraní.

Qt
Qt [23] je velmi populární multiplatformní framework pro vývoj GUI aplikací. Hlavní výhodou je rozsáhlá dokumentace a aktivní vývoj, s podporou velkého množství platforem. Kromě samotného frameworku nabízí projekt i různé nástroje:

QtCreator je vývojové prostředí, cílené na zjednodušení vývoje Qt aplikací, ovšem použitelné i na obecné C++ programování. Qt je možné integrovat čistě s použitím *CMake*, ovšem projekt sám implementuje některé funkce nepodporované standardním C++ jazykem a sestavování projektu je složitější než u běžných aplikací.

QtDesigner je flexibilní grafický návrhář jednotlivých GUI obrazovek. Výstupem jsou pak XML soubory, které je možné dynamicky nahrát za běhu aplikace. Návrhář mimo jiné umožňuje specifikaci vlastních podtříd existujících grafických elementů a propojení jednotlivých částí. Samotný zdrojový kód je pak jednodušší a lépe odolává pozdějším změnám v samotných změnách návrhu.

Jednou z nevýhod je způsob licencování, kdy je projekt nabízen pod dvěma různými licencemi – placenou komerční licencí a svobodnou licencí (s možností výběru mezi GPL a LGPL), která přináší některé limitace (například nutnost dynamického linkování, zajištění přístupu k původnímu zdrojovému kódu spolu s případnými změnami).

WxWidgets
Další z dlouhodobě fungujících C/C++ GUI frameworků [24]. Oproti

Qt se jedná o čistě open source projekt a nabízí tak svobodnější licencování. Vzhledem k menší komplexitě je rovněž snadněji integrovatelný do různých vývojových prostředí.

Oproti Qt ekosystému nenabízí vlastní oficiální nástroje, při vývoji je třeba využít různých nástrojů od třetích stran. Nejpodporovanějším grafickým návrhářem je *WxFormBuilder*, nicméně v porovnání s Qt návrhářem nabízí méně možností (není možné definovat vlastní typy, veškeré propojování prvků musí být provedeno na úrovni zdrojového kódu). Vzhledem k nekomerčnosti projektu je v porovnání s Qt rovněž hůře dokumentovaný.

Pro vývoj grafického rozhraní byla zvolena technologie Qt, která je v dnešní době prakticky standardem pro vývoj nativních aplikací. Detailům implementace se věnuje sekce 5.3.

3.6 Statické analyzátory

Pro zlepšení kvality kódu budou použity různé statické analyzátory, což jsou nástroje sloužící k prozkoumání neběžícího zdrojové kódu programu s různými možnými cíli: od detekce funkčních problémů jako špatná práce s pamětí, špatné dodržení specifikace, nedosažitelný kód, po kosmetické změny jako nekonzistentní pojmenovávání nebo atypické rozdělení kódu.

Oproti dynamické analýze, prováděnou za běhu programu (např. nástroj *valgrind*), nemůže odhalit tak širokou škálu problémů, nicméně absence nutnosti spouštění umožňuje snadněji udržovat kód do určité míry kvalitní, i při rozšířeních a přepracování kódu.

Různé analytické nástroje existují pro prakticky všechny programovací jazyky, zde se budeme zaměřovat čistě na nástroje pro jazyk C++. Vzhledem k různým zaměřením nástrojů (a různým rozdílům v jejich schopnostech) je obecně vhodné použít více nástrojů najednou. Pro použití v tomto projektu bude provedena krátká rešerše volně dostupných nástrojů. Na trhu nicméně existuje také mnoho komerčních nástrojů, typicky založených na bázi pravidelných předplatných (například klocwork [25], CppDepend [26], PVS-Studio [27]).

Kompilátory

Ačkoli se nejedná o primární účel, prakticky všechny kompilátory jsou schopny při sestavování aplikace upozornit programátora na různé potenciální problémy. Většinou kompilátory nabízí nějaké vlastní možnosti pro specifikaci, jaké skupiny problémů mají být hledány. Prováděné kontroly se mezi kompilátory mohou lišit, může tak být výhodné (zvláště u multiplatformních projektů jako tento) využít pro analýzu více než jeden kompilátor.

cpplint
Program od společnosti Google, který kontroluje dodržování Google konvencí pro psaní kódu [28]. S možností volby kategorií, které budou dodržovány, se může jednat o rozumnou volbu pro dodržení kódovacího stylu v projektu, nicméně v této práci se bude používat komplexnější *clang-tidy*.

cppcheck
Zaměřuje se na vyhledávání potenciálně nebezpečného kódu, nebo kódu vedoucího k nedefinovanému chování (např. správné zacházení s pamětí, přistupování k polím mimo meze, kontrola korektního použití STL) Oproti ostatním nástrojům se snaží vyhledávat chyby v širším kontextu celkového kódu (například v závislosti na možných argumentech funkce). Mimo verzi s otevřeným zdrojovým kódem je k dispozici i placená prémiová verze s podporou a dodatečnými možnostmi. Dostupný ze zdroje [29].

clang-tidy
Jedná se o velmi rozsáhlý a konfigurovatelný program [30], který je úzce svázaný s projektem Clang/LLVM (analýzu provádí pomocí kompilace clang kompilátorem).

Program umožňuje možné vybrat širokou škálu různých kontrol (verze 14 jich nabízí přes 450), rozdělených do různých skupin. Například kontrola různých programovacích konvencí, výkonnostní problémy, doporučení modernějších konstruktů, vizuální ověření, kontrola různých externích funkcí (OpenMP, MPI, Boost).

Pro některé zvolené kontroly je možné nastavovat další možnosti, jako třeba *readability-identifier-naming* sloužící ke kontrole konzistentního pojmenovávání identifikátorů a umožňuje individuálně pro každý typ identifikátorů specifikovat požadovanou konvenci. Aplikace rovněž podporuje pro některé kontroly automaticky aplikovat opravy. Vzhledem k rozsáhlé konfigurovatelnosti umožňuje program pro spouštění použít vlastní konfigurační soubory (v různých formátech).

Dalším nástrojem v Clang ekosystému je *Clang Static Analyzer* (dále *ClangSA*). Ten se oproti kontrolám různých konvencí spíše komplexněji zaměřuje na hledání chyb v programu, podobně jako *cppcheck*.

include-what-you-use
Nástroj s cílem omezit počet vkládaných hlavičkových souborů [31]. Snaží se najít soubory zbytečně vkládané vícekrát v jedné jednotce překladu nebo vkládané soubory, obsah kterých není nijak používán. Rovněž se snaží vyhledat situace, kdy lze vkládání souboru nahradit dopřednou deklarací (takové situace nastávají především v hlavičkových

souborech). Cílem je snížit celkovou provázanost mezi soubory a tím zvýšit rychlost kompilace a zpřehlednit jednotlivé soubory.

clazy
Rozšíření pro *clang* kompilátor, zaměřující se na vyhledávání problémů v kódu specificky pro framework Qt. Nabízí přes 50 volitelných kategorií problémů, od špatné práce s pamětí, problémové použití funkcí nebo nevhodné vzory v kódu.

CodeChecker
Nástroj [32] zjednodušující použití různých analyzátorů (sám o sobě tedy analyzátorem není). Umožňuje uživateli zvolit vícero spouštěných nástrojů, mimo jiné jsou podporovány již zmíněné *clang-tidy*, *Clang Static Analyzer*, clazy a *cppcheck*. Kromě C/C++ podporuje i další jazyky, například Java, Python, JavaScript. Jednou z výhod je ukládání výsledků analýz do souborů a při opakovaném spouštění jsou znovu analyzovány pouze upravené soubory. Dále umožňuje vizualizovat výstup ve formě statických HTML stránek.

Pro vývoj budou použity analyzátory *clang-tidy* a *Clang Static Analyzer*, spolu s nástrojem *CodeChecker* pro jednodušší vývoj. Samotnému použití se věnuje sekce 5.4 v implementační části.

Kapitola 4

Návrh

4.1 Diagram tříd

V rámci předělání projektu byl návrh zdrojového kódu předělán tak, aby více odpovídal objektově orientovanému přístupu. Snahou bylo přistupovat racionálně a prioritizovat přehlednost. Primární cílem bylo do struktur integrovat relevantní funkce jako metody a přehledně vyhradit každé třídě vlastní soubor. Pro zachování podpory s existujícím kódem nebylo nijak řešeno zapouzdření tříd, z tohoto pohledu stále fungují jako čisté *C* struktury. Na obrázku 4.1 je znázorněn UML *class diagram* ([5, kapitola 11]), který poskytuje přehled celého projektu.

Na předělaných strukturách byly provedeny následující změny:

- Ve třídě `Configuration` byly atributy pro vyjádření výpočetní metody, krystalové soustavy a typu zařízení pro výpočet předělány z primitivních celočíselných hodnot do vlastních „výčtových tříd“ (*enum class*). To zpřehledňuje samotný kód, kde se místo primitivních čísel používají konkrétně pojmenované hodnoty. Vnitřní reprezentace zůstává stejná, tedy číselná konverze z a do konfiguračních souborů funguje stejně.

  Samotnou inicializaci z konfiguračního souboru je možné provést přímo z konstruktoru místo specifické inicializační funkce. Pro zpětné uložení do podoby konfiguračního souboru po úpravách v GUI byla přidána metoda *toConfigFileString*.

  Pro konzistentní porovnávání a ukládání konfigurací byla přidána metoda *normalize*, která upraví atributy tak, aby byly mezi sebou logicky konzistentní (pokud například zvolená soustava požaduje určení pouze jednoho parametru a konfigurace specifikuje různé rozsahy pro různé strany).

  Dále byly přidány atributy pro podporu rozšíření do konfiguračních souborů, které budou zmíněny v sekci 4.3.

- `Measured` zůstala po stránce atributů stejná. Funkce pro načítání obou formátů datových souborů (`loadDat`, `loadDic`) byly integrovány jako metody, které je možné pustit i na inicializované instance, podobně jako v původním programu.

  Vzhledem k velkým staticky alokovaným polím je instance této třídy paměťově extrémně velká, a při vytvoření instance na zásobníku může dojít k překročení jeho kapacity (a pádu programu). Proto je výchozí konstruktor nedostupný (označen jako *private*) a pro vytváření nových instancí poskytnuta statická metoda `createPtr`, která zajistí dynamickou alokaci pomocí *chytrého* ukazatele.

  Pro podporu zobrazování chyb při úpravách v grafickém prostředí je přidána metoda static `isValidDataFileInput`, která ze vstupních dat pokusí zkonstruovat validní instanci specifikovaného formátu. V případě chyby návratová hodnota typu `DataFileValidity` obsahuje informace o chybě (problémový řádek, informace o chybě).

- `GPU_data` zůstala stejná, pouze inicializační a ukončovací funkce byly přesunuty do konstruktoru/destruktoru. Specifické inicializace pro metodu *dichotomy* byly rovněž zahrnuty do konstruktoru.

- `Database` zůstala identická, pouze metoda pro přidání řešení (`insert`) byla integrována jako metoda.

Mezi nové přidané třídy patří:

- `BatchConfig` slouží k reprezentaci dávkových souborů, podobně jako třída `Configuration` pro konfigurační soubory. Rozhraní je prakticky stejné, přímo v konstruktoru jsou inicializovány atributy ze souboru, metoda `toBatchFileString` konvertuje instanci zpět do textové podoby.

  Jako atributy třída obsahuje požadavky na kritéria M_{20} a F_n (pokud byly zadány) a cesty k jednotlivým konfiguracím. Ty jsou při inicializaci zkontrolovány, zda ukazují na platné soubory. Očekává se, že cesty ke konfiguračním souborům budou relativní pro přenositelnost napříč zařízeními.

- Samotné vyhledávací výpočty byly přesunuty do hierarchie třídy `Solver`. Sdílené části výpočtu jsou implementovány v hlavní metodě `solve`, jednotlivé podtřídy poté implementují `search` metodu pro hledání kandidátů. `SolverFactory` pak slouží k vytvoření konkrétní instance podle specifikované metody z konfiguračním souboru. Tento přístup zabraňuje komplexním rozhodovacím výrazům z původního kódu, kde v rozsáhlém *switch* bloku nejdříve proběhl výběr podle metody a následně podle krystalové soustavy. Zároveň pro každou implementovanou metodu je snadno dohledatelný její specifický kód. Třída rovněž vlastní ukazatel na instanci třídy `GPUBackend` (zmíněná dále) pro vykonávání GPU výpočtů.

- Pro abstrahování GPU výpočtů byla vytvořena hierarchie tříd s hlavní abstraktní třídou *GPUBackend*, která deklaruje hlavní rozhraní. Kromě hlavních výpočetních metod zmíněných v sekci 3.1.5 obsahuje rozhraní metody pro práci s pamětí, které je navrženo tak, aby co nejvíce odpovídalo existující CUDA technologii.

  Každá z používaných technologií má vlastní implementující podtřídu (*CUDABackend*, *OpenCLBackend*, *VulkanBackend*).

  Pro volání GPU funkcí byla třída *Solver* rozšířena o ukazatel na instanci *GPUBackend*, přes který jsou metody volány. Konkrétní instance je při inicializaci vytvořena pomocí třídy *GPUBackendFactory*, která rovněž ověřuje, zdali je daná technologie podporována.

- ArgParser schraňuje funkcionality pro zpracování argumentů z příkazové řádky (především cesty k souborům) pro jednodušší zpracování v programu.

4.2 Komponentový diagram

Původní aplikace se sestavovala z jednoho spustitelného souboru, *paracell*. Vzhledem k přidání testů, grafického rozhraní a celkové konfigurovatelnosti projektu byla aplikace rozdělena do několika knihoven a spustitelných souborů, vizualizovaných na diagramu 4.2.

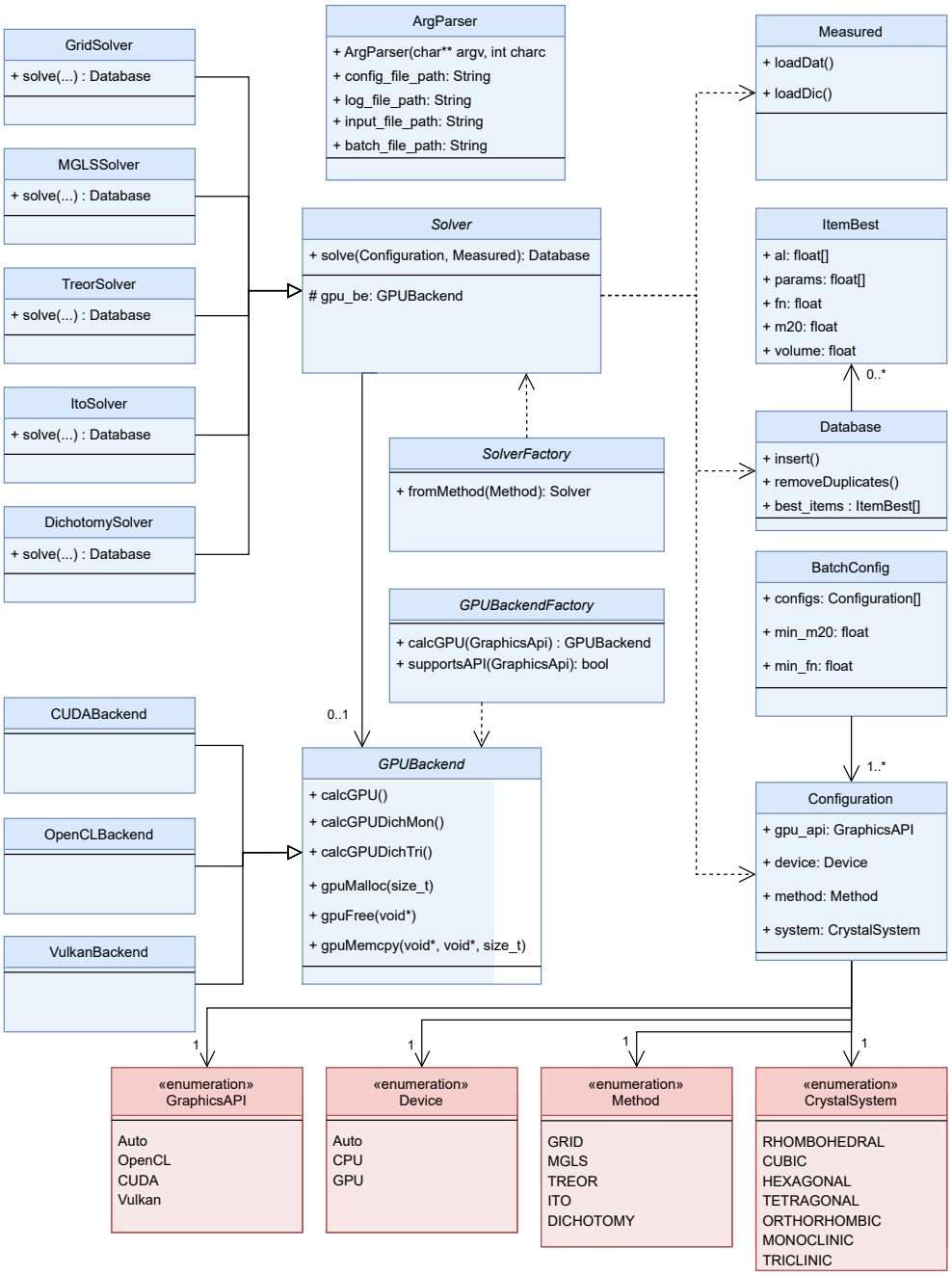
- *paracell* a *paracell-gui* jsou hlavní spustitelné soubory pro terminálové, respektive grafické rozhraní. Pro testování obou částí jsou vyhrazeny vlastní spustitelné soubory *paracell-tests* a *paracell-tests-gui*.

- Funkcionality sdílené napříč spustitelnými soubory jsou odděleny do knihoven *paracell-core* pro hlavní výpočet a *paracell-gui-core* pro grafické rozhraní. GUI knihovna je rovněž závislá na hlavní knihovně.

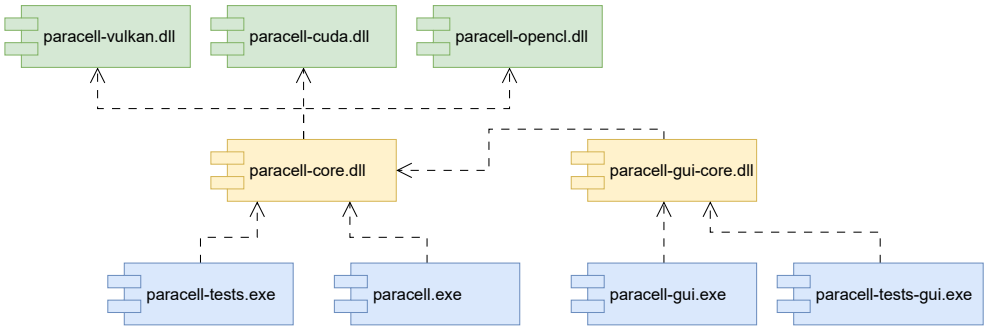
- Každá z technologií pro grafické výpočty je navíc zkompilována do své vlastní malé knihovny. To umožní za běhu programu zjistit podporované technologie a umožní spouštění aplikace i na systémem, které některé z technologií nepodporují.

4.3 Úprava konfiguračních souborů

Specifikace konfiguračních souborů byla rozšířena o nové hodnoty pro zajištění přidaných funkcí a některé z původních hodnot byly rozšířeny pro zpřehlednění a zjednodušení. Hlavní úpravy budou představeny zde, celková specifikace (i s předělanými úpravami) je dostupná v příloze D.



Obrázek 4.1: Diagram tříd v hlavní části aplikace.



Obrázek 4.2: Komponentový diagram jednotlivých částí aplikace. Použity jsou standardní souborové přípony systému Windows.

- GPU_API umožňuje zvolit konkrétní technologii pro grafické výpočty (*CUDA*, *OPENCL*, *VULKAN*). Výchozí hodnota (*AUTO*) automaticky zvolí první podporovanou technologii, v pořadí podle přechozího seznamu.

- DEVICE nastavení je obohaceno o novou výchozí hodnotu AUTO. Automaticky je použita GPU, pokud je podporována alespoň některá z technologií, jinak se použije CPU. Stále je možné vynutit čistě CPU výpočty.

- GPU_HINT umožnuje dobrovolně specifikovat řetězec, podle kterého bude použita konkrétní GPU. Zadaný text se jednoduše pokusí najít jako podřetězec ve jménech jednotlivých zařízení. Pokud se podřetězec nikde nevyskytuje (nebo hodnota není zadána), bude automaticky použito první nalezené zařízení.

Šablony konfiguračních souborů rovněž vyžadují možnost specifikovat metadata o souboru. Možné je specifikovat název konfiguračního souboru, kterým se bude soubor identifikovat v seznamu šablon, a dále je možné přidat popis souboru.

Pro jednoduchost jsou metadata specifikována buď 2 symboly „##“ na začátku řádku pro název souboru nebo jedním symbolem „#“ pro popis. Celý zbytek řádku se interpretuje jako hodnota. Název souboru je jednořádkový, pro popis je možné použít více řádků (každý uvozený stejným způsobem).

4.4 Návrh uživatelského prostředí

V následující části budou představeny hlavní obrazovky implementovaného grafického rozhraní. Jakožto desktopová aplikace se aplikace odehrává téměř výhradně na hlavní obrazovce, bez složitých přechodů. Pro každou obrazovku byl vytvořen jednoduchý statický prototyp v nástroji *Figma* [33].

Hlavní obrazovka

Primární okno, které uživatel používá, okamžitě viditelné po spuštění aplikace (4.3).

V horní části obrazovky se nachází lišta s hlavními ovládacími záložkami:

- *File* pro načtení existujících souborů nebo celých složek, uložení upraveného souboru (nebo všech upravených souborů) a vytvoření souborů nových. Zároveň umožňuje otevření obrazovky nastavení.
- *Edit* pro úpravy aktuálního souboru.
- *Run* umožňuje otevření spouštěcí karty.

V levé části jsou ve stromové hierarchii zobrazeny otevřené soubory a složky. Zobrazují se pouze relevantní typy souborů, které jsou od sebe graficky odlišeny ikonou. Každý soubor je možné upravovat otevřením karty po dvojkliku. Po kliknutí pravým tlačítkem na otevřené složky nebo soubory se otevře kontextové menu. Z menu je možné tlačítkem uzavřít položku a odstranit ji ze seznamu (aplikace kontroluje, zdali byl soubor nebo soubory ve složce změněny a v případě uzavření nebo ukončení celé aplikace je uživatel vyzván k uložení změn). Konfigurační soubory je z menu možné přidat mezi seznam šablon.

Nejdůležitějším prvkem je seznam otevřených karet uprostřed okna, které slouží k hlavní práci – každá karta buď slouží k úpravě jednoho konkrétního souboru (konfigurační, dávkový, datový), nebo se jedná o kartu pro spouštění výpočtů.

Po prvním spuštění je automaticky zobrazena obrazovka pro spuštění, karty pro jednotlivé soubory je možné otevřít ze seznamu. Mezi kartami lze přepínat záložkami v horní části seznamu. Karty je možné zavřít, v takovém případě změny zůstanou v paměti a kartu je možné znovu otevřít přes příslušný soubor.

Ve výchozí podobě jsou záložky organizovány do jednoho seznamu a viditelná je pouze jedna karta, uživatel však může seznam horizontálně rozdělit (případně následně spojit) přetažením karty k levému nebo pravému okraji a zobrazit více karet najednou. Jednotlivá rozdělení je možné zvětšit/zmenšit posunovačem mezi každými dvěma rozděleními. Karty je rovněž možné přesunou do jiného rozdělení přetažením záložky z jedné lišty do jiné.

V dolní části obrazovky se ještě nachází jednoduché textové pole, poskytující informace o stavu aplikace (např. „Soubor byl uložen“).

Karty

Dále bude představen návrh dříve zmiňovaných karet, které mohou být otevřeny na hlavní pracovní ploše:

Konfigurační soubor

Veškeré nastavitelné hodnoty z textové podoby je možné upravit pomocí grafických prvků. Hodnoty jsou logicky rozděleny do několika kategorií:

- Obecné hlavní nastavení – typ zařízení, výpočetní API, krystalická mřížka.
- Omezení na vyhledávanou mřížku (minimum a maximum délek stran, velikost úhlů, objem).
- Nastavení dodatečných úprav po hlavním výpočtu (optimalizace, nalezení „superbuňky“, ...).
- Nastavení specifické pro zvolenou metodu.

Pro možnosti s malou možností různých hodnot (výběr CPU nebo GPU, grafické technologie, cílová krystalická soustava) se používají rádiová tlačítka nebo *combo boxy*, pro ostatní se používají textová pole, která automaticky umožnují zadávání pouze smysluplných hodnot. Při změnách hodnot jsou podle sémantiky automaticky deaktivovány nebo schovány některé grafické prvky:

- Při specifikaci výpočtu pomocí CPU je deaktivováno zadání grafického API.
- Po změně krystalové soustavy jsou ponechány aktivní pouze relevantní prvky pro nastavení prohledávaných rozsahů parametrů mřížky. Například pro kubickou mřížku je možné nastavit rozsah pouze jedné strany.
- Některé metody nabízí nastavení dodatečných parametrů, zobrazení jako poslední skupina na kartě. Zobrazena je vždy pouze skupina pro aktuální metodu.

Při najetí kurzorem nad libovolný prvek aplikace zobrazí malé vyskakovací okno s vysvětlením dané hodnoty.

Obrazovka podporuje možnost krokově navracet/opakovat provedené změny nad všemi prvky.

Po každé změně je automaticky soubor označen jako modifikovaný, a je označen hvězdičkou v záložce a v seznamu souborů.

Dávkový soubor

Návrh je podobný návrhu pro konfigurační soubor. Jednotlivé omezující hodnoty je možné nastavit pomocí textových polí. Po jakékoli změně je podobně označen jako modifikovaný.

Pro výběr jednotlivých konfigurací pro dávkový soubor slouží seznam v dolní části, z počátku prázdný. Pomocí 4 tlačítek po pravé straně je možné přidat novou konfiguraci (výběrem ze souborového systému), měnit pořadí metod v seznamu (posunutí nahoru a dolů) a odstranit zvolenou konfiguraci.

Rovněž je možné přidat konfiguraci ze seznamu otevřených souborů na hlavní obrazovce, a to kliknutím a přetažením souboru na seznam.

Datový soubor

Oproti konfiguračním a dávkovým souborům jsou datové soubory vyobrazeny pouze v textové podobě, kterou je možné upravovat. V porovnání se standardním textovým editorem je aplikace schopná zjistit, zdali je datový soubor validní a zvýraznit řádek, na kterém došlo k chybě. Po ukázání kurzorem na zvýrazněný řádek se zobrazí malé vyskakovací okno s informací o chybě. V případě, že se soubor nachází v nevalidním stavu, nebude uživateli umožněno jej uložit. Ostatní typy souborů není možné dostat do nevalidního stavu.

Aplikace podporuje více formátů datových souborů, což je reflektováno v dolní části obrazovky polem s aktuálním formátem souboru. Formát je automaticky nastaven při načtení souboru, nicméně jej uživatel může manuálně změnit. Aplikace po změně formátu ihned znovu překontroluje soubor podle nového formátu a zobrazí případné chyby.

Spouštění

Na obrazovce je možné zvolit příslušný konfigurační/dávkový soubor, datový soubor a případně logovací soubor. Podobně jako na obrazovce dávkového souboru je možné soubory dialogem vyhledat soubor v systému, nebo použít již načtené soubory přetažením ze seznamu na příslušnou řádku (během tažení souboru aplikace zobrazí malé informační okénko nad místem, kam může soubor pustit). Konfigurační soubor je rovněž možné zvolit ze seznamu šablon. Po spuštění je výstup zobrazován v okně v dolní části.

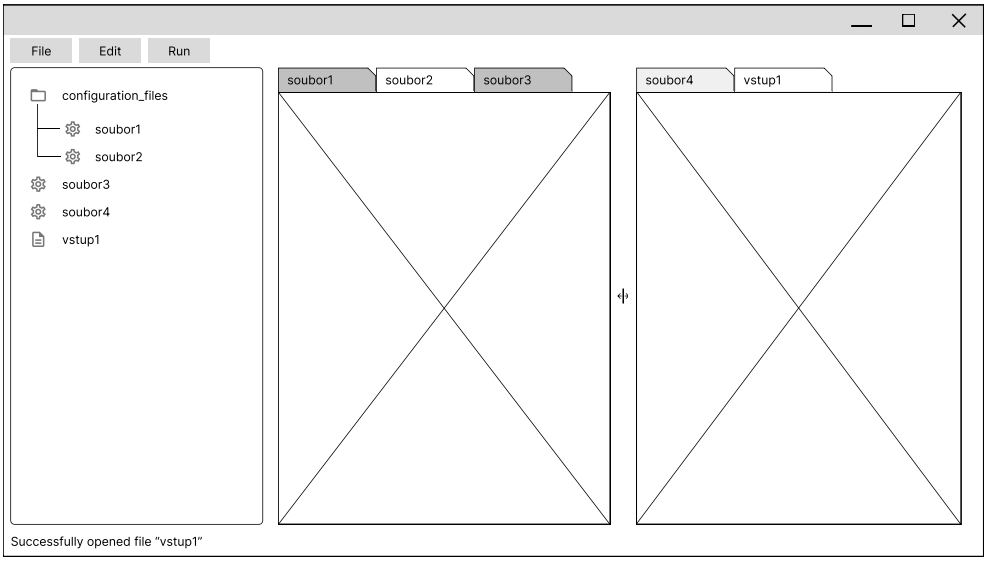
Během výpočtu se stane aktivním tlačítko pro přerušení běhu, pokud by uživatel chtěl běh přerušit. Zároveň se stanou neaktivní ostatní prvky, není možné spustit několik běhů najednou.

Po dokončení běhu bude uživatel informován jednoduchým dialogem (uživatel v době výpočtu může obrazovku s výpočtem opustit nebo ji i zavřít). Dialog bude mít jinou podobu v případě, že běh skončil s chybou.

Výběr konfigurační šablony

Dialogové okno, pomocí kterého uživatel může ze seznamu vybrat konfigurační soubor sloužící jako šablona (při volbě konfigurace pro spuštění nebo při vytváření nových konfiguračních souborů).

Na levé straně dialogu se nachází abecedně seřazený seznam se všemi načtenými šablonami, používajícími názvy definovanými přímo v souborech, jak bylo zmíněno v sekci 4.3. Textovým polem v horní části je možné filtrovat zobrazené soubory podle názvu.



Obrázek 4.3: Návrh hlavní obrazovky.

Po kliknutí na soubor se v pravém horním okně zobrazí popis souboru (pokud je zapsán), a v pravém dolním rolovacím okně je zobrazena textová podoba souboru.

Šablonu je možné zvolit buď tlačítkem v dolní části nebo dvojklikem na seznam.

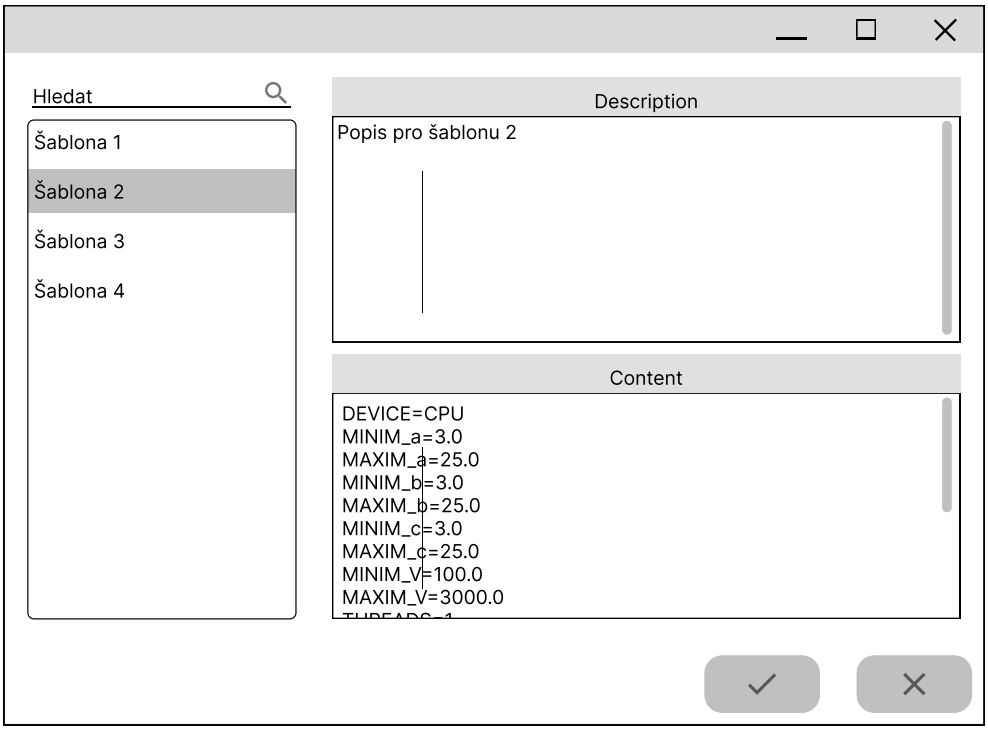
Nastavení

Obrazovka umožňující konfigurovat aplikaci. Uživatel může změnit jazyk aplikace. Zároveň si může nechat otevřít složku obsahující uživatelsky specifická data, která se nacházejí mimo hlavní adresář s aplikací.

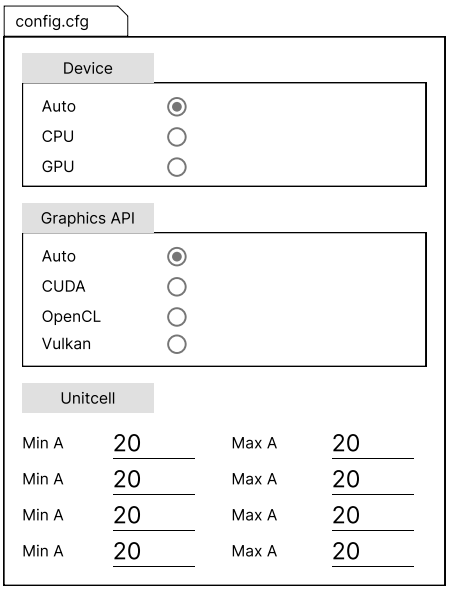
4.4.1 Architektura GUI

Návrh architektury grafické aplikace je zobrazen na diagramu 4.9. Vzdáleně může připomínat vzor *Model-view-presenter*, který rozděluje architekturu na 3 oddělené vrstvy. *View* vrstva představuje samotné uživatelské rozhraní, bez žádné komplexnější aplikační logiky. Tu zajišťuje vrstva *Presenter*, která reaguje na uživatelský vstup od vrstvy *View* a zároveň podle změn může grafické prvky upravovat. Vrstva *Model* představuje sdílená aplikační data, podle kterých se *Presenter* může rozhodovat, případně může data upravovat.

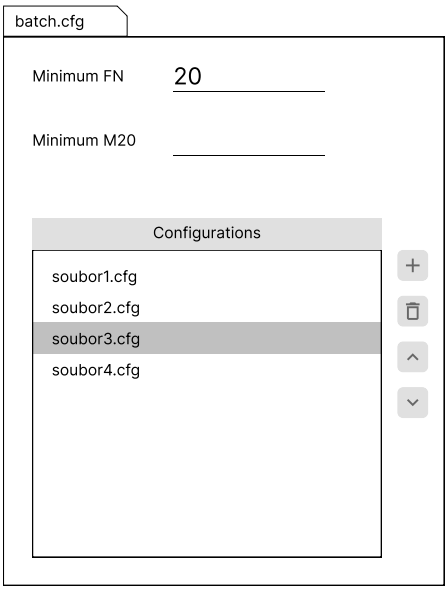
Horní vrstva architektury, představovaná XML soubory popisujícími jednotlivé obrazovky, představuje návrh rozhraní, bez jakékoli aplikační logiky (analogie *View* vrstvy). Za běhu jsou podle souboru vytvořené jednoduché grafické objekty, které následně řídí instance tříd z prostřední vrstvy (analo-



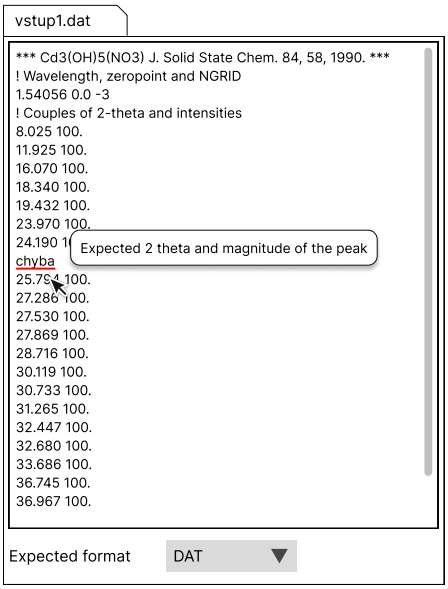
Obrázek 4.4: Obrazovka pro výběr konfigurační šablony.



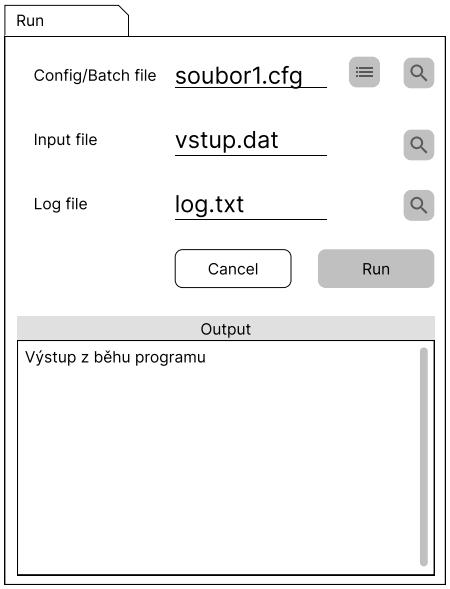
Obrázek 4.5: Karta pro úpravu konfiguračních souborů.



Obrázek 4.6: Karta pro úpravu dávkových souborů.



Obrázek 4.7: Karta pro úpravu datových souborů.



Obrázek 4.8: Karta pro spouštění výpočtů.

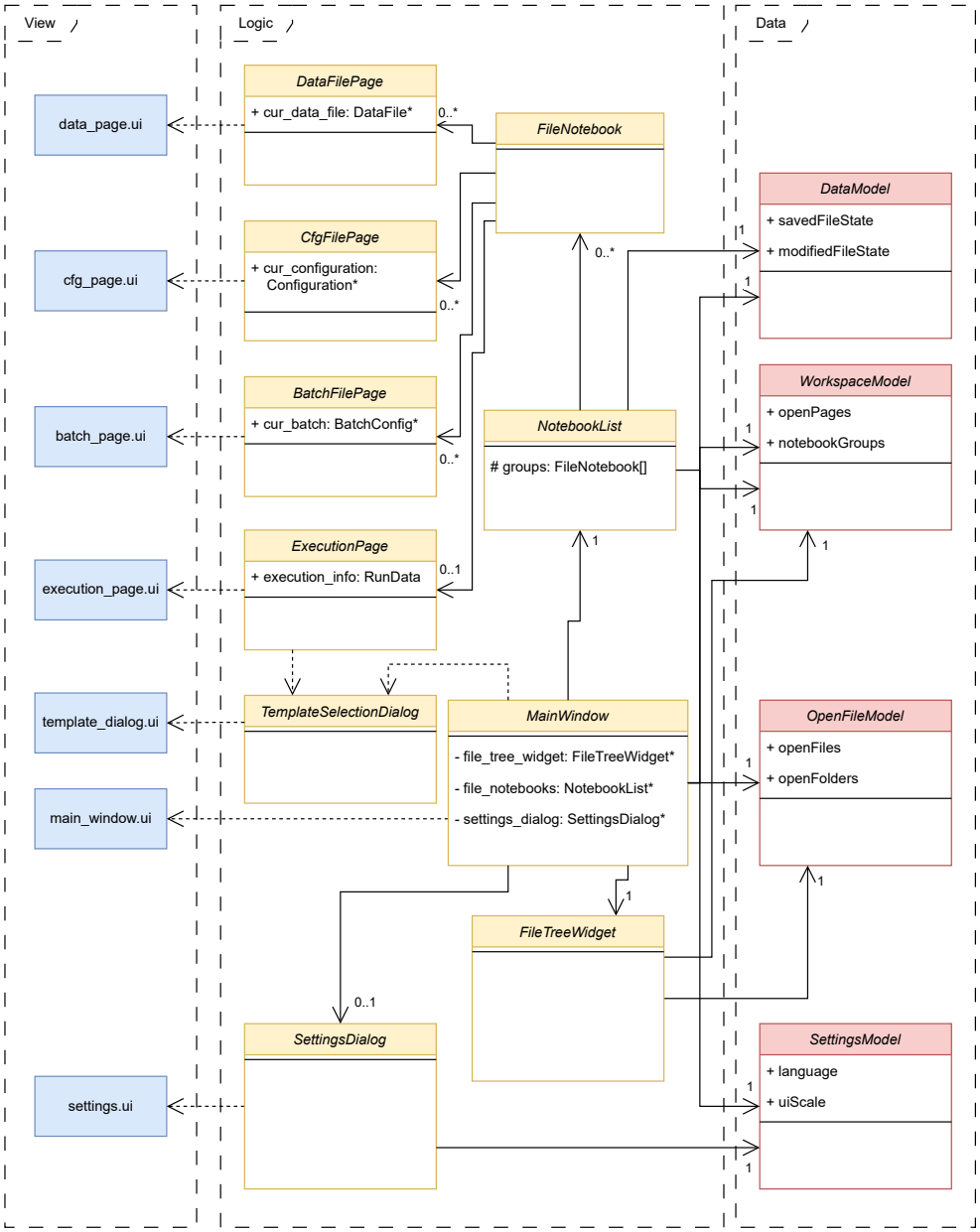
gie *Presenter* vrstvy). Každému souboru rozhraní odpovídá právě jedna řídící třída.

Třídy logické vrstvy tvoří vlastní stromovou strukturu, vyplývající ze samotného návrhu aplikace. Hlavní, kořenová třída `MainWindow` zajišťuje logiku hlavního okna (ukládání souborů, otevření nastavení, ...). Přímo spravuje instance `SettingsWindow` pro logiku nastavení, `FileTreeWidget` pro seznam otevřených souborů (otevírání karet souborů, zavírání otevřených souborů) a `NotebookList` pro hlavní seznam karet (otevírání a zavírání karet, rozdělování a sjednocování jejich skupin). Jednotlivé skupiny jsou rozděleny do instancí třídy `FileNotebook`. Pro každou vytvořenou kartu jsou vytvářeny instance příslušné spravující třídy (`CfgFilePage`, `DataFilePage`, ...).

Poslední vrstvu představují hlavní data aplikace, kdy je jediná instance každé třídy sdílená mezi několika instancemi tříd logické vrstvy. Pro omezení provázanosti („všechny třídy mají přístup ke všemu“) jsou data rozdělena do 4 kategorií (pouze `MainWindow` má přístup ke všem):

- OpenFilesModel pro informace o otevřených souborech a složkách (jaké soubory jsou otevřeny, jakého jsou typu, soubory ve složkách).

- DataModel obsahuje data o aktuálně upravovaných souborech (uložený stav a aktuální, modifikovaný stav). Jednotlivé karty potom obsahují pouze referenci na data relevantní pro svůj soubor.



Obrázek 4.9: Architektura GUI aplikace.

- SettingsModel ukládá nastavení aplikace. Ačkoli je dostupná pro celou aplikaci, pouze *MainWindow* a *SettingsWindow* mohou nastavení upravovat, pro zbytek aplikace jsou pouze ke čtení.
- WorkspaceModel pro data o otevřených kartách a rozdělených skupinách (tato data jsou potřebná pro podporu uložení a obnovení stavu aplikace při dalším spuštění).

Kapitola 5

Implementace

5.1 CMake

V této části bude popsána integrace sestavovacího systému CMake do projektu.

Popis projektu začíná v souboru *CMakeLists.txt* pomocí vlastního skriptovacího jazyka. Ten vzdáleně připomíná standardní programovací jazyky: příkazům jsou argumenty předávány pomocí kulatých závorek, je podporováno větvení nebo definice vlastních funkcí. Místo uvozování kódu do bloků (např. složenými závorkami) se používají dvojice příkazů, podobně jako v preprocesoru jazyka C (např. pro větvení se používá příkaz `if()`, ukončený příkazem `endif()`). Proměnné lze definovat pomocí příkazu `set` a následně k nim přistupovat pomocí výrazu `${}`, například `set(var TRUE)`, následný přístup pomocí `${var}`. Jedním z běžných použití proměnných je definice zdrojových souborů pro kompilaci.

V případě komplexních projektů je možné vytvořit vlastní *CMakeLists* soubor pro samostatné podprojekty, v tomto projektu je však použit pouze jeden hlavní soubor.

Úplným základem definice je specifikace minimální požadované verze pomocí příkazu `cmake_minimum_required` a definice základních informací o projektu příkazem `project`. Následně je možné definovat kompilované výstupy, hlavními příkazy jsou `add_executable` a `add_library`, které vytvoří spustitelný soubor, respektive knihovnu. Příkazům jsou mimo jiné předány zdrojové soubory a název, který se stane takzvaným „cílem“ (*target*).

Cíl je pak možné používat jako argument příkazů pro další konfiguraci: `set_target_properties` pro specifikaci některých obecných vlastností (jako `CXX_STANDARD` pro určení verze standardu C++), `target_compile_options` pro různé přepínače pro kompilátor nebo `target_link_libraries` pro specifikaci linkovaných knihoven.

CMake rovněž automaticky u zdrojových souborů vyhledá závislosti na vkládaných souborech, a automaticky zajišťuje opětovné sestavení souborů

pouze, pokud některá ze závislostí byla změněna. Není tak nutné například specifikovat závislosti na hlavičkových souborech jako při manuálním psaní *Makefile* souborů.

Rovněž je usnadněna integrace s externími závislostmi, například pomocí příkazu `find_package`, který je schopen automaticky vyhledat danou závislost (pokud podporuje integraci s CMake). V tomto projektu se příkaz používá pro OpenMP, OpenCL, Vulkan a Qt. Jednou z výhod je snazší linkování požadovaných knihoven, které jsou povětšinou importovány jako CMake cíle (příkladem může být Qt) nebo jako proměnné (v případě OpenCL). Není tak nutné manuálně vypisovat všechny názvy požadovaných knihoven pro danou technologii.

Velmi jednoduchá je integrace s jazykem CUDA, pro jeho použití je možné použít příkaz `enable_language(CUDA)`. Při kompilaci budou potom soubory s příponou .cu automaticky kompilovány s *nvcc* překladačem.

Dále CMake umožnuje kromě samotného překladu definici vlastních nestandardních příkazů. Pomocí `add_custom_command` lze zadefinovat vlastní příkaz z příkazové řádky. U něj je možné specifikovat jeho výstup a závislosti (tedy v případě závislosti na souborech se příkaz provede pouze při jejich změně, v případě závislosti na jiném cíli se pak příkaz provede po jeho aktualizaci). Výstupy z příkazu je poté možné použít jako závislosti u cílů. V projektu jsou vlastní příkazy použity pro kompilaci Vulkan shaderů a překopírování některých dodatečných souborů do výstupní složky.

Konfigurovatelnost projektu je možné zajistit definicí proměnných při inicializaci projektu, podle kterých je následně možné větvit kód. Konkrétní seznam a popis těchto proměnných, spolu s návodem k použití jsou uvedeny v příručce B.

5.2 GPU

Tato část se věnuje popsání některých implementačních detailů technologií OpenCL a Vulkan a bude provedeno srovnání s dosavadním CUDA kódem.

5.2.1 OpenCL

Jak již bylo dříve zmíněno, OpenCL je podobnější frameworku CUDA než Vulkan, ale stále se jedná o složitěji použitelnou technologii oproti snadné integraci za pomocí specifického kompilátoru (OpenCL se obejde čistě se standardním C/C++ kompilátorem).

Pro základní inicializaci jsou potřeba alespoň funkce `clGetPlatformIDs`, `clGetDeviceIDs`, `clCreateContext`, `clCreateCommandQueue` pro výběr platformy, zařízení a vytvoření kontextu spolu s frontou pro příkazy. Platformy prakticky představují odlišné implementace OpenCL (kromě různých implementací pro různý hardware může existovat více implementací pro jedno konkrétní zařízení).

Velmi podobně probíhají operace s pamětí. Pomocí funkcí `clCreateBuffer` a `clReleaseMemObject` je možné alokovat a uvolňovat paměť (respektive v případě OpenCL buffer objekty). Následný zápis a čtení lze provádět pomocí clEnqueueWriteBuffer a clEnqueueReadBuffer. Všechny funkce obecně vyžadují výrazně větší množství argumentů než CUDA ekvivalenty, nicméně nabízejí bohatší konfiguraci, ke kterým CUDA nabízí specifické funkce (jako například blokující/neblokující přenos).

Výrazným rozdílem je samotné spouštění GPU kódu. Zatímco CUDA podporuje transparentní integraci s ostatním kódem v jednom zdrojovém souboru (volání GPU funkcí vypadá prakticky stejně jako volání běžné funkce, pouze obohacenou o specifikaci počtu vláken/bloků/mřížek), OpenCL vyžaduje explicitní načtení GPU kódu za běhu.

Nejpřirozenější formou psaní OpenCL kódu je pomocí jazyka *OpenCL C*, nepřekvapivě podobného standardnímu jazyku *C*. Interně se však v OpenCL používá mezikód *SPIR-V* (konkrétně *kernel* výpočetní model) do kterého musí být standardní kód přeložen. Pro přeložení lze použít externí překladač, nebo přímo za běhu použít funkci `clCreateProgramWithSource`, které je předán přímo zdrojový kód. Takto přeložený program je možné rovnou použít, nicméně je možné si vyžádat a případně uložit binární podobu programu. Tu je pak možné načíst funkcí `clCreateProgramWithBinary`, bez nutnosti opětovné kompilace, která je poměrně časově náročná. Tímto způsobem funguje i implementace v *ParaCell*, která po prvním překladu binární podoby uloží, a při dalších spuštěních je automaticky použije. Vzhledem k možnosti použití vícero různých GPU jsou pro každou z nich uloženy individuální binární podoby.

Tento způsob překladu prakticky znamená, že GPU kód existuje odděleně ve vlastních souborech. Jistou výhodou oproti CUDA je však snazší iterování daného kódu, kdy není třeba při libovolné změně překompilovávat celé zdrojové soubory a znovu linkovat aplikaci.

Obtížnější je i samotné spouštění kernelů. Jednak vykonávání příkazů probíhá přes dříve zmiňovanou frontu, což vyžaduje použití dvojice funkcí `clFlush` pro přenos dříve specifikovaných příkazů a `clFinish` pro vyčkání na jejich dokončení. Pro vykonání kernelu slouží `clEnqueueNDRangeKernel`, které je mimo jiné předána instance `cl_kernel`, který má být vykonán (kernel je vytvořen ze dříve zmíněného `cl_program` pomocí funkce `clCreateKernel`, ve které se specifikuje název vstupní funkce).

Výrazný rozdíl existuje i při předávání argumentů. U obou API se funkce deklarují velmi podobně jako v jazyce C, pouze u OpenCL je třeba věnovat větší pozornost specifikaci adresního prostoru u proměnných (například ukazatele na data v alokované globální paměti zařízení musí být označeny jako `__global`). CUDA funkce jsou ovšem volány přímo z kódu, a překladač může zajistit předání argumentů, spolu s ověřením typů. V OpenCL je třeba před spuštěním kernelu nastavit každý argument pomocí funkce `clSetKernelArg`, které se předává instance daného kernelu, index (pořadí) argumentu ve funkci, velikost argumentu a ukazatel na samotná data. Vzhledem k tomuto dy-

namičtějšímu přístupu je třeba větší kontroly, zda jsou data předávána korektně.

5.2.2 Vulkan

Vulkan se oproti oběma předchozím API výrazně liší. Jakožto primárně grafické API je výrazně obsáhlejší a vzhledem k velké obecnosti je velmi konfigurovatelné a náročné na inicializaci. Jakožto minimum je třeba vytvořit instance struktur `VkInstance`, `VkPhysicalDevice` a `VkDevice` (fyzické a logické zařízení), `VkQueue`. Typické pro Vulkan funkce je, že místo velkého množství argumentů přijímají specifické struktury, které obsahují samotná konfigurační data. Samotné struktury mohou být i vnořeny do sebe, například pro vytvoření `VkInstance` funkcí `vkCreateInstance` je předávána struktura `VkInstanceCreateInfo`, která sama ještě obsahuje strukturu obsahující informace o aplikaci (`VkApplicationInfo`). Navíc se některé struktury mohou řetězit dynamicky – atribut `pNext` typu `void*` může ukazovat na nějakou další strukturu. Pro zjištění konkrétního typu obsahují struktury atribut `sType` s hodnotou odpovídající jejich datovému typu.

Vzhledem k obecnému zaměření pro co nejvíce zařízení je mnoho funkcí dostupných formou různých rozšíření. Například i podpora 64bitových číselných typů musí být explicitně vyžádána ve struktuře `VkPhysicalDeviceFeatures` při vytváření logického zařízení. V základu Vulkan nenabízí prakticky žádné ladící mechanismy. Zajímavostí v tomto ohledu je doplňkový systém „validation layers“. Tento systém mimo jiné umožnuje zaznamenávání volaných Vulkan funkcí, s možností ověření, zdali byla dodržena specifikace. Rovněž zajišťuje možnost výpisů (ekvivalent `printf`) ze spouštěných shaderů. Jejich použití musí být explicitně specifikováno při vytváření instance – je tak možné je zcela vypustit při reálném nasazení a jejich použití tak nemusí mít žádný vliv na konečný výkon. V tomto projektu je použití *validation layers* odděleno použitím přepínače `VULKAN_DEBUG` při inicializaci pomocí CMake.

Odlišně probíhá i alokace paměti. Vulkan jednak pracuje se samotnou pamětí na zařízení (struktura `VkDeviceMemory`, která je alokovaná pomocí `vkAllocateMemory` a uvolněna pomocí `vkFreeMemory`), ale rovněž používá buffer objekty (`VkBuffer`), které jsou *mapovány* na alokovanou paměť a které se později používají pro předávání dat do shaderů (konkrétnější popis následuje níže). V ideálním případě by tak pro jednu alokaci paměti existovalo více bufferů, což lépe odpovídá grafickému programování, kde je pravděpodobnější nutnost použití menších datových struktur pro velké množství různých grafických objektů. Vulkan umožňuje navázat jeden buffer na více paměťových míst v shaderu pomocí *offsetů*. Pro jednoduchost implementace a vzhledem k relativně malému množství alokací v konkrétních výpočtech je však pro každou paměťovou alokaci vytvořen jediný buffer, který odpovídá celému rozsahu paměti.

Jinak probíhají i samotné paměťové operace, kdy místo jedné funkce pro přenos je nutné nejdříve alokovanou paměť namapovat do adresního prostoru hostujícího procesu, na ní provést příslušnou operace a následně paměť odmapovat. Oproti CUDA se tak místo jedné funkce `cudaMemcpy` použije kombinace `vkMapMemory`, standardní `memcpy` a `vkUnmapMemory`.

Nejvýraznějším rozdílem je samotné psaní kódu. Podobně jako OpenCL používá Vulkan interně *SPIR-V* mezijazyk, ale s výraznými rozdíly. Zatímco OpenCL používá *kernel* výpočetní model (*execution model*), Vulkan používá výpočetní (*compute*) shadery, s výpočetním modelem *GLCompute*, který je výrazně limitován. Oproti OpenCL Vulkan implementace neobsahují kompilátor a je tak nutné kód přeložit předem a za běhu pouze načíst přeložený *SPIR-V* binární kód.

Pro psaní Vulkan shaderů je hlavním jazykem GLSL, dříve vytvořeného pro OpenGL (OpenGL shading language). V oficiálním SDK je pro překlad GLSL kódu dodáván referenční překladač *glslangValidator*), spolu s dalším překladačem *glslc* od společnosti Google, který interně používá referenční nástroje, obohacené o některé další funkce (například Cčková makra `#define` a `#include`).

Jazyk *GLSL* rovněž vychází ze Cčkové syntaxe, nicméně obsahuje výrazné limitace. Pravděpodobně nejdůležitějším limitem je absence ukazatelů. Je však možné definovat pole se známou délkou a k prvkům lze přistupovat klasickým operátorem `[]`. Výstupní argumenty funkcí je možné použít, pomocí deklarace `out` nebo `inout` před názvem parametru. Nicméně není možné psát obecné funkce, které by prováděli operace nad daty různé délky. Jistou alternativou je psaní takových funkcí jako Cčková makra.

Tato limitace by samozřejmě zabraňovala práci nad nějakými dynamicky alokovanými daty různé délky, nicméně je možné předávaná data definovat jako pole nespecifikované délky (konkrétně však pouze u interface bloků kvalifikovanými jako buffer) a libovolně přistupovat k prvkům (v rámci specifikace). Samotná uživatelská data nejsou předávaná jako argumenty funkcí jako je tomu u CUDA a OpenCL frameworků (Vulkan začíná povinnou bezparametrovou funkcí `main`), ale namísto toho jsou nastavovány globální proměnné opatřené specifikátorem `layout` spolu s kvalifikátory pro identifikaci z hostujícího kódu. Příklad definice takové proměnné může vypadat následovně:

```
layout(set = 0, binding = 3, std430) buffer BufferExample{
        float vals[];
} buf;
```

Jméno proměnné je pak `buf` a funguje podobně jako běžná struktura z jazyka C – k hodnotám lze přistoupit pomocí tečkové notace (např. `buf.vals[5]`). V hostujícím kódu pak mapování dat probíhá poměrně komplexně pomocí *popisovačů* (descriptor sets). K propojení s existující instancí `VkBuffer` probíhá pomocí struktury `VkDescriptorBufferInfo`, které je specifikován daný buffer spolu s offsetem do paměti a rozsahem, který má být zobrazen. Pro

propsání samotné změny se pak používá struktura `VkWriteDescriptorSet`, která mimo jiné obsahuje atribut `dstBinding` pro indexaci globální proměnné, které se změna týká (v předchozím příkladu by se tedy rovnal hodnotě 3). U bufferů je možné specifikovat, jakým způsobem mají atributy zarovnávány. Například možnosti `std140` nebo `std430` fungují podle specifikovaného standardu, některé jiné (`packed` a `shared`) potom záleží na konkrétní implementaci.

Primitivní datové typy mohou být předávány stejným způsobem, nicméně je možné v jednom bloku definovat více atributů. Je třeba zajistit, že mapovaný buffer obsahuje korektně uložená data (především odpovídající zarovnání), jinak mapování funguje stejně jako v předchozím příkladě. Možností je použití namísto storage bufferu takzvaný *uniform buffer*, který má omezenou kapacitu a jehož proměnné jsou konstantní, nicméně výhodou je možné zlepšení výkonu.

```
layout(set = 0, binding = 2, std140) uniform ValuesBuffer {
        int pocet;
        float prec1;
        int limit;
} values;
```

Samotné spouštění shaderů je opět poměrně komplikované, mimo jiné zahrnuje použití `VkDescriptorSet` (dříve zmíněné), `VkCommandPool`, `VkPipeline`, `VkCommandBuffer`. Na nejvyšší úrovni není princip příliš odlišný od OpenCL, pomocí `vkQueueSubmit` jsou do fronty zaslány instance `VkCommandBuffer` s danými příkazy k vykonání a pomocí `vkQueueWaitIdle` lze blokovaně čekat na jejich dokončení.

5.2.3 Dynamické linkování

Jedním z nedostatků původní verze programu byla nemožnost spuštění na zařízeních bez podpory CUDA frameworku, ani pokud bylo cílem využít pouze CPU výpočty. Dosud bylo zmíněna možnost v čase kompilace dobrovolně zapnout/vypnout podporu libovolného API, což však stále neřeší přenositelnost na jiná zařízení nepodporující některé z API.

Jako řešení byl přidán do CMake přepínač *RUNTIME_GPU_SUPPORT*. Při povolení budou při kompilaci vytvořeny tři další knihovny pro každé z grafických API (paracell-vulkan, paracell-opencl, paracell-cuda). Žádná z knihoven se rovněž nebude linkovat do hlavního spustitelného programu, který nebude obsahovat ani žádné závislosti na externích grafických knihovnách. Při spuštění se pokusí každou z vytvořených knihoven dynamicky načíst pomocí systémově závislých funkcí (např. `LoadLibraryW` pro Windows nebo `dlopen` pro Linux) a zaznamená si korektně načtené a podporované knihovny. Pokud zařízení některé API nepodporuje, načtení selže a aplikace sama bude vědět, že není podporováno.

Původní chování je stále zachováno například pro lokální vývoj, kde je sestavování dalších knihoven při menších změnách zcela zbytečné.

5.3 GUI

V této sekci bude popsán způsob implementace grafického rozhraní pomocí frameworku Qt – jakým způsobem je integrován, některé jeho základní principy a jakým způsobem probíhá implementace samotného rozhraní. Použita byla starší verze 5, která je však dlouhodobě podporována.

5.3.1 Sestavování, integrace s CMake

Celková komplexita Qt frameworku byla zmíněna již dříve, plynoucí především z toho, že kromě standardní kompilace je framework závislý na dodatečných vlastních procesech pro zajištění některých funkcionalit.

Hlavním takovým příkladem je *moc* kompilátor (meta-object compiler), který analyzuje zdrojový kód (konkrétně hlavičkové soubory tříd označené makrem *Q_OBJECT*), a z něj generuje soubory s některými informacemi o daných typech. Cílem je možnost se dynamicky dotazovat na vlastnosti daných typů za běhu programu (např. vyhledávání metod nebo atributů podle názvu). Zjednodušeně se jedná o náhradu reflexe, která je často podporována vysokoúrovňovými programovacími jazyky (např. Java), nicméně v jazyce C++ podporována není.

Dalším příkladem může být podpora překladů aplikace pomocí nástroje *linguist*, kdy jsou ze zdrojových (a dalších souborů) vyextrahovány přeložitelné řetězce (konkrétněji je nástroj popsán v sekci 5.3.4).

Celkový sestavovací proces Qt aplikace je tak složitější než pouhé linkování některých knihoven. Pro snadný vývoj Qt framework nabízí vlastní vývojové prostředí *QtCreator*, spolu s vlastním sestavovacím nástrojem *qmake*. Nicméně i nástroj CMake nabízí poměrně snadnou podporu pro sestavování Qt projektů. Pro nalezení knihoven a usnadněním linkování je možné použít příkaz `find_package`, například:

```
find_package(Qt5 REQUIRED COMPONENTS
        Widgets UiTools Test LinguistTools)
```

Pro podporu *moc* kompilátoru je možné nastavit vlastnost pro specifický cíl, kdy jsou potom automaticky analyzovány přiřazené zdrojové soubory (například `set_property(TARGET paracell-gui PROPERTY AUTOMOC ON)`).

5.3.2 Komunikace objektů

Podobně jako mnoho jiných GUI frameworků funguje komunikace uvnitř Qt aplikace na bázi „eventů“ (událostí). Při výskytu nějaké události (pohyb myši, stisknutí tlačítka myši) je informace automaticky propagována na místo, které

událost může obsloužit. Při vytváření události nemusí být známo, kdo ji zpracuje, stejně jako příjemce neví, kdo ji vyslal. Vlastní chování je pak možné implementovat v podtřídách pomocí vlastních virtuálních metod (například pro implementaci specifického „drag and drop“ chování jsou v podtřídách implementovány metody jako `dropEvent(QDropEvent*)`). Je samozřejmě možné vytvářet a posílat vlastní eventy (jako podtřídy `QEvent`).

Častěji používaným mechanismem je však specifický mechanismus signálů a slotů (*signals and slots*). Na úrovni kódu se jedná o specificky označené třídní metody, deklarované podobně jako standardní C++ přístupové specifikátory, např. `public:` versus `slot:`, `signal:`, nebo v kombinaci `public slot:`.

Tento mechanismus umožnuje instancím vyslat signál, načež dojde k zavolání všech připojených slotů. Oproti eventům tento mechanismus připomíná spíše návrhový vzor *observer*, kdy je nutné, aby se příjemce explicitně zaregistroval k upozornění na signál od nějaké instance. Jednoduchý příklad může vypadat následovně:

```
connect(
        button, SIGNAL(clicked()),
        this, SLOT(handleButtonClicked())
);
```

Proměnná `button` představuje instanci třídy `QAbstractButton`, která obsahuje signál `clicked`. Zároveň předpokládáme, že aktuální objekt (`this`) má slot s názvem `handleButtonClicked`. Slot musí signaturou (přijímanými argumenty) odpovídat danému signálu (signály je možné předávat různá data). Takto je možné k jednomu objektu, i ke stejnému signálu, vytvořit libovolný počet propojení s různými sloty. Po kliknutí na tlačítko jsou postupně synchronně zavolány všechny připojené sloty (oproti obecně asynchronnímu zpracování eventů). Do jisté míry se jedná o jednoduchou registraci callbacků.

Signály a sloty se využívají prakticky ve všech komunikacích mezi různými grafickými prvky, typicky se nějaký kontrolér připojí ke spravovaným prvkům uživatelského prostředí.

5.3.3 Návrh rozhraní

Qt samozřejmě umožňuje celé grafické rozhraní napsat pouze na úrovni zdrojového kódu, nicméně pro dynamičtější vývoj je nabízen grafický návrhář *QtDesigner*. Vytvořený návrh je možné následně exportovat do *.ui* souborů (ve formátu XML), které lze následně za běhu programu načíst pomocí třídy `QUiLoader`. Jednotlivým prvkům je možné upravovat jejich atributy a vytvářet propojení mezi sloty a signály.

V tomto ohledu je zajímavá možnost deklarace vlastních podtříd existujících grafických tříd, u kterých je poté možné definovat vlastní signály a sloty. Pro korektní načtení vlastních podtříd ze souborů je rovněž nutné vytvořit vlastní podtřídu `QUiLoader`, která bude schopna tyto instance vytvářet.

Přestože implementace samotného chování musí existovat na úrovni zdrojového kódu, samotné propojení je možné provádět pouze pomocí grafického návrháře. K tomu je využíván dříve zmíněný meta-object kompilátor, pomocí kterého lze za běhu u jednotlivých instancí najít jednotlivé signály a sloty pouze podle textového pojmenování. V případě, že se nepodaří vytvořit specifikovanou podtřídu nebo neexistuje specifikovaný slot/signál, aplikace pouze na chybu upozorní, ale sama zůstane stále funkční (místo podtřídy se použije její nadtřída a vadná propojení se ignorují).

Všechny obrazovky v aplikaci jsou vytvořeny tímto návrhářem, s cílem minimalizovat rigiditu zdrojového kódu a usnadnění budoucích úprav.

5.3.4 Podpora překladů

Jak již bylo zmíněno, Qt framework umožnuje snadno vytvářet překlady aplikace pomocí nástroje *QtLinguist*. Základem je extrakce všech textů k přeložení ze samotného zdrojového kódu a z .ui souborů v době kompilace. Ve zdrojovém kódu jsou všechny řetězce k přeložení označeny obalením do funkce `tr()`, kterou obsahuje každá *Q_OBJECT* třída. V CMake je možné použít příkaz `qt5_create_translation`, kterému se předá seznam souborů k analýze a výstupem jsou soubory s příponou *.ts* pro každý překládaný jazyk.

Tyto soubory je následně možné otevřít v samotném *QtLinguist* grafickém nástroji, ve kterém je možné pro každý řetězec poskytnout překlad. Z jednotlivých překladů jsou následně vytvořeny *.qm* soubory, které je možné nahrát za běhu do instance třídy `QTranslator`, kterou je možné aplikovat pomocí funkce `installTranslator` hlavní třídy `QApplication`. Při vytváření grafických objektů pak budou řetězce automaticky překládány (pokud pro nějaký řetězec překlad chybí, je použita výchozí hodnota).

5.3.5 Konfigurační šablony

Nabízené šablonové soubory se skládají z výchozích, předpřipravených souborů a uživatel má možnost své vlastní šablony vložit do složky *templates*, ze kterých jsou automaticky načítány všechny (korektní) soubory.

Šablony jsou automaticky za běhu aplikace nahrány znovu, pokud dojde ve složce k nějaké změně (odebrání, přidání nebo úprava souboru). K tomu framework nabízí třídu `QFileSystemWatcher`, která vyšle signál, pokud dojde ke změnám ve specifikovaných cestách.

Výchozí konfigurace nejsou dostupné ve formě souborů, ale jsou vestavěné přímo v binárních souborech. Hlavních důvodem je zajištění jejich stálé dostupnosti (aby například uživatel soubory omylem neodstranil nebo neupravil). Vzhledem k tomu, že standard C++ nenabízí možnost nějakým způsobem vložit obsahu souborů do proměnné[3], je toto chování delegováno na vlastní CMake příkaz, který před sestavením ze specifikovaných šablonových souborů

[3]Standard C23 (nicméně ne C++23) pro takové použití specifikuje direktivu `#embed`

vygeneruje C++ zdrojový kód, kde je obsah souborů uložen do proměnné jako pole řetězců.

5.4 Úpravy kódu

V předchozích sekcích byly popsány některé způsoby, jakým byl existující zdrojový kód přepracován, především z vyššího architektonického pohledu. Tato část se více věnuje granulárním změnám, s využitím analytických nástrojů ze sekce 3.6.

Prioritou bylo především odstranění varování od kompilátorů (použity byly *GNU G++* pro Linux a *MSVCC* pro Windows). Zvláště *MSVC* překladač je schopný poskytnout velké množství různých upozornění. Mezi hlavními úpravami byly:

- Konzistentní typy různých proměnných při jejich přiřazování nebo porovnávání, především použití float/double pro zachování přesnosti nebo použití znaménkových a bezznaménkových celočíselných typů. Změny byly provedeny pouze na některých částech kódu, například různé algoritmické funkce často používají různé přesnosti plovoucích čísel, kde by změny byly zbytečně náročné.

- Formátování použitých *printf*, *scanf* a podobných funkcí.

- Odstranění nedosažitelného kódu ve funkcích (např. umístění za návrat nebo ve větvi, která nemůže nastat).

- Korektní inicializace proměnných (to někdy vyžaduje i úpravu OpenMP direktiv při specifikaci privátních proměnných vláken).

- Vyřazení některých nepoužívaných proměnných a parametrů funkcí.

Některé z varování byly záměrně ignorovány, např. zarovnání(*padding*) u tříd, který by bylo možné minimalizovat přeuspořádáním atributů, ale zároveň mají třídy minimální vliv na používanou paměť.

Dále byly použity analyzátory *ClangSA*, *cppcheck* a *clazy*. Dále bude provedeno srovnání jednotlivých nástrojů a jaké problémy pomohly vyřešit.

Nástroj *clazy* se zaměřuje na vyhledávání problémů v *Qt*. Nalezeno bylo několik problémů, ze kterých však žádné nebyly kritické.

- Neideální propojení signálů, kdy v některých případech (zde především při použití lambda funkcí) nebyl specifikován kontext příjemce (propojení je automaticky odpojeno, pokud je kontext příjemce nebo odesílatele destruován). To může v určitých situacích způsobit paměťově nekorektní chování. Konkrétně při použití v programu k problémům nedocházelo, nicméně z důvodů bezpečnosti je vhodnější kontext specifikovat.

- Zbytečné překopírování (např. při *for-each* cyklech) netriviálních knihovních typů místo použití referencí.
- Vlastní datové typy použité v signálech a slotech nebyly plně kvalifikované (neměli specifikované jmenné prostory). To by mohlo způsobit problémy při konfliktu jmen tříd v jiných jmenných prostorech.
- Špatné použití klíčového slova *emit* pro vyslání signálu na některých místech.

Nástroje *ClangSA* a *cppcheck* byly použity v rámci frontendu *CodeChecker*. Povoleny byly všechny dostupné ověřovací kategorie. Každý z nástrojů je možné individuálně konfigurovat, nicméně i ve výchozím nastavení poskytují poměrně komplexní analýzu projektu. Povoleny byly všechny dostupné kategorie ověření. Kromě těchto dvou nástrojů automaticky používá *CodeChecker* i nástroj *clang-tidy*, který je později rovněž použit samostatně pro konzistentní pojmenovávání. Hlavními nalezenými problémy byly:

- Použití virtuální metody v konstruktoru, což je chování nedefinované standardem a závisí na konkrétní implementaci.
- Atributy třídy neinicializované v konstruktoru.
- Přístup ke statickému poli mimo rozsah (odvozením možných hodnot přístupové proměnné).
- Nesmyslné logické operace (podmínky které vždy platí/neplatí, opakující se podmínky se stejným výsledkem).
- Potenciální použití *null* třídního objektu.
- Absence virtuálního destruktoru pro virtuální třídu.
- Vyhození výjimky v paralelním *OpenMP bloku*.

Kromě možných problematických varování nástroje upozorňují na různé stylistické úpravy, například:

- Nekonzistentní pojmenování parametrů v deklaraci/definici.
- Označení metod, které mohou být konstantní.
- Proměnné, které mohou mít užší rozsah platnosti.
- Absence závorek za výrazem (především *if*).
- Možné nahrazení části kódu některými standardními algoritmickými funkcemi (typicky operace pracující nad prvky nějaké kolekce).

Kromě změn podle analytických nástrojů byly některé části modernizovány pro novější verze standardu jazyka C++, jak bylo zmíněno v sekci 3.1.5. Provedeno bylo například:

- Práce se soubory byla přepracována s použitím souborových proudů (*file stream*) místo Cčkových souborů. To umožňuje jednodušší a bezpečnější práci, např. načítáním do *std::string* instancí.
- Snaha využít chytré ukazatele (*unique_ptr*, *shared_ptr*) namísto standardních dynamických alokací. Související je snaha využít RAII princip (tedy konstruktory a destruktory), zmíněná v návrhové kapitole, ve snaze zabránit různých chybám související s různými inicializačními a deinicializačními funkcemi.
- Úpravy přetypovávání v jazyce C na standardní C++ výrazy (především *static_cast*).
- Použití referencí pro předávání argumentů místo jednoduchých ukazatelů. To zlepšuje bezpečnost zajištěním předávání korektních instancí (bez ověřování *nullptr*).
- Upravení některých zastaralých algoritmických funkcí (např. std::sort místo qsort).

5.4.1 Clang-tidy integrace

Kromě funkčních úprav byl samostatně použit nástroj *Clang-tidy* pro podporu dodržování konzistentních pojmenovávání. Vzhledem k závislosti na kompilátoru *clang* (potažmo celé platformě *LLVM*) je pro podporu třeba specifikovat proměnnou *USE_CLANG_TIDY* při CMake inicializaci. V systémové cestě pak musí být nalezen clang-tidy, spustitelný soubor.

Při kompilaci bude v kořenovém adresáři zdrojových souborů vytvořen spustitelný soubor *clang-tidy-runner*, který po spuštění ve výchozím stavu spustí clang-tidy na všechny zdrojové soubory (případně lze pomocí přepínače *-f* filtrovat vstupní soubory pomocí regulárního výrazu).

Konfigurace je specifikovaná v souboru *.clang-tidy* ve formátu Json (byť standardnější bývá formát .yml). Clang-tidy konfigurace funguje specifikací „ověření“ (*checks*) která mají být spuštěna. To je vidět na řádku s klíčem ”Checks”, jehož hodnotou je středníky oddělený seznam. Každé ověření má následně množství dodatečných nastavení, které lze konfigurovat. Tomu v souboru odpovídá klíč „CheckOptions“, který je polem dvojic. Každá dvojice specifikuje klíč možnosti, která má být nastavena. Například:

```
{
 "Checks": "readability-identifier-naming",
 "CheckOptions": [
```

```
    {
     "key" : "readability-identifier-naming.ClassCase",
     "value" : "CamelCase"
    },
    {
     "key" : "readability-identifier-naming.ClassMemberCase",
     "value" : "lower_case"
    }
  ]
}
```

Jednotlivá nastavení jsou rozdělena podle kategorií identifikátorů, například *Class*, *Struct*, *ClassMember*, *Function*. U každé kategorie je možné zvlášť nastavit:

- *Case*, základní ověřovaná jmenná konvence u identifikátorů, například *CamelCase*, *lower_case*, *UPPER_CASE*.
- *Prefix* a *Suffix* jako požadovaná předpona, respektive přípona.
- *IgnoredRegexp*, regulární výraz, pomocí kterého lze specifikovat výrazy, na které se nemají aplikovat ostatní pravidla.

5.5 Dokumentace

K původnímu i rozšířenému zdrojovému kódu byly přidány komentáře, jak k samotnému rozhraní tak k některým implementačním detailům. Kromě okomentování samotného kódu byla přidána podpora pro generaci externí dokumentace pomocí systému *Doxygen*, který je často používaný pro jazyk *C++*.

Komentáře pak mohou být obohaceny o tagy pro okomentování specifických částí (např. *param* pro okomentování parametrů funkce, *return* pro popis návratové hodnoty a mnohé další). Extrahovanou dokumentaci je možné vizualizovat ve formátu webových stránek.

Kapitola 6

Testování

6.1 Automatické testy

Součástí práce bylo vytvoření automatizovaných testů, které v původním projektu chyběly a které mohou usnadnit budoucí vývoj. Primárně byly vytvořeny testy pro ověření, že při úpravě aplikace nedošlo ke změně chování, tedy jestli výsledky výpočtů odpovídají původní verzi (nazývané jako „regresní“ testy).

Samotné testy byly rozděleny do dvou samostatných částí, jednak pro samotnou výpočetní část a dále pro automatické testy grafického rozhraní.

6.1.1 Testování výpočetní aplikace

Pro testování hlavní části byla použita knihovna *googletest* od společnosti Google. Ta je mimo jiné snadno integrovatelná s CMake, kdy je automaticky stažen a zkompilován zdrojový kód z repozitáře. Umožňuje snadno definovat jednotlivé testy pomocí makra *TEST* spolu s názvy testu a skupiny, není nutné implementovat vlastní *main* funkci. Spustitelný soubor poté umožňuje filtrovat spouštěné testy podle jejich názvu pomocí přepínače `--gtest_filter`.

Hlavní část testů tvoří jednotkové (*unit*) testy, které se soustředí na testování malé, samostatné části kódu (typicky jedné třídy nebo jednoho souboru). Jednotkové testy byly vytvořeny mimo jiné pro:

- Zpracování argumentů z příkazové řádky (třída `ArgParser`).
- Třídy reprezentující konfigurační (`Configuration`), datové (`Measured`) a dávkové (`BatchConfig`) soubory. Testováno je především korektní načítání ze souborů, a zpětná serializace do řetězcové podoby. U třídy `Configuration` je rovněž testována korektní normalizace při sémanticky redundantních datech.
- Jednotlivá grafická API, pro každé z nich jsou všechny grafické funkce spouštěny s předpřipravenými daty a výstup je překontrolován vůči

předpokládaným výsledkům. Zde existuje riziko mírných odchylek vzhledem k odlišnostem jednotlivých technologií, například Vulkan podporuje pouze goniometrické funkce s 32bitovou přesností a poskytuje tedy lehce jiné výsledky než CUDA a Vulkan.

Pro ověření funkčnosti celé aplikace byly vytvořeny systémové testy. Celá aplikace je spouštěna s předpřipravenými konfiguračními soubory a je kontrolován jejich výsledek – k tomu jsou používány výstupní logovací soubory, do kterých jsou zaznamenávány všechny nalezená řešení s jejich parametry.

6.1.2 Testování grafického prostředí

Pro testování nabízí Qt framework vlastní modul *QTest*, který je použit pro automatické testování grafického rozhraní. Pro základní použití lze použít makro `QTEST_MAIN` s předaným názvem třídy, následně je jako test spouštěna každá metoda dané třídy. Testy je možné opět filtrovat předáním názvů metod v příkazové řádce.

Jmenný prostor QTest nabízí funkce pro simulaci uživatelského chování, například `keyClick` pro stisk klávesy, `mouseMove` pro posunutí kurzoru myší, `mousePress` a `mouseRelease` pro stisknutí tlačítek na myši. Zároveň je možné přímo využívat metod samotných grafických prvků pro získání a nastavování hodnot. Jednotlivé prvky lze snadno vyhledávat pomocí šablonové metody `findChild`, která podle jména může rekurzivně najít libovolný prvek v hierarchii.

Snahou je testovat veškeré dostupné funkcionality formou podobnou jednotkovým testům, nicméně různá provázanost mezi jednotlivými testy je nevyhnutelná (např. úprava souboru přes otevřenou kartu vyžaduje jeho korektní načtení do seznamu). Testováno je:

- Vytváření, načítání a uzavírání souborů a správné chování: správný stav seznamu souborů, upozornění na neuložené změny, zavření karet uzavřených souborů.
- Otevírání záložek otevřených souborů a celková práce se záložkami (horizontální dělení seznamu, přesun záložek mezi rozděleními, opětovné sjednocení).
- Správné uložení stavu otevřených souborů a karet, správné načtení po spuštění.
- Úprava všech typů souborů přes otevřené karty, kontrola správné reakce aplikace na změnu souborů, korektní uložení. U konfiguračních souborů kontrola reakce grafického rozhraní (skrytí některých parametrů podle zvolené metody, omezení prohledávaných rozsahů podle krystalické mřížky).

- Otevření karty pro výpočet, zvolení argumentů, spuštění výpočtu a zastavení výpočtu. Kontrola maximálně 1 otevřené karty jednoho výpočtu najednou, správná reakce prvků po spuštění výpočtu.

6.2 Uživatelské testování

Pro zjištění kvality návrhu bylo uživatelské prostředí otestováno na několika uživatelích, s cílem zjistit, jestli jsou všechny funkce dostatečně intuitivní a zdali se uživatelé dokáží samostatně orientovat.

Pro testování byly sestaveny scénáře (dostupné v příloze E), které uživatele postupně provedou celou aplikací a prezentují jednotlivé problémy k vyřešení. Scénáře se skutečně snaží pouze uvést nějaký problém a nechat uživatele zjistit, jak dospět k řešení.

Aplikace byla testována na 3 uživatelích, z nichž 2 byly na úrovni běžných počítačových uživatelů a 1 na technicky pokročilé úrovni. Před testováním byla každému uživateli zjednodušeně představena doména a jaké problémy aplikace řeší. Vzhledem k tomu, že se jedná o specializovaný software, není příliš reálné uživatelům detailně popsat jednotlivé části. Například datové soubory jsou nejasné kolekce řádků, u konfigurací není ani z popisů jasné, co které hodnoty znamenají. To sice může mít negativní dopad na testování, nicméně základní funkce by měli být dostatečně pochopitelné, aby aplikaci mohl uživatel vyzkoušet (přestože uživatel neví, co soubory reprezentují, tak koncept jejich otevření, úprav a jejich použití pro spuštění by měl být jasný).

Následuje představení hlavních problémů, které byly při testování nalezeny. U každého z nich je rovněž popsáno, jakým způsobem byl vyřešen.

- Při vytvoření nového konfiguračního souboru byl všem uživatelům nejasný výběr šablony – nevěděli, co na obrazovce dělají. Jako řešení byl na dialog přidán popisný text, informující uživatele že vybírá obsah vytvářeného souboru.

- Při ukládání souboru nebo používání krokování zpět/vpřed není dostatečně jasné, jaký soubor je aktuálně aktivní – především pokud je najednou otevřeno více souborů vedle sebe. Pro vylepšení je záložka aktuálního souboru tučně zvýrazněna a název aktuálního souboru zobrazován v názvu okna aplikace.

- Jeden uživatel se pokusil uložit soubor přes kontextové menu souboru v seznamu, kde chyběla. Možnost uložit soubor byla do menu přidána.

- Při úpravě dávkových souborů chyběli tlačítkům pro ovládání seznamu konfigurací vyskakovací popisky – kromě ikonek nebylo jasné, co tlačítka dělají. K tlačítkům byly popisy přidány.

- Jeden z uživatelů při úpravě dávkového souboru zkoušel přidat konfiguraci přetažením souboru, ale nepustil soubor nad seznamem. Podobně jako u výpočetní obrazovky bylo k seznamu při tažení přidáno okénko navádějící uživatele.

- Žádný z uživatelů samovolně nepřišel na možnost horizontálně rozdělit plochu a většina z nich nepřišla na možnost přidat soubory pro výpočet přetažením. Pro informování uživatele u této možnosti byly přidány jednoduché „výukové“ dialogy, které upozorní uživatele na tyto možnosti. Každý z dialogů je uživateli zobrazen pouze jednou, tedy nijak nenarušuje efektivitu práce.

6.3 Srovnání výkonu

V této sekci budou provedena dvě hlavní srovnání výkonu – první pro srovnání s původní verzí (zdali nedošlo k nějakému poklesu výkonu) a druhé pro porovnání jednotlivých implementací grafických API, zdali nově implementované technologie časově odpovídají původní CUDA implementaci.

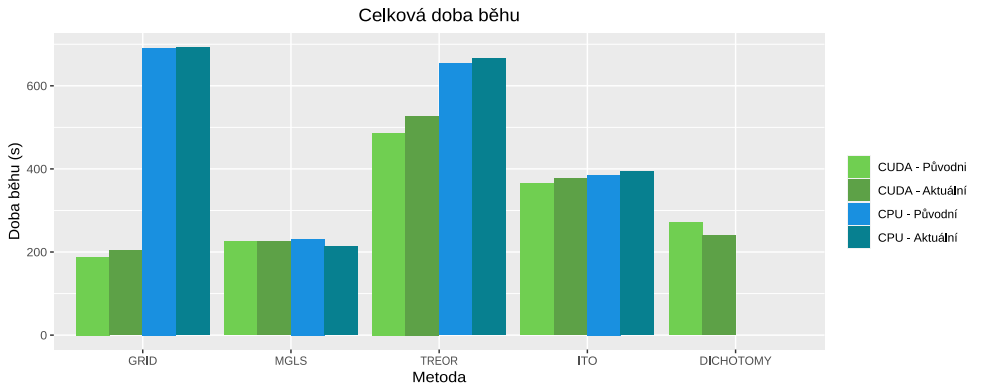
Testováno bylo všech 5 implementovaných metod. Kromě *DICHOTOMY* metody všechny ostatní metody používají pouze stejnou ověřovací metodu kandidátů, tedy by se u nich dalo předpokládat podobné chování při použití různých technologií. Pro zkvalitnění výsledků byly časy průměrovány přes 5 běhů. Jednotlivé konfigurace metod byly nastaveny tak, aby měli přibližně stejnou dobu výpočtu (není tím však možné srovnávat kvalitu mezi jednotlivými metodami).

Pro srovnání širšího rozpětí hardwaru bylo identické testování provedeno na 2 různých platformách (notebook[4] s podporou CUDA, stolní počítač[5] bez podpory CUDA). Kromě grafických karet byly GPGPU technologie testovány i na grafických procesorech integrovaných spolu s CPU. U DICHOTOMY metody nejsou podporované CPU výpočty, naměřeny jsou tedy pouze grafické výsledky. Pro tuto metodu jsou všechny 3 rozhraní ve výpočtech prakticky ekvivalentní. U integrovaných GPU nejsou pro metodu výsledky naměřeny, protože nepodporují 64bitové výpočty v plovoucí řádce.

Výsledky prvního testování srovnávající původní a aktuální verzi aplikace jsou dostupné v tabulce 6.1 s grafickým znázorněním na grafu 6.1. Měřeny byly pouze CPU a CUDA výpočty, použita tak byla pouze první platforma. Na výsledcích je možné vidět, že doba výpočtu je ve všech případech prakticky ekvivalentní. V této oblasti tedy k regresi při přepracování aplikace nedošlo.

Výsledky druhého testování jsou vizualizované na grafech pro obě sestavy (6.2, 6.3). Na grafech je kromě celkového výpočetního času vizualizován i čas pro ověřování kandidátů (tedy přímé srovnání jednotlivých grafických rozhraní

[4]Intel Core i5-8250U, Nvidia GeForce MX150, 8 GB DDR4 RAM 2133MHz
[5]Intel Core i5-13500, AMD Radeon RX 570, 32 GB DDR4 RAM 3200MHz



Obrázek 6.1: Srovnání celkové doby běhu mezi původní a předělanou verzí.

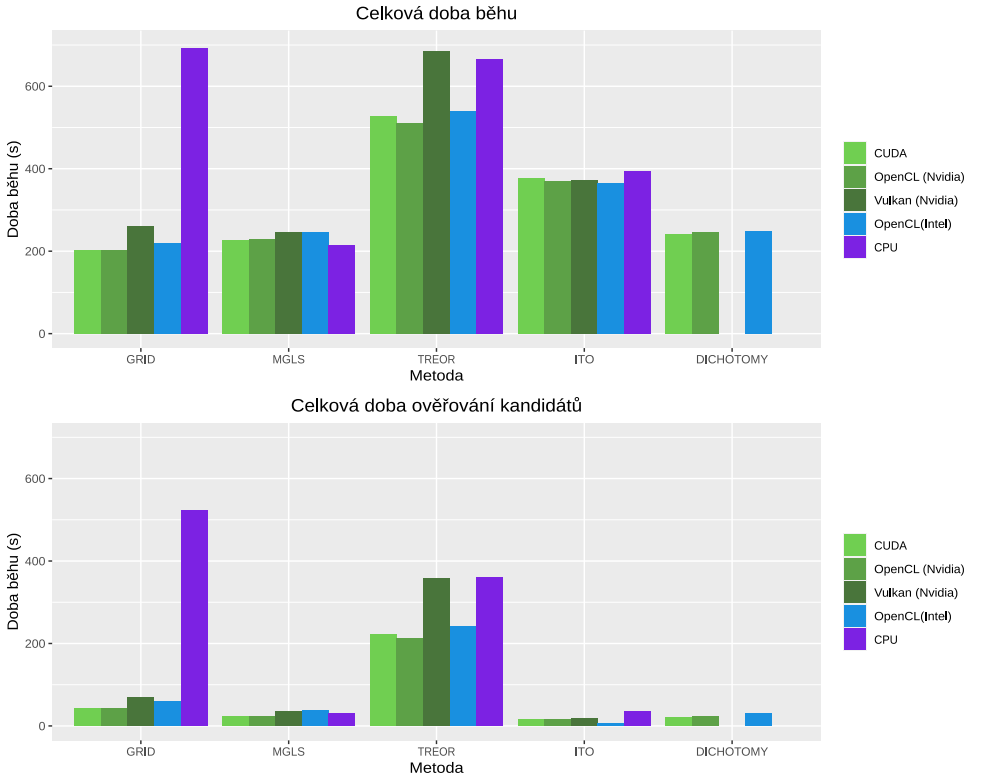
	GRID	MGLS	TREOR	ITO	DICH
CUDA - Původní	187.89	225.28	486.41	365.00	270.24
CUDA - Aktuální	203.42	226.11	527.14	376.76	240.64
CPU - Původní	691.01	231.17	653.70	385.43	-
CPU - Aktuální	692.61	213.77	664.88	394.59	-

Tabulka 6.1: Srovnání doby běhu mezi původní a přepracovanou verzí.

spolu s CPU). Přesné hodnoty jsou zpracované v tabulkách 6.2 a 6.3. Různé doby běhu jednotlivých metod nereflektují kvalitu jejich výpočtu.

Z výsledků je možné vidět, že nejvíce z GPU výpočtů benefitují metody GRID a TREOR, naopak vylepšení u metod ITO a MGLS je výrazně menší. Jednoduché vysvětlení je, že první 2 metody generují více kandidátů, naopak další metody komplexněji vybírají menší množství kandidátů. Nicméně všechny metody jsou limitovány jednovláknovým procesorových výkonem při hledání kandidátů, který není paralelizovaný. U těchto 4 metod je možné vidět, že CUDA a OpenCL jsou časem výpočtů prakticky ekvivalentní. Vulkan už je přibližně 2krát až 3krát pomalejší, nicméně nabízí stále nabízí rychlejší výpočet než přes procesor. U první platformy se slabším procesorem je možné vidět i výhodu, kterou mohou nabídnout integrované grafické procesory, které mohou výrazně snížit čas výpočtu. Naopak u stolního výpočtu s vysokým paralelním výkonem hlavního procesoru je použití integrované GPU i regresivní.

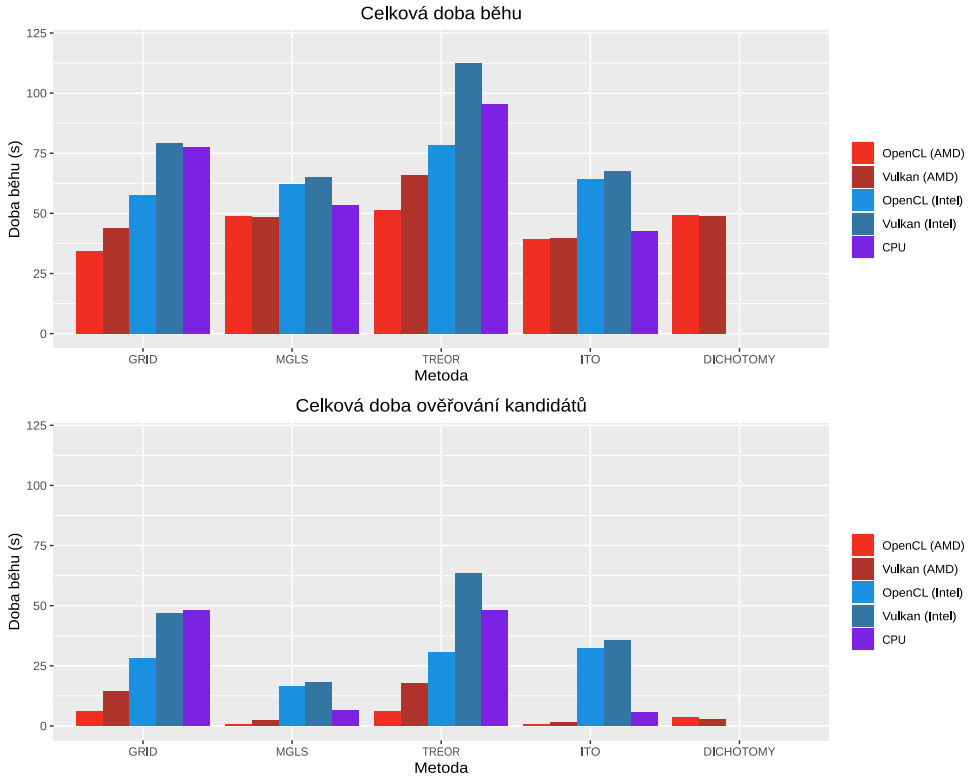
Celkově se dá implementace OpenCL hodnotit v porovnání se stávající CUDA pozitivně, nabízející prakticky identický výkon. Vulkan vykazuje dobré výsledky u *DICHOTOMY* metody, nicméně u ostatních metod jsou výsledky výrazně horší na všech měřených platformách a to i přes značnou osobní snahu pro zlepšení výkonu. Pro případný budoucí rozvoj grafických výpočtů se i díky výrazně náročnějšímu vývoji zdá jako neperspektivní.



Obrázek 6.2: Měřená doba běhu napříč metodami, první sestava.

		GRID	MGLS	TREOR	ITO	DICH
Celkem	CUDA	203.42	226.11	527.14	376.76	240.06
	OpenCL Nvidia	203.60	229.10	510.44	370.68	246.00
	Vulkan Nvidia	260.07	246.51	686.23	372.42	-
	OpenCL Intel	219.01	245.42	538.88	364.61	248.16
	CPU	692.61	213.77	664.88	394.59	-
Ověření kandidátů	CUDA	40.89	23.68	224.79	16.13	20.34
	OpenCL Nvidia	41.59	22.61	212.86	15.57	21.94
	Vulkan Nvidia	70.05	35.93	357.44	18.48	-
	OpenCL Intel	59.80	38.32	240.99	5.98	31.12
	CPU	523.64	30.02	361.04	34.45	-

Tabulka 6.2: Naměřená doba běhu na první sestavě.



Obrázek 6.3: Měřená doba běhu napříč metodami, druhá sestava.

		GRID	MGLS	TREOR	ITO	DICH
Celkem	OpenCL AMD	34.46	48.89	51.53	39.45	49.43
	Vulkan AMD	44.01	48.56	65.69	39.87	48.92
	OpenCL Intel	57.58	62.15	78.47	64.18	-
	Vulkan Intel	79.21	64.99	112.65	67.71	-
	CPU	77.69	53.63	95.56	42.77	-
Ověření kandidátů	OpenCL AMD	5.97	0.82	5.93	0.51	3.68
	Vulkan AMD	14.36	2.22	17.79	1.50	2.57
	OpenCL Intel	28.04	16.21	30.37	32.05	-
	Vulkan Intel	46.64	18.24	63.21	35.62	-
	CPU	48.22	6.54	47.88	5.62	-

Tabulka 6.3: Naměřená doba běhu na druhé sestavě.

Závěr

V rámci této práce byla analyzována a vylepšena existující aplikace sloužící pro indexaci práškové difrakce. Nejdříve byly představeny teoretické základy z oblastí krystalografie a práškové difrakce, důležitých pro pochopení principu aplikace. V následující analytické části byl popsán aktuální stav projektu, požadavky na hlavní část aplikace a požadavky na samostatné grafické rozhraní, které doplňuje existující terminálovou aplikaci.

Grafické rozhraní bylo implementováno jako nativní aplikace v multiplatformním frameworku Qt a splňuje vytyčené funkční a nefunkční požadavky. Pro výpočty na grafických kartách byly k technologie CUDA dále implementovány hardwarově nezávislé technologie OpenCL a Vulkan, ze kterých se především OpenCL jeví jako dobrou alternativou. Pro zjednodušení sestavování byl integrován systém CMake, který zjednodušuje integraci s použitými technologiemi (OpenMP, CUDA, OpenCL, Vulkan, Qt), umožňuje manuální výběr sestavovaných částí projektu (jednotlivé GPU technologie, grafické rozhraní, testy, ...) a zajišťuje vestavění některých dat do binárních souborů. Zdrojový kód aplikace byl zdokumentován a přepracován tak, aby lépe odpovídal moderním standardům jazyka C++.

Pro finální aplikaci byly vytvořeny automatické jednotkové testy pro hlavní i grafickou část, spolu s komplexnějšími systémovými testy ověřující samotné výstupy z kompletního průběhu výpočtu. Rovněž bylo provedeno uživatelské testování, na základě kterého bylo grafické rozhraní ještě dodatečně upraveno. Pro další ověření kvality byly rovněž na 2 různých platformách provedeny výkonové testy používající všechny implementované metody, s cílem ověřit, že nedošlo k regresi výkonu při přepracování aplikace. Mezi sebou byly otestovány jednotlivé GPU technologie pro ověření kvality jejich implementací. Pro uživatelské použití byly vytvořeny instalační a uživatelské příručky.

S přihlédnutím ke všem vypracovaným částem lze cíle práce považovat za splněné. Na přiloženém fyzickém médiu je dostupný zdrojový kód celé aplikace, spolu s přeloženou aplikací pro systém Windows.

Literatura

[1] Kraus, I.; Fiala, J.: *Krystalografie*. ČVUT, 2021, ISBN 978-80-01-06868-7.

[2] Kříž, D.: *Úvod do krystalografie a strukturní analýzy*. 2000. Dostupné z: https://www.xray.cz/krystalografie/

[3] Dinnebier, R.; Simon, A.: Principles of Powder Diffraction. 02 2023. Dostupné z: https://www.researchgate.net/publication/264843670_Principles_of_Powder_Diffraction

[4] Altomare, A.; Cuocci, C.: Indexing a powder diffraction pattern. 2018. Dostupné z: https://onlinelibrary.wiley.com/iucr/itc/Ha/ch3o4v0001

[5] OMG® Unified Modeling Language® (OMG UML®). 2.5.1, 2017, https://www.omg.org/spec/UML/2.5.1/PDF.

[6] Šimeček, I.; Rohlíček, J.; Zaloga, A.: Overview of ParaCell package for indexing in powder diffraction. *Journal of Applied Crystallography*, ročník 56, č. 1, 2023: s. 293–301, doi:https://doi.org/10.1107/S1600576722011554, https://onlinelibrary.wiley.com/doi/pdf/10.1107/S1600576722011554. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1107/S1600576722011554

[7] Šimeček, I.; Rohlíček, J.; Zahradnický, T.; aj.: A new parallel and GPU version of a TREOR-based algorithm for indexing powder diffraction data. *Journal of Applied Crystallography*, ročník 48, 02 2015, doi:10.1107/S1600576714026466.

[8] Louër, D.; Boultif, A.: *Powder pattern indexing and the dichotomy algorithm*. Oldenbourg Wissenschaftsverlag, 2007, ISBN 9783486992540, s. 191–196, doi:doi:10.1524/9783486992540-030. Dostupné z: https://doi.org/10.1524/9783486992540-030

[9] Boultif, A.; Louër, D.: Indexing of powder diffraction patterns for low-symmetry lattices by the successive dichotomy method. *Journal of Applied Crystallography*, ročník 24, č. 6, Dec 1991: s. 987–993, doi: 10.1107/S0021889891006441. Dostupné z: https://doi.org/10.1107/S0021889891006441

[10] Ron Ghosh: *Crysfire2020*. [online], [cit. 2023-03-23]. Dostupné z: http://ccp14.cryst.bbk.ac.uk/Crysfire.html#DVD

[11] OpenMP: *Home - OpenMP*. [online], [cit. 2023-04-22]. Dostupné z: https://www.openmp.org/

[12] Google: *Bazel*. [online], [cit. 2023-04-22]. Dostupné z: https://bazel.build/

[13] The Qt Company: *qmake Manual*. [online], [cit. 2023-04-22]. Dostupné z: https://doc.qt.io/qt-6/qmake-manual.html

[14] Gradle Inc.: *Gradle Build Tool*. [online], [cit. 2023-04-22]. Dostupné z: https://gradle.org/

[15] JetBrains s.r.o.: *C++ Programming - The State of Developer Ecosystem in 2022*. [online], [cit. 2023-04-22]. Dostupné z: https://www.jetbrains.com/lp/devecosystem-2022/cpp/

[16] NVIDIA Corporation: *CUDA Toolkit*. [online], [cit. 2023-03-06]. Dostupné z: https://developer.nvidia.com/cuda-toolkit

[17] Khronos Group: *OpenCL Overview*. [online], [cit. 2023-03-06]. Dostupné z: https://www.khronos.org/opencl/

[18] Khronos Group: *Vulkan*. [online], [cit. 2023-03-06]. Dostupné z: https://www.vulkan.org/

[19] Advanced Micro Devices, Inc.: *AMD ROCm*. [online], [cit. 2023-03-06]. Dostupné z: https://rocmdocs.amd.com/en/latest/

[20] Intel Corporation: *oneAPI Programming Model*. [online], [cit. 2023-03-06]. Dostupné z: https://www.oneapi.io/

[21] Khronos Group: *SYCL Overview*. [online], [cit. 2023-03-06]. Dostupné z: https://www.khronos.org/sycl/

[22] Apple Inc.: *Metal Overview*. [online], [cit. 2023-03-06]. Dostupné z: https://developer.apple.com/metal/

[23] The Qt Company: *Qt*. [online], [cit. 2023-03-06]. Dostupné z: https://www.qt.io/

[24] wxWidgets: *wxWidgets: Cross-Platform GUI Library*. [online], [cit. 2023-03-06]. Dostupné z: https://www.wxwidgets.org/

[25] Perforce Software, Inc: *Klocwork*. [online], [cit. 2023-03-26]. Dostupné z: https://www.perforce.com/products/klocwork

[26] CoderGears / CppDepend: *CppDepend*. [online], [cit. 2023-03-26]. Dostupné z: https://www.perforce.com/products/klocwork

[27] PVS-Studio LLC: *PVS-Studio*. [online], [cit. 2023-03-26]. Dostupné z: https://pvs-studio.com/en/pvs-studio/

[28] *Cpplint*. [online], [cit. 2023-03-06]. Dostupné z: https://github.com/cpplint/cpplint

[29] *Cppcheck*. [online], [cit. 2023-03-06]. Dostupné z: https://cppcheck.sourceforge.io/

[30] The Clang Team: *Clang-Tidy*. [online], [cit. 2023-03-06]. Dostupné z: https://clang.llvm.org/extra/clang-tidy/

[31] *include-what-you-use*. [online], [cit. 2023-03-06]. Dostupné z: https://include-what-you-use.org/

[32] *CodeChecker*. [online], [cit. 2023-03-06]. Dostupné z: https://codechecker.readthedocs.io/en/latest/

[33] Figma, Inc.: *Figma: the collaborative interface design tool*. [online], [cit. 2023-03-17]. Dostupné z: https://www.figma.com/

[34] Kitware Inc.: *CMake*. [online], [cit. 2023-03-17]. Dostupné z: https://cmake.org/

[35] LunarG, Inc.: *Vulkan SDK*. [online], [cit. 2023-03-18]. Dostupné z: https://www.lunarg.com/vulkan-sdk/

[36] The Qt Company: *Qt for Open Source Development*. [online], [cit. 2023-03-06]. Dostupné z: https://www.qt.io/download-open-source

[37] Microsoft: *vcpkg*. [online], [cit. 2023-03-06]. Dostupné z: https://vcpkg.io/en/index.html

Příloha A

Seznam použitých zkratek

CPU Central processing unit

CUDA Compute Unified Device Architecture

GPU Graphical processing unit

GUI Graphical user interface

OpenCL Open Computing Language

RAII Resource acquisition is initialization

Příloha **B**

Instalační příručka

Tato část slouží jako návod pro konfiguraci projektu pomocí nástroje CMake pro jeho sestavení. Návod bude cílen na operační systémy Windows a Ubuntu.

Základním předpokladem je instalace nástroje CMake. V systému Ubuntu (a pravděpodobně na většině Linuxových distribucí) je dostupný jako oficiální balíček, pro Windows je možné jej nainstalovat z oficiálních stránek [34]. Dále se předpokládá, že je možné nástroj spustit jednoduše pomocí příkazu *cmake* z příkazové řádky. CMake rovněž nabízí grafické prostředí, zde se však předpokládá použití pouze příkazové řádky.

CMake umožňuje konfigurovat projekt pro různé systémy (nazývány generátory), které je možné specifikovat přepínačem *-G*. Seznam dostupných generátorů na aktuálním systému je možné vypsat pomocí příkazu `cmake --help`. Specifikace generátoru tak může vypadat například:

```
cmake -G "Unix Makefiles" .
```

Projekt byl na Ubuntu testován pomocí Makefile souboru s GNU G++ kompilátoru, na Windows bylo použito vývojové prostředí Visual Studio 2019, nicméně by neměl být problém použít i jiné systémy.

Pro nejjednodušší konfiguraci stačí spustit příkaz (`cmake .`), kdy bude projekt nakonfigurován pro výchozí generátor. Zde může dojít k problému, pokud CMake nedokáže najít C++ kompilátor (k čemuž by však nemělo dojít při standardní instalaci Visual Studia nebo při použití g++ kompilátoru jakožto balíčku na Ubuntu). Cestu ke kompilátoru je možné specifikovat nastavením proměnných CMAKE_C_COMPILER a CMAKE_CXX_COMPILER, více o nastavení proměnných níže.

Výstupem pro Visual Studio je soubor paracell.sln, který je možné přímo otevřít jako řešení. Pro Makefile stačí zahájit sestavení příkazem *make*. V této výchozí konfiguraci je sestaven pouze hlavní spustitelný soubor *paracell*, pro další části projektu je třeba při konfiguraci specifikovat vybrané proměnné. Obecně je možné proměnné definovat pomocí přepínače *-D*, následovaného výrazem *název proměnné = hodnota* (např. `cmake . -D BUILD_GUI=TRUE`).

Od konfiguračních proměnných se předpokládá nastavení hodnoty na *TRUE*. Nastavené proměnné jsou ukládány do souboru, tedy pro zakázání je nutné explicitně změnit proměnnou na *FALSE* (`cmake . -D BUILD_GUI=FALSE`).

Následuje seznam všech volitelných proměnných pro projekt. Proměnné je možné různě kombinovat za sebou.

- BUILD_DEBUG zkompiluje všechny části projektu s přepínačem pro ladění a bez optimalizací. Pro Visual Studio je tato proměnná zbytečná, vždy je možné řešení sestavit jako *Debug* pro ladění nebo *Release* pro nasazení.

- CUDA_ON aktivuje podporu pro CUDA výpočty. Samotný framework spolu s kompilátor *nvcc* by měli být dohledatelné v systémových cestách, jinak konfigurace selže.

- OPENCL_ON pro podporu OpenCL výpočtů. Opět by framework měl být dostupný ze systémových cest.

- VULKAN_ON pro podporu Vulkan výpočtů. Pro samotný vývoj je třeba nainstalovat SDK [35], které je dostupné pro všechny hlavní operační systémy. Opět by měli být vývojové soubory, spolu s kompilátorem glslc (který je součástí SDK) dostupné ze systémových cest.

- VULKAN_DEBUG pro podporu ladění Vulkan výpočtů. Při spouštění jsou používány *validation layers*, pomocí kterých jsou ověřována volání knihovních funkcí (zdali jsou v souladu se specifikací) a rovněž zajišťují výpisy ze shaderů, které jinak nejsou podporované. Pro reálné nasazení však mohou představovat výrazné zpomalení a mimo ladění by měli zůstat vypnuty.

- RUNTIME_GPU_SUPPORT zapíná podporu pro načítání knihoven pro grafické výpočty za běhu, blíže popsána zde 5.2.3. Pro každou z povolených grafických knihoven je vytvořena vlastní knihovna, která je dynamicky načítána až za běhu programu, s možností zjistit, zdali je technologie podporována nebo není. To umožňuje větší flexibilitu při spouštění binárních souborů na zařízení, které nepodporuje všechny technologie specifikované při kompilaci.

- BUILD_GUI sestaví grafické rozhraní jako samostatný spustitelný soubor *paracell-gui*. Závislost je na frameworku Qt, který však může být obtížněji instalovatelný pro systém Windows – instalaci je věnována následující samostatná sekce.

- USE_VCPKG, spolu s ALLOW_DOWNLOAD, slouží pro specifickou konfiguraci Qt frameworku, popsáno více v následující sekci.

- BUILD_TEST pro podporu testování. Vytvořeny jsou spustitelné soubory *paracell-tests* pro spuštění samotných testů a *paracell-tests-gen* pro vygenerování log souborů pro komplexnější testy (samotný běh pro všechny výpočty je v řádu několika minut, proto je jejich generování odděleno od samotného průběhu testů). Pokud bylo rovněž specifikováno *BUILD_GUI*, je vytvořen *paracell-tests-gui* pro otestování grafického rozhraní. Testy mohou být rovněž rozšířeny podle zapnutých technologií pro GPU výpočty.

Integrace s Qt

Na systému Ubuntu je framework Qt dostupný jako systémový balíček, spolu s dalšími balíčky pro vývoj. Po nainstalování by měl CMake najít potřebné soubory a sestavení grafického rozhraní by mělo proběhnou bez problémů.

Obtížnější situace je na systému Windows. Open source verze je možné stáhnout z oficiálního zdroje [36]. Pro použití instalátoru je vyžadováno vytvoření uživatelského účtu. Při instalaci je při výběru komponent nutné zvolit verzi 5 (v době psaní 5.12).

Po nainstalování nejsou nijak upravovány systémové proměnné, je tedy třeba před CMake konfigurací definovat cestu k dané instalaci. Možností je nastavit proměnnou CMAKE_PREFIX_PATH na kořenovou složku dané implementace. Instalátor nicméně nainstaluje vícero binárních souborů pro různé kompilátory nebo bitové verze, a je třeba najít správnou složku. Příklad konfigurace v cmake může být následující:

```
cmake -D BUILD_GUI=TRUE
      -D CMAKE_PREFIX_PATH="E:/Qt/5.15.2/msvc2019_64"
```

E:/Qt je kořenová složka dané implementace, následuje složka verze frameworku a msvc2019_64 je kořenová složka konkrétních binárních souborů.

Jako alternativa je v CMake projektu přidána podpora pro *vcpkg* [37], což je správce balíčků pro jazyk C++. Pomocí něj je možné automaticky nainstalovat potřebné Qt moduly sestavením přímo ze zdrojových kódů. Tento proces je možné zcela automatizovat po zapnutí dvou proměnných *USE_VCPKG* a *ALLOW_DOWNLOAD*. Nevýhodou je časová náročnost samotného sestavení, v řádu několika hodin, nicméně se jedná o jednorázový a zcela automatický proces.

Oba postupy rovněž automaticky po sestavení samotného grafického prostředí instalují potřebné Qt soubory do výstupní složky. Pro samotné spuštění tedy nejsou požadované žádné závislosti.

Příloha C

Uživatelská příručka

Výpočetní aplikace

Hlavní aplikace (*paracell*, *paracell.exe*) nabízí pouze terminálové rozhraní. Po spuštění je automaticky proveden výpočet, bez možnosti nějaké interakce od uživatele.

Pro zahájení výpočtu je povinně nutné předat cestu k souboru s konfigurací a cestu k souboru se vstupními daty. Dobrovolně je možné zadat i soubor pro logovací výstup (v případě nezadání je log přesměrován na standardní výstup).

Program podporuje starší způsob zadávání argumentů, kdy je postupně zadán konfigurační soubor, vstupní soubor a případně log soubor (přesně v tomto pořadí). Například:

```
/paracell config.cfg input.dat log.txt
```

Tímto způsobem není možné specifikovat dávkové soubory.

Novější způsob umožnuje specifikovat argumenty v libovolném pořadí pomocí přepínačů:

--input, -i pro vstupní soubor

--config, -c pro konfigurační soubor

--batch, -b pro dávkový soubor

--log, -l pro log soubor

Původní příklad tak může vypadat takto:

```
./paracell -i input.dat -c config.dat -l log.txt
```

Program rovněž podporuje přepínač *--help* pro zobrazení pomocných informací.

Grafické rozhraní

Detailní popis návrhu a způsobu použití grafického rozhraní je v hlavním textu v sekci 4.4. Pro lepší ilustraci použití byly ještě vytvořeny snímky z reálné aplikace s popisky o použití, které se nacházejí na následujících stránkách.

Testy

Testy pro hlavní výpočetní část lze spustit pomocí souboru *paracell-tests*, testy pro grafickou část pomocí *paracell-tests-gui*. Automaticky jsou v obou případech spouštěny všechny testy, pro oba případy je však možné spouštěné testy filtrovat:

- Pro hlavní výpočet je možné použít přepínač *–gtest_filter=* s filtrem pro název testů. Znak * je možné použít jako doplnění libovolných znaků. Např. *paracell-tests –gtest_filter=Vulkan** spustí pouze Vulkan testy.
- Pro grafickou část je možné předat jako argument jednotlivé názvy testovacích metod, například

  ```
  ./paracell-tests-gui gridConfigTest treorConfigTest
  ```

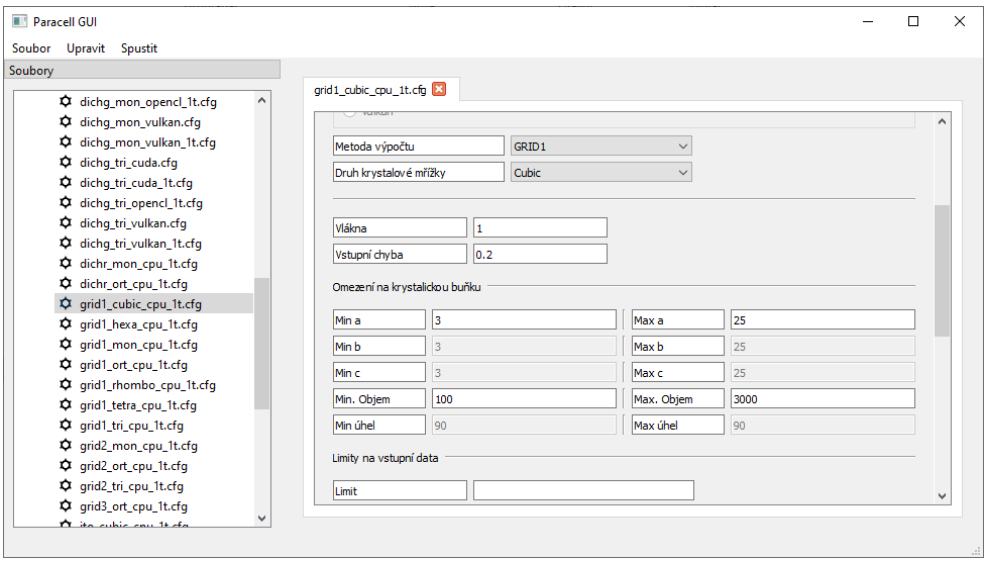
Obrázek C.1: Hlavní obrazovka. V horní části lišta s ovládacími záložky, vlevo seznam otevřených souborů a složek (prázdný), uprostřed seznam otevřených karet (prázdný).

Obrázek C.2: Záložka „Soubor“ s možností vytvoření nového souboru (podle typu), otevření existujících jednotlivých souborů nebo celých složek, uložení aktuálního souboru (nebo uložení všech otevřených modifikovaných souborů), otevření nastavení a ukončení aplikace.

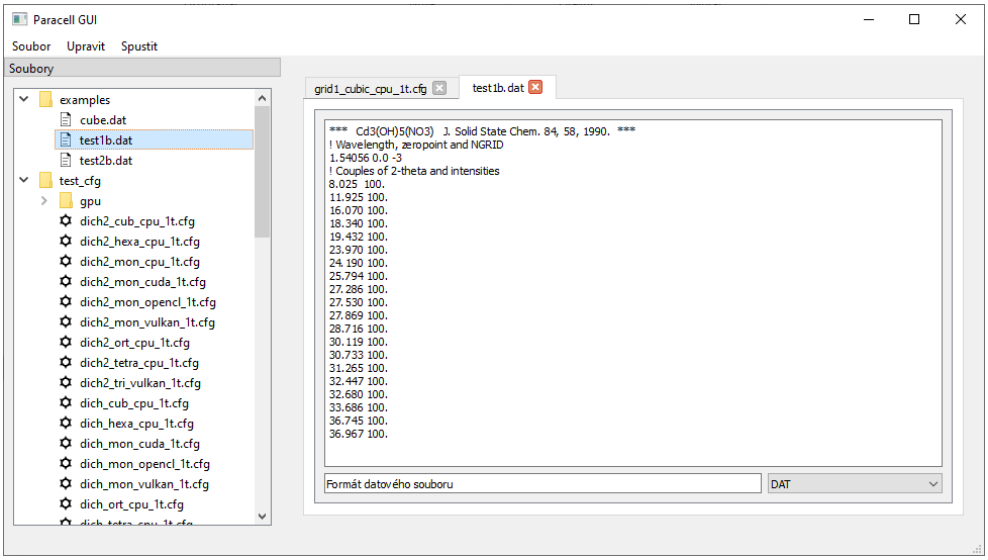
Obrázek C.3: Záložka „Upravit“ s možností kroku zpět a vpřed na aktuálně otevřeném souboru. Pro každý otevřený soubor jsou změny zaznamenávány zvlášť.

Obrázek C.4: Záložka „Spustit“ s možností otevření karty pro výpočet.

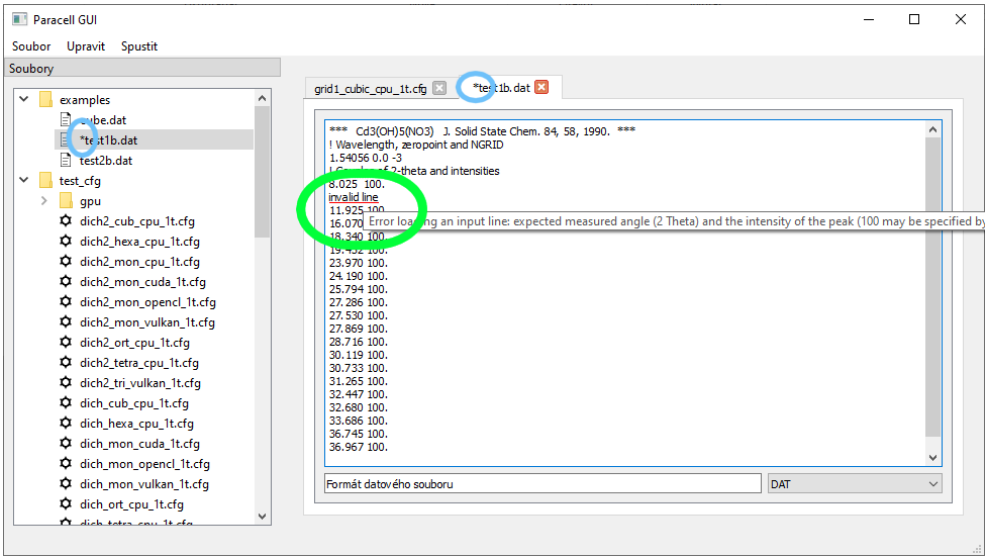
Obrázek C.5: Stav po otevření dvou složek. Otevřené složky a soubory jsou viditelné v seznamu souborů, řazené abecedně. Je možné je opětovně uzavřít po kliknutí pravým tlačítkem. Ze složek jsou automaticky otevírány pouze relevantní soubory, které jsou rozlišeny ikonami (jedno ozubené kolečko pro konfigurační soubor, více ozubených koleček v řádku pro dávkový soubor, list papíru pro datový soubor). Po dvojkliku na soubor se otevře karta pro jeho úpravu.



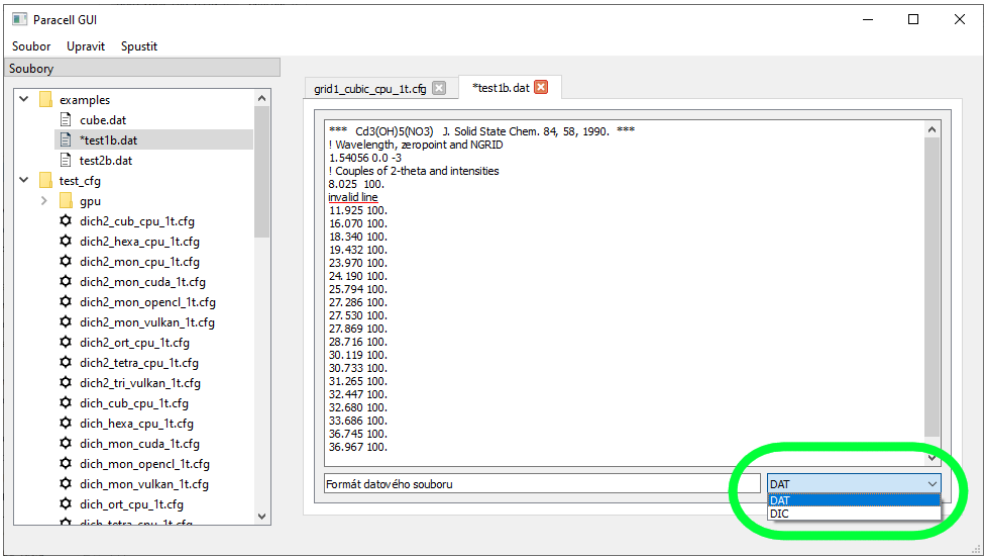
Obrázek C.6: Otevřená karta pro konfigurační soubor. Upravitelné jsou všechny hodnoty z textové podoby. Viditelné jsou needitovatelné prvky v omezení pro buňku, vyplývající ze zvolené soustavy (pro krychlovou mřížku má smysl nastavit pouze jednu stranu a objem).



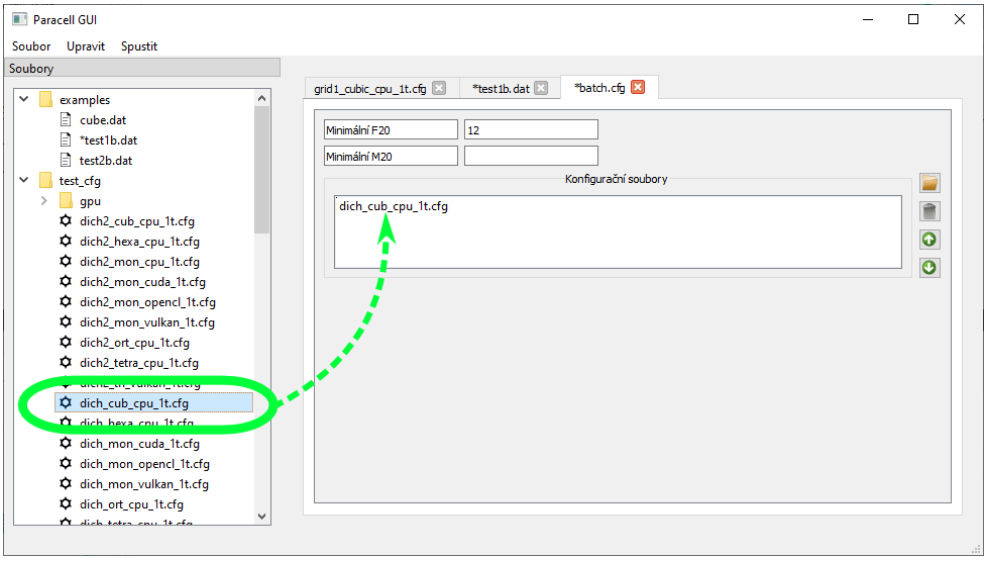
Obrázek C.7: Karta pro úpravu datového souboru - úprava je v textové podobě.



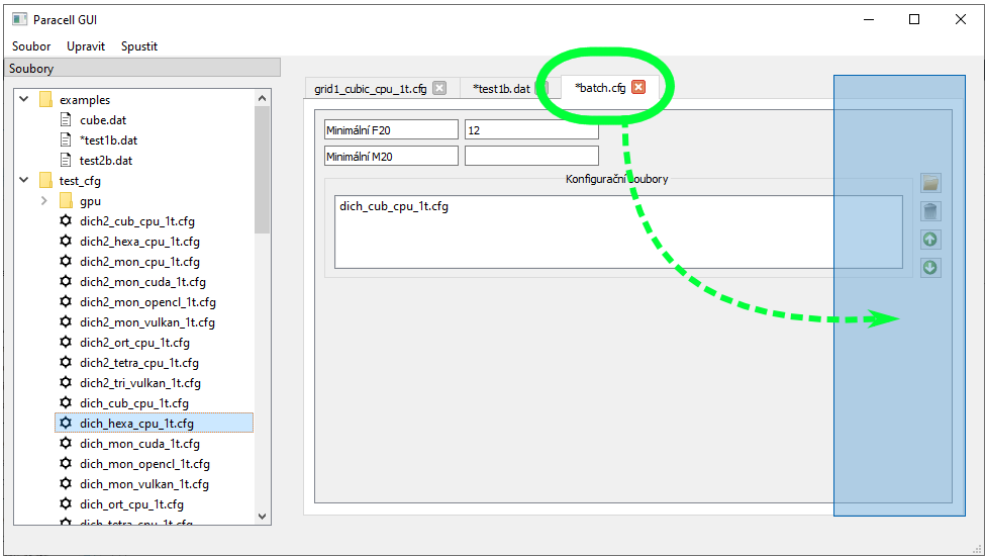
Obrázek C.8: V případě, že se při úpravě souboru stane soubor neplatným, je první problematická řádka červeně podtržena (soubory s chybou rovněž není možné uložit). Je možné zobrazit okénko s popisem problému po najetí kurzorem nad problematickou řádku. Dodatečně je možné vidět, že změna souboru je značena hvězdičkou na záložce karty a v seznamu souborů.



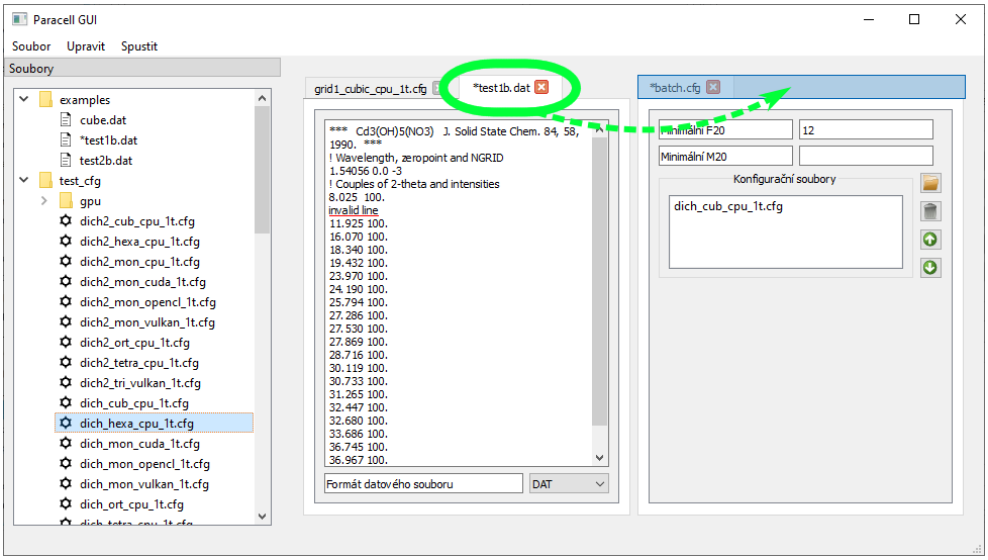
Obrázek C.9: Na kartě je rovněž možné zadat požadovaný formát souboru (ten je jinak automaticky odvozen při načtení). Po změně se automaticky změní detekce chyb dle zvoleného formátu.



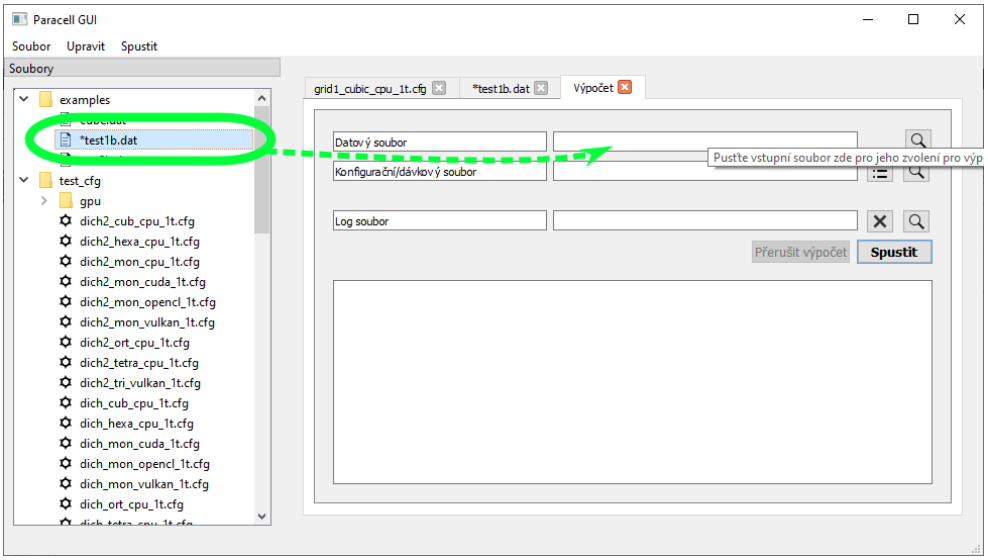
Obrázek C.10: Karta pro úpravu dávkového souboru. Kromě požadavků na hledané řešení je vidět seznam zvolených konfigurací, které budou postupně spouštěny. Po pravé straně seznamu jsou 4 tlačítka pro přidání konfigurace z filesystému pomocí dialogu, odebrání zvolené konfigurace a změna pořadí. Konfigurace je rovněž možné přidat přetažením souboru ze seznamu otevřených souborů.



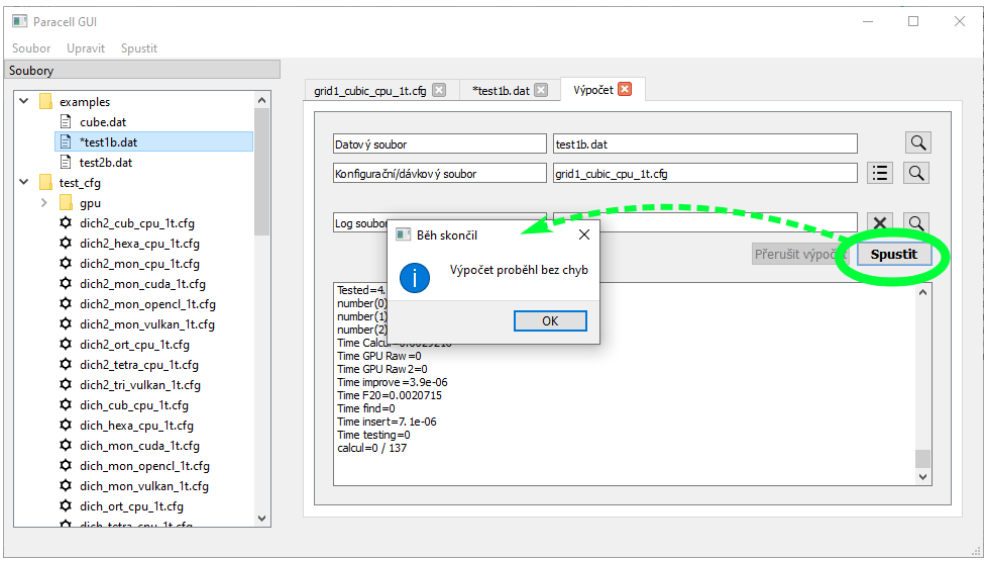
Obrázek C.11: Seznam s kartami je možné horizontálně rozdělit na libovolný počet částí přetažením záložky k levému nebo pravého okraji. Zde je viditelné rozdělení na půlku.



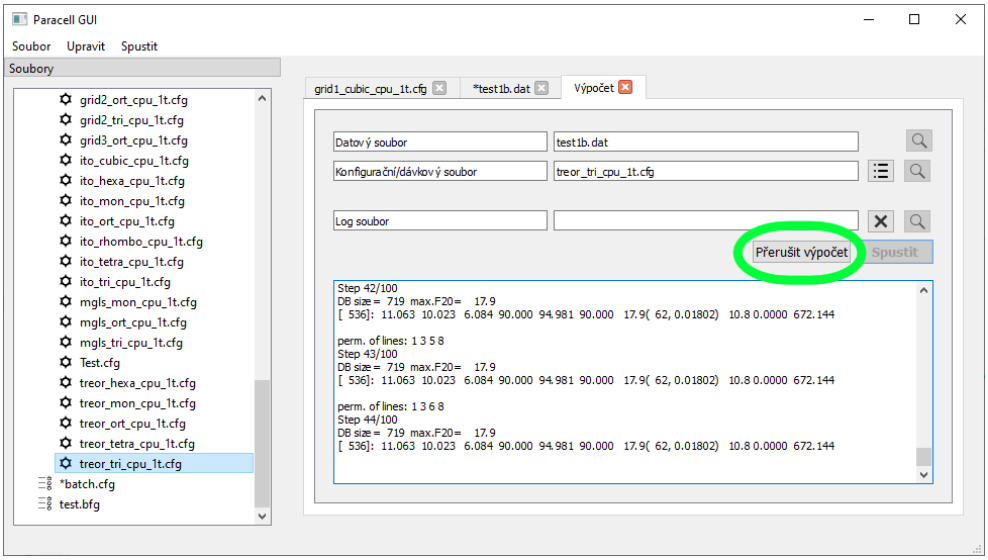
Obrázek C.12: Karty je možné přesouvat mezi rozděleními (nebo i v rámci jednoho seznamu) přetažením záložky.



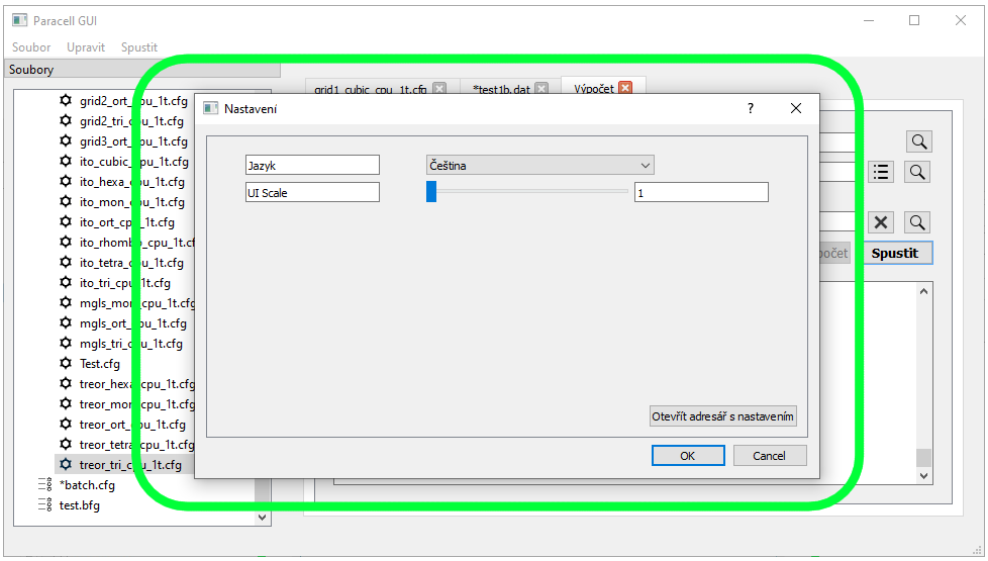
Obrázek C.13: Karta pro výpočet po otevření. V horní části jsou viditelné řádky pro zadání vstupních souborů (datový soubor, konfigurační soubor, log soubor). Výběr je možné provést tlačítky po pravé straně pomocí dialogu. V aktuálním návrhu je možné mít najednou spuštěný pouze jeden výpočet (a pouze jednu výpočetní kartu). Podobně jako u dávkového souboru je možné příslušné soubory zvolit přetažením ze seznamu souborů. Na položku je možné přetáhnout pouze správný typ souboru.



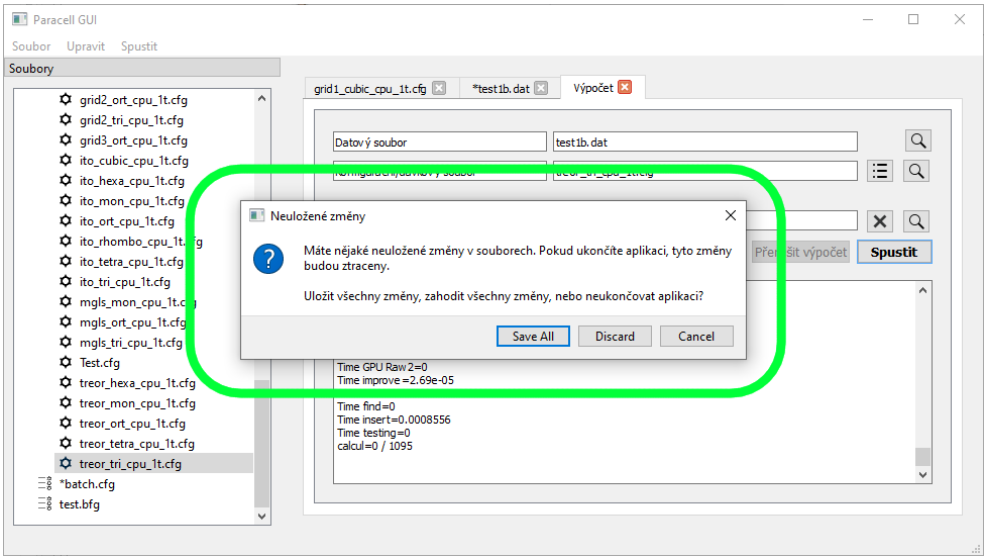
Obrázek C.14: Po zvolení souborů je možné pomocí tlačítka „Spustit“ spustit výpočet. Do textové pole se automaticky propisuje výstup výpočtu (log výstup se zapisuje pouze do zvoleného souboru). Dokončení výpočtu je oznámeno dialogem, viditelným uprostřed obrazovky.



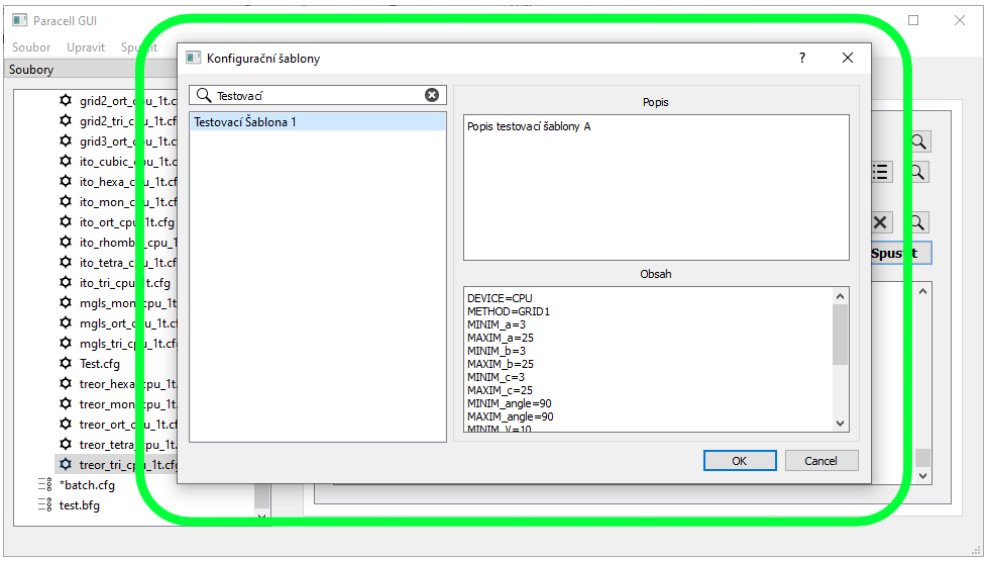
Obrázek C.15: Dlouho trvající výpočty je možné přerušit pomocí druhého tlačítka.



Obrázek C.16: Obrazovka s nastavením, aktuálně pouze s možností se zvolením jazyka a otevřením složky s uložením souboru s nastavením. V této složce se nachází i soubor ukládající aktuální stav aplikace při ukončení (otevřené složky, soubory, karty), který načítá při dalším spuštění.



Obrázek C.17: Při pokusu o ukončení aplikace bez uložení změn je uživatel upozorněn dialogem.



Obrázek C.18: Při vytváření nového konfiguračního souboru nebo jako možnou volbu pro výpočet může uživatel vybrat konfigurační šablonu. Po levé straně je seznam šablon, mezi kterými je možné vyhledávat, v pravém horním rohu je popis šablonu, v pravém dolním rohu obsah šablony.

Příloha D

Specifikace konfigurací

V následující tabulce jsou zpracovány všechny konfigurovatelné možnosti, jejich možné hodnoty a vysvětlující popis. Informace o některých hodnotách byly čerpány ze zdroje [6].

Klíč	Hodnota	Popis
Povinná nastavení		
DEVICE	CPU GPU AUTO*	Výběr typu zařízení pro výpočet. AUTO se pokusí zvolit GPU, v případě chyby se použije CPU.
METHOD	GRID1 GRID2 GRID3 TREOR DICHOTOMY DICHOTOMYR DICHOTOMY2 DICHOTOMYG DICHOTOMY3 MGLS ITO	Výběr metody a případné varianty pro výpočet.

SYSTEM	0, 1, ..., 6	Výběr krystalové soustavy. Odpovídající soustavy: 0 - Rhombohedrální 1 - Kubická 2 - Hexagonální 3 - Tetragonální 4 - Ortorombická 5 - Monoklinická 6 - Triklinická
MINIM_A MAXIM_A MINIM_B MAXIM_B MINIM_C MAXIM_C	*Reálné číslo větší než 0, např. 24.5*	Omezení na délky stran hledané primitivní buňky, v angstromech.
MINIM_V MAXIM_V	*Reálné číslo větší než 0*	Omezení objemu hledané primitivní buňky.
MINIM_ANGLE MAXIM_ANGLE	*Reálné číslo z intervalu (0; 180)*	Omezení na úhly (stupně) mezi stranami buňky.

Dobrovolná nastavení

DB_BEST_LIST	*Celé číslo,* > 0	Počet nejlepších řešení, které při výpočtu udržuje každé vlákno.
DB_DETAIL_LIST	*Celé číslo,* > 0	Počet nejlepších řešení, u kterých jsou po výpočtu zapsány detailní výsledky.

GAPI*	CUDA OPENCL VULKAN AUTO	Výběr technologie pro GPU výpočty. AUTO automaticky zvolí první funkční rozhraní v pořadí CUDA, OCL, Vk.
GPU_HINT*	*Textový řetězec*	Nápověda pro zvolení konkrétního grafického procesoru. Pokud je hodnota nastavena, program se pokusí najít daný text jako podřetězec buď v názvu výrobce nebo v názvu zařízení. Pokud se takové zařízení najde je okamžitě vybráno, jinak se zvolí výchozí zařízení.
IMPROVE_CELL	0, 1	Zapnutí automatického vylepšení všech řešení před přidáním do databáze.
INPUT_ERR	*Reálné číslo větší než 0*	Možná absolutní chyba v naměřených vstupních úhlech. Implicitně 0.
LIMIT	*Celé číslo,* > 0	Tyto 3 nastavení umožňují limitovat počet vstupních dat, které musí splnit řešení. LIMIT specifikuje přesný počet který splnit. NON_INDEXED počet, kolik splnit nemusí. RATIO určuje limit jako poměr ku počtu vstupů.
NON_INDEXED	*Celé číslo,* > 0	
RATIO	*Reálné číslo z intervalu (0; 1]*	

OPTIMIZE	*Celé číslo,* > 0	Počet nejlepších řešení, které jsou na konci výsledku různými způsoby upravovány pro nalezení ještě lepších řešení.
THREADS	*Celé číslo,* > 0	Počet použitých procesorových vláken. Implicitně se tolik vláken, kolik nabízí procesor.
SUPER_CELL	0, 1	Zapnutí možnosti po výpočtu najít superbuňku z nejlepšího řešení (stejná struktura, menší objem).
ZERO_SHIFT	*Reálné číslo*	Chyba v naměřených úhlech, určená konstantní hodnotou. Oproti INPUT_ERR, které specifikuje možnou odchylku, u každé pozorované difrakce jinou, se specifikovaná ZERO_SHIFT přičte ke každému úhlu.

GRID a MGLS nastavení

STEP1 STEP2 STEP3 STEP4 STEP5 STEP6 STEPS	*Celé číslo,* > 0	Každý krok specifikuje kolikrát má být dělen rozsah každého z (až 6) parametrů. STEP1 až 3 odpovídají délkám stran a, b, c, STEP4 až 6 uhlům α, β, γ. STEPS potom nastavuje hodnotu pro všechny parametry najednou.
DELTA_LATTICE DELTA_ANGLE	*Reálné číslo větší než 0*	Alternativa ke *STEPS*. Nastavuje konkrétní hodnotu, o kterou se při každém kroku posune velikost stran/úhlů.

TREOR nastavení		
MAX_H	*Celé číslo,* > 0	Omezení prohledávané množiny osnov rovin podle indexů HKL. Kromě jednotlivých indexů je možné omezit celkovou sumu.
MAX_K	*Celé číslo,* > 0	
MAX_L	*Celé číslo,* > 0	
MAX_SUM	*Celé číslo,* > 0	

ITO nastavení		
Q_EPS	*Reálné číslo větší než 0*	Hraniční hodnota, při které jsou 2 podobně ohodnocené zóny kategorizovány jako odlišné
Z_TOP	*Reálné číslo větší než 0*	Počet prohledávaných zón

DICHOTOMY nastavení		
AXIS_PRECISION ANGLE_PRECISION	*Celé číslo,* > 0	Nastavení velikosti intervalů (délky stran v Å, respektive velikost úhlů), kdy se zastaví dělení.

Příloha E

Testovací scénáře

Podle následujících scénářů probíhalo uživatelské testování grafického rozhraní. Aplikace na začátku testování byla ve výchozím stavu, bez otevřených souborů.

Základní práce se soubory

- Postupně otevřete soubory „davkova_konfigurace.cfg“, „vstup.dat“.
- Dále otevřete složku s názvem „soubory“.
- Vytvořte nový konfigurační soubor ve složce soubory pojmenovaný ”test.cfg”. Soubor může mít libovolnou šablonu.
- Zavřete soubory „davkova_konfigurace.cfg“ a „vstup.dat“.

Práce s pracovní plochou

- Ze složky otevřete pro úpravy soubory ”konfigurace1.cfg”, ”konfigurace2.cfg”, ”konfigurace3.cfg”.
- Kartu pro „konfigurace3.cfg“ uzavřete.

Úprava konfiguračních souborů

- V soubory „konfigurace1.cfg“ nastavte následující hodnoty:
  - Zařízení na GPU,
  - Grafické API na Vulkan,
  - Metodu na MGLS,
  - Krystalovou soustavu na Triklinickou,
  - Zbytek hodnot nastavte tak, aby byly ekvivalentní souboru „konfigurace2.cfg“
- Po provedení změny uložte.
- Navraťte soubor do původní podoby a změny uložte.

Pokročilejší práce s pracovní plochou

Cílem rozdělení je zjistit, zdali uživatel některé funkce nezačne používat z vlastní iniciativy.

- Zařiďte, aby byly najednou viditelné obsahy obou souborů „konfigurace1“ a „konfigurace2“.
- Otevřete soubor „davka1.cfg“ ze složky. Nastavte plochu tak, aby byl viditelný obsah souborů „konfigurace1.cfg“, „konfigurace2.cfg“ a nového ”davka1.cfg”a aby plocha zůstala pouze rozpůlená.
  - Karta souboru ”davka1.cfg”se otevře v levé polovině. Cílem je zařídit, aby uživatel přemístil jednu ze dvou záložek z levé poloviny do pravé.

Upravení dávkového souboru

- Nastavte požadavky na kritéria M_{20} na 5 a F_n na 10.
- Přiřaďte dávce soubor „konfigurace3.cfg“.
  - Snaha je, aby uživatel přiřadil přetažením ze seznamu.

Upravení datového souboru

- Otevřete pro úpravy soubor „vstup1.dat“.
- Změňte formát soubory na *DAT*.
  - V dolní části karty změní uživatel v *combo boxu* formát z *DIC* na *DAT*.
- Opravte soubor, aby odpovídal novému formátu, a změny uložte.
  - Uživatel by měl následovat červeně podtržené řádky a podle zmíněných chyb upravit soubor.

Spouštění výpočtu

- Spusťte výpočet.
  - Úkol je záměrně vágní, aby nebyl uživatel naveden k otevření nové karty a co nejobecněji se snažil zjistit, kde se dá výpočet spustit.
- Jako vstup zvolte soubor ”vstup.txt”. Vyberte libovolnou vhodnou konfiguraci.
  - Cílem je, aby se uživatel pokusil využít šablony a z nich vybral vhodnou konfiguraci. Případně zdali se pokusí soubor vybrat ze seznamu nebo z filesystému.
- Spusťte výpočet, a následně jej zastavte.

Úpravy šablon

- Přidejte soubor *konfigurace1.cfg* do šablon.
- Spusťte stejný výpočet se šablonou *Testovací šablona 1*.
- Odeberte ze seznamu šablonu *Testovací šablona A*.

Příloha F

Obsah přiloženého média

- `license.txt` ..... licence použitého obsahu třetích stran
- `src` ..... adresář se zdrojovým kódem aplikace
- `bin` ..... adresář se spustitelnou verzí aplikace pro systém Windows
- `text`
  - `src` ..... zdrojový kód práce ve formátu LaTeX
  - `thesis.pdf` ..... text práce ve formátu PDF